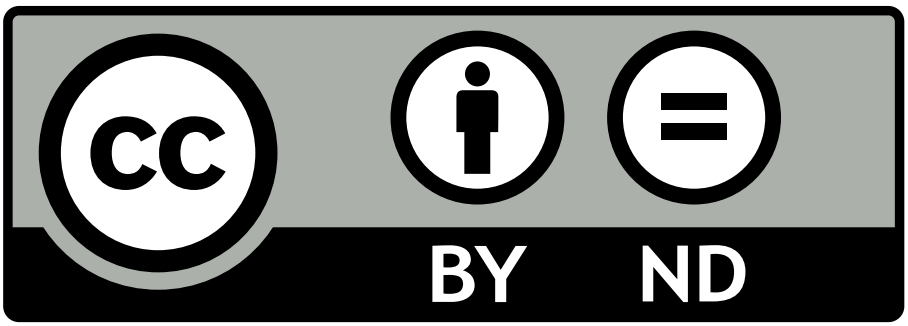
Modern Distributed Systems (Part I/III)
Version 0.22

Kishore Kothapalli and Smruti R. Sarangi

December 8, 2025



Contents

Preface

The subject of distributed systems has grown to span the entire gamut of computing over the past few decades. A firm understanding of distributed systems is now essential for building scalable, efficient, planet-scale, 24x7, user-facing systems that permeate our daily lives. Examples of such systems include the Unified Payments Interface (UPI) available in India and select other countries, social networking sites such as Facebook®, e-commerce websites such as BigBasket®, Amazon® and Flipkart®. Today, such big firms boast a market capitalization exceeding hundreds of billions of dollars and impact the daily lives of almost everyone on the planet.

Designers of such systems have to make important decisions based on the semantic guarantees they wish to provide to users while ensuring that such guarantees are possible within the realm of practical distributed systems. These guarantees come in the form of availability, consistency, read/write semantics, transaction processing outcomes, fault-tolerance and consensus. In practice, many trade-offs exist. It is seldom possible to satisfy all the requirements simultaneously.

Hence, building such systems and understanding the intricacies, challenges, and implications, requires one to have a complete understanding of the overall discipline of distributed systems. This includes its fundamentals, recent developments, theoretical and practical aspects. Many textbooks in this area are quite dated and were written two decades ago. They do not capture the recent developments in the field. Distributed systems, as of 2025, are far more ubiquitous as compared to where they were two decades ago. They run at a planet scale, and this has necessitated concomitant developments in all areas of theory and practice. The aim of this textbook is to capture many of these developments.

Generally, a course on distributed systems is taught as a first-level elective in almost all Computer Science departments. The ACM Curriculum Guidelines for Computer Science includes topics spanning distributed systems in multiple courses. Specifically, the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing recommends including various components of distributed computing and distributed systems in core undergraduate courses. A lot of the material covered in this textbook is mandatory reading for graduate students in the area and industry professionals who delve in such topics.

Against this background, this book has multiple aims. The first aim is to provide extensive coverage of the field of distributed systems including foundational aspects and latest developments. The second aim of the book is

to provide a comprehensive treatment of the recent developments in the field via suitable abstractions such that material has a long shelf life yet enable the reader to get an accurate understanding of contemporary systems. The third aim is to connect the theory of distributed systems with its practice. This is done via examples and case studies that illustrate how classical principles in large real-world practical deployments. The chapter on programming principles and frameworks for distributed systems was added to highlight such connections between theory and practice. The overarching aim of the book is to serve as a standard text for topics concerning advanced distributed systems. To supplement the overall learning process, we provide a list of probing and insightful questions at the end of each chapter. Additional material is provided in the form of slides, virtual laboratory experiments for enhanced understanding, and project ideas to allow students to explore the field further.

Organization of this Book

The book is divided into three parts. The first part covers foundational concepts. Any serious student of distributed systems should read this part in great detail. It starts with explaining the basic notion of *time* in distributed systems. It moves on to discuss the methods of distributed data storage and lookup. The subsequent chapters focus on distributed algorithms. In the area of distributed systems, *mutual exclusion* is a very important problem. This gives any single process exclusive control over a resource, albeit for a short time. This chapter is like a bridge between the traditional centralized paradigm and distributed computing. Hence, we have devoted a chapter exclusively to the study of distributed algorithms. In the next chapter we focus on the full gamut of foundational distributed algorithms that include both synchronous algorithms (settings where a global clock is shared) and asynchronous algorithms. It will be apparent after a study of these chapters that the problem of *consensus* or *agreement* is fundamental to distributed systems. Many algorithms can be reduced to the problem of agreeing on a common value. Hence, we shall discuss consensus problems in the next chapter and present a fundamental result that says that in an asynchronous distributed system with a faulty process, guaranteeing consensus all the time is not possible. Finally, we shall discuss consistency-related problems in distributed settings. These problems arise when multiple replicas of the same datum are maintained and they need to be synchronized.

The first part lays a strong foundation for the rest of the book. The second part is about distributed services. It has four chapters. The first two

chapters are on distributed file systems and distributed databases, respectively. They extend the basic distributed data storage concepts introduced in Part I. We shall discuss the design principles of distributed file systems and data stores, and then introduce the principles underlying some of the popular designs. In a distributed system, it is typically not possible to provide very strong consistency guarantees. The classical PACELC theorem indicates that there is a trade-off between the degree of data consistency and latency. If we wish to create a fast high-throughput system, then some consistency guarantees need to be sacrificed. Practical systems need to find the appropriate point in this trade-off spectrum that suits them. Then we shall cover blockchains, which are becoming extremely important these days. They are the core technology for highly secure systems including cryptocurrencies. The last chapter in this part deals with distributed machine learning. It is not possible for a single machine to perform complex machine learning tasks with today's large models. The model and data sizes overwhelm the constraints of the machine. Hence, there is a need to perform both training and inference in a distributed setting, such that multiple machines can cooperatively perform an ML-based task.

The third part is concerned with managing distributed systems. Just building a distributed system is not sufficient. It is important to manage and operate it such that it is highly available, scales with demand and adjusts itself with evolving customer preferences. This part has five chapters. The first two chapters cover different aspects of middleware. Middleware is defined as a software layer between the underlying operating system and applications. It provides services to distributed applications running on top of it. The first chapter will discuss generic aspects of middleware and the next chapter will focus on edge and fog computing use cases. The next chapter shall focus on programming distributed systems. This links the applications, the middleware and the physical aspects of the distributed system. A good programming language should enable applications to extract as much of performance as possible from the underlying platform, yet be platform agnostic as far as possible. Next, we shall look at security and performance evaluation. It is important to assess the suitability of a distributed system for practical use. It needs to be secure and highly performant for modern scalable workloads.

Use of this Book

Before starting to read this book, the reader should be familiar with the basics of discrete mathematics, data structures, algorithms, operating systems

and programming. We would advise readers with a non-CS background to pick up these topics before starting to read this book. Ideally, this course should be taught to senior undergraduate students or entry-level postgraduate students.

This book can support two kinds of courses in distributed systems. The first is a typical first-level elective course on Distributed Systems. Most Computer Science departments across the world teach such a course at least once every year. The topics in this first-level course rely mostly on courses covered in Part I of this book. Such a course should put more emphasis on fundamental concepts in distributed systems including the notion of time, consistency, consensus, mutual exclusion, distributed algorithms and their limitations.

The second is a course or module on advanced distributed systems. This course can focus on more contemporary aspects of distributed systems such as blockchains, distributed machine learning, fog computing and security. Such an implementation-oriented course can provide the reader a comprehensive view of the practical aspects of distributed systems as they are used and operated as of today.

Resources

The website `http://www.thedistsysbook.com` has a link to all the slides, YouTube lectures and a PDF version of this book. The book is released under the CC-BY-ND 4.0 license. The book and its website are updated continuously. Hence, the reader is advised to check updates frequently.

Acknowledgments

The first part of the book took two years to write. This work could not have come to this level without the help and support of a lot of individuals and institutions.

First, we would like to thank the respective host institutions namely IIIT Hyderabad and IIT Delhi, respectively. They graciously extended their facilities for writing this book. During the course of these two years, a lot of students and faculty colleagues contributed their views about teaching concepts in the broad area of distributed systems. Early versions of the manuscript were shared with them. Their feedback proved to be invaluable.

We were very fortunate to be supported by top companies in the field. We would like to thank Google® and Microsoft® for their generous support. We would also like to profusely thank Dr. Rama Govindaraju for making a personal donation.

Such an intensive and prolonged effort was only possible because of the support of our families. Without their patience and sacrifice, this book would never have seen the light of the day.

List of Trademarks Used

The following are trademarks or registered trademarks of their respective owners:

Amazon is a trademark or registered trademark of Amazon.com, Inc. or its affiliates.

Anthos is a trademark of Google LLC.

AWS is a registered trademark of Amazon Technologies, Inc.,

BigBasket The trademark for the brand BigBasket is owned by Innovative Retail Concepts Private Limited

ChatGPT is a registered trademark of OpenAI.

CloudFlare is a registered trademark of CloudFlare, Inc.

CORBA is a registered trademark of the Object Management Group (OMG).

Facebook is a registered trademark of Meta Platforms, Inc.

Flink is a registered trademark of the Apache Software Foundation (ASF).

Flipkart is a registered trademark of Flipkart Internet Private Limited.

Google is a registered trademark of Google LLC.

Google Cloud is a registered trademark of Google LLC.

gRPC is a registered trademark of the Linux Foundation.

Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.

Kafka is a registered trademark of the Apache Software Foundation (ASF).

LinkedIn is a trademark of LinkedIn Corporation.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Meta is a registered trademark of Meta Platforms, Inc.

Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Netfix is a registered trademark of Netflix, Inc.

NVIDIA is a registered trademark of NVIDIA Corporation in the U.S. and other countries.

NxtGen is a registered trademark of NXTGEN Data Center and Cloud Technologies Private Limited.

RabbitMQ is a registered trademark owned by Broadcom, Inc..

Spark is a registered trademark of the Apache Software Foundation (ASF).

Uber is a registered trademark of Uber Technologies, Inc.

Unix is a registered trademark of The Open Group.

UPI UPI (Unified Payments Interface) is a registered trademark of the National Payments Corporation of India (NPCI).

Windows See Microsoft.

All other trademarks are the property of their respective owners.

Chapter 1

Introduction

> *“If you want to go fast, go alone. If you want to go far, go together.”*

African proverb

As the proverb suggests, there are limits to what one can do with a single computing machine. Regardless of the size of the machine, the number of cores it has, the amount of memory and the frequency of each core, it fails to scale beyond a certain point. Given today's large problems and enormous amounts of data, we need to create large systems comprising thousands of machines. There is a need to partition large problems into small sub-problems and run each sub-problem on a constituent machine. Additionally, there is a need for synchronization and communication across the machines to ensure that high-level objectives are met. Such a system that comprises hundreds or thousands of machines that are connected to each other using a network, yet appears to be “one” is known as a *distributed system*. Its role is to appear to be a single, cohesive unit to software such that it can be effectively programmed to solve large problems. A point to note here is that the fact that there are a multitude of machines in the distributed system will not be hidden from software. Instead, a convenient interface will be provided to leverage their capabilities in a cooperative fashion to partition large problems, execute pieces of them concurrently, and possibly combine the partial results to yield the final solution. There is a unity of purpose among such disparate machines that often communicate via conventional networks.

The influence of distributed computing can be seen in many traditional and emerging areas of Computer Science such as programming languages,

algorithms, operating systems, file systems, storage, group communication and Internet-of-Things (IoT systems). In fact, it is not incorrect to say that distributed computing has pervaded almost all aspects of enterprise and web-scale computing and storage.

Specifically, in the last decade, there have been several developments that have had a huge impact on society. We have seen search engines such as Google®, LLMs such as ChatGPT®, large e-commerce giants such as Amazon®, cab aggregators such as Uber® OTT content providers such as Netflix® and real-time online payment support systems such as UPI®.

All of them rely on large distributed systems and deftly combine theoretical foundations with practical system-level concerns. Understanding the design of such distributed systems is thus a necessity in today's day and age. Consequently, it is important to appreciate the theoretical underpinnings of this fascinating area, understand all the services and features that modern distributed systems provide, and finally get a grasp of the applications that power today's businesses and web-scale systems.

The current book is a step in that direction and covers foundational material from distributed systems including the notion of logical time, mutual exclusion, transaction processing protocols, and blockchains. It is always important to read and understand basic fundamental concepts. They help us build large petascale distributed file systems, No-SQL database systems that store the planet's data, newer models of consensus that is used by cryptocurrencies, so on and so forth. In this book, we shall look at futuristic applications (as of 2025). Other than cryptocurrencies, we shall look at fog computing, distributed machine learning, federated learning and ultra-secure distributed systems.

This book is divided into three parts: **Concepts**, **Services** and **Management**. Specifically, the last part deals with creating and managing futuristic applications that run in a distributed environment.

1.1 The Evolution of Distributed Computing and Distributed Systems

In this section, we shall briefly trace the evolution of distributed computing and distributed systems. Doing so will help us understand how the scale, richness, and complexity of such systems has been steadily increasing. We will also comment on quintessential examples of popular distributed systems (across generations).

1.1.1 First-Generation Distributed Systems

The earliest distributed systems are referred to as first-generation distributed systems. The creation of ARPANET in the the seventies started this area. It was suddenly possible to connect multiple machines across the United States. Ultimately, this project led to the creation of the modern internet. Much of the progress in creating distributed systems can be attributed to Xerox PARC in the mid eighties that started creating very early versions of distributed systems using RPCs (remote procedure calls). Such primitives led to the development of distributed file systems such as AFS and NFS that suddenly allowed a single file server to store the files for an entire administrative unit. Individual client machines could access files from this file server. These primitives were a natural precursor to network operating systems like Amoeba [Tanenbaum et al., 1990] released by Tanenbaum in the late eighties. The main architecture was based on remote procedure calls, where the code resided on machine A and it was invoked by machine B. The arguments and return values were sent over the network. This led to concepts such as network resources where there are a fixed set of services that a network provides – these services can be used by processes. These processes themselves can migrate across machines. The Berkely Sprite [Ousterhout et al., 1988] operating system exemplifies this paradigm.

1.1.2 Second-Generation Distributed Systems

With the arrival of networking, especially Ethernet technologies, the second generation of distributed systems began. Some of the early services were CORBA, DCOM and Java RMI. All of these were technologies to implement distributed objects. A distributed object resided on a set of servers. Client machines get a handle to each distributed object. They can invoke methods on the distributed objects. The method calls are routed to the servers along with the arguments, the method is invoked remotely and the return value is sent back. These evolved into generic systems that could be used by all kinds of platforms and programming languages. File systems also started to get more sophisticated. The Google File System [Ghemawat et al., 2003] and HDFS [Borthakur et al., 2008] were two prominent products in this category. HDFS, especially, was suitable for very large datasets. This era also saw the map-reduce paradigm come to the fore. Large computations could be mapped to a cluster of nodes and their results could be aggregated and processed at a single node.

The area also popularized the area of peer-to-peer (P2P) systems. A

large amount of data is stored on a network of peer machines. The key operation is to locate the machine that stores a given file. Complex mathematical functions are used to establish such a mapping. The data structure that stores files in such a distributed fashion is known as a distributed hash table (DHT). Several generations of P2P systems were proposed. Napster started out as a centralized architecture. Gnutella, Kazaa and BitTorrent had fully distributed architectures.

Early-stage grid computing architectures with a large distributed set of machines started getting proposed. It was possible to add even privately held machines to the grid. Globus [Foster and Kesselman, 1998] and Condor [Thain et al., 2005] were two popular architectures in this space. They were sophisticated grid schedulers that could manage a complex heterogeneous set of machines and match complex jobs with machines.

1.1.3 Third-Generation Distributed Systems

Large-scale cloud computing systems were the hallmark of this era. Many companies created large planet-scale public clouds, which could be used by individuals, businesses, academic researchers and the public at large. Some of the prominent offerings in this space were Amazon AWS®, Google Cloud® and Amazon AWS®. They are widely used and are familiar to most computer science students. In the early days of cloud computing, such cloud platforms gave virtual machines (VMs) to users. These VMs migrated across machines if there was a need. Nowadays, VMs are being replaced by lightweight solutions namely containers. Docker and Kubernetes are important solutions in this space. A container can be thought of as a very thin abstraction layer on top of a conventional operating system. It primarily virtualizes the file system and the network.

The advent of containers led to the *serverless* paradigm. Here we have ephemeral (short-lived) containers that are created only for the purpose of executing a specific function. In a distributed system, different nodes can specialize in providing different services. For example, one machine may provide database services, one can verify the input and one can perform ML-model inferencing. In the serverless paradigm, a request can be sent to a machine, it can quickly start a container, execute the request and destroy the container. Such services can be quickly and seamlessly provided in a large distributed system. AWS Lambda and Google Cloud Functions are popular serverless platforms that provide such services. They provide ultra-fast compute services and also access to underlying storage engines.

Along with large-scale distributed compute and storage infrastructure,

high-throughput stream processing is also a key feature of third-generation distributed systems. This era saw the advent of Apache Kafka®, Flink® and Spark™. These are ultra-high-throughput stream processing engines, which are used in almost all large corporations today.

Blockchains and crypto-currencies, IoT systems and large secure content providers such as CloudFlare® (for disseminating content) are the key innovations in the third generation. The growth potential of such solutions is expected to be quite high in the fourth generation (latest as of 2025). The traditional aims of any distributed system was high-throughput and scalability with an acceptable failure rate. However, this generation saw new goals getting added to this list. The aforementioned examples embody security guarantees, anonymity and the notion of zero-trust architectures.

1.1.4 Fourth-Generation Distributed Systems

We are currently in the fourth generation (as of 2025). Note that many of the performance and scalability concerns have been taken care of by third-generation systems.

The key aspects of latest fourth-generation systems are security, interoperability and ultra-large planetscale data stores. The focus is on hybrid clouds, where it is possible to create a large virtual cloud computing system that encapsulates multiple cloud providers, IoT systems and control systems. Such heterogeneous are extremely flexible and elastic. Some examples of products in this space are Crossplane, Karmada and Google Anthos®.

Any heterogeneous system that has inter-operable components requires strict security guarantees, well-defined communication protocols, trust-enforcing mechanisms and methods for auditing all actions. We will see a lot of research in this space.

1.2 Impact of Distributed Systems and Future Challenges

1.2.1 Algorithms and Data Structures

We are used to sequential algorithms, primarily because it is intuitive to think sequentially. Hence, data structures and algorithms are taught as a first course to undergraduate students in Computer Science. However, with the advent of distributed systems, it is important to design fresh algorithms and data structures that work in the distributed setting. There is a need to design fresh algorithms.

Figure 1.1: Photographs of a data center (Infiniband NVIDIA® H200 cluster). [courtesy: NxtGen® Data Center and Cloud Technologies Private Limited (`https://www.nxtgen.com`)]

The inputs are also quite large for such algorithms. They are often distributed across multiple machines. Shared memory models do not scale for such large machines. Hence, modern distributed systems mostly rely on message passing. Large distributed systems are typically asynchronous

systems – they do not rely on a global clock. Let us elaborate on the computation models for distributed systems that have evolved based on the features provided by the underlying platform (see Section 1.3).

Small distributed systems can afford to be synchronous. This means that they share a global clock and support the notion of *rounds*. All the participating processes know when a round ends and the next round begins. This makes writing algorithms quite easy. There are several paradigms in the space of synchronous algorithms. There can be bounds on the number of bytes exchanged per message or per communication round. Furthermore, there can be restrictions on the set of nodes that a given node is allowed to send a message to.

Real-world practical distributed systems typically find it hard to provide such guarantees. Hence, they are typically asynchronous where no such assumption of having a common clock is made. This makes writing algorithms for them quite difficult. Partially synchronous environments are like a hybrid. They are asynchronous till the Global Stabilization Time (GST), and then they become synchronous.

Chapters 3,4 and 5 provide an overview of popular distributed data structures and distributed algorithms. We shall observe that the main problem with such algorithms is that they are hard to design and scale. Moreover, in any large distributed system, faults inevitably happen. Hence, any algorithm that is designed to run on such systems needs to be immune to faults. Regardless of the incidence of faults, distributed computations need to converge to an acceptable state. We shall specifically discuss self-stabilizing algorithms that are generic methods to make distributed algorithms more robust.

1.2.2 Programming, Software Engineering, Verification and Debugging

After obtaining an appreciation of distributed algorithms, it is important to learn how to program such systems. We will study this in Chapter 14. Distributed programming frameworks such as MPI, RPC, and high-level programming language abstractions such as Hadoop and map-reduce have been proposed over the years. They have substantially simplified the process of coding distributed algorithms.

On the other hand, verification and debugging distributed programs is still quite difficult mainly because bugs in this space are quite complex in character and are seldom reproducible. Specifically, replicating the exact event sequence to recreate a fault is quite difficult, especially if it is not

logged and replayed. Lack of synchrony and faults make this even more difficult. To handle state space explosion, modern tools use a wide variety of techniques including AI to ensure that the entire process is tractable.

1.2.3 Systems Stack

Distributed systems also need their dedicated software stack. Conventional systems use an operating system like Windows or Linux. They may also run on virtual machines. Over the underlying OS layer, there are programming language runtimes that provide a virtual environment to running programs. The Java and .Net application-level virtual machines are important examples in this category.

Similarly, distributed systems need their software stack. In this case, it is assumed that any distributed program runs on a fundamentally heterogeneous system. The machines, operating systems and platforms could be different. The entire distributed system should still run seamlessly. This means that there is a need to create a virtual distributed operating system that allows processes to coordinate with each other, send messages, collect safe snapshots and manage all low-level details. Such a software layer is known as *middleware*.

There are several common paradigms in the middleware space. They could enable message passing using remote procedure communication such as gRPC® and Java® RMI. Some other types of middleware create a generic message exchange framework such as RabbitMQ® and Kafka®. These messaging frameworks are typically used to access distributed objects. There are dedicated protocols to send parameters to invoke methods defined by distributed objects and pass return values. The CORBA® and DCOM stacks used to be popular in this space. Similarly, there can be middleware for database access, web/application servers and distributed coordination. We shall discuss such middleware-based solutions in Chapters 12 and 13. In Chapter 13, we shall specifically focus on IoT and fog computing systems, where small devices participate in the distributed computation. We shall see that all distributed system concepts extend to networks of such ultra-small devices as well, albeit with their unique modifications.

1.3 Types of Distributed Systems

A distributed system comprises N independent processes – a process runs on a *node*, which can be a real machine or a virtual machine. The processes in general act independently, yet try to achieve a common goal. This is where

we would like to define three models of distributed computation [Dwork et al., 1988]: synchronous, partially synchronous and asynchronous. In a *synchronous* system, the processes either share a clock or are loosely clock-synchronized with each other. There is a known upper bound for the time it takes a message to get delivered (Δ) and the ratio between the clock speeds of two processors running the processes (η). Typically, we can divide the computation into a sequence of *rounds*. In each round, all the processes perform a set of actions and all of them agree on the fact that the round has finished. Once all the actions in the previous round finish, a new round begins. Note that at all points of time, processes are aware of the current round and the actions that they need to perform in this round.

Definition 1.3.1 A Distributed System

A distributed system comprises N independent processes that communicate with each other using a shared media such as shared memory or a message passing channel. Most algorithms and applications running on distributed systems make the entire system appear as a single process that can concurrently handle multiple user requests. Such systems can typically scale automatically, handle faults, and hide details of the underlying hardware and software platforms.

In a partially synchronous model, Δ and η exist, but they are not necessarily known a priori. In other flavors of this computation model, these bounds are known, but they start holding from an unknown time T. In a third variant, these bounds are loosely respected – these bounds are valid under certain conditions, but frequently may not hold.

Definition 1.3.2 Partially Synchronous System

Let the upper bound for the time it takes a message to get delivered be Δ, and the maximum ratio between the speeds of two processors running the processes be η. In a synchronous system, Δ and η are defined and known. In a partially synchronous system, they are typically defined but not known. Furthermore, these bounds begin to hold after the Global Stabilization Time (GST). The GST is normally not known; however, proofs in this model just need to assume that it *exists*. This is sufficient to prove many liveness properties.

The more general setting for a distributed system is an *asynchronous* setting, where the processes do not share a physical clock and the algorithm cannot be broken into rounds – Δ and η are not defined. This is a

purely event-driven system, where a process receives a set of messages and in response changes its state and sends some new messages. In a practical scenario, this setting is more useful because it does not rely on any form of clock synchronization.

As we shall see in Section 6.1, most distributed systems find it very hard to deal with faults. Furthermore, assuming that a large system will not have faults is also not a practical assumption. We need to create protocols that continue to work even in the presence of faults. Let us describe the nature of faults next.

1.4 Fault Models in Distributed Systems

The main aim of any distributed system is to make the entire system appear as one entity that works towards a common objective. It can be thought of as a large, single machine that provides a bunch of services. In general, distributed systems are complicated and ensuring a good quality of service is hard. However, in most practical settings, we need to deal with faults of various types. The moment such faults in a system arise, the system can behave in very unpredictable ways. As a result, making a distributed system fault-tolerant is challenging, yet is required.

Let us distinguish between three terms that are often very confusing and in practice are used interchangeably: fault, error and failure.

Fault A *fault* is a deviation from ideal behavior in a system. Assume that there is a bug in the code. This would be a fault.

Error Even if there is a bug in the code, we may not always be executing that piece of incorrectly written logic. However, when we execute the incorrect code, the internal state gets corrupted. This situation represents an *error*. For instance, a node going down is an error. However, the entire system may not go down, or the output may still be correct. An error is a manifestation of a fault but does not make the system unusable. It is an incorrect internal state.

Failure A *failure* is defined as an event when the error is visible at the output and the system (to a large extent) is unusable. Either the output is wrong or it is not generated in the first place because the entire system goes down.

Distributed systems should be fault tolerant, which means that if there are parts of the system that are not designed correctly or not operating cor-

rectly, then the protocol should be designed to mask the effects of the fault. The fault should not progress to a failure. There are many ways in which a given process can fail in a distributed system. Note that a process failure may not imply a system failure, in fact in the scope of the overall system, a process failure may be a fault because there may be adequate redundancy in the distributed system to nullify the effect of a process crashing. In this context, let us look at the different type of failures.

Fail-stop The given process simply stops operating. This is visible to the rest of the system.

Fail-silent In this case, the failure is not easily visible to other components of the system. The system becomes unresponsive like in the case of fail-stop failures. However, other parts of the system are not immediately aware of this fact. As a result, other processes may perceive the failed process to be just slow.

Byzantine failure This is the most pernicious type of failure. In this case, the process can behave maliciously. A process can either stop responding or provide erroneous outputs. It can also collude with other processes and try to confuse other honest processes to bring down the system. Such a process can appear as "failed" to one process and appear perfectly functioning to another one. In short, no assumption can be made about the behavior of such processes.

Distributed systems become interesting only when we consider asynchrony and faults. This mimics the real-world scenario where clocks are often not synchronized, there can be a large delay between two events, and processes can crash or fail (in numerous ways) at random. The challenge here is to keep a large system running in spite of all such exigencies. Consider a large system with 100k to a million machines. We see a node failure at least once or twice every hour. All the processes running on it crash as well. In fact, this is a common occurrence in all the systems that we use every day and take their reliability for granted. The reason we are not able to perceive such failures is because the distributed system continues to execute as if nothing happened and clearly no data is lost. We shall try to unravel the mysteries behind how this is achieved in this book. Of course, it is going to be a long journey across many fascinating chapters. Let us conclude this short chapter by discussing the organization of the book.

1.5 Organization of the Book

This book comprises three parts: Concepts, Services and Management.

1.5.1 Part I: Distributed Systems Concepts

The first part covers core distributed systems "concepts", which are essential to understanding most modern distributed systems and applications. Arguably, the most important concept is the notion of *time* (Chapter 2). Given that machines in a distributed system do not share a common clock and do not subscribe to a global notion of time, there is a need to provide a different definition of the notion of "time". Most systems typically rely on logical time as opposed to the physical time, where time is not tied to an absolute notion based on wall clocks. Instead, it is based on the ordering of events of interest. Almost all modern distributed systems that are concurrent in nature focus on events, the relationships between events (which event occurred after which one) and use this information to maintain and update the local time. Most messages are timestamped, and the destination uses the time embedded in the messages to update its local clock.

Along with time, the second most important concept is distributed data storage (Chapter 3). When we are looking at large web-scale data, one or a few nodes are not sufficient to store all the data, which can be petabytes or exabytes. We need to thus store data on a large number of nodes that are separated geographically. Furthermore, to ensure reliability, many replicas of data need to be made, and we need to ensure that the data store is accessible at all points of time. In other words, data is never lost and is always available. This requires many creative solutions for both structured and unstructured networks. We shall discuss such notions in this chapter especially DHTs (distributed hash tables), which have off late become very popular. We shall also look at data propagation algorithms in networks that do not have a well-defined structure. They mimic epidemic propagation in human or animal populations.

The next two chapters (Chapters 4 and 5) on mutual exclusion and distributed algorithms focus on classical algorithms in the distributed systems space. These algorithms are used in practical systems to implement a wide variety of functions. For example, mutual exclusion algorithms are required when there is a need for only one process to access a resource. They are a subset of the broad area of distributed algorithms that encompass a plethora of techniques to solve problems in a distributed fashion. Some notable examples are vertex coloring, facility location, tree traversal, computation of

the minimum spanning tree and synchronization. The reason we specifically focus on mutual exclusion algorithms in a separate chapter is because they are in a sense special. They solve important problems with respect to coordination in distributed systems, fault tolerance, efficiency, scalability and fairness. Moreover, their basic design mechanisms teach the learner basic distributed algorithm primitives. The same techniques can be applied in other settings to solve more complex problems (as we do in the next chapter).

We shall observe that computing something in a distributed fashion is almost always tantamount to solving the consensus problem in some or other way (Chapter 6). The consensus problem is quite simple per se. In a system with n processes, each process proposes a value and one among the proposed values is chosen. Consensus has a notion of universality [Jayanti and Toueg, 1992], which means that it can be used to solve any problem in distributed systems, subject to certain conditions and limitations. Sadly, even if there is one faulty process, it is not possible to achieve consensus all the time. This is the classical FLP result. Notwithstanding this impossibility result, there are many practical algorithms that have been designed for the consensus problem. They are known to work well. We shall study many such algorithms in this chapter. They are designed to solve any form of distributed agreement. Some work well in general settings, some work well when the fault rate is low and some work reasonably well in the presence of Byzantine faults.

We will end the first part in Chapter 7 with a thorough discussion on the trade-offs in distributed systems. Distributed systems provide different kinds of *consistency* guarantees. A distributed program can produce many different kinds of outcomes based on the rates at which the constituent processes execute and the delays of messages. This is similar to a classical parallel program. A consistency model *specifies the valid outcomes*. If we increase the number of messages and reduce the delays, then it is possible to make the system behave like multicore processors. A strong form of consistency can be provided. On the other hand, if messages take an arbitrary amount of time and processes often crash, then a looser form or consistency can be guaranteed. Such systems are said to be eventually consistent, which basically translates to the fact that a message eventually reaches its destination. We shall discuss such trade-offs between the availability of the system, the consistency guarantees provided, the partition tolerance and latency of operations using the results of the classical CAP and PACELC theorems.

1.5.2 Part II: Distributed Systems Services

In the second part, we will focus on the services that are provided in distributed systems to distributed applications. There are many facilities that a distributed system needs to provide to its constituent applications such that they can use them to implement complex functionalities.

The first such basic facility is a distributed file system (Chapter 8). If all the nodes see a common view of a file system and can see the changes made by each other, then they can seamlessly coordinate among each other. There are several important file systems in this space such as AFS, NFS and Google's GFS. They make different assumptions regarding the underlying system and user behavior. We shall also discuss Facebook's photo storage system that is an example of a specialized file system primarily used for storing user-uploaded photos.

The next such basic facility that needs to be provided to distributed applications is a database (Chapter 9). Files are for unstructured data. However, for structured data storage in either SQL-based data stores or simple key-value stores, we need databases that all the processes can access. The database additionally needs to provide the classical four ACID (atomicity, consistency, isolation and durability) properties. In this context, it is important to look at the notion of database transactions. We want a transaction to execute in entirety or appear to not execute at all. Basically, it should appear to execute at a single instant of time. Furthermore, all the distributed processes need to agree on whether a given transaction should be allowed to make changes to permanent/durable state or not: whether it should commit or abort. In this context, distributed commit protocols are very important. This chapter will also discuss some distributed database designs that are used by Google and Amazon. We will appreciate the design decisions that prove to be useful in designing such large web-scale systems.

The next service that we shall discuss in Chapter 10 is blockchains. They are increasingly becoming very important. They are a special kind of data storage system where data once stored cannot be removed. A blockchain is an *immutable* data store. Let us think of it as a large linked list. All the participating processes (or a majority) agree with its contents. Any action in the distributed system such as updating the balance in a bank account can be added to such a blockchain. It permanently changes the account balances and moves the system to a new global state. Along with data nodes, it is possible to assign nodes that have some amount of logic in them. These are known as smart contracts in the parlance of the Ethereum blockchain. The idea here is that the code gets triggered when a certain

property becomes true. Along with the theory behind permissioned and permissionless blockchains, we will also discuss cryptocurrencies that use blockchains as their basic infrastructure.

Finally, we shall discuss distributed machine learning in Chapter 11. Most modern distributed systems need to ingest petabytes of data, train models and run inferencing tasks. This requires them to split the data as well as the model across machines. Because of partitioning the data and models, there is a need for a great deal of synchronization across machines, and we also need to have elaborate concurrency control mechanisms. The objectives are mostly the same: scalability, consistency and quick convergence. Achieving these objectives on modern hardware that may consist of thousands of multicore CPUs, GPUs, dedicated AI chips and running a wide variety of software, is very challenging. There are fundamental trade-offs between throughput and latency, which we shall look at in great detail.

1.5.3 Part III: Management of Distributed Systems

The third part of this book focuses on managing distributed systems and making them ready to run large real-world applications.

In Chapter 12, we shall look at the software framework in great detail. The middleware plays a very crucial role in any distributed system. It sits right above the operating system and is placed below the applications. Its main job is to facilitate seamless communication between all the participating processes and provide many additional services such as synchronization and authentication. The middleware has some more key responsibilities as well that include providing access to databases, caching services and executing scheduling routines. The latter performs load balancing and ensures that the system remains scalable. Modern middleware architectures provide other services as well such as access to microservices (small specialized services that run on dedicated nodes), monitoring/logging and stream processing of continuous data in real time.

In the next chapter (Chapter 13), we shall discuss fog and edge computing, which take computing closer to the end user and process data at the periphery of the network. We shall first study different edge and fog architectures. In many edge/fog based systems, low-latency and real-time computing is given a high priority. This is because such systems are typically deployed on industrial shop floors and smart grids. Moreover, resource management, scalability, data management and interoperability problems look very different in such systems that are not based on traditional compute servers. For example, a classical problem is how to split a large task

between an edge node, a fog node (closer to the cloud) and a cloud server. Where do we place the data? Over the last few years, a lot of specialized frameworks have come up for operating such systems where the data and computational tasks are spread all over the network. We shall study them and also look at how machine learning can be done at the edge/fog nodes.

Distributed programming languages are integral to the study of distributed systems (Chapter 14). In the last two decades, a lot of paradigms have emerged to program distributed systems and use their resources effectively. We shall look at some popular programming languages and paradigms in this space such as MPI, Map-Reduce and Google RPC, and a few rare ones such as Chapel and Erlang. This discussion will provide the reader an all-round perspective on programming distributed systems and introduce her to a wide variety of ideas in this space.

Then, we shall move to security and privacy (Chapter 15). These are increasingly becoming first-class concerns in the design of distributed systems. We shall review some key security protocols in the distributed space such as Kerberos and OAuth, authorization and access control mechanisms, secure data encryption and communication (TLS/SSL) and classical intrusion detection techniques. These concepts will provide the necessary background to the reader to understand more advanced concepts. We shall then move on to more advanced topics such as privacy-preserving computations, key management in distributed systems, multiparty computation and zero-trust architectures.

Finally, we shall discuss performance evaluation and benchmarking strategies in Chapter 16. We shall first discuss metrics such as throughput and latency, and variants thereof. These metrics will allow us to lay the foundation for understanding the effectiveness of load balancing, fault recovery and message passing algorithms that almost all distributed systems rely on. We will also discuss standard benchmarking tools and protocols along with bespoke methods tailored towards monitoring energy efficiency, elasticity, benchmark quality and performance bottleneck identification.

Summary

Summary 1.5.1

1. A distributed system is a collection of a plurality of machines that work together to achieve a common goal. The external world often perceives the entire system to be just one machine with a simple interface.

2. There are three kinds of distributed systems:

   (a) In synchronous systems, processes share a clock. There is a notion of a *round*. All the processes are expected to know when a round of computation ends and the next round begins.

   (b) In *partially synchronous* systems, the clocks are not fully synchronized with each other. There could be a drift between them that can be additive, multiplicative or both. There are many variants of partial synchrony. Sometimes these clock drift factors exist but are not known. In a few other models, these factors keep changing over time.

   (c) An *asynchronous* distributed system does not rely on any form of shared global clocks. Every process maintains its local time. The local time is updated based on messages that contain the local times of the sender nodes. The key idea is to preserve the notion of happens-before relationships where the cause appears to "happen before" the effect.

3. Synchronous and partially synchronous systems are easy to design and program. However, asynchronous systems are much more difficult primarily because of the Fischer-Lynch-Paterson (FLP) [Fischer et al., 1985] result that states that the agreement/consensus problem cannot be solved even if there is a single faulty process. The consensus problem is a problem of great importance in distributed systems. It can be shown that if this problem can be solved, then it is possible to use the solution to solve almost any other problem (subject to various reasonable assumptions and constraints).

4. In any large system, faults tend to happen and are unavoidable. Hence, any practical distributed system should have some degree of immunity to faults and be quite fault tolerant. Three terms in this space are quite relevant: fault, error and failure.

   Fault A *fault* is a deviation from ideal specifications. For example, if the specifications say that there should be no memory leak and there is a memory leak somewhere, it is a fault. However, a fault may never manifest (get activated).

   Error An *error* is a manifestation of a fault. It is defined as an erroneous/incorrect internal state. This means that if there is a situation, where a new data structure cannot be allocated, this situation indicates an error.

   Failure A *failure* is an erroneous state that is visible to an external entity (outsider). This means that the system does not produce any usable output and has completely failed to do its job. For example, the memory leak could lead to a program crash once the available memory gets exhausted. In this case, the program does not produce any output. Hence, this situation is defined to be a failure. On the other hand, we can have a situation where an error such as the system running out of memory does not lead to a failure. This can for instance happen when there is a perception that there is no memory left, and then the garbage collector is able to free up a lot of memory space. The system in this case does not fail.

5. Failures can be of several types depending upon their severity.

   Fail-stop The process simply crashes. Other processes are aware of it.

   Fail-silent The process crashes, but other processes may not be aware of it. Note that in both of the above definitions, we can replace a process with a node as well. A node runs a set of distributed processes.

   Byzantine failure This is the most pernicious kind of failure. The process becomes malicious. It sends erroneous mes-

sages to other processes to induce errors. It can even collude with other processes that have suffered from Byzantine failures, and try to bring down the system.

6. The book has three main parts: distributed system concepts, distributed services and management of distributed systems.

Part I

Concepts

Chapter 2

Time in Distributed Systems

> *A man with a watch knows what time it is. A man with two watches is never sure.*

Segal's law

Time and its measurement are concepts of fundamental importance. Ancient civilizations too practiced various ways of representing and measuring time with varying degrees of accuracy. People use the notion of time in many tasks such as preparing their daily schedules, coordinating various events, ordering events, and so on. It is important to note that in all of these applications, we have some inherent notion of how we benefit from knowing the time. Often, we are less concerned with the exact time and are more focused on whether the current time is before or after a particular event. In general, in distributed systems we may not have access to synchronized physical clocks. In such cases, we are more interested in the cause-effect relationships between events. We would like to effect to always appear to happen after the cause. The relationship between a cause and an effect is also known as *causality*, which is of vital significance to us in asynchronous distributed systems that do not rely on a global clock (refer to Chapter 1 for a definition).

When we move to computer systems, even one second of physical time is a long time in the processor's timescale. It corresponds to a window in which numerous actions can be realized. In fact, if we wish to specify actions at the instruction level, we need to specify time in terms of nanoseconds. We shall see that often there is no need to do so and representing the happens-before relationships (causality in some cases) between events is good enough.

In the following chapters, we will explore how the concepts of time and causality play a crucial role in computer systems, and in distributed systems specifically.

Before proceeding, let us motivate the problem and quickly see how the notion of "time" is currently used in computer systems.

- Assigning Timestamps: For many events that take place in a computing system, the operating system assigns a timestamp. Examples include the time at which a file is last accessed or written to.
- Performance and Resource Usage: Timestamps offer a way for the system to monitor the performance of various applications and also track the usage of various resources by processes.
- Timeouts: Some user programs may need to stop after a certain time duration has elapsed. This requires the program to know the time.
- Scheduling and Statistics: The operating system may use time information in process scheduling. Further, the operating system may use time information to keep track of the system time assigned to each process, the overall time each process is running for, tracking the login information of users and maintaining other types of bookkeeping information.
- Event Ordering: Timestamps are a way to arrive at a monotonic ordering of events that happen in the system.

From the above discussion, we note that one of the fundamental activities in a computer is the measurement and processing of time. In the physical world, 1 second is formally defined as the time it takes the Cesium 133 atom to make exactly 9,192,631,770 transitions. Atomic clocks rely on this phenomenon. However, most computers keep track of time by counting the number of oscillations of a quartz crystal. Specifically, one second on a computer corresponds to 32,768 oscillations of a quartz crystal. Sadly, the oscillation frequency of quartz crystals tends to vary with changes in temperature, voltage and even aging. Without frequent calibration, it can vary by 2-5 ppm [Walls and Gagnepain, 1992], which can translate to a second lost every 7-11 days. Therefore, unless corrected, the time maintained by a machine (laptop, desktop or server) can differ from the actual clock time, and the errors can also accumulate over time. If this is the case with a single machine, then maintaining globally synchronized time across multiple machines and that too for years at a stretch becomes extremely challenging.

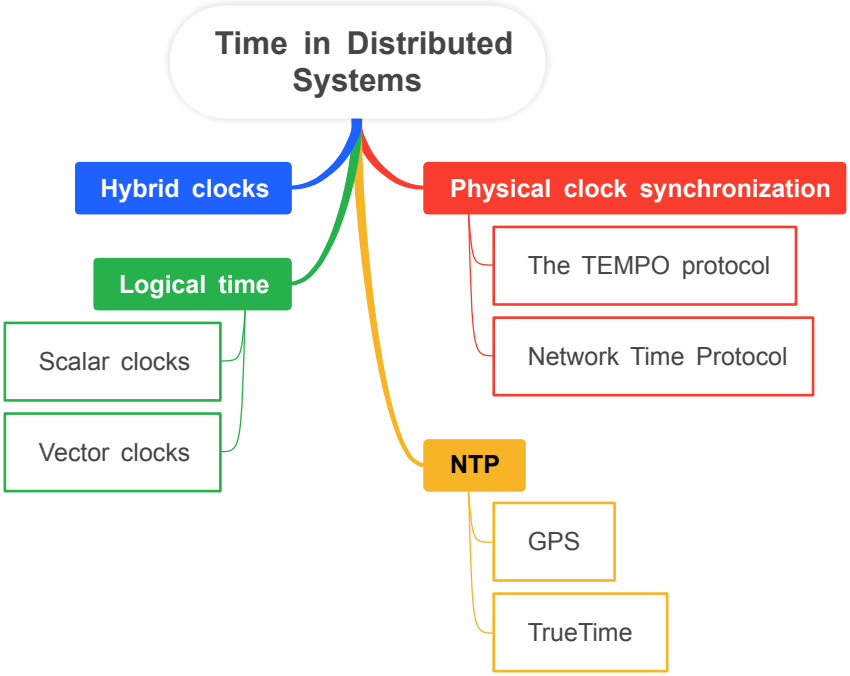


Figure 2.1: Organization of this chapter

Let us first start our study with clock synchronization algorithms that try to maintain a single clock time across a cluster of machines. We shall observe that we need to send frequent messages between them to ensure that no single machine drifts away. Henceforth, we shall use the term *node* to refer to a machine in a distributed system.

Organization of this Chapter

Figure 2.1 shows the organization of this chapter. We will start out with looking at the synchronization of physical clocks. This is a tempting solution given that we are used to thinking in terms of real times, intervals and rounds of computation. However, it turns out that there are practical limits to clock synchronization. It is limited by the amount of jitter in the network. Nevertheless, it is possible to design protocols at the level of small clusters

as well as for planet-scale systems. Network time protocols can be used to set the time accurately in devices. It is further possible to enhance the accuracy using the GPS protocol that fetches the real time from at least 4 satellites.

Google's TrueTime extends these ideas to create a composite solution with different kinds of time servers. In this case, an interval is returned where the absolute time is guaranteed to lie within the interval. Designing algorithms that use intervals instead of crisp times is more involved.

Hence, it is a better idea to use logical clocks that reflect the causality of events. The flip side is that timestamps may not always be totally ordered (or comparable). Even when all pairs of timestamps are comparable, it is not necessarily true that if one timestamp is less than the other, then the former event happened before the latter event. We will study scalar clocks and vector clocks. The latter is able to detect causality between events if the clocks are totally ordered.

Finally, we shall introduce a hybrid clock that combines both physical and logical clocks. In a certain sense, such clocks take the best of both worlds. They limit the state that logical clocks need to maintain. They leverage loosely synchronized physical clocks for coarse time keeping and then use logical clocks to break ties.

2.1 Physical Clock Synchronization

Having a synchronized clock in a distributed system is extremely beneficial. A clock synchronization algorithm refers to a mechanism that is used to align the clocks of nodes in a distributed system such that every node in the system is sure that the maximum clock drift (across the nodes) is limited to a threshold Δ.

2.1.1 Synchronization of Physical Clocks: Overview

It turns out that maintaining synchronized physical time across the nodes in a system is unusually difficult. Some difficulties arise from the fact that the local time at each node can drift due to varying ambient conditions. Any attempt at clock synchronization is limited in its efficacy due to unpredictable network delays and the fact that these links can sometimes fail. Moreover, the lack of access to global knowledge or global state limits the decisions that nodes can take with respect to correcting their clocks. They are limited to using local knowledge and the information that they get by exchanging messages with other nodes. Errors tend to carry over and get

amplified in a distributed system. There are additional challenges such as the lack of any centralized control and lack of limits on physical distances between the nodes.

This situation calls for the design of efficient algorithms and protocols that address how nodes in a network can efficiently *synchronize* their clocks. A lot of the foundational work was done by Marzullo [Marzullo, 1984] in his PhD thesis (published in 1984). There have been a sequence of developments in this direction since then.

Algorithms for this problem use a range of techniques including peer-to-peer information exchange and master-slave models for synchronizing time across nodes in a network. It is easier to solve such problems when there is a centralized server. The quintessential technique, in both the centralized and distributed cases, is for a pair of nodes to estimate the average transmission delay and their clock offset by using a sequence of ping and reply messages. For obvious reasons, algorithms that assume an upper bound on the transmission delay are usually simpler than those that do not make such assumptions.

Fact 2.1.1

A system with N clocks cannot be synchronized with a synchronization error less than $(max - min) \cdot (1 - \frac{1}{N})$. max and min denote the maximum and minimum transmission delays across the nodes in the network, respectively.

Other considerations that play a role in the design of these algorithms include the size and extent/spread of the network, the desired accuracy of synchronization and the auxiliary support available. There are theoretical limits to the accuracy that can be achieved by *any* algorithm. Specifically, Lundelius and Lynch [Lundelius and Lynch, 1984] show that a system with N clocks cannot be synchronized with a synchronization error less than $(max - min) \cdot (1 - \frac{1}{N})$. max and min denote the maximum and minimum transmission delays across the nodes in the network, respectively. This means that there are practical limits on the accuracy. They depend on multiple factors including the network conditions and the accuracy of the clocks at the individual nodes.

Most of the algorithms for time synchronization employ a range of techniques for fault tolerance. For instance, in a classical ping and reply interaction, if one side does not respond within a reasonable time period, the other side will typically call this round of interaction as unsuccessful and ignores the measurements from such unsuccessful rounds. Given the uncertainties

in message transmission and other potential pitfalls such as failed or inaccurate clocks, another common technique is to discard all measurements that show significant deviations.

2.1.2 The TEMPO Protocol

One of the early algorithms/protocols to maintain the time in a local network is the TEMPO protocol proposed by Gusella and Zatti [Gusella and Zatti, 1983]. This algorithm adjusts the time of the day of the devices in a local network. Based on this algorithm, Berkeley UNIX® (version 4.2 BSD) introduced a system call `adjtime()` that allowed the user to set the time on a system.

The Algorithm

Consider a situation where processes A and B are running on two different nodes in the network. We want to estimate the clock skew between them. A sends a message to Process B at time t_A. On receiving such a message, Process B sends a reply to Process A at time t_B.

Given that this paper was published way back in 1983, the authors also took into account the measurement delay. This is also relevant today given that measuring the time often requires a system call, which can take several microseconds. We thus add two error terms e_A and e_B. Hence, the timestamps are governed by the following relationships: $\text{timestamp}_A = t_A - e_A$ and $\text{timestamp}_B = t_B - e_B$.

Moreover, the clocks of A and B may themselves be adrift by Δ_A and Δ_B, respectively. Using the above two relationships, and denoting the real transmission delay from A to B as t_{AB}, we can formulate the following equation:

$$\begin{aligned} d_{AB} &= timestamp_B - timestamp_A \\ &= (t_{B1} - e_{B1}) - (t_{A1} - e_{A1}) = (\underbrace{t_{AB}}_{t_{B1}-t_{A1}} + \Delta) - \underbrace{\epsilon_{AB}}_{e_{B1}-e_{A1}} \end{aligned} \tag{2.1}$$

Here d_{AB} is the delay that B computes based on the differences of the timestamps. This is related to the differences of the actual times of transmission and receiving the message (resp.), the relative clock skew (Δ) and the errors in reading the timestamps. t_{A1} or t_{B1} indicate the respective times in round 1. On similar lines, t_{A2} and t_{B2} (resp.) indicate the times in communication round 2.

Subsequently, Process B sends a timestamped reply to Process A. The same set of computations are repeated.

$$d_{BA} = (t_{A2} - e_{A2}) - (t_{B2} - e_{B2}) = (t_{BA} - \Delta) - \epsilon_{BA} \tag{2.2}$$

Here, d_{BA} is the delay that A computes on the basis of the difference of the timestamps. Note that in this case, the clock skew gets subtracted.

Using Equations 2.1 and 2.2, Process A can compute the difference in the transmission delays as follows:

$$\Delta' = \frac{d_{AB} - d_{BA}}{2} = \Delta + \frac{t_{AB} - t_{BA}}{2} - \frac{\epsilon_{AB} - \epsilon_{BA}}{2} \tag{2.3}$$

We can repeat the aforementioned steps to estimate Δ. Assume that the random variables corresponding to ϵ_{AB} and ϵ_{BA} are independent and symmetric. We can repeat the above experiment N times so that the average Δ' can be computed as follows.

$$\begin{aligned} \Delta' &= \frac{\sum_{i=1}^{N} d_{AB_i} - d_{BA_i}}{N} \\ &= \Delta + \frac{\sum_{i=1}^{N} \frac{t_{AB_i} - t_{BA_i}}{2}}{N} - \frac{\sum_{i=1}^{N} \frac{\epsilon_{AB_i} - \epsilon_{BA_i}}{2}}{N} \end{aligned} \tag{2.4}$$

Notice that from Equation 2.4 that the second and the third terms on the right-hand side have a mean of 0, subject to some assumptions. We assume that independent measurements are identical and independently distributed, and N is a large value. We further assume that the transmission times are measurement errors are symmetric. Using the strong law of large numbers, we arrive at this result. Hence, we can conclude that the right-hand side converges to Δ. Once Δ is found out, the local times can be adjusted accordingly. A variant of this algorithm that does not assume any errors in reading the timestamps is the Cristian's algorithm.

This observation can be used to design the protocol that nodes connected on a LAN use to synchronize their clocks. The TEMPO protocol designates one computer in the local network as the *master* and the rest as *slaves*. The master is responsible for initiating and coordinating the time synchronization process. It specifically executes the following steps:

1. The master interacts with each of the slave nodes, say $S_1, S_2, \cdots$, and obtains estimates of Δ_{S_1}, Δ_{S_2}, $\cdots$, with respect to each of the slave nodes.

2. The master then computes the mean of the following quantities: Δ_{S_1}, Δ_{S_2}, $\cdots$.

3. The master directs each slave node S_i to adjust its clock by an offset equal to the corresponding clock skew Δ_{S_i}.

Notice that the above protocol requires some slave nodes to set their time to a point in the past. This can create situations that make applications behave inconsistently. For example, there are many applications like the *make* or *rsync* utilities that explicitly rely on file access timestamps. If they see a file that has been modified in the future, they will find it hard to process this information.

Fault Tolerance

The TEMPO protocol is designed with simple fault tolerance in mind. The failure of a slave to communicate with the master within a specified timeout interval lets the master conclude that the slave node is non-operational. Similarly, if the slave nodes do not receive any message from the master within a specified time, they conclude that the master node is unavailable and start the procedure to elect a new master.

2.1.3 The Network Time Protocol (NTP)

The Network Time Protocol (NTP) is described in RFC 958 [Mills, 1985]. It specifies how computers on a network need to periodically set their times such that all of them have a consistent view of the real (physical) time. To this end, the protocol organizes computers into two sets of entities: clients and servers. Clients ask the real time from a server within the same network, and are thus passive in the protocol. Servers are arranged into multiple strata. Servers within a stratum are referred to as *peers*. As of 2024, NTP is ubiquitously used. For example, Microsoft® uses it to synchronize the times on all Windows® nodes (part of the Windows Time Service). It is used in all kinds of sectors such as telecommunications, IoT installations, corporate networks and ISPs.

Stratum 0, the highest level, consists of the most accurate set of time servers. These are also referred to as *reference clocks*. They use devices such as atomic clocks, and are thus extremely accurate.

Devices in Stratum 1 have a direct connection to one or more devices in Stratum 0 in addition to having connections to their peers in the same stratum. Devices in this stratum are also known as primary clocks or primary time servers. They might be a few microseconds off as compared to Stratum 0 clocks. These primary servers can also work as fallback clocks that can

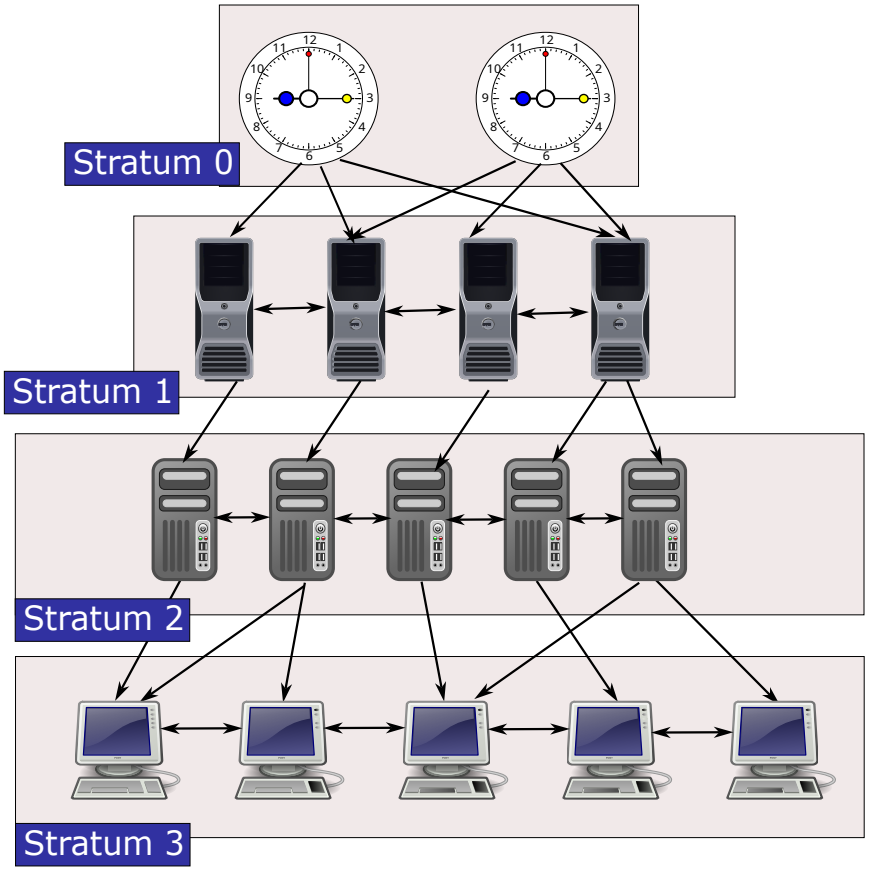


Figure 2.2: A set of servers arranged into various strata in the NTP protocol.

be made to synchronize with each other just in case Stratum 0 servers are unavailable.

Devices in Strata 2 and 3 synchronize with devices in the next higher strata and with those in the same strata. Figure 2.2 shows a typical scenario.

There are two types of communication between the servers to obtain the current time: one is between peers, and the other is between devices in one stratum to those in the immediately higher stratum. Since servers across strata and also those within a stratum may have differences in clocks, and these differences can grow arbitrarily over time, an initial set of synchronization operations may not be enough. Therefore, peers in the NTP protocol exchange periodic messages to set and reset their local time. The interaction between two peers A and B is characterized by the following sequence of

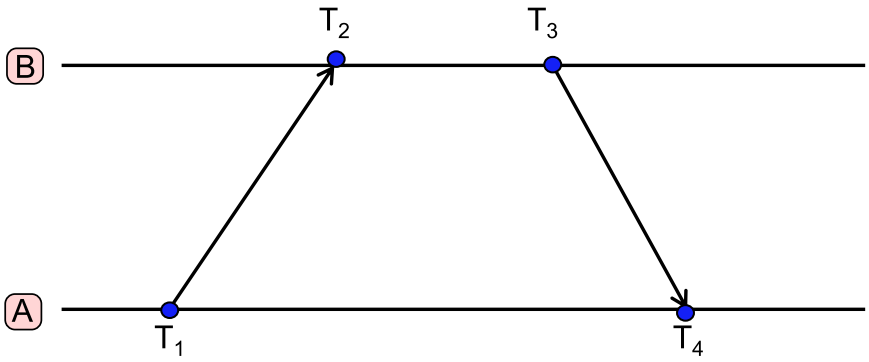


Figure 2.3: A pair of messages exchanged between A and B. The send and receive times at A and B (resp.) are locally recorded.

messages.

Peer A sends a message to Peer B and records the time that the message is sent as t_1. Peer B records the time at which the message is received at B as t_2, and then sends an acknowledgement to A at time t_3. This acknowledgement is received by A at time t_4. Each such message and acknowledgement is referred to as a trial. Figure 2.3 shows these four messages between A and B.

For each trial, the clock skew Δ and the round trip delay λ of are approximately given by the following relationships. Our line of reasoning is similar to that followed in the TEMPO protocol (Section 2.1.2).

B computes the following:

$$t_2 = t_1 + \lambda/2 + \Delta \tag{2.5}$$

Similarly, A computes the following equation:

$$t_4 = t_3 + \lambda/2 - \Delta \tag{2.6}$$

Of course, we can make the estimate more accurate by considering multiple trials. Let us nonetheless confine our attention to a single trial. We can solve these two equations. The values of λ and Δ are as follows:

$$\begin{aligned} \lambda &= (t_2 - t_1) + (t_4 - t_3) \\ \Delta &= \frac{(t_2 - t_1) - (t_4 - t_3)}{2} \end{aligned} \tag{2.7}$$

Each peer maintains offset-delay pairs of the form $\langle o_i, d_i \rangle$, where o_i is a measure of the clock skew (Δ) and d_i is a measure of the transmission delay

(λ). The eight most recent (o_i, d_i) pairs are retained. The value of o_i that corresponds to the minimum d_i is chosen as the clock skew. This can be used to correct the clocks of nodes in the lower stratum.

Limitations of NTP

Notice from the above that by having a set of continuously interacting peers, it is possible to have a physical time that is synchronized across computers. The typical errors in synchronization are normally small. However, in a large-scale distributed system, the errors can go up to hundreds of milliseconds.

NTP works on the premise that all the systems can be arranged in a hierarchy and the required communication infrastructure can be set up. This assumption is difficult to maintain in a fully distributed system. Even if such a hierarchy and communication support is created, the typical physical distances can potentially involve large round trip times resulting in reduced accuracy. Particularly, the accuracy using NTP in networked systems is of the order of a few milliseconds. This error can increase to a few hundreds of milliseconds in rare cases (see the experiment suggested in Question 2.6). This accuracy compares very unfavorably to running NTP on a LAN, which can be accurate at the level of a few milliseconds. A planet-scale distributed system may not be able to function correctly using physical time that has errors of the order of milliseconds or hundreds of milliseconds. As a result, NTP in its present form is not usable in distributed systems, particularly those that are very large and rely on millisecond-level synchronization across nodes.

Nevertheless, distributed systems too need to measure time for a variety of reasons. Some of these are listed below.

- Distributed Transaction Processing: Measuring time and assigning timestamps is important in transaction processing.

- Distributed Algorithms: Timestamps may also be essential in applications such as mutual exclusion. In some situations, distributed systems use the notion of timeout to abort actions. This requires measuring the elapsed time.

- Performance Monitoring: Many times, it is important to measure how long certain events took or measure the throughput of a given system. All of these require access to the physical time.

Given the above requirements and the difficulty of adopting the NTP approach, what do distributed systems do to maintain the time? It turns out that by their nature, distributed systems make progress in spurts and one way to measure time is to understand how various events in a distributed system can be ordered. The ordering can be used as an alternative mechanism, and the entire theory of distributed systems can be created based on how events are ordered with respect to each other as opposed to relying on the wall clock time. In computer science terms, accurate physical clocks allow us to *totally order* events, which as we have seen is difficult. Instead, it is better to rely on a partial order that relies on ordering. This partial order can incorporate cause-effect relationships (also known as causality).

2.1.4 Recent Developments

There are two notable developments related to time synchronization in distributed systems. First, use cases such as distributed transactions, financial transactions and distributed databases have rekindled interest in synchronizing the physical time in a distributed system. Such applications require a degree of precision that is of the order of nanoseconds [Geng et al., 2018]. These use cases led to the design of physical clock synchronization algorithms that used novel techniques from signal processing and estimation theory to achieve better accuracy.

2.2 NTP at a Distributed Scale

2.2.1 GPS

We are all familiar with the term GPS, especially after the advent of location-based services such as ride-sharing apps and food-delivery apps. It provides our precise location to service providers such that the pizza delivery person knows exactly where to come, even though the address is not provided. However, it turns out that physical time synchronization lies at the heart of this approach.

GPS stands for the "Global Positioning System". It is a satellite-based navigation system that is owned by the US government. Many other governments also own similar systems. They are useful for both civilian and military use. They are classified as GNSS systems (Global Navigation Satellite Systems). We shall limit our discussion to GPS in this section. Note that the principles behind the operation of other systems is almost the same.

Basic Operation

Let us understand the unknowns first. A GPS receiver such as a mobile phone has a local time and coordinate (with respect to a standardized coordinate system). Let us assume that its clock skew is Δ – the difference between the local time and the real physical time (at that timezone or with respect to the Greenwich Mean Time (GMT)). Δ is expected to be a small value. Next, it has $\langle x, y, z \rangle$ coordinates. We thus have four unknowns, and we will consequently require four equations. The precise location is used to service-providing apps, and the clock skew Δ is important for synchronizing the clocks. This can be thought of as a very desirable side effect of the location discovery process using GPS signals.

Every satellite broadcasts its local time (read from an atomic clock) and its coordinates in 3-dimensional space. Assume that the GPS receiver receives messages from four satellites. For satellite i, it receives its 3D coordinates $\langle x_i, y_i, z_i \rangle$ and the time t_i. Assume that the message was received at time t (local time of the receiver).

$$\sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2} = c \times (t - t_i + \Delta) \tag{2.8}$$

We know that the distance between the receiver and the satellite is equal to the product of the speed of light (c) and the time it takes the signal to reach the receiver from the satellite. The time of transmission is $t - t_i$ if the clock skew is 0. However, we need to add Δ because the clock skew is often non-zero. The final relationship is shown in Equation 2.8. Similarly, we can generate three more such equations for three other satellites. Note that Δ will be the same for all the satellites because we assume that the satellites are mutually clock synchronized, and their time is produced by an ultra-accurate atomic clock.

This appears to be quite simple as long as the signals from four different satellites are available. However, in practice, things are quite complicated. First is that the satellite itself is moving, and the receiver may be moving as well. Hence, there will be Doppler shifts, and this will impact the timing. Second, there are various sources of delay in the troposphere and ionosphere. The ionosphere (60-1000 kms above the surface) can delay and refract radio waves. This changes the timing of the signals. Similarly, in the troposphere (0-12 km), GPS signals can get slowed down based on atmospheric conditions. The refractive index is determined by the temperature, humidity, pressure and mix of gases. Note that even if the effect is minimal, the error induced will still be significant because we are working with very short time durations here. Even a 1% error, can induce an error of 200 kms.

Thankfully, in real life, the delays are not that much, hence, the inaccuracy is in meters, not kilometers.

Accurate GPS Estimates of the Time and Position

Given these reasons, it is necessary to accurately estimate the time and position of a GPS receiver. We thus require signals from as many satellites as possible. It is not uncommon to process data from 10-15 satellites. The aim is to arrive at a precise estimate of time and location by using sophisticated signal processing techniques to remove noise, and then performing least-squares-based adjustments.

In GPS, there are 32 satellites in the medium earth orbit ($\approx$ 20,000 km). They rotate in six different orbital planes. The aim is to have at least four satellites over any point on earth at all points of time. In practice, we need many more satellites.

GPS satellites send a lot of additional data such that it is easy for receivers to correct the information that they receive and remove noise. For example, they send information related to satellite orbits, their health, ionospheric and tropospheric data, clock correction data and sometimes additional ground-based stations also assist in this process by sending more information. They specifically send two kinds of data: ephemeris data and almanac data. Ephemeris data is sent frequently (once every 30 seconds); it contains position and time information. Almanac data is sent infrequently (once every 12.5 minutes). It contains information about the satellites in the system, their orbits and their health. It also informs the device about the satellites that are currently in its vicinity.

Receivers on their side also collect a lot of additional data such as the signal strengths, signal-to-noise ratios, Doppler shifts, uncertainties and drifts in these values, etc. A modern GPS chip in phones produces around 32 parameters of interest. This is raw GPS data, which needs to be processed using signal-processing techniques such as Kalman filters to remove all types of noise, and arrives at precise estimates for the position and time.

2.2.2 TrueTime

The next challenge is to design a clock synchronization protocol that is suitable for a system with a very large number of nodes distributed worldwide. Corbett et al. [Corbett et al., 2013] discuss one such protocol – *TrueTime*. They design it for planet-scale systems.

Consider a setting where data centers are spread across the planet. Each

data center has one or more *Time Master* servers. There are two kinds of such servers (refer to Figure 2.4).

- GPS Time Master: These servers contain GPS receiver nodes that can receive GPS signals, specifically precise time information from satellites.
- Armageddon Master: These Time Master servers contain atomic clocks that are highly accurate. Time information from local atomic clocks can be supplemented with the time received from GPS Time Masters. These servers are particularly useful when satellites are out of range due to extreme weather events.

The engineering decision behind using GPS-based time information and atomic clocks is to ensure redundancy and create an opportunity for constructing an accurate time base by fusing multiple sources of information. Usually, it is observed that the factors that affect the failure of these two time systems are independent. For instance, satellite communication on which GPS relies may be impacted by interference from radio waves or weather events. These radio waves do not impact the functioning of atomic clocks.

Time Masters in TrueTime periodically exchange and compare their time with each other. The time exchange protocol is along the lines of the NTP protocol. Each Time Master also checks the local clock for integrity. In case these checks fail, the Time Master voluntarily stops being a time server until the issue is resolved.

Let us now outline how clients use TrueTime services. A client pings a set of Time Master servers that can either be GPS Time Masters or Armageddon Time Masters. Next, there is a need to *fuse* the information gathered from the replies received from the Time Masters. Clients use statistical techniques to drop the outlier responses and compute an accurate time estimate from the set of valid responses.

There are several strategies that can be followed here. The conventional strategy is as follows. Normally, we take the mean after discarding the outliers (on the lines of the TEMPO protocol). After synchronizing the local clock with time servers, clients can use the time or send it to other nodes in the system. Note that till the next point of time re-synchronization, the error in the physical time of the client will continue to increase due to local clock drift.

Applications using TrueTime follow the TrueTime API, which actually does something quite unconventional. In this API, the timestamp that is

returned is actually an interval. Each such timestamp is of the datatype `TTInterval` of the form [earliest, latest]. The two values correspond to the start and end times of the interval. The actual time is guaranteed to lie within the interval.

Specifically, the API has three functions. The function `TT.now()` returns the current time as an interval. The function `TT.after(t)`, returns **true** if time t has certainly elapsed, and the function TT.before(t) returns **true** if time t is yet to occur (with absolute certainty).

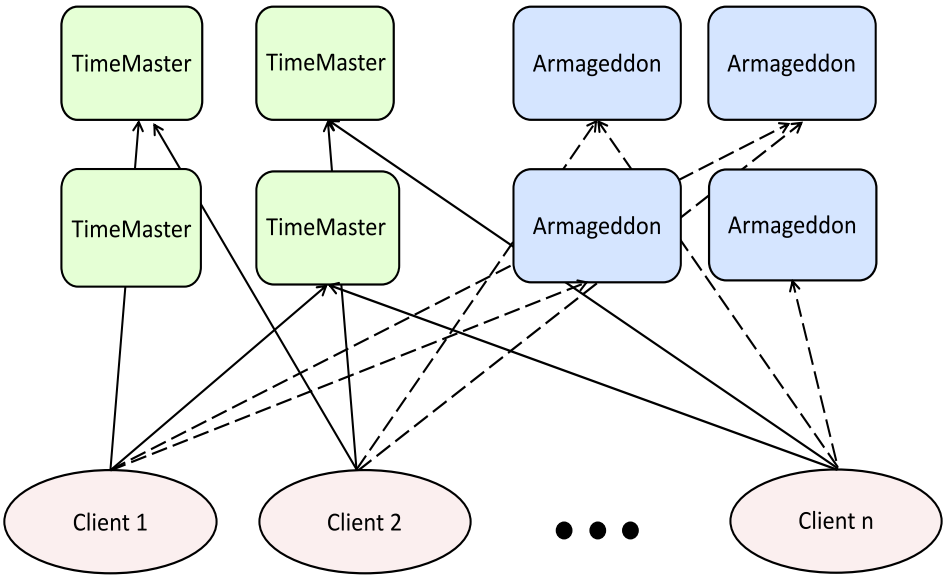


Figure 2.4: An illustration of the Google TrueTime architecture. Solid lines represent clients interacting with TrueTime Time Master servers and dashed lines show a client interacting with Armageddon masters.

TrueTime guarantees ensures that an invocation of the form `tt = TT.now()`, `tt.earliest` $\leq$ $\texttt{t}_{\texttt{abs}}(\texttt{e})$ $\leq$ `tt.latest`, where t_{abs} is the absolute time when the event happened. Google states in its official documentation that the duration of this interval is between 1 ms to 7 ms (in most practical scenarios).

Notice that TrueTime has two novel contributions. One is to represent time as an interval instead of as an absolute scalar quantity. The second novel aspect of TrueTime is to utilize GPS and atomic clock based time references, and fuse their results. The latter allows TrueTime to use algorithms similar NTP to synchronize clocks, and provide strong guarantees on the interval of uncertainty.

TrueTime has facilitated many distributed applications that rely on the

availability of accurate physical time. A good example is the Google Spanner distributed database system [Bacon et al., 2017] that uses TrueTime to realize strict concurrency control for transactions at a global scale. Further details are provided in Chapter 9.

Other Related Systems

The design of TrueTime by Corbett et al. [Corbett et al., 2013] led to the design of similar systems. Amazon launched its version of a time service called *TimeSync*, which uses both GPS-synchronized clocks and atomic clocks. This service also uses many ideas similar to that of TrueTime and NTP. The TimeSync service is now used in Amazon products such as Amazon's distributed database DynamoDB [Sivasubramanian, 2012].

Facebook in 2020 also moved to a large-scale NTP solution called `Chrony` [Facebook, 2020].

Limitations of TrueTime and Similar Systems

TrueTime and other related systems come under a class of highly engineered solutions that require the presence of special purpose hardware on time servers in the form of GPS receivers and precise atomic clocks. The atomic clocks used by Google are accurate to within 1 second in a span of 10 million years. Such high level of precision cannot be feasibly achieved in many settings. Second, it is not the hardware and instrumentation alone that supports systems such as TrueTime. The communication latency in the interactions between clients and TrueTime servers and the Armageddon Masters does play a very important role too. After all, the clock synchronization accuracy is bound by the Lundelius and Lynch bound (see Section 2.1.1). This is why the TrueTime system relies on having a dedicated, fault-tolerant and high-speed communication fabric that has a very low uncertainty in the network latency. Recall that the error in clock synchronization is proportional to the jitter in the network (uncertainty in latency).

Additionally, TrueTime returns a time interval as a timestamp. When the interval is small enough, this suffices to order events and compare the timestamps of events to know which event precedes another. Clearly, if two intervals are not overlapped, then they are comparable. In the unlikely situation that such an ordering is not possible due to overlapping intervals, TrueTime recommends that applications wait till the period of uncertainty elapses. This introduces unexpected delays in event processing, and also reduces the achievable concurrency in the system.

Remark 2.2.1

The LL (Lundelius and Lynch) bound makes accurate clock synchronization quite challenging. Almost always, there will be a minor amount of network jitter. As a result, accurately synchronizing the clocks and running computations across the network that rely on a globally synchronized time base is very difficult. Protocols such as TrueTime make the task simpler by returning an accurate interval that encompasses the true time. If the application is willing to tolerate some delays, then it is still possible to rely on a single time base. Nevertheless, designing algorithms where the physical time is actually an interval is quite challenging.
However, in distributed computing we simply need to order events most of the time. The relative order is more important than an absolute order of events. Furthermore, we do not need to order all pairs of events, we only need to order specific pairs. Hence, we often desire a partial order between events of interest. Let us thus explore logical clocks that allow us to create such an order without a necessity for expensive clock synchronization.

2.2.3 Leap Seconds and Smearing

Despite all the careful theoretical and engineering efforts, there is still one issue that bedevils most practical systems. Notice that time is also a measure of how the earth revolves around the sun. The speed with which the earth revolves around the sun varies slightly due to several factors. This results in a difference of roughly one second over a couple of years between physical clocks and the "real" time (measured by the revolution of the earth around the sun). To adjust for this difference, the international body named the General Conference on Weights and Measurements (CGPM) [tous droits réservés, 2023] adopted the practice of adding or removing one second from physical clocks. The last such addition to the local time happened on December 31^{st} 2016. This extra second added or subtracted from physical clocks is referred to as a *leap second*. In formal terminology, this is known as *adjustment*.

Systems such as TrueTime also adopt the model of making such adjustments whenever there is a need. One additional technique that is used is to propagate the change across the multiple instances. This practice is also known as *leap second smearing*.

2.3 Logical Time

Let us use a partial ordering of events in a distributed system to create a notion of time. Such a mechanism allows distributed systems to use the "logical time", which is based on the ordering of events, instead of the physical time. It was originally proposed by Leslie Lamport in 1978 (refer to [Lamport, 1978]).

Logical time is a mapping from events in a distributed system to either scalar or vector timestamps. The events in a distributed system correspond to a local computation, message send and receive events. A distributed system can thus be visualized as a system of nodes processing *events*. Events within a node and those across nodes may have causal (cause-effect) dependencies, where one leads to the other or one what happens before the other. For example, an event corresponding to a message *send* on node P has to *happen before* the message is received at node Q. These causal dependencies induce a partial order on the set of events across the nodes in a distributed system.

Logical time aims to capture such dependencies between events. To make the description formal, let us define a few terms. Let E_i be the set of events happening in Process P_i. Let $E = \cup_i E_i$ denote the entire set of events happening across all processes (the entire system). We need to also number the events that happen in each process. In Process P_i, an event is represented as e_i^n, where $n \in \mathbb{N}$.

Now, e_i^m *happens before* e_i^n if and only if $m < n$ (for all positive integers m and n). This means that there is a total ordering between all the events happening in the same process. Similarly, if P_i and P_j are two different processes, and P_i sends a message to P_j, the corresponding send event must *happen before* the receive event. If e is the send event and e' is the receive event, then we represent this scenario using the following relationship:
$e \overset{msg}{\rightarrow} e'$.

These are the two most common rules for setting the order between events. We shall use the binary relation $\rightarrow$ across pairs of events to define a generic happens-before relationship. Let us define it formally.

$$e_i^m \rightarrow e_j^n \equiv \begin{cases} i = j \text{ and } m < n, & \text{or} \\ e_i^m \overset{msg}{\rightarrow} e_j^n, & \text{or} \\ \exists e_k^p \in E \text{ such that } e_i^m \rightarrow e_k^p & \text{and } e_k^p \rightarrow e_j^n \end{cases} \tag{2.9}$$

Notice from the above definition that indeed the relation $\rightarrow$ is a partial order. It follows the rule of transitivity. Now, there could be events that are

not related according to $\rightarrow$. We say that for two events e_1 and e_2, if $e_1 \rightarrow e_2$ or vice versa, then the two events *causally affect* each other. Otherwise, the two events are said to be *concurrent*. We use the symbol $e_1 \parallel e_2$ to indicate that the two events are concurrent.

We now build on the relation $\rightarrow$ between pairs of events to define the notion of logical time. Just like physical time, logical time also assigns a timestamp to each event. It is defined as a timestamping function: $T : E \rightarrow \mathbb{N}$. These timestamps ensure that if there is a happens-before relationship of the form $e_1 \rightarrow e_2$, then the timestamp of e_1 ($T(e_1)$) is smaller than the timestamp of e_2 ($T(e_2)$). Note that the converse need not be true. If $T(e_1) < T(e_2)$, it does not mean that $e_1 \rightarrow e_2$. We cannot infer causality this way. It needs to follow three rules (refer to Definition 2.3.1).

Definition 2.3.1 Logical Clock

A logical clock follows three rules while assigning timestamps to events e_1 and e_2. Specifically, when one of the three conditions holds $T(e_1) < T(e_2)$.

Same-Process Rule If e_1 and e_2 are two events in the same process and e_1 happens before e_2. Note that events can be totally ordered within the same process.

Send-Receive Rule If e_1 is a send event and e_2 is the corresponding receive event (possibly in a different process).

Transitive Rule There is an event e_3 such that $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$.

2.3.1 Scalar Clocks

A logical clock can be implemented quite easily. Every process u maintains a monotonically increasing local clock $\mathtt{l_u}$, which is initialized to 0. Every time it processes a new event, it increments its local clock. A message is timestamped with the local time of the sender (timestamp of the send event). When the receiver receives a message, it computes the maximum of the timestamp of the receive event and the timestamp that it has received (as a part of the message). It sets its local clock to the maximum value and adds one to it. Let us elaborate. Let the send event be e_1 (by Process $\mathtt{u}$) and let the receive event be e_2 (Process $\mathtt{v}$).

Algorithm 1 Lamport's Logical Clock (Scalar)

1: $\mathtt{l_u} \leftarrow \mathtt{l_u} + 1$
2: $T(e_1) \leftarrow \mathtt{l_u}$
3: Timestamp the message with $\mathtt{l_u}$
4: $\mathtt{l_v} \leftarrow \max(\mathtt{l_v}, T(e_1)) + 1$
5: $T(e_2) \leftarrow \mathtt{l_v}$

The Same-Process Rule is satisfied here because before the send event, we increment $\mathtt{l_u}$. Similarly, the receive event's (e_2's) timestamp is greater than the previous timestamp of the receiver ($\mathtt{l_v}$). The Send-Receive Rule is also satisfied here. The receive event always has a greater timestamp than the send event given that we are adding 1. We can now prove by mathematical induction that the Transitive Rule is also satisfied. This is left as an exercise for the reader. Figure 2.5 shows an example.

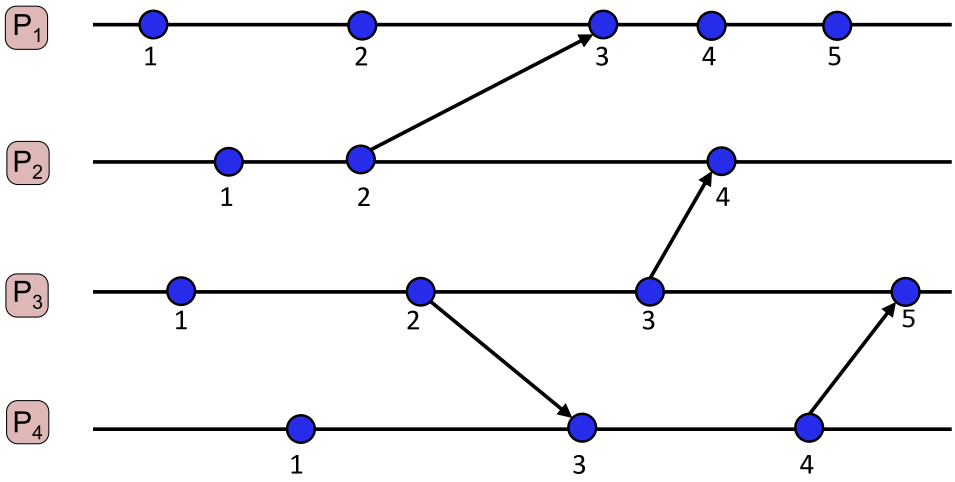


Figure 2.5: A set of four processes using scalar time. Note the use of the Same-Process and Send-Receive Rules.

The notion of logical scalar time has found applications in several fundamental problems in distributed computing. For instance, we will see in Chapter 4 that scalar time is useful in designing algorithms for ensuring mutual exclusion in a distributed setting.

Scalar clocks have a significant limitation. We can observe in Figure 2.5 that causality cannot be inferred from logical times. We cannot infer that $e_1 \rightarrow e_2$ if $T(e_1) < T(e_2)$. Many times we want causality information.

In this case, vector clocks [Mattern et al., 1988, Fidge, 1988, Liskov and Ladin, 1986] are used [1]. It is possible to further extend vector clocks to matrix clocks [Wuu and Bernstein, 1984]. However, this is beyond the scope of the book. Let us thus proceed to look at vector clocks.

Notwithstanding this limitation, there are cases when there is a requirement to create a total ordering between events. In this case, we can think of time as a tuple $T(e_1) = \langle u_1, t_1 \rangle$. u_1 is the ID of the process and t_1 is the scalar timestamp associated with event e_1. The following relationship orders the logical time across events.

$$T(e_1) < T(e_2) \equiv (t_1 < t_2) \mid\mid ((t_1 = t_2) \wedge (u_1 < u_2)) \tag{2.10}$$

This is also known as lexicographic ordering, where we first order logical times by their scalar timestamps, and then if there is a tie, it is resolved using the process id. The advantage of this is the creation of a total order. However, that is still not very useful because causality cannot be inferred.

Observation 2.3.1

Scalar timestamps provide the notion of logical time by ensure that causally-related events have monotonically increasing timestamps. However, the converse is not true. Causality cannot be inferred from the value of the scalar timestamp.

2.3.2 Vector Clocks

Vector time solves the problem of causality inference by representing time as an n-dimensional vector, where n is the number of nodes in the distributed system. Consider a vector timestamp v. It represents the local vector time of process P_i. Each element of the vector is a non-negative integer. The i^{th} component of vector v_i can be thought of as the local logical (scalar) time at P_i. $v_i[j]$ $(i \neq j)$ represents the latest *estimate* that P_i has about the local time at Process P_j. If $v_i[j] = \eta$, then P_i knows that the local time at P_j is at least η. The entire vector v_i constitutes a view of the global logical time at P_i, and is therefore used by P_i to timestamp all its events.

[1] There is some debate about the person who first proposed the notion of logical vector timestamps. See the blog by Kuper `https://decomposition.al/blog/2023/04/08/who-invented-vector-clocks/`

Remark 2.3.1

The timestamp of an event is now a vector of n dimensions.

When P_i executes an event e, it increments $v_i[i]$ by 1 and sets the timestamp of e as the value of v_i. This logic is the same as scalar clocks. When P_i sends a message to P_j, the current time v_i is sent along with the message as v_{msg}. Akin to scalar clocks, P_j updates its local time v_j as follows. It computes the max operation, and then increments the local component (j^{th} component) of v_j (refer to Equation 2.11).

$$\begin{aligned} &\forall k : v_j[k] = \max(v_j[k], v_{msg}[k]) \\ &v_j[j] = v_j[j] + 1 \end{aligned} \tag{2.11}$$

Equation 2.11 is similar to the corresponding equations for scalar timestamps. We just apply the *max* operation across each pair of corresponding elements across the two vectors, and then we increment $v_j[j]$. Figure 2.6 shows an example that uses vector time.

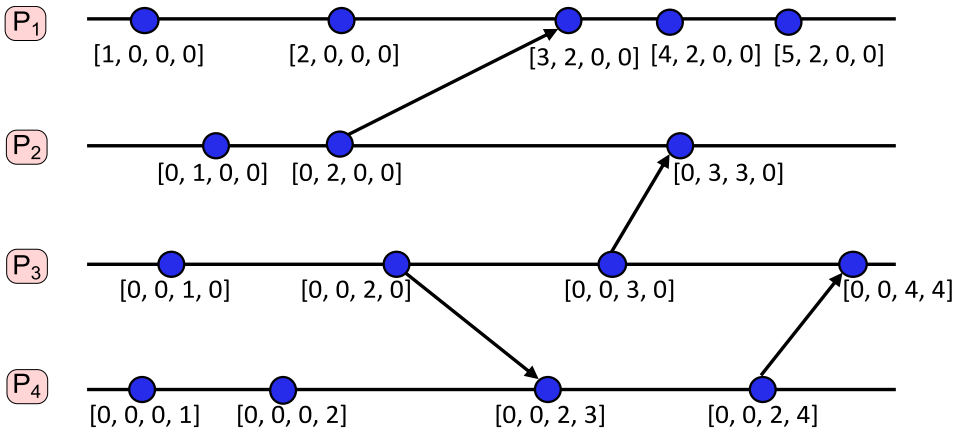


Figure 2.6: A set of four processes using vector clocks

To compare two vector time stamps, we use the following rule. Let v and w be two vectors of n dimensions each. We say that $v = w$ if for every $1 \leq i \leq n$, $v[i] = w[i]$. Next, we say that $v \leq w$ if for every $1 \leq i \leq n$, $v[i] \leq w[i]$. $v < w$ if $v \leq w$, and there exists an index i ($1 \leq i \leq n$) such that $v[i] < w[i]$. Finally, we say that $v \parallel w$ if neither $v < w$ nor $w < v$. As an example, the vector $[1, 3, 4]$ is less than the vector $[1, 5, 6]$.

The vectors $[2, 5, 3]$ and $[3, 4, 4]$ are concurrent. We can formally represent these relationships as follows.

$$\begin{aligned} v = w &\equiv \forall i : v[i] = w[i] \\ v \leq w &\equiv \forall i : v[i] \leq w[i] \\ v < w &\equiv (v \leq w) \wedge (\exists i : v[i] < w[i]) \\ v \parallel w &\equiv (v \not\leq w) \wedge (w \not\leq v) \end{aligned} \tag{2.12}$$

From the definition of vector time, it is obvious that the i^{th} element of the vector clock at process P_i, $v_i[i]$, denotes the number of events that have occurred at P_i until that instant. No other process can increment the i^{th} element. This means that if an event e has timestamp w, then $(w[j] - 1)$ denotes the number of events executed by process P_j that causally precede e.

Theorem 2.3.1

Consider events e_1 $(T(e_1) = v)$ and e_2 $(T(e_2) = w)$. If $T(e_1) < T(e_2)$ $(v < w)$, then e_1 must have happened before e_2.

Proof: If both the events are issued by the same process, then the statement of the theorem is trivially true. Hence, let us consider the general case when the events are executed by different processes.

Let e_1 be executed by Process P_i and e_2 by Process P_j. Consider $v[i]$ and $w[i]$. Given $v < w$, $v[i] \leq w[i]$. However, P_i is the only process that can increment the i^{th}. No other process can do it. Another process can only perform a max operation when a message is received. This is tantamount to choosing one of the values that already exist – a new value cannot be created in this process. Only incrementing it (by P_i) creates a new value. Hence, $v[i]$ has to be the latest value of the i^{th} element at the time of executing e_1. Recall that before executing e_1, $v[i]$ is incremented.

Let us now consider the point of time at which e_2 was executed by P_j at time w. We know that $w[i] \geq v[i]$. Given that no other process is allowed to increment the i^{th} element, the value of $v[i]$ must have been somehow communicated to P_j. This can only happen if P_i sent $v[i]$ (or a later value $> v[i]$) to another Process P_k and ultimately after a series of events and message transmissions the value reached P_j. $k = j$ is a special case of such an interaction. Using the max operation, the vector clock of P_j was updated to set its i^{th} element to a value greater than or equal to $v[i]$, and the value persisted. When e_2 was executed, w thus contained a value $w[i] \geq v[i]$. There is no other way in which the value of $v[i]$ could have reached P_j.

We thus need to have a chain of events from e_1 to e_2 of the form $e_1 \rightarrow \ldots \rightarrow e_2$. There is a causal interaction between successive events on the chain, which is established by either a message transmission or two events being executed by the same process. Therefore, there is a happens-before relationship between them. Given that such a relationship is transitive, the fact that $e_1 \rightarrow e_2$ naturally follows. ■

Optimized Implementation of Vector Time

In a vector clock, the vector used for representing time linearly increases with the number of nodes in the distributed system. This can lead to overheads, especially because the current time is attached to each message. There are some techniques to reduce the overheads of this process. One such technique (refer to Singhal and Kshemkalyani [Singhal and Kshemkalyani, 1992]) is as follows. It uses two additional arrays (per process).

The `lastUpdate` array stores the local time at which the current process i last updated its information about the time of another process j, and the `lastSent` array stores the time when process i last sent its time information to process j. With the help of these two arrays, we can optimize the vector clock algorithm. Let process i be P_i and let process j be P_j.

Consider a message that is being prepared to be sent to P_j from P_i. P_i need not piggyback the message with its full vector clock. It needs to only include those entries from its vector clock that belong to the following set S.

$$S = \{v_i[k] | \texttt{lastSent}[j] < \texttt{lastUpdate}[k]\} \tag{2.13}$$

However, these techniques are not generic and work only as good heuristics. We refer the reader to [Singhal and Kshemkalyani, 1992] for more details on these optimizations.

Other notable contributions in this space include the direct dependency work of Fowler and Zwaenepoel [Fowler and Zwaenepoel, 1990] and that of Jard and Jourdan [Jard and Jourdan, 1994]. In the direct dependency technique of Fowler and Zwaenepoel [Fowler and Zwaenepoel, 1990], processes maintain information pertaining to the direct dependence of events on other processes. The actual vector time of an event can be computed offline by using a recursive search on the recorded direct dependencies to account for transitive dependencies. Specifically, if an event e_j at a process P_j happens before the event e_i at another process P_i, it can be inferred that events e_1 to e_{j-1} at P_j also happen before event e_i. If e_j is the latest such event in

P_j, then we can perform the following optimization. It suffices to record at P_i that the latest event in P_j that has happened before e_i is e_j.

A recursive set of such dependencies can be created to obtain the vector clock of e_i when needed. One important step in implementing this approach is to make sure that every process updates the dependence information after receiving a message and before it sends out any messages. Additional details along with the recursive algorithm needed to identify the transitive dependencies can be found in reference [Fowler and Zwaenepoel, 1990]. Since this technique involves additional computation, its applicability is limited to specific settings such as causal breakpoints and asynchronous checkpoint recovery.

Beyond vector time, Wuu and Bernstein [Wuu and Bernstein, 1984] show use cases for matrix time. In this scheme, each process stores time as a two-dimensional matrix. For any process P_i, the $(i, j)^{th}$ entry in the time matrix represents the knowledge that P_i has about P_j's progress. At process P_i, the k^{th} row of the matrix ($k \neq i$) stands for the knowledge that P_i has about P_k. While being useful in some specific applications, matrix time is not meant for general use. The size of the timestamp is large and processing it at the receiver is time-consuming.

2.3.3 Limitations of Logical Time

The solution proposed by Lamport [Lamport, 1978] to circumvent using physical time in a distributed system and instead use logical time is elegant. However, logical time has its own shortcomings that impact its practicality. If one insists on inferring causality, then scalar clocks do not suffice, and one needs to use vector matrix clocks. These come with a huge overhead in terms of their impact on message sizes. Sadly, the overhead grows with the number of processes.

Logical clocks also assume that all interactions of the participants in the distributed system happen within the system, and there are no other communication channels. With the rapid rise of cyber-physical systems and Internet-of-Things (IoT), there is now an enhanced scope for systems to talk via multiple channels that may not use logical clocks.

From the preceding discussion, we note that existing solutions fall short of today's practical requirements. Using physical time comes with limitations, as we have discussed earlier. Logical clocks also have their drawbacks. They can only monotonically increase. Sometimes there is a need to correct the time by reducing it, especially if some messages need to be rolled back or there is an error in the system. Second, the network of processes needs

to be always connected and partition-tolerant.

2.4 Hybrid Clocks

Kulkarni et al. [Kulkarni et al., 2014] combine ideas from logical time and physical time to arrive at a simpler way of maintaining time in a distributed system. They refer to this as a Hybrid Logical Clock (HLC). The basic insight is as follows. The physical time is normally used to order events across nodes. This is very convenient and there is no need to store a vector clock. However, as we have seen earlier, we need expensive clock synchronization algorithms. In spite of that, there is a *window of uncertainty*. The key idea here is to use logical clocks only for resolving confusions within this window of uncertainty. Otherwise, the physical time can be used. In many ways, this solution combines the best of both worlds.

2.4.1 Overview

The requirements of an HLC can be captured formally as follows. We use the notation $\rho_{\mathtt{u}}(e)$ to denote the physical timestamp of event e at node $\mathtt{u}$. We shall slightly abuse the notation and write $\rho(e)$, when the node ID is clear from the context. For finding the current time on a given node, we invoke $\rho()$ without any arguments. Let $T(e)$ be the *hybrid* time assigned to event e.

- For any two events e_1 and e_2, $e_1 \rightarrow e_2 \Rightarrow T(e_1) < T(e_2)$.
- $T(e)$ requires $O(1)$ space, and all updates take $O(1)$ time. This means that we cannot have vector timestamps (one scalar per node)
- $|T(e) - \rho(e)|$ is bounded.

The above requirements can be understood in the following manner. The first requirement captures the notion of causality in distributed systems. The second requirement ensures that updates to $T(e)$ can be effected using $O(1)$ operations. Unlike vector clocks, the size of the timestamp does not grow with the number of nodes. The last requirement captures the constraint that $T(e)$ is close to $\rho(e)$, which enables the HLC to be used in lieu of the physical time.

The state variables maintained by each node $\mathtt{u}$ are as follows: $\phi_{\mathtt{u}}$ and $\mathtt{l}_{\mathtt{u}}$. The former approximates the physical clock and the latter is a logical clock. Specifically, $\phi_{\mathtt{u}}$ is the largest physical time seen by the current node.

The source of the physical time could be its own clock or the physical time received via a message sent from another node. Let t_e be the timestamp associated with event e. $T(e) = t_e = \langle p, l \rangle$. The individual components are referred to as $t_e.p$ and $t_e.l$, respectively. We say that $t_{e_1} < t_{e_2}$ if and only if the following conditions hold. Either $t_{e_1}.p < t_{e_2}.p$, or $t_{e_1}.p = t_{e_2}.p$ and $t_{e_1}.l < t_{e_2}.l$.

Let us define $|T(e) - \rho(e)|$ as $|t_e.p - \rho(e)|$. We wish to make the hybrid time and the physical time comparable. The method to compare them is to extract the p component from the hybrid time of event e ($t_e.p$) and find the difference with respect to the physical time of the event $\rho(e)$.

2.4.2 The Algorithm

Algorithm 2 HLC Algorithm: Sending a message or processing a local event

```
1:  procedure SEND(Node u)
2:      ϕ_tmp ← ϕ_u
3:      ϕ_u ← max(ϕ_tmp, ρ())
4:      if ϕ_u = ϕ_tmp then
5:          ▷ The physical clock was ahead. Increment the logical clock.
6:          l_u ← l_u + 1
7:      else
8:          l_u ← 0
9:      end if
10:     return ⟨ϕ_u, l_u⟩
11: end procedure
```

Algorithm 2 shows the algorithm that is run to update the time when a message is sent, or there is a local event. We assume that the current node $\mathtt{u}$ either is in the process of sending a message or executes some other local event.

The first task is to store the current value of the global state variable $\phi_\mathtt{u}$ in $\phi_\mathtt{tmp}$. Next, we compare this value with the current time (obtained using the function $\rho()$). If both are equal, then it means that we are within the window of uncertainty. In this case, all ties need to be resolved on the basis of the logical time. We thus increment $\mathtt{l_u}$. Otherwise, we are out of the window of uncertainty, hence $\mathtt{l_u}$ can be set to 0. The timestamp assigned to the event is equal to $\langle \phi_\mathtt{u}, \mathtt{l_u} \rangle$.

In Algorithm 3, we show the procedure that needs to be followed when a message is received. We start with setting the value of the physical timestamp ϕ_u to ϕ_{tmp}. Next we find the maximum of three quantities: ϕ_{tmp}, ϕ_{msg} (received from the sender) and the current time.

The first case is when all three are equal. Then, we are definitely within the window of uncertainty and the physical time is not very useful. In this case, it makes sense to increment the logical time the same way it is done for scalar clocks. We set l_u to $(\max(l_u, l_{msg}) + 1)$.

Next, let us consider the scenario where ϕ_{msg} is different and $\phi_u = \phi_{tmp}$. We have encountered a similar situation while sending a message. All that needs to be done is that the logical time l_u needs to be incremented.

The other case is when $\phi_u = \phi_{msg}$. Here also we do a simple scalar clock update. We just set l_u to $(l_{msg} + 1)$.

Finally, if none of these cases are true, then it is quite clear that the physical times (and their windows of uncertainty) do not overlap. There is no need to use logical times here. l_u can just be set to 0. Like the previous algorithm (Algorithm 2), in this case the timestamp that is assigned to the receive event is $\langle \phi_u, l_u \rangle$.

Algorithm 3 HLC Algorithm: Receiving a message

```
1:  procedure RECEIVE(Node u (receives a message), ⟨φ_msg, l_msg⟩)
2:      φ_tmp ← φ_u
3:      φ_u ← max(φ_tmp, φ_msg, ρ())
4:      if φ_u = φ_tmp = φ_msg then
5:          l_u ← max(l_u, l_msg) + 1
6:      else if φ_u = φ_tmp then
7:          l_u ← l_u + 1
8:      else if φ_u = φ_msg then
9:          l_u ← l_msg + 1
10:     else
11:         l_u ← 0
12:     end if
13:     return ⟨φ_u, l_u⟩
14: end procedure
```

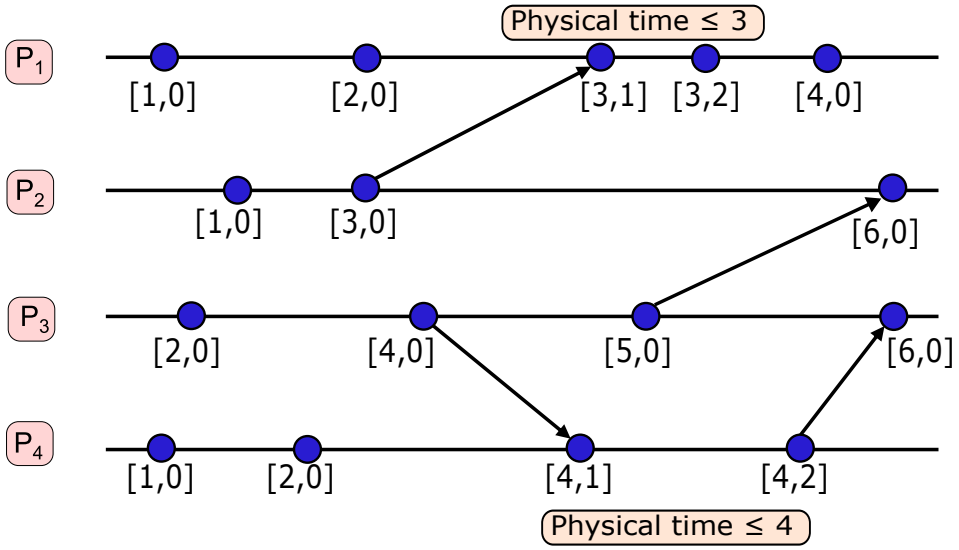


Figure 2.7: Usage of hybrid logical clocks

2.4.3 Proof of Correctness

Causality

The following theorems prove that the HLC construction satisfies the three conditions: causality, $O(1)$ time and space, and the proximity to the physical time.

Theorem 2.4.1

For any two events e_1 and e_2, if $e_1 \rightarrow e_2$ then $T(e_1) < T(e_2)$.

Proof: If e_1 and e_2 belong to the same process, then we can proceed as follows. If the physical times (or windows) are the same, then we increment the logical clock for e_2. Otherwise, t_{e_2} has a larger physical time. Given that we follow lexicographic ordering, regardless of the scenario, $T(e_1) < T(e_2)$.

If a message is sent from one node to another. We set the timestamp of the receiving node such that it is at least as large as the timestamp of sending node. Hence, in this case also $T(e_1) < T(e_2)$ holds.

These results can be applied transitively to prove the statement of the theorem. ■

O(1) Time and Space

We only store two values per timestamp: p and l. These can be represented as integers. We thus need O(1) space. Updating the time stamp is also very simple and requires O(1) comparisons. Hence, both are trivially true.

Accurate Approximation of the Physical Time

Let us now prove a series of results that show that the physical time of the event and its timestamp $t_e.p$ are close.

Lemma 1

For any event e, $t_e.p \geq \rho(e)$.

Proof: This follows from the *max* operation in both the algorithms. The physical time that is set in the timestamp is always greater than or equal to the current time ($\rho(e)$). ■

Lemma 2

Let e be an event executed by node $\mathtt{u}$. $t_e.p > \rho(e) \Rightarrow (\exists e' : e' \rightarrow e \wedge \rho(e') = t_e.p)$.

Proof: In this case, the timestamp of event e (t_e) has a physical clock timestamp that is more than the actual time ($\rho(e)$). This must have happened because the max operation yielded a larger physical time. This is possible only if the physical clock of the node has been set to a higher value because of a message receipt at some point of time. Given that every physical clock value that is set must have been produced by $\rho()$ (fetch current time) at some point of time, there must be some event e' that has produced this value. Given that the value has propagated to the timestamp of e, there needs to be a causal relationship between e' and e. The relationship $e' \rightarrow e$ must hold. Next, the time of execution of e' ($\rho(e')$) must be equal to $t_e.p$. ■

Let ϵ denote the physical clock synchronization uncertainty. Notice that ϵ depends on physical factors including the protocol used to synchronize the physical time. We now use Lemma 2 and the notion of the error ϵ to show that $|t_e.p - \rho(e)|$ is bounded for any event e.

Theorem 2.4.2

For any event e, $|t_e.p - \rho(e)| \leq \epsilon$.

Proof: Let us do a case-by-case analysis.

Case I: Assume $t_e.p = \rho(e)$. In this case, the current time is the latest. As a result, the inequality is trivially satisfied.

Case II: The only other case is $t_e.p \geq \rho(e)$ (as per Lemma 1). In this case, there is another event e' of the form $e' \rightarrow e$ and $\rho(e') = t_e.p$. This basically means that the recorded physical time in the past (for e') is greater than the recorded time in the present (for e). This is possible only because of a clock skew. From its definition, we have the following relationship.

$$\rho(e') - \rho(e) \leq \Delta$$
$$\Rightarrow t_e.p - \rho(e) \leq \Delta$$

If we set $\Delta = \epsilon$, the result naturally follows. ■

Hence, all the three conditions of the hybrid logical clock are satisfied by the algorithms (Algorithm 2 and 3).

The next question is to see if limits can be placed on how much the logical clock grows. Having a limit is essential because we do not want any overflows while incrementing the logical clock. The following corollary provides the answers.

Corollary 2.4.1

$t_e.l \leq |\{e' : e' \rightarrow e \wedge t_e.p = t'_e.p\}|$

Corollary 2.4.1 simply says that the value of the logical clock associated with event e is bounded by the number of events that have the same physical time and *happen before* it. This is easy to prove. Let the physical time on the node that is executing the current event be t. The logical clock is only incremented when the physical time of an event that has happened before the current event has the same physical time t. This naturally characterizes all the events that can lead to an increment in the logical clock, which is captured by the corollary.

The next question is whether we can place a limit on $t_e.l$. The following theorem answers it.

Theorem 2.4.3

$t_e.l \leq N \times (\epsilon + 1)$

The proof is left as an exercise for the reader.

2.5 Summary and Further Reading

Summary 2.5.1

1. Measuring time in distributed systems is difficult because the constituent nodes have independent clocks that may not be synchronized with each other.
2. The TEMPO protocol and its variants like the Cristian's algorithm are frequently used to synchronize physical clocks.
3. Here the main idea is to repeatedly send the local (physical) time to a server, and receive its response. If the number of samples is large, then it is possible to accurately estimate the clock skew (difference between the clocks of the local node and the remote server).
4. Any such clock synchronization algorithm needs to follow the Lundelius and Lynch bound (LL bound). It says that a system with N clocks cannot be synchronized with an error less than $(max - min)(1 - \frac{1}{N})$. max and min denote the maximum and minimum transmission times in the network (between the client and the server), respectively.
5. The Network Time Protocol (NTP) is commonly used to synchronize the clock across large networks. Most modern systems use them.
   (a) Stratum 0 comprises atomic clocks that are deemed to be the most accurate.
   (b) Stratum 1 clocks get their time from Stratum 0 clocks.
   (c) Devices in Strata 2 and 3 synchronize their time with clocks at higher levels.
   (d) NTP provides a coarse estimate of time. However, it cannot be used to run distributed algorithms that rely on a precise estimate of time.
6. GPS devices synchronize their time with multiple satellites. Given that there are four unknown variables – $\langle x, y, z \rangle$ coordinates and the clock skew – at least four satellites are required.

Given that there are unpredictable atmospheric delays, we need more satellites in practice for getting a precise time estimate.

7. Google TrueTime creates two kinds of master servers: GPS Time Masters rely on GPS and Armageddon Time Masters contain atomic clocks. TrueTime servers return a time interval as opposed to the current time. The absolute real time is guaranteed to lie within the interval.

8. Given the LL bound, exact clock synchronization is not possible. Hence, there is a need to rely on logical clocks.

9. Logical clocks can either be scalar or vector clocks. For two events e_1 and e_2, if $e_1 \rightarrow e_2$ then the following relationship holds: $T(e_1) < T(e_2)$.

   (a) The happens-before relationship $\rightarrow$ follows three rules. ① If e_1 and e_2 are executed by the same process and e_1 was executed first, then $e_1 \rightarrow e_2$. ② A send message event always happens before the corresponding receive message event. ③ The happens-before relationship is the transitive closure of rules ① and ②.

   (b) If $e_1 \rightarrow e_2$, then both are said to be causally related.

   (c) $T(e)$ returns the logical time for event e. Logical times are not always strictly comparable – they may not be totally ordered. If they are comparable, then the $<$ and $>$ relations are defined.

10. In scalar clocks, causality between events e_1 and e_2 based on comparing $T(e_1)$ and $T(e_2)$ cannot be inferred. They can be implemented by incrementing the clock for every local event and compute the maximum of clocks when a timestamped message is received.

11. Vector clocks allow us to infer causality when $T(e_1) < T(e_2)$. They require one element for each process in the distributed system. Here, timestamps are partially ordered, and all vector times are not mutually comparable.

12. To mitigate the space overheads of vector clocks, hybrid clocks have been proposed. Each hybrid clock produces two values

when asked to timestamp an event. One corresponds to an estimate of the physical time and the other is a scalar logical timestamp. The former is guaranteed to be always close to the real physical time of the event (based on the level of clock skews in the network). The size of the latter is bounded. It is dependent on the number of nodes in the network and the maximum clock skew. It has the same property as scalar logical clocks – causality between events leads to a total ordering of the timestamps.

Further Reading

One of the earliest references in this area is the paper by Ellingson and Kulpinski [Ellingson and Kulpinski, 1973] published way back in 1973. It was a pioneering work for its age. The authors many applications that are commonplace as of today (2024): navigation, airborne systems and systems with multiple sites. It was one of the first papers to propose the basic equations that GPS systems use. The next important development is Marzullo's thesis [Marzullo, 1984], TEMPO-based algorithms [Gusella and Zatti, 1983] and probabilistic clock synchronization [Cristian, 1989].

Lamport introduced the idea of logical clocks in his classical 1978 paper [Lamport, 1978]. Subsequently, the theory of vector clocks was introduced in the following papers: Fidge [Fidge, 1988], Mattern et al. [Mattern et al., 1988] and Schmuck [Schmuck, 1988]. Schmuck introduced a linearization function that converts a partial order of events into a total order. They prove that the general solution is undecidable.

Some modern implementations of vector clocks include the work related to creating efficient implementations [Singhal and Kshemkalyani, 1992], resettable clocks [Arora et al., 2000], resettable encoded vector clocks [Pozzetti and Kshemkalyani, 2020] and tree cks [Mathur et al., 2022].

Questions

Question* **2.1.** Prove that $t_e.l \leq N \times (\epsilon + 1)$.

Question** **2.2.** In the context of the Hybrid Logical Clocks, if the time for message transmission is so long that the physical clock of every node is incremented by at least d. Recompute the bound on $t_e.l$ for any event e.

Question* **2.3.** While using logical scalar and vector time, does the increment to the clocks d need to be common across processes in the distributed system? Justify your answer.

Question 2.4. Explain how we can achieve totally ordered multicasting with Lamport clocks? Assume FIFO channels. (HINT: try to modify the algorithm for totally ordered mutual exclusion)

Question** **2.5.** The space overhead of a vector clock's timestamp is $O(N)$, where there are N processes in the system. What is the space overhead of a hybrid clock's timestamp? Is it really $O(1)$?

Design Problems

Question 2.6. Set up a small experiment to simulate NTP in a LAN environment. You can scale the simulation to systems running across larger physical distances by introducing a delay in proportion to the physical distance (along with some noise).

Chapter 3

Distributed Data Storage

> *I have always imagined that paradise will be a kind of library.*

Jorge Luis Borges

Distributed systems typically comprise a large number of nodes. This means that they have a lot of storage space. This space can effectively be used to store a large amount of data, and also perform computations on it. Bespoke storage systems are expensive, difficult to maintain and difficult to scale. It is much easier to take regular machines and servers, connect them together and treat the ensemble as one large storage unit. Users or clients can be provided a single standardized interface to run their queries in this system. The data storage model in such large distributed systems is typically very simple. The storage space presents itself as a simple key-value store, where the *key* is the name of a file or object, and the *value* is the contents of the file/object. This is sufficient to store exabytes of data as modern web-scale systems do.

Fact 3.0.1

Modern web-scale systems do not use traditional storage mechanisms such as relational databases. They, instead, use large-scale distributed storage techniques where data is stored as a set of key-value pairs. This is a very simple, scalable and reliable method. It is possible to store exabytes of data using such a simple abstraction.

There are many ways to create such large distributed systems. We can use a large array of cheap servers with inexpensive storage devices, or we

can co-opt the machines of regular users. An example of the latter type of system is BitTorrent where the machines of regular users are used to store some data as well. Regardless of the architecture of the system, the fact is that storing a large amount of data in a distributed system is reasonably mature as of today and has proved to be extremely useful.

Such storage systems consist of numerous machines that play identical roles. They are referred to as "peers" and such distributed systems are consequently known as peer-to-peer (P2P) systems. P2P systems need not be purely distributed systems, they can have some degree of centralization. Over the last three decades, several generations of P2P systems have been proposed. There are at least three major generations and many believe that each major generation can be subdivided into many minor generations.

Broadly speaking, the first generation of P2P networks used some degree of centralization. There were some servers that were tasked to store index files. An *index file* maintains a mapping between file names and machines that store them. While looking up a file name, it is necessary to fetch the corresponding index file and then fetch the contents of the file. There are many problems with such centralized systems. The nodes that store the index files become a performance bottleneck, result in scalability issues and also become a legal liability (as was the case for Napster).

The first generation was quickly replaced by the second generation of P2P systems. These systems were true distributed systems and lacked centralized structures. The peers worked collaboratively to locate keys and disseminate information. We shall discuss two kinds of methods in this space. The first type of methods disseminate information using protocols that are akin to classical gossiping or epidemic propagation. The second type of methods assume an approach that is based on limited broadcast or multicast communication. Such systems had scalability issues, which necessitated the development of third-generation networks.

In third-generation P2P networks, the focus is on creating large scalable systems. They use a network with a structured topology. Note that regardless of the actual physical connections, it is possible to create any logical topology. This is readily achieved by creating a virtual network (network over a network). Such networks are known as *overlay networks*, which are critical to third-generation networks. Ring-based overlays are commonly used. Other popular topologies for overlay networks include the hypercube, the butterfly network and the de Bruijn network. They have nice structural properties, which can be used to design efficient key-value storage systems.

An important variant of such systems are highly secure networks such as Freenet. In such networks, the receiver does not know where a file is stored,

and the sender does not know who the receiver is. Such networks protect the identities of all the parties involved in the file transfer process. We shall see that such networks had to be created to circumvent legal issues.

Organization of this Chapter

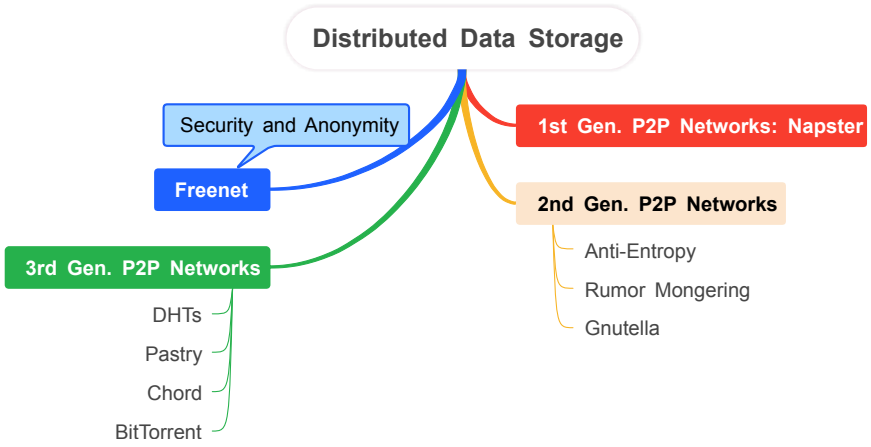


Figure 3.1: Organization of this chapter

Figure 3.1 shows the organization of this chapter. We will start with first-generation networks. With a short introduction to Napster, which is the most popular network in this space, we will move on to discussing second-generation networks. We will start our discussion with unstructured networks that rely on classical gossip or epidemic propagation methods. Then we shall discuss the popular Gnutella protocol. The next section discusses third-generation networks notably distributed hash tables (DHTs). They form the basis of most modern data storage networks. They scale to exabytes. We shall end our discussion with secure networks such as Freenet that hide the identities of the sender and the receiver.

3.1 First-Generation P2P Networks: Napster

Let us start our discussion with first generation P2P networks. We will discuss the most popular network in this category, the Napster system.

3.1.1 Napster

Let us go back to 1999. At that point of time, the MP3 file format had become very popular. The reason was that it suddenly became possible to store a 5-minute song using 5 MB of storage. The quality of the sound was very nice and clear. Your authors clearly remember that at that point of time, streaming videos or even storing videos in small pocket-size devices was unthinkable. However, storing audio had become possible, and thus MP3 players were very popular. They could fit in pocket-sized devices called MP3 players that especially found a lot of use while exercising. They became one of the most visible devices in gyms. Major record labels started distributing their songs in the MP3 format. Sites like `mp3.com` also sprung up. They enabled the seamless sharing of MP3 files.

At that point of time, the creators of Napster realized that it would be great if there was a search engine that was dedicated to MP3 files only. He envisioned a large distributed system that would allow users to exchange or trade MP3 files. Along with exchanging MP3 files, the engine could help users connect with each other and make friendships. This is how the idea of Napster came about. Think of it as a combination of a file sharing service and a social network. In terms of its implementation, Napster was a client application that was installed on a client machine that connected to a central server, known as the *broker*. Napster is now a well-recognized protocol. However, note that it has been replaced by newer technologies that solve the same problem more efficiently.

Basic Operation

All that a user has to do is open the Napster utility and login to a broker. Client machines that host users are expected to share the list of MP3 files that they store with the broker. Brokers collate the set of files obtained from different clients and create an index (or a directory).

The first action taken by the broker after a client logs in is to fetch the list of MP3 files that it has. After this, the client machine is free to search for an MP3 file on the broker. All that the user (on the client machine) has to do is enter a search term (query term). If the file is present on some other client machine, and it features in the broker's list, then the broker responds with the IP address of the remote client machine that contains the MP3 song. The broker provides a straightforward index or directory service. The files per se are transferred between client machines after getting the IP address from the broker. The requesting client needs to establish a

connection with the serving client machine, and get a copy of the song (MP3 file). This process is shown in Figure 3.2, where we can see multiple client machines and a broker. Note that the file transfer happens between the client machines directly without involving the broker.

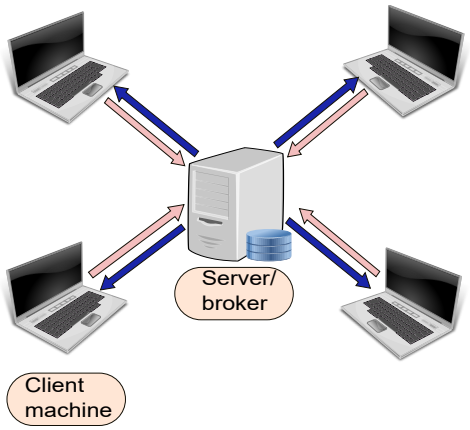


Figure 3.2: The Napster network

Napster Protocol

Let us now look at the Napster protocol. The broker runs on port numbers 7777, 8888 or 8875. The message format is quite simple. Any Napster message has three fields namely the length, type and data. The *length* field stores the length of the message in bytes. Two bytes are allocated for storing the length. There can be several types of Napster messages such as the error, login, login acknowledgement, version and upgrade messages. Finally, the *data field* corresponds to the actual payload – it represents the contents of the MP3 file.

In Napster, along with broker nodes, we have client machines and lookup servers. The client first finds the address of a broker by contacting a lookup server. The addresses of lookup servers are publicly known, whereas the addresses of brokers need to be provided by lookup servers. There is some opportunity for load balancing here. The lookup server can find the least-loaded broker. To use this mechanism, the client needs to exchange messages with the lookup server using the regular TCP/IP protocol. The broker can then be sent queries from the client.

Five Types of Napster Processes

A Napster client runs five kinds of processes in parallel. There is a main coordination process whose job is to connect and communicate with brokers. This is also known as the *main coordination instance*. The *listener* instance handles incoming connections from other clients. The *upload*, *download* and *push* instances are used to transfer files between client machines. They are also known as *peers*.

The state diagram of the main coordination instance is shown in Figure 3.3.

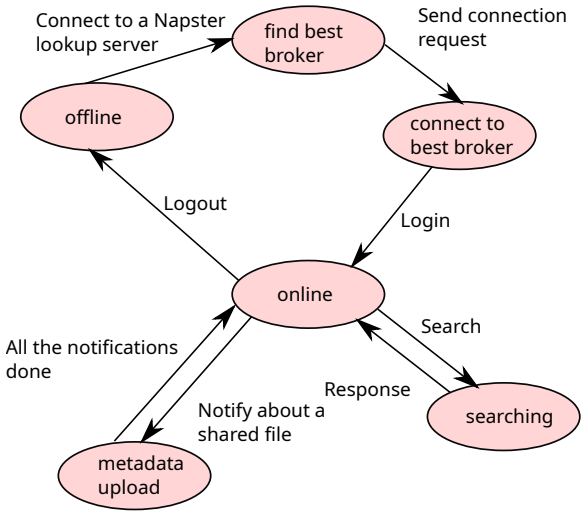


Figure 3.3: Actions of the coordination instance

Let us start from the offline state. The first action is to connect to a Napster lookup server. After the connection is set up, the state transitions to the find best broker state. The next task is to find the least-loaded broker. In this regard, the lookup server is free to use any kind of load balancing algorithm. Once a broker is found, the login process is activated and the state transitions to online (after successfully logging in). In this state, the client machine can send queries to the broker and receive responses. It can also send metadata to the broker. The metadata contains the list of files that the client has. Each row contains information such as the name of the file, the title of the song, names of the artists, etc.

The download instance starts from the download state. It then sends a download request to the broker and awaits a reply. There are many reasons

for the reply to be denied at this stage. The request may be illegal, the remote host (remote peer) may not be able to upload the file, or the file may simply not be there. If the file can be downloaded, then the broker sends the download ack message. This message contains the following: location of the file (hostname, IP address) and the TCP port.

There is an interesting twist here. Let us name the requesting client P_u and the remote client that has the file as P_v. In the normal sequence of events, P_u sends a request to P_v, or in other words P_u initiates a connection with P_v using the TCP port that the broker has provided to it. P_v can either accept or deny the connection. If it accepts the connection, it scrutinizes the request and if all is well, then it provides a copy of the requested file to P_u. All of this happens in the remote client download state.

However, the twist here is that P_v may be in a private network where all the network addresses are only locally visible. It is not possible to make a connection to P_v directly from outside the network. In this case, the TCP port that is returned by the broker is 0. To tackle this situation, P_u enters the remote client upload state. In this case, P_v needs to initiate the process of contacting P_u, and the broker needs to facilitate this. When P_v subsequently contacts the broker, it is asked to initiate contact with P_u. P_v may contact the broker if it needs to send or receive some information. P_v subsequently sends a connection request to P_u and once it is accepted, it sends the metadata and the contents of the MP3 file. The sequence of actions is shown in Figure 3.4

The last instance is the remote client download instance. It does not have the complexities of the earlier instance. It directly initiates a connection with a remote peer and downloads the file.

Legal Issues

Napster was for its time a revolutionary idea. It suddenly allowed users all over the world to freely access MP3 files. This was clearly illegal. Music is owned by its creator. It is clearly not meant to be freely distributed all over the world, that too without paying any money. It is true that Napster did start a revolution in internet-based content access, but it was clearly not on the right side of the law.

Moreover, its centralized architecture had problems. It was hard for clients to anonymize themselves. They had to make their identities visible to the server (central site). The central site had all the control, was aware of all the client's identities and was a single point of failure. For obvious reasons, a scalable distributed system should not rely on a central site.

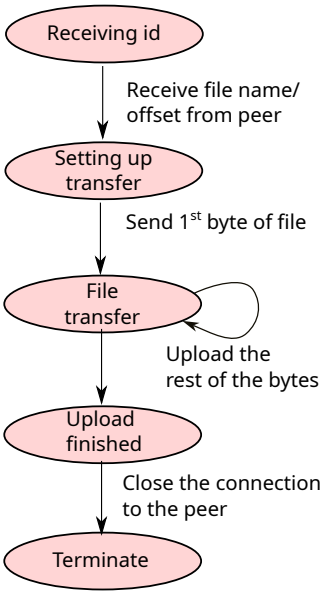


Figure 3.4: Remote client upload instance

Regardless of the technical shortcomings, the legal issues stood out and ultimately led to its downfall. By the late nineties, it was neck deep in its legal troubles and was sued by multiple music bands including the Recording Industry Association of America. It finally closed its service in 2001 and filed for bankruptcy in 2002.

Now, looking back at Napster in 2024, we get mixed feelings. The fact that it allowed free sharing of potentially copyrighted music and created financial losses for music creators is mostly true. However, there is also a school of thought that allowing free sharing of MP3 files on the internet actually boosted their sales. Regardless of which side of the argument we are on, we can unequivocally conclude that centralized file sharing has a lot of drawbacks. Projects such as Napster got phased out for techno-legal reasons.

This led to the second generation of P2P networks that did not have a central server. Instead, all the machines were *peers* – there was no designated client and server. They followed elaborate protocols to exchange information between them and provide the abstraction of a large data store, especially a hash table like key-value store.

3.2 Second-Generation P2P Networks

Second-generation peer-to-peer networks eliminated the central server. The nodes share information with each other as true peers. Algorithms in this space work on completely unstructured networks, where we assume that nodes can be arbitrarily connected to each other. Protocols rely on classic epidemic propagation models to disseminate information via gossip-based mechanisms (nodes exchange their contents). These approaches are effective and solve most of the problems in first-generation networks.

3.2.1 Communication in Unstructured Networks

A classical problem in unstructured networks is to simply broadcast a value to the rest of the nodes. This is a simple problem if we consider a structured network such as a ring or a tree. For example, in the case of a ring, all that we need to do is send a message on one side of the ring and wait for it to move a full circle and come back to the starting point. In the case of a tree, a node needs to simply forward the message to its children, and continue doing so recursively. However, in an unstructured network where there is no such predefined structure, sending a message with an updated value to the rest of the nodes is a difficult problem. We need to ensure that there is no exponential blowup in terms of the number of messages that are sent.

There are three generic approaches in this space.

Direct Mail In this case, the IDs of all the nodes in the network are known. For example, if the IDs are IP addresses, then all that needs to be done is to send messages individually or multicast them (wherever possible) to the rest of the nodes. This can degenerate into a sequential process. The node sending all these messages can become overloaded and become a point of contention. Nevertheless, this approach known as *direct mail* has its share of benefits especially when the network is small and all the nodes in the network are known to the sender.

Anti-Entropy In this case, a node need not have perfect knowledge of the network. It only needs to know the IDs of some other nodes. It can choose one of them uniformly at random and mutually synchronize its contents with that node or just send its updates to that node. For example, if a timestamped message needs to be sent and that message is not there in the log of the other node, then the message can be transferred to the log of the other node, and it can be instructed to

propagate the message further in the network. This is a very effective protocol and is still heavily used in commercial settings as of 2024.

Rumor Mongering The problem with the anti-entropy approach is that it is a protocol without brakes – it may keep going on without possibly terminating. Another way of broadcasting an update is called *rumor mongering*, where a node sends updates to other nodes and when it observes that most of its neighborhood is already *infected* (has the update), it reduces its aggressiveness in sending the update and ultimately stops. This protocol ensures that the overall process can ultimately come to a stop, and by that time we can guarantee that at least most of the nodes have received the update. This is an example of a complex epidemic, which is arguably faster than the anti-entropy protocol. Sadly, it does not guarantee that the update will reach all the nodes. It is indeed possible that we terminate the protocol before all the nodes have received a copy of the update.

Anti-entropy and rumor mongering are examples of gossip-based protocols. Nodes "gossip" with other nodes and exchange their contents in different ways. This information propagates like an *epidemic*, hence such protocols are classically modeled on the lines of classical epidemic propagation.

3.2.2 Anti-Entropy

Let us model the process of information dissemination using the anti-entropy protocol in this section. Consider a system with n nodes. The modeling proceeds in roughly the same manner as we model epidemic propagation. Propagating an update in an unstructured network is quite similar to an epidemic propagating in the human population. There are *infected* nodes whose job is to propagate the update, there are *susceptible* nodes that have not received the update up till now and then there are *removed nodes* that are not involved in the protocol at all. We assume that the protocol proceeds in rounds. In each round, all the infected nodes send updates to one or more nodes (typically 1).

Mathematical Modeling

There are three different kinds of information exchange in an anti-entropy protocol (refer to [Demers et al., 1987]). They are known as *push*, *pull* and *push-pull*, respectively. In the push mode, the update is simply sent to a

target node, and if the target already has the update, then it discards the message. On the other hand, in the pull mode, a node proactively requests other nodes to send it any updates that they may have. It simply adds all the new updates that it receives to its local log. The push-pull interaction does both. It *synchronizes* the contents of two nodes – it ensures that they have the same set of updates after the push-pull operation. Here we are assuming that every update has a timestamp associated with it, and there is a notion of time such that we can achieve a total order of all the updates. Even if we rely purely on local time, the same can still be achieved using Lamport clocks as we have seen in Chapter 2. Let us thus make our life easy and assume that a notion of time exists.

Let us now explain the basics of epidemic theory and provide a mathematical foundation for understanding anti-entropy protocols.

Pull-based Algorithm

Consider a pull-based algorithm. Let p_i be the probability of a node remaining susceptible (not having the update) after i rounds. In the next $((i+1)^{th})$ round, it will not remain susceptible if it is contacted by another node that has the update. If we assume that any node can contact the current node in the next round, then we can find the probability of a node continuing to remain susceptible (refer to Equation 3.1).

$$p_{i+1} = p_i^2 \tag{3.1}$$

A node remains susceptible if it is already susceptible in the i^{th} round and is contacted by a susceptible node in the $(i+1)^{th}$ round. The result is the product of the two probabilities: both are equal to p_i. Being contacted by a susceptible node means that either a message was sent without any updates or no message was sent (both the scenarios are equivalent here).

This is a rapidly declining function. We have $p_{i+k} = p_i^{2^k}$, which approaches 0 very quickly if p_i is a small value.

Push-based Algorithm

Let us now look at push-based methods. The expected number of infected nodes after i rounds of information exchange is $n(1-p_i)$. The probability of not contacting a node is $(1-1/n)$. Hence, the probability of not contacting any infected node in the $(i+1)^{th}$ round given that a node is still susceptible in the i^{th} round is given by the following equation.

$$p_{i+1} = p_i \left(1 - \frac{1}{n}\right)^{(n(1-p_i))} \tag{3.2}$$

Note that we first multiply by p_i because the node needs to be susceptible at the end of the i^{th} round. This is multiplied with the conditional probability – probability that it is still susceptible in the next round. This can happen if it does not contact any infected node.

We use the fact that $(1 - 1/n)^n$ tends to e^{-1} as $n \to \infty$. Thus, for large n and small p_i (low probability of being susceptible), Equation 3.2 can be simplified. The result is shown in Equation 3.3.

$$p_{i+1} = \frac{p_i}{e} \tag{3.3}$$

We should not forget that Equation 3.3 only holds when p_i is a small value, which happens towards the end of the protocol.

Combination of Push and Pull Algorithms

Let us look at the structure of the two results – $p_{i+1} = p_i^2$ for pull-based algorithms and $p_{i+1} = \frac{p_i}{e}$ for push-based algorithms. Both hold towards the end of the dissemination process. It is clear that for low values of p_i – when there are very few susceptible nodes left – a pull-based strategy is more effective. Consider, the stage of dissemination where $p_i \leq \frac{1}{e}$. At this point of time $p_i^2 \leq \frac{p_i}{e}$, which means that pull is better than push. This is because nodes that are susceptible can find infected nodes very easily. This also aligns with common sense. When we are nearing the end of the protocol (last few rounds), a push-based strategy is unlikely to yield a lot of dividends because we don't know which nodes haven't received the update yet. Sadly, everything is left to random chance. However, susceptible nodes shall always find it very easy to locate a node that has the update and fetch the update in a pull-based strategy. This is precisely why a pull-based strategy works towards the end of the protocol when there are few susceptible nodes left.

Let us now understand what happens at the beginning (when the anti-entropy protocol starts). When the information dissemination is starting, a pull-based strategy is unlikely to work because all the nodes that get contacted will most likely be susceptible themselves. On the other hand, a push-based strategy is guaranteed to disseminate the update far better. The number of infected nodes will exponentially increase because the probability of propagating the update is high. The chances of meeting another infected node are very low. This will happen till a certain point. After that, a

situation will arise when most of the nodes that are contacted will already have the update (are infected). It is then a better idea to switch to a pull-based method at that stage.

It is clear that at the beginning, the number of infected nodes will get multiplied by a factor of two every round. Whenever we multiply a quantity with a constant in every round, it grows exponentially. On similar lines, we can say that the number of susceptible nodes will reduce at an exponential rate in the starting phase. Gradually, diminishing returns will set in and the rate of growth will slow down. At this point of time, we can switch to a pull-based mechanism.

Proving that we actually need $O(\log(n))$ rounds is slightly more complicated, even though it intuitively seems to be so. Hence, it is out of the scope of this book. The reader can look up the following references [Özkasap et al., 2010a, Pittel, 1987] for a deeper treatment of this subject.

3.2.3 Rumor Mongering

Let us now consider rumor mongering, which is a complex epidemic. The information dissemination process gradually slows down and ultimately stops. In this case, we have three kinds of nodes: *infected*, *susceptible* and *removed*. The first two classes of nodes retain the same meaning as that in the anti-entropy protocol. The removed nodes no longer participate in the information dissemination protocol. Even if they were infected earlier, they are not counted as being infected after being removed. Moreover, only infected nodes can be removed. For modeling such processes, typically differential equations are used (as we shall see next).

Remark 3.2.1

There are three kinds of nodes in a class rumor-mongering setting: *infected*, *susceptible* and *removed*. The removed nodes no longer participate in the information dissemination protocol. Even if they were infected earlier, they are not counted as being infected after being removed. Finally, note that only infected nodes can be removed.

Mathematical Model

Let s be the fraction of nodes that are susceptible, i be the fraction of nodes that are infected and r be the fraction of nodes that are removed. Clearly, $s + r + i = 1$.

The rate of decrease of susceptible nodes is proportional to the number of susceptible nodes itself and the number of infected nodes as well (refer to Equation 3.4). This follows from simplistic arguments because if we have a lot of susceptible nodes, then their count will decrease very quickly primarily because it will be easier for an infected node to contact them. On the other hand, if we have a lot of infected nodes, then they will be able to contact more susceptible nodes in a round. This will reflect in the rate of decrease of susceptible nodes. Kindly note that this is an approximate equation, because we are not taking into the consideration the fact that it is possible that different infected nodes may end up contacting the same susceptible node, or they may end up contacting another node that is infected. Nevertheless, approximate equations help us understand the broad structure of the solution. The trends become clear.

$$\frac{ds}{dt} = -si \tag{3.4}$$

These equations hold better when there is a large number of nodes, and we are away from the extremes (all susceptible or most of them are infected).

The other aspect of rumor mongering is that nodes gradually lose interest in propagating the update. This is a good idea. Otherwise, the algorithm will continue forever and like anti-entropy a lot of messages will get sent that actually reach infected nodes, and thus do not serve a very useful purpose. However, the flip side of this is that the protocol may terminate too soon, and we may miss infecting a lot of nodes. Equation 3.5 shows the loss of interest in propagating updates. Note the negative term $1/k \cdot (1-s)i$.

$$\frac{di}{dt} = si - \frac{1}{k} \cdot (1-s)i \tag{3.5}$$

The first term si indicates the rate of growth of the number of infected nodes. It is proportional to the number of already infected nodes and the number of susceptible nodes. The latter term – the number of susceptible nodes – is an approximation and basically serves to indicate that as the number of susceptible nodes increases, it is easier to infect more nodes in a round. We need to add another term that artificially damps this process – it can be delinked from the actual physical phenomena of update (epidemic) propagation as it is an artificial term, which is being extraneously added. Specifically, Equation 3.5 uses a damping factor k (higher it is, lower the degree of damping). In this case, the rate of damping is proportional to the number of infected nodes, which basically means that as the number of

infected nodes increases, the need to damp the process also increases proportionally. This means that the need for aggressive transmission needs to reduce. It is also proportional to the number of nodes that are not susceptible – they are either infected or they are removed. The aim is to explicitly factor in the number of nodes that are not susceptible anymore and use that as a separate variable for computing the degree of damping. The idea here is to take account of nodes that have been removed from the update/epidemic propagation process. It is important to note that the damping factor is something that we are adding to deliberately slow down the propagation process – it thus cannot be fully explained using the mathematics of epidemics.

We now have two equations here that need to be solved – Equations3.4 and 3.5. The solution is shown in Equation 3.6.

$$i(s) = \frac{k+1}{k}(1-s) + \frac{1}{k} \cdot \log(s) \tag{3.6}$$

Here, we are expressing the fraction of infected nodes i as a function of the fraction of susceptible nodes s. This is a complex function and there is a need to plot it to understand it. For different values of the damping factor k, the results are shown in Figure 3.5.

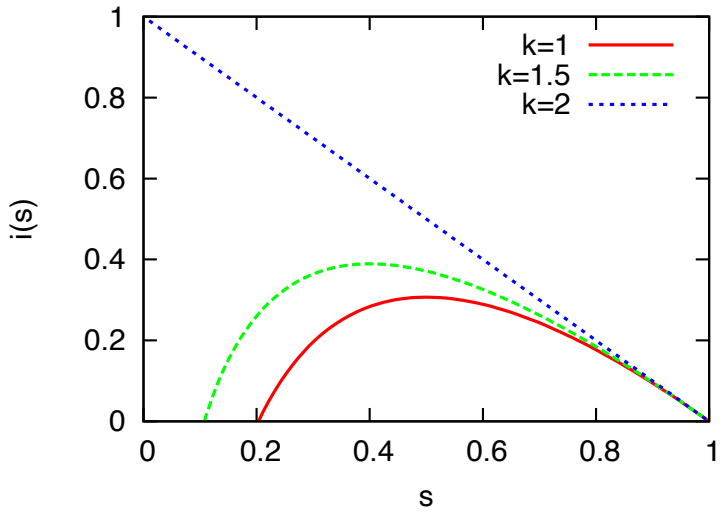


Figure 3.5: i as a function of s

Consider the case of $k = 2$ (lightly damped). We reach a point when no

nodes are susceptible anymore – all the nodes are infected. We can assume that the plot starts from the point where no node is infected and all the nodes are susceptible (rightmost point on the x-axis). In the case of $k = 2$, the number of infected nodes keeps on increasing roughly linearly until all the loads become infected.

This is however not the case with systems that are more damped, $k = 1.5$ or $k = 1$. In these cases, a situation indeed does arise when no node is infected, yet there are some susceptible nodes still left. This means that because of the high degree of damping, the process terminates prematurely. Due to that some nodes are left out – the update message does not reach them. Given that lower the value of k, higher the degree of damping, we find that the *residue* (susceptible nodes left out) is higher in the case of $k = 1$ as compared to when $k = 1.5$. This is on expected lines. We can see that as we move from $s = 1$ to $s = 0$, the number of infected nodes keeps increasing till a certain point. After that because of damping, this number starts to reduce and ultimately becomes zero. Due to the premature nature of the termination, the protocol leaves back a residue.

Observation 3.2.1

Lower the value of k, higher the residue. This is because as we decrease k, we increase the level of damping. Inevitably some nodes are left out because we remove too many infected nodes from the system.

The Residue

It is thus important to understand that there is a trade-off between the number of messages that is sent and the *residue* that is left. We would ideally want the residue to be zero. However, we typically never have such an accurate understanding of the system. Hence, there is always a little bit of under-damping or over-damping, which basically means that either we send many more messages than what is required, or we leave a residue (resp.).

Let us now try to compute the residue. It is easily observed that when $i(s) = 0$, s is equal to the residue. It is given by the following equation.

$$s = e^{-\frac{k+1}{1-s}} \tag{3.7}$$

Equation 3.7 is not a very convenient equation to solve. However, we can arrive at some general insights from the form of the equation. For example, we can observe that the residue is roughly an inverse exponential function

of k. This basically means that as we increase k, we expect a substantial reduction in the residue.

Let us now study a few more fundamental results in this space. Let us try to find a relationship between the average traffic and the value of the residue. Assume that there are m messages sent in the system per node, and there are n nodes. Then, for every update we send $n \times m$ messages. The probability that a given node misses all of these messages is $(1 - 1/n)^{nm}$, which as $n \to \infty$ becomes e^{-m}. In other words, the fraction of nodes that we expect to form the residue will also be the same, i.e., e^{-m}. This can be proven with a simple probabilistic argument starting from the basic notion of expectations.

Let us now understand this result. If m messages are sent (per node), and no message is received by a few nodes, then we expect the fraction of such nodes to be e^{-m}. For instance, if $m = 1$, then we expect roughly 37% of the nodes to miss the update. If $m = 2$, we expect 13.5% of the nodes to miss the update. For $m = 3$, this number becomes 5% (so on and so forth). m is clearly related to k – they are two different ways of quantifying the message throughput per node. There is clearly an inverse exponential relationship between the per-node traffic and the residue.

Given that rumor mongering can miss nodes, there is often a need to augment it with anti-entropy protocols such that at least all the nodes receive the update. Hence, it is often necessary to run the anti-entropy protocol towards the end of the rumor mongering phase such that any of the nodes that are left out can get the update in logarithmic time.

Deletion of Entries

Most data storage networks treat a deletion of an entry as an *update*. This provides a simple and consistent interface for managing all the operations in a data distribution and storage network. A *death certificate* is thus issued to remove an entry from the data network. Such death certificates need to be propagated to the entire network using either anti-entropy or rumor mongering. If the death certificate meets the original update (request that added the entry in the first place) at some node, then the update needs to be discarded for obvious reasons.

The death certificates themselves create a separate problem. They can continue to propagate and remain in the network. The problem of unceasing propagation can be solved easily using rumor mongering. Nevertheless, the death certificates need to be kept in the network until all copies of the corresponding entry are removed. This means that we need to define a time

threshold. Beyond that death certificates need to be discarded. It is possible that the entry may not be completely removed from the network.

In fact, it is possible for this process to go out of control. Many data dissemination algorithms start anti-entropy protocols when they realize that there are a lot of susceptible nodes left. This is fine if the update is not propagating quickly. However, if the network is trying to remove the entry by issuing death certificates, then this is a false signal. Unbeknownst to the nodes that are busy deleting the entry, other nodes are actually overreacting and trying to create more copies of the entry !!! There is thus a need to accurately detect this situation and also keep a few death certificates at some nodes known as *retention sites*. These certificates can be activated and propagated if we detect the propagation of updates that should have been removed from the system long back.

It is possible that a later update is issued to update the same entry. Consider a key-value store. If a given value is deleted for a key, then a death certificate needs to be issued, and it needs to be duly timestamped. If a later update is issued, i.e., a new value is created for the same key, then it becomes the most recent update. It can thus override all death certificates and get them discarded.

Spatial Distribution

We have up till now assumed that any node can contact any other node at will. This means that a node pretty much has knowledge of a large number of nodes in the network. This may or may not be true in practical situations. However, if it can indeed reach all the nodes and decide to *exchange* contents with any node at random, then we can guarantee that within $O(\log(n))$ rounds the update (epidemic) can be propagated.

However, let us now turn our attention to more practical scenarios. In this case, nodes have knowledge of their neighbors or neighbors' neighbors. In fact, we can model this situation mathematically as follows. Let the probability of contacting a node that is d hops away be proportional to d^{-a}. Here, a is a constant. Depending upon the value of a, it is possible to work out the average traffic per network link and an estimate of the time that it takes to propagate the update to all the nodes (epidemic dissemination time). In fact, we can derive reasonably accurate expressions for the average traffic per link (see [Demers et al., 1987]); however, finding the update dissemination time is not that straightforward. It is sometimes inversely proportional to the message traffic per link but in most cases this proves to be a very crude estimate.

3.2.4 Gnutella

Gnutella is a distributed network that does not rely on a central server. This is one of the key advantages of Gnutella over Napster. Owing to the fact that there is no central server, the onus of joining a network and even helping another node locate a file falls on peer nodes. This is more work, but it is a more scalable solution, and it does not have many of the legal issues that plagued Napster. A client node joins the network by contacting a known Gnutella host that is already there in the network.

The client can then send file search messages to its neighbors in the network. They in turn forward the message to their neighbors, so on and so forth. There is a possibility of an exponential blowup in terms of the number of sent messages. Hence, this process needs to slow down and ultimately be terminated (similar to rumor mongering). This is why a time-to-live field (TTL) is appended to every message. As the message is propagated through the network, this field gets decremented (once per hop) and ultimately when this field becomes zero, the node stops propagating the message. Before that point is reached, if a file server (a peer node in Gnutella) is able to locate the file in its file system, then it sends its IP address to the client. The client machine then directly establishes a connection with the node that has a copy of the file and starts a download process.

Akin to Napster, a Gnutella node runs numerous processes (known as instances) that handle different aspects of the protocol. There is a connection handler that manages connections with other Gnutella peers, a coordination instance that manages the sharing and discovery of files, a download and upload instance, respectively. The latter two perform a similar task as their counterparts in the Napster protocol. The download instance handles the download from a remote node and the upload instance handles the upload to a requesting client node.

Requests in the Protocol

Initially, the connection handler is in the offline state. Whenever there is a desire to connect to the network, it opens a coordination instance and sends a ⟨CONNECT⟩ message to another Gnutella peer. When the connection request is accepted, the state changes to online. Now at this stage, the client may try to contact other Gnutella nodes and also try to find the size of the network. This process of querying the Gnutella nodes and increasing the list of peer nodes is carried out in the ping state. The ping state is used to initialize the network state of the node that has recently joined the Gnutella

network. Subsequently, the node can easily search for files by contacting live Gnutella nodes in its list of internal nodes. For searching a file, it enters the search state.

As we have already discussed, each message has a TTL field. This ensures that the message is ultimately removed from the network and there is no flooding. Messages thus do not live in the network forever.

With regard to transferring files, Gnutella faces the same problem as Napster – the nodes are behind firewalls or are in a private network (NAT). In this case, initiating contact with such nodes is not possible.

Let us consider three different cases: the server (remote peer with the file) and client (peer requesting the file) are both behind firewalls, the client is behind a firewall and the server is not, and the server is behind a firewall, but the client is not. In the first case, it is not possible to establish a file transfer connection unless both of them are connected to an intermediary server at exactly the same time via their coordination instances. This is difficult and somewhat improbable, and thus the protocol does not consider this case.

If the client is behind a firewall, then it is reasonably easy. The client can establish a connection with the server after it receives its IP address from the node that responded to the search reply. The client can then simply download the file. In this case it is like creating a connection with a regular website.

In the last case, if the server is behind a firewall, but the client is not, then the client needs to request the server to establish a connection with it. If the server is connected to some intermediary node via a coordination instance, then a message needs to be passed to it via that connection. The message should indicate that the server needs to initiate a connection with the client and transfer a given file. This is known as the ⟨`push`⟩ message. Given that the client is not behind a firewall, a connection can easily be established with it and the requested file can be uploaded.

The Coordination Instance

The coordination instance can receive five types of messages. We have already seen a few of them earlier. Let us now list down all of them.

The first is a ⟨`ping`⟩ message. After receiving this message, the receiver decrements the TTL field, and forwards the message to other remote peer nodes. This message is primarily used for network discovery. When a node joins a network, it sends this message over the Gnutella network such that it can discover more of the network and find more peer nodes. This will be

helpful when it is trying to search for files because it can send more search messages to its peer nodes.

The reply to a ⟨`ping`⟩ message is a ⟨`pong`⟩ message. This message is sent back to the node that just joined the network. This process basically works like a bidirectional handshake. The remote node also makes a note of the newly joined node. Given that all the nodes are peers of each other in the Gnutella network, it is also helpful for the remote node to be aware of the existence of the newly joined node.

Next, let us consider the ⟨`search`⟩ message. This message is sent with the details of the file (mostly audio files and videos). A remote node searches for the file locally in its database. Then it returns a copy of the file using the ⟨`search result`⟩ message. Otherwise, it forwards the message down the network after decrementing the TTL field.

As the name suggests, the ⟨`search result`⟩ message indicates that a given peer node has a copy of the file. It is important to note that at this point of time, only the existence of the copy of the file is noted, and this information is sent to the requesting node. However, the node that has a copy of the file does not send the copy directly to the requesting node. This is primarily because the requesting node may have gotten positive search replies from several nodes. It cannot possibly accept copies of the file from all of them. This would be very inefficient, hence, the ⟨`search result`⟩ message is a very short message that simply has the identity of the remote peer node that can possibly supply a copy of the file. Subsequently, the requesting client node is expected to start a connection with one of the nodes that has replied in the affirmative and download a copy of the requested file.

Note that in the latest official documentation [Forum, 2024] of Gnutella, the message names ⟨`search`⟩ and ⟨`search result`⟩, have been replaced with ⟨`Query`⟩ and ⟨`QueryHit`⟩, respectively.

Next, let us consider the ⟨`push`⟩ message, which is relevant only when the remote node that shall provide the file is behind a firewall. In this case, the requesting client sends the ⟨`push`⟩ message to the remote file-serving node via any open connections with other nodes that it may have. The remote node subsequently initiates a connection with the client node by sending a ⟨`giv`⟩ message. This allows the client to request for files. The request may contain the original file that it was searching for or possibly other files that the serving node is willing to share. Once, this list of files is shared, the remote node starts to upload the files to the client node using its upload instance.

At this point, the following doubt may arise. How do we send a message to a node that is behind a firewall that cannot accept incoming connections?

The only way, as we have discussed, is to send a message to a node that has an open connection with the node behind the firewall. It can pass on the message via that open connection. A logical follow-up question is, "How do we find a node with an open connection with a node behind a firewall?" This can easily be ensured by following these rules while transmitting messages.

- A ⟨`pong`⟩ message follows the same path as the original ⟨`ping`⟩ message (reverse order of hops).
- A ⟨`search result`⟩ message follows the same path as the original ⟨`search`⟩ message (reverse order of hops).
- A ⟨`push`⟩ message follows the same path as the original ⟨`search result`⟩ message (reverse order of hops).

These rules ensure that if connections are kept alive for a slightly additional duration, then the reply message (⟨`pong`⟩,⟨`search result`⟩ and ⟨`push`⟩) can be sent back via a chain of open connections. By following this method, it is possible to reach nodes that are behind a firewall.

Observation 3.2.2

An important point that has emerged out of the previous discussion is that it is often necessary for the reply to follow the same path as the original message. This ensures that bidirectional communication is possible between the requesting client machine and the remote server that has a copy of the file.

Comparison with Napster

Given that both the protocols, Napster and Gnutella, are reasonably simple and more or less achieve the same objective, we are in a good position to compare them. Napster has its share of problems in the sense that it is centralized, and it is possible to fix the legal liability. Napster nevertheless has some basic load balancing capabilities. There are ways of finding the least-loaded broker and ensure that no single server becomes a point of contention. Gnutella, on other hand, is more naturally load balanced. Even if there are network partitions, the protocol is still reasonably resilient. This is because as long as a node can at least contact some partition, it can get access to many files that are popular. So in a certain sense, the service still remains available. However, the quality may be *degraded* because the files in the inaccessible partitions will not be available for download.

In terms of traffic, both Napster and Gnutella do not do a very good job. Napster, of course, has more of centralization. Even if multiple brokers are used, the brokers can nevertheless become a point of contention. Furthermore, they also can be targets of denial-of-service (DoS) attacks. In a DoS attack, a malicious attacker can keep sending dummy requests and choke the network bandwidth or cause a server to crash.

Gnutella is more scalable and more immune to DoS attacks. However, a major concern with its operation stems from the fact that a large number of ⟨`ping`⟩ and ⟨`pong`⟩ messages are sent to discover the network. The same is true for ⟨`search`⟩ messages as well. Recall that they are sent throughout the network to search for a file. The reason for this is that Gnutella does not maintain a centralized directory. Hence, there is a need to send a lot of messages. The reason for not having a centralized directory is to avoid uncomfortable legal consequences. The price to pay is the large number of messages sent for network discovery and file search.

There is an issue related to the search quality as well in Gnutella. It is better in Napster mainly because there is a fixed set of brokers that can be linked to each other, and they are guaranteed to have an entry for a file if it is there in the network. Of course, there are legal issues here, but at least we have one single central repository of files stored in one place. Gnutella does not have such a repository. This avoids the obvious pitfalls of centralization – enhanced legal liability, single point of contention, reduced reliability and susceptibility to DoS attacks.

It, instead, relies on a self-limiting form of message-broadcast to find nodes and files in the network. The TTL *limits* the number of messages. Nevertheless, the overall number of messages sent is much higher than Napster. There is another aspect to this other than the obvious problem of high message overheads – it can cause Gnutella to miss a lot of files. This reduces its accuracy, which more advanced protocols need to rectify.

3.3 Distributed Hash Tables

Creating an efficient distributed storage solution is a classical problem in distributed systems. In a large distributed system such as Facebook, Amazon or Netflix there are millions of data items (search keys) and thousands of requests per second for retrieving petabytes of data. The data can comprise personal pages, products for sale or movies. There is thus a need for a very high-throughput search mechanism that can handle such *big data* applications: millions of keys, millions of values, where each "value" can be

a large object ranging in size from megabytes to gigabytes. There is thus a need to design a distributed data structure that can store all this data across thousands of nodes in a distributed system. The reason for using a distributed system is clear. All this data will not fit in one node or even a loosely-connected cluster of nodes. Hence, there is a need to distribute the data and the corresponding access requests across a large number of distributed nodes.

There are many advantages of doing so. This approach automatically provides high throughput and scalability. We can also provision for some built-in redundancy where a given key-value pair is distributed across nodes. This strategy allows us to provide high fault tolerance, resilience and all-round availability.

Definition 3.3.1 Hash Table

A hash table is a key-value store. Given a key, a lookup operation can quickly find if it exists or not in the key-value store. If the key exists, it returns the corresponding value. Hash table operations such as INSERT and LOOKUP roughly take $\theta(1)$ time as long as they are lightly loaded. To eliminate hash collisions, the common methods are linear probing and chaining.

Any key-value system is at its core a hash table. In this case, we are looking at a large and highly complex distributed hash table, which in popular parlance is known as a DHT. Let us quickly recapitulate classical hash tables. Then we shall move on to discussing different types of DHTs.

3.3.1 Classical Hash Tables

A classical hash table is a data structure that stores $\langle key, value \rangle$ pairs. Given a key, we can quickly find its corresponding value. The idea is to first convert the key to a uniformly varying random value, and then find the index where it is *most likely* stored in an array of values (A). For instance, if we are storing the names of a list of students and their corresponding heights, we can use a hash table. The key, which is a student's name, needs to get first *hashed* to an integer.

Assume that the array of values has `S` entries. We hash the key k to compute a uniformly varying integer $\mathcal{H}(k)$. The index in the array where we search for the value is $l = \mathcal{H}(k)\%S$. To start with, we can check if $A[l]$ is empty or not. If it is empty, we can conclude that no value exists for the key in the hash table (refer to Figure 3.6).

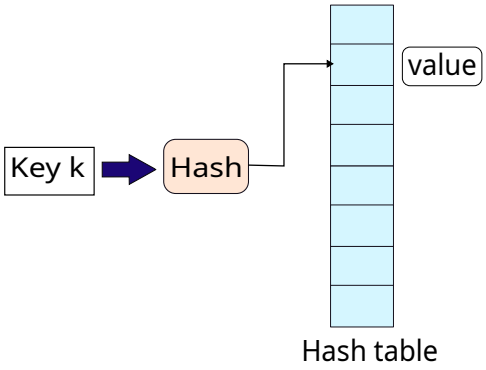


Figure 3.6: A generic hash table

However, if $A[l]$ is not empty, then we have a problem on our hands. The value that is present could correspond to key k or some other key k' that maps to the same array index l. This problem is known as *aliasing*, which can be mathematically represented as follows (% represents the modulo(remainder) operator):

$$\mathcal{H}(k)\%S = \mathcal{H}(k')\%S \tag{3.8}$$

If there is no aliasing, then a hash table is unarguably a very efficient data structure. We can fetch the value for a key or determine that the value does not exist in $\theta(1)$ time. However, if we have aliasing, then there is a problem. We need to first ascertain if the value at the location corresponds to a given key. This means that each cell in the array needs to store the value of the key as well. This does solve the correctness aspect of aliasing. However, there are going to be performance issues. We may be slightly unlucky. It may be the case that some array locations may have a lot of contention. As a result, keys will constantly be displacing each other. This will harm performance. We thus need a better solution to manage such contention. There are two popular options available to us: chaining and linear probing.

Chaining

Chaining is a simple solution where each entry of the array is a linked list of $\langle key, value \rangle$ pairs. All the keys map to the same array entry in this case. For searching for a given key, we need to iterate through the list stored at its corresponding array entry. In the worst case, the search operation can take

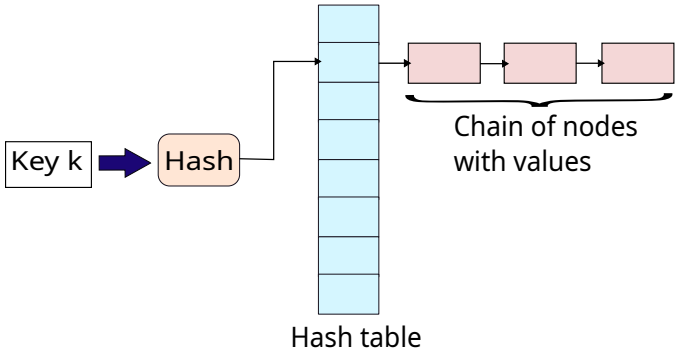


Figure 3.7: A hash table with chaining

$O(n)$ time, where n is the total number of values in the hash table. This is because in the worst case, all the keys can map to the same array entry. This will not happen in practice because we use a uniform hashing algorithm that ensures that keys map to all the entries roughly uniformly. We can also limit the size of the linked list if we are allowed to either discard entries or deny key addition requests to the hash table. If the maximum length of any linked list is α, then we will require at the most $\theta(\alpha)$ steps.

Linear Probing

Other than complicating the data structure, we can adopt a simpler method. If the value is not available at location l, then we search the array locations $l+1$, $l+2$, and so on. Basically, we can search the next κ entries ($l \ldots (l+\kappa-1)\%S$). If an entry is found, then we can return the value, or we can return a null value (indicating that the key is not present in the hash table). Recall that in the chaining algorithm, we could store more keys than the number of array entries (`S`). This is not possible here. The worst case complexity with this method is $O(\kappa)$. As a thumb rule, when the array is *sparse* (far fewer keys than the number of array entries), this method should be chosen.

Extension to Distributed Hash Tables (DHTs)

Consider a distributed system. We wish to use it as a hash table that stores (key-value) pairs. The keys and values can be stored on different nodes. Insofar as an external user is concerned, it should not matter on which physical node a key or its corresponding value are stored. The interface should look the same as a regular hash table. We would like the entire

system to seamlessly scale with the number of keys and number of nodes. Moreover, this method allows us to dynamically add/delete nodes at run time and also enhance fault tolerance by storing a key-value pair on multiple nodes. We can thus adjust to high- and low-demand scenarios. Note that two problems need to be solved here. Given a key, we need to first find the IP address of the physical node that stores it. Next, we need to search for the key-value pair within the node.

Definition 3.3.2 DHT

A distributed hash table or a DHT is a distributed version of a hash table. It primarily answers two queries: *get* (the value corresponding to a key) and *put* (insert a key-value pair). It is a highly scalable and robust data structure.

3.3.2 Pastry

Overview of Pastry

Let us now discuss one of the simplest DHTs, Pastry [Rowstron and Druschel, 2001]. We start out by computing the hash of every node's IP address (or some other unique identifier). Any *uniform hashing function* ensures that the distribution of hashes is uniform. We create a 128-bit hash out of each node's unique information. We can safely assume that we will not have any aliasing. The probability of aliasing in any case is very low: 2^{-128}.

There are two ways in which we will interpret this 128-bit number. First, we will represent it in base 16. This means that any 128-bit number can be represented as a sequence of thirty two 4-bit (hexadecimal) numbers. This specific aspect will be dealt with later. The more immediate aspect of this numbering is that if we have a large number of nodes (order of thousands), we can arrange all the 128-bit numbers (hashes of node IDs) in a circular ring in ascending order of numbers. This circular arrangement of nodes can be thought of as a virtual network or an *overlay*. An overlay is a network over a network where the nodes are logically connected to each other such that we can achieve a specific purpose. In this case, we are superimposing a circular overlay (ring) such that we can use it to implement a DHT.

The way that the circular overlay enters the picture is as follows. We also compute a hash of the key that we intend to search or add. Given the hash of the key and circular overlay of (node id) hashes, we need to map the key to a node on the ring. The key is mapped to the node whose hash value

(proxy for the *node id*) is *closest* to the hash of the key. Henceforth, let us not prefix the term "hash" before node IDs and keys, and slightly abuse the notation in favor of simplicity. We shall henceforth use the term *nodeId* or *node id* to refer to the hash of a node's unique information/location (e.g. IP address) and use the term *key* to refer to the hash of the key.

Definition 3.3.3 Overlay

An overlay is a virtual network. Every node is logically connected to a few of the other nodes to realize a structured topology. This is independent of the actual physical connections.

The overview of the *key search* process is as follows. We start with a node that is close to the node that is serving the request. A client machine is supposed to contact a server in the DHT, which initiates the process to search for the key. The server that receives the request is a regular peer node. It starts out by comparing the length of the shared prefix between the key and its *nodeId*. A shared prefix is computed by matching bytes in both the hexadecimal representations; we start from the most significant position. The node forwards the request to another node that has one more matching hexadecimal digit in the shared prefix. This way, the request gets closer and closer to the node whose *nodeId* is closest to the key (final destination). Ultimately, this process converges, and the request reaches the destination node. If it is a search request, the destination node searches for the key, otherwise if it is an *add* or *delete* operation, then the node executes the corresponding operation.

In the aforementioned discussion, an important point has come up – we need a routing table with each node that has lists of nodes with different levels of node ID similarity. This routing table will be used to navigate to a node that has one more matching prefix digit. Hence, with every node, we maintain a *routing table* that has a list of node lists. Each entry stores a node ID and some additional node-related information.

We shall feel the need for a few additional data structures later. The first is a neighborhood set (of nodes) that captures the physical proximity of nodes with respect to the current node. This data structure is required to create/initialize the routing table. The second data structure is a *leaf set* that contains the nodes with the closest k *nodeId*s. Here, k is the size of the leaf set. Towards the end of the search process, it is important to iterate through the leaf set because the routing table ceases to be useful.

Remark 3.3.1

A node in Pastry has three data structures.

1. It contains a routing table, which maintains a list of node lists (akin to a 2D table). It helps a node identify another node that has one more matching prefix digit.
2. It maintains a neighborhood set that contains the nodes that are *physically* closest.
3. Every node has a leaf set that contains the set of nodes that are closest to it in terms of their node IDs.

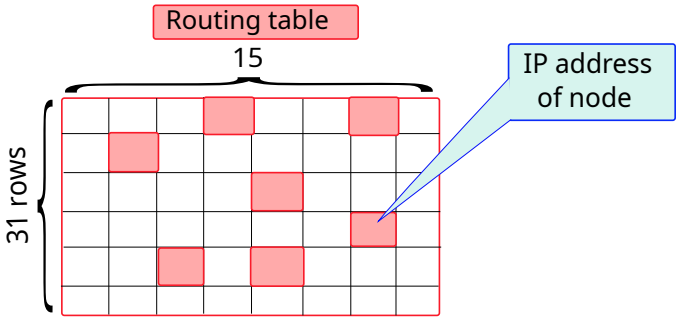


Figure 3.8: Pastry's routing table

Routing in Pastry

Figure 3.8 shows the structure of the routing table in Pastry. We have 32 rows and 15 columns. Each entry stores the IP address of a node. As indicated earlier, we represent a 128-bit ID as a sequence of hexadecimal digits – each digit is 4 bits (base-16 hexadecimal number). Given that we have 128 bits in an id, we can represent it using 32 hexadecimal digits.

The algorithm in a nutshell is as follows. We try to match the key with the node ID starting from one end. Assume without loss of generality that we start from the most significant digit (the choice does not matter). Let us number the rows of the routing table from 0 to 31. If the length of the matching prefix is i, it means that the first i digits match and the $(i+1)^{th}$ digit does not match. For example, if the length of the matching prefix is 0, the first digits differ. Assume that the length of the prefix match is l. The

aim is to increase it to $l+1$. The way to do this is to access the l^{th} row of the routing table. Given that the $(l+1)^{th}$ digit differs, it has 15 possible choices. Each column of the routing table stands for one such choice. Assume the $(l+1)^{th}$ digit is 0x9, we access the cell in the column that corresponds to 0x9 in the $(l+1)^{th}$ row. This cell is expected to store the address of a node that has a prefix match of at least $(l+1)$. We can then navigate to that node and continue to increase the length of the prefix match. If the cell is empty, then we need to do something else, which we shall address later.

In principle, this is a very straightforward design. As we can see all that we need to do s store a 32×15 matrix. The process of routing is also very simple. We start with the first digit of the key and the first digit of the ID of the current node. If they match, then well and good, we proceed to the second digit. Assume that they don't match. Then we access the first row of the routing table and navigate to the column that corresponds to the first hexadecimal digit of the key. Let this digit be d. We access the column corresponding to digit d. We expect to see a node's location stored there, whose first digit is d.

Next, the protocol moves to a new node, where we are sure that at least the first digit has matched. Now, we look at the second digit. The protocol in this case is also similar. If the second digit is equal to d', then we access the second row and go to the column that corresponds to the digit d'. If we find a node there (in the corresponding table cell), then we visit that node, and access the third row and so on. Gradually, we will match more and more digits, and the key will get closer and closer to the current node's id. Hence, as we observe, ultimately we will get very close to the node that is actually responsible for storing the key's value. Note that in Pastry, our goal is to search for the node whose ID is closest to the key. By following this process, we are essentially increasing the length of the prefix match between the key and the node ID by one in every routing step.

Up till now, we have been assuming the best case, which is that we always find a node at the cell that we are interested in. However, it is possible that the routing table may not be aware of any node that has a certain digit at that specific position. In this case, we need to do something else because clearly increasing the length of the shared prefix is not one of the options that we have. The other problem is that there is no clear-cut termination clause defined. We do not know if we need to keep searching, or we need to ultimately declare a node as the final result. It is the node that is supposed to contain the key.

This is where we need two additional data structures namely the *leaf set* and the *neighborhood set*. The leaf set $\mathcal{L}$ contains L nodes. Out of these L

nodes, $L/2$ nodes have a smaller node ID as compared to the current node's id. They are the closest node IDs arranged in ascending order. Similarly, the other half of the leaf set contains the closest $L/2$ nodes with larger IDs (sorted in ascending order again). The protocol is so designed that the leaf set is accurate all the time. This basically means that at all points of time, we are aware of the closest L nodes that are the closest in terms of their IDs.

The neighborhood set $\mathcal{M}$ contains M nodes that are the closest in terms of geographical proximity. We will see that this information is also important particularly during a lookup operation because we don't want messages to travel very far. Geographical proximity ensures that message latencies are within limits. An important point needs to be made here. In any distributed system, we are not really concerned with the number of operations that happen within a node – they are deemed to be very fast. Instead, we look at the message complexity: the messages that are sent to remote nodes. Sending a message is often a very slow operation and determines the critical path. This is why, we have a strong interest in ensuring that all the nodes that we visit are relatively close by.

Given that we have defined the leaf set and the neighborhood set, we are in a position to handle the two special cases that we discussed. We shall refer to Algorithm 4 in the subsequent discussion.

Progress Guarantees

The first question is what happens when we don't find an entry in the routing table. In this case, we are not in a position to increase the length of the shared prefix by one. However, we can still get closer to the destination node (the node that should contain the key). This is because of the leaf set. Assume that the key is greater than the node id. The leaf set is guaranteed to have $L/2$ nodes that have greater IDs than the current node id. This means that we can always find a node within the leaf set, whose ID is closer to the key than the current node. Let us consider the node that has the closest such distance between its ID and the key and forward the current request to that node. Then, in this case, even though we are not increasing the length of the shared prefix, we are still reducing the distance (between the key and the node id). There is *forward progress*.

Again at that node, we will resume the process of routing. If we can increase the length of the shared prefix, well and good, otherwise we can continue to decrease the distance with the key by using the leaf set. Having a full (or near full) routing table is always the better solution because we can make rapid progress. However, in case the routing table has blank cells, we

can still make progress as long as the leaf set is maintained correctly. Note that as nodes enter and leave the system, the leaf set has to be updated continuously. Hence, the protocol will never get stuck, and we will always be able to reach the destination node.

Remark 3.3.2

In Pastry, forward progress is always possible. In every step, we get closer to the destination node.

Ensuring Proximity in Accesses

This is where the neighborhood set comes handy. Recall that it is a set of M nodes that are closest to the current node in terms of geographical proximity. We can for instance quantify proximity as the latency of sending a message (ping time for example). When a node needs to join the network, it contacts another node T that is close to it. Node T helps construct the routing table for the original node. It populates its routing table with all the nodes in its neighborhood set to start with. It then asks its neighbors for suggestions for blank cells in its routing table. Given that the new node (to be added) is close to T, it is also "close" to all the neighbors of T.

We can now apply some mathematical induction here. Given that the routing table of node T was constructed in exactly the same manner, it is expected to have pointers to nodes in its routing table that are *close to it*. Now, note that nodes from the routing table of T and its neighbors are being used to populate entries in the routing table of the new node. Hence, by induction and the fact that proximity is a transitive property, we can argue that all the routing table entries in the new node are relatively close it. This is how a degree of *locality* (proximity) is maintained. Such a strategy ensures that the latency of the protocol is reduced because messages are always sent between nodes that have some degree of proximity to each other.

A Look at the Formal Algorithm

Algorithm 4 shows the pseudocode for routing. The idea is to find the node that contains the value for key `K` (let this represent the hash of the key).

We first check if the key is within the range of the leaf set. If it is within the range, then we forward the key to the node whose ID is the closest to the key (Line 3). This is the termination phase of the algorithm. This means that we have found the node that needs to contain the key. Note that this operation relies on the accuracy of the leaf set. Often in a distributed

Algorithm 4 Pastry's routing algorithm

1: **procedure** ROUTE(Hash of the key (K), Routing table $\mathcal{R}$, Leaf set $\mathcal{L}$, Neighborhood set $\mathcal{M}$)
2: **if** $\mathcal{L}_{-\mathtt{L}/2} \leq \mathtt{K} \leq \mathcal{L}_{\mathtt{L}/2}$ **then** ▷ K is within the range of the leaf set
3: Compute $\mathtt{L_i}$ such that $\mid \mathtt{L_i} - \mathtt{K} \mid$ is the least
4: **if** $\mathtt{L_i} = \mathtt{nodeId}$ **then**
5: **return** $\mathtt{L_i}$
6: **else**
7: Forward K to $\mathtt{L_i}$
8: **end if**
9: **else** ▷ Search in the routing table
10: $\mathtt{l} \leftarrow$ common_prefix(K, nodeId)
11: **if** $\mathcal{R}(\mathtt{l}+1, \mathtt{K}_{\mathtt{l}+1}) \neq \mathtt{null}$ **then**
12: Forward to $\mathcal{R}(\mathtt{l}+1, \mathtt{K}_{\mathtt{l}+1})$
13: **else**
14: **if** $\exists \mathtt{u} \in \mathcal{L} \cup \mathcal{M}$ such that prefix $(\mathtt{u}, \mathtt{K}) \geq \mathtt{l}$ **then**
15: Forward to $\mathtt{v} \in \mathcal{L} \cup \mathcal{M}$ s.t. prefix$(\mathtt{v}, \mathtt{K}) \geq \mathtt{l}$ and $|v - K|$
16: is minimized.
17: **else**
18: Forward to $\mathtt{w} \in \mathcal{L} \cup \mathcal{M} \cup \mathcal{R}$ s.t. $\mid \mathtt{w} - \mathtt{K} \mid$ is minimized
19: **end if**
20: **end if**
21: **end if**
22: **end procedure**

system, we have many concurrent actions. As a result, it is seldom possible to make very precise guarantees owing to race conditions (concurrent events that many negatively influence each other).

Let us now come to the *else* part of the algorithm. We try to increase the length of the shared prefix by one. If we find a corresponding entry in the routing table, then we route the request to that node (as specified in the routing table's cell). Refer to Line 12. Note that the term $\texttt{K}_{\texttt{l}+1}$ refers to the $(\texttt{l}+1)^{\texttt{th}}$ digit in the hash of the key, and `nodeId` refers to the current node's id.

Assume that this is not the case (the corresponding entry in the routing table is empty). Then we still need to ensure forward progress. We create a combined set comprising all the nodes in the leaf set and neighborhood set. We first try to see if we can increase the length of the prefix match (Line 16). If we can find a few such nodes, then we minimize the distance between the key and the node's id.

Otherwise, we find the node whose ID is the closest to the key (see Line 18). Recall from our earlier discussion on forward progress guarantees that it is always possible to find such a node because of the way the leaf set is constructed. This means that we can always move to a new node where the distance between its ID and `K` is lower than the corresponding distance for the current node. This guarantees forward progress, albeit slowly.

A Note about Complexity

So, what is the complexity of the overall routing process? One may want to trivially argue that it is $O(1)$ because we can at most look at 32 digits (assuming we always find an entry in the routing table). However, it turns out that things are not that simple because we have a multitude of cases. Also, it is not necessary to use the routing table at every step, particularly when we do not find entries in the routing table. Let us analyze this mathematically.

Let us start with a question. What is the probability that a hash does not have its first m digits (from the MSB) in common with any other node's hash? Let us represent this probability as $P(m, n)$, where n is the number of nodes in the system. Let us write a quick list of points that will take us towards the solution. We will put a restriction on m here. We will assume that it is slightly more than $log_{16}(n)$ (details follow).

1. The probability that two hashes have their first m digits in common is 16^{-m}.

2. The probability that there is at least one mismatch is $(1 - 16^{-m})$.

3. The probability that the key does not have a prefix match of length m with all the nodes is $(1-16^{-m})^n$. Assume there are n nodes.

4. Let $m = c + log_{16}(n)$.

5. It follows that:

$$\begin{aligned} P(m,n) = (1-16^{-m})^n &= \left(1-16^{-c-log_{16}(n)}\right)^n \\ &= (1-16^{-c}/n)^n \qquad (\lambda = 16^{-c}) \\ &= \left((1-\lambda/n)^{n/\lambda}\right)^{\lambda} \\ &= e^{-\lambda} \qquad (n \to \infty) \\ &= e^{-16^{-c}} \end{aligned} \tag{3.9}$$

Let us take a deeper look at the expression $\boxed{\text{m = c + log}_{\cdot}16(\text{n})}$. c is the excess amount over and above $log_{16}(n)$. It is a constant that is added to the log term. Figure 3.9 shows the relationship between $P(m,n)$ and the constant c.

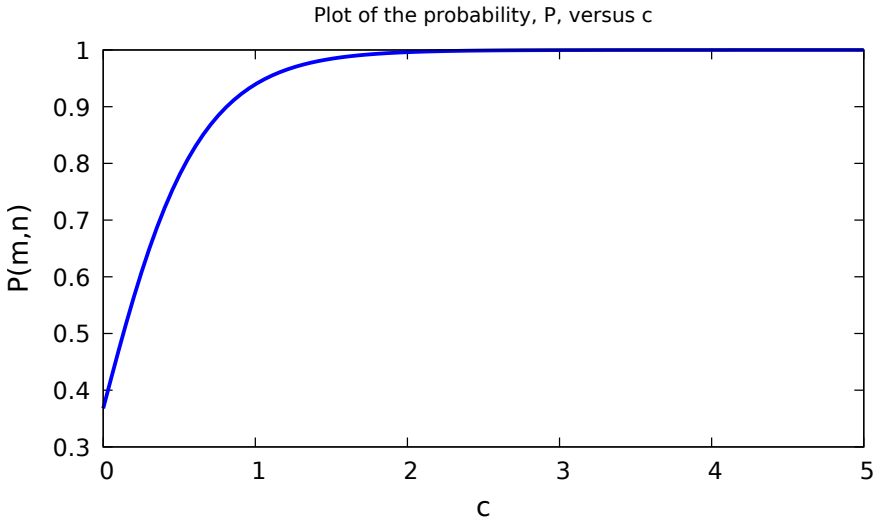


Figure 3.9: $P(m,n)$ versus c

We observe that $P(m,n)$ quickly saturates to 1 as c exceeds 3. This basically means that the maximum size of the prefix match that we can

expect in a real setting is roughly $log_{16}(n)$. Beyond this number, the probability of not finding a match becomes near zero ($\because P(m,n) \rightarrow 1$). This further means that the number of messages that we need to send because of matching entries in the routing table is limited to $log(n)$. Note that we have eliminated the base 16 here, because it is not relevant to our discussion anymore.

Does this mean that the complexity (read message complexity) of Pastry is $O(log(n))$? Well, not yet. We have not considered the corner cases. A skeptic can always argue that the routing table will mostly be found to be empty, and as a result, the request will be sent to either a leaf node or a neighbor node (members of the leaf set and the neighborhood set, respectively). Let us now argue that this will not happen frequently enough to change the overall complexity from $O(log(n))$ to something else, subject to the fact that the routing table is well constructed (details follow).

We need to make some assumptions about the quality of the routing table. We need a broadcast mechanism to propagate node arrival information to all the interested nodes. Let us define an *interested node* as a node that has a long prefix match with the new node. In this case, a prefix match of $log_{16}(n) + 3$ is long enough. It is longer than what is going to arise from random chance. Intuitively, if a long enough match exists between node IDs, then they know about each other. It turns out that as long as this can be guaranteed, we can guarantee completion using $O(log(n))$ messages.

In other words, if we can ensure that using a combination of mechanisms the routing table information is up-to-date in the sense that we just defined, and it is never the case that we find an entry to be empty, when it actually should contain a pointer to another node, then we can guarantee that we indeed complete the routing using $O(log(n))$ steps. It is possible for theoreticians to look at the degree of deviation from this ideal behavior and assess its potential to increase the overall complexity beyond $O(log(n))$. Let us however not proceed in this direction given the complexity of such methods. They are out of the scope of this book. We simplistically assume that using a combination of protocols, the routing tables contain up-to-date information. Then we can safely say that routing in Pastry has a logarithmic complexity.

Node Arrival

The advantage of DHTs is that they can handle dynamic node arrival and departure. This allows them to automatically scale with the demand. For instance, if we have deployed a DHT on an e-commerce site, at the time

of festivals, we can add more nodes. In the lean season, we can take nodes out of the DHT, and use them for some other purpose. In fact, this is why many companies that own large server farms start a cloud business on the side such that they can make good use of their servers when they are not required.

Node addition in Pastry is quite similar to the key lookup process. The node X that needs to be added computes the hash $\mathcal{H}$ of its node ID and contacts a node S that is already there in the DHT. Assume that there is a directory of nodes in the DHT that can be looked up. We can always choose a node S that is geographically the closest to X based on the information that is present in this directory. Another method of achieving the same is to use expanding ring multicast, where we multicast a message with increasing time-to-live values. If any node is there in the DHT, it replies in the affirmative. This way we can quickly find the node S that is the closest in a geographical sense to X. The ping time can be a proxy for geographical distance here.

Initializing the State of Node X

Let the node that $\mathcal{H}(\texttt{X})$ is closest to be E. S (one that the hash of X is the closest to) needs to run the regular lookup process using $\mathcal{H}(\texttt{X})$ as the key. It finds a sequence of nodes from S to E. We now have a lot of information with us. We have access to leaf sets, neighborhood sets and routing tables of all of these nodes. All of this information can be used to initialize the state of node X. The neighborhood set of X can be initialized as the neighborhood set of S. Given that S and X are close by, this is the right thing to do. Similarly, the leaf set of E (the final node) can be used to initialize the leaf set of X. For the routing table, we can scan through all the entries in the leaf sets, neighborhood sets and routing tables of all the nodes from S to E and add them to the routing table of X.

Informing Other Nodes about X

Initializing, the state of X is not enough. We need to let the other nodes know about the existence of X. This means X needs to be added to the neighborhood set of S that initiated the node addition process on its behalf. Recall that S is geographically close to X. It can be possibly added to the neighborhood sets of other neighbors of S. It definitely should be there in the leaf set of E, and possibly in the leaf sets of some of the nodes that are in the leaf set of E. This means that the information about X needs to be broadcasted by E to all the nodes in its leaf set such that they can modify their leaf sets and routing tables accordingly. Finally, all the nodes in the path from S to E also need to modify their routing tables to include X.

Redistribution of Keys

The last step is the redistribution of keys. By definition, a key is mapped to a node that its hash is the closest to. Now, assume that `X` is placed between nodes `E` and `E'` on the ring. With `X` coming in, it is possible that some keys stored in `E` and `E'` need to be moved to `X` along with their values. This redistribution of $\langle key, value \rangle$ pairs needs to be done.

Finally, after updating all this state, and redistributing keys, we will consider our work to be done. If you look at it, even though the overall complexity is $O(log(n))$, we still need to send a fair number of messages and move a lot of data around.

Node Deletion

There are two reasons why we would want a node to be deleted from the ring. The first is that it just voluntarily decides to leave on its own. The second is that it actually fails and other nodes in its leaf set or neighborhood set discover this fact. In both the cases, the node needs to be removed.

The Leaf Set

A node is supposed to periodically ping all the nodes in its leaf set and find if they are alive or not. If a node is found to be failed, there is a need to repair the leaf set. This is not very hard to do. Let us say that a node with an ID greater than the current node's ID is found to have failed. Then, we can just contact any other node whose ID is more than the current node's ID and fetch its leaf set. We can then use the information present in its leaf set to find additional nodes that can possibly be added to the current node's leaf set (based on the proximity of hashes). Assume the current node's ID is i. Assume that the node that failed had ID $i + j$. We can then choose another node in the same direction (clockwise or anticlockwise) that has ID $i + k$ and read its leaf set. We can then get the ID of a node in its leaf set that we need to substitute the failed node with.

The Neighborhood Set

On similar lines, we can assume that nodes are periodically pinging nodes in their neighborhood sets. If a failure is suspected, then we can do something similar as the previous case (nodes in the leaf set). The current node can ask other nodes in its neighborhood set to send their neighborhood sets to it. It can then find a node that should be added based on proximity information. We should add the closest node that is not there in the set.

The Routing Tables

Given that a node has failed, a lot of routing tables will need to be repaired.

The routing process also needs to be modified to take failed nodes into account.

In general, it is a good idea to have multiple entries in each cell in the routing table. This provides a degree of robustness. If a node fails, we can always look up other nodes. Moreover, if there is network congestion, we can forward the lookup message to an alternative node that can be accessed quickly. However, we may not be able to create so much of redundancy for all the cells in the routing table all the time. We need to consider the possibility of having a single entry in each cell, which we were assuming to be the case up till now. In this case, a node will contact the failed node, and it will not get a response. This case is no different from seeing an empty cell in the routing table. Recall that we have an elaborate protocol for handling such cases.

The other issue is removing the failed nodes from routing tables. A node `X` that finds node `Y` to have failed can remove it from its routing table. It can additionally try to do a lookup for `Y` and start a sequence of messages. Each message needs to be annotated with the information that node `Y` needs to be removed. This way, we can clean up some of the routing tables – at least in nodes that are close to the failed node.

Nodes can also proactively contact nodes in their routing tables and check if they are alive or not. If they are not alive, then a routing table cleanup procedure (as described in this section) needs to be commenced.

3.3.3 Chord

Let us now discuss another simple DHT algorithm that has had a lot of impact. As the name suggests, Chord [Stoica et al., 2003] is related to the geometrical definition of a chord, which is defined as a line segment that connects two points on the circumference of a circle. Akin to Pastry, it also uses a hashing algorithm to map nodes to different positions on a ring-shaped overlay. It has inherent advantages in terms of simplicity and robustness as compared to Pastry. Consider an erroneous entry in Pastry's routing table. It could have been added maliciously or because of some bug in the system or the protocol's implementation. Then the message will be sent to a node that perhaps has a very low "prefix match". It will then take a long time for the request to arrive at the correct node. In the case of Chord, the protocol will recover very quickly. This makes Chord a very suitable choice for networks where we expect such kinds of bugs and system errors.

We shall start with a discussion on consistent hashing, which provides

a mathematical framework for assigning keys to nodes on a circular ring. Many of the results and concepts will be used to explain design choices in the Chord algorithm.

Consistent Hashing

The key idea in a distributed hash table is to create an overlay of nodes, and attach a key with a node based on the closeness of hash values. A structured overlay is needed to locate the node associated with a key efficiently (as we have seen in the case of Pastry). The crux of this idea is to design a hash function that uniformly assigns keys to nodes. If the distribution is non-uniform, then some nodes will be overly loaded. This can cause a slowdown. Furthermore, if we add or delete a node some keys need to be moved around to ensure that there is a uniform distribution and no key is dropped. We wish to minimize such data movement.

Definition 3.3.4 Consistent Hashing

Consistent hashing is a method to assign hash values to keys and nodes such that the following two properties are ensured:

1. The keys are uniformly assigned to nodes.
2. If a node is added/deleted, data movement is minimized across nodes.

Towards formalizing consistent hashing, we start with the following definitions. Let us imagine a distributed hash table with $\mathcal{T}$ as the set of items and $\mathcal{B}$ as the set of buckets. A bucket has a value associated with it. It is like a point on Pastry's ring. It is not a set of items as the name would imply. Items are added to buckets. A single item cannot be added to two buckets. Let a *view* V be defined as a set of buckets. Formally, $V \in 2^{\mathcal{B}} - \phi$. This means that a view is an element of the power set of all buckets – it is a set of buckets and is non-null. Let the set of all possible views be $\mathcal{V} = 2^{\mathcal{B}} - \phi$.

In Pastry, we assumed that every node ID is a bucket. Keys were mapped to buckets based on proximity. If a node was added or deleted, it basically meant that a bucket was added or removed. If we assume that the view comprises all the node IDs, then the view changes when a node is added or removed.

Let us define a ranged hash function. It is a function of the form: $2^{\mathcal{V}} \times \mathcal{T} \rightarrow \mathcal{B}$. The hash function takes as its input a view and an item. It maps

the item to a bucket that is a part of the view. Note that the last statement is quite important – the bucket has to be a part of the view.

Let us now quickly highlight a few desirable properties of ranged hash functions. Assume a ranged hash function is of the form f_V, where V is its associated view.

Balance: A hash function f_V is said to be balanced if the fraction of items mapped to any bucket is $O(1/|V|)$. Note that we expect any hash function to be balanced.

Monotonicity: This property is specific to a family of ranged hash functions. Assume any two views V_1 and V_2 such that $V_1 \subseteq V_2$. The monotonicity property is as follows: $f_{V_2}(i) \in V_1 \Rightarrow f_{V_1}(i) = f_{V_2}(i)$. This means that if in an enlarged view (V_2), item i is mapped to bucket b, where $b \in V_1$, then even in view V_1 ($\subseteq V_2$), item i is mapped to b.

Let us explain this in the context of Pastry. Assume that we add a new node. The view expands. The new view is now a strict superset of the older view. Assume a key k that got mapped to node n in the new view. It so happens that node n exists in the older view as well. The property of monotonicity implies that even in the older view that is smaller, key k maps to the same node n. In other words, if the view is expanding keys either stay where they are or move to freshly added nodes. The movement is basically restricted.

Spread: The *spread* refers to the number of different values that a ranged hash function has for an item i under a set of views. Let $V_1, V_2, \cdots, V_k$ be k different views. Let the set $W = \cup_{i=1}^{k} V_i$. The size of each view is at least $|W|/\omega$, where ω is a constant. Let us use the symbol n to represent the total number of buckets. $n = |W|$. Let us refer to this property as the *View Property*.

For a given item $t \in \mathcal{T}$ and a random-ranged hash function f, we define the spread $\sigma(t)$ of t as follows: $\left|\cup_{i=1}^{k} f_{V_i}(t)\right|$. The spread of t is basically the number of distinct buckets that it can be hashed to in the k different views. We can proceed to define $\sigma(f)$ as $\max_{t \in \mathcal{T}} \sigma(t)$, which is the spread of the ranged-hash function f.

In the context of Pastry, the spread can be interpreted as the tendency for an item to move to a new bucket if the view is changed (a node is added or deleted). If the spread is low, it means that items tend to stick to their buckets. Alternatively, if the spread is high, it means

that items tend to move around a lot as the view changes. The reason for using the constant ω is that we don't want a view to be very small. Otherwise, when we swap the view, there will be a lot of movement of keys, which will skew the results. We want all the views to roughly have the same size, which captures a realistic situation quite well. For example, the number of nodes in Pastry does not change drastically over short durations of time.

Load: Intuitively, the load specifies the number of items that are mapped to any given bucket by a hash function. When we move to consistent hashing, this property needs to be further nuanced.

Let $V_1, V_2, \cdots, V_k$ be k different views. Let b be a bucket in $\mathcal{B}$ and f be a random-ranged hash function. The load $\lambda(b)$ of b under f is $\left|\cup_V f_V^{-1}(b)\right|$. In other words, the load of a bucket is the number of items that are mapped to b under *any* view. The load of a ranged-hash function f is $\max_{b \in \mathcal{B}} \lambda(b)$. Let us represent it as $\lambda(f)$.

Intuitively, the load represents the number of items that are possibly mapped to a bucket given a set of k views. We would want this quantity to be as balanced as possible across the buckets.

As can be noticed, apart from balance, all the other three properties are beyond what a traditional hash function provides. A good consistent hash function should have a low spread and load.

Overview of Chord

The basic idea is the same as Pastry – we have a circular overlay, where nodes are placed in an increasing order of hash values. There are however some differences. We don't map a key's hash to the node whose hash value is the closest. We instead map a key to the next node on the circle that has the same or higher id. Basically, given a hash value, we locate it on the circular overlay and then traverse clockwise (increasing IDs in a modular sense (modulo with respect to the max id)). We stop at the first node that we arrive at while traversing the circle clockwise. This is the node that the key is mapped to. It is important to note that traversing the id circle clockwise is equivalent to incrementing the position on the ring using modular arithmetic. If we move Δ points, then the new position *new_pos* is related to the *old_pos* as follows:

$$new\_pos = (old\_pos + \Delta) \ mod \ 2^m \tag{3.10}$$

Let us now look at the same activities such as routing, node addition and deletion (as we had explained for Pastry). In this case, we look at hashes bitwise as opposed to considering them as a sequence of hex digits. The bitwise treatment is not that inefficient as we shall quickly see. In fact, in our view, it leads to a more elegant implementation. Let us assume that there are m bits in a node ID (its hash basically). Let every node maintain a dedicated routing table, which we shall refer to as the *finger table*.

The Finger Table

For each node, let us define two terms: *successor* and *predecessor*. For a given node on the circular overlay (just referred to as *ID circle* or just *circle* henceforth), let us define the next node (higher id) as the successor and the previous node (lower id) as the predecessor. The definition of the i^{th} finger is as follows for a given node with ID n. Assume that each hash of a key or a node ID is m bits.

$$\begin{aligned} finger[i].start &= (n + 2^{i-1}) \ mod \ 2^m, \quad (1 \leq i \leq m) \\ finger[i].end &= (n + 2^i - 1) \ mod \ 2^m \\ finger[i].node &= successor(finger[i].start) \end{aligned} \tag{3.11}$$

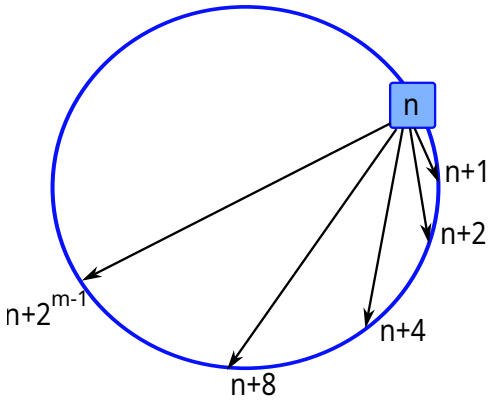


Figure 3.10: Illustration of chords

Figure 3.10 shows the set of starting positions of all the chords from a given node (with hash n). Figures 3.11 and 3.12 show a few examples. They show nodes on an ID circle and the fingers that they are mapped to. Figure 3.12 shows the mapping of a key to a succeeding node on the ID circle.

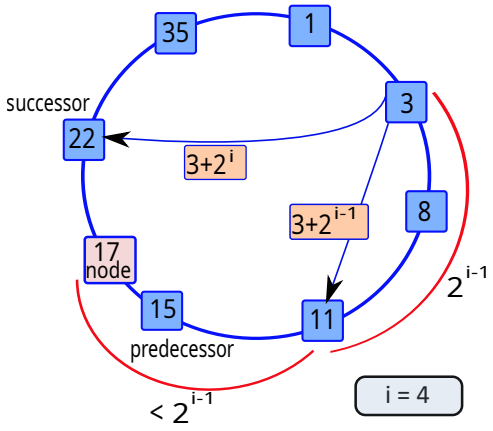


Figure 3.11: Fingers of node 3 where $i = 4$.

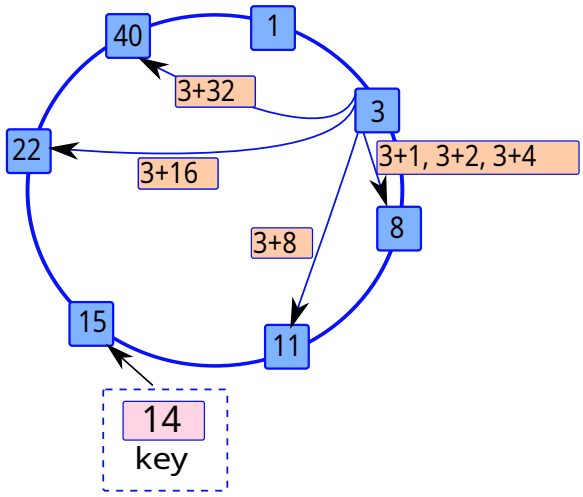


Figure 3.12: Fingers of node n

We observe that the span of fingers increases successively by a factor of 2. They cover bigger and bigger parts of the ID space. We shall see that this design where the sizes of the fingers increases exponentially will help us design a very efficient search algorithm. *Fingers* partition the ID space. Let us list the span of fingers in a tabular form (refer to Table 3.1). In the table we perform all the additions modulo 2^m. For the sake of readability, we omit the term *mod* 2^m in each expression; however, it is important to keep this

Finger (i)	Start	End
1	$n + 1$	$n + 1$
2	$n + 2$	$n + 3$
3	$n + 4$	$n + 7$
4	$n + 8$	$n + 15$
$\ldots$	$\ldots$	$\ldots$
m	$n + 2^{m-1}$	$n + 2^m - 1$

Table 3.1: A tabular list of fingers. Note that all these additions are modular additions ($mod\ 2^m$).

in mind. Let us look at the last finger, which is also the largest finger. This finger is the m^{th} finger. It terminates at $(n + 2^m - 1)\, mod\ 2^m$. Since we are assuming that each ID is represented using m bits, the total number of IDs that can be stored on the circle is 2^m. We thus see that the end of the last finger is the ID that is just before the ID of the current node (position n on the circle). Hence, all the fingers cumulatively cover the entire circle. The size of the first finger is just one point and the size of the last finger is 2^{m-1} points on the ID circle. In fact, the sizes of the fingers – number of points contained in them – increase exponentially: 1, 2, 4, 2^{m-1}. We are basically spanning larger and larger chunks of the ID space, which nicely summarizes our discussion. Such an arrangement allows us to achieve logarithmic search complexity.

We shall see that the finger's node plays a key role in the overall key lookup process. Recall that it is the successor of the starting point of the finger. It may lie within the finger, or may actually be in a different finger.

Chord's Hashing Algorithm

The SHA-1 algorithm is used to hash node IDs and keys. It produces a 160-bit (20-byte) value at the end of the process. We need a *consistent hashing* scheme where regardless of the number of nodes, keys are uniformly distributed among them. It has other desirable properties as we have discussed earlier: monotonicity, low load and low spread.

For example, if there are n nodes and k keys, then we expect to have k/n keys per node. If the hashing process produces random numbers that are uniformly generated, then theoretically this will be the case for large values of k and n (as per the law of large numbers). However, there is a finite probability of a deviation from the uniform distribution. This unduly

loads some nodes and reduces the load at many other nodes.

Let us mathematically analyze this situation (refer to [Stoica et al., 2003, Karger et al., 1997]). We can prove that with a high probability, each node stores at the most $(1 + \epsilon) \times k/n$ keys. Any addition of a node and subsequent redistribution of keys moves around $O(k/n)$ keys. However, the key result is that $\epsilon = O(\log(n))$, which is sadly large. We would have liked ϵ to be $O(1)$. An overhead of a factor of $O(log(n))$ militates against the philosophy of consistent hashing. There is a need to achieve more uniformity, particularly if the number of keys is on the higher side (like in a large e-commerce site).

The following trick ensures that this $\log(n)$ factor is successfully removed. We use $\kappa = log(n)$ hash functions. All of them need to be high-quality hashing algorithms (similar to SHA-1). A standard approach to do this is to append a few additional bits (per hash function) known as *salt* values. The SHA-1 algorithm is designed in such a way that even if there is a small variation in the text (a few bits), the output is very different. Our hash functions are thus as follows: $\mathcal{H}_1 \ldots \mathcal{H}_\kappa$. Now, for every single node we create κ different virtual nodes, all placed at different positions on the ID circle. We are thus artificially inflating the number of nodes by creating $log(n)$ copies of each physical node and placing them at different points on the ID circle.

In total, we have $\kappa \times n$ virtual nodes. If $\kappa = \log(n)$, we have $n \log(n)$ nodes on the ID circle. It can be proven that this trick ensures that the key distribution is roughly balanced across the physical nodes. At the cost of additional computational and routing complexity, using the notion of virtual nodes helps in better (and more uniform) key distribution.

Remark 3.3.3

Using virtual nodes in Chord ensures balanced key distribution across the physical nodes. This comes at the cost of increased routing complexity.

Chord's Routing Algorithm

Algorithm 5 shows the high-level overview of the algorithm. We find the predecessor `P` of the key whose hash value is `id`. Every node has a pointer to its immediate successor and predecessor on the ring. The node that the key is mapped to is just the successor of `P` (by definition). This can be easily visualized with an example. The `id` will lie between `P` and `P.successor`.

Hence, the node that `id` is mapped to is `P.successor`. Since every node has a successor, it is quite easy to find `P`'s successor.

Algorithm 5 Find the node that is mapped to the key with hash `id`

```
1: procedure findSuccessor(id)
2:     P ← findPredecessor (id)
3:     return P.successor
4: end procedure
```

Now, the primary task is to find the predecessor of the key. This is where the finger table will prove to be useful. Let us first look at the algorithm to find the closest preceding finger node for a key with hash `id` (refer to Algorithm 6). The idea is to traverse the list of fingers in descending order – largest finger first. The process of traversal terminates when the finger's node lies between the ID of the current node and the hash of the key (`id`). The order should be: ⟨current node initiating the search⟩ → ⟨finger's node⟩ → ⟨hash of the key⟩. This ensures that we reach a node that is closer to the key than the current node. We shall later on prove that we are at least halving the distance; however, at the moment just saying that *we are getting closer* suffices. Note that in Algorithm 6, the operation ∈ takes two arguments that correspond to two different nodes on the ID circle. We can assume this to be an overloaded function whose arguments can either be nodes or hashes or any combination thereof. It will be clear from the context. In the following algorithm, the node `cur` refers to the node that is initiating the lookup operation.

Algorithm 6 Find the closest preceding finger's node

```
1: procedure closestPrecedingFinger(id)
2:     for i ← m to 1 do
3:         if finger[i].node ∈ (cur, id) then
4:             return finger[i].node
5:         end if
6:     end for
7:     return cur                          ▷ The current node is the closest.
8: end procedure
```

Let us now take a deeper look at Algorithm 6. In Line 2, we iterate through the fingers (largest-first). The m^{th} finger (or chord) corresponds to the semicircle. We first check if the `id` is in the other semicircle with respect

to the current node. Of course, it is possible that the finger's node may not be close to its starting point. Hence, the check in Line 3 is slightly more nuanced. We find if the current finger's node is between the current node (`cur`) and the hash of the key (`id`). If that is the case, then the current finger's node is closer to the key.

Assume that the check in Line 3 fails. It means that the current finger's node is not in between `cur` and `id` – it is beyond `id`. We iterate and consider the next finger. Assume that the check in Line 3 is successful for finger `j`. We return `finger[j].node`, which is the node corresponding to the j^{th} finger (lies between the current node and the key). If no such node is found, then we come to Line 7, where we conclude that the current node is the closest to the key, and we return it.

Using this algorithm we need to find the predecessor of a key. The steps are shown in Algorithm 7.

Algorithm 7 Find the predecessor

```
1: procedure findPredecessor(id)
2:     tmp ← cur
3:     while id ∉ (tmp, tmp.successor) do
4:         tmp ← tmp.closestPrecedingFinger(id)
5:     end while
6:     return tmp
7: end procedure
```

The idea behind Algorithm 7 is to keep getting closer and closer to the key until it is not possible to get any closer. Here, *getting closer* means getting closer while still preceding the key on the ID circle. We keep calling the function closestPrecedingFinger () repeatedly in Line 4. In each iteration of the loop, we find the finger node that is the closest to the `id` yet still precedes it, as the name of the function suggests. This is our new starting point. We again start from here. Given that we are closer to the key now, we can rely on the fine-grained information in its finger table to get even closer. Ultimately, we reach the real predecessor of the key.

Corner case: An astute reader may argue that these algorithms do not handle the case in which the ID of a node matches the ID (hash) of a key. This is in reality very rare given that the key space is very large (2^{128}). Theoretically, this can happen. These algorithms will require minor changes to incorporate this condition. For the sake of readability, this condition has been ignored.

Remark 3.3.4

In Chord, the finger table replaces the routing table. We traverse it in descending order of the finger's size to find the next node that we need to route the message to. The protocol is more robust unlike Pastry. It is not dependent on how well the routing table is populated and thus guaranteeing quick forward progress is easier.

Node Arrival

In any system that uses DHTs, nodes keep on arriving and departing. Hence, there is a need to continuously modify the DHT to include new nodes and remove older nodes. This is in fact one of the key advantages of a DHT – we can grow and shrink it at will. There are two key aspects of adding a new node. We need to update the state of the node that is added and inform other nodes about the newly added node. This is quite similar to Pastry. However, given that Pastry uses more state information (leaf set and neighborhood set), this process requires more messages. Node addition in the case of Chord is far easier. We precisely know which nodes will change their state – they can be informed accordingly.

Initialize the Finger Table

Initializing the finger table is the only step in initializing the state of a new node that is going to be added. This process is shown in Algorithm 8. We first update the predecessor and successor information. Then we find the nodes corresponding to each of the fingers and update the finger table with this information.

The current node `cur` calls the function initFingerTable(). For getting all the information that it needs as a part of this function, it needs the help of another node `T`. Node `T` exports a couple of functions that the newly added node (`cur`) needs to use.

For every finger, we know its start and end points as per Equation 3.11. We need to find each finger's node, which is the successor of the starting point (on the ID circle). Doing this for the first finger is not very hard (Line 3). The successor of `finger`[1].`start` is the finger's node (by definition). We need the help of node `T` to achieve this task. Node `T` is already there in the DHT and its finger tables are fully initialized. It can search for any id's successor using the default lookup algorithm.

Once, we have found the successor, we initialize two local variables associated with each node: `successor` and `predecessor`. The `successor`

Algorithm 8 Initialize the finger table

```
1: procedure initFingerTable(Node T)
2:     /* Update the predecessor and successor */
3:     finger[1].node ← T.findSuccessor (finger[1].start)
4:     successor ← finger[1].node
5:     predecessor ← successor.predecessor
6:
7:     /* Update the predecessor and successor of neighboring nodes */
8:     successor.predecessor ← cur
9:     predecessor.successor ← cur
10:
11:    /* Initialize the rest of the fingers */
12:    for i ← 1 to m − 1 do
13:        if finger[i + 1].start ∈ (cur, finger[i].node) then
14:            finger[i + 1].node ← finger[i].node
15:        else
16:            finger[i + 1].node ←
                                  T.findSuccessor (finger[i + 1].start)
17:        end if
18:    end for
19: end procedure
```

was obtained in the previous step. The predecessor of the current node is nothing but the successor's predecessor.

Then we move on to changing the state of the successor and predecessor. They need to be made aware of the current node that is being added. Their successor and predecessor pointers are appropriately modified in Lines 8 and 9, respectively.

Then, we proceed to initialize the rest of the fingers. We need to take the sparsity of nodes along the id circle into account. If `finger[i + 1].start` is in between the current node `cur` and `finger[i].node`, then it means that if we start traversing the ID circle clockwise from `finger[i + 1].start`, the first node that we will reach is `finger[i].node`. Then we set `finger[i + 1].node` to `finger[i].node` (see Line 14). If this is not the case, then we need to initiate a search to find `finger[i + 1].node`. We can leverage node `T`'s FIND-SUCCESSOR() method that finds the successor of `finger[i + 1].start`.

Update the Finger Tables of the Rest of the Nodes

Given that the finger table of the current node has been updated, we next need to update the finger tables of the rest of the nodes. We have already done some work in Algorithm 8 by updating the predecessor and successor pointers of the succeeding and preceding nodes, respectively. The only task that is remaining is updating the finger tables of other nodes. It is possible that the current node `cur` (that is being added) is a finger's node in many of the other nodes. This information needs to be propagated to them.

Algorithm 9 shows the pseudocode for updating the finger tables of other nodes. The current node that is being added (`cur`) calls the function UPDATEOTHERS(). We start with the smallest finger and keep moving towards larger fingers. Consider the i^{th} finger. The current node can be in the finger table as the i^{th} finger's node for any node that has a hash value less than (`cur.hash` - $2^{i-1} + 1$ (mod 2^m)) ($=\lambda$). Starting from λ, we search for nodes with smaller hash values. This is the same as finding the predecessor of λ (see Line 4). Subsequently, starting from this predecessor node, we start updating the finger tables of all the relevant preceding nodes (refer to the function UPDATEFINGERTABLE()).

The function UPDATEFINGERTABLE invoked by a predecessor node `pred` takes two arguments: the node that is going to be added (`T`) and the number of the finger (`i`). The vital check is performed in Line 11. Let us understand the relationship between the node to be added (`T`), `finger[i].node` (in the node `pred`) and `pred` itself.

We are interested in the i^{th} finger of `pred`. We know the following:

Algorithm 9 Update the finger tables of other nodes

```
1:  procedure cur.updateOthers(())
2:      for i ← 1 to m do
3:          ▷ This node may have the current node in its finger table
4:          pred ← findPredecessor (cur.hash - 2^(i−1) + 1)
5:          pred.updateFingerTable (cur, i)
6:      end for
7:  end procedure

8:  ▷ This function is invoked by the node pred
9:  procedure pred.updateFingerTable(Node T, int i)
10:     ▷ Update the finger table of the node if T is a finger's node
11:     if T ∈ (pred, finger[i].node) then
12:         finger[i].node ← T
13:         predecessor.updateFingerTable (T, i)        ▷ Recursive call
14:     end if
15: end procedure
```

$$\begin{aligned} &\texttt{pred.finger[i].start} = \texttt{pred.hash} + 2^{i-1} \\ &\texttt{pred.finger[i].node.hash} \geq \texttt{pred.finger[i].start} \\ &\texttt{T.hash} \geq \texttt{pred.hash} + 2^{i-1} = \texttt{pred.finger[i].start} \end{aligned} \tag{3.12}$$

The relationship between `T` and `pred.finger[i].node` is thus not clear. It is clear that both are at least as large as `pred.hash` and `pred.finger[i].start`. There are two cases: either `T` is between `pred` and `pred.finger[i].node` (Case I) or not (Case II).

Case I:
Given that `T` is located after `pred.finger[i].start` on the ID circle, and it is located before `pred.finger[i].node`, we observe that `pred.finger[i].node`'s new value should be `T`. It is closer to `pred.finger[i].start`. This is exactly what is done in Line 12. Now, it is possible that additional nodes are also affected. We thus perform the same procedure on `pred`'s predecessor (Line 13) and keep doing this as long as the check in Line 11 is satisfied.

Case II:
In this case, `T` is located after `pred.finger[i].node` on the ID circle. As a result, the value of `pred.finger[i].node` does not have to be updated. It

is not affected by the addition of the new node. The same is the case for `pred`'s predecessors as well. As a result, in this case, we can just return from the updateFingerTable (...) function without performing any action.

Algorithm 10 Delete the node from finger tables

```
1: procedure cur.deleteNode(())
2:     for i ← 1 to m do
3:         pred ← findPredecessor (cur.hash - 2^(i−1) + 1)
4:         pred.deleteNode (cur, i)
5:     end for
6: end procedure

7: procedure pred.deleteNode(Node T, int i)
8:     if finger[i].node = T then
9:         finger[i].node ← T.successor
10:        predecessor.deleteNode (T, i)
11:    end if
12: end procedure
```

Node Departure

Let us now see what happens if a node leaves the Chord network. This can happen in two ways. One is that the node voluntarily leaves. For instance, we decide to scale down the Chord network and power servers down. In this case, there is a case for a controlled exit where the exiting node can initiate a departure protocol. If a node crashes, then other nodes need to detect its exit and initiate a departure protocol. In both cases, the departure protocol is quite similar.

Let us assume the simpler case where a node voluntarily leaves. The first action is to inform the successor and predecessor nodes. They need to update their respective successor and predecessor pointers. The more difficult task is to update the finger tables of other nodes.

We follow a similar procedure as adding a new node. The current node `cur` that is going to be deleted calls the deleteNode() function. We iterate through all the fingers and delete the current node from the finger tables of all the nodes that may contain it. Similar to the case of adding a new node, we find the predecessor node of (`cur.hash` - $2^{i-1} + 1$). Let this node be `pred`. We then call `pred`.deleteNode (...).

If `pred.finger[i].node` = `T`, then we need to set a new value for the

finger's node. This node in this case will be `T.successor`. It is the new node for `pred`'s i^{th} finger. Now, it is possible that other nodes would have `T` as the node of their i^{th} finger. We follow the same procedure and recursively call the DELETENODE (...) function for the predecessor of `pred` and so on (until the check in Line 8 becomes false).

3.4 Third-Generation P2P Networks

Third-generation P2P networks are highly structured networks. They typically have well-defined overlays structured as trees, meshes, rings, etc. They are highly scalable and do not have centralized servers. We will discuss two popular networks in this space. BitTorrent is a very popular file discovery and sharing service, whereas FreeNet (now Hyphanet) is a secure network that keeps the ID of the sender and receiver anonymous.

3.4.1 BitTorrent

Let us now introduce, BitTorrent [Izal et al., 2004], the most popular second generation peer-to-peer file-sharing network as of 2024. This network has stood the test of time. It was primarily designed to distribute large files, particularly videos. BitTorrent was released in 2001. By that time videos had already become quite popular. The market for images and MP3 songs had diminished. Over time BitTorrent started getting used to distribute operating system images, open-source packages, large scientific and government datasets, educational content, archived data and backup/recovery images.

Torrent Files

The first step in using BitTorrent is to create a torrent descriptor file (example shown in Listing 3.4.1). It is popularly known as a *torrent*. This file has the following contents: details of the file (metadata) and an SHA-1 hash of the file's contents. It is meant to be stored and distributed via search engines. As opposed to Napster and Gnutella, the aim is to operate a legal network and not circumvent law enforcement authorities or violate legal provisions.

```
1 {
2   "announce": "http://tracker.example.com:8080/announce",
3   "announce-list": [
4     ["http://tracker.example.com:8080/announce"],
5     ["http://backuptracker.example.com:8080/announce"]
```

```
  ],
  "info": {
    "piece length": 524288,
    "pieces": "20-byte SHA-1 hashes concatenated",
    "name": "example_file.txt",
    "length": 12345678
  },
  "creation date": 1622488800,
  "comment": "This is an example torrent file.",
  "created by": "ExampleTorrentCreator 1.0",
  "encoding": "UTF-8"
}
```

After a server starts hosting a file, it creates a torrent file and distributes it to other servers. It plays the role of a *seeder* in the sense that it disseminates information about a file throughout the network. The torrent files are then cached by dedicated trackers that store a mapping between a torrent file and the locations at which parts of the file are stored.

Pieces of a File

In a BitTorrent network, there is a *swarm* of hosts. Every host needs to participate in the protocol. It can be a simultaneous downloader and uploader. The key idea here is to break a large file into multiple segments called *pieces*. The size of each piece is limited to 256 KB. The pieces are then distributed to peers (host machines on the networks). Each peer hosts the pieces of different files; it can further redistribute these pieces by sending them to other peer nodes. This paradigm represents a departure from other designs that treated a file as a single atomic unit. They consider smaller files and try to restrict network usage. However, by the time BitTorrent was released, there was a demand for bigger files. Simultaneously, network bandwidths had also increased. This is why its design is based on redistributing pieces of a single file.

A torrent client can then simultaneously download the different pieces of a file from different hosts. It is helped by *trackers* in this process that are dedicated servers, which maintain a mapping between pieces of a file and BitTorrent nodes. The tracker further coordinates the process of downloading the file.

The overall approach is thus as follows. A BitTorrent client connects to a tracker and obtains a list of peers that contain the different pieces of a file. Then it establishes separate connections with the peers to fetch the different pieces. Trackers are similar to the Napster server. However, the

key difference is that in this case there is no need to try to hide anything given that the BitTorrent protocol was designed to run in full compliance of existing laws.

There is an alternative approach that avoids trackers altogether. We can use DHTs. Then we can store the pieces on a DHT after naming them appropriately. The version of BitTorrent that uses a DHT is also quite popular. It is called the *mainline DHT* that is based on the Kademlia DHT protocol [Maymounkov and Mazieres, 2002].

Downloading and Shared Files

Let us now consider the process of downloading and sharing files. Users need to use regular search mechanisms that includes Google searches to find torrent descriptor files. If a server has a new file, it hosts it. Subsequently, it distributes the torrent file to let client machines and trackers know about it.

Once a client gets a torrent file, it connects to a tracker and gets the list of peers. Then, the individual pieces are downloaded simultaneously. The pieces can be downloaded in a random order concurrently. We can use different strategies here.

For example, we can prioritize traffic for those nodes that have already sent a lot of data on the network. This basically means that we reward altruism. When such a node decides to download a file, the sender also follows a tit-for-tat policy, where it preferentially sends data to nodes that have sent it data in the past. Furthermore, some bandwidth can be reserved by a node for itself. It needs to multiplex its network between uploading, downloading and other network-related activity that is unrelated to BitTorrent. The key idea here is that we follow a transactional approach where if a node has been nice to the network, then it is rewarded.

Scalability

Let us also comment on scalability. The mainline DHT, which is based on Kademlia is the largest DHT in the world. As of 2024, it has somewhere between 10 million to 25 million nodes. This is what makes the BitTorrent network so popular. All the current versions of the BitTorrent clients are compatible with the mainline DHT. This is why a lot of important pieces of software, including variants of the Linux® kernel, are available on BitTorrent. Given that the BitTorrent network is widely accepted now, it is perfectly legal to provide a torrent link to someone who is interested in

downloading a file or a piece of software.

Alternative approaches also exist. For instance, we can use a gossip-based protocol to share BitTorrent files. The Tribler protocol [Pouwelse et al., 2008] enables this. We can also use anti-entropy based messaging to regularly exchange lists of torrents. BitTorrent clients get smarter with time because they gradually get to learn the interests of the user. They search for matching torrents on the network. Based on the user's interests, they provide information about torrents that the user is most likely going to like.

Security and Privacy Concerns

Let us now address security and privacy concerns. As we have discussed, anonymity and privacy are not a part of the BitTorrent protocol. The legal onus lies on the clients and the site that indexes and hosts the torrents: the tracker and server nodes. In this case, culpability can easily be established. Everybody involved in the hosting and subsequent propagation and dissemination of potentially copyrighted or illegal material is culpable and liable to legal action (depending upon the laws of the specific country).

3.4.2 Freenet

Let us now look at "secure" third-generation peer-to-peer networks. The quintessential example of such a network is *FreeNet* [Clarke et al., 2001] It was designed to remedy the "visibility" problems of second-generation peer-to-peer networks. Its main aim is to provide a high degree of anonymity: it is not possible to find the origin of a file or the ID of the requesting node. This was the beginning of the *dark web* that offers reasonably anonymous browsing and file hosting. Of course, there can be many opinions about how useful the dark web is to human society. It is the favorite visiting spot of cybercriminals, terrorists and for-profit hackers. Nevertheless, we can always study it from an academic point of view.

In Freenet, along with a guarantee of anonymity, there is also a great degree of deniability. This basically means that a server can deny that it ever stored a file; there is no way to disprove it. It is also possible for a requesting client to claim that she never requested for the file in the first place – she was simply participating in the network without playing any malicious role. Freenet (renamed to Hyphanet in 2023) is now available as an open source project at the following website: `https://github.com/hyphanet`.

Freenet Node

Let us start with looking at a Freenet *node*. Each node maintains a local data store (private database). Like Gnutella, each node also maintains a routing table. This routing table stores the addresses of a few of the other nodes in the network and the keys that they possibly might be storing. To maintain anonymity and secrecy, no node should have knowledge of the entire network. Instead, it should only have knowledge of its immediate neighborhood, which is the logical neighborhood, not the geographical neighborhood that is related to physical proximity (like DHTs).

The queries are sent to the local Freenet node that is expected to pass it on to neighboring nodes. Each message has a TTL (time to live) field, which we have also seen in second-generation unstructured networks. This is decremented at every hop. A query also has a pseudorandom identifier. This ensures the while forwarding it, cycles are not introduced, otherwise it is possible that a query message goes round and round in cycles in the network. If a node observes that it has forwarded a query in the recent past with the same pseudorandom identifier, which is stored in an internal table, it refuses to accept the message again from a neighboring node. This strategy ensures that the query message can go deep into the network without propagating in cyclic loops (subject to the TTL). Note that in Freenet, a node does not replicate messages. Similar to DHTs, this is an informed process, with every hop we get closer to the key with high probability.

Searching for a Key

Let us now understand the steps in the overall key search process. The querying node hashes the name of the file and produces the key K. It then looks up the key in its routing table. Assume it finds key K_1 that is the closest to K in its routing table. Let the node that stores K_1 be N. The request is then forwarded to N. If N finds the file, it returns its contents along with its IP address. N lies and claims to be the owner of the file, which may or may not be true. This is the crux of the idea. N may not be the original owner. It may just possess a copy of the file because it may have seen a copy of the file earlier.

However, if N does not find the file in its local database, then it follows the same process. It finds the key K_2 in its routing table that is the closest to K and forwards the request to the node that stores K_2.

Ultimately, either the TTL will expire or the file will be found. If the TTL expires, nothing needs to be done. The file is declared to be unavailable

in the network. However, if the file is found in some node N_x, then there is a need to send its contents back to the requesting node without revealing where exactly the contents of the file were found. This is because we cannot trust the nodes that are in the network. Hence, strict anonymity by obfuscation is required here. The result of the query (if the key is found) follows the same path as the original search request (similar to Gnutella). Each node on the way claims to be the owner of the file – this hides the real owner.

A few more things are done to introduce some additional confusion while processing later requests. Every node on the way *caches* a copy of the file and creates an entry in its routing table with the key and ID of the data source. Note the game that is being played here. The nodes falsify the ID of the owner of the file and forward that information towards the requesting node.

Let us now consider a few corner cases. If a node cannot forward the request to another node because there is a cycle or a link failure, it tries the second-closest key in its routing table – the one with the second-closest distance to the search key K. If there is some problem in contacting the corresponding node that stores the second-closest key, then we try with the third-closest key, so on and so forth.

Figure 3.13 shows an example of the execution of the Freenet protocol. Look at the order of the messages, the lack of cycles and the path that the request takes.

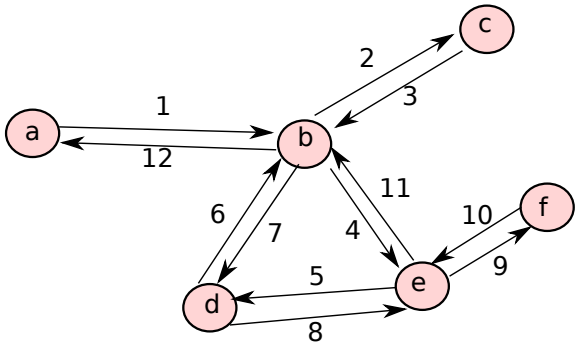
Gradually over time, the information about files disseminates. Furthermore, nodes start aggregating files with similar keys. This reduces the search distance because more keys with similar hash values are now available on the same node. This also ensures that the corresponding file is available with a higher probability owing to the reduced search distance. Also, another positive side effect is that popular data will tend to get replicated at many nodes, and it is thus easy to locate and fetch it. However, on the flip side, routing tables will get bigger because new entries shall get created and added to routing tables all the time. This increases the storage overheads. But it also allows nodes to quickly reach a large and relevant part of the network from the point of view of the key search process. This happens without sending a lot of messages and possibly revealing the identity of the requesting node in this process.

File Storage Protocol

Let us now look at the file storage protocol in more detail. The user first creates a file key. She then sends an ⟨`insert`⟩ message to its own node.

(a) Routing Table

key	address	content (?)
key	address	content (?)



(b) Example: Multiple messages being sent

Figure 3.13: Example of communication in Freenet

Every node is supposed to have a record of the files that it has. A record is a 3-tuple: file, key and TTL. If the node finds the key in its own routing table, it tries to modify the key and tries again till it succeeds. There are several ways of generating a key. A simple approach is to just hash the title or the contents of the file and use a part of the bits as the key. If there is a collision, we can use another part of the hash or keep on appending known pieces of text to the file, and keep generating new hashes. Once there is no hash collision in the local node, it finds the closest key in the routing table and forwards the ⟨`insert`⟩ message to the corresponding node N'. If the ⟨`insert`⟩ message causes a hash collision in node N', priority is given to the file present in node N' (one that led to the hash collision). Let this file be F. It gets priority. A message is sent back to upstream nodes (nodes that sent the message to N') asking them to delete the new file. A revised attempt can be made with a new hash. Finally, note that the TTL field will determine how far the file's information is disseminated in the network.

Now assume that there are no hash collisions. The ⟨`insert`⟩ requested is forwarded until the TTL becomes 0. It means that the key insertion was successfully done. Furthermore, each node on the path that has been followed adds the file and the key to its respective routing table. It also

caches a copy of the file. The original node also adds the file and key to its internal data store and routing table. It is important to note that inserting a new file in the network is an elaborate process. Hash collisions are not allowed for all the nodes that lie on the path of the insertion request.

Now, the interesting part is that it is possible to beat the security of the nodes along the way. A node can lie about the data source of a file. For the data source, nodes either store their own address or some other node's address. There is some randomness here. This makes it difficult for a third party to figure out the actual source of the file.

The advantages of this mechanism are as follows:

Newly inserted files are placed on nodes with similar keys. This makes the search process more efficient. Furthermore, any information about newly inserted files can be very quickly disseminated to other nodes that store similar keys. Given that we have an elaborate protocol to avoid hash collisions, it will be very difficult for any intruder to deliberately introduce hash collisions.

Let us now look at data management. The routing table and data stores are finite-sized structures. Hence, they need to follow a replacement policy. We need to keep data that most likely will be accessed in the near future. For making such replacement and eviction decisions, the LRU (Least Recently Used) replacement policy is followed. LRU ensures good performance in most realistic settings. It ensures that unused files get deleted from the system.

Two-Level Hierarchical Structure

Note that a key feature of the Freenet protocol is that a file can be hashed with numerous hash functions. We first try with the first hash function. If there is a hash collision, then we use the second hash function, so on and so forth. This is required because a malicious adversary may deliberately cause hash collisions by adding spurious entries into the network. A question that arises is which hash function do we use when searching for a file? How does the searching node know that the k^{th} hash function needs to be used? This means that the searching node clearly needs to have more information. This is how the process works.

A simple solution is to make this information public or somehow known to the client node that is searching for the file. This can happen by an out-of-band mechanism like maybe sending an e-mail to the user who needs to search the file. Another brute force method is to simply try different hash values and keep searching for the file. Here, we assume that the user has

some way of knowing whether the file that has been received is the correct file or not. She can try the first hash function, initiate a search, try the second hash function, so on and so forth. This process terminates either when all the tries have been exhausted or the file has been received.

Let us explore a 2-level hierarchical solution that modern versions of Freenet use.

Freenet uses a Keyword Signed Key (KSK) that is supposed to be unique for a file. It can be created out of its title, keywords, portions of its text, or any combination thereof. To ensure uniqueness, we need to have relatively few KSKs such that the probability of a hash collision is low. It needs to be very difficult to find another set of keywords that create the same KSK value. A large file can then be divided into multiple chunks, and each chunk can be separately stored (like BitTorrent). Each such chunk will have a content-hash key (CHK), which will also serve to verify its contents. The probability of hash collision for a CHK is much higher because there are many more CHKs than KSKs, and some chunks may just have all 0s. This is why we need to use multiple hash functions for chunks, and generate a different CHK if there is a collision.

Freenet realizes this abstraction by first searching the network using the KSK for an index file. Each entry of the index file contains the CHK for every constituent chunk. Subsequently, the client node searches for every chunk using its CHK. To realize this, we need to create two different routing tables and storage structures: one for KSKs and the other for CHKs (resp.).

Message Format

Let us now look at the message format. Freenet is a packet-oriented protocol that can run on either TCP or UDP. There are two kinds of basic transactions – search and insert – where every transaction is given a unique transaction id. Every message contains the 64-bit transaction id, the TTL counter and a depth field. The TTL value can be an important source of side information. An attacker can guess how far the requesting node is from it. For example, if the TTL is equal to 5 and the attacker knows that it was initialized to 10, then it can be sure that the requesting node is 5 hops away. To avoid this a node can skip updating the TTL value once in a while (randomly). When the TTL becomes 1, nodes can still continue to propagate the request to other nodes for a random number of hops. The TTL can also be initialized with a random value. Freenet uses a *depth* field that is incremented at each hop. It is initialized with a randomly generated number. Now, before the destination sends the message back to the source,

it sets the TTL equal to the depth. This ensures that the message does not expire before reaching the source and the TTL field does not leak a lot of information.

Upon sending a request, the requesting node can start a timer. It needs to wait for the response unless the timer times out. If the timer times out, a failure can be inferred. It is also possible that sometimes downstream nodes face a lot of network congestion. They send a ⟨`restart`⟩ message back to the requester to inform it that there is a possible delay in the network. The requesting node can extend its timer. Note that a downstream node does not know who the requesting node is. Hence, it needs to send this message back to its upstream node, so on and so forth, until it reaches the requesting node.

Now, if a request is successful, the remote node sends the ⟨`Send.Data`⟩ message back via its upstream nodes. Let us refer to this as the *reply path*. It can send the ID of the data source, which in all likelihood will be fake. However, if a situation arises when no message can be sent because cycles are created and the TTL is still non-zero, the remote node sends the ⟨`Reply.NotFound`⟩ message back to the requesting node via the reply path. The requesting node then sends the key search message to other nodes based on the contents of its routing table.

Security and Legal Issues

In any such decentralized peer-to-peer system, it is important to also look at the legal issues. We need to understand that one of the key reasons behind the creation of Freenet was to tackle or rather avoid all kinds of legal issues. In this case, a data store can totally deny the knowledge of the files that it has. It can simply say that it participates in the protocol, and is thus duty-bound to cache copies of some files that pass through it, and also keep its routing table populated. However, it is not the primary provider of those files, and thus it can escape legal liability. Another idea is to take the hash of the key and use it to encrypt the contents of the file. A node can always decrypt it if it knows the key. This also can help a node escape legal liability because it can always say that it was unaware of the contents of the files that it was storing – the files are encrypted, and the node did not have the key.

Let us look at some other subtler aspects. From the point of view of performance, it is a good idea to have a directory of keys and *group* files with similar names. However, there could be legal issues. Give such a directory, we can immediately get pointers to all similar files; this can be a treasure

trove for law enforcement agencies. However, if there are no such issues, then such directories can be maintained. They can also act like file servers as well (similar to the first-generation network Napster). This does militate against our design goal of ensuring anonymity and thus is the method of choice only in very limited settings, where some anonymity can be sacrificed for performance. It is easier to create a distributed directory that is spread out across the network. Such a directory can store the mapping between a list of keywords and the keys of files. We can augment this information with hashes to verify the integrity of files and ensure that they haven't been tampered with.

We need to note that the sender's anonymity is preserved because a node can never say for sure that the node that just sent it a request is the original sender, or it is merely forwarding a request. Second the receiver's identity is also protected because the TTL and depth fields have some degree of randomness, and all that the true receiver needs to do is simply locally store a copy of the file silently when it gets the search result. To confuse the nodes in its neighborhood it can forward the search result to them, even though they were not a part of the original search process. They will have no idea of knowing if the node forwarding the search result to them is the true receiver or is simply participating in the protocol.

Furthermore, it is always possible to use secure channels between nodes to encrypt messages such that eavesdroppers do not learn anything. This is not a part of the original protocol but can serve to be a useful addendum. It is also possible to pre-route messages where the requester decides the route in advance. To enable this, it would need to know some details of the routing tables on the way. Knowing the nodes along the way will sadly reduce the degree of anonymity. If this is acceptable, then the message can be successively encrypted with the public keys of nodes on the path.

3.5 Summary and Further Reading

3.5.1 Summary

Summary 3.5.1

1. Modern enterprise and web-scale system store petabytes of data. All this data cannot be stored on a single machine. It needs to be stored on a distributed system.

2. Distributed systems for storing data are peer-to-peer (P2P) systems, where the participating nodes are similar and do not have differentiated responsibilities. There can however be a central server in some designs.

3. First-generation P2P systems like Napster relied on a set of central servers.

   (a) Files were stored on individual nodes (peers or clients). Client nodes found the location of files by querying the central servers. After finding the ID of the remote node that stores the file, a connection is initiated between them for transferring the file.

   (b) Such architectures have serious legal issues because all the nodes that store potentially copyrighted content are known to the central server(s).

   (c) The owner of the central server is legally liable.

4. Second-generation P2P networks do not use a central server. They rely on exchanging information exclusively between peers.

5. The first category of networks uses unstructured networks.

   (a) The anti-entropy protocol involves random nodes exchanging their contents in each round. This process has logarithmic complexity.

   (b) At the beginning, a push-based algorithm is more useful and at the end, a pull-based algorithm is more useful.

   (c) Anti-entropy algorithms typically do not have a termination strategy. Rumor mongering algorithms on the other hand reduce their aggressiveness with the progression of time. They sometimes leave a residue – some nodes that remain susceptible.

6. The Gnutella network uses a structured network. There is no central server. Nodes send multicast messages with fixed time-to-live (TTL) fields. If any node has a copy of the requested file or knows the location of the file, it forwards this information to the requesting node.

7. Distributed hash tables or distributed key-value stores are important data structures used in highly-scalable third-generation P2P networks. They use a highly structured arrangement of nodes arranged as a mesh or a circle.

   (a) The Pastry DHT maps every node to a point on an ID circle with 2^{128} points. This is achieved using hashing. A key is mapped to the node whose hash is closest to its hash on the id circle. Each 128-bit hash value is represented using 32 hexadecimal digits. We try to maximize the length of the shared prefix in each step (number of matching hex digits).

   (b) The Chord DHT maps the hash of every key or node to a point on the ID circle. From the point (corresponding to a key) we traverse the ID circle clockwise. The first node that we reach is the one that stores the key.

8. DHTs are used to create scalable third-generation P2P networks.

9. As of 2025, BitTorrent is the most popular file-sharing network. A file is broken into multiple pieces and these pieces are distributed across different peer nodes. It can store and locate files using the mainline DHT, or use a gossip-based protocol.

10. Freenet is a third-generation network that maintains strict secrecy and anonymity. The identities of the sender and receiver are very difficult to discover because all the nodes along the way collaborate to obfuscate information.

3.5.2 Further Reading

A good starting point is the paper by Howe [Howe, 2000]. It introduces and compares Napster and Gnutella. Carlsson et al. [Carlsson and Gustavsson, 2001] chronicle the reasons for the rise and fall of Napster. They introduce a mathematical framework for assessing user behavior in such scenarios. The following references [Douglas, 2004,Scharf, 2011] look at the legal aspects of first-generation P2P networks. They chronicle their impact on society and the reasons for their downfall.

We looked at some simple results in epidemic data dissemination in the

chapter. There are more complex models. For a much deeper treatment of anti-entropy protocols, readers should read the paper by Ozkasap et al. [Özkasap et al., 2010b] and Islam et al. [Islam et al., 2014]. It is possible that even after several rounds of gossiping, all the updates have not been propagated. There is a need for explicit state reconciliation. The paper by Renesse et al. [Van Renesse et al., 2008] is an important reference in this area. There are more scalable versions of traditional gossiping. It is possible to gossip with a randomly chosen subset of peers [Jelasity et al., 2007] or perform a probabilistic broadcast [Eugster et al., 2003].

We covered the most important DHTs in this chapter. However, there is a lot of work that we have not covered. It is important to understand consistent hashing thoroughly first. The best reference is Lewin's PhD thesis [Lewin, 1998]. The associated research paper [Karger et al., 1997] captures the major aspects of the thesis. An important area of work in this space is DHTs that route messages in $\theta(1)$ hops. Two references in this space stand out: Amazon Dynamo [DeCandia et al., 2007] and ZHT [Li et al., 2013]. There are other influential proposals that look at different designs of hash tables [Kaashoek and Karger, 2003, Ghodsi, 2006]. Security is an important problem in this space. There is a fair amount of work in this space. The Whanau paper [Lesniewski-Laas and Kaashoek, 2010] is a representative work in this space.

Questions

Question 3.1. Give an epidemic algorithm (anti-entropy or rumor mongering) to estimate the size of the network for these two scenarios.

1. Assume that any node can communicate with any other node.
2. Assume that a node can communicate with only its neighbors.

Estimate its size, complexity, and analyze the trade-offs.

Question 3.2. Why does the pong message in Gnutella come back along the same path, not directly?

Question 3.3. Why is the average lookup time in Pastry $O(log_{2^b}(N))$? If we are using a k bit hash, shouldn't it be equal to $O(log_{2^b}(2^k)$? Note that while constructing the routing table, we do not have an estimate of the number of nodes in the network.

Question 3.4. The Pastry protocol does not take reliability into account. If a node fails, we lose all the data that it contains. How can you fix this problem in the Pastry protocol? Explain your answer.

Question 3.5. Why is forward progress in Pastry guaranteed?

Question 3.6. Why is it necessary to have two phases in Paxos? Why can't we conclude the result of the consensus algorithm after the first phase itself?

Question 3.7. In Chord, why does the lookup algorithm start from the last finger and proceed towards the first finger. Why not proceed in the reverse order?

Question 3.8. Can you modify the Chord protocol to take the locality of nodes into account. Provide an overview of the algorithms for lookup, insert, and delete.

Question 3.9. Assume that we modify the Chord algorithm. Instead of creating fingers as $id + 1, ..., id + 2^{i-1}, id + 2^i, ...$, we make fingers with powers of K: $id + 1, ..., id + K^{i-1}, id + K^i, ...$ Does setting $K = 2$ yield the most efficient solution? Explain your answer.

Question 3.10. What is the tradeoff between encrypting contents and cacheability in Freenet?

Chapter 4

Mutual Exclusion Algorithms

> *Two stars keep not their motion in one sphere,*
> *Nor can one England brook a double reign, Of*
> *Harry Percy and the Prince of Wales.*

William Shakespeare, King Henry IV, Part I

In a distributed system, we have a plethora of shared resources. Different processes that run concurrently may attempt to acquire and use these shared resources in an arbitrary order. This can lead to *race conditions* where different processes simultaneously try to access or modify the state of such shared resources and such behaviors can potentially lead to errors. As a result, there is a need to ensure that only one process can access any of these critical resources at any given point of time. This property is formally known as *mutual exclusion*. Mutual exclusion ensures correctness from the point of view of the shared resource/object since it provides an illusion of sequential execution. We shall start the discussion in this chapter by looking at some classical mutual exclusion algorithms. Let us introduce the notion of a lock first.

A generic shared resource that can be used to protect other shared resources is known as a *lock*. It allows exclusive ownership by a single process. If a shared resource is associated with a lock, then acquiring the lock is equivalent to acquiring the shared resource. Similarly, releasing the lock is equivalent to releasing the resource. The reasons for using a lock is because it is a bespoke data structure that makes distributed synchronization easier.

Definition 4.0.1 Lock

A generic resource that can be used to restrict access to another shared resource or a set of resources is known as a *lock*. A lock can either be free or be acquired. Furthermore, at any point of time, it can only be acquired by one process. The process that has acquired the lock can freely use the associated resource(s) until it has released the lock.

In all distributed algorithms, we need to prove two basic properties: safety and liveness. *Safety* means that something bad never happens. For example, in a traffic signal, it is never the case that two lights for perpendicular roads are green at the same time. This may lead to a collision. The other important property is *liveness*. This means that something good always happens. Consider the traffic signal example again. If we can guarantee that a light eventually turns green, then this is a good thing – it guarantees forward progress. We can conclude that for a traffic signal to work, we need to guarantee both safety and liveness.

Definition 4.0.2 Safety and Liveness

Safety means that something *bad* never happens in a system. For instance, it is never the case that traffic lights in two perpendicular directions are green at the same time. *Liveness* means that something good always happens. In the example of a traffic intersection, this means that a traffic light always turns green.

In the case of distributed algorithms like mutual exclusion, safety typically means that two processes do not access the lock at the same time. A concurrent access would break the fundamental premise of mutual exclusion. Liveness, in this case, means that every process is guaranteed to get the lock. There can be different kinds of definitions for liveness. We may want to ensure that every process gets the lock in a finite amount of time. Some may argue that this is too open-ended and would like every interested process to get access to the lock in a bounded amount of time. Regardless of the definition of liveness, the basic idea is that there is no *starvation*. Starvation means that it is never the case that a process waits forever while trying to acquire a lock. In practice, timers can be set, whose timeout value depends on the definition of liveness that we are using.

Practically speaking, *starvation freedom* guarantees access to the lock before the user decides that it is too late. Theoretically, the wait time

should either be finite or bounded (depending upon the setting). Starvation freedom implies several things. For instance, it implies that processes cannot *deadlock*, which is defined as a circular wait condition: Process `P` waits on Process `Q`, which waits on Process `R`, so on and so forth, and finally the last process in the list waits on Process `P`. As a result, none of the processes in this circular dependence loop make any progress. It is clear that starvation freedom implies deadlock freedom. The converse is not necessarily true.

Fact 4.0.1

Starvation freedom implies deadlock freedom. The converse is not necessarily true.

Keeping these concepts in mind, let us move forward and introduce distributed algorithms for mutual exclusion in this chapter.

Organization of this Chapter

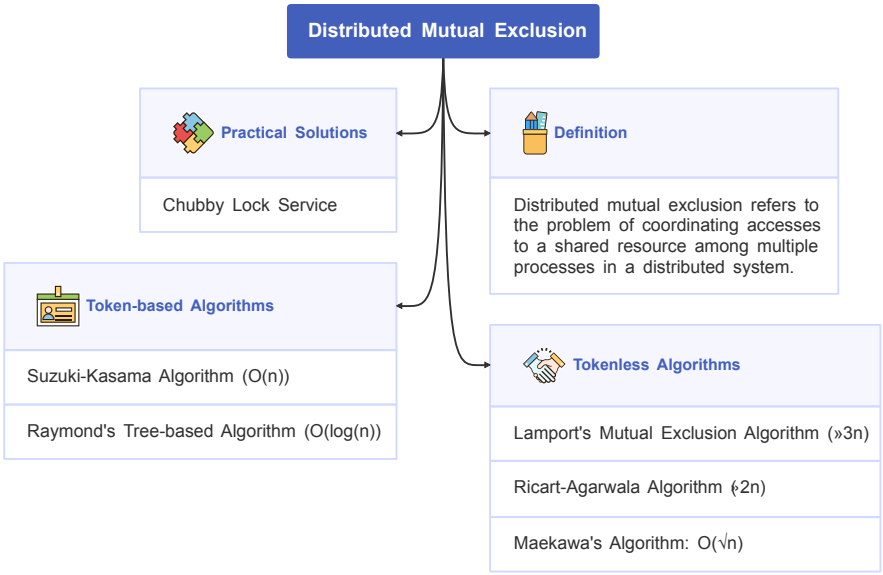


Figure 4.1: Organization of this Chapter

Figure 4.1 shows the organization of this chapter. There are two kinds of distributed mutual exclusion algorithms. The first class of algorithms do

not rely on tokens that are passed between processes. They instead rely on variants of logical clocks or algorithms that rely on causality. In the field of distributed algorithms, we do not rely on classical computational complexity, instead we measure the asymptotic message complexity. This is the number of messages (in big O or θ/Ω notation) per operation.

We will study three algorithms in the first part of this chapter that do not rely on tokens. Their message complexities per lock acquisition gradually reduce with increasing complexity. The straightforward Lamport's algorithm has a message complexity equal to $3(n-1)$, where there are n processes in the system. It reduces to $O(\sqrt{n})$ in the Maekawa's algorithm.

Token-based algorithms have lower message complexities; however, they require a token that cannot be lost. This reduces the fault tolerance of such algorithms. We shall study two algorithms in this space with message complexities $O(n)$ and $O(log(n))$, respectively.

Finally, we will conclude with introducing practical implementations of large systems that require mutual exclusion.

4.1 Tokenless Distributed Algorithms for Mutual Exclusion

In this section, we shall introduce a simple variety of distributed algorithms for ensuring mutual exclusion. We will focus on asynchronous algorithms that do not have faulty processes (see Section 1.3).

4.1.1 Lamport's Mutual Exclusion Algorithm

This is an algorithm with three phases: *request*, *reply* and *release*. The general idea is as follows. Whenever a process wants to acquire a lock, it sends a message to all the n processes (including itself). Every process maintains a queue of requests. It sends a ⟨`reply`⟩ message to a lock request message as soon as it gets it. Once a process has gotten $n-1$ replies that are newer than its request, and no other requests precede its request, it can acquire the lock. After releasing the lock, the process sends a ⟨`release`⟩ message to the rest of the processes. After receiving this message, each process can remove the original request from its request queue and can then proceed with the rest of the requests. This is a very high-level description of the algorithm. Let us now look at the algorithm in its full glory.

Algorithm 11 shows the algorithm. Here, we make several assumptions. We assume that a scalar clock (refer to Section 2.3.1) is used to maintain a

notion of logical time. All such scalar clocks can be made comparable even if they have the same value – we can break ties based on the process id. Furthermore, we assume FIFO channels. This means that all the messages sent from Process `u` to `v` are ordered. It is never possible for a later message to overtake an earlier message and reach `v` sooner.

Algorithm 11 Lamport's mutual exclusion algorithm

1: **procedure** REQUEST(Process $\mathtt{P_u}$)
2: $\mathtt{P_u}$ Sends a $\langle \mathtt{request} \rangle$ message $\langle \mathtt{T_u}, u \rangle$ to all processes
3: It puts this message in its own request queue
4: When $\mathtt{P_v}$ receives a $\langle \mathtt{request} \rangle$ message from $\mathtt{P_u}$, it sends a timestamped acknowledgement
5: **end procedure**
6:
7: **procedure** ACQUIRE(Process $\mathtt{P_u}$)
8: ▷ Acquire the lock if the following two conditions hold, else wait for them to become true.
9: ▷ The original $\langle \mathtt{request} \rangle$ message $\langle \mathtt{T_u}, u \rangle$ is the earliest message in the queue
10: ▷ Process $\mathtt{P_u}$ has received a message with a timestamp greater than $\mathtt{T_u}$ from all the other nodes.
11: **end procedure**
12:
13: **procedure** RELEASE(Process $\mathtt{P_u}$)
14: $\mathtt{P_u}$ removes the original $\langle \mathtt{request} \rangle$ message from its queue
15: Timestamp a $\langle \mathtt{release} \rangle$ message, and send it to rest of the nodes
16: When a process $\mathtt{P_v}$ receives a $\langle \mathtt{release} \rangle$ message from $\mathtt{P_u}$, $\mathtt{P_v}$ removes the corresponding $\langle \mathtt{request} \rangle$ message
17: **end procedure**

Let us consider the function REQUEST. The main aim of this function is to register a request with every process including itself. This bears the local timestamp of $\mathtt{P_u}$. We assume that every process has a queue where it can store requests and their replies in ascending order of their timestamps. Note that since the timestamps are scalar timestamps, they are always mutually comparable – we can decide which one is greater using the process ID as the tiebreaker, even though this many not necessarily imply causality. Every process other than $\mathtt{P_u}$ acknowledges the request by sending a timestamped reply. The timestamp of the reply needs to be strictly greater than the

timestamp of the request (based on the properties of scalar clocks).

The request's timestamp is $\mathtt{T_u}$. When it is received by process $\mathtt{P_v}$, it will update its local time if there is a need. In any case, its updated local time is guaranteed to be at least as large as $\mathtt{T_u} + 1$. The timestamp of the reply will thus be at least $\mathtt{T_u} + 2$. This means that when $\mathtt{P_u}$ receives a reply from any other process, its reply timestamp will at least be $\mathtt{T_u} + 2$. It is of course possible that $\mathtt{P_u}$ gets another message from $\mathtt{P_v}$ with a timestamp greater than $\mathtt{T_u}$ before the reply is sent. We will not distinguish between these two cases.

Process $\mathtt{P_u}$ acquires the lock when two conditions hold, else it keeps waiting. The first condition is shown in Line 9: the original message with timestamp $\mathtt{T_u}$ is the earliest message in the request queue of Process $\mathtt{P_u}$. This means that in the queue of requests, it has the least timestamp. The second condition is listed in Line 10: $\mathtt{P_u}$ should have received a message with a timestamp greater than $\mathtt{T_u}$ from every other process.

Remark 4.1.1

Condition 1 The original lock request message needs to have the lowest timestamp in the request queue of the process trying to acquire the lock.

Condition 2 The lock-acquiring process should have gotten a reply from the rest of the processes with a timestamp greater than that of the original request.

Now, we need to read both these conditions together. The key question that we need to answer is whether it is possible for two processes, $\mathtt{P_u}$ and $\mathtt{P_v}$, to be in the critical section at the same time. The critical section is a piece of code that is executed after acquiring a lock and before releasing it. A process is said to be in the critical section, if it is executing some instruction in the critical section. The second question is whether a process is guaranteed to get the lock once it has made a request. Let us assume that every time the lock is acquired, it is released in a bounded amount of time.

Definition 4.1.1 Critical Section

The *critical section* is a piece of code that is executed after acquiring a lock and before releasing it. A process is said to be in the critical section, if it is executing some instruction in the critical section.

Let us first prove the former statement, which implies mutual exclusion

– no two processes can be in the critical section at the same point of time (see Lemma 3).

Lemma 3

It is not possible for two processes $\mathtt{P_u}$ and $\mathtt{P_v}$ to concurrently access the locked resource at the same time.

Proof: Let us assume to the contrary that the processes are indeed accessing the shared/locked resource at the same time. Process $\mathtt{P_u}$ must have sent a message with timestamp $\mathtt{T_u}$ to $\mathtt{P_v}$. It would have then found out that its request had the least timestamp in its request queue, and it had received a message with timestamp greater than $\mathtt{T_u}$ from $\mathtt{P_v}$.

We need to see what exactly must have happened at $\mathtt{P_v}$. It would have received the message from $\mathtt{P_u}$, set its internal timestamp to at least $\mathtt{T_u} + 1$ and sent a reply. It could not have generated its lock acquire request after this point because its request would then have a timestamp greater than $\mathtt{T_u}$ and Condition 1 will get violated.

Hence, we can conclude that $\mathtt{P_v}$ must have generated its request before it received the $\langle\mathtt{request}\rangle$ message from $\mathtt{P_u}$. The question is whether the timestamp of its request $\mathtt{T_v}$ is greater than $\mathtt{T_u}$ or not. Note that we will never have equality because the process ID will be used to break ties.

Case I: $\mathtt{T_v} > \mathtt{T_u}$. In this case, $\mathtt{P_u}$ would have received the $\langle\mathtt{request}\rangle$ message from $\mathtt{P_v}$ before it received the reply to its own $\langle\mathtt{request}\rangle$ message. Given that $\mathtt{T_v} > \mathtt{T_u}$, it means that $\mathtt{P_u}$ would have sent its $\langle\mathtt{request}\rangle$ message by now, otherwise it would have had a higher timestamp ($> \mathtt{T_v}$). Given that $\mathtt{P_u}$'s request message was on the way, when $\mathtt{P_v}$'s $\langle\mathtt{request}\rangle$ message was received at $\mathtt{P_u}$, and we assume FIFO channels, we can safely conclude that $\mathtt{P_v}$ would have seen $\mathtt{P_u}$'s $\langle\mathtt{request}\rangle$ message before seeing a message from $\mathtt{P_u}$ with a timestamp greater than $\mathtt{T_v}$. This means that before seeing $\mathtt{u}$'s response, $\mathtt{P_v}$ would have seen $\mathtt{u}$'s request. Now, Condition 1 will not succeed at $\mathtt{P_v}$ because $\mathtt{P_u}$'s request has a lower timestamp than its own request (assumption: $\mathtt{T_u} < \mathtt{T_v}$). In other words, before seeing the response of $\mathtt{u}$, $\mathtt{P_v}$ would already have $\mathtt{u}$'s request in its request queue. That request would have a lower timestamp than its request and as a result Condition 1 will not succeed. Hence, unless $\mathtt{P_u}$ releases the lock, $\mathtt{P_v}$'s request cannot go through. It is thus not possible for $\mathtt{P_v}$ to be in the critical section.

Case II: $\mathtt{T_v} < \mathtt{T_u}$. This means that the $\langle\mathtt{request}\rangle$ message was sent by $\mathtt{P_v}$ before it received $\mathtt{P_u}$'s $\langle\mathtt{message}\rangle$. After sending a message timestamped with $\mathtt{T_v}$, $\mathtt{P_v}$ must have sent a message with a timestamp greater than $\mathtt{T_u}$ (because $\mathtt{P_u}$ ended up getting the lock). This means that before Condition

2 started to hold at $\mathtt{P_u}$, it had received $\mathtt{P_v}$'s request, and this message had a lower timestamp that $\mathtt{P_u}$'s original $\langle \mathtt{request} \rangle$ message. Hence, Condition 1 could never have been true at $\mathtt{P_u}$. Hence, $\mathtt{P_u}$ could not have accessed the lock when $\mathtt{P_v}$ was accessing it. In this case, $\mathtt{P_u}$ has to wait for $\mathtt{P_v}$ to release the lock.

Given cases I and II, we can clearly see that concurrent access to the resource is not possible. ■

Proving that there is no *starvation* – a process does not wait forever – is easy. We need to note that Lamport timestamps increase monotonically. They never decrease. As a result, ultimately a request will become the oldest request in the system. Furthermore, the requesting process will eventually get a reply to its request. The reply is guaranteed to have a greater timestamp than the request. This means that Conditions 1 and 2 will ultimately get satisfied and the requesting process will get access to its resource. In fact, we can easily prove that in the worst case a process needs to wait for the rest of the $n-1$ processes to access the resource. At that point, the request from the process will become the oldest in the system. This is easily provable because once a process replies to a request, its subsequent request can only have a greater timestamp.

The Lamport's mutual exclusion algorithm is simple and straightforward. However, proving that it works, especially that it guarantees mutual exclusion is difficult (as we saw in the proof for Lemma 3). The standard approach in all such proofs is the same. We prove by contradiction. We first assume that the property that we are trying to prove is violated. Then, we break the symmetry and consider different cases: ($\mathtt{T_u} > \mathtt{T_v}$ and $\mathtt{T_u} < \mathtt{T_v}$). For each case, we derive a contradiction. Proving starvation freedom typically involves proving that ultimately the starving request becomes the oldest request in the system, and then it will have the highest priority.

Message Complexity

Unlike sequential algorithms, the complexity of distributed algorithms is measured differently. We do not count the number of steps that a process takes. We instead count the total number of messages sent in an asynchronous algorithm and the total number of rounds required to complete the protocol in the case of a synchronous algorithm. The number of messages or rather the message complexity is by far the most common metric. This is because the time that a network message takes is typically much more than the time taken by the distributed application to process the message.

In this case, there are three phases for accessing a resource. In each

phase, a message needs to be sent to the rest of the $n-1$ processes. Thus, we arrive at a total count of $3(n-1)$ messages.

4.1.2 Ricart-Agarwala Algorithm

Is there a need to send a reply acknowledging the receipt of a message immediately? It is actually not required. We can instead send the acknowledgement later on and piggyback it with a release message. This will reduce the total message complexity from $3(n-1)$ to $2(n-1)$.

This algorithm is deceptively simple (see Algorithm 12).

Algorithm 12 Ricart-Agarwala algorithm

```
1:  procedure REQUEST(Process P_u)
2:      P_u Sends a timestamped ⟨request⟩ message to all nodes
3:  end procedure
4:  procedure RECEIVEREQUEST(P_v receives a request from P_u)
5:      if P_v is not holding the lock and not interested in it then
6:          return Timestamped ⟨reply⟩ message
7:      end if
8:      if P_u's ⟨request⟩ timestamp is lower than P_v's and P_v is not holding
    the lock then
9:          return Timestamped ⟨reply⟩ message
10:     end if
11:     Queue the request
12: end procedure
```

The function REQUEST simply sends a timestamped message to all the processes including itself. The logic of the algorithm is embedded in the function RECEIVEREQUEST. There are two conditions that need to be true for sending a timestamped reply. If these conditions are not true, then the request is queued.

Remark 4.1.2

Condition I: If P_v is not interested in the lock or is not currently holding it, then it should not have any objections to P_u acquiring the lock. A timestamped ⟨reply⟩ message can be sent (see Line 6).

Condition II: P_v is interested in acquiring the lock but does not

currently hold it (refer to Line 9). The algorithm makes P_v relinquish its claim if P_u's request has a lower timestamp. In this case also P_v sends a timestamped reply. Essentially, P_v passes the baton to P_u because the latter's request was earlier.

The process of acquiring the lock is similar to the Lamport's algorithm. When a process has received replies from all the nodes (including itself), it is ready to acquire the lock. Subsequently, when the lock is released, the process replies to all pending requests.

The message complexity is $2(n-1)$. The reply message in this case was deliberately delayed. As a result, the algorithm uses fewer messages. Let us now look at the proof.

Proof of Correctness

We shall follow the same approach here as well. Assume that processes P_u and P_v acquire the lock concurrently. Let P_u have the lower request timestamp. This basically means that after P_u generated its request, it must have gotten P_v's request. Otherwise, it would have had a higher request timestamp – it would have incorporated the time information of P_v and increased the value of its Lamport clock.

This means that as per Line 9, P_u could not have sent a reply to P_v. It would have been aware that it is interested in acquiring the lock and its request timestamp is lower than that of P_v's. As a result, P_v would not have gotten any timestamped reply from P_u. This means that P_v could not have acquired the lock, which leads to a contradiction.

The proof of starvation freedom is similar to that of Lamport's mutual exclusion algorithm.

4.1.3 Maekawa's Algorithm

Can we reduce the message complexity even further? Can it be made $O(\sqrt{n})$? This would require some new insights. Let us take a second look at the proof of the Ricart-Agarwala algorithm and explain its high-level picture. The proof hinges on the fact that there is a Process P_u, which somehow impedes the progress of another Process P_v. A process *impedes* another by delaying its reply. In the algorithms that we have seen, if P_u's request's timestamp has a lower timestamp than the request received by P_v, then P_u delays its reply. The insight that we can derive is that the process that holds back the reply to P_v performs a key synchronizing function (ordering

one process after the other). It does not allow concurrent lock accesses by deliberately delaying one process.

Hence, note that for any two requests, all that we need to synchronize them is *just one process*. This process holds something akin to veto power. It may choose to reply to one lock-requesting process and delay the reply to the other.

Let us develop this idea further. Let us associate a set of processes with a given process $\texttt{P}_\texttt{u}$. This set is known as its *request set* R_u. Informally, R_u refers to the processes that $\texttt{P}_\texttt{u}$ requests for acquiring the lock. Similarly, Process $\texttt{P}_\texttt{v}$ shall have its corresponding request set R_v. The key property that we need to ensure is that for any given $\texttt{P}_\texttt{u}$ and $\texttt{P}_\texttt{v}$, $R_u \cap R_v \neq \phi$ (not null). The intersection of any two request sets needs to always be non-null – there is at least one process in common. An easy method of constructing such a request set is shown in Figure 4.2. Here, we arrange all the processes in a square grid. The request set of a process is defined as the set of processes in its row and column in the grid. It is easy to see that the request sets of two different processes will always have two processes in common. This approach uses a request set with size $2\sqrt{n} - 1$ (counting the process that is requesting the lock). It turns out that we can do much better and bring this number down to roughly $\sqrt{n}$, if we use results from field theory. For the sake of simplicity, let us just assume that the number of processes in a request set is equal to $O(\sqrt{n})$.

The summary of this discussion is that given any two processes, the intersection of their request sets is always non-null. In this case, instead of sending messages and expecting replies from all processes, the lock requester needs to only ask its request set. Given that there will always be a common process between two request sets, this (common) process can ensure mutual exclusion. Given that we are not considering faulty or inactive processes, the common process will always participate in the algorithm.

Acquiring the Lock

Let us first look at acquiring the lock. The philosophy is the same as in earlier algorithms. We send a request to the request set, which in this case is much smaller, and wait for the replies. Given that the request set is much smaller than the total number of processes, there are additional complexities. The problem is that just delaying a reply is not always the best option because it can lead to deadlocks – there is a need to thus send additional messages.

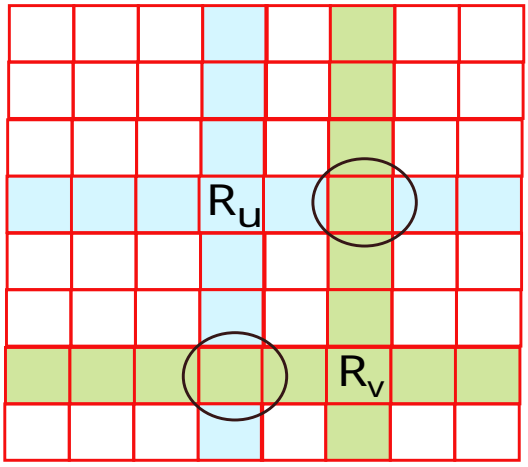


Figure 4.2: Arrangement of the processes in a matrix. The figure shows two overlapping request sets for the requests R_u and R_v (resp.).

Algorithm 13 Maekawa's algorithm: Acquiring a lock (Part I)

1: **procedure** REQUEST(Process $\texttt{P}_\texttt{u}$)
2: $\texttt{P}_\texttt{u}$ sends a timestamped $\langle\texttt{request}\rangle$ message to all the nodes in R_u (including itself)
3: Wait till a $\langle\texttt{locked}\rangle$ message is received from all the processes in the request set, then acquire the resource
4: **end procedure**

Algorithm 14 Maekawa's algorithm: Acquiring a lock (Part II)

```
1: procedure receiveRequest(P_v receives a request from P_u)
2:     if P_v is not locked then
3:         P_v marks itself as locked
4:         return ⟨locked⟩ message to P_u
5:     else
   ▷ This means that P_v is already locked by a request sent from P_w
6:         Place P_u's request in the request queue
7:         if P_u's request is not the earliest request in the queue then
8:             Send a ⟨failed⟩ message to P_u
9:         else
10:            Send an ⟨inquire⟩ message to P_w
11:        end if
12:    end if
13: end procedure

14: procedure receiveInquire(P_w receives an ⟨inquire⟩ message from
    P_v)
15:    if P_w has already received a ⟨failed⟩ message or is not interested
    anymore then
16:        Send back a ⟨relinquish⟩ message
17:    else
18:        Defer the reply
19:    end if
20: end procedure

21: procedure receiveRelinquish(P_v receives a ⟨relinquish⟩ message
    from P_w)
22:    Add the request from P_w to the queue of waiting requests
23:    Find the earliest request in the queue; send its sender a ⟨locked⟩
    message
24: end procedure
```

The REQUEST function broadly looks the same (see Algorithm 13). The requesting process $\mathtt{P_u}$ sends a timestamped ⟨`request`⟩ message to all the processes in its request set (including itself). Note that every process is also a member of its own request set. However, the way that a receiving process reacts to this message is different. The receiving process $\mathtt{P_v}$ first checks if it is locked in Line 2 of Algorithm 14. The term "locked" here basically

means "committed". This means that process $\texttt{P}_\texttt{v}$ has allowed some process (that may include itself) to acquire the lock (access the resource). It has *allowed* such a process by sending a reply to its request. Either that process is waiting for a lock or has already acquired the lock. Clearly, if $\texttt{P}_\texttt{v}$ is not locked, then it can send a message to $\texttt{P}_\texttt{u}$ with an "all-clear" from its side. In other words, $\texttt{P}_\texttt{u}$ can assume that there are no objections from $\texttt{P}_\texttt{v}$'s side. This is conveyed using a $\langle\texttt{locked}\rangle$ message. At this point $\texttt{P}_\texttt{v}$ has basically given a commitment to $\texttt{P}_\texttt{u}$. It will not send a reply to any other process, unless its commitment is somehow reversed (we shall discuss this shortly).

Let us consider the other case where $\texttt{P}_\texttt{v}$ has given a commitment (sent a $\langle\texttt{locked}\rangle$ message) to another process $\texttt{P}_\texttt{w}$. The first thing to do is to queue the message from $\texttt{P}_\texttt{u}$. Note that here a deadlock scenario is possible: process `P` waits on process `Q`, `Q` waits on `R`, and finally `R` waits on `P`. As a result, there is a need to do something other than plain old-fashioned waiting.

In Line 8, we check if the request from $\texttt{P}_\texttt{u}$ is the earliest in $\texttt{P}_\texttt{v}$'s queue. If it is not the earliest (lowest timestamp), then it means that the request from $\texttt{P}_\texttt{u}$ does not have the highest priority. As a result, a $\langle\texttt{failed}\rangle$ message is sent to $\texttt{P}_\texttt{u}$. It is an instantaneous negative acknowledgement. In case, if it is the earliest message, then there is a need to tell $\texttt{P}_\texttt{w}$ that a higher priority message has arrived. $\texttt{P}_\texttt{v}$ thus sends a message to $\texttt{P}_\texttt{w}$ inquiring if it is done (in Line 10). The idea is to nudge $\texttt{P}_\texttt{w}$ to relinquish its claim to the resource. $\texttt{P}_\texttt{w}$ calls the receiveInquire message when it receives an $\langle\texttt{inquire}\rangle$ message from $\texttt{P}_\texttt{v}$.

In the receiveInquire function, $\texttt{P}_\texttt{w}$ checks if it has already received a $\langle\texttt{failed}\rangle$ message, or it is not interested in the lock anymore because it has used the resource. In such a situation, it sends back a $\langle\texttt{relinquish}\rangle$ message. In either case, it means that $\texttt{P}_\texttt{w}$ is not the first in line to acquire the resource. Either there are no competitors or there are competitors with earlier requests. In any case, the $\langle\texttt{relinquish}\rangle$ message indicates that $\texttt{P}_\texttt{w}$ is allowing $\texttt{P}_\texttt{v}$ to go ahead. If either of these conditions are not true, then the reply is deferred.

Let us next discuss the receiveRelinquish function. Once $\texttt{P}_\texttt{v}$ receives the $\langle\texttt{relinquish}\rangle$ message, it knows that its $\langle\texttt{locked}\rangle$ message to $\texttt{P}_\texttt{w}$ stands invalid. It is free to send a $\langle\texttt{locked}\rangle$ message to another process. It thus adds $\texttt{P}_\texttt{w}$'s request to its queue of waiting requests, and chooses the earliest request in the queue and sends a $\langle\texttt{locked}\rangle$ message to it. This could be $\texttt{P}_\texttt{u}$'s request or could be a request from some other process (received in the time being).

$\texttt{P}_\texttt{u}$ in the meanwhile waits till it has gotten a $\langle\texttt{locked}\rangle$ message from all the processes in its request set (see Line 3, Part I). Once all the $\langle\texttt{locked}\rangle$

messages have been received, the resource can be acquired.

Releasing the Lock

Algorithm 15 Maekawa's algorithm: Releasing the lock

1: **procedure** RELEASE(Process $\texttt{P}_\texttt{u}$)
2: It sends ⟨`release`⟩ messages to all the processes in its request set.
3: Once another process gets the ⟨`release`⟩ message, it locks the earliest message in its request queue or marks itself unlocked if the queue is empty.
4: **end procedure**

Algorithm 15 shows the logic for releasing the lock. $\texttt{P}_\texttt{u}$ sends a ⟨`release`⟩ message to rest of the processes. Each process then initiates the locking procedure – mark itself as locked and send the ⟨`locked`⟩ message – for the earliest message in its request queue. If its request queue is empty, then the process marks itself as unlocked.

This part is quite straightforward bookkeeping. All the processes in the request set delete $\texttt{P}_\texttt{u}$'s request. They remove/reset all the state associated with $\texttt{P}_\texttt{u}$'s request. The processes in the request set are subsequently free to lock themselves for other requests. Subsequently, a fresh iteration of the algorithm is started.

Proof of Mutual Exclusion

Assume that two processes $\texttt{P}_\texttt{A}$ and $\texttt{P}_\texttt{B}$ acquire the lock concurrently. It is not possible for the common node $\texttt{P}_\texttt{C}$ to send a ⟨`locked`⟩ message to both the nodes at the same time. Without loss of generality, it would have sent a ⟨`locked`⟩ message to one process $\texttt{P}_\texttt{A}$ and marked itself as locked. Hence, it could not have subsequently sent a ⟨`locked`⟩ message to $\texttt{P}_\texttt{B}$. This means that it is not possible for two processes $\texttt{P}_\texttt{A}$ and $\texttt{P}_\texttt{B}$ to hold the lock at the same time.

4.1.4 Proof of Starvation Freedom

Consider process $\texttt{P}_\texttt{A}$, whose request is the earliest in the system. If it gets the lock in a finite amount of time, then there is no problem. However, let us consider the case when it waits indefinitely.

It means, it is waiting for some process $\texttt{P}_\texttt{B}$ to reply with a $\langle\texttt{locked}\rangle$ message. $\texttt{P}_\texttt{B}$ is not sending the $\langle\texttt{locked}\rangle$ message because of one of several reasons.

$\texttt{P}_\texttt{B}$ may be using the resource itself. In this case, it will finish using the resource soon (our assumption) and then it will scan its request queue. Since $\texttt{P}_\texttt{A}$'s request is the earliest, $\texttt{P}_\texttt{B}$ will send it the $\langle\texttt{locked}\rangle$ message. This is not a case of waiting because within a *short* period of time, $\texttt{P}_\texttt{A}$ will get the $\langle\texttt{locked}\rangle$ message that it needs from $\texttt{P}_\texttt{B}$.

Consider the other case, where $\texttt{P}_\texttt{B}$ has sent a $\langle\texttt{locked}\rangle$ message to $\texttt{P}_\texttt{C}$. In this case, $\texttt{P}_\texttt{B}$ sends an $\langle\texttt{inquire}\rangle$ message to $\texttt{P}_\texttt{C}$. If $\texttt{P}_\texttt{C}$ is holding the lock, it defers. However, we are not counting such *short* periods of time. The case that we are interested in, is when $\texttt{P}_\texttt{C}$ defers indefinitely. This means that it has not gotten a $\langle\texttt{failed}\rangle$ message from any process in its request set. The following question arises. If $\texttt{P}_\texttt{C}$ has not received a $\langle\texttt{failed}\rangle$ message, then why has it not gotten access to the lock yet?

This further means that there is at least one process $\texttt{P}_\texttt{D}$ that has not sent a $\langle\texttt{locked}\rangle$ message to $\texttt{P}_\texttt{C}$. Now, $\texttt{P}_\texttt{C}$'s request is the earliest at $\texttt{P}_\texttt{D}$ because $\texttt{P}_\texttt{C}$ has not gotten back a $\langle\texttt{failed}\rangle$ message from $\texttt{P}_\texttt{D}$. At $\texttt{P}_\texttt{D}$, the same logic holds. It is waiting for another process $\texttt{P}_\texttt{E}$ to reply with a $\langle\texttt{relinquish}\rangle$ message and $\texttt{P}_\texttt{E}$ is not sending it.

Let us now quickly take a look at our processes. $\texttt{P}_\texttt{A}$, $\texttt{P}_\texttt{C}$ and $\texttt{P}_\texttt{E}$ are lock requesters. $\texttt{P}_\texttt{B}$ and $\texttt{P}_\texttt{D}$ are processes in the request sets of $\texttt{P}_\texttt{A}$ and $\texttt{P}_\texttt{C}$, respectively. Furthermore, $\texttt{P}_\texttt{A}$'s request is earlier than $\texttt{P}_\texttt{C}$'s request, and $\texttt{P}_\texttt{C}$'s request is earlier than $\texttt{P}_\texttt{E}$'s request. This process cannot continue indefinitely. We have a finite number of processes and ultimately a situation will arise where we will observe that $\texttt{P}_\texttt{A}$'s request is earlier than itself, which is not possible. Hence, there is a contradiction here and there will be no circular wait for an indefinite period.

We thus observe there will be no deadlocks and all the messages will be replied to within a finite period of time. **There will be no indefinite waiting**. This means that it will never be the case that a reply is deferred for perpetuity. All lock requests will get a reply in finite time, which will either be $\langle\texttt{locked}\rangle$ or $\langle\texttt{failed}\rangle$. Given that we are armed with this result, let us now prove that the system is starvation free.

Starvation Freedom: Let us prove this by contradiction. Consider a process $\texttt{P}_\texttt{A}$ that starves. This means that for its $\langle\texttt{request}\rangle$ message, it gets at least one $\langle\texttt{failed}\rangle$ message. Then it does not get a $\langle\texttt{locked}\rangle$ message from the corresponding process in the request set. Consider a process $\texttt{P}_\texttt{B}$ in its request set. The request for $\texttt{P}_\texttt{A}$ can at the most be preceded by $\eta - 1$ processes in the queue of pending requests at $\texttt{P}_\texttt{B}$. η is the size of the request

set. In our construction, $\eta = 2\sqrt{n} - 1$, whereas it can be reduced to $O(\sqrt{n})$. The reason for this limit is that it is not possible for the same process's request to precede the request made by $\mathtt{P_A}$ twice at any process in $\mathtt{P_A}$'s request set. Let us explain. Consider a process $\mathtt{P_C}$. Assume that $\mathtt{P_A}$'s original request preceded $\mathtt{P_C}$'s request at some process $\mathtt{P_B}$. Assume that $\mathtt{P_C}$'s request is ultimately granted (before $\mathtt{P_A}$'s). $\mathtt{P_C}$ then issues one more request. It is not possible for $\mathtt{P_A}$'s original request to also precede the second request made by $\mathtt{P_C}$. From the point of view of common sense, this is a fair requirement. Otherwise, $\mathtt{P_A}$ will continue to get discriminated because fresh requests made by other processes will have lower timestamps, and thus $\mathtt{P_A}$'s request may never become the earliest in the system. Let us look at the proof.

From the problem statement, we can assume that when $\mathtt{P_A}$'s request with timestamp τ arrives at $\mathtt{P_B}$, the request from $\mathtt{P_C}$ is already there in $\mathtt{P_B}$'s request queue. At a later point of time a $\langle\mathtt{locked}\rangle$ message is sent to $\mathtt{P_C}$. The timestamp of this message is greater than the timestamp of the message received from $\mathtt{P_A}$, τ (by the property of Lamport clocks). This means that the next request from $\mathtt{P_C}$ will have a timestamp greater than τ. It is thus not possible for another process $\mathtt{P_C}$ to generate request after request in a bid to stop $\mathtt{P_A}$'s request.

Ultimately after servicing a maximum of $\eta - 1$ requests, the request from $\mathtt{P_A}$ will become the earliest in the queue of pending messages at $\mathtt{P_B}$. It will then need to be sent a $\langle\mathtt{locked}\rangle$ message. For a similar reason, $\mathtt{P_A}$ will get $\langle\mathtt{locked}\rangle$ messages from the rest of the processes in the request set, and thus $\mathtt{P_A}$ will get the lock. Hence, there is no starvation.

Message Complexity

Let us assume that the size of the request set is η. As we have mentioned earlier, η can be made equal to approximately $\sqrt{n}$, whereas in our system it is $2\sqrt{n} - 1$. For every request, a maximum of $(\eta - 1)$ $\langle\mathtt{request}\rangle$ messages are sent, $(\eta - 1)$ $\langle\mathtt{locked}\rangle$ messages are received, $(\eta - 1)$ $\langle\mathtt{failed}\rangle$ messages may be received, $(\eta - 1)$ $\langle\mathtt{inquire}\rangle$ messages may be sent, and finally $(\eta - 1)$ $\langle\mathtt{relinquish}\rangle$ messages may be received. Thus, the total message complexity is limited to $5(\eta - 1)$, which is $O(\sqrt{n})$.

Remark 4.1.3

The Lamport's algorithm requires $3(n - 1)$ messages, the Ricart-Agarwala algorithm requires $2(n - 1)$ messages and the Maekawa's algorithm requires $O(\sqrt{(n)})$ messages.

4.2 Token-based Distributed Algorithms for Ensuring Mutual Exclusion

Let us now look at a set of algorithms that explicitly use a *token*. The token is passed from process to process. The owner of the token is deemed to be the holder of the lock. This is a simple method of ensuring mutual exclusion. It is also quite easy to prove that the property of mutual exclusion is never violated.

4.2.1 Suzuki-Kasami Algorithm

Assume that every process maintains a sequence number – its request id. A request is of the form (u, m). This means that process $\texttt{P}_\texttt{u}$ wants to access the lock for the m^{th} time. Let $\texttt{P}_\texttt{u}$ also maintain an array $seq_u[1 \ldots n]$, where n is the number of processes. Here, $seq_u[v]$ is the largest sequence number received from process $\texttt{P}_\texttt{v}$. This sequence number works like a Lamport clock. When $\texttt{P}_\texttt{u}$ receives (v, m), it does the following:

$$seq_u[v] = max(seq_u[v], m) \tag{4.1}$$

The job of this array is to maintain the latest state of $\texttt{P}_\texttt{v}$ (from the point of view of process $\texttt{P}_\texttt{u}$).

The token contains the following pieces of information:

- A queue (Q) of requesting sites.
- An array of sequence numbers, C. $C[u]$ is the latest request that $\texttt{P}_\texttt{u}$ completed.

The token is stateful. It contains a queue of pending requests. This maintains the ordering between requests. It also contains the current state of completed requests across the system.

4.2.2 Acquiring the Lock

In Line 2 of Algorithm 16, we update the local clock of process $\texttt{P}_\texttt{u}$. Note that the entire algorithm is described from the point of view of $\texttt{P}_\texttt{u}$. In this case, it increments $\texttt{seq}_\texttt{u}[\texttt{u}]$. The new value is stored in `val`. This is sent to all the processes. When a different process $\texttt{P}_\texttt{v}$ receives $\langle \texttt{u}, \texttt{val} \rangle$, it updates its internal sequence number as per Equation 4.1.

Algorithm 16 Suzuki-Kasami Algorithm: Lock acquire

```
1: procedure ACQUIRE(Process P_u)
2:     val ← (++ seq_u[u])
3:     Send ⟨u, val⟩ to all the processes
4:     Acquire the lock, when P_u has the token
5: end procedure

6: procedure RECEIVEACQUIRE(P_v receives ⟨u, val⟩)
7:     seq_v[u] ← max(seq_v[u], val)              ▷ update the local counter
8:     if Token.Q is empty, and the token is there with P_v then
9:         if seq_v[u] = token.C[u] + 1 then       ▷ Check for staleness
10:            Send the token to P_u
11:        end if
12:    end if
13: end procedure
```

There is a fast path here (straightforward path). If $\texttt{P}_\texttt{v}$ has the token and the queue of processes in the token (`Q`) is empty, then the token can be sent to $\texttt{P}_\texttt{u}$ subject to a condition. It should not be the case that $\texttt{P}_\texttt{v}$ is reacting to a stale message. This means that $\texttt{P}_\texttt{u}$ has already used the token (acquired the lock) and one of its old requests is pending with $\texttt{P}_\texttt{v}$. There should be a method to adjudge the recency of the request. This is done using the check $\texttt{seq}_\texttt{v}[\texttt{u}] = \texttt{C}[\texttt{u}] + 1$. This means that the token has not been used by $\texttt{P}_\texttt{u}$ and $\texttt{P}_\texttt{v}$ rightly believes that $\texttt{P}_\texttt{u}$ hasn't used the lock and thus should get the token. The token is thus passed over to $\texttt{P}_\texttt{u}$. Recall that $\texttt{C}[\texttt{u}]$ is the sequence number of the latest lock acquisition request that $\texttt{P}_\texttt{u}$ has completed. This means that if $\texttt{seq}_\texttt{v}[\texttt{u}]$ is equal to $\texttt{C}[\texttt{u}] + 1$, it is a fresh request.

Sadly, most of the time the token queue will not be empty. There will be other contenders for the lock. This means that they will have entries in `token.Q` (the queue `Q` in the token).

Let us now look at the algorithm for releasing the lock (see Algorithm 17). It is important to keep the state of the token synchronized with the state of the process. It could have gotten requests from other processes while the token was in transit. For every process, the following relationship is checked: $\texttt{seq}_\texttt{u}[\texttt{v}] = \texttt{token.C}[\texttt{v}] + 1$. This means that for Process $\texttt{P}_\texttt{v}$, there is a pending request. This is added to the queue, in case it is not there. In case the request is already there, then there is no effect.

The token can now be passed to the earliest request in the queue. If there is such a request, then the token can be passed to it.

Algorithm 17 Suzuki-Kasami Algorithm: Lock release

```
1: procedure RELEASE(Process P_u)
2:     token.C[u] ← seq_u[u]                    ▷ Update the state of the token
3:     ▷ Check whether P_v is a contender for the lock
4:     ∀v, add P_v to Q if seq_u[v] = token.C[v] + 1

5:     ▷ Dequeue the request with the earliest timestamp
6:     Dequeue P_w from Q and send the token to P_w
7: end procedure
```

Proof

Given that we have a single token that passes from process to process, proving mutual exclusion is simple. We have a single copy of the token. Whichever process needs to get the lock, it needs to first get access to the token. After that, it can access the resource. Because of this, it is never possible to violate mutual exclusion. Two processes will not have the token at the same point of time.

Let us look at starvation. If a process is interested to get access to the resource, is it guaranteed to get the lock within finite time?

Whenever a process is interested in acquiring the lock, it sends a message to all the processes. The token maintains a queue (`token.Q`), which is an ordered list. Every request is guaranteed to get into the token's queue. The process that holds the token will add the request to the token's queue as soon as it gets the chance. There is a check for staleness to ensure that a request is not added to the token's queue after it has been serviced.

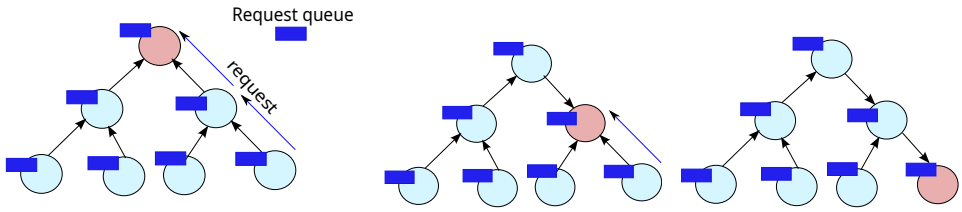
Once, the request is added to `token.Q`, it will gradually get promoted and will ultimately become the head of the queue. At this point of time, the requesting process is guaranteed to get the lock.

Message Complexity

There are two phases of the algorithm. $n-1$ messages are sent to the rest of the processes in the "lock acquire" stage. The token is then sent back to the requesting process once the corresponding request reaches the head of the queue. Thus, we have a total of n messages.

4.2.3 Raymond's Tree Algorithm

Let us now consider a far more scalable version of this algorithm. Let us reduce the message complexity even further from $O(n)$ to $O(log(n))$. This will require a tree-like structure. Basically, we need to organize all the processes as a balanced tree. Any balanced tree will have $O(log(n))$ levels. The idea is to pass the token between nodes between adjacent levels of the tree, and guarantee all the desirable safety and liveness properties.



(a) Token at the root (b) Token on the way (c) Token at the destination

Figure 4.3: Different scenarios in the Raymond's tree algorithm

Figure 4.3(a) shows one such scenario. In this case, the token is present with the root of the tree. The root is shown with a dark shade (dark pink). Every node points to its parent. Ultimately, the pointers are used to reach the root (refer to the arrows in Figure 4.3(a)). Moreover, every node has a queue of requests.

The interesting part of the algorithm is shown in Figures 4.3(b) and 4.3(c). The token moves from node to node across adjacent levels based on the pattern of requests. The token holder is always the root. The pointers orient themselves accordingly.

Remark 4.2.1

The token holder in the Raymond's tree algorithm is always the root node. The token moves between adjacent nodes across levels. The pointers reorient themselves accordingly.

Algorithm 18 Raymond's tree algorithm: send a lock request

```
1:  procedure sendRequest(Process P_u)
2:      if P_u is the root and the request queue is empty then
3:          Grab the lock
4:      else
5:          P_u adds itself to its request queue
6:          if P_u.alreadyForwarded = false then
7:              Forward a ⟨request⟩ message to its parent
8:              P_u.alreadyForwarded ← true
9:          end if
10:     end if
11: end procedure

12: procedure receiveRequest(P_v receives a message from P_u)
13:     if The request queue at P_v is empty and P_v has the token then
14:         Forward the token to P_u
15:     else
16:         Add the request from P_u to the request queue (of P_v)
17:         if P_v.alreadyForwarded = false ∧ P_v is not the root then
18:             Send a ⟨request⟩ message to P_v's parent
19:             P_v.alreadyForwarded ← true
20:         end if
21:     end if
22: end procedure
```

In the subsequent discussion, assume that the requesting process is P_u. The function sendRequest does the following. First, it checks if it is the root and there are no pending requests in its request queue. If this is indeed the case, then it grabs the lock and the process terminates there. Otherwise, P_u adds the request to its own resource queue. This is because this is a pending request that will get satisfied later. Then, the request is forwarded to the parent (one step towards the root). Recall that all the pointers *point* towards the root. It is possible, as we shall see later, for a process to forward messages on behalf of other processes (in its subtree). If a message has already been sent (`alreadyForwarded` = **true**), then there is no need to send a message to get the token again. There is an outstanding request already.

When the parent node gets a ⟨`request`⟩ message, it invokes the function receiveRequest. If the parent's request queue is empty, and it has the

token (is the root), then this situation is quite straightforward. All that the parent needs to do is forward the token to the requester – Process P_u in this case.

In the other case, if the parent is not the root (token holder), then it should forward the request in the direction of the root (to its parent). Of course, it is possible that the parent has already sent a request to obtain the token – on behalf of some other request. In this case, there is no need to send another request. Otherwise, it needs to send a new request in the direction of the root to obtain the token (refer to Line 18 in Algorithm 18). This process continues till we reach the root. Note that in every step, we get one step closer to the root; hence, this process is guaranteed to terminate.

Let us now look at the next procedure, RELEASETOKEN. During this period, it is possible that several requests would have arrived at the root node and the requests would be queued in its request queue. The root node dequeues the first request and sends the token to it. Note that requests can only arrive from children of the root. The pseudocode is shown in Algorithm 19.

When a process finishes executing the critical section (using the resource), it releases the token. The first step is to check the request queue. If there are no entries, then there is nothing to do. We need to just wait for a request to arrive. However, if there are entries, then there is a need to dequeue the earliest entry (head of the queue) and pass the token to it (see Line 3 in Algorithm 19). This node now becomes the new root. It becomes the token holder. Recall that it used to be a child node of the erstwhile root. Assume that process P_R forwards the token to P_C. P_R was the old root and P_C is the new root. P_R updates its parent pointer. It now points to P_C.

When a process receives a token, it invokes the function RECEIVETOKEN. We first set `alreadyForwarded` to *false*. This means that the response to the token request has been received. Next, we check the request queue. If the request queue is empty, there is nothing to do.

Let us assume that the request queue is non-empty. P_v needs to dequeue the head of the queue. This entry could be P_v itself. In this case, P_v can access the resource (enter the critical section). Once it is done using the resource, it needs to dequeue its request queue again. If the queue is empty, then there is nothing to do. It can return from the RECEIVETOKEN function and just wait. Consider the other case, when there is an entry in the request queue of P_v. We dequeue the entry. P_v forwards the token to the dequeued entry and updates its parent pointer (same was done in the function RELEASETOKEN).

Now, it is possible that there could be other requests in P_v's request

Algorithm 19 Raymond's tree algorithm: release the token

```
1: procedure releaseToken(Process P_u)
2:     Forward the token to the process that is at the head of the request queue
3:     Dequeue the entry
4:     Update the parent pointer
5: end procedure

6: procedure receiveToken(P_v receives the token from P_u)
7:     P_v.alreadyForwarded ← false
8:     If the request queue is empty return
9:     Dequeue an entry from the request queue
10:    if P_v was the head of the request queue then ▷ P_v is itself interested
11:        Grab the lock and access the resource
12:        After using the resource, if the request queue is non-empty then return and wait
13:    else
14:        Dequeue the request queue again.
15:    end if
16:    Forward the token to the dequeued entry. Update the parent pointer.
17:    if P_v's request queue is still non-empty then
18:        Send a fresh request for the token (to the current token holder)
19:        P_v.alreadyForwarded ← true
20:    end if
21: end procedure
```

queue. Given that it has forwarded the token, it needs to get it back to help in serving those requests. Therefore, there is a need to issue a fresh request for the token in Line 18 (in Algorithm 19) for the token. We again set $\texttt{P}_\texttt{v}$.`alreadyForwarded` to *true*. This is because it is anticipating that it will get the token in the future. Moreover, it cannot avoid sending more messages in case it gets other requests from downstream nodes.

Proof of Correctness

The proof of mutual exclusion is trivial. There is only one token, and that can only be held by one process at any given point of time. Hence, it is not possible for two processes to be in the critical section at the same time.

Let us now look at the liveness properties. Ultimately a request from process $\texttt{P}_\texttt{w}$ will become the head of all the request queues – when it becomes the oldest in the system. At that point of time, the token will need to come back to the process $\texttt{P}_\texttt{w}$, which had initially placed that request. This is because of the way we process received tokens. We always dequeue the request queue and act on the dequeued request first. We never reorder requests within a queue. This means that even if the request from $\texttt{P}_\texttt{w}$ has been starving, the moment it becomes the oldest in the system, the token will have to come to it.

Message Complexity

Let us consider a conservative worst case by looking at each phase of execution. The $\langle$`request`$\rangle$ message is sent over $O(log(n))$ links. For each request, a reply is sent by sending the token towards the original requester. This accounts for $O(log(n))$ more messages.

Sadly, more messages need to be sent because it is possible that there are other requests that have a higher priority. The token will be sent to them and a request needs to be sent to get the token back. Here, we need to carefully count the number of messages (see Line 18 in Algorithm 19). Whenever, the token is sent in a certain direction, it is because there is a request in that direction that has the highest priority in the request queue of the token sender. Consider an example.

Process `P` needs the token. It sends a request to Process `Q`, which is the current token holder. `Q` sends the token back in the direction of `P`. However, a node in the middle running Process `R` finds that there is a request from another process `S` that is earlier. It thus sends the token in the direction of `S`. Given that it already has a request that was originally initiated by node

P, it sends a fresh token request to get the token back (after S uses the token and possibly other nodes). Now, the question is how do we account for this message that Process R sends to get the token back? We can charge this message to the account of Process S. We can assume that whenever a token is sent in the direction of some process, then we also add the overhead of one request to get the token back. This simplifies things. In the path from Q back to P, there will be several such nodes that will send the returning token in other directions because they will have earlier requests from processes like S. However, in every such case the request that is made to get the token back is accounted for by a different process (not P).

We only add the messages that account for directly passing the token from Q to P, and 1 request per link (to get the token back), to Process P's message complexity.

Hence, the total overhead is limited to $O(log(n))$.

Remark 4.2.2

The Suzuki-Kasami algorithm requires $O(n)$ messages and the Raymond's tree algorithm requires $O(log(n))$ messages.

4.3 A Practical Locking Service

Let us now propose the design of a large-scale conceptual implementation of a locking system in a practical context. We will be loosely inspired by Google's Chubby lock service [Burrows, 2006]; however, our design will be simpler and also have non-trivial differences.

To start with, one may argue that in general it is a bad idea to have locks in very large systems that are loosely coupled. Sadly, they are a necessary evil. Sometimes in cases such as distributed file systems, there is a need for locks, especially if we have strict file-access semantics. They are useful for leader election as well – the process that gets the lock first is the leader. Let us thus be more invested with practical concerns in this section and look at this problem from an implementation perspective.

There are a few important research questions that need to be answered first. Should the lock service be mostly centralized or fully distributed? Given that we are reading a book on distributed systems, one may be tempted to go for the latter answer. However, given the ease of implementation, practical concerns would point towards the former solution given that it is easier to implement. Of course, we cannot rely on a single server

otherwise it is a single point of failure. It is thus a wiser idea to rely on a single centralized service, which internally comprises numerous servers. Note that there is still a single centralized service and a master server that has a special position as compared to the rest of the servers. This is a case of limited centralization, where there is a centralized service that is internally a distributed system. Here, availability is the key. This means that at all times the lock service should be responsive, and it should not go down.

Next, do we choose coarse-grained or fine-grained locking? Practical concerns again dictate that we should opt for coarse-grained locking. Lock access requests will be relatively infrequent, and the system will scale much better. A counter-argument could be that many processes interested in the lock may have to wait for a long time. This is why we shall use reader-writer locks. They allow concurrent reads at a time but for obvious reasons allow only a single writer. The protocol is inspired by classical invalidation-based cache coherence protocols in computer architecture (refer to the book on Next-Gen Computer Architecture by one of your authors [Sarangi, 2023]).

4.3.1 Components

A distributed locking service typically has the following components: a primary lock server, secondary lock servers (to add redundancy) and a client-side locking library. Clients communicate with the server using functions in their locking library. The communication is via remote procedure calls (RPC). A remote procedure call is a mechanism of calling a function from a remote node. The arguments and return values are converted to a standardized format and transported across the network. This process is known as marshaling and unmarshaling, respectively.

Let us define a cluster as a set of N (≥ 5) servers that communication between each other. One of them is the master server. The rest are referred to as *replicas*. They run a leader election protocol to elect a master server. A simple way of doing so is to just acquire a specific leader-lock – the server that is the first to acquire it is the leader. We shall look at more efficient leader election algorithms in Section 5.3.1. Once a master has been elected, it is guaranteed to maintain its position for a *lease period*. This basically means that before the end of this period, it cannot be impeached – the replicas will not elect another master. After the period is over, the replicas may initiate another election round to elect a new master. Note that the same master can be re-elected as well.

All the servers – the master and the replicas – maintain information about a large number of locks. The resources that are actually being locked

are not important. Furthermore, we shall differentiate between read locks and write locks. The former allow a client to just read, whereas the latter allow both read and write accesses. Now, all the clients need to send their read and write lock requests to the master who services them. The master runs the protocol, maintains state information and also synchronizes its contents with the replicas; fortunately, the clients are blissfully unaware of this. DNS servers (name servers) provide clients with the IP address (proxy for location) of the master.

In case, the master fails, then the rest of the replicas in the cluster elect a new master. This information is then relayed to the DNS servers. The clients need not be aware. For example, the clients may only need to access a URL of the form `lock.xyz.com`. Its actual mapping to an IP address needs to be time-varying. Any change is automatically taken care of by the master server in consultation with its replicas. For instance, if the master server changes, then the DNS mapping (hostname $\leftrightarrow$ IP address) simply changes. Clients simply treat `lock.xyz.com` as the URL of the master server and send their read/write lock requests to it. The DNS servers dynamically provide the IP address of the actual server that this URL is mapped to.

Replicas within a cluster can also crash. In this case, the master replaces a crashed replica with a spare machine and updates the state of the processes in the cluster accordingly.

4.3.2 Lock Files and Directories

Given that numerous locks need to be managed, there is a need to arrange them systematically. Akin to the idea in Chubby, it is a good idea to expose a file system like organization of "lock files" to clients. The state of each lock is recorded in its associated lock file. The path of a lock file is of the form `/ls/cn/srsarangi/test`. Here `ls` stands for *lock service*. The name of the cluster is `cn`. Users need to note that this mechanism should not have all the features of a regular file system. For example, it should not be allowed to move lock files across cells, lock services and even directories. That would make little sense.

The idea here is to just represent a path with '/' separated names and not implement a full-fledged file system. Each lock is a reader-writer lock (multiple readers or one writer). It is technically an "advisory" lock mainly because there are no hardware or OS mechanisms that explicitly forbid a process from acquiring the corresponding resource. Processes still should respect the locking process because the aim is to use this locking service to achieve cooperation. This system is thus not immune to faults where a

process can deviate from ideal specifications. Sometimes this mechanism is useful especially if all the clients are trusted. Note that in some cases, it should be possible to forcefully access a file for debugging and administrative purposes like a security audit.

Both files and directories can be locked. A file corresponds to the lock associated with a specific resource. Acquiring a lock on the file implies that the corresponding resource can be accessed. The lock follows standard reader-writer semantics. This means that either multiple readers can be supported at the same time or only a single writer can be supported. A reader and writer cannot concurrently access a file. For obvious reasons, a writer can always perform a read but not vice versa. Note that such locks have an inherent issue of fairness. Assume there are 10 readers and 1 writer. Who do we give priority to? If we prefer the writer, then we will unnecessarily end up delaying all 10 readers. However, if we start giving unbridled priority to readers, then new readers will keep coming in, and it is possible that the writer will starve. There is thus a need to find a fine balance and ensure that a writer can at the most wait for a certain duration beyond which it supersedes new readers in terms of priority.

Locking a directory has other implications. A directory does not correspond to a single resource nor a set of resources corresponding to its constituent files. Instead, it is just a container for many lock files, whose corresponding locks need to be separately acquired. This means that if a directory is locked, it does not mean that all the files contained within it are also locked. It simply means that the directory's contents are locked – no file can be added or removed, and no subdirectories can be created.

Lock Metadata

Every lock has some metadata associated with it, which is stored in associated lock file. This metadata comprises the name of the lock, its state (read-locked, write-locked or free), its generation number and details of lock holders. The state of the lock indicates whether it has been acquired by a reader or via a writer. Furthermore, we store a generation number, which is incremented every time the lock undergoes the following state changes: free $\rightarrow$ read-locked, free $\rightarrow$ write-locked or read-locked $\rightarrow$ write-locked. Every time a lock is acquired, the server sends the current generation number to the client. We shall discuss the part of the file that stores the details of the lock holders next.

4.3.3 Leasing

Let us now get into the internals of the protocol. Let us assume a partially synchronous systems where the clock skew is bounded. In this case, each lock grant can be associated with a lease. The client can use the lock until the end of the *lease period*. Beyond that the client needs to request for an extension, and the same may be denied. If the lease is not extended, then the lock access is deemed to have been revoked. Leasing has its advantages. All lock grants are ephemeral, and they are revoked automatically at the end of the lease period. Without any subsequent communication both the clients and server are aware of when a granted lock ceases to be valid. This is a good algorithm in theory, however, there are some challenges. Often to access the resource, messages need to be sent over the network. They can get delayed. It is possible that when the message arrives at the resource, the lease has expired.

In this case, the lease-related metadata needs to be sent along with every request to the resource. They will match the version of the lock in the metadata with the version stored on the server and check the lease expiration time. The request proceeds if the generation is the same and the lease is still valid. If there is a mismatch, then it clearly means that the request is currently invalid. It thus needs to be denied.

One criticism of a pure leasing-based approach is that even if the resource is not being used, clients still hold the lease. There is no way of taking a lease away and giving it to another client that can make more effective use of the resource. This is thus a delay-prone and inefficient solution. Moreover, metadata needs to be sent along with every request to access the underlying resource. Let us now look at a client-side caching solution that does not use leasing. Finally, we shall combine both caching and leasing.

4.3.4 Caching

All high-performance systems maintain local caches to speed up accesses. Let us design a protocol that is similar to the invalidation-based protocol in classical multiprocessor cache coherence. For example, if the resource is a file, then a file cache can be maintained. Let us not rely on leasing.

Let us thus assume that each client contains a cache that stores data files. Furthermore, the servers also maintain the list of clients that are currently caching a copy of the file. Assume a write access. In this case, there is a need to get a write-lock. The process starts with sending a request to the master server. Before the request is granted, there is a need to *invalidate*

all the locks that are currently held. This is because we do not want to support readers and writers concurrently. Hence, the master server sends invalidation messages to all the clients other than the client that initiated the request.

There is a need to acknowledge an invalidation message by sending a reply back. This is needed because the server needs to be sure that a given client has invalidated its lock on the resource. Note that this process needs to be atomic. It is not possible to stop this process in the middle and move on to serving another request for the same file.

Once, all the invalidations have been received, the master allows the client to proceed with acquiring the write-lock with an incremented generation number. If the client does not have a copy of the file, then it is first provided a copy of the file by requesting it from one of the erstwhile readers. Reads, on the other hand, are much faster. The client just needs to get a read-lock. There is no need to invalidate the locks held by the rest of the clients.

Sometimes, it is necessary to drain the cache and send the file updates to some centralize repository – the distributed file system in this case. This is mandatory if we are invalidating a write-lock because the file has been modified. The modifications need to be stored in permanent storage. In this case, it is necessary to do so prior to sending an acknowledgement to a lock invalidation request. The server thus can be sure that the modified file contents have been written to the durable storage. Subsequently, if locks are given to other clients, then they are guaranteed to get an up-to-date copy of the file from the distributed file system, which we are treating as a form of durable storage.

Comparison between Leasing and Caching

We have seen that caching and leasing have their own pros and cons. Let us thus propose a solution that uses both and tries to derive the best of both worlds. In a certain sense, both leasing and caching are opposites of each other. The former reduces communication by ensuring that both the server and the client can take mutually synchronized actions. Both of them are independently aware of the point in time till which the client can access the resource (the lease period). There is no need to send messages. The latter (caching) instead tries to make the lock-access protocol as responsive as possible. Whenever, there is a high-priority writer, all the locks can be taken away from clients.

Let us now point out the disadvantages. Leasing reduces responsiveness.

Processes may hold a lock with a long lease, yet still not make any use of the resource. If leases are prematurely terminated, then tracking or even recalling in-flight messages is tricky. On the other hand, a pure caching-based solution does not have a built-in lock ownership termination mechanism. Clients do not voluntarily return locks back to the server.

4.3.5 Design of a Hybrid Solution

Let us thus propose a hybrid solution. Consider the cache-based protocol proposed in the previous subsection. We make an important modification. Every time a process acquires a lock, it is assigned ownership for a given period (lease period). Once the lease period expires, the lock is deemed to be taken away from the process. Any client can request for a lock extension. This is treated as a separate lock access request and needs to be treated as such. A client that has an outstanding lock extension request cannot automatically presume that it is still the owner of the lock after the expiry of the lease period. It is possible that an extension may not be granted. Even if an extension is granted, it is for all practical purposes a separate lock grant.

Note that once the lease period expires, both the client and server are aware of it. The client updates its state, flushes its cache, and the server removes the client from its list of sharers. Specifically, in the lock file, the server maintains a list of sharers (clients that have a lock on the file). Along with each sharer, the lease expiration time is mentioned along with the type of the lock. Note that if read-locks have been given, then there may be multiple sharers. However, if a write-lock has been given, then there will only be a single sharer. The rest of the actions such as invalidating all the clients that have read-locks, etc., remain the same.

Let us now look at a bunch of corner cases. Leasing and caching introduce complexities when operating together.

In-flight Messages

There could be messages in flight when the lease expires or an invalidate message arrives. For example, if the file has been modified, there is a need to write it back to the storage system. A conventional cache coherence protocol in multiprocessors will wait for them to be acknowledged before initiating any subsequent action. For example, if we wish to invalidate a write-lock, then it is possible that the write has been sent to the underlying storage system, which is hosted on a remote node. Cache-based protocols

typically wait for the write to be acknowledged. In classical multiprocessor systems, this delay is not very high. However, in large distributed systems, this delay could be substantial. The message could also get lost. Hence, an acknowledgement-based system is not useful here. We should instead wait for a short duration that is normally enough for all messages to reach their destinations. Let us refer to this duration as the *timeout period*. During this period, if a response or acknowledgement arrives, then there is no problem. However, if no such message arrives, then we need to go ahead with the full knowledge that the in-flight message is either lost or delayed.

The exact mechanism is as follows. We timestamp every message with its lock generation number. Assume that a message has been sent to the underlying storage system, and in the same time frame its lease expires. This is a race condition, which needs to be handled. Note that after the lease expires, the client cannot initiate new requests. However, messages that are in flight cannot be recalled, and it is not a wise idea to wait for their acknowledgements beyond a certain timeout period. Hence, let these messages reach their destination.

Note that we increment the generation number only when a write-lock is granted and when a read-lock is granted immediately after the corresponding write-lock is released. The generation number is not incremented if there are other readers for the same file holding read-locks. This information is maintained in the lock file. For example, when the lease for a writing process expires and there is another waiting reader or writer, the master server increments its generation number, informs the rest of the replicas and the main storage server. Once this process is deemed to have been completed successfully, the client machine is informed about the new generation number. Given that these are privileged servers with high network priorities, we expect such operations to complete quickly and seamlessly most of the time. Furthermore, assume that this happens atomically with an atomic broadcast protocol. Let the underlying storage system be represented by a single storage server, which could actually be a representative of a cluster of servers. Given our protocol, it is guaranteed to always have the latest generation number because it receives generation information at the same time as the replicas in the cluster. It is not possible for a client to have a newer generation number. Now, if the storage server receives a request with a lower generation number, then this request can be happily discarded and a negative acknowledgement (nak) can be sent. The same is true for read requests as well.

Summary 4.3.1

To summarize, all invalidation requests operate on the local cache and invalidate all the associated locks. We wait for the acknowledgement of any operation initiated on the local machine. However, we do not wait for the acknowledgements or responses to in-flight messages beyond a small timeout period. This period should normally be sufficient for messages to reach their destination and update the state there. However, beyond this period, we let the storage server decide their validity based on the generation numbers.

Failure of the Master Server

In any distributed system, any server can fail including the master server. If this server fails, then we need to design a recovery mechanism. We have up till now assumed that the replicas are always maintained in sync with the master server. The master server does not act on any piece of information, unless it is replicated in the replica servers. As a result, if it fails, then another replica can seamlessly take over.

In the worst case assume that a replica takes over as the new master and because it has just joined the system, it has incomplete information about the sharers. We can always wait till all lease periods expire. After the timeout period, all the locks will be free, and the new master can start with a fresh slate.

Observation 4.3.1

Protocols that even involve $O(N)$ messages are typically not scalable. There is thus a need for either skipping nodes such as in a DHT, or by using some degree of centralization. The nodes in the centralized cluster can use an elaborate protocol among them to maintain full synchrony. The clients can then interact with the centralized interface. Here also, there is a need to trade off the pros and cons of leasing-based approaches that involve less communication and invalidation-based synchronization approaches that need more messages.

4.4 Summary and Further Reading

4.4.1 Summary

Summary 4.4.1

1. Most resources in distributed systems are protected by a *lock*, which is a generic resource that can have only one owner at any point of time. The resource can only be accessed by the owner of the lock. Every lock supports an acquire function and a release function.

2. This property of a distributed object, which allows only one process to access it at any given point of time is known as *mutual exclusion*.

3. Solving the mutual exclusion problem in a large distributed system is difficult. Processes need to coordinate among themselves and ensure that all of them agree on which process should get the lock next.

4. The simplest algorithm in this space is the Lamport's algorithm, which has three distinct phases – ① send a lock request message to the rest of the processes, ② receive replies from them and then access the resource when all the replies are received, and finally, ③ release the lock by sending a message to all the processes. This requires approximately $3N$ messages, where N is the number of processes.

5. The Ricart-Agarwala algorithm improves on this by piggybacking the reply with the lock release message. This reduces the message complexity to roughly $2N$.

6. It is possible to do even better and reduce the complexity to $O(\sqrt{N})$ in the Maekawa's algorithm by creating a custom request set – a set of processes that need to give permission to a process to acquire the lock. The key idea is that the request sets of any two processes have at least one process in common. This process ensures mutual exclusion.

7. There are a class of algorithms known as token-based algorithms where a token is passed between processes. The owner of the to-

ken can access the resource. The Suzuki-Kasami algorithm uses such a token with a built-in queue of lock-requesting processes. This queue ensures fairness and starvation freedom. Given that every process needs to make a single request for the token by sending a message to the rest of the processes, the message complexity is roughly N.

8. It is possible to improve on this significantly, by arranging all the processes as a complete binary tree (Raymond's algorithm). The token can be passed from parent to child. This results in $O(log(n))$ complexity.

9. Chubby is a sophisticated protocol for managing locks in large-scale systems. It combines caching and leasing-based strategies to ensure that it is responsive as well as scalable.

4.4.2 Further Reading

The right point to start is the paper by Raynal and Taubenfeld [Raynal and Taubenfeld, 2022] that looks at the history of mutual exclusion algorithms from 1965-2020, particularly on shared memory systems. Dijkstra's mutual exclusion algorithm [Dijkstra, 1965] was the first proposal in this space. He used atomic read/write registers. This requirement was relaxed by Lamport in 1974 [Lamport, 1974]. Subsequently, the basic algorithm was improved significantly to make it much faster. In 1987, Lamport proposed his fast algorithm [Lamport, 1987]. In the 1990s, the need arose to make mutual exclusion locks memory-aware. The classical MCS, CLH and Anderson locks were proposed that minimize remote memory accesses [Herlihy et al., 2020].

Singhal's [Singhal, 1993] taxonomy of distributed mutual exclusion algorithms is a good starting point for pure message-passing-based algorithms. For a deeper theoretical treatment, the paper by Sanders [Sanders, 1987] provides a lot of relevant information. It looks at the information structure of such algorithms. In terms of practical algorithms two simple quorum-based algorithms are presented in reference [Luk and Wong, 1997]. They are extensions of the Maekawa's algorithm. Generalizations of the classical mutual exclusion problem are proposed in [Hadzilacos, 2001] that propose group mutual exclusion. In the group mutual exclusion problem, sessions contend for a resource. All the processes in a session can execute concurrently.

Questions

Question 4.1. In the Ricart-Agarwala algorithm, what are the properties of the clock that is used? Is it a scalar clock, is it a vector clock, or is it something else? Explain your answer.

Question 4.2. Prove that the Ricart-Agarwala algorithm achieves mutual exclusion.

Chapter 5

Distributed Algorithms

> *“In a distributed system, simplicity is hard, but it is simplicity that allows complexity to emerge.”*
>
> Generated by ChatGPT

A large distributed system often has to deal with large amounts of data, typically structured as graphs. It is necessary to process such large graphs in a distributed fashion. Distributed algorithms for processing such large amounts of data often prove to be very useful primitives in large distributed systems. We have discussed specialized algorithms for mutual exclusion and data storage in earlier chapters. In this chapter, we shall focus on algorithms that are mostly designed for graph problems such as leader election, graph traversal, and graph coloring. These will be used as the basic primitives to implement complex functionalities such as finding the termination of a computation or collecting a safe checkpoint of an execution.

Recall that a distributed system can be seen as a collection of processes (nodes) with communication links between them. The links can be unidirectional or bidirectional. This allows us to model the distributed system itself as a graph G. The nodes of G can be the processes in the distributed system, and the edges of G can be the communication links of the distributed system.

We will start with the simple synchronous model of execution. We can assume that there is perfect clock synchrony across all processes and all messages reach their destinations within a bounded delay. The computation is divided into rounds that have the same structure. At the beginning of

a round, all the processes receive the messages meant for them that were sent in the previous round. In the first round, each process just reads its inputs. Then it does local computation that takes a bounded amount of time. Finally, each process sends messages to other processes with the outcome of its local computation. This sequence of events repeats itself. Any synchronous algorithm runs for a finite number of rounds. The advantage of such algorithms is that they are relatively simple to write. However, on the flip side, their practicality is a concern. Most distributed systems do not provide such strong guarantees.

Distributed systems are asynchronous. This means that there is no global clock and message delays can be very large. Many computation models also consider node and link failures. There are algorithms to convert asynchronous systems to synchronous systems as long as they provide some guarantees. They are known as *synchronizers*. However, synchronizers still rely on strong assumptions, which are not possible to guarantee in generic systems. Hence, there is a need to design algorithms for purely asynchronous systems.

We shall look at some of the most important algorithms for asynchronous systems in this chapter such as leader election, computation of the minimum spanning tree (MST), snapshot collection and termination detection. They are efficient and robust. The key advantage is that they don't rely on a global clock and bounded message delivery times.

We will finally look at the field of distributed computation models. Computation models place restrictions on the size of each message, network topology, amount of local memory and the nature of clock synchrony.

Organization of this Chapter

Figure 5.1 shows the organization of this chapter. We will start with synchronous algorithms in Section 5.1. We shall focus on three important problems: breadth-first search (BFS), maximal independent set and vertex coloring. These are generic problems that find uses in many applications. They are sadly bedeviled by limitations of the synchronous communication framework. We will study this further in Section 5.2 by looking at the constraints placed by different synchronous programming models. The time complexity (round complexity) of synchronous algorithms is a function of the constraints that are placed on the bandwidth of links, storage space on each node and the topology of the network.

Next, we shall look at asynchronous algorithms in Section 5.3. A quintessential problem is leader election where the processes elect a leader. The leader

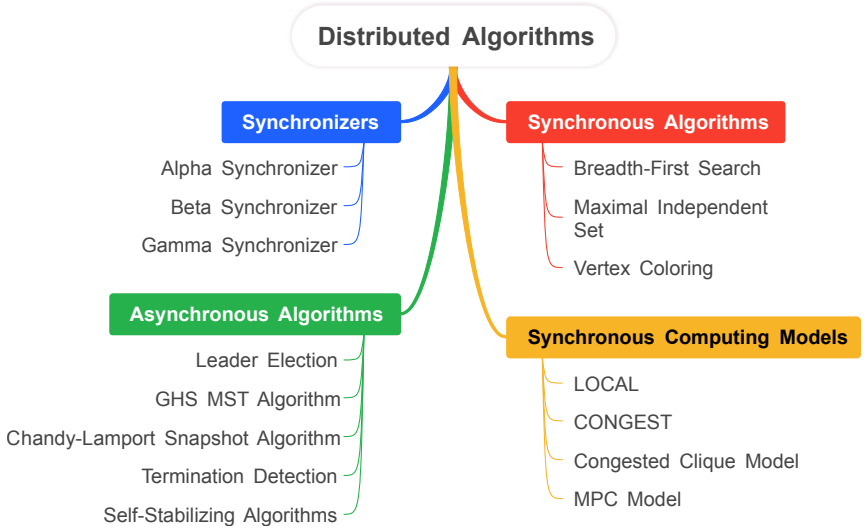


Figure 5.1: Organization of this chapter

can take centralized decisions in some cases, where it is deemed to be reliable and no point of contention will be created. We shall look at three different algorithms in this space. Next, we will discuss a reasonably complicated algorithm to find the minimum spanning tree (MST). An MST finds generic use in distributed systems. It can be used to broadcast messages to all the nodes very efficiently. Subsequently, we shall look at more sophisticated algorithms that use the basic primitives. The Chandy-Lamport algorithm is used to collect consistent distributed snapshots. These are generic algorithms that run on top of other algorithms. They provide generic services. Other examples that we shall look at include termination detection and self-stabilizing algorithms. Notably, self-stabilizing algorithms guarantee recovery from an arbitrary state.

Finally, it is possible to bridge the synchronous and asynchronous worlds. It is possible to create middleware that runs on asynchronous distributed systems to provide the illusion that the system is synchronous. This allows synchronous algorithms to run on asynchronous systems subject to some caveats and restrictions. These mechanisms are known as *synchronizers* (Section 5.4). Specifically, we shall study three different synchronizers namely the α, β and γ synchronizers.

5.1 Synchronous Algorithms

In this section, we shall study synchronous algorithms for processing graphs. Any synchronous algorithm's execution can be divided into a sequence of *rounds*. As discussed earlier, in a round, each process receives all the messages sent in the previous round, processes them and possibly sends messages to other processes. These messages are meant to be processed in the next round. There is clear agreement across the processes with regard to when a round ends and the next round begins.

In this section, we shall assume that every node in the graph is associated with a process. A graph $G = (V, E)$ has V as its vertex set and E as its edge set. This means that for all $v \in V$, there is a process in the distributed algorithm and vice versa. Now, instead of writing that processes send messages to each other, we shall write that nodes send a message to each other. This is more intuitive. Here a *node* is a vertex in the graph.

5.1.1 Breadth-First Search

The first algorithm that we shall study in the distributed setting is breadth-first traversal of graphs, popularly known as BFS (Breadth-First Search) [Cormen et al., 2009]. Here, we consider the links to be bidirectional, and the algorithm to be divided into *rounds*.

To start with, BFS requires a designated source node. The function SOURCE can be used to identify the source node. It evaluates to **true**, only when the argument is the source node. In a typical breadth-first search, we first visit all the nodes that are directly connected to the source. Then in the second round, we visit all the nodes that are directly connected to some node in the first round, so on and so forth. The procedure continues until all the nodes in the graph have been exhausted. It is easy to see that the number of rounds required to mark all nodes is at most the diameter (D) of the network [1].

More formally, in each round, a set of nodes that are *marked* enroll their unvisited neighbors and mark those nodes. During this process, such unvisited nodes are also assigned the *correct* level number that corresponds to the distance of these nodes from the source. The parent is also set during this marking step. The computation stops when there are no further nodes to mark. If the graph is connected, then we see that all nodes will eventually

[1] The "distance" between any pair of vertices is defined as the length of the shortest path between them. We can thus think of a graph's diameter as the longest "shortest path".

be marked. If the graph is disconnected, and there are nodes that are not reachable from the source, then some nodes will remain unmarked. This is because they are not reachable from the source.

Let us now discuss the algorithm line by line (refer to Algorithm 20). Initially, the source node sets its level to 0. It is its own parent. The source then sends a ⟨`mark`⟩ message to all its neighbors. The message includes the new level number, which is 1.

Subsequently, we iterate for D rounds, where D is the diameter of the graph. In the next round, the neighbors mark themselves as nodes at level 1. Then they propagate their ⟨`mark`⟩ messages with an increased level (=2) to their children, so on and so forth. Note that a node may receive more than one ⟨`mark`⟩ message. Given that we are interested in the shortest path, a node should pick the first such message that it gets. A node then sets the message source as its parent and set its level number to be one more than the level of its parent (same level also contained in the ⟨`mark`⟩ message). Given that in this graph, the weights of all the edges are 1, then level number actually corresponds to the length of the shortest path from the source vertex.

If a node has been marked, then it means that its shortest path from the source has already been computed, and it has also propagated the ⟨`mark`⟩ message to its neighbors. Its job is done. It can thus stop participating in the protocol. This process repeats until all nodes are marked with a valid level number. It is easy to see that this process naturally terminates in at most D rounds.

Figure 5.2 shows an example of the execution of the algorithm.

Analysis

Let us analyze Algorithm 20 in terms of its performance. As can be noticed, the loop in Line 8 runs for D iterations, where D is the diameter of the graph. This follows from the observation that the node farthest from the source is at a distance that is at most the diameter of the graph. Furthermore, a node receives (or sends) κ ⟨`mark`⟩ messages. κ is equal to the number of neighbors. Therefore, the time complexity of the algorithm is $O(D)$.

Since each node needs to know its neighbors, the local space complexity is $O(\text{degree})$. Each ⟨`mark`⟩ message contains the level number of the sender and the ID of the sender. So, each message has a size equal to $O(\log n)$ when n is the number nodes in the graph. Finally, note that each edge carries at most two ⟨`mark`⟩ messages.

In the simple algorithm presented, each node knows its parent via the

Algorithm 20 Synchronous BFS Algorithm

```
1: procedure BFS
2:     marked ← false, level ← 0, parent ← φ
3:     if SOURCE (u) then
4:         marked ← true, level ← 0
5:         Send ⟨mark, u, 1⟩ to all the neighbors of u
6:     end if
7:     ▷ D is the diameter of the graph
8:     for round ∈ [1 . . . D] do
9:         parallel
10:            repeat
11:                ▷ Node u receives a message from node v
12:                Receive ⟨mark, v, l_v⟩
13:                if marked = false then
14:                    parent ← v
15:                    marked ← true
16:                    level ← l_v + 1
17:                    Send ⟨mark, u, level⟩ to neighbors
18:                end if
19:            until Node u receives messages in the current round
20:        end parallel
21:    end for
22: end procedure
```

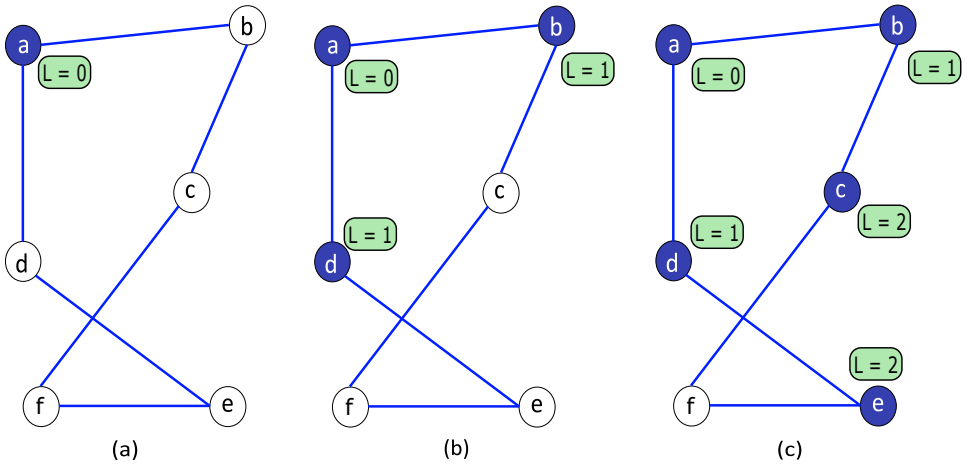


Figure 5.2: An example run of the synchronous BFS algorithm (Algorithm 20). Nodes in the dark blue color are marked (visited) and the nodes with a white background are unmarked. Node a is the source. It is marked first (Fig. (a)). Then its neighbors b and d are marked in the next round (Fig. (b)). In the next round, the other nodes, c and e are marked (Fig. (c)). L stands for the level.

assignment to the `parent` variable in Line 14.

5.1.2 Maximal Independent Set (MIS)

One of the fundamental problems on graphs is to find independent sets. In a graph $G = (V, E)$, an independent set is a set of nodes U, where $U \subseteq V$ and for each $v, w \in U$, $(v, w) \notin E$. Such an independent set U is called a *Maximal Independent Set* (MIS) if no proper superset of U is also an independent set. Figure 5.3 shows an example of a graph and two of its maximal independent sets (Figures 5.3(b)–(c)).

Note that an MIS is not a Maximum Independent Set, which is the largest MIS (in terms of the number of constituent nodes) in G. The problem of finding a maximum independent set is an NP-complete problem [Garey and Johnson, 1979], whereas a maximal independent set can be found in $O(|V| + |E|)$ sequential time using a simple greedy algorithm.

Deterministic Sequential Algorithm

Let us start with a simple algorithm that runs in a sequential setting to produce an MIS. Algorithm 21 shows a sequential algorithm that can identify

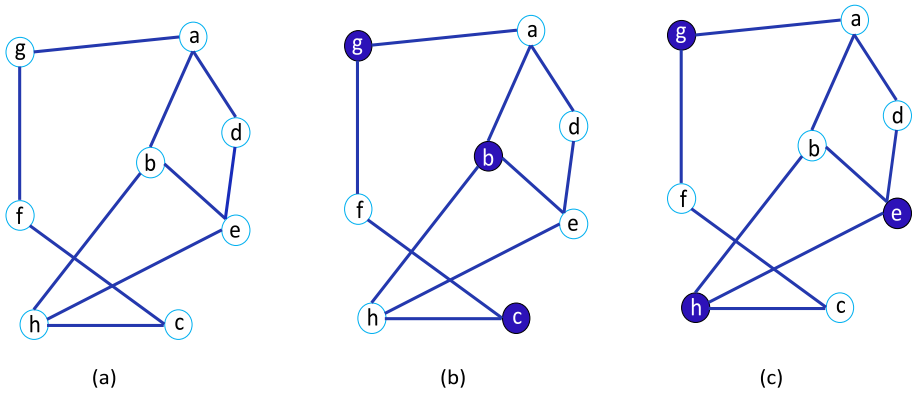


Figure 5.3: (a) The graph (b,c) Two maximal independent sets (MIS). The nodes in the MIS have a dark color.

an MIS in a graph. This greedy algorithm starts with an empty MIS and adds one vertex to the MIS under construction in each iteration of the `while` loop in Line 3. Notice that the size of the MIS depends on the choice of the vertex `v` made in Line 4.

Algorithm 21 Sequential Greedy Algorithm for Identifying an MIS

1: `U` $\leftarrow \phi$
2: `S` $\leftarrow$ `V`
3: **while** `S` $\neq \phi$ **do**
4: Pick a vertex `v` $\in$ `S` and add `v` to `U`
5: Delete `v` and all its neighbors from `S`
6: **end while**
7: **return** `U`

Some more observations concerning Algorithm 21 can be made. We can convert it to a distributed algorithm using standard techniques. The key operation is to find the node that can be added to `U` in any iteration of the algorithm.

In a distributed system, this can be simulated using messages. Note that it is, however, necessary to include the sets `U` and `S` in the messages. This is because distributed algorithms do not have access to global state. Nevertheless, the algorithm is still sequential in character in the sense that in each iteration, at most one node can be added to `U`. This algorithm is clearly slow, unwieldy and hence is not an efficient distributed algorithm.

One drawback of Algorithm 21 is the large number of iterations of the

`while` loop in Line 3. In the worst case, there can be $O(n)$ ($n = |V|$) iterations. This leads to $\Omega(n)$ rounds in the naive distributed version of the algorithm. We thus need to design faster distributed algorithms, and reduce the number of rounds needed to identify an MIS.

Fact 5.1.1

Deterministic distributed algorithms, which do not rely on any form of randomization, for obtaining an MIS have a rich history. The seminal result of Awerbuch et al. [Awerbuch et al., 1989] describes a deterministic distributed algorithm, which requires $e^{\sqrt{\log n \log\log n}}$ rounds to obtain an MIS. The current best deterministic distributed algorithm for the MIS problem requires $e^{\sqrt{\log n}}$ rounds [Panconesi and Srinivasan, 1996]. All these results rely on network decomposition that partitions the graph into a few clusters of a small diameter. Even though the overall round complexity is less than $O(n)$ (as compared to a naive distributed version of Algorithm 21), the algorithms are reasonably complicated. Hence, we seek simpler algorithms that may not necessarily be deterministic.

Observation 5.1.1

The fundamental task is to quickly identify a *large* set of vertices that can be part of an independent set. These decisions have to be made simultaneously by several nodes with little coordination. Since all nodes are alike, we view them as being *symmetric* and hence some mechanism is needed to break the symmetry among competing nodes. This problem is often termed *symmetry breaking* and is a fundamental problem in distributed computing.

Luby's Randomized Algorithm

While efficient deterministic algorithms are hard to design, randomization has proved to be quite helpful in symmetry breaking. In this section, we shall study one of the classical algorithms for obtaining an MIS (Luby's algorithm [Luby, 1986]). This algorithm was also designed independently by Alon et al. [Alon et al., 1986]. We will present the variant described by Wattenhofer [Wattenhofer, 2014].

The basic idea of the algorithm is to find a sufficiently large set of nodes that can be added to the independent set in each iteration. To do so,

the algorithm uses randomization to identify a set R that is not necessarily independent but can be trimmed to get the set S (refer to the sequential algorithm). The challenge is to ensure that R is big enough and trimming R still leaves a big enough set S. Let us explain a short version of the algorithm first. Then we will explain the detailed pseudocode.

The following set of synchronous super-rounds keeps repeating until there are active nodes. Each super-round can be seen as a collection of three rounds.

1. Each node u marks itself with probability $\frac{1}{2 \cdot \text{DEG}(u)}$ if it has neighbors. If it is a singleton node, it marks itself.

2. If the node higher-degree node is marked, then the node is added to the MIS. Ties between degrees are broken on the basis of the node id.

3. The node that joined the MIS is removed from the graph along with its neighbors.

More formally, Algorithm 22 shows the pseudocode. The four rounds as mentioned in the pseudocode constitute one super-round. Each node has a state, which is either active or inactive. Initially, we consider each node to be in the active state. Every node in the active state attempts to join the independent set. If u joins the independent set or becomes a neighbor of a node in the independent set, then, the state of u is set to inactive. In the algorithm, DEG(u) refers to the degree of the node u and NBR (u) refers to its active neighbors.

Explanation

Consider the fact that an active node and one of its active neighbors set their flag to **true** in round 1 of the same super-round. Round 2 of this super-round ensures that only one of them will get marked since the node with the lower degree gets unmarked. If the degrees are the same, then ties are resolved by using the number of the node as the tie breaker (refer to Line 16).

Rounds 3 and 4 of a super-round ensure that once a node is added to the MIS, the state of the node changes to inactive. Along with it, the states of its active neighbors are also changed to inactive.

This process continues until all the nodes are marked inactive. At this point of time, either a node is a part of the MIS, or one of its neighbors is. We thus have a maximal independent set (MIS), because no other node can be added to it.

Algorithm 22 Randomized Algorithm for Finding an MIS

repeat ▷ Round 1
  if ACTIVE (u) **then**
    flag ← **false**
    if DEG(u) = 0 **then**
      flag ← **true** ▷ flagged
    else
      flag ← **true** $\left(\text{with probability } \frac{1}{2\times \text{DEG}(u)}\right)$ ▷ flagged
    end if

    if flag **then**
      MARK (u) ▷ Assume u is in the MIS
      Send $\langle u, \text{DEG}(u)\rangle$ to NBR (u)
    end if
  end if
▷ Round 2
  if ACTIVE (u) **then**
    for $\forall$ w $\in$ NBR (u) **do**
      if ($\langle w, d_w\rangle$ received) $\wedge$ ($\langle u, \text{DEG}(u)\rangle < \langle w, \text{DEG}(w)\rangle$)) **then**
        UNMARK (u) ▷ u yields
      end if
    end for
  end if
▷ Round 3
  if MARKED (u) **then** ▷ Last round successful
    MAKEINACTIVE (u) ▷ Remove
    Send $\langle u, \text{InMIS}\rangle$ to NBR (u)
  end if
▷ Round 4
  if received $\langle w, \text{InMIS}\rangle$ **then**
    MAKEINACTIVE (u) ▷ Remove
    Send $\langle u, \text{inactive}\rangle$ to NBR (u) ▷ Ask neighbors to remove it
  end if
  ▷ Nodes adjust the list of neighbors
until $\sum_{u\in V}|\text{ACTIVE}(u)| > 0$ ▷ There are active nodes

Analysis

Let us now compute the time complexity in terms of the number of rounds. We shall start with defining some mathematical terminology that will increase the readability of our proofs. Let us use the function d to represent the function DEG in the algorithm (degree of a node). Similarly, we use the function η to represent the function NBR (set of all the neighbors). Our analysis is on the lines of that presented in reference [Wattenhofer, 2014].

Lemma 4

A node v is *marked* (is a part of the MIS) in Round 3 with a probability $p \geq \frac{1}{4d(v)}$.

Proof: Let M be the set of flagged nodes in Round 1. Let us compute the probability that v is not there in the MIS. Let $h(v)$ be the set of neighbors with a higher degree.

$$\begin{aligned}
P(v \notin MIS | v \in M) &= P(\exists w \in h(v), w \in M | v \in M) \\
&= P(\exists w \in h(v), w \in M) \quad \text{(independence)} \\
&\leq \sum_{w \in h(v)} P(w \in M) \quad \text{(union bound)} \\
&= \sum_{w \in h(v)} \frac{1}{2d(w)} \qquad (5.1) \\
&\leq \sum_{w \in h(v)} \frac{1}{2d(v)} \quad (d(w) \geq d(v)) \\
&\leq \frac{d(v)}{2d(v)} = \frac{1}{2}
\end{aligned}$$

We can thus infer the following.

$$P(v \in MIS \mid v \in M) \geq \frac{1}{2} \qquad (5.2)$$

We also know that $P(v \in M) = \frac{1}{2d(v)}$. This follows from the logic in Round 1.

Hence, $P(v \in MIS) = P(v \in MIS | v \in M) \cdot P(v \in M) \geq \frac{1}{2} \cdot \frac{1}{2d(v)}$.

This proves the statement of the theorem. $p = P(v \in MIS) \geq \frac{1}{4d(v)}$. ■

Definition 5.1.1 Good and Bad Nodes

A node v is *good* if $\sum_{w \in \eta(v)} \frac{1}{d(w)} \geq \frac{1}{3}$. Otherwise, the node is *bad*.

Lemma 5

A good node v is removed from the graph with a probability $p \geq \frac{1}{36}$.

Proof: If the node v has a neighbor with degree 2, then we know from Lemma 4 that it is a part of the MIS with probability at least $\frac{1}{8}$. This trivially proves the statement of the theorem because the moment this neighboring node is added, v will be removed. Recall that once a node is added to the MIS, all its neighboring nodes are removed.

Let us thus consider a situation where all the neighbors at least have a degree equal to 3. This means that $d(w) \geq 3$, which implies that $\frac{1}{d(w)} \leq \frac{1}{3}$, where w is a neighbor of v ($w \in \eta(v)$).

Given that v is a good node, $\sum_{w \in \eta(v)} \frac{1}{d(w)} \geq \frac{1}{3}$. This follows from the definition of a good node.

Let us prove that there is some subset of nodes $S \subseteq \eta(v)$ such that $\sum_{w \in S} \frac{1}{d(w)} \in \left[\frac{1}{3}, \frac{2}{3}\right]$ (the sum is between $\frac{1}{3}$ and $\frac{2}{3}$). We know that the sum of $\frac{1}{d(w)}$ across all the nodes in $\eta(v)$ is $\geq \frac{1}{3}$ ($\because$ it is a good node). If the sum is $\leq \frac{2}{3}$, then we are done.

Otherwise, it is more than $\frac{2}{3}$. We can start subtracting $\frac{1}{d(w)}$ from it. During this process, it is not possible that no value is reached, which is between $\frac{1}{3}$ and $\frac{2}{3}$. This is because every value of $\frac{1}{d(w)} \leq \frac{1}{3}$. We have assumed that $d(w) \geq 3$.

Hence, the following equation holds.

$$\frac{1}{3} \leq \sum_{w \in S} \frac{1}{d(w)} \leq \frac{2}{3}$$

Next, let us find the probability that a node is removed (marked as inactive). Let the set of removed nodes be R, and the set of initially marked nodes after Round 1 be the set M. The IE principle, refers to the well-known inclusion-exclusion principle in probability theory. Additionally, we will use the result ($P(u \in M) \geq P(u \in MIS)$) in the derivation shown below. It will be referred to as Result:1. This follows from the fact that if a node is marked in Round 1, then later on it can get unmarked if a node with a higher degree is marked.

Let us find the probability of removing node v from the graph in Round 4. This probability is represented using the expression $P(v \in R)$. Recall that the set $S \subseteq \eta(v)$ was chosen such that the sum of $1/d(w)$ across all its nodes is in the range $\left[\frac{1}{3}, \frac{2}{3}\right]$.

$$
\begin{aligned}
P(v \in R) &\geq P(\exists u \in S, u \in MIS) \\
&\geq \sum_{u \in S} P(u \in MIS) - \sum_{u,w \in S, u \neq w} P(u \in MIS \wedge w \in MIS) \quad \text{(IE princ.)} \\
&\geq \sum_{u \in S} P(u \in MIS) - \sum_{u \in S} \sum_{w \in S} P(u \in M) \cdot P(w \in M) \quad \text{(Result:1)} \\
&\geq \sum_{u \in S} \frac{1}{4d(u)} - \sum_{u \in S} \sum_{w \in S} \frac{1}{4d(u)d(w)} \\
&\geq \sum_{u \in S} \frac{1}{2d(u)} \left(\frac{1}{2} - \sum_{w \in S} \frac{1}{2d(w)} \right) \\
&\geq \frac{1}{6} \left(\frac{1}{2} - \frac{1}{3} \right) = \frac{1}{36}
\end{aligned}
$$

This proves the lemma. ■

Next, we need to find how many nodes are actually *good*. If quite a few nodes are good, then we can consider Lemma 5 to be a strong result. It would imply that there is a sizable reduction in the number of good nodes in every iteration. Sadly, in a star graph only the central node is good, and the rest are bad. Hence, Lemma 5 is not that powerful in practice. Let us thus define *good* and *bad* edges.

Definition 5.1.2 Good and Bad Edges

- An edge is *bad* if both of its endpoints are bad.
- An edge is *good* if at least one of its endpoints is good.

Lemma 6

In any iteration, at least half the edges are good.

Proof: Let us create a directed graph out of the original undirected graph. Consider an edge (u, v). Assume without loss of generality that the degree of node v is greater than the degree of node u. We orient the edge from u to v.

Consider a bad node v. Let the number of edges with v as the source be q_{out}. Let the number of edges with v as the destination be q_{in}. We wish to find a relationship between q_{out} and q_{in}.

Assume $q_{out} \leq 2q_{in}$. Let set L be the set of nodes in $\eta(v)$ that have a lower degree than v. This means that there is an edge from each node in L to v. We thus have $\mid L \mid = q_{in}$.

$$\begin{aligned} \mid L \mid &= q_{in} \\ &= \frac{3 \cdot q_{in}}{3} \\ &= \frac{2 \cdot q_{in} + q_{in}}{3} \\ &\geq \frac{q_{out} + q_{in}}{3} \\ &= \frac{d(v)}{3} \end{aligned} \tag{5.3}$$

We have the following relationship.

$$\begin{aligned} \sum_{w \in \eta(v)} \frac{1}{d(w)} &\geq \sum_{w \in L} \frac{1}{d(w)} \geq \sum_{w \in L} \frac{1}{d(v)} \\ &= |L| \cdot \frac{1}{d(v)} \\ &\geq \frac{d(v)}{3} \cdot \frac{1}{d(v)} \\ &= \frac{1}{3} \\ \Rightarrow \sum_{w \in \eta(v)} \frac{1}{d(w)} &\geq \frac{1}{3} \end{aligned} \tag{5.4}$$

This means that node v is good (by definition). However, we have assumed it is bad. Thus, there is a contradiction. Hence, we have the following relationship: $q_{out} > 2q_{in}$. Alternatively, $q_{in} < \frac{q_{out}}{2}$.

Let us consider the set $\mathcal{B}$ of all the bad nodes. Let us define a few additional terms. Let $q_{in}(w)$ be the number of edges that enter node w. We define $q_{out}(w)$ similarly.

$$
\begin{aligned}
Q_{in} &= \sum_{w \in \mathcal{B}} q_{in}(w) \\
Q_{out} &= \sum_{w \in \mathcal{B}} q_{out}(w)
\end{aligned}
\tag{5.5}
$$

Given that for every bad node, $q_{in} < q_{out}/2$, we have $Q_{in} < Q_{out}/2$.

Next, let us derive a few relationships. We need to bear in mind that Q_{out} can never exceed E because every edge is counted at most once while computing Q_{out}. Similarly, the number of bad edges is limited to Q_{in}. Q_{in} cannot be exceeded because the destination of every bad edge has to be a bad node.

$$
\begin{aligned}
& Q_{out} \leq E \\
\Rightarrow & \frac{Q_{out}}{2} \leq \frac{E}{2} \\
\Rightarrow & Q_{in} < \frac{E}{2} \\
\Rightarrow & \mid \text{bad-edges} \mid < \frac{E}{2} \\
\Rightarrow & \mid \text{good-edges} \mid \geq \frac{E}{2}
\end{aligned}
\tag{5.6}
$$

This proves the lemma. At least half the edges have been proven to be good. ■

Theorem 5.1.1

The average number of rounds required by the randomized version of Luby's algorithm is $O(\log n)$.

Proof: Based on the two previous lemmas, we know that at least half the edges (which are good) will be removed with a probability that is at least $\frac{1}{36}$ in a given iteration. If there are m edges, then the expected number of edges that are removed is at least $\frac{1}{2} \cdot \frac{m}{36}$, which is $\frac{m}{72}$. So, in $O(\log m) = O(\log n)$ rounds, on an average, all edges are removed. Once all the edges are removed, the algorithm terminates. ■

5.1.3 Vertex Coloring

The problem of vertex coloring is to assign colors (labels) to vertices such that no two neighbors receive the same color. Such a coloring is called a

proper coloring. The minimum number of colors needed to achieve such a coloring is the *chromatic number* of the graph G (denoted by $\chi(G)$). It is well-known that finding $\chi(G)$, or approximating $\chi(G)$ up to $O(n^{\epsilon})$ is also NP-hard. This means that finding an algorithm that requires $O(n^{\epsilon})\chi(G)$ colors is also NP-hard.

On the other hand, there exists a very simple sequential algorithm to obtain a proper coloring using $\Delta + 1$ colors where Δ is the degree of the graph. A simple greedy algorithm can achieve a coloring with $\Delta + 1$ colors. We number the vertices and visit them in order. For each vertex, we find a color that is not the same as one of its neighbor's. Given that we can use $\Delta + 1$ colors, we will always find such a color.

One of the earliest randomized algorithms for obtaining a $(1 + \epsilon)\Delta$ coloring was inspired by Luby's MIS algorithm. In this case, we can think of ϵ as a small positive constant such that $(1 + \epsilon)\Delta > \Delta$. Assume that nodes know the value of Δ. Each node can thus start with a palette of $(1 + \epsilon)\Delta$ colors.

Algorithm 23 describes the pseudocode nodes use to obtain a coloring. We shall prove that the number of rounds is $O(log(n))$ with high probability, where n is the number of nodes. This means that the probability of a node not getting colored after $O(log(n))$ rounds is a $O(n^{-c})$ $(c > 0)$.

Let us quickly describe the algorithm. Each node that is yet to be colored picks an available color from its palette independently and uniformly at random. If the choice of a node `u` conflicts with the choice of its neighbor `v`, then both `u` and `v` relinquish their choice and stay uncolored. This is indicated by the variable `yield`. If the choice of a node is not in conflict with the choice of any of its neighbors, then it gets colored with the color of its choice. Note that this becomes the final color. Subsequently, the node informs its uncolored neighbors. These neighbors then delete that color from their palettes. This process repeats until all the nodes obtain a color.

Correctness

It is clear that two neighboring nodes cannot choose the same color. If a node finally chooses a color, then it informs all its neighbors. All the uncolored nodes remove the chosen color from their palettes. This ensures that in subsequent rounds, this color cannot be chosen. This is because in Line 3 for tentatively assigned colors, nodes inform their neighbors. If a node finds that the color it has tentatively chosen has also been chosen by one of its neighbors, then it decides to *yield*. This means that it lets go of its choice. This clearly means that a color is chosen by node `u` only if all its

Algorithm 23 Randomized distributed algorithm for obtaining a coloring

```
1:  repeat
                                                  ▷ Round 1: Select a Color
2:      if STATE (u) = uncolored then
3:          CHOICE (u) ← RANDOM_COLOR (PALETTE (u))
4:          Send ⟨u, CHOICE(u)⟩ to all v ∈ NBR (u)
5:      end if
                                                  ▷ Round 2: Detect Conflicts
6:      if STATE (u) = uncolored then
7:          for w ∈ NBR (u) do
8:              if received ⟨w, CHOICE(w)⟩ ∧ CHOICE (u) = CHOICE (w) then
9:                  YIELD (u) ← true
10:             end if
11:         end for
12:     end if
                                  ▷ Round 3: Fix a Color and inform neighbors
13:     if YIELD (u) ≠ true then                          ▷ Last round successful
14:         COLOR (u) ← CHOICE (u)
15:         STATE (u) ← colored
16:         Send ⟨u, CHOICE(u)⟩ to all w ∈ NBR (u)
17:     end if
                                  ▷ Round 4: Clean up for the next iteration
18:     if (STATE (u) = uncolored) ∧ received ⟨w, c_w⟩ then
19:         Remove c_w from PALETTE (u)
20:     end if
21: until ∃w, STATE (w) = uncolored
```

neighbors have either tentatively chosen or have been colored with different colors.

The key difficulty is in proving that the time complexity in terms of the number of rounds is $O(log(n))$ with high probability, where n is the number of nodes in the graph.

Theorem 5.1.2

The time complexity of the randomized algorithm for finding a coloring is $O(log(n))$ with high probability.

Proof:

Consider any given round. Assume s neighbors of node v are uncolored. Further, assume that the palette of v comprises N colors. Let us find the probability of v choosing a color that is not the same as that chosen by one of its s uncolored neighbors. We wish to find the minimum probability. Hence, we can consider the worst case, which is that all the uncolored neighbors use the same palette. If they can make other choices, then the success probability of v increases, which we do not want because we want to compute the lower bound of the probability.

The probability that none of the s neighbors make a choice that is the same as v is $(1 - 1/N)^s$. This is v's probability of success $P(v)$. Clearly, if $s = 0$, it is 1. Let the initial size of the palette be $(1 + \epsilon)\Delta$, where ϵ is a small positive constant. Then we have the following relationship:

$$\begin{aligned} N &= (1+\epsilon)\Delta - (deg(v) - s) \\ &\geq (1+\epsilon)\Delta - \Delta + s \quad (deg(v) \leq \Delta) \\ &\geq \epsilon\Delta + s \\ &\geq (1+\epsilon)s \quad (\Delta \geq s) \end{aligned} \tag{5.7}$$

We can now simplify $P(v)$.

$$\begin{aligned} P(v) &= \left(1 - \frac{1}{N}\right)^s \\ &\geq \left(1 - \frac{1}{(1+\epsilon)s}\right)^s \\ &\geq \left[\left(1 - \frac{1}{(1+\epsilon)s}\right)^{(1+\epsilon)s}\right]^{\frac{1}{1+\epsilon}} \end{aligned} \tag{5.8}$$

Let us ignore the case where $s = 0$; the node is trivially colorable. Given that $(1 + \epsilon)\Delta > \Delta$, we can always find a color that is free. Hence, let us consider the non-trivial case, which is when $s > 0$.

Let us set $\lambda = (1+\epsilon)s$. Consider the behavior of the expression $(1-1/\lambda)^{\lambda}$, when $\lambda > 1$. This is an increasing function. The limiting value is $1/e$. The lowest value is at $s = 1$. Hence, for cases where $s > 0$, we have the following relationship.

$$P(v) \geq \underbrace{\left(1 - \frac{1}{(1+\epsilon)}\right)}_{\kappa} \tag{5.9}$$

This is a constant ($= \kappa$). This means that in each iteration of the repeat-until loop, a node has at least a constant probability of getting *colored*.

Let us compute the probability of a node remaining *uncolored* after $\alpha \cdot log(n)$ iterations. It is $(1 - \kappa)^{\alpha \cdot log(n)}$. Let us simplify this expression.

$$\begin{aligned}
(1-\kappa)^{\alpha log(n)} &= e^{ln(1-\kappa)\cdot \alpha log(n)} \\
&= e^{\beta \cdot log(n)} \qquad (\beta = ln(1-\kappa)\alpha) \\
&= \left(e^{log(n)}\right)^{\beta} \\
&= n^{\beta}
\end{aligned} \tag{5.10}$$

We know that $\beta < 0$ because $1 - \kappa < 1$ and consequently $ln(1 - \kappa) < 0$. This means that the probability that a node is uncolored after n iterations can be made to reduce polynomially with exponent β (< 0). For example, we can make it $1/n^2$, $1/n^3$, so on and so forth. Whenever the probability of failure is a polynomial function, we can write that we guarantee success with *high probability*.

Hence, in this case we can say that the time complexity in terms of the number of rounds is $O(log(n))$ with high probability. ■

5.2 Synchronous Computation Models

The field of distributed computing has over the years seen multiple models of computing. They have addressed various theoretical and practical concerns. Popular among them are the LOCAL, CONGEST, CONGESTED_CLIQUE and MPC models.

The LOCAL model is a simple synchronous model, where in each round, each machine exchanges messages of arbitrary size with its neighbors. This

model allowed early algorithm designers to study the round complexity of a distributed computation even though the assumption of arbitrary-sized message sizes was impractical. The CONGEST model is akin to the LOCAL model except for placing a bound of $O(\log n)$ bits on the message size, where n is the number of nodes in the network. Therefore, this model is closer to a practical model as it limits the communication across each link. The CONGESTED_CLIQUE model is useful for studying the interplay between communication and round complexity. In this model, akin to the CONGEST model, the bandwidth on each communication link is limited to $O(\log n)$. However, the communication network allows all-to-all communication. Finally, the MPC model takes the amount of available memory on a machine into account.

The area of synchronous algorithms is an active and popular area of research. To formally design and analyze such algorithms, the computation model needs to be clearly specified. In the synchronous algorithms that we studied in this chapter, we made use of a very simple model where in each round, each node can perform local computation and send/receive messages. We did not dwell significantly on the size of the messages that nodes can send/receive in a round, the space available on a machine, and so on.

There exist several models for designing distributed algorithms. In this section, we shall review some popular models.

5.2.1 LOCAL Model

This model was first introduced by Linial [Linial, 1992]. In each round, a node can send and receive messages of arbitrary size with each of its neighbors. There is no limit on the message size. The intention of algorithm design in the LOCAL model is to to restrict communication to immediate neighbors. Nodes do not have global knowledge of the network. The cost of local computation is ignored, and it is assumed that nodes know their own unique identifier as well as the identifiers of their neighbors. Nothing else needs to be known. The name "LOCAL" arises from the fact that the communication is local and is done only with immediate neighbors.

A naive algorithm to solve *any* problem in this model is to gather the entire graph at a single node, say the node with number 1. This can be easily done because there is no limit on the message size. If the entire graph's information is available at a single node, then that node can perform the required local computation and send the answers to the rest of the nodes. Since each of these two steps can be accomplished in a number of rounds equal to the diameter $diam(G)$ of the graph, the entire algorithm runs in $O(diam(G))$

rounds. Algorithms that improve on this trivial solution are thus of interest. For many problems of interest such as approximate dominating sets, vertex coloring and edge coloring, there are efficient polylogarithmic-round distributed algorithms in the LOCAL model. On the other hand, there are problems such as the minimum spanning tree for which $\Omega(diam(G))$ is a lower bound on the number of rounds needed.

One of the design mechanisms for efficient graph algorithms in this model is to use the initial rounds of the algorithm to decompose the graph into multiple components with a bound on the diameter of the components. One can then appeal to the naive idea of gathering the graph at a single node to solve the original problem within each component. This technique has been used by many references [Barenboim et al., 2016,Kothapalli and Pemmaraju, 2011, Kothapalli et al., 2006].

5.2.2 CONGEST Model

The CONGEST model is another popular model for designing distributed algorithms (refer to Peleg [Peleg, 2000]). It shares several commonalities with the LOCAL model such as the graph resembling the distributed system being fault-free and synchronous. A node can only communicate with its immediate neighbors. However, in this model, there is an upper limit on the message size. The message size is usually limited to $O(\log n)$ bits where n is the number of nodes. This size is enough to represent the number of a node.

We can thus view the CONGEST model as a restricted form of the LOCAL model. It has an additional bandwidth restriction, which has some immediate consequences. It is not possible to gather the entire graph at a single node in $O(diam(G))$ rounds. Therefore, the naive approach of gathering the graph and then solving any problem locally does not work in the CONGEST model. In fact, it is known that there are graph problems that do not have any $O(n)$-round algorithm in the CONGEST model [Bacrach et al., 2019].

Nevertheless, there are distributed algorithms for several graph problems that work in this model efficiently. The MIS algorithm of Luby from Section 5.1.2 is a good example. In that algorithm, each node has to send/receive at most $O(\log n)$ bits of information per round.

The LOCAL and the CONGEST models do not allow communication between arbitrary pairs of nodes. This is an unnatural restriction as of today. Nodes in most systems are aware of the entire network. In fact, it is quite easy to make them aware of the network and also keep this information

updated. Hence, as of today, all-to-all communication models are more relevant from a practical standpoint. Let us explore two models in this space.

5.2.3 The Congested Clique Model

The Congested Clique model is a simple extension of the CONGEST model. In this case, the graph is a clique. Therefore, it does not have any locality properties. Given that $O(log(n))$ bits can be exchanged between any pair of nodes in a round, the total number of bits a node can receive is limited to $O(nlog(n))$. However, here the assumption is that the messages arrive via different links. What if the communication pattern is not so uniform?

In this context, let us consider the *routing problem*. Each node has n messages that need to be sent; each message is $O(log(n))$ bits long. The messages need not be destined to unique destinations. Their destinations could be very non-uniformly distributed. There is also a constraint on the number of messages that a node can receive. It is equal to n. The aim here is to minimize the number of rounds.

Clearly, if the destinations are more or less uniformly distributed, the solution is trivial. Given that the network is a clique, the source node can simply send each message directly. In a constant number of rounds, the operations will finish. However, the problem arises when multiple messages are destined for the same node and the distribution is very skewed. The bandwidth limitation comes into force and it becomes necessary for some messages to get deferred, get delayed and possibly traverse multiple hops. However, the seminal result by Lenzen [Lenzen, 2013] proves that the routing problem can still be solved in $O(1)$ rounds. The implications of this result are profound.

It basically means that regardless of the choice of destinations, the sending throughput and receiving throughput per node is $O(nlog(n))$ because in a constant number of rounds, we can route n messages per node with size $O(log(n))$. This result holds for both sending and receiving messages. The distribution of destinations does not seem to have any effect in the asymptotic time. This leads to very fast algorithms that are either take $O(1)$ rounds or $O(loglog(n))$ rounds.

Fact 5.2.1

In the Congested Clique model, the problem of obtaining an MST of a graph takes $O(1)$ rounds [Jurdziński and Nowicki, 2018].

5.2.4 The Massively Parallel Computing (MPC) Model

Recall that the Congested Clique model was more concerned with the fact that the network is a clique and there is a logarithmic bandwidth limitation (per round) between any pair of nodes. The model assumed that there are no limits on the amount of space a machine has. This aspect is important in the context of practical settings especially cloud computing. Today's *big data* workloads use very large inputs that do not fit within a single machine. There is thus a need to distribute the input across multiple machines. The same is the case for ephemeral data and in many cases the output as well.

Addressing the above limitation, a model that closely resembles the cloud computing model is the Massively Parallel Computing (MPC) model. The MPC model has the following characteristics. The MPC model envisages a system with a set of nodes, each having at most W words of memory.

In a modern network, we can assume all-to-all communication in an abstract sense. This means that any node can send a message to any other node. The exact low-level details of the communication are irrelevant. The more important point to note is that there is a memory space limitation per node, which is relevant in today's era of big data. The model assumes synchronous rounds. Assume that each node can store at most W words. We can also assume that $O(W)$ bits can be sent or received by each node per round. In this case as well, our goal is to minimize the number of rounds. This measure ignores the time taken by local computation. We justify this by the widespread belief that communication costs usually dominate the overall cost in big data computations (see Reed and Dongarra [Reed and Dongarra, 2015]).

Assume an input of size N and R machines. The total storage space is $R \times W$ ($\in \Omega(N)$). Typically, both are functions of N and are not constants. A key characteristic of the MPC model is that both the memory bound per machine, W, and the number of machines, R, are typically sublinear in terms of the input size N. They are typically bounded by a function whose complexity is $O(N^{\delta})$ $(0 < \delta < 1)$. If this was not the case, then the entire input will fit in one node and a local algorithm can be used. Setting $\delta < 1$ ensures that the algorithm remains truly distributed in character. Moreover, for ease of analysis, we set W to be in $\tilde{O}(N^{\delta})$ and $R \in O(N^{1-\delta})$, where $\tilde{O}(N^{\delta})$ is the class of functions that are in $O(N^{\delta} log^{O(1)}(N))$. Logarithmic factors arise naturally due to the nature of distributed computation. Given that they grow very slowly, they are often ignored.

Unlike other distributed computing models such as the Congested Clique model, the number of machines needed in the MPC model ($R = O(N^{1-\delta})$)

grows sub-linearly as the input size grows. Such sublinear growth makes the model more realistic than other distributed computing models. The dependence on the number of machines also allows the model to be elastic. It automatically captures the *elasticity* of cloud computing that enables it to adjust resources on demand. We need to note that unlike models such as BSP [Valiant, 1990] and the k-machine model [Klauck et al., 2015a], keeping the number of machines dependent on the size of the input may be perceived by some as a weakness of the model. However, this is practical. We often provision more resources when the problem size is large.

5.3 Asynchronous Algorithms

In this section, we shall study asynchronous algorithms that do not rely on a single global clock. This is a robust and realistic model because it is not possible to provide synchronous execution guarantees all the time. We shall start this section with introducing three leader election algorithms. Then, we shall move to computing a distributed spanning tree, collecting a consistent snapshot and detecting the termination of computation. Finally, we will introduce the interesting field of self-stabilization algorithms.

5.3.1 Leader Election

In distributed algorithms, there is often a need to elect a leader – some kind of coordinating process. Skeptics may quickly argue that electing a leader is against the philosophy of distributed systems. After all, we want all the processes to be symmetric, do the same things and have the same responsibilities. Furthermore, if the leader goes down because of a fault, we will be left leaderless, and it may be very hard to find if the leader is just slow or is genuinely out of service (similar to the classic FLP result, see Section 6.1). This is undoubtedly a valid criticism. However, there are algorithms and that too very popular ones that rely on the existence of leaders. They are fully aware that leaders may suffer from faults and also go out of service during the execution of the protocol. At that point of time a new leader may be elected, but it is very much possible that the earlier leader suddenly wakes up and for a short period there are two leaders in the system. Keeping these disclaimers in mind, having a leader is often a good option, and it simplifies the protocol significantly.

Chang-Roberts Algorithm

Algorithm 24 Chang-Roberts Algorithm

```
1:  procedure runInitiator
2:      if state = lost then
3:          return                          ▷ Let someone else become the leader
4:      end if
5:      state ← find
6:      send ⟨u⟩ to next(P_u)                          ▷ Send u along the ring
7:      while state ≠ leader do
8:          receive(P_v)
9:          if u = v then
10:             state ← leader
11:         else if v < u then
12:             if state = find then
13:                 state ← lost
14:             end if
15:             send v to next(P_u)
16:         end if
17:     end while
18: end procedure

19: procedure runOtherProcesses
20:     while true do
21:         receive request ⟨r⟩ from P_v
22:         send ⟨r⟩ to next(P_u)
23:         if state = asleep then
24:             state ← lost                ▷ Lose interest in becoming a leader
25:         end if
26:     end while
27: end procedure
```

Let us describe the simplest algorithm in this space, the Chang-Roberts algorithm. The processes are arranged in a ring (like a DHT); however, unlike a DHT, messages can only be sent to immediate neighbors. All the processes are numbered from 1 to n. Note that they may not be contiguously numbered along the ring. Moreover, several processes may simultaneously initiate the process of leader election. Out of them, let the convention be that the process with the *smallest id* that is interested to be a leader shall

win the election. This is an arbitrary choice, however, it makes our life simple. Algorithm 24 describes the protocol from the point of view of a given process that initiates leader election for itself. Let us refer to this process as $\texttt{P}_\texttt{u}$.

For the rest of this section, we shall assume that the current process that initiates the protocol or executes a method is referred to as process $\texttt{P}_\texttt{u}$, whereas the *other* process that is sending the message is referred to as $\texttt{P}_\texttt{v}$. This is the standard convention that we shall use.

Now, consider the method runInitiator. If the state is lost, it basically means that most likely some other leader has been elected. Given that an elected leader cannot be displaced and there cannot be two leaders at a time, the initiator has no choice but to stop the process of leader election. In this case, it knows that another leader is definitely either elected or going to be elected in some time. The system will not remain leaderless.

If this is not the case, we set the state to find. This means that we are in the process of finding the leader. The initiator starts the algorithm be sending its ID $\langle \texttt{u} \rangle$ to its neighbor, $\texttt{next}(\texttt{P}_\texttt{u})$.

As long as the state is not set to leader, the algorithm continues. A process continuously listens for messages. Note that in the ring topology a process can only receive a message from its immediate neighbor and there is only one such neighbor. Assume it receives the message $\langle \texttt{v} \rangle$. There are several cases here.

If $\texttt{u} = \texttt{v}$, then it means that the message initiated by process $\texttt{P}_\texttt{u}$ went through the entire ring and then was received back. This means that no process on the ring absorbed the message. We can thus assume that the message has the highest priority and there is no other claimant that has a higher priority. In this case, the state can be set to leader.

Next, assume that $\texttt{v} < \texttt{u}$. Let us then analyze the state. If the state is equal to find, then it means that the process of searching for a leader has not terminated. Given that we want to elect that process as the leader, which has the smallest id, we can conclude that $\texttt{P}_\texttt{u}$ has lost the leader election when it is found that the request from $\texttt{P}_\texttt{v}$ has a higher priority. This is indeed the case when $\texttt{v} < \texttt{u}$. We can set the state to lost. Henceforth, if any message is received from the neighboring node, it needs to be dutifully forwarded along the ring (Line 15).

Note that if none of these conditions hold, which basically means that $\texttt{v} > \texttt{u}$, we can absorb its message and remove it from the ring. Given that we know that there are other high priority requests in circulation, there is no need to forward such requests. Either the request from $\texttt{P}_\texttt{u}$ is alive because $\texttt{P}_\texttt{u}$ has not received any message with a higher priority (lower id) or the state

has been set to lost. In the latter case when the state has been set to lost, it means that there is a request with higher priority than the request from P_u. In either case, it means that there is a request in the ring whose priority is more than the request from P_v and thus the request from P_v needs to be removed from the ring.

Let us now consider what other processes do, which are not initiators, i.e., claimants to become leaders (see the method runOtherProcesses). This method is very straightforward. Assume a request $\langle r \rangle$ is received from P_v. We just forward it to $next(P_u)$.

If the state of the node is asleep, we set the state to lost. This is because the process has already forwarded a message. If later on it is interested in becoming a leader, then there will be a problem particularly if its request has a higher priority than the priority of the requests that it forwarded in the past. Hence, the best strategy is to keep it out of the leader election race.

Proof

Assume that to the contrary, two processes have been elected as leaders. Let them be processes P_u and P_v, and let $u < v$. This means that process P_u has a higher priority.

Consider the case when P_v's request arrived at P_u (Line 11). At that point of time, if P_u was still asleep, then its state would be set to lost (as per Line 24). This means that in the future, it will not show any interest in becoming a leader. This is because it has been superseded by a request that was issued before (either P_v's request or something earlier than that). Given that it will not subsequently initiate any leader election (see Line 3), it cannot be elected as a leader. Hence, it is not possible that P_u and P_v are simultaneously elected as leaders.

Now, consider the other case when P_u's request for becoming a leader has been sent down the ring. In this case, P_v's request will never be forwarded. It is a lower priority request and the request will basically be dropped. As a result, P_v's request will not propagate down the ring, and it can never be elected as a leader.

Hence, we have proven that in both cases the leader election algorithm will elect only one leader. Note that there is no starvation because the request of the highest priority process (lowest id) is bound to succeed. No other process will absorb it.

Message Complexity

To compute the worst case message complexity, we need to find a case where the maximum number of requests are sent. We want each request to travel

as far (as many hops) as possible. Consider an arrangement of processes in chronological order. This means that process $\texttt{P}_\texttt{u}$ sends a request to process $\texttt{P}_{\texttt{u+1}}$ and then process $\texttt{P}_{\texttt{u+1}}$ sends it to process $\texttt{P}_{\texttt{u+2}}$, so on and so forth. Finally, process $\texttt{P}_\texttt{n}$ sends its request to process $\texttt{P}_1$.

Let us assume that all the processes are interested in becoming a leader, and they operate in lockstep. This is a synchronous computing model, which is fine. All synchronous settings are also asynchronous settings, not the other way around.

Say at $t = 1$, all the processes have their leader election requests ready. They send it down the ring to their neighbors. At $t = 2$, assume that all the requests have reached the neighbors. Other than the request from node n, which has reached process 1 and will get absorbed (not transmitted any further), the rest of the requests are not absorbed. This is because they are at processes that have sent requests that have a lower priority. Hence, the processes are bound to forward them to their neighbors. Ultimately, all the requests will reach node 1 at which they will get absorbed.

We can easily count the number of hops that all the requests traverse. The request from process n traverses just one hop $n \rightarrow 1$. The request from process $\texttt{P}_\texttt{i}$ traverses $n - \texttt{i} + 1$ hops. The total number of hops traversed or the number of messages sent is as follows:

$$M_{tot} = \sum_{i=1}^{n}(n - i + 1) = \frac{n(n+1)}{2} \tag{5.11}$$

We thus need to send $O(n^2)$ messages to elect a leader. For any process, the maximum number of messages sent is n (process 1 in this case). An overhead of $O(n^2)$ is not acceptable when we have many processes. Let us thus design a more efficient solution.

Algorithm with an O(n log(n)) Message Complexity

Consider Algorithm 25 (reference: [Singhal and Shivaratri, 1994]). It makes similar assumptions as the Chang Roberts algorithm. The processes are organized as a ring and multiple leader election requests can be made simultaneously. The request with the lowest process ID wins. However, in this case, all the requests need not traverse the full ring. We run many mini-algorithms – local leader election algorithms in arcs of increasing sizes on the ring. This ensures that unless a process has a chance of winning the leader election, its request does not go very far.

Let us start with the INITIALIZE method. In this case, two ⟨`probe`⟩ messages are sent to the left and right neighbors. A ⟨`probe`⟩ message contains the `id` of the current process and two more parameters `k` (log of the search distance) and `d` (distance), which are set to 0 and 1, respectively.

The algorithm operates in a series of phases. `k` is set to 0 for the first phase, 1 for the second phase, so on and so forth. We assume that the leader election request (⟨`probe`⟩ message) goes 2^{k} hops in both the directions (left and right) in a given phase. If it is not *absorbed*, the algorithm proceeds to the next phase – `k` gets incremented by 1. `d` maintains a count of how far the request has been sent in a given direction (left or right). It will never be allowed to exceed 2^{k}.

The *field of view* of a process is limited to 2^{k} hops to its left and right for a given value of `k`. A phase succeeds if a process has the highest priority in its field of view. This further means that the leader initiation request (`probe` request) was not *absorbed* by any process in the field of view. We can thus think of the initiating process as the leader of all the processes in its field of view. Given that the field of view increases by a factor of 2 in every phase, ultimately one arm of it shall encompass the entire ring. This is when the initiating process will receive its own ⟨`probe`⟩ message assuming it has the highest priority. It will thus be elected as the leader. Let us now do a deep dive into the pseudocode shown in Algorithm 25 before we mathematically prove that this algorithm is indeed more efficient.

The node that wishes to become a leader invokes the function INITIALIZE. It finds its current ID (`u`) and sends a ⟨`probe`⟩ message to its left and right neighbors. Initially, its field of view is equal to 1.

Consider the RECEIVEPROBE method defined in Line 5. A process calls this method when it receives a leader election request from its left or right neighbor. Assume that it gets the request from process `v`. Consider the simplest case first when the process gets back its own leader election message (`u` = `v`, Line 8). In this case, we can just declare the current process to be the leader and terminate the algorithm.

Next, consider the case where `v` < `u`. In this case, the process making the request (`v`) has a higher priority as compared to the current process. Hence, there is a need to propagate its request. Of course, if a leader has already been elected, then the request from `v` can be absorbed. We are not showing that case in the interest of readability.

Assuming that no leader has been elected, if $\texttt{d} < 2^{\texttt{k}}$, then there is a need to propagate the request further. If it came from the left neighbor, it needs to be sent to the right neighbor, and vice versa. Every time the message is forwarded, `d` is incremented. It is like a running count.

Algorithm 25 An optimized leader election algorithm

```
1: procedure INITIALIZE
2:     u ← GETID()
3:     send ⟨probe, u, 0, 1⟩ to the left and right neighbors
4: end procedure

5: procedure RECEIVEPROBE(⟨probe, v, k, d⟩ from left(right))
6:     u ← GETID()
7:     ▷ The current process is the leader if u = v (sender of the message)
8:     if u = v then                                  ▷ own message comes back
9:         leader ← v
10:        terminate()
11:    end if
12:    if v < u and d < 2^k then
13:        send ⟨probe, v, k, d + 1⟩ to right (left)          ▷ keep propagating
14:    else if v < u and d = 2^k then
15:        send ⟨reply, v, k⟩ to left (right)                 ▷ reply back
16:    end if
       ▷ Absorb the request if none of the aforementioned conditions hold
17: end procedure

18: procedure RECEIVEREPLY(⟨reply, v, k⟩ from left(right))
19:    if v ≠ GETID() then
20:        send ⟨reply, v, k⟩ to right(left)
21:    else if received ⟨reply, v, k⟩ from left and right then
22:        send ⟨probe, v, k + 1, 1⟩ to left and right
23:    end if
24: end procedure
```

Once $\texttt{d} = 2^{\texttt{k}}$ (Line 14), it means that the end of the current phase has been reached. There is a need to send a ⟨`reply`⟩ message in the reverse direction (towards the initiating process). If $\texttt{u} \neq \texttt{v}$, there is nothing much to do. The processes simply need to forward the reply message. However, if $\texttt{u} = \texttt{v}$, then there is some additional work to do because the reply has been received at the initiating process. Note that two ⟨`probe`⟩ messages were sent in the left and right directions, respectively. We need to wait till replies arrive from both the sides. This would indicate that no leader election request has been sent by a higher priority process (lower id). Once both the replies are received, the initiating process becomes the leader in its field of view. It can now double the size of its field of view by incrementing `k`. It needs to then send a pair of ⟨`probe`⟩ messages to its left and right neighbors (shown in Line 22).

Ultimately, one process will receive its ⟨`probe`⟩ message back. It is the leader.

Proof

Consider the highest-priority process (lowest id)in the system. Its messages will never get absorbed and its field of view will keep expanding by a factor of 2. Ultimately, the termination criteria shown in Line 8 will become true, and the algorithm will terminate after declaring a leader.

Let us now prove that it is not possible for two or more leaders to be elected simultaneously. Assume that processes with IDs `i` and `j` are elected as leaders. Let $\texttt{i} < \texttt{j}$ without loss of generality. It is obvious that when `j`'s ⟨`probe`⟩ message reaches `i`, it will be absorbed, whereas the reverse will not happen. Hence, it is not possible to elect the process with `j` as a leader.

We will thus always have one leader elected as per this algorithm.

Message Complexity

Let us first consider the message complexity from the point of the view of the process that will be elected as a leader. `k` will vary from 0 to $log(n)$, where n is the total number of nodes. For every hop that a ⟨`probe`⟩ message traverses in one direction, it traverses one hop in the opposite direction as well. There are two associated replies as well. Hence, the total number of messages is:

$$
\begin{aligned}
4 \times \sum_{k=0}^{log(n)} 2^k &= 4 \times (2^{log(n)+1} - 1) \\
&= 4 \times (2n - 1) \\
&= 8n - 4
\end{aligned}
\tag{5.12}
$$

The leader thus requires $8n - 4$ messages to be sent over the network. However, we also need to count the number of messages that the rest of the processes send. Here the assumptions matter. Let us assume a synchronous algorithm where we treat every phase as a round.

- The maximum number of winners after k phases is as follows:
  - Two winners need to at least be 2^k entries apart.
  - Thus, the total number of winners after k phases is limited to $n/(2^k + 1)$
- The total number of messages for each initiator in phase k is 4×2^k
- Hence, the total number of messages in the k^{th} phase is:

  $$4 \times 2^k \times \frac{n}{2^{k-1} + 1}$$

  Note that we need to count the number of winners after the $(k-1)^{th}$ phase. This is $n/(2^{k-1} + 1)$, which is defined for all $k \geq 1$.
- The total number of messages is therefore limited by the following equation:

$$
\begin{aligned}
M &< \sum_{k=1}^{log(n)} 4 \times 2^k \times \frac{n}{2^{k-1} + 1} \\
&= 4n \times \sum_{k=1}^{log(n)} \frac{2 \times 2^{k-1}}{2^{k-1} + 1} \\
&= 8n \times \sum_{j=0}^{log(n)-1} \frac{2^j}{2^j + 1} \quad (j = k - 1) \\
&< 8n \times \sum_{j=0}^{log(n)-1} 1 \\
&= 8nlog(n) \\
&= O(n\ log(n))
\end{aligned}
\tag{5.13}
$$

We thus observe that the total number of messages sent is $O(n\,log(n))$. Recall that the message complexity was $O(n^2)$ in the Chang Roberts algorithm. In both cases, the process that is going to be elected as a leader causes $O(n)$ messages to be sent.

Tree-based Leader Election

Let us now look at a fast tree-based algorithm (source: book by Gerard Tel [Tel, 1999]). Note that we will continue to follow the same convention. `u` is the current node or process, and the other "node" or "process" is referred to as `v`.

In Algorithm 26, we define two state variables $\texttt{min}_\texttt{u}$ and `received`. The former variable records the id of the highest-priority process whose leader election request has been received by the current node (`u`). The other variable `received` records the number of leader election messages received up till now.

Each node (referred to as `u`) calls the method receiveAll. Every node waits to get messages from all its children. From a given child `v`, node `u` receives a message in each iteration. This message contains the ID of the leader `w` of the subtree rooted at `v`. Process `u` maintains an array $\texttt{rec}_\texttt{u}$ that has one entry for each child. All of its entries are initialized to **false**. The moment a message is received from a process `v`, $\texttt{rec}_\texttt{u}[\texttt{v}]$ is set to **true**. We then increment the variable `received`, and set the value of $\texttt{min}_\texttt{u}$ to $\texttt{min}(\texttt{min}_\texttt{u}, \texttt{w})$. The latter action basically means that we always set the smallest received ID as the ID of the current leader. Lower the ID of a process, higher is its priority. Once a leader election message has been received from each child, the smallest ID received ($\texttt{min}_\texttt{u}$) is sent to the parent node.

Ultimately, the root of the tree processes messages from all its children. The ID of the leader computed by the root is the leader of the entire tree (all the nodes). Note that every node got a chance to stand in the election. Once the root gets all the requests from its children, it can compute the minimum ID (including its own) and find the leader. The last phase is to broadcast the ID of the elected leader to all the nodes in the tree.

This is easily achieved by calling the method receiveFromParent. Each process `u` gets the ID of the computed leader `w` from its parent node. If $\texttt{u} = \texttt{w}$, the current node `u` is the leader. It can set its state to leader, otherwise it can set its state to lost (lost the leader election process). Next, we forward the message to each $\texttt{v} \in \texttt{children}(\texttt{u})$. When a message is received with the ID of the leader, each process calls the method receiveFromParent.

The overall algorithm is thus quite simple. The leader election requests

Algorithm 26 Leader election in trees

```
1: received ← 0
2: min_u ← u
3: procedure receiveAll
4:     while received < #children do
5:         receive ⟨w⟩ from v
6:         rec_u[v] ← true
7:         received ← received + 1
8:         min_u ← min(min_u, w)
9:     end while
10:    send min_u to parent
11: end procedure

12: procedure receiveFromParent(Leader w)
13:    if w = u then
14:        state ← leader
15:    else
16:        state ← lost
17:    end if
18:    for each v ∈ children(u) do
19:        send w to v
20:    end for
21: end procedure
```

traverse up the tree. Given that every process gets a chance to stand in the leader election, the root sees a comprehensive picture of the entire leader election process. All that it needs to do is compute the lowest process ID that it has received. This is the ID of the leader process. It just needs to be broadcast down the tree. The correctness of this algorithm is quite obvious. We ask every process and ultimately filter out the lowest process ID (among all the processes that are interested in getting elected as the leader).

Message Complexity

A tree with n nodes has $n-1$ edges. A message traverses an edge only once in a phase. There are two phases of the algorithm: accumulate the leader election requests at the root and then broadcast the ID of the computed leader to all the nodes. Each phase thus requires $n-1$ messages (one message sent per edge), and there are two such phases. Thus, the message complexity is $2(n-1)$.

5.3.2 Distributed Minimum Spanning Tree Algorithm

Given a set of distributed nodes, computing the minimum spanning tree (MST) is a very important problem. It is an important primitive that can be used to implement many other algorithms. For instance, a tree can be used to broadcast values to all processes or conduct tree-based leader election. As a result, the distributed minimum spanning tree problem is very important. Let us discuss the famous Gallager, Humblet and Spira (GHS) algorithm [Gallager et al., 1983,Tel, 1999] in this section (algorithm adapted from the version shown in [Tel, 1999] and the original paper [Gallager et al., 1979]).

Basic Results regarding MSTs

Before we design the algorithm, let us look at some basic properties of MSTs. Let us first prove a theorem regarding the uniqueness of MSTs.

Theorem 5.3.1

Assume that each edge of the graph has a unique weight. The MST is unique.

Proof: Assume this statement is false. This means that there must be two different MSTs T_1 and T_2 with the same cumulative weight.

Let edge e be in T_1 but not in T_2. Let us add the edge e to T_2. This will create a cycle. There has to be an edge e' in this cycle in T_2, which is not

present in T_1. Otherwise, the same cycle will be present in T_1 also. Given that T_1 is a tree, this can never happen. Now, given that edges have unique weights, we can conclude that the weight of e is not equal to the weight of e'. Let the function $w(e)$ represent the weight of edge e.

There are two cases.

Case I: ($w(e) < w(e')$) – We can always remove e' from T_2 and add edge e. This will lead to an MST whose weight is less than T_2. This is not possible because it is assumed that T_2 has the least overall weight.

Case II: ($w(e') < w(e)$) – Let us do the reverse in this case. Let us remove edge e in T_1 and replace it with e'. We will end up with a tree with lower cumulative weight than T_1 (which is a MST). This is not possible.

Hence, in both cases, we arrive at a contradiction. Therefore, there is a single unique MST. ■

Let us define the notion of a *fragment*. It is a subtree of an MST. Consider all the *outgoing* edges of a fragment. An outgoing edge is defined as an edge that has one node in the fragment and one node outside it. The least-weight outgoing edge (abbreviated as an LWO edge) is the edge with the least weight among all the outgoing edges of a fragment. Note that there will be only one such edge because all the edge weights are unique. Let us now prove a simple lemma.

Lemma 7

Assume a fragment F. Let e be its LWO edge. $F \cup e$ is a fragment.

Proof: Assume $F \cup e$ is not a fragment. Given that F is a fragment, it is a part of the MST. It must be connected to other nodes that are not in the fragment. Let one such edge be e'. This means that $F \cup e'$ is a fragment. Given our assumption, $e \neq e'$.

Let us now add edge e to the MST that contains fragment F. This will create a cycle given that e is not a part of the MST. This cycle will have edge e' because that connects F to the rest of the tree. In this cycle, we can always replace e' with e. The resulting structure will still remain a tree because we have the same number of edges and the structure is connected. Given that the weight of e is less than the weight of e' owing to the fact that e is the LWO edge from F, this tree will have a lower weight than the assumed MST. This leads to a contradiction.

$\therefore F \cup e$ is a fragment. ■

Overview

Initially, each process (node in the tree) is a fragment. The fragments gradually grow in size annexing neighboring nodes and edges. We can also merge fragments of a roughly similar size. This process of enlarging and merging fragments happens in parallel (concurrently) until the process terminates when only one fragment is left. That fragment is the MST.

An astute reader will agree that the Prim's algorithm [Cormen et al., 2009] works conceptually in a similar manner. Initially, we start with a one-node fragment. The fragment is gradually enlarged, one node at a time until the fragment cannot be grown any further. This is the MST. It is important to take into account that the edge that we add to the fragment is as per Lemma 7 – it is the LWO edge between the fragment and the nodes in the rest of the graph. The Prim's algorithm has an elaborate mechanism for finding such an edge. A min-heap is maintained to find this edge. Sadly, in the Prim's algorithm, we add one node at a time (via the LWO edge). The algorithm is strictly sequential in nature, which militates against the ethos of pure distributed computing.

We desire a distributed algorithm that operates on the nodes of the tree concurrently. We can always find the LWO edge of a fragment concurrently, add it to the fragment, and grow it by one node. However, that is not enough. We need to look for more ways to enhance parallelism. A truly parallel algorithm should not add one node at a time – it should take much larger steps. Typically, we want such algorithms to run in $O(log(n))$ time. This is possible when we have a tree-structured computation where we merge subtrees (fragments) of the same size and keep doing the same. We thus need to store the approximate size of a fragment, and we need to be able to merge two fragments of the same size to create a larger fragment that is roughly twice the size of the original fragments. If we can do this frequently enough, we can achieve MST creation in approximately $O(log(n))$ time.

We implement this as follows. Each fragment has a name and a level. Consider two fragments F_1 and F_2. If $level(F_1) < level(F_2)$ and fragment F_1 wants to combine with F_2, we allow this to happen. In this case, all the nodes in F_1 take up the name of F_2. This is like a smaller fragment completely getting absorbed in a larger fragment.

If $level(F_1) = level(F_2)$, then we can merge both the fragments as equals. We increment the level of the nodes in F_1 and F_2 by 1. The nodes in the merged fragment $F_1 \cup F_2$ take up a new name. Let us formalize this into two combining rules. The context is set in Figure 5.4 where there are two fragments F_1 and F_2. Assume that some node in F_1 initiates the process

of merging. Let L_1 be the level of F_1 and L_2 be the level of F_2. Let the LWO edge from F_1 be e_{F_1} and from F_2 be e_{F_2}, respectively. If $L_1 = L_2$ and $e_{F_1} = e_{F_2}$, it means that two fragments with equal levels want to merge. Their LWO edges are the same. This means that we can happily merge the two fragments F_1 and F_2 and create a larger fragment $F_1 \cup F_2$. In this case, we increment the level of all the nodes in the merged fragment by 1. The name of the new merged fragment is the ID of the common LWO edge ($e_{F_1} = e_{F_2}$). Note that if the LWO edges are not the same, then we do not let the fragments merge even if their levels are equal.

The last case arises when a fragment with a higher level wants to merge with a fragment with a lower level. This is not allowed. A large fragment cannot gobble smaller fragments. We will see that this can effectively sequentialize the algorithm. Hence, this is prohibited.

Let us create two rules to capture our discussion regarding merging fragments: Rule LT (less than) and Rule EQ (equal to).

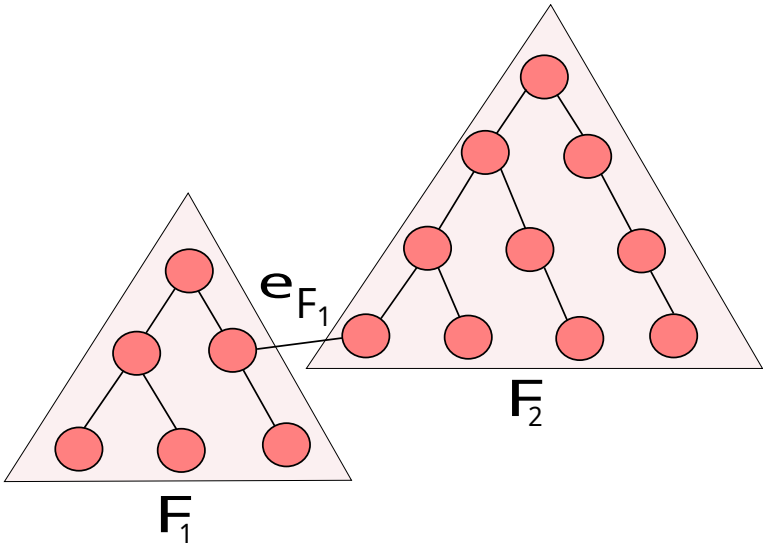


Figure 5.4: Two fragments with a common least-weight edge

Rule LT If $L_1 < L_2$, then we combine F_1 and F_2. All the nodes in the new fragment acquire the name of F_2 and level L_2.

Rule EQ If $L_1 = L_2$ and $e_{F_1} = e_{F_2}$, the two fragments combine as follows:

- The level of all the nodes in the merged fragment $F_1 \cup F_2$ is $L_1 + 1$.

- The name is set to the LWO edge e_{F_1} $(= e_{F_2})$.

The Algorithm

Messages and Data Structures

Each node has three states: asleep, find and found. It starts with the quiescent asleep state. Once the protocol starts, each node enters the find state and then starts finding the LWO edge. Once the LWO edge is found, it transitions to the found state and then tries to combine using the *EQ* or *LT* rules. After the fragment has combined, it is time for the bigger fragment to find its LWO edge once again. Every node in the merged fragment enters the find state.

Each edge in the graph of nodes is tagged with a status. Initially, all the edges are marked basic. This means that their status is not determined as yet. Once an LWO edge of a fragment is found, we know that it is a branch in the MST (as per Lemma 7). Sometimes, the algorithm tries to connect with another node that already belongs to the partially constructed MST. This edge will induce a cycle. Hence, it is marked as reject.

Each node has the following fields: name, level, parent and a bunch of temporary variables. The parent in this case is a complicated concept as we shall see later. The temporary fields maintain bookkeeping information such as the least-weight edge and node found up till now. These are required to compute the LWO edge of the fragment.

Initialization

Algorithm 27 shows the initialization procedure. Here, the convention is that the current node is `u` and any other node is represented as `v`. Let `u` $\rightarrow$ `v` (`uv`) be the least weight outgoing edge from `u`. Given that every node is a fragment in itself, by Lemma 7 `uv` is a branch in the tree. Hence, we set the status of the `uv` edge as branch. The level of each node is initialized to 0 and the `state` of node `u` is set to found. The number of received messages `rec` is set to 0. This is the initial state of each node. Subsequently, each node sends a ⟨`connect`⟩ message to `v`. `u` sends its current level, 0. The reason that a ⟨`connect`⟩ message is sent is that `v` needs to know that `uv` is `u`'s LWO edge. Once a node `v` receives a ⟨`connect`⟩ message, it invokes the receiveConnect procedure.

Algorithm 27 The initialization procedure

```
 1: procedure INITIALIZE
 2:     ▷ uv is the LWO edge from u
 3:     status[v] ← branch                                  ▷ Edge in the MST
 4:     level ← 0
 5:     state ← found
 6:     rec ← 0
 7:     send ⟨connect, 0⟩ to v
 8: end procedure
 9:
10: procedure RECEIVECONNECT(receive ⟨connect, L⟩ from v)
11:     if L < level then                                   ▷ Combine with rule LT
12:         status[v] ← branch
13:         send ⟨initiate, level, name, state⟩ to v
14:     else if status[v] = basic then
15:         WAIT ()
16:     else if L = level and status[v] = branch then       ▷ Rule EQ
17:         ▷ uv is the combining edge
18:         send ⟨initiate, level + 1, uv, find⟩ to v
19:     end if
20: end procedure
```

Let us assume that node u receives a ⟨connect⟩ message from node v whose level is L. The first case considers the case where L is less than the current level of u. In this case a smaller fragment wants to merge with a larger fragment. The smaller fragment can be absorbed using the *LT* rule. The status of edge uv becomes branch. Once, the connection has been established, it is important to let the smaller fragment know that its request for connection has been accepted. The level of all the nodes in the smaller fragment needs to change to that of the larger fragment and the name should also change to that of the larger fragment.

Now, let us consider the rest of the two cases. In Line 13, we consider the case where we have an asymmetry: the least-weight edge from v to u is not the same as the least-weight edge from u to v. This is indicated by the fact that the status of the edge is basic at the time of checking. This can either indicate that the current node has not completed the initialization process or an asymmetry indeed exists. In either case, rule *EQ* is not applicable. If the current node u has not completed the process of discovering which edge is its fragment's LWO edge, then that process needs to complete first.

Assume that the uv edge is indeed found to be the LWO edge. Then, we have a symmetric situation. As a result, a ⟨connect⟩ message will be sent back; however, this time the status of the node at v will not be basic, it will change to branch, and thus the process of merging will succeed in accordance with the *EQ* rule (subject to the level being equal).

The process of merging is realized in Line 18, which we enter if the levels of both the fragments are the same. In this case, we can apply rule *EQ*. An ⟨initiate⟩ message is sent on the edge uv with an incremented level. The status find is also sent because it is a message to the other fragment that after it is merged, it needs to participate in the process of finding the LWO edge of the combined fragment. This edge is known as the *combining edge*.

Processing the ⟨initiate⟩ Message

When an ⟨initiate⟩ message is received, the procedure RECEIVEINITIATE (Algorithm 28) is invoked. This basically means that a neighboring node has accepted a ⟨connect⟩ message because of either rule *LT* or *EQ*, and it is responding in the affirmative.

The initiating node sends its current level, name and state (level′, name′, state′), respectively. The level, name and state of the current node are set to the respective values of the neighboring node (respectively) in (Line 2 of Algorithm 28). Now, the aim is to propagate this information to the rest of the nodes of the fragment.

Now, note that every node needs to have a parent. It is initialized to the node itself. However, the way the parent is subsequently set is quite interesting (see Line 3). In this case, given that u got the ⟨initiate⟩ message from v, v is set to u's parent. If we are combining with the *LT* rule then it is clear that a subtree is becoming a part of a larger tree. The more interesting case is the *EQ* rule. In this case, the LWO edge has to be the same for both the fragments. If the LWO edge is edge uv, u is set as v's parent and vice versa. In fact, after initialization, this will continue to be the case for all fragments whose two constituent sub-fragments were merged with the *EQ* rule. However, the structure will change once this fragment merges with some other fragment.

The next step is to propagate the update indicating that the fragment's name and level have changed to all the nodes in the combining subtree. Prior to doing so, we reset a few node-specific state variables: leastWtNode (node in the fragment that is one end of the LWO edge), leastWt (weight of the LWO edge found up till now) and testNode (defined later). We forward the ⟨initiate⟩ message to each neighbor of u (other than v).

Once this has been done, then it is time to call the procedure FINDMIN,

Algorithm 28 Receipt of the initiate message

```
1: procedure RECEIVEINITIATE( ⟨initiate, level′, name′, state′⟩, v)
   ▷ Set the state
2:     (level, name, state) ← (level′, name′, state′)
3:     parent ← v                          ▷ Set the parent to the initiator

   ▷ Propagate the update
4:     leastWtNode ← φ
5:     leastWt ← ∞
6:     testNode ← φ
7:     for each r ∈ neigh(u): (status[r] = branch) ∧ (r ≠ v) do
8:         send ⟨initiate, level′, name′, state′⟩ to r
9:     end for

   ▷ Find the LWO edge
10:    if state = find then
11:        rec ← 0
12:        FINDMIN()
13:    end if
14: end procedure
```

whose job is to find the LWO edge of the fragment.

Algorithm 29 The FINDMIN method

```
1: procedure FINDMIN
2:     if argmin_{v∈neigh(u)∧status[v]=basic}(w(uv)) ≠ φ then
3:         testNode ← v
4:         send ⟨test, level, name⟩ to testNode
5:     else
6:         testNode ← φ                          ▷ No more neighbors left
7:         PROCESSMIN()
8:     end if
9: end procedure
```

Here again, the current node is u. We look at all the neighbors of u and find the edge with the least weight as long as its status is basic. Let us assume that there is such a node v. We set it as the testNode. Then, we send a ⟨test⟩ message to it to explore a possible fragment merging opportunity. If this is not the case, then we record the fact that no such testNode was found. We then report the same by calling the PROCESSMIN method.

Processing a ⟨test⟩ Message

The method RECEIVETESTMSG (see Algorithm 30) is invoked when a node receives a ⟨test⟩ message. The key arguments are the ID of the message sender v, its name and level. The latter two are name′ and level′, respectively. If level′ > level, it means that v (the sending node) has a higher level. We do not allow such a merge as per the *EQ* and *LT* rules. Thus, there is no option but to wait.

Let us consider the rest of the cases where the level of v is less than or equal to the level of u, and the fragment names are the same. It clearly means that the message was sent from one node of the fragment to another. It is thus an internal edge. We set its status to reject (cannot be an edge of the MST) if it was hitherto unlabeled (the state was basic).

Let us subsequently draw our attention to Line 8, which follows this check and action. In this case, we test whether v is equal to testNode. We are basically checking if there is any outstanding ⟨test⟩ message that has already been sent to v. If they are not equal, we send a ⟨reject⟩ message to v. The reason for this is that if they are equal then v is the testNode, and it would have gotten a ⟨test⟩ message from the current node u already. v would inevitably realize that uv is an internal edge, and

it would mark the status of the edge as reject. There is no need for sending any additional message. Furthermore, if v = testNode, then we can proceed with the findMin method because the current attempt with v has failed. This method will in turn find the next LWO edge and send it a ⟨test⟩ message.

Algorithm 30 Receive a test message

```
1:  procedure receiveTestMessage(⟨test, level′, name′⟩, v)
2:      if level′ > level then
3:          wait()
4:      else if name = name′ then                    ▷ This is an internal edge
5:          if status[v] = basic then
6:              status[v] ← reject
7:          end if
8:          if v ≠ testNode then
9:              send ⟨reject⟩ to v
10:         else
11:             findMin()
12:         end if
13:     else
14:         send ⟨accept⟩ to v                       ▷ different fragment
15:     end if
16: end procedure
```

The last case that we need to consider is when the names of the respective fragments of u and v are not the same. In this case, the edge uv is acceptable. This not a fragment's internal edge. It is the LWO edge from u to nodes in other fragments. We need to accumulate all such edges and compute the LWO edge for the entire fragment. Given that uv edge has been found to be acceptable, we send an ⟨accept⟩ message to v. This tells v that the edge uv is acceptable (not an internal edge and the LT and EQ rules can potentially be applied). Let us look at both the ⟨reject⟩ and ⟨accept⟩ responses (resp.).

Processing ⟨accept⟩ and ⟨reject⟩ Responses

Whenever a node receives a ⟨reject⟩ message (method receiveReject in Algorithm 31), it simply marks the status of the edge as reject and invokes the findMin method again until it is either successful or runs out of edges. If it is successful, it receives an ⟨accept⟩ message and calls the method

receiveAccept (see Algorithm 31). Specifically, when an ⟨accept⟩ message is received from v, we set the testNode to v. Given that the edge uv has been found to be acceptable, we can compare it with the least weight (leastWt) found up till now. If it is smaller, then we set leastWtNode to v and leastWt to w(uv) (weight of the edge uv). This action is then reported to the root node by calling the processMin method.

Algorithm 31 The receiveReject and receiveAccept methods

```
1:  procedure receiveReject(v)
2:      if status[v] = basic then
3:          status[v] ← reject
4:      end if
5:      findMin()                          ▷ Keep searching for the LWO edge
6:  end procedure
7:
8:  procedure receiveAccept(v)
9:      testNode ← φ                                  ▷ Reset the testNode
10:     if w(uv) < leastWt then
11:         leastWt ← w(uv)
12:         leastWtNode ← v
13:     end if
14:     processMin()          ▷ Convey this information to the root node
15: end procedure
```

Algorithm 32 The processMin method

```
1: procedure processMin
2:     cnt ← | {v : status[v] = branch ∧ v ≠ parent } |
3:     if (rec = cnt) ∧ (testNode = φ) then
4:         state ← found
5:         send ⟨report, leastWt⟩ to parent
6:     end if
7: end procedure
```

The code of the processMin method is shown in Algorithm 32. A node waits to get a message regarding the least-weight edge from all its children (in the fragment). This happens as follows. cnt is a temporary variable that represents the number of children in the fragment. rec is a node-specific variable that is incremented when a message with the least-weight edge is

received from a child. This will be discussed later when we talk about the RECEIVEREPORT method. If `rec` = `cnt`, it means that a message regarding the least-weight edge has been received from each child.

We also test if `testNode` = ϕ, which means that the current node has finished its least-weight edge search process. If both the conditions are true, then we can set the state to `found`. This node is done. `leastWt` can be reported to the parent by sending the `report` message to it (Line 5).

Reporting the Best-Weight Edge

Algorithm 33 The `receiveReport` method

```
1:  procedure RECEIVEREPORT(v, ω)
2:      if v ≠ parent then
3:          if ω < leastWt then
4:              leastWt ← ω
5:              leastWtNode ← v
6:          end if
7:          rec ← rec +1
8:          PROCESSMIN()
9:      else
10:         ▷ v = parent
11:         if state = find then
12:             WAIT()
13:         else if ω < leastWt then
14:             CHANGEROOT()
15:         else if (ω = leastWt = ∞) then
16:             STOP()
17:         end if
18:     end if
19: end procedure
```

The RECEIVEREPORT method's code is shown in Algorithm 33. There are two arguments: the node sending the message (`v`) and the LWO edge in its subtree (whose weight is ω). There are two cases.

The first case is when `v` is not the current node's parent. This is the regular case, where a child node reports its least-weight edge to its parent. In this case, we just update the `leastWt` variable if $\omega <$ `leastWt` and set `leastWtNode` to `v`. We also increment `rec` (number of report messages received), and then call the PROCESSMIN method. There is a need to invoke

this method after every report message is received from a child node. Once we get messages from all the children and the current node has explored its edges, we send a `report` message to the parent (part of the processMin method).

Now, let us consider the next case when `v` is the parent of the current node `u` (Line 10). A node will receive a message from its parent only if it is a *combining edge* (see the description of the method in receiveConnect). In this case, if the state is find, then it means that the current node is not done with searching for the least-weight edge itself. There is a need to wait. Let us consider the other case where the current node is done with finding the LWO edge in its subtree.

Otherwise, we compare ω with `leastWt`. If $\omega <$ `leastWt`, then it means that the LWO edge is in the subtree rooted at `v`. We can thus proceed to change the root of the tree by calling the method changeRoot. Note that in this case the root of the tree for the entire fragment will be set to the node of the LWO edge (in the fragment). This is the role of the method changeRoot.

It is of course possible that no least weight edge is found because the tree is fully formed. In this case $\omega =$ `leastWt` $= \infty$. At this point, the algorithm can stop – the MST has been computed.

Changing the Root

Algorithm 34 The changeRoot method

```
1:  procedure changeRoot
2:      if status[leastWtNode] = branch then
3:          send ⟨changeroot⟩ to leastWtNode
4:      else
5:          ▷ Proceed along the LWO edge
6:          status[leastWtNode] ← branch
7:          send ⟨connect, level⟩ to leastWtNode
8:      end if
9:  end procedure

10: procedure receiveChangeRoot
11:     changeRoot()
12: end procedure
```

The pseudocode of the changeRoot method is shown in Algorithm 34.

If `status[leastWtNode]` is branch, then it means that we have not arrived at the LWO edge yet. There is a need to thus send this information towards the LWO edge (starting with `leastWtNode`). We thus send the message ⟨`changeroot`⟩ to `leastWtNode`. It receives this message and calls its CHANGEROOT method (see Line 11).

Now consider the other case when `status[leastWtNode]` is not branch. In this case, we have arrived at the LWO edge. Given that the LWO edge from any fragment is a part of the MST (see Lemma 7), we can set the value of `status[leastWtNode]` to branch. This edge is a part of the MST now. We can initiate the process of connecting the fragment that contains `leastWtNode`. It will be added as a part of the LT or EQ rules at some point of time.

Proof

We always find the LWO edge of a fragment as follows. Every node in the fragment finds its outgoing LWO edge on its own. It verifies that the other end of the LWO edge is not in its fragment, and after it receives an ⟨`accept`⟩ message from it, it reports the ID of the edge to its parent. All the nodes in the fragment collect this information and relay this information to the root nodes. The job of a root node is to compute the global minimum and decide the LWO edge.

Subsequently, the fragment tries to combine using the LWO edge. The request may wait for some time. Finally, the fragment shall combine as per the LT or EQ rules. As per Lemma 7 this is how a fragment grows. Note that we are continuously making bigger and bigger fragments. Ultimately, this process will stop when the full MST has been built.

Message Complexity

Let us make an assumption. Given a level k, assume that the fragment contains at least 2^k nodes. This trivially holds true when $k = 0$. When there is a combination with the LT rule, the level does not change. Hence, our assumption still holds true. Now consider combinations using the EQ rule. Let us assume by induction that the assumption holds. Let their levels be equal to k. The total number of nodes in the merged fragment (with level $k+1$) is at least 2^{k+1}. Hence, the assumption still holds. Using mathematical induction we can say that the results hold for all k. If $k = log(N)$, the size of the fragment will at least be N. The fragment becomes the MST. We thus observe that per node, there will be a maximum of $O(log(N))$ level

changes. For all the nodes, the maximum number of level changes will be limited to $O(Nlog(N))$.

Let us count the number of unsuccessful messages. Consider the edges. A node is rejected only once (for a given edge). A ⟨`test`⟩ message is sent, and the response is a ⟨`reject`⟩ message. Thus, a total of $2E$ messages (at most) are sent in this phase.

Next, let us count the successful messages. Consider the number of messages a node sends or receives for a given level. At the most, we receive a single ⟨`initiate`⟩ message, receive an ⟨`accept`⟩ message, send a ⟨`report`⟩ message, send a ⟨`changeroot/connect`⟩ message and a ⟨`test`⟩ message. Therefore, the total number of messages in this category is limited to $5Nlog(N)$.

Hence, the total message complexity is $O(2E + 5Nlog(N))$.

5.3.3 Chandy Lamport Snapshot Algorithm

Algorithm 35 Chandy Lamport Algorithm

```
1:  procedure INITIALIZE
2:      Take local snapshot
3:      taken ← true
4:      for v ∈ neigh(u) do                    ▷ u is the current node
5:          send ⟨mkr⟩ to v
6:      end for
7:  end procedure

8:  procedure RECEIVE(marker)
9:      if taken = false then
10:         Take local snapshot
11:         taken ← true
12:         for v ∈ neigh(u) do                ▷ u is the current node
13:             send ⟨mkr⟩ to v
14:         end for
15:     end if
16: end procedure
```

The Chandy Lamport algorithm (Algorithm 35) is a very popular algorithm for collecting a snapshot in a distributed system. Ideally, a snapshot should be like a photograph of the entire distributed system collected at exactly the

same point of time. However, this is almost impossible to collect in practice unless there is dedicated hardware support in the distributed system. Hence, we need to create an alternative definition of a snapshot that can be practically collected in a modern distributed system. A distributed snapshot should contain the local states of all the processes and some record for all the messages in flight. Here, we define the notion of a *consistent snapshot*. If we have the record of a message receipt, then the corresponding send event should also have been recorded. In other words, we do not want to have orphan messages where the snapshot records that it has been received by some process, yet the sender's information is not recorded. Such snapshots are said to be *consistent*. If we do not have such a caveat, then we will be breaking causality, where the effect is recorded, but the cause is not.

The second requirement for a snapshot is that it should be atomic. This is not relevant for systems where all messages have a single sender and a single receiver (point-to-point). However, in system that define multicast and broadcast messages, the message needs to appear to be atomic. The following options are permissible for a snapshot: either no receive event has been recorded or all the receive events have been recorded. It is never the case that a snapshot contains the record of only a partial subset of receivers receiving the message.

We also need to consider whether messages are allowed to overtake each other or there are restrictions. A lot of distributed algorithms including the Chandy Lamport algorithm assume that between a given source node and destination node, messages should arrive in FIFO (first-in first-out) order. This means that for a given source-destination pair, messages cannot be reordered. This is easy to ensure in practice. Every message has a sequence number. Each source node maintains the sequence number of the last message sent to each destination that it can send a message to. This sequence number is incremented (like a Lamport clock). Note that there is no need to have a direct link between the source and destination nodes.

Each receiver maintains a list of "expected" sequence numbers for each source node. Let us explain the notion of an "expected" sequence number with an example. Assume that from source node i messages with sequence numbers $1 \ldots 5$ have been received. This means that a message with sequence number 6 is expected. Thus, the list of expected sequence numbers $S_i = \{6\}$. Now assume that a message with sequence number 8 arrives. This means that the message with sequence number 7 has been delayed. At this point, $S_i = \{6, 7, 9\}$. The receiver cannot process the message with sequence number 8. This will violate the FIFO property. We need to wait for the sequence number 6 to be received from i. This message can then be processed

by the receiver and 6 can be removed from S_i. It is easy to observe that this method ensures that messages are processed in FIFO order.

Definition 5.3.1 Distributed Snapshot

A distributed snapshot represents the state of the distributed system collected across all the processes at roughly the same point of time. It comprises the local state of all the processes, and the details of all the messages sent and received. A snapshot should be *consistent*, which means that if a message receive has been recorded, then the corresponding send event should have been recorded. Furthermore, in systems that support multicast and broadcast operations, snapshots should be atomic. This means that no event should be partially visible. Either all the receivers should receive a message or none of them should.

The Chandy Lamport was designed for a system that supports point-to-point messages. Every node has a list of neighbors. There is one node that initializes the snapshot creation process. It takes its local snapshot and sends a marker message ($\langle$`mkr`$\rangle$) to each of its neighbors. They then start snapshot collection.

Each such node in the graph of processes first checks if it already has taken a snapshot or not. It is possible that a node may get marker messages from multiple neighbors. If a snapshot has not been collected, then it takes a local snapshot and sets the local variable `taken` to **true**. It proceeds to do the same. It sends the $\langle$`mkr`$\rangle$ message to each of its neighbors. The algorithm finally terminates on its own. At that point every process would have taken its snapshot. All the local snapshots that includes details of sent and received messages can be combined to create a global snapshot.

Technically speaking, the global snapshot is not collected at the same point of time. It is safe to say that the local snapshots that are a part of it are collected at roughly the same point of time. We shall prove in Theorem 5.3.2 that the Chandy Lamport algorithm only collects consistent snapshots. The system can be restarted from a snapshot seamlessly. Special attention needs to be paid to messages that are recorded as sent but they haven't been recorded as received. If the messages are still in flight, then we can wait. Otherwise, there is a need to send the messages again.

Theorem 5.3.2

The Chandy Lamport algorithm collects consistent snapshots.

Proof: Assume that this is not true. This means that there is some message whose receipt has been recorded but the corresponding sent event has not been recorded.

Assume node u sent message ⟨m⟩ to node v. ⟨m⟩ is present in v's snapshot, but it is not present in u's snapshot.

Consider the point of time when u receives the ⟨mkr⟩ message for the first time. It takes a local snapshot. At this point of time ⟨m⟩ has not been sent. Otherwise, it would have been a part of u's local snapshot. Subsequently, u sends the ⟨mkr⟩ message to v. If v hasn't already taken a snapshot, then it takes a snapshot immediately. Given that messages from u are received in FIFO order at v, it is not possible for the ⟨m⟩ message to have arrived at v by this time. It has to be sent after the ⟨mkr⟩ message is sent by u and it will also be received after v receives mkr from u. Thus, it cannot be part of v's local snapshot. We thus have a proof by contradiction here. The global snapshot is consistent. ■

5.3.4 Termination Detection Algorithms

From the previous section, we see that algorithms designed for the asynchronous model of distributed systems have to also contend with the question of when and how a node knows about terminating its computation. In the synchronous model, this question is easy to answer as nodes terminate when the specified completion condition is reached or the specified number of rounds have completed. In an asynchronous setting, checking this condition may depend on the receipt of messages from other nodes. Since the delivery of messages does not follow strict time guarantees, nodes do not easily know when they can terminate their computation.

In general, the question of when a distributed computation can terminate is non-trivial. In this chapter, we will study algorithms that help in detecting termination. In a map-reduce style computation, when the destination process has received a given number of messages and it has processed all of them, we can be sure that the entire algorithm has terminated. However, in many cases, detecting termination is non-trivial. There is no well-defined termination point. In this case, it is necessary to run a separate protocol, and find out if the original algorithm has terminated or not. It should preferably be a generic mechanism that works for all kinds of asynchronous algorithms.

Let us introduce the key concepts. Every process has two states: active and passive. An active state indicates that either the process is doing some local computation or has sent a message and is waiting for an acknowledge-

ment. Otherwise the state is passive. Initially, the state of all the processes is passive. Let us now consider each communication channel between processes. An active channel has an outstanding message that needs to be delivered to its destination. On the other hand, a passive channel does not contain any messages. The condition for termination is that after a process has become active, the entire system becomes passive. This means that all the processes are passive and all the channels are empty. The termination indicates that the entire system has come to a standstill and there is no activity. Let us note that there are two algorithms here. One is the original algorithm. Let us refer to all the messages that are sent as a part of its operation as basic messages. Then there is the termination detection algorithm that requires processes to send and receive messages. These are known as control messages. The termination detection algorithm clearly cannot influence the behavior of the original algorithm. Its job is to simply execute in the background and signal termination.

Dijkstra-Scholten Algorithm

Let us discuss a classical algorithm (Algorithm 36) in this space namely the Dijkstra-Scholten algorithm [Dijkstra and Scholten, 1980]. All the processes start as passive. One of them becomes active and starts the original algorithm. At the same time, the termination detection algorithm begins its operation. It finally declares the computation to have terminated once the entire system is quiescent – all the processes and communication links become passive. The messages sent by the original algorithm are referred to as *basic* messages, whereas the messages exclusively sent by the termination detection algorithm are known as *control* messages.

The function SENDMSG is used to send a message in the original algorithm from process `u` to `v`. It comprises two steps: send the message and increment `u.deficit[v]`. `u.deficit[v]` represents the number of unacknowledged messages sent from `u` to `v`.

When a message is received by a process, it invokes the function RECVMSG. Assume a process `u` receives from process `v`. It is important to first assess the state of `u.active`. If it is **false**, then the parent of `u` is set to `v` and then `u.active` is set to **true**. This means that when a passive process receives a message, it becomes active. Otherwise, if the process was already in the active state, then an ⟨`ACK`⟩ message is sent back to the sender (`v` in this case). The message can subsequently be processed. When a process receives an `ACK` message, its original message gets acknowledged. Hence, `deficit[v]` can be decremented now. There is one less unacknowledged

Algorithm 36 Dijkstra-Scholten Algorithm

```
1:  procedure SENDMSG(u sends ⟨m⟩ to v)
2:      Send ⟨m⟩ to v
3:      u.deficit[v] ← u.deficit[v] + 1      ▷ # unacknowledged messages
4:  end procedure

5:  procedure RECVMSG(u receives ⟨m⟩ from v)
6:      if u.active = false then
7:          u.parent ← v
8:          u.active ← true
9:      else
10:         Send ⟨ACK⟩ to v
11:     end if
12:     Process message ⟨m⟩
13: end procedure

14: procedure RECVACK(u receives an ⟨ACK⟩ from v)
15:     deficit[v] ← deficit[v] - 1      ▷ Decrement # outstanding ACKs
16: end procedure

    ▷ Propagate ACK s towards the root
17: procedure BECOMEIDLE(u)
18:     if (u.active = true) ∧ (∀v : deficit[v] = 0)  then
19:         if u is not the root then
20:             Send ⟨ACK⟩ to u.parent
21:             u.active ← false
22:         else
23:             Declare termination
24:         end if
25:     end if
26: end procedure
```

message now. This means that outstanding acknowledgements ultimately remain for parent-child relationships. All other communications are immediately acknowledged.

Another point that needs to be kept in mind is that a tree is being created here. A parent-child relationship is being created in Line 7. Let us prove that this will lead to a tree. First, let us prove that there are no cycles. u is a child of v, if v wakes up u from a passive state. This means that v woke up from the passive state before u. An edge of the form u_1 to u_2 (u_1 is u_2's parent) is a happens-before relationship; it means that u_1 woke up before u_2. We cannot have a cycle of happens-before relationships. Hence, the structure is acyclic. It cannot be a DAG (directed acyclic graph) because every child has a single parent. Hence, it is a tree.

Now, consider the situation where a process u has no work to do. It calls the function BECOMEIDLE. If there are any outstanding acknowledgements, i.e., $\exists v : \texttt{deficit}[v] \neq 0$, then the process is not truly idle yet. It is waiting for an outstanding acknowledgement from some other process that it has sent a message in the past. In this case, it waits. Assume a situation arrives when a node has received all its acknowledgements, which means that $\forall v : \texttt{deficit}[v] = 0$. Then we do further processing. First, consider the case when u is not the root. In this case, an acknowledgement is sent back to the parent. In other words, nodes in the tree send back $\langle \texttt{ACK} \rangle$ messages to their parent node. Note that this did not happen during the normal course of execution that preceded this phase. In the normal course of execution, $\langle \texttt{ACK} \rangle$ messages were only sent by non-child nodes. After sending an $\langle \texttt{ACK} \rangle$ to the parent, u can also mark itself to be passive (u.active $\leftarrow$ **false**). This basically means that u has finished its computation, and so has the subtree rooted at it. This process continues and propagates towards the root.

Finally, we reach a situation where the root process becomes idle. This means that the entire tree has finished its computation and there are no live messages. All the processes have become passive. At this stage termination can be declared.

Message Complexity

For every basic message sent by the original algorithm, a control message is sent by the termination detection algorithm. Let us elaborate. When a message is sent to an active process, it is immediately acknowledged. If it is sent to a passive process, then an acknowledgement is sent later, once it becomes idle. Nevertheless, for every basic message, an acknowledgement ($\langle \texttt{ACK} \rangle$) is sent, either immediately or later. Hence, the additional message

overhead is 100%.

Safra's Algorithm

The Dijkstra-Scholten algorithm relies on a tree that gets constructed implicitly during the execution of the algorithm. Nodes in the tree signal the completion of their execution to their parent, and this message is propagated till the root. Ultimately, when the root finishes its work, the entire system becomes quiescent and the computation terminates. Sadly, the message overhead is quite high (100% to be precise).

The Safra's algorithm uses a different paradigm. Even thought the network can have a generic topology, the termination detection algorithm assumes a ring of processes. The processes circulate a token. The `startNode` initially has the token. Once the token returns back to the initiator node (`startNode`) without encountering any activity, we can conclude that the network is quiescent and the computation has terminated.

Let us introduce the state variables. Each process has a local count and a local color. The count is initialized to 0 and the color is initialized to white. The count is equal to the number of messages sent minus the number of messages received. The sum of the counts across all the processes should be zero when there are no in-flight messages. A token circulates among the processes. The token too has a count and a color; they are initialized to zero and white, respectively. The significance of the color will be revealed shortly. It is basically present to resolve race conditions caused by concurrency.

The sendMsg (see Algorithm 37) is used to send a message in the original algorithm. Let process `u` send a message to process `v`. Every time a message is sent, `u.local` is incremented. Similarly, when a message is received at any node along the ring, the process's state is set to active, `u.local` is decremented. The logic is very clear. The sum of local counts will always be zero in the steady state, i.e., when there are no messages in flight. However, if there are messages in flight, then the sum of counts will be non-zero. We will precisely use this insight to find out if there are messages in flight. There are two kinds of in-flight messages. The first set of messages are sent to nodes between `u` and the `startNode` along the ring. The second set of messages are sent to the rest of the nodes. The first set of messages are easier to process, as we shall see in Algorithm 38. In this case, all in-flight messages are accounted for correctly. However, it is much harder to account for messages that belong to the second set. It is possible to miss them altogether. This is why there is a need to maintain some additional state.

Let us clarify. The relative position of `v` with respect to `u` along the ring

Algorithm 37 Safra's Termination Detection Algorithm

```
1:  procedure sendMsg(u sends ⟨m⟩ to v)
2:      u.local ← u.local + 1
3:      if isBehind(v,u) then
4:          u.color ← black                          ▷ Backward send
5:      end if
6:      send ⟨m⟩ to v
7:  end procedure

8:  procedure recvMsg(u receives ⟨m⟩ from v)
9:      u.active ← true
10:     process (⟨m⟩)
11:     u.local ← u.local - 1
12:     tryPassive()
13: end procedure

14: procedure tryPassive
15:     if noLocalWork () then
16:         u.active ← false
17:     end if
18: end procedure
```

is important. We are precisely interested to answer the following question. Can the token pass from `u` to `v` without visiting the `startNode`? If it can, then `v` is said to be *ahead of* `u`, otherwise it is *behind* `u`. A send or receive event is said to be recorded when the token passes through the node that either sent or received the message (resp.).

In Line 3, we check if `v` is behind `u`. There is a need to do so for the following reason. When a token visits a node whose color is white and it is inactive, it effectively extracts a promise from it that the node will remain quiescent and will not create any additional activity unless it receives an external message. If all the nodes in the ring make this promise, then it can be inferred that the network is quiescent. Now, assume that the token moves from the `startNode` to node `u`. All the nodes on the way make this promise. It is possible for one of them to later on break this promise if nodes like `u` send a message to them. It is not hard to visualize that this will be the case when `v` is *behind* `u`. Hence, this event is explicitly recorded in the state of the token. It is colored black.

When a message is received (RECVMSG), the process becomes active again, and its local count `u.count` is decremented. At this point, it may decide to send additional messages or become passive again.

This logic is implemented in the function TRYPASSIVE in Line 12. There is a chance that the process has immediately become passive. This possibility should be explored. If there is no local work, then `u` sets its state to passive.

Algorithm 38 shows the code of the RECVTOKEN function. When a token is received by process `u`, it adds its local count to the token's count and resets its local count (`u.local`). If the color of `u` is black, then it means that some node between `startNode` and `u` may not remain quiescent in the future. If `u` is active, it can send messages in the future. Similarly, if a message is sent to a node behind `u`, then they can become active in the future. The token has already passed through these nodes. This means that if they make a promise to remain quiescent in the future, they need to break it because they will become active again by receiving a message. This fact is recorded by changing the color of the token to black, and then `u.color` is reset to white. A black-colored token indicates that termination cannot be inferred. We wait till `u` becomes passive.

Let us now consider the case when the token arrives at the `startNode`. Line 11 shows the termination detection criterion. If at `u` (=`startNode`), the count of the token is zero and its color is white, then it means that the system is quiescent and the computation has terminated. It is easy to see why this is the case. For the computation to terminate, all the nodes have to be passive. Furthermore, the token's `count` field basically counts the number

Algorithm 38 Safra's Termination Detection Algorithm - II

```
1: procedure recvToken(u receives token T)
2:     T.count ← T.count + u.local
3:     u.local ← 0
4:     if u.color = black then                    ▷ Check if the node is black
5:         T.color ← black
6:         u.color ← white
7:     end if
8:     Wait till u is passive

9:     if u is the startNode then
10:        ▷ Termination detection
11:        if ¬ u.active ∧ T.count = 0 ∧ T.color = white then
12:            declare termination
13:            return
14:        else
15:            send ⟨Token(T.count, white)⟩ to u.next
16:        end if
17:    else
18:        send ⟨Token(T.count, T.color)⟩ to u.next
19:    end if
20: end procedure
```

of send events minus the number of receive events. If it is non-zero, then it means that there are in-flight messages and thus the computation has not terminated. Finally, if the token's color is black, it means that some activity is expected in the future. Hence, termination cannot be declared.

If termination cannot be declared, then the token's color is reset and propagated down the ring once again. Note that the `count` is not reset. The logic in Line 18 shows the logic for simply propagating the token when `u` is not the `startNode`.

Proof of Correctness

Let us prove a sequence of lemmas that help us understand the Safra's algorithm better. It is clear that the token completes a full round trip and returns back to `startNode` when all the nodes that the token has passed through have declared themselves to be passive. A passive process cannot become active on its own. It needs to receive a message from another process. We expect the token's `count` and `color` to indicate if this could have happened.

Let us thus focus our attention on messages that are in flight. There are two kinds of messages:forward and backward. A message from `u` to `v` is a *forward* message if `v` is ahead of `u`. Otherwise, it is a backward message.

First, consider forward messages. If the *send event* is recorded because the send happens before the token arrives at the node, then the count is incremented. If there is no matching receive, then it means that the system has not terminated. It is possible that the send event happens after the token has passed through the node. In this case, the receive event may be recorded or may not be recorded. The relative order of the basic message and the token will decide the outcome. Regardless of the order, let us look at the source of this message. It means that a node has been woken up after it turned passive and has passed the token. Consider the first instance at which this happens. This means that after the token passes `startNode`, for the first time a passive process that has passed the token wakes up and becomes active. It cannot wake up because of a forward messaage. This is because no node in the ring is active, and thus it cannot send forward messages. Hence, a backward message must have been sent. The node sending the backward message must have turned black.

Whenever a backward message is sent, it can wake up any node that has passed the token and has declared itself to be passive. There is a need to record this fact by recording the color of the sending process to black. Hence, the token needs to record this fact and changes its color to black. Termination cannot be declared.

Let us now quickly prove why termination is declared correctly. It means that the token has completed a round trip and its color is white. Given that no node has turned black, no backward message has been sent. This means that only forward messages are sent. Moreover, every send event is recorded in the token's count. No process that has declared itself passive and has passed the token has subsequently woken up. Hence, all send events are accounted for. If all of them have reached their destinations, then the token's `count` field is zero. This means that all forward messages are accounted for. Finally, given that the token is reaching `startNode` after a round trip, all the processes (nodes in the ring) have declared themselves to be passive. This means that unless they receive an external message, they will not participate in the original algorithm and send any basic messages. Given that there are no messages in flight, the system is *quiescent*. The system has terminated.

5.3.5 Self-Stabilization Algorithms

A lot of distributed algorithms rely on the correct functioning of the underlying system. Sometimes if there are node failures or message losses, they risk going to an illegal state from which it may be hard to recover from. We thus need a self-stabilizing algorithm where the system will reach a *legal state* in a bounded amount of time. The risk of it getting stuck in illegal states or getting into a situation such as a deadlock or livelock is near zero. If the processes have access to some form of global shared memory, then it is easy to make such guarantees. However, making guarantees about eventual correctness are hard to make in a pure message-passing based distributed system. These are the kind of systems that we have been considering up till now.

Dijkstra's Token-based Algorithm

Consider a token-based algorithm such as the Safra's algorithm. We need to rely on the following invariant: there is only one token circulating in the network at all points of time. The token is not removed from the network, and there are no duplicate tokens. There is a need to run an algorithm to continuously ensure this (shown in Algorithm 39).

Algorithm 39 Dijkstra's Self-Stabilizing Token Ring

```
1: procedure passToken
2:     ▷ u is the current node
3:     With till ⟨prev_val⟩ arrives from u.prev
4:     if u is the startNode then
5:         if pred_val = u.val then                          ▷ root enabled
6:             u.val ← (u.val + 1) modK
7:             send ⟨u.val⟩ to u.next                        ▷ pass the token
8:         end if
9:     else
10:        if pred_val ≠ u.val then
11:            u.val ← pred_val
12:            send ⟨u.val⟩ to u.next                        ▷ pass the token
13:        end if
14:    end if
15: end procedure
```

Let us explain the algorithm first. Assume that there are n processes. Consider a number K such that $K \geq n$. Each process maintains a value val. Let u be the leader process (also referred to as the startNode). A node on the ring is assumed to have the token if u.pred.val is not equal to u.val. Let us refer to such a discrepancy as a *ripple*. In case, process u's value is equal to its predecessor's value, then it is deemed to not have the token. Furthermore, the *value* of the token is the value forwarded by the predecessor to the current node u.

First, consider the leader process startNode. It checks if its value is equal to its predecessor's value. If they are not the same, then it means that it does not have the token. It also means that the previous token that it had generated has completed a full circle. Hence, it makes sense to create a new token and propagate it down the ring. This is done by incrementing the value stored in startNode (in this case process u). Note that this is a modular increment: u.val is set to (u.val + 1) mod K.

Next, consider the logic for any other process u. u compares its own value with the value received from its preceding process in the ring. If the values are unequal, then the u's value is set to its previous process's value. This means that the token has propagated by one step. This value is then sent to the next process (u.next) along the ring. This process continues until the token reaches the startNode.

Proof of Correctness

Let us now look at a brief, informal proof of correctness. Assume a case where there are no tokens in circulation. Then, sooner or later, the leader will process will wake up and execute the code on Line 5. The leader's value and its predecessor's value will be found to be the same. Consequently, a new token will be inserted by incrementing the leader's value. On the flip side, if we assume that there are $n-1$ unique token values already in circulation (one with each of the rest of the nodes), the new value is still guaranteed to be different from the rest of the values because $\texttt{K} \geq \texttt{n}$. This means that if there is no token, a new token is always inserted and if the rest of the nodes have tokens, then an new, unique token is inserted into the ring. The system will thus never be devoid of tokens.

Now, consider the case when we have multiple tokens that are alive at the same time. This can happen if there was a link failure or if the network suddenly became sluggish. As per the logic of the algorithm, each token moves towards the leader node as a ripple (discrepancy across adjacent values). Consider the point of time at which two ripples collide. Let us look at an illustrative example. There is a node with value 4, and its next node has value 3. This situation can be represented with the ripple $4 \rightarrow 3$. Now, consider another ripple propagating towards this point. It is of the form $5 \rightarrow 4$. When it collides with the earlier ripple it will set the value of the node as per Line 11. The new ripple will be of the form $5 \rightarrow 3$. In the next step, the token with value 5 will get propagated. This means that the value 5 will be sent down the ring until either a previous ripple overwrites it or it reaches the leader node.

At this point (when the token reaches the leader node), there are two choices. The first choice is that $\texttt{pred}_\texttt{v}\texttt{al} = \texttt{u.val}$. This will trigger the logic on Line 5. The old token will be destroyed, and a new token in the form of a ripple will be created and sent down the ring. Next, let us consider the other case where there is an inequality. This means that a new token has already been generated by the leader node and propagated to downstream nodes along the ring. Thus in this case, nothing needs to be done. It is an example of a case where an outdated token is absorbed by the system and removed from circulation.

5.4 Synchronizers

In the previous sections, we have seen the design and analysis of multiple distributed graph algorithms. Many of these algorithms work in the synchronous model of distributed computing. As you may recall, one of the

fundamental properties of the synchronous model is that there is a notion of a *round*. All the messages sent in the previous round are assumed to reach their respective destinations by the start of the current round. Further, the model does not handle node or link failures nor does it consider unbounded message delays. On the other hand, we also saw the design and analysis of asynchronous algorithms and the difficulties that asynchrony brings. Asynchronous algorithms are not easy to design for all problems. They are hard to design and verify.

Sadly, the real-world does not operate in a synchronous manner. It appears therefore that synchronous algorithms do not have practical use and one has to actually design algorithms that work in the asynchronous model of computation. Clearly, the synchronous and asynchronous models are two ends of a spectrum of possibilities. One is very idealistic, and the other is very pessimistic. Sadly, designing algorithms in the asynchronous setting is more challenging as the algorithm has to contend with multiple challenges with respect to message delivery.

From an algorithm designer's point of view, it will be good to explore mechanisms that provide an illusion of synchronous execution in an otherwise asynchronous environment. The assumption still is that there are no node and link failures. This means that any message that is sent is eventually received; however, messages can get arbitrarily delayed and reordered in transit. All that the sender needs is an acknowledgement from the receiver, which cannot be forged.

One mechanism that is often useful in this context is *simulation*. The idea is to create a method to execute an algorithm designed for one model to run on a different model. Let us discuss the idea of *synchronizers* that enable such a simulation of a synchronous algorithm on an asynchronous platform. In the rest of this section, we will describe how to design the synchronizer mechanism.

5.4.1 Simulation

At a high level, the idea of the synchronizer simulation is to create a mechanism at every node that allows each process to safely move from one round to the next round in an asynchronous environment. This allows a process executing an asynchronous algorithm to simulate the actions of a node in the synchronous algorithm.

The key mechanism is a *pulse* that is generated by every process in the system. It is generated every time a round finishes and the next round begins. Let us define the notion of a safe pulse. Kindly note that unless

specified, the term "pulse" henceforth refers to a safe pulse.

Definition 5.4.1 Safe Pulse

When we are simulating a synchronous algorithm $\mathcal{A}$ on an asynchronous framework, each process can generate a *safe pulse* that has the following properties. A safe pulse generated by process P signals the completion of the previous round i at P. This means that at this point P has received all the messages that $\mathcal{A}$ would have received from round 1 till i. Furthermore, it has not acted upon any message that $\mathcal{A}$ would have received in rounds $\geq i$.

Lemma 8

Consider a synchronous algorithm $\mathcal{A}$ that runs in T_s rounds. If all processes generate a pulse indicating the completion of every round, then $\mathcal{A}$ can be simulated in an asynchronous model.

Proof: Suppose that process P terminates at the end of T_s rounds in algorithm $\mathcal{A}$. When P is executing in the asynchronous model, it generates pulses to signal the completion of a round. This means that P has received all the messages for the current round, used them to recompute its state and send all the messages that it needed to send. Let us subsequently prove the result using induction. In the first round, each process reads the program inputs and then computes the next state. The output for this round is the same for both the synchronous and asynchronous algorithms. Assume that the execution is identical till the i^{th} round. At the beginning of the $(i+1)^{th}$ round, all the messages that needed to be received have been received. Even if a message has been received that belongs to a later round, we assume it is stamped with the round number, and thus will not be processed. Hence, both the synchronous algorithm and its asynchronous simulation compute the same state and send the same state of messages. Therefore, by using mathematical induction, we observe that at the end of T_s rounds, the outcomes of both the algorithms are the same. A subtlety in the case of asynchronous systems is that these T_s rounds are "simulated". ■

It is clear that the existence of a mechanism to create a *safe pulse* provides the illusion of synchronous execution on an asynchronous platform. If a process knows when to generate a safe pulse, then we are done. However, it often does not have such oracle knowledge.

When can process P generate a pulse for round $i+1$? P needs to conclude

that all the messages intended for it have been received till "simulated" round i. Notice that P does not know if it has missed a message or not. In the absence of such information, P cannot formulate a strategy. It does not know whether it should wait or go ahead. If P waits, it may have to do so indefinitely. Such waiting may lead to deadlocks also. On the other hand, if P decides too soon that it needs to increment the round by issuing a pulse, then P may actually miss messages destined for it.

One way to resolve this dilemma is to add a few additional caveats. For example, we can assume that every process is aware of the messages it will receive in a given round. This assumption leads to our first simple synchronizer called the α-synchronizer.

5.4.2 Alpha-Synchronizer

The α-synchronizer is the simplest synchronizer. We can characterize it by understanding the conditions needed to move to the next round. It generates a safe pulse for itself and advances to the next round by making local decisions. The set of conditions that must be satisfied for a process `u` to move from round i to $i+1$ are as follows.

1. Process `u` has received exactly one message from each of its neighbors.
2. It has finished its local computation.
3. If messages from future rounds arrive ($> i$), they are buffered.

We need to understand that every process/node expects exactly one message from each of its neighbors in a round. Once it has received its expected set of messages from its neighbors, it is ready to start its local computation. After the local computation finishes, each process sends messages to its neighbors and transitions to the next round once the aforementioned conditions are satisfied.

Given that decisions are purely local, processes can progress at different rates. It is possible for some processes to race ahead. Given that two neighboring nodes wait for each other to progress to the next round, there is very little divergence in term of the round numbers across neighbors. However, the divergence can accumulate across the diameter of the network. This will increase the message buffering requirements throughout the network. Recall that when messages from advanced rounds are received, they need to be buffered. Further, it is possible for a slow process to delay a large part of the system. If it does not send its message corresponding to round i, then a neighboring process cannot transition to round $i+1$.

5.4.3 Beta-Synchronizer

We observed that round advancement based on purely local decisions can lead to excessive buffering and inefficiency. The β-synchronizer fixes some of these problems. It adds a layer of explicit coordination. Its round transition rules and mechanisms are as follows. Assume that process `u` is transitioning from round i to $i+1$.

1. Process `u` has received one message from each of its neighbors timestamped with the round number (i in this case).
2. It has finished its local computation.
3. After finishing the local computation, `u` sends a $\langle$`done`$\rangle$ message to its neighbors.
4. It transitions to the next round only after it receives a $\langle$`done`$\rangle$ message from all its neighbors.

The basic idea is that an additional layer of synchronization is added to the α-synchronizer. This idea prevents untimely messages from advanced rounds. This is because no neighbor can progress to the next round without receiving a $\langle$`done`$\rangle$ message from the current process `u`. Only after `u` is done with its local computation, it sends the $\langle$`done`$\rangle$ message to all its neighbors. Until the neighbors receive this message, they cannot transition to the next round. Without transitioning to the next round, they cannot send any message. Hence, messages from subsequent rounds cannot be received before they are expected.

It is possible to modify this algorithm by arranging processes as a tree. All the processes can send $\langle$`done`$\rangle$ messages towards the root of the tree. The algorithm is straightforward. Every node in the graph waits to receive $\langle$`done`$\rangle$ messages from its children. Then it propagates the $\langle$`done`$\rangle$ message to its parent. Once the root receives $\langle$`done`$\rangle$ messages from its children, it sends a pulse to all the processes. They transition to the next round. If message delays are constant, this process takes logarithmic time.

5.4.4 Gamma-Synchronizer

The γ-synchronizer is a hybrid synchronizer. The main idea is to divide the set of processes (nodes in the process graph) to clusters. Inside a cluster, the β-synchronizer is used such that they remain tightly synchronized and round numbers do not diverge. There is basically no drift (in terms of round

numbers) within a cluster. This keeps the message buffering requirements low and also ensures that processes are making progress at roughly the same rate. For inter-cluster communication, the α-synchronizer is used. Recall that it is a lightweight mechanism that can be used for small graphs. The assumption is that inter-cluster messages are infrequent. Clusters are created in a way to minimize communication between nodes across clusters.

The logic to advance to the subsequent round is as follows.

1. Intra-cluster β synchronization should be complete.
2. All the required α synchronization messages are received.

The overall synchronization overhead is much lower than conventional β synchronization. This is because messages are limited to clusters. There is no drift or round divergence within clusters. There can be some drift across clusters; however, it is expected to be limited. Hence, we can conclude that the algorithm is scalable and the drift is lower than α synchronization. The advantage of the γ-synchronizer depends on the quality of the clustering.

5.5 Summary and Further Reading

5.5.1 Summary

Summary 5.5.1

1. Synchronous algorithms assume that there is a global clock. Further, the computation can be divided into rounds, where each process is aware of when a round ends and the subsequent round starts. In each round, a process gathers all the messages it received in the previous round, changes its local state and sends new messages.
2. The BFS (breadth-first search) algorithm is a quintessential algorithm in the area of synchronous algorithms. In every round, nodes simply mark their unmarked neighbors.
3. Luby's randomized problem is used to solve the maximal independent set problem. It is a randomized algorithm. The key idea is that a node marks itself with a probability that is proportional to its degree. If no neighboring node with a higher degree has marked itself, then the current node becomes a part of the

independent set. The average number of rounds is $O(log(n))$.

4. Vertex coloring also uses a randomized strategy. Whenever two neighbors choose the same color, both of them try again. It can be shown that this simple strategy terminates with high probability in $O(log(n))$ rounds.

5. Asynchronous algorithms do not rely on a global clock. Each node acts as a simple state machine. It changes its state and sends messages depending upon its current state and the messages that it receives.

6. In the Chang-Roberts algorithm processes are organized as a ring. Every process broadcasts its willingness to be a leader along the ring. The request is propagated by another process only if it has a higher priority than the process's request. Ultimately, only the leader's message will complete the full circle and reach it back. It requires $O(n^2)$ messages.

7. This algorithm can be further optimized by consider windows of increasing sizes. A leader election is only done within a limited window of processes (arc along the ring). A process considers a large window only if it wins the election in a smaller window. The largest window is the entire ring of processes, which the leader only wins. This process requires $O(nlog(n))$ messages.

8. It is possible to further reduce the number of messages by organizing processes in a tree. This method requires only $O(n)$ messages. To create a tree of processes, most of the time a minimum spanning tree (MST) algorithm is used.

9. The Gallager, Humblet and Spira (GHS) algorithm is a popular asynchronous algorithm to compute the MST. Here, nodes collaboratively find the least-weight outgoing edge in a fragment. Every fragment has a level, which is indicative of its size. A fragment can only merge with another fragment if the level of the latter is greater than or equal to that of the former. This algorithm's message complexity is $O(2E + 5nlog(n))$.

10. The Chandy Lamport algorithm is an important algorithm in the space of reliable distributed computing. It helps collect a

"consistent" distributed snapshot. If a message is recorded by the receiver in the snapshot, then its corresponding send event should also have been recorded.

11. Termination detection algorithms are widely used in practical distributed systems. Termination detection is easy to do in trees – each node needs to send its termination status to the root. This algorithm can be extended to graphs by first computing an MST.

12. A self-stabilizing algorithm ensures that after a fault happens, the system enters a legal state in bounded time. Dijkstra's token-ring algorithm is a popular algorithm in this space.

13. The synchronous computing model is attractive because it is easy to design and verify algorithms in it. We can add many real-world constraints to this model.

    LOCAL In the LOCAL model, a node can only communicate with its immediate neighbors in a round. There is no limit on the message size. This makes it possible to gather the entire network at one node in $O(diam(G))$ rounds and perform computations on it such as finding a vertex coloring or a minimum spanning tree.

    CONGEST This model has a limit on the message size. It is limited to $O(log(n))$ bits.

    Congested Clique In this model all-to-all communication is the default given that the network is a clique. The routing problem is a quintessential problem in this model. Each node has n messages, each $O(log(n))$ bits long. The aim is to route these messages to their respective destinations in the minimum number of rounds subject to destination receiving exactly n messages. A constant round solution is possible, which allows many problems to be solved very quickly in this model – typically in $O(1)$ or $O(log(log(n)))$ rounds.

    Massively Parallel Computing (MPC) The MPC model is for big data systems, where n is the size of the input (not the number of nodes). The assumption is that every

node has limited local memory (limited to W words). In practical systems, $W = n^{\epsilon}$ $(0 < \epsilon < 1)$. Computations are slower in this model as compared to the Congested Clique model.

14. A synchronizer is a framework that allows a synchronous algorithm to run on an asynchronous platform. The basic mechanism is a *pulse* that is generated when a node has received all the messages to start the subsequent round.

    α-synchronizer It is a receiver-oriented protocol. The assumption is that every process sends exactly one message to every neighboring process in a given round. Every message is stamped with the round number. Once a process receives all the messages from its neighbors for the current round and finishes its local computation, it is time to transition to the next round.

    β-synchronizer This synchronizer reduces the divergence in terms of round numbers across the network. After a node has received all its messages, it sends a $\langle$`done`$\rangle$ message to its neighbors. Once a node has received all the messages for the current round, finished its local computation, and received $\langle$`done`$\rangle$ messages from all its neighbors, it can transition to the next round. This additional control layer ensures that the constituent processes progress at roughly the same rate (do not drift).

    γ-synchronizer This is a hybrid scheme. Nodes are divided into clusters. The β-synchronizer is used for nodes within clusters and the α-synchronizer is used for nodes across the clusters. It achieves a trade-off between drift and additional messages.

5.5.2 Further Reading

The area of designing distributed algorithms using various models of distributed computing has a long and rich history. Beyond the graph problems discussed in the chapter, other graph problems such as computing shortest paths find application in the context of routing in the Internet [Lougheed

and Rekhter, 1989]. A closely related problem is that of obtaining compact routing tables, which has a long history of distributed algorithms. Refer to the early works of Cowen [Cowen, 2001], Thorup and Zwick [Thorup and Zwick, 2001], Brady and Cowen [Brady and Cowen, 2006] and the references therein.

Unlike this book that looks at various aspects of distributed computing in addition to distributed algorithms, there are multiple excellent textbooks that focus exclusively on distributed algorithms. Some of the popular books on distributed algorithms are written by Lynch [Lynch, 1996], Peleg [Peleg, 2000] and Tel [Tel, 1999]. These books discuss algorithms from multiple domains beyond graph algorithms.

Distributed computing models such as LOCAL, CONGEST and Congested Clique have seen a tremendous amount of research interest. Along with these models, there are many algorithm design paradigms that are commonly used to design distributed algorithms for diverse problems. Some such paradigms include network decomposition [Awerbuch et al., 1989, Panconesi and Srinivasan, 1996, Awerbuch and Peleg, 1990, Rozhon and Ghaffari, 2020, Ghaffari et al., 2021], scattering and gathering [Barenboim et al., 2016], partitioning based on a graph's structural properties such as a H-decomposition [Barenboim and Elkin, 2010], etc. These techniques have found tremendous use in designing efficient algorithms for several graph problems such as maximal independent sets, coloring, dominating sets and maximal matching.

The MPC model was first introduced by Karloff [Karloff et al., 2010] and refined further by Balliu et al. [Balliu et al., 2019], Beame et al., [Beame et al., 2013] and Goodrich et al. [Goodrich et al., 2011]. In recent years, there has been a strong surge in designing algorithms in various MPC models. Techniques such as sketching, exponential gathering [Ghaffari, 2019], have been found useful in a variety of real-world algorithms. Some of the success of the MPC model can be attributed to its resemblance to the practical cloud computing model, which is very popular as of 2025.

A related all-to-all communication model for distributed algorithm design is the k-machine model. In the k-machine model, we consider k pairwise connected machines with computation taking place in synchronous rounds [Klauck et al., 2015b]. The k-machine model differs from the MPC model in two ways: the model constrains the bandwidth on the links yet does not constrain the ability of machines to store data. In particular, the k-machine model restricts communication bandwidth across any communication link to B bytes per round. However, each machine can store the entire input. The big question that the k-machine model attempts to answer is

whether algorithms can achieve $O(1/k^2)$ speedup. It is possible to achieve this for the minimum spanning tree problem [Pandurangan et al., 2016].

Questions

Question 5.1. Why is it important to have the notion of *level* in the GHS algorithm?

Question 5.2. In the GHS (minimum spanning tree) algorithm, we rely on the concept of a *combining edge*. Let us say that instead of relying on the concept of a core edge, we want to rely on the concept of a *combining vertex*, what changes do we need to make in the basic GHS algorithm?

Question 5.3. Does the Chandy-Lamport algorithm require FIFO channels? Explain your answer.

Question 5.4. Assume a network with a star-like topology. There is a central node, that sends messages to child nodes. There are no interconnections between child nodes. Provide an algorithm to take a consistent snapshot of the entire system with the central node as the initiator. Assume that the channels do not obey the FIFO (first in, first out) property.

Question 5.5. We wish to design a checkpoint and replay system for a DHT. This system should be able to maintain snapshots for the last K versions. Describe one such system. Try to propose as many optimizations as you can.

Question** 5.6. Propose a synchronous distributed algorithm for finding the shortest path between two nodes in a weighted graph.

Question 5.7. Consider Safra's algorithm. Let us add an additional rule. We have FIFO behavior across arbitrary pairs of nodes. All messages sent from `u` to `v` follow a FIFO order regardless of the path they take from `u` to `v`. They could either travel directly (single hop) or via multiple hops. Does this simplify the Safra's algorithm? Can we prove that it will never be the case that a message receive is recorded but the corresponding send event is not recorded?

Chapter 6

Consensus and Agreement

> *"If everyone is moving forward together, then success takes care of itself."*
>
> Henry Ford

In the world of distributed systems, if there is one fundamental question that is the cornerstone of almost all the major algorithms and designs, it is the *consensus problem*. This problem is known by many names. It is often referred to as the *agreement problem* as well. The basic idea is very simple. There are n processes (independent entities). Each one of them proposes a value. Let us say that process i proposes value v_i. In some generic versions of the problem, it is not necessary that a process propose a value. It can instead decide to remain quiescent. After the protocol starts, the processes start communicating with each other via either sending messages or communicating by other channels such as shared memory. At the end of the protocol, we want all the processes to *agree* on a single value, which one of the processes has proposed. For example, they cannot all agree to the value zero, if it is not been proposed by any process. Hence, the caveat is that the value that is agreed to by all the processes is a value that at least one process has proposed.

Let us now further complicate this problem. Often in a distributed system, we can have faulty processes. In fact, if we do not have faulty processes, most algorithms in this area become very simple and trivial. Hence, almost all realistic distributed systems take faulty processes into account. Faulty processes can be of various types. We can have simple fail-stop processes that become unresponsive when they fail. Furthermore, we could have pro-

cesses that are intermittently faulty. This means that they are active for some time, then they become inactive, then they can become active and so on. This appears to be a very innocuous kind of fault model because the processes cannot cause a lot of damage. A process can become unresponsive and not participate in the protocol, but it looks like it is not capable of damaging the functioning of the full system significantly. Let us compare this with another class of faults known as *Byzantine faults*. In this case, a process can genuinely act maliciously. It can deliberately lie and provide different inputs to different processes. Furthermore, a set of malicious processes can collude (act together) and deliberately mislead the rest of the processes. Such Byzantine faults are in many ways more powerful than regular fail-stop faults; they make the process of agreement quite difficult.

Definition 6.0.1 Consensus

The consensus problem is defined as follows. We have a set of processes, where each process may propose a value. Finally, one of the values is accepted by all the non-faulty processes. This value is known as the *consensus value*, which must have been proposed by at least one of the processes.

In the presence of faults, there is a need to slightly modify our definition. We need to say that when the protocol terminates, all the non-faulty processes agree on the same value. We do not really care what the faulty processes agree on because given the fact that they are faulty, their conclusions do not matter. Insofar as we are concerned, we don't take their actions very seriously after the protocol terminates.

A second question that arises is whether the value that is finally agreed upon by all the non-faulty processes needs to be proposed by a non-faulty process. Can it be proposed by a faulty process, which is possibly guilty of duplicity and has communicated different values to different processes? The answer lies in the fact that it is not possible for a process to ascertain for sure whether another process is faulty or not. It needs to take whatever message that it gets at face value unless it runs a different protocol to determine whether a given process is faulty. Hence, it is a wise idea to not impose any restriction on the value that is agreed upon as long as it has been proposed by some process and the same has been communicated to at least one non-faulty process. Definition 6.0.1 captures these augmentations to the basic definition.

Let us now consider an example and see why this problem is so important. Often it is the case that while we are booking an airline ticket, we

require multiple entities to be in sync. These are the credit card company, the mobile app, the backend server (the travel company) and the airline. Sometimes it happens that money is deducted from the credit card, however the ticket has not been booked. This can be a very irritating and annoying situation. Sadly, this has happened to one of your authors on many occasions. The reason that this happened in the first place is because there was no agreement, or there was no consensus, between all of these interacting parties. Hence, it was possible for the money to be deducted from the credit card account, without any ticket being actually booked. Of course in this case, the backend server of the travel site did realize that something was amiss, and the money was refunded in due course of time. But sadly, your author was not aware of whether a ticket was actually booked or not. This caused a lot of delay and confusion. Finally, when the ticket was booked in the second attempt, the prices had increased !!! Had a consensus protocol been operational, this would not have happened.

In this chapter, we will start with some classic impossibility results, which basically say that in many situations of practical interest, it is not possible to design an algorithm that guarantees reaching a consensus all the time. In fact, we shall see a very surprising result in Section 6.1 that says that if we have one faulty process in an asynchronous system (without a common clock), it is impossible to guarantee consensus in a distributed setting. Even though, we cannot achieve consensus reliably all the time, there are a lot of algorithms that help us achieve consensus most of the time. We will discuss many such algorithms in this chapter.

Organization of this Chapter

Let us read the figure in a clockwise direction. We shall start with discussing the FLP result, which proves that even if there is one faulty process in an asynchronous setting, it is impossible to create an algorithm that guarantees consensus all the time. This is a fundamental result that sets the limits of what needs to be done. Next, we shall discuss synchronous algorithms and discuss consensus/ agreement problems with Byzantine faults. We shall observe that to solve the binary consensus problem we need to send an exponential number of messages. Notwithstanding these limitations, it is possible to design practical consensus algorithms that solve real-world problems quite well.

Subsequently, we will discuss three algorithms that are practical and implemented in the real world. We will start with a practical version of Byzantine tolerance (the PBFT algorithm), and then discuss two algorithms

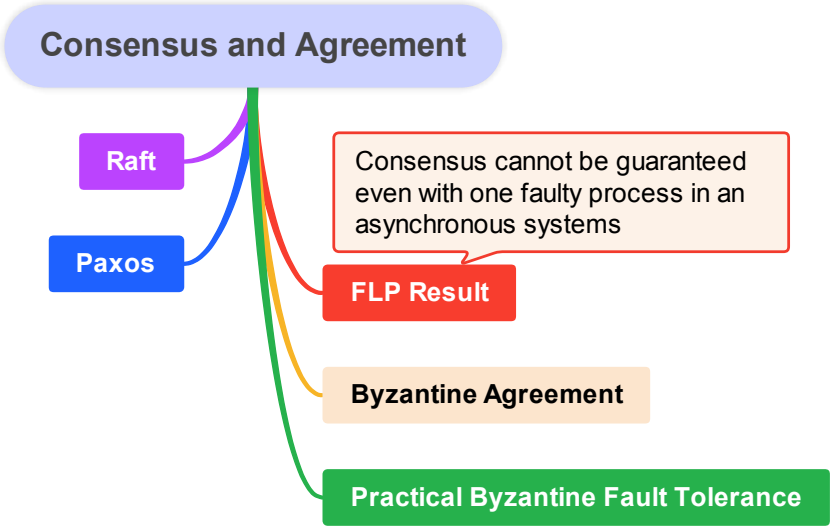


Figure 6.1: Organization of this chapter

in asynchronous settings: the classical Paxos algorithm and a contemporary algorithm, Raft.

6.1 FLP Result

Let us prove one of the most important results in distributed systems that deals with the impossibility of consensus in an asynchronous setting with one faulty process. It was proposed by Fischer, Lynch and Paterson in 1985 [Fischer et al., 1985]. Based on the last names of the authors, this result is known as the FLP result. In this case, the faulty process can either be genuinely faulty, malicious or just slow. We sadly do not have any way of finding out why a process is unresponsive. We further simplify the problem by considering binary consensus: 0 or 1. Our line of argument in this section is quite similar to the arguments made in the original FLP paper, albeit with a different method of explanation and modified terminology.

6.1.1 Fundamentals

Processes and Registers

Every process is modeled as a finite automaton. In a single step, it can either receive a message from another process, change its state upon receiving a message, or send a message to another process. Processes have broadcast capability. This means that if a message is sent to a non-faulty process, then it is guaranteed that all non-faulty processes will eventually receive it. However, messages can be delayed arbitrarily and also be delivered out of order. The proof relies on the fact that there is a window of time, where it is not possible to exactly ascertain if a process is slow, unresponsive or dead.

Definition 6.1.1 Register

A *register* is a shared variable in a distributed system that can be concurrently accessed by multiple processes. It is possible to issue read and write requests to a shared register. It is an abstraction of a unit of information in a distributed system. It may be stored in a single location or can also be stored across multiple locations.

Each process p has an input register x_p and an output register y_p associated with it. A *register* conceptually represents a shared variable – a single bit in this case. Let the registers store one among the three values $0, 1, b$. The input register contains the value that the process is proposing. Here, b is a special value that means uninitialized. All the output registers start with the value b. This means that we have not committed to an output when the algorithm starts running. Once the output register is set to either 0 or 1, it means that a decision has been made. The consensus value is either 0 or 1. The output register is "write once". This means that it can be written to only once, and subsequently cannot be overwritten. This means that once a decision regarding the consensus value has been made, it cannot be reversed. Let the **state of the entire system** be the union of the contents of the internal registers, output registers and internal states of all processes.

Assume that the processes send messages to a network with a message buffering system. It contains all the messages that are in flight. Let the term (p, m) be a tuple representing the destination process p and message m, where message m is meant to be delivered to process p. Now, given that the network can have an arbitrary delay, there is no guarantee that the message will be delivered immediately or within a bounded amount of time.

There are two basic primitives here: SEND (p, m), and RECEIVE (p). The function SEND (p, m) is invoked to deliver message m to process p either now or at a later point of time. The process-message tuple is internally placed in a message buffer, and delivered at a later point in time. The function RECEIVE (p) deletes a message (p, m) from the message buffers in the buffered network, and then delivers it to p. If the message is not ready to be delivered, and the destination process calls receive(p), then a null value (ϕ) is returned.

Events and Schedules

The configuration of the entire system is a combination of the internal states of all the processes and the contents of the message buffers. In the initial configuration C_{init}, the states of all the messages buffers is empty and the state of all the processes is the initial state. Let us assume that a process takes one step at a time: receive a message (could also be null), do some internal computation and send messages to the rest of the processes. Let us refer to an *event* as a tuple of a process and a message (p, m). *Applying* an event (p, m) to a process p means sending message m to p and then allowing it to change its internal state and send other messages in response. If an event is applied to a configuration C, $e(C)$ denotes the resulting configuration. Note that an event (p, ϕ) can always be applied to a process p, and thus some forward progress is possible. Assume $D = e(C)$, we can represent the same as $C \xrightarrow{e} D$.

Let us define a *schedule* of events as a sequence of events that can be applied to a configuration, C. Applying a schedule is equivalent to applying the events one after the other in sequence. We will use the term σ to represent a schedule. $\sigma(C)$ represents a schedule applied to the starting configuration C. Let all the configurations of the form $\sigma(C)$ be called *accessible configurations* – they are reachable from the starting configuration C.

We can quickly deduce a few common sense results. Assume that there are two schedules σ_1 and σ_2 where the processes that participate in them are disjoint. We claim that they are commutative (the order in which they are applied does not matter). Consider a starting configuration C. We claim that $\sigma_1(\sigma_2(C)) = \sigma_2(\sigma_1(C))$. This is because the schedules are disjoint and as a result their order does not matter because the processes in the two different schedules do not interact. Figure 6.2 shows this graphically.

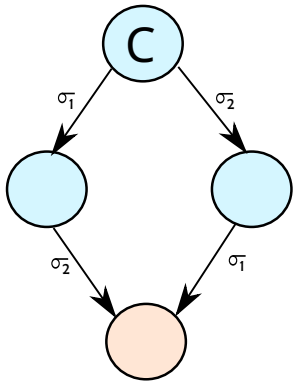


Figure 6.2: The commutativity of schedules (if they do not have overlapping processes)

Basic Definitions

A few more definitions are due. A configuration C has a decision value v if some process has reached a decision state. This means that its output register y_p is equal to v. A consensus protocol is said to be *partially correct* if it satisfies the following conditions.

- Consider all the accessible configurations. None of them should have more than one decision value.
- For each value v in the set $\{0, 1\}$, some accessible configuration has value v.

A *partially correct* protocol means that the protocol can lead to at least two types of configurations. One set of configurations conclude 0 and the other type of configurations conclude 1. Of course, it is not necessary that a decision is made all the time. There is a possibility that we get into an infinite loop and no decision is eventually made. The important aspect is that we never reach a configuration that is incorrect (two different values are decided) and it is always possible that we can reach configurations that decide either value. This means that the decision is not *fixed* before the protocol starts.

A process p is said to be *non-faulty* if it can take infinitely many steps. This means that it may never stop. Note that a process can always react to the ϕ message, which is a vacuous transition. An infinite sequence of ϕ messages should lead to an infinite number of transitions and internal steps.

It is otherwise faulty, which means that at some point it abruptly stops without deciding. A run (or execution) is said to be *admissible* if at most one process is faulty. The rest of all the processes are non-faulty. Since the message buffers never drop messages, the non-faulty processes eventually receive every message.

A run is a *deciding run* if a process reaches a decision state (output register is 0 or 1). Any partially correct consensus protocol is said to be *totally correct* if every admissible run is a deciding run. This means that a decision is made all the time. Recall that in the theory of complexity, we typically deal with such questions.

Definition 6.1.2 Basic Definitions

- Decision value of a configuration – If a process in a configuration has decided value v, then v is the decision value of the configuration.
- A consensus protocol is *partially correct* if no accessible configuration can decide two values, there is at least one accessible configuration that decides 0, and another that decides 1.
- Faulty and non-faulty processes – If it is possible for a process to take an infinite number of internal steps, then it is non-faulty. Otherwise, it is faulty.
- Admissible run – An execution (run) is *admissible* if at most one process is faulty.
- Deciding run – A *deciding* run always reaches a decision state.
- A protocol is *totally correct* if every admissible run is a deciding run.

6.1.2 Univalent and Bivalent Configurations

Now, the key question that we need to answer is as follows. Do totally correct protocols exist? Or in other words, do all partially correct protocols always have deciding runs? Note that we are only considering admissible runs here – at most one process is faulty. The key FLP result says that it is not possible to design a totally correct protocol if there is one faulty process. Let us start with a sequence of lemmas to set the stage.

A configuration is *univalent* if it only leads to configurations with a deci-

sion that is either only 0 or only 1. This means that for this configuration, a decision has already been made. Even if it hasn't been made, any reachable configuration with a decision value always decides the same value. On the other hand, a configuration that can reach configurations that decide either of the two values are called *bivalent*. This means that no decision has been committed to. Things are open-ended at this stage.

Lemma 9

Every totally correct protocol has a bivalent initial configuration.

Proof: Let us prove this by contradiction. If this is not the case, then all initial configurations are either 0-valent (decide only 0) or 1-valent (decide only 1). Two initial configurations can be termed to be *adjacent* if the only difference is in the initial values of some process p. For instance, the input register in one case contains 1 and in the other case contains 0.

Consider the space of all *initial* configurations. We create a graph to represent this space, where each node is an initial configuration. We draw an edge between configurations C and C', if the only difference between them is that a single process had different initial values. In other words, they are adjacent. Note that in this graph, there is a path between all pairs of initial configurations. Given one initial configuration, we can always flip the initial input of one process and make the resultant configuration closer to the final configuration. Ultimately, after a sequence of such flips, we can arrive at any intended configuration.

Recall that we are assuming that all initial states are univalent. This means that if we start with some 0-valent configuration C_0 and keep on flipping the values of initial processes, ultimately a time will come up when due to a flip, the configuration changes from C_0 to C_1 (1-valent). Let this process be p.

Let us consider these two configurations C_0 and C_1. Consider an admissible and deciding run σ from C_0 in which process p does not take any steps. Note that such a run always exists because by assumption every admissible run is a deciding run (totally correct protocol). $\sigma(C_0)$ thus leads to a 0-valent state. Now, consider $\sigma(C_1)$. Given that process p does not take any steps, there is no way in which its initial value can affect the final outcome. Hence, $\sigma(C_0) = \sigma(C_1)$. The LHS is a 0-valent configuration and the RHS is a 1-valent configuration. This equality cannot hold. Hence, we have a contradiction and the lemma stands proven. ■

6.1.3 Impossibility of Consensus

Given that there is an initial bivalent configuration (by Lemma 9), the question is whether we can reach another bivalent configuration from it. If we can prove that we indeed can, then we are essentially laying the foundation for a proof where we can endlessly continue to traverse through bivalent configurations. If this is indeed true, then we can never arrive at a decision.

Lemma 10

Consider a totally correct protocol. There is a maximum of one faulty process. Assume that C is a bivalent configuration. Consider all configurations of the form $\sigma(C)$ and the set of all configurations of the form $e(\sigma(C))$, where $e \notin \sigma$. There is a σ such that the configuration $e(\sigma(C))$ is bivalent.

Proof: Let the set $\mathcal{C}$ be the set of all configurations that are reachable from C without applying event e. Let the set $\mathcal{D} = \{e(C) | C \in \mathcal{C}\}$. We need to prove that $\mathcal{D}$ has a bivalent configuration. Let us assume that this is not true. This means that all the configurations in $\mathcal{D}$ are univalent.

Now, given that the configuration C is bivalent, there need to be two univalent configurations C_0 (decides 0) and C_1 (decides 1) that are reachable from C. Note that such configurations will always exist given that C is bivalent. Let us refer to the configurations as C_i, where $i \in \{0, 1\}$. If $C_i \in \mathcal{C}$, let us define $D_i = e(C_i)$. It is also possible that the event e was applied while reaching C_i from C, then there exists a configuration C_x such that the following sequence of events holds: $C \rightarrow \ldots C_x \xrightarrow{e} D_i \ldots \rightarrow C_i$. Recall that we are assuming that all D_i are univalent.

We have shown two cases. In the first case, $C_i \xrightarrow{e} D_i$ and in the second case $D_i \rightarrow \ldots \rightarrow C_i$. We thus see that the set $\mathcal{D}$ has both 0-valent and 1-valent configurations.

Now, let us look at two neighboring configurations in $\mathcal{C}$. Two configurations are said to be *neighboring* if one results from the other in a single step. We claim that we can find two such configurations E_0 and E_1 (both elements of $\mathcal{C}$) that satisfy the following properties: $E_0 \xrightarrow{e'} E_1$, $E_0 \xrightarrow{e} (D_0)$ and $E_1 \xrightarrow{e} D_1$. D_0 and D_1 are univalent (decide 0 and 1, respectively) by assumption.

Let us now prove that such a pair of configurations, E_0 and E_1 exist in $\mathcal{C}$. Consider all pairs of univalent configurations in $\mathcal{D}$ that decide different values. For each pair of configurations in $\mathcal{D}$, consider the corresponding pair in $\mathcal{C}$. It is not possible that there is no pair where one configuration (in $\mathcal{C}$)

is not reachable from the other configuration (also in $\mathcal{C}$). If that is the case, then there is a strict partition among the configurations in $\mathcal{C}$ that lead to 0 and 1-valent configurations in $\mathcal{D}$, respectively. Hence, every configuration is also univalent in $\mathcal{C}$. However, given that the initial configuration $C \in \mathcal{C}$ and C is bivalent, we have a contradiction. Hence, this cannot be true. Hence, without loss of generality, such a pair of configurations $(E_0, E_1 \in \mathcal{C})$ will always exist.

We thus have the following set of relationships (without loss of generality).

$$\begin{aligned} E_0 &\xrightarrow{e'} E_1 \\ E_0 &\xrightarrow{e} D_0 \\ E_1 &\xrightarrow{e} D_1 \end{aligned} \tag{6.1}$$

Let us assume that the event e' happens for process p', and e happens for process p. Consider the following cases.

$p \neq p'$ In this case the events e and e' are commutative. We thus have $e'(e(E_0)) = e(e'(E_0))$. This implies that $e'(D_0) = e(E_1) = D_1$. One univalent configuration D_0 cannot lead to another univalent configuration D_1 that decides the reverse. This situation is shown in Figure 6.3

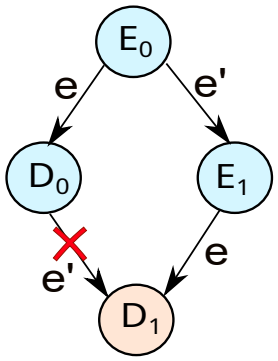


Figure 6.3: The illustration of Case 1 ($p \neq p'$)

$p = p'$ Consider a run with starting configuration E_0 that is also deciding. In this run, process p remains quiescent (does not take any steps). Given that the run is deciding, it ends up in a univalent configuration E_x. Let this run be σ. This situation is shown in Figure 6.4.

In this case, the events e and e' commutate with σ (disjoint set of processes). We thus have $e(\sigma(E_0)) = e(E_x) = F_0$. $e(e'(\sigma(E_0))) = \sigma(e(e'(E_0))) = \sigma(e(E_1)) = \sigma(D_1) = F_1$. F_0 and F_1 are univalent configurations that decide 0 and 1, respectively. If we look at Figure 6.4, we observe that there is a path from the configuration E_x to F_0 and F_1. Now, the contradiction here is that there is a path from the univalent configuration E_x to two other univalent configurations, F_0 and F_1, which decide two different values. There is a contradiction in this case also.

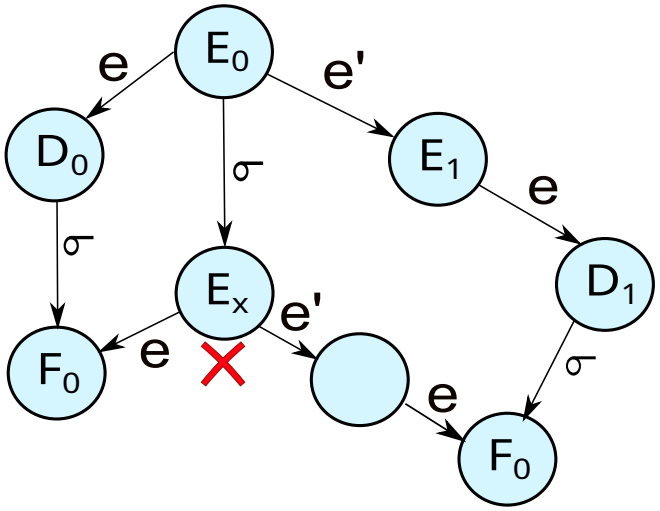


Figure 6.4: The illustration of the second case ($p = p'$)

This means that $\mathcal{D}$ does not have purely univalent configurations. It definitely has one bivalent configuration. ■

Theorem 6.1.1

It is not possible to design a totally correct protocol if there is one faulty process.

Proof: By Lemma 9, we can conclude that there is a bivalent initial configuration C_b. This means that at this point, we have not decided whether the final outcome is 0 or 1. This is a fair assumption in any protocol that always terminates with a result as we have seen. The result is not fixed a priori, which should be the case with consensus protocols. If the outcome

has already been decided a priori, then there is no point of running the consensus protocol.

Now by Lemma 10, we know that there is a schedule (in which e is applied at the end) such that we end up with a bivalent configuration. Using mathematical induction, we can argue that we will continue to have bivalent configurations regardless of the number of messages and events. This means that ultimately we will not be able to make a decision, because we will always be moving between bivalent configurations. The algorithm will never terminate because we will never reach a univalent configuration. ■

The conclusion that we can derive from Theorem 6.1.1 is that if a process is slow and does not take any steps, we may never be able to reach a deciding configuration. This means that the protocol may run forever without terminating. As a result, in an environment with at least one faulty process, it is impossible to design a consensus protocol that terminates all the time. However, in practice, a lot of consensus protocols can be designed that are simple, elegant and terminate *most of the time*. Just because, in all possible scenarios, it is not possible to guarantee termination does not mean that consensus protocols should not be designed and optimized.

6.1.4 Consensus with Initially-Dead Processes

Assume there are n processes in the system. Let us slightly relax the assumptions here. Assume that no process suffers from any failure after the execution has started. This ensures that a lot of the problems that we saw in the case of a single faulty process will not happen. There however could be *dead processes* that are unresponsive throughout the execution. Moreover, assume that a majority of the processes are not faulty. Note that this information including the information regarding dead processes is not known to all processes. A process only knows about itself. It does not know whether another process is alive or not unless that process sends it a message.

The Algorithm

We will use the notion of a *quorum* here. A quorum in this case is a simple majority, which for n processes is $\lceil (n+1)/2 \rceil$ $(= Q)$ processes. The ceiling arises because the number of processes can both be even and odd. For example, if the number of processes is 7, $Q = 4$. This is a majority. If $n = 8$, $Q = 5$, again a majority.

We can design a 2-stage algorithm. In the first stage, every process broadcasts its ID and proposed value to the rest of the processes. Note that

we can have slow links, which may delay a few messages. This means that all the messages need not reach their destinations at the same time or even within the same time window. Each process waits till it gets $Q-1$ messages from other processes. This is a fair assumption even in an asynchronous setting. Given that processes do not die in the middle, this will ultimately happen. Once a process receives $Q-1$ first-stage messages, it enters the second stage.

In the second stage, it broadcasts a message to the rest of the processes with its process ID, proposed value and the IDs of the $Q-1$ processes it heard from in the first stage.

We create a node for each process and add an edge from process i to j if i sends a message to j in the first stage. Let this graph be G (maintained per process). The moment a process receives a message it adds nodes and edges to this graph. For example, if some process u says that it received a message from process v, then a $u \rightarrow v$ edge is added to the graph. Subsequently, it figures out the set of its ancestors in this graph. Note that its immediate ancestors had sent it a message in the first stage. The second-degree ancestors had sent messages to its first-degree ancestors (immediate neighbors) in the first stage, so on and so forth. Given that it receives many more messages in this stage even from unrelated processes, a lot of additional ancestors can be discovered. The second stage ends when messages from $Q-1$ immediate ancestors of every node in G has been received. Finally, note that the in-degree of every node in G needs to be $Q-1$.

Next, we convert G to G^+, which represents its transitive closure. G have the same set of nodes. If there is a path from i to j, then there is an edge from i to j in G^+. Note that every cycle in G is a clique in G^+. Given that all pairs of nodes in a cycle are reachable from each other in G, there is an edge between each pair in G^+. Let us find the maximum-sized clique in G^+. We shall refer to it as the *max-clique ($\mathcal{C}$)*. Let the set $\mathcal{S}$ comprise all the values proposed by all the processes in max-clique.

We shall prove that this proposed set of values, $\mathcal{S}$ (by all the processes in $\mathcal{C}$), is agreed upon by all non-faulty processes. They can use any function to find the consensus value from the values contained in $\mathcal{S}$. For example, they can choose the minimum value, maximum value, mean value or the most frequent value. The choice of this function does not matter. The fact that all the processes apply the same function on the same set and consequently choose the same consensus value is the only point to be noted.

Proof of Correctness

Lemma 11

There is at least one node that is the ancestor of itself in G and G^+. In other words, G has a cycle, and G^+ has a clique.

Proof: Assume that this is not the case.

Consider each node v. It has $Q-1$ predecessors that point to it based on the messages sent in the first stage. Let u be one of its predecessors. If we count node u and its $Q-1$ predecessors, then we have Q nodes. Let the set S_u contain these nodes. The number of nodes that are not in S_u is $n-Q$. Based on the definition of Q, we know that $n-Q < Q$. This can be verified by considering even and odd values of n, where n is the number of total nodes. We can thus deduce $n-Q \leq Q-1$.

By our assumption, there can be no path from v to u. This means that v is a part of $\overline{S_u}$ (complement of S_u). Consider all its successors. None of them can point to any node in S_u. Otherwise, there will be a path from v to u, which is not allowed. v thus has at the most $Q-2$ successors in $\overline{S}$.

We can reason about all the nodes similarly. The maximum out-degree is limited to $Q-2$.

Let us now do some accounting across all the nodes. Every node has exactly $Q-1$ predecessors in G. Hence, the sum of in-degrees is $\sum_i (Q-1)$ across all non-faulty nodes. This cannot be equal to the sum of out-degrees because each one of them is limited to $Q-2$. However, both should be equal. There is thus a contradiction here. Hence, this situation is not possible.

This further means that for some node the out-degree is $\geq Q-1$. This would thus lead to a cycle. ■

Corollary 6.1.1

There is a node with an out-degree of at least $Q-1$. It is part of a cycle in G and the corresponding clique in G^+.

Theorem 6.1.2

Let $\mathcal{C}$ be the clique with a node with out-degree at least $Q-1$. Let it have an incoming edge from node u, where $u \notin \mathcal{C}$. $\mathcal{C} \cup u$ is a clique.

Proof: Let us create a set T_u with node u and its $Q-1$ predecessors. It has Q elements. Let $v \in \mathcal{C}$ have an out-degree that is at least $Q-1$. Let S_v

contain v and all its successors. $|S_v| \geq Q$. We thus have $S_v \cap T_u \neq \phi$. There must be a common node w.

There is a path from v to w because both are in S_v, and there is a path from w to u because both are in T_u. We thus have a cycle of the form $u \rightarrow v \rightarrow w \rightarrow u$. u is reachable from the clique and also has an edge to reach at least one node in the clique. A node y is *reachable* from node x, if there is a path from x to y. Hence, $\mathcal{C} \cup u$ is a clique. ■

Using the results of Theorem 6.1.2, we can keep expanding the size of the clique till it does not have any incoming edges. The claim is that this is the max-clique. Given that it does not have any incoming edges, we clearly cannot add any more nodes to it. This is because if there is some node that is not a part of it but should be, then there must be an incoming edge into some node of the clique. Since this is not the case, we can easily conclude that the clique cannot be grown any further. Moreover, given that the clique contains all the predecessors of every node, its size is at least Q. Hence, we cannot have two different max-cliques.

The next question that we need to answer is whether the max-clique recorded by one node is the same as the max-clique recorded by another node. We need to note that every node maintains private data structures such as G, G^+ and the max-clique. They are not shared across nodes. The following theorem shows that two different processes cannot have different max-cliques. Both need to contain exactly the same list of processes.

Theorem 6.1.3

The max-cliques recorded by two different nodes u and v cannot be different.

Proof: Assume that they are different. This means there is some node x that (w.l.o.g) is present in the max-clique of u but is not there in the max-clique of v.

Let $\mathcal{C}_u$ be the max-clique of u, and $\mathcal{C}_v$ be the max-clique of v. This means that $x \in \mathcal{C}_u$ and $x \notin \mathcal{C}_v$. Given that any clique contains a node and at least its $Q-1$ predecessors, we have the following relationships: $|\mathcal{C}_u| \geq Q$ and $|\mathcal{C}_v| \geq Q$. This means that they will have a node in common.

Let $w \in \mathcal{C}_u \cap \mathcal{C}_v$. It is a part of both the cliques. x is an ancestor of w. This means that after the information for w is received at a node, it will wait till it gets messages from all its ancestors (the transitive closure = predecessors, predecessors' predecessors and so on). This means that if information about w is received, the node will wait till it hears about x. This will be done by both the nodes u and v. As a result, it is not possible

for v not to know about x.

This means that v has x in its graph; however, by our assumption x is not present in its max-clique. There is a path from x to w because x is an ancestor of w. Given that $w \in \mathcal{C}_v$, there is an incoming edge into the clique $\mathcal{C}_v$. By Theorem 6.1.2, a maximal clique cannot have an incoming edge. Hence, $x \in \mathcal{C}_v$. We thus have a contradiction. No such node can exist.

This means that the max-cliques cannot be different across the non-faulty nodes. ■

Theorem 6.1.3 proves that all the non-faulty nodes record the same max-clique. This means that they record exactly the same set of processes and proposed values. The consensus value is just a function of this set of proposed values. Given that all the nodes use the same set as input and use the same function, the consensus value is guaranteed to be the same.

Observation 6.1.1

As per the FLP result, even a single faulty process can render any algorithm useless and make consensus impossible to achieve with certainty. However, this theorem indicates that if for a reasonably long duration there is no process failure, then the consensus algorithm can complete and a decision can be made. It is a two-stage algorithm, where the first stage disseminates information, and the second stage aggregates sufficient information to make a decision. We shall see examples of many more algorithms that require more than two stages.

6.2 Byzantine Agreement

Let us now consider solving the consensus problem in synchronous settings with Byzantine faults (see Section 1.4). In this case, we can make an assumption that a message or a reply will be received in a bounded amount of time. If a process is down, this can be detected because it will be found that it is not sending messages when it is supposed to. Given that all the processes share the same clock, this is very easy to ensure. Most synchronous algorithms can be divided into rounds. In a *round*, a set of messages are sent, and they are received. Based on the messages that a process receives, it updates its state accordingly and prepares messages to be sent in the next round. The absence of a message can thus easily be detected in a round. Moreover, we also assume that a message cannot be tampered with and cannot be deleted from the system. This means that the receiver is always

sure who the sender of a message is and the fact that the message is not tampered with.

6.2.1 Overview of the Problem

Let us consider the diagram shown in Figure 6.5. There is one commander and then there are multiple generals or *lieutenants*.

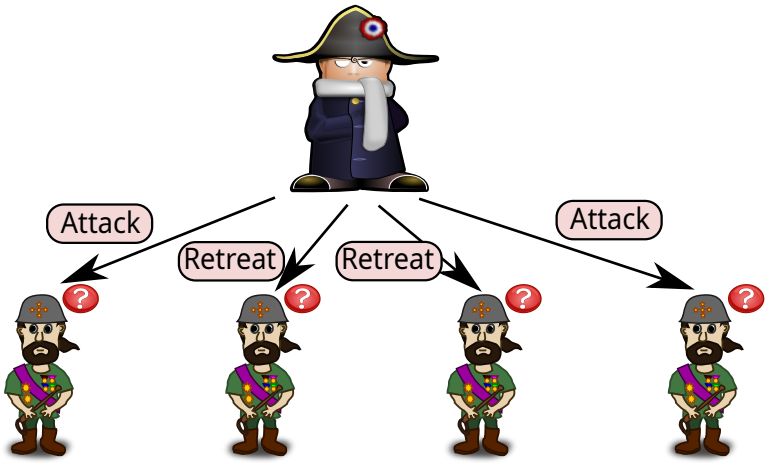


Figure 6.5: A Byzantine commander and generals

Consider a battle. The commander issues orders to his lieutenant generals. However, both the commander and his generals suffer from Byzantine faults. They can either be loyal or disloyal. Figure 6.5 shows a situation in which the commander is disloyal. He very smartly sends $\langle$`Attack`$\rangle$ (A) orders to two generals and $\langle$`Retreat`$\rangle$ (R) orders to the rest of the two generals. In this case, utter confusion prevails. The lieutenant generals do not know what to do, and they are confused. A lieutenant general does not know what orders have been sent to the rest of the lieutenant generals. As a result, they are not sure how to act. There also could be another scenario where the commander is loyal and dutifully sends the same order to all lieutenant generals. However, some lieutenant generals themselves could be disloyal. They may decide to disobey the order or incorrectly report the order that they have received to other lieutenant generals.

Now, the question is how does a loyal lieutenant general distinguish between these two cases? Both appear identical to him. He does not know whom to trust. One option is to of course ask the other generals about the

orders that they have received. However, what is the guarantee that they are speaking the truth? There could be divergent answers. We thus have a complicated situation on our hands where a general does not know whom to trust.

If we want to create an effective protocol in such a scenario, we need to first define our objectives. Let us thus define two conditions.

Condition C1: All loyal lieutenant generals obey the same order.

Condition C2: If the commander is loyal, then every loyal general obeys the commander's order.

Definition 6.2.1 Byzantine Consensus

In any Byzantine consensus problem, there is a commander and a set of generals – any subset of them can suffer from Byzantine faults. The commander is a process that sends a value to the rest of the processes for agreement. The rest of the processes are the lieutenant generals who don't know if the commander process is fault-free or not. The solution to the Byzantine consensus problem needs to satisfy two properties.

- C1 All the loyal generals follow the same order. This means that all fault-free processes arrive at the same decision.
- C2 If the commander is loyal, then all loyal generals follow the commander's order. This basically means that if the commander process is fault-free, then every fault-free process accepts the value sent by it.

Note that this is a binary consensus problem, where we want all the loyal generals (including the commander) to agree on one of two values: $\langle \texttt{Attack} \rangle$ (A) or $\langle \texttt{Retreat} \rangle$ (R). Of course, this is not a pure consensus problem, where all the generals are equally privileged. The commander has a special place. If he is loyal, then his order is the consensus value.

6.2.2 Impossibility Results

Consider a situation with 3 generals: one commander and two lieutenant generals.

Theorem 6.2.1

When we have one commander, two lieutenant generals and one traitor, the Byzantine consensus problem is not solvable.

Proof:

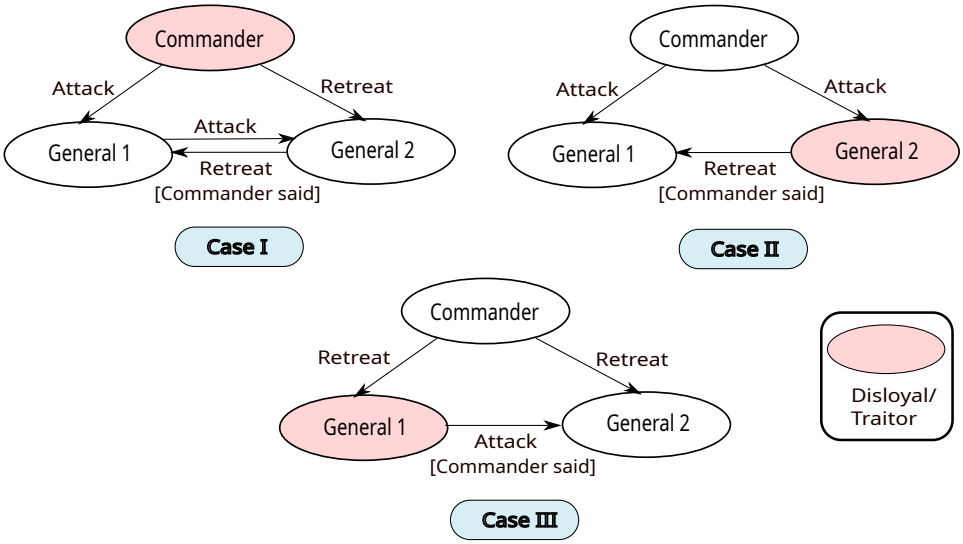


Figure 6.6: Three cases: disloyal commander (Case I), loyal commander sending ⟨`Attack`⟩ (Case II) and loyal commander sending ⟨`Retreat`⟩ (Case III)

We show three cases in Figure 6.6. Consider cases I and II, and look at it from the point of view of General 1. He asks General 2 about the order that he has received from the commander. In both the situations, General 1 receives ⟨`Attack`⟩ and ⟨`Retreat`⟩ (resp.). General 1 cannot distinguish between the two cases. General 1 should follow the order ⟨`Attack`⟩ in Case II (as per C2). Given that the first and second cases look identical to General 1, in the first case also, it needs to choose ⟨`Attack`⟩. This means that in Case I, **General 2 also needs to decide on** ⟨`Attack`⟩. Now, look at the third case.

In this case, the commander is loyal and sends the order ⟨`Retreat`⟩ to both the lieutenants. Hence, the order ⟨`Retreat`⟩ needs to be followed by General 2. From the point of view of General 2, the first and third cases look exactly the same. Hence, in the first case also, **it needs to decide to**

⟨`Retreat`⟩.

We thus have a contradiction here. General 2 cannot simultaneously decide to ⟨`Attack`⟩ and ⟨`Retreat`⟩. Hence, the consensus problem is not solvable here. ■

The question that arises here is whether this result holds if we have m traitors out of $3m$ generals (including the commander). It turns out that it does. In fact there is a simple proof that is just a mere extension of Theorem 6.2.1.

Theorem 6.2.2

Assume that there are a total of $3m$ generals (including the commander) and m traitors. It is not possible to solve the Byzantine consensus problem.

Proof: Consider an instance of a regular Byzantine consensus problem with 3 generals, one of them being a commander. In a real setting, each general is a process.

Let us now create $3m$ separate processes, where each process acts like a general. On the lines of the proof in the original paper [Lamport et al., 2019], let us refer to these $3m$ generals as Albanian generals. Furthermore, let us assign m Albanian generals to one Byzantine general. This means that one Byzantine general process simulates m Albanian general processes.

The Albanian commander will be simulated by the Byzantine commander. The Byzantine commander will additionally simulate $m - 1$ Albanian generals. Now, given that only one Byzantine general is a traitor, this automatically translates to the fact that m Albanian generals are traitors, which is consistent with the statement of the theorem.

Let us now assume that we have an algorithm to solve the Byzantine consensus problems for the Albanian generals. Let us run the following algorithm then. Each Byzantine general simulates the actions of all of its constituent Albanian generals (m in total). Assume that the end result of the protocol is a solution for the Albanian general's problem.

This means that the solution obeys both our conditions: C1 (all loyal generals obey the same order) and C2 (if the commander is loyal, then all the loyal generals obey the commander).

Both the loyal Byzantine generals that simulate the $2m$ loyal Albanian generals can obey the same order that their simulated loyal Albanian generals obey. Given that all the loyal Albanian generals obey the same order, the Byzantine generals that are simulating them can also obey the same order. Let us assume that they do so. As a result, Condition C1 is satisfied

for the simulating Byzantine generals as well.

Next, consider C2. If the Albanian commander is loyal, then all the loyal Albanian generals obey the order sent by the commander. If the Albanian commander is loyal, then it automatically implies that the simulating Byzantine commander is also loyal. Let it issue the same order. Hence, the Byzantine commander and all the m Albanian generals that it simulates (including the Albanian commander), all obey the same order. There will be another loyal Byzantine general that simulates m loyal Albanian generals. All those m loyal Albanian generals will obey the message sent by their commander because it is loyal. Let the simulating Byzantine general, who is loyal, also obey the same message. Hence, we observe that C2 holds for the Byzantine generals as well. We clearly do not care about the third Byzantine commander and all the m Albanian generals that it simulates because they are presumed to be traitors.

This means that using the solution for the problem of Albanian generals, we were able to solve the Byzantine consensus problem with three Byzantine generals where there is one traitor. This is clearly impossible and violates the results of Theorem 6.2.1. As a result, there is a contradiction.

Hence, it is not possible to solve the Byzantine consensus problem with Albanian generals. In other words, if we have $3m$ processes and m processes have Byzantine faults, then we cannot solve the Byzantine consensus problem. ■

Theorem 6.2.2 is a powerful result. It basically says that no Byzantine consensus or agreement can be achieved if (for instance) we have 33 faulty processes among 99 processes. An interesting question that can be posed here is what if there are 100 processes? In this case, we have m traitors and $3m + 1$ generals. It turns out that we can. This is one of the most popular results in fault-tolerant distributed systems.

6.2.3 Consensus in the Presence of Byzantine Faults

Solution of the Problem with 4 Generals – 3 Loyal + 1 Traitor

Let us consider a simple version of the general problem where we have four generals (including the commander). At most one of them could be suffering from a Byzantine fault. We claim that there is a very straightforward solution to this problem. Assume that we have one of two cases (see Figure 6.7): the commander is a traitor and without loss of generality General 1 is a traitor.

In the first round, the commander sends an order to each of the three

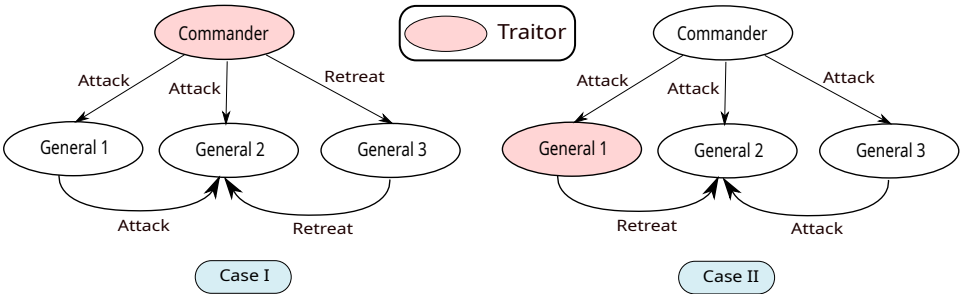


Figure 6.7: Case I: Commander is a traitor. Case II: General 1 is the traitor.

lieutenant generals. Given that the loss of a message can be detected by the receiver, let us assume that a receiver assumes that it has received the ⟨`Retreat`⟩ message if no message arrives by the end of the round. In all cases, each receiver gets a message by the end of the round.

Consider Case I in Figure 6.7. The commander is a traitor and the lieutenants are clearly not aware of this fact. The commander sends its messages. In the next round, all the lieutenant generals send the message that they received from the commander in the previous round to each other. This can be thought of a smaller version of round 1, where the original commander is not there. Instead, there are 3 groups, where each group comprises the three lieutenant generals. Each group has a message-sending general (commander) who broadcasts the message that it got in the previous round to the rest of the two generals. Hence, in the second round, each general gets two messages – one from each of the other two lieutenant generals.

Let us look at all the messages that General 2 receives, then we have a set of three messages. It receives ⟨`Attack`⟩ in the first round and {⟨`Attack`⟩, ⟨`Retreat`⟩ } in the second round. The set of messages that it receives is therefore: {⟨`Attack`⟩, ⟨`Attack`⟩, ⟨`Retreat`⟩ }. Given a set of values, we need to decide on a single value.

We can use a majority function that returns the most common value. If there are an equal number of ⟨`Attack`⟩ and ⟨`Retreat`⟩ messages, then we can decide on ⟨`Retreat`⟩. Now, consider the set of messages that all the three lieutenant generals receive (in Case I) and the corresponding majority values (see Table 6.1).

As we can see from the table, in this case all the three lieutenant gen-

General	Messages	Majority value
General 1	{ ⟨Attack⟩, ⟨Attack⟩, ⟨Retreat⟩ }	⟨Attack⟩
General 2	{ ⟨Attack⟩, ⟨Attack⟩, ⟨Retreat⟩ }	⟨Attack⟩
General 3	{ ⟨Retreat⟩, ⟨Attack⟩, ⟨Attack⟩ }	⟨Attack⟩

Table 6.1: Set of messages received by each of the three generals

erals receive the same set of messages and the majority function computes exactly the same result – ⟨Attack⟩. Condition C1 gets satisfied (all loyal generals obey the same order). Condition C2 is not relevant here because the commander is not loyal.

Now, consider the second case (Case II) in Figure 6.7. In this case the commander is loyal but General 1 is a traitor. Let us look at the messages that Generals 2 and 3 receive. Assume that General 1 told General 3 that it got the ⟨Attack⟩ message from the commander. Then, the respective sets of messages are {⟨Attack⟩, ⟨Retreat⟩, ⟨Attack⟩ } and {⟨Attack⟩,⟨Attack⟩, ⟨Attack⟩ }.

General	Messages	Majority value
General 2	{ ⟨Attack⟩, ⟨Retreat⟩, ⟨Attack⟩ }	⟨Attack⟩
General 3	{ ⟨Attack⟩, ⟨Attack⟩, ⟨Attack⟩ }	⟨Attack⟩

In both the cases, the majority value is ⟨Attack⟩. As a result, all loyal lieutenants – Generals 2 and 3 in this case – obey the same order (Condition C1). This order is ⟨Attack⟩, which was sent by the loyal commander (Condition C2). Hence, we see that both the conditions are satisfied.

There are two important conclusions that we can draw here. The first is that it makes sense to create smaller groups in subsequent rounds and share the messages received in the previous round with them. For example, we created three 3-general groups of lieutenant generals in the second round. Each group had a group leader (commander) that sent the message it had received in the previous round to members of the group. This is suggestive of a *recursive algorithm*. Second, we computed a *majority function* over of all the messages received. We showed that the majority values are the same at the end of the protocol. Hence, both our conditions C1 and C2 are satisfied.

We can furthermore conclude that in this setting with at most one faulty general out of 4 generals, the Byzantine consensus problem is solvable. The

only theoretical abstraction we are using is the fact that a recursive algorithm needs to be used where each lieutenant general in the previous round becomes the commander in the next round for a smaller set of generals. Second, we relied on the majority function that was able to filter out the erroneous information sent by disloyal generals.

Solution to the General Problem

Algorithm 40 Algorithm for Byzantine agreement

```
1: procedure runCommander(Value v, set of lieutenants ℒ)
2:     Send v to each lieutenant in ℒ
3:     If by the end of the round, a lieutenant does not receive a message,
       then it assumes it has gotten the ⟨Retreat⟩ message
4: end procedure

5: procedure runConsensus(m traitors, value v, set of lieutenants ℒ)
6:     if m = 0 then
7:         runCommander(v, ℒ)
8:         return v
9:     end if
10:    𝒱 ← ϕ
11:    for each Lieutenant i ∈ ℒ do          ▷ Treat i as the commander
12:        ℒ′ ← ℒ − i
13:        𝒱 ← 𝒱 ∪ runConsensus(m − 1, v, ℒ′)
14:    end for
15:    return majority(𝒱) ▷ Expected to be the same for all loyal generals
16: end procedure
```

Algorithm 40 shows the algorithm for solving the general problem. The first procedure runCommander works when there are no traitors. Note that the commander can still go down without sending any messages. It will never be the case that it will send some messages and fail for the rest. We assume that broadcasting a message to all $\mathcal{L}$ lieutenants is atomic (happens in one go). The commander sends its value v to all the lieutenants. Given that we are running a synchronous algorithm here, we have the notion of *rounds*. At the end of the round, if a lieutenant does not receive a message from the commander, then it can assume that it has gotten the ⟨Retreat⟩ message. Given that there are zero traitors, either all the lieutenants receive the same message or all of them do not receive a message and assume that they got

the *Retreat message*. In the latter case, it is clear that the commander has gone down. In both the cases, all the axioms of Byzantine consensus hold.

Let us now look at the more general case (runConsensus procedure), which assumes a maximum of m traitors. Let the set of all the lieutenants be $\mathcal{L}$. If there are no traitors, we run the procedure runCommander. In this case, the value simply needs to be broadcasted to all the lieutenants in $\mathcal{L}$. No other step needs to be taken because we are assuming that there are no traitors ($m = 0$).

If $m > 0$, we have the general case. For each lieutenant `i`, we run a new instance of the procedure runConsensus, where we assume that lieutenant `i` is the commander and the set $\mathcal{L} - \texttt{i}$ represents the lieutenants. This is a reduced instance of the problem, where there is one less lieutenant. We, in this case, assume that there is one less traitor, hence, we send $m - 1$ as the argument to runConsensus.

Let us broadly see what is happening here. If there are no traitors, and this fact is known, the algorithm is clearly correct. We simply run the runCommander procedure, where we simply need to broadcast the value. In the other case, for each lieutenant `i`, all the loyal generals in $\mathcal{L} - \texttt{i}$ agree on the value that was received by `i`.

Let us assume that there are a total of n generals including the commander. In the first round, the commander sends its value to $n-1$ lieutenants. In the second round, for each lieutenant `i`, we initiate a procedure for the rest of the $n-2$ lieutenants to agree on what value `i` received from its commander. Let us not care about how many rounds this process takes. Consider the point of time when this process concludes successfully. **At this point of time**, each loyal lieutenant agrees on what value was received by the rest of the lieutenants. Now, consider lieutenant `i` again. It has the value that it received from its commander, and it has $n - 2$ values that it believes the rest of the $n - 2$ lieutenants received from the commander. This *belief* is a consensus belief in the sense that all loyal lieutenants share the same set of values. The claim is that the majority value of this set is the same for all loyal lieutenants (something that we need to prove). This is the consensus value of runConsensus, which is the return value for an instance with n generals and m traitors.

This algorithm is a classic recursive construction, where to solve a bigger problem, we solve smaller instances of the same problem. Ultimately, we reach the base case where there are no traitors and a simple broadcast suffices.

Proof of the General Solution

Let us now prove Algorithm 40. Let us abbreviate the function runConsensus $(m,\mathtt{v},\mathcal{L})$ as $RC(m)$, where m is the first argument. It corresponds to the maximum number of traitors in the runConsensus algorithm. Let us assume that the rest of the two arguments are known.

Lemma 12

For any p and q, $RC(p)$ satisfies Condition C2 if there are more than $2q+p$ generals and at most q traitors.

Proof: Let us have n generals. From the statement of the lemma, we can conclude that $n > 2q+p$. In this case, we need to interpret p slightly differently. It corresponds to the number of times we shall make nested recursive calls.

Since we are evaluating Condition C2, which says that the order of a loyal commander is obeyed by all loyal generals, let us assume that the commander is loyal.

If $p = 0$, this is trivially satisfied. This is because the commander broadcasts its value to all the generals and the loyal ones obey him.

Assume it is true for $p-1$. Let us use induction to prove that this result holds for p. In the first round of the algorithm, the commander sends the value $\mathtt{v}$ to all $n-1$ lieutenants. Each loyal lieutenant will then call $RC(p-1)$.

We have the following result.

$$\begin{aligned} & n > 2q+p \\ \Rightarrow & n-1 > 2q+p-1 \end{aligned} \tag{6.2}$$

This means that runConsensus $(p-1,\ldots)$ will run correctly with the rest of our parameters (induction assumption). This further means that all loyal lieutenants who will be commanders now for their respective instances of the reduced problem ($RC(p-1)$) will broadcast the value $\mathtt{v}$ correctly.

If $p \geq 1$, then we have $n-1 > 2q$. This means that a majority of the lieutenant generals are loyal. Now, in $RC(p-1)$, a majority of the lieutenants receive the value $\mathtt{v}$ because each loyal general dutifully broadcasts $\mathtt{v}$ to the rest of the lieutenants. Given that they have also received $\mathtt{v}$ from their commander, the majority function returns $\mathtt{v}$. As a result, all the loyal lieutenant generals decide the value $\mathtt{v}$. This proves Condition C2 for $RC(p)$. ■

Lemma 12 basically establishes the fact that the if the traitors are in a minority after subtracting p from the number of generals, then Condition C2 holds. We were able to use majority-based arguments to prove that this holds because all the loyal generals will compute the same majority result. Let us now use this lemma to prove the main theorem.

Theorem 6.2.3

$RC(m)$ satisfies both the conditions – C1 and C2 – when we have at most m traitors and at least $3m+1$ generals (including the commander).

Proof: Let us use induction to prove this result.

Consider the base case, i.e., $m = 0$. In this case, we have one general who is himself the commander. He clearly obeys his own order if he is loyal. This is thus a trivial case.

Let us now consider the general case. Assume that the theorem holds for $RC(m-1)$.

Commander is loyal: In this case, let us consider the results in Lemma 12. Assume $p = m$ and $q = m$. Then the lemma stipulates that we have more than $2m + m$ $(=3m)$ nodes. The protocol follows C2. Consider now C1. This is trivially satisfied given that the commander is loyal and C2 holds (all the loyal lieutenants obey the commander's order).

Commander is a traitor: This means out of the lieutenant generals, at most $(m-1)$ generals are traitors. When we run $RC(m-1)$, we have at least $3m-1$ generals (minus the commander in the previous round and the lieutenant initiating the protocol). Now, $3m-1 > 3(m-1)$. This means that when we run $RC(m-1)$, we have more than $3(m-1)$ generals. As a result, the RC algorithm can be used with the argument set to $m-1$. It is covered by the induction hypothesis, and thus is assumed to run correctly.

Consensus Condition: Now, all loyal lieutenant generals while solving $RC(m)$ initiate an instance of $RC(m-1)$. Each instance of $RC(m-1)$ further initiates instances of $RC(m-2)$, so and so forth. However, let us not bother about this recursive process. Let us assume that $RC(m-1)$ works correctly. Let us use this fact to prove that $RC(m)$ is also correct. Given that we have n generals that also includes the commander, we can assume that each of the $n-1$ lieutenant generals starts an instance of $RC(m-1)$.

Now consider all the instance of $RC(m-1)$. Each general stores the values that it agrees to in a vector. The vector has $n-1$ entries, where

each entry corresponds to the consensus value of $RC(m-1)$ that was either initiated by itself or another lieutenant general. Note that the contents of this vector are going to be the same for all loyal lieutenant generals. Let us prove this. Given that $RC(m-1)$ works correctly, for each instance of $RC(m-1)$, all the loyal generals shall agree on the same value (Condition C1). Similarly, they have a consensus for the rest of the entries in their $(n-1)$-entry vectors.

Each vector V will be of the form $\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_{n-1}}$. $\mathtt{v_i}$ is the consensus value of $RC(m-1)$ initiated by lieutenant i. This is the greatness of this algorithm that because of Condition C1, which $RC(m-1)$ satisfies, the vectors will be identical for all loyal generals. As a result, a majority function computed on the vectors will always return the same result. If there are an equal number of $\langle$Attack$\rangle$ and $\langle$Retreat$\rangle$ entries, we can always arbitrarily choose either $\langle$Attack$\rangle$ or $\langle$Retreat$\rangle$. As long as this is a consistent choice, all the loyal generals will arrive at the same decision.

Hence, they satisfy Condition C1. Note that given that the commander is a traitor in this case, Condition C2 cannot be satisfied.

Hence, for all cases, Byzantine consensus holds. We have thus proven that the runConsensus algorithm is correct as long as we have m traitors and at least $3m+1$ generals. ■

Theorem 6.2.3 is a very important theorem. It basically says that the number of traitors have to be less than one-third the number of generals. Only then is a solution possible. Otherwise, we showed that it is not possible to find a solution to the Byzantine consensus problem. A simple majority does not suffice mainly because we cannot trust the commander. Otherwise, a simple majority would have been enough (see Lemma 12).

Message Complexity

Consider the problem $RC(m)$ with n generals (including the commander). We send $n-1$ messages to $n-1$ lieutenant generals. Each lieutenant starts its instance of $RC(m-1)$. If the time complexity of $RC(m)$ is $T(m)$, then we have the following recurrence relation.

$$T(m) = (n-1)T(m-1) \tag{6.3}$$

The solution can be expanded as follows:

$$T(m) = (n-1)T(m-1) = (n-1) \times (n-2) \times (n-m) \times T(0) \tag{6.4}$$

In this case $T(0)$ corresponds to broadcasting the value to the rest of the $n-m-1$ generals. There is no need for running any additional protocol because we are assuming that there are no traitors. Equation 6.4 thus has a solution, which is as follows.

$$T(m) = (n-1) \times (n-2) \times (n-m) \times (n-m-1) = \frac{(n-1)!}{(n-m-2)!} \quad (6.5)$$

The message complexity is indeed very large. In fact, if we use the Stirling's approximation [Cormen et al., 2009], it can be shown that it is exponential in terms of the number of traitors. This is the price for finding a consensus solution with Byzantine faults. Clearly, if we have a few traitors like 1 or 2 traitors, then the cost of the solution is not much – it is either quadratic or cubic. However, the moment we have close to $n/3$ traitors, the message complexity blows up. Over the years, many algorithms have been proposed to speed up Byzantine consensus. However, a lot is still left to be done.

Fact 6.2.1

The Stirling's approximation is as follows:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad (6.6)$$

6.2.4 Solution with Signed Messages

Given the complexity of achieving Byzantine consensus in a general setting, consider the case where we use signed messages. In this case, a general can *sign* a message indicating that he is the originator or propagator of a message. The signature of a loyal general cannot be forged. Any attempted forgery can be detected. However, it is possible to forge the signature of a traitor and traitors can also collude in this process. The key property here is that the signature of any general (loyal or traitor) can be verified.

It turns out that if we have m traitors, we can find a solution for any number of generals. The notion that a solution can only be found out for more than $3m$ generals does not hold here.

Let us number the commander as General 0. A signed message looks like this: $\mathtt{v{:}0{:}g_1{:}g_2{:}g_3}$. Here, the value is $\mathtt{v}$. It was first sent by the commander, which is numbered 0. Then, it was received by general $\mathtt{g_1}$, which was received and propagated by general $\mathtt{g_2}$. It was subsequently, received and propagated

by general g_3. The entire path is embedded in the message. Note that we can only append a list of signatures to a message. We cannot remove any existing signature. Similar to the majority function, let us define a CHOICE function here. It takes a set of values and returns one of them. It obeys the following properties.

Presence of a default value Assume that V is empty ($V = \phi$), then CHOICE (V) returns a default value that is known a priori.

Case of a singleton set Assume that V has only one element v. Then CHOICE (V) is v.

Order independent The output of CHOICE (V) for a vector of values V is independent of the order of values in V.

Choice criterion The CHOICE function needs to choose one of the values in the set V. It cannot return a value that is not present in V.

Algorithm for the Case with Signed Messages

Let us now look at the algorithm for the case with signed messages (Algorithm 41).

We start with the INIT procedure. Here, the commander sends an order to every lieutenant. If it is loyal, then only a single copy of its order will be sent. Since the commander's signature cannot be forged, all the lieutenants are bound to receive the same order. Hence, in this case both the conditions for Byzantine consensus hold and it is easy to guarantee the same.

We should concern ourselves with the other case, which is far more tricky – the commander is a *traitor*. In the INIT procedure, the commander sends its order to every lieutenant general. On receipt of the order from the commander, each lieutenant calls the RCVORDERFROMCOMMANDER function. If this is the first order that lieutenant i has received from the commander, then it adds the value v that is sent to it in a list of values (V_i). Subsequently, it sends the message v:0:i to every other lieutenant. This basically means that lieutenant i has received the value v from the commander – it thus appends its signature. The value that is received is made known to the rest of the lieutenants. Every lieutenant does the same.

Algorithm 41 Byzantine Consensus with Signed Messages

```
1: procedure INIT
2:     The commander signs an order and sends it to every lieutenant
3: end procedure

4: procedure RCVORDERFROMCOMMANDER(Order v:0, Lieutenant i)
5:     if this is the first order then
6:         Send v:0:i to every lieutenant
7:         V_i ← {v}
8:     end if
9: end procedure

10: procedure RECEIVEMSG(Message v:0:j_1:...:j_k)
11:     i ← GETLIEUTENANTID()
12:     if v ∉ V_i then
13:         Add v to V_i
14:         if k < m then                    ▷ The algorithm has not terminated
15:             send v:0:j_1:...:j_k:i to all lieutenants other than j_1 ... j_k
16:         end if
17:     end if
18: end procedure

19: procedure TERMINATE
20:     i ← GETLIEUTENANTID()
21:     Wait till m rounds complete or no more messages need to be sent
22:     Choose CHOICE (V_i) as the consensus value
23: end procedure
```

Subsequently, the commander exits the picture. The lieutenants run a protocol among themselves. They send messages to each other for m rounds, where m is the maximum number of traitors.

When a message is received from a lieutenant, the RECEIVEMSG function is called. We assume that the function GETLIEUTENANTID() returns the ID of the lieutenant invoking the function. Any message that is received has the format $\mathtt{v{:}0{:}j_1{:}\ldots{:}j_k{:}i}$. This basically means that the value $\mathtt{v}$ is received from the commander (id 0) and then it is signed by a sequence of lieutenant generals – $\mathtt{j_1 \ldots j_k}$. Here $\mathtt{j_1 \ldots j_k}$ are the IDs of the lieutenant generals. In this sequence there should be no repetition, which basically means that the ID of no general should appear twice and all the signatures should be

verifiable. A message holding this property is said to be *well-formed*.

Once a message has been verified to be well-formed (not shown in the algorithm), we proceed to check if the value that has been received (v) is a part of the set of values that lieutenant i has already received (stored in the set V_i). This check is done on Line 12. If it is already there, then nothing needs to be done. However, if it is not there, then there is a need to proceed further.

We add v to V_i (Line 13) and check the round number stored in the variable k. as mentioned earlier, we run the protocol for a total of m rounds. If $k < m$, which basically means that we are not in the last round, then the value v is sent to all the generals other than the generals who have signed the message. The generals who are excluded are $j_1 \ldots j_k$. The main aim of this action is to ensure that whatever value has been received, it is transmitted to all the generals who perhaps have not seen it (see Line 15). It is thus important to let all such generals know that such a value is in circulation. We would like to reiterate the fact that we are considering the case where the commander is not loyal. Hence, the commander has sent different values to different generals. In the case of this algorithm with signed messages, it is possible to establish that the commander is not loyal. This is because different values are in circulation with the commander's signature, and that cannot be forged ($\because$ the commander is loyal). Nevertheless, deciding a consensus value is still not easy. There is an inevitable need to send multiple messages between the generals to agree on a consensus value. Note that the final consensus value that needs to be chosen must be one of the values that the traitorous commander had sent.

This process of informing the rest of the generals about a new value that was received by lieutenant i in Line 15 has a small twist. Lieutenant i appends its signature to the message. Hence, the message looks like this: $v{:}0{:}j_1{:}\ldots{:}j_k{:}i$. It is thus clear to the rest of the generals that lieutenant i has also seen the value v.

Now, note that we are assuming a synchronous algorithm where the beginning and end of a round can be detected. If the message has k signatures other than the commander's signature, then we have completed round k. Of course, a traitor may decide not to add a signature, but this can easily be detected by a loyal lieutenant and the message can be discarded. It is not well formed in this case. The algorithms proceeds from round to round in this manner.

Let us finally look at the procedure TERMINATE. In the worst case, we wait for m rounds to complete. After that we claim that all loyal lieutenant generals have received the same set of values. This basically means that for

generals $\mathtt{i}$ and $\mathtt{j}$, $\mathtt{V_i} = \mathtt{V_j}$. Then, we can simply take the consensus value as CHOICE ($\mathtt{V_i}$). As per our claim, this will be the same for all loyal lieutenant generals. Most of the time, this may not be required because prior to this point, the condition in Line 12 that checks if ($\mathtt{v} \in \mathtt{V_i}$) will evaluate to false for all generals. If all the generals indicate that there is no message to send, then the algorithm can terminate earlier.

A word needs to be added about the feasibility of such algorithms. Everything relies on creating unforgeable signatures. This can easily be achieved with a public key based mechanism. Any standard textbook on cryptography [Stallings, 2006] or secure systems can provide a background of these techniques.

Proof of the Algorithm

Let us now look at the proof of this algorithm. It is reasonably simple and straightforward. For the case when the commander is loyal, there is no issue. Only one message will be in circulation. It will be accepted by all the loyal generals because for lieutenant $\mathtt{i}$, $\mathtt{V_i}$ will contain just one element, which the commander had initially sent. The tricky case is when the commander is a traitor. In this case, the commander can send different values to different lieutenant generals.

Let us assume for the sake of argument that there is a value $\mathtt{v}'$ such that $\mathtt{v}' \in \mathtt{V_i}$ and $\mathtt{v}' \notin \mathtt{V_j}$ for two loyal lieutenants $\mathtt{i}$ and $\mathtt{j}$. Let us consider the moment at which $\mathtt{v}'$ was added to $\mathtt{V_i}$. First, assume that it was a value that was sent by the commander. Then in the very next round, this value would have been sent to the rest of the lieutenants that would include $\mathtt{j}$. This means that $\mathtt{j}$ would have $\mathtt{v'inV_j}$. However, this is not the case. This means that $\mathtt{v}'$ was sent in a later round to lieutenant $\mathtt{i}$.

Assume the message was $\mathtt{v'{:}0{:}j_1{:}\ldots{:}j_k}$. Note that lieutenant $\mathtt{j}$ cannot be in the list $\mathtt{j_1 \ldots j_k}$, otherwise it would have already seen $\mathtt{v}'$. Lieutenant $\mathtt{i}$ can then send the value to $\mathtt{j}$ as per the logic in Line 15 assuming that the round number is not equal to m. $\mathtt{j}$ would then get $\mathtt{v}'$ and add it to $\mathtt{V_j}$ (assuming it is not already there). Hence, this case is not possible.

The only case that is remaining when $\mathtt{k} = \mathtt{m}$ and $\mathtt{i}$ receives $\mathtt{v}'$ in the last (m^{th}) round. Consider the list of lieutenants $\mathtt{j_1 \ldots j_m}$. Given that the commander is a traitor, this list can have at the most $m - 1$ traitors. This means that at least one of the generals in this list is loyal. Let this general be $\mathtt{j_x}$. $\mathtt{j_x}$ will send the value $\mathtt{v}'$ to lieutenant $\mathtt{j}$ (see Line 15). Hence, $\mathtt{j}$ will learn the value.

Our analysis indicates that there is no case in which $\mathtt{j}$ will miss the

value $\mathtt{v}'$. Hence, for any two loyal lieutenant generals, $\mathtt{i}$ and $\mathtt{j}$, $\mathtt{V_i} = \mathtt{V_j}$. As a result CHOICE ($\mathtt{V_i}$) and CHOICE ($\mathtt{V_j}$) will be the same. The output of this function will be the same for all loyal lieutenants.

This satisfies the requirements of Byzantine consensus.

Message Complexity

There is sadly no appreciable decrease in the message complexity when we have a traitorous commander – it is still exponential in terms of the number of traitors. The exact expressions are complex and non-intuitive. Let us provide a very simple reasoning. Consider the sequence of signatures in any round. They will be of the form – $\mathtt{v{:}0{:}j_1{:}\ldots{:}j_k}$ – in the k^{th} round. Note that if one of these signatures is by a loyal general, then it means that it will broadcast the value to all the generals who haven't signed and by the beginning of the next round, all the loyal generals will get the value. Traitorous generals will not be able to create new copies of this message with altered values because a loyal general has already signed (its signature cannot be forged). Hence, they will not see any value in a message that has been signed by a loyal general because they are sure that all loyal generals have received a copy of the value.

However, traitorous generals can unfortunately create new messages from thin air that bear the signatures of other traitorous generals only. We can think of traitorous generals colluding and having access to each other's signatures. They basically can forge each other's signatures with full consent. As a result, it is possible to generate $m(m-1)\ldots(m-k+1)$ messages that bear k signatures of traitorous generals. All of these messages can have different values and can be sent to the rest of the $n-m$ loyal generals. For every round, the traitorous nodes can generate such messages that have all combinations and permutations of signatures of traitorous generals. The associated values also can all be different. This will ensure that the loyal generals do not discard the messages.

This process can continue right up till m rounds.

As a result, the number of messages is in the ballpark of $m!$ (set $k = m$), which is basically exponential in terms of the number of traitors, m.

6.3 Practical Byzantine Fault Tolerance (PBFT)

There are two problems with the Byzantine agreement protocol that we have discussed. The first is that it is a synchronous algorithm. These algorithms are difficult to realize in practice. On the other hand, asynchronous

algorithms are far easier to implement in all kinds of environments. Furthermore, it requires exponential time, which makes the algorithm impossible to execute in practice. There is thus a need to design a practical version of the Byzantine agreement algorithm, which is the topic of this section. We can take inspiration from the second part of the solution to the Byzantine agreement problem, where we created a protocol based on unforgeable signatures. That algorithm was also slow, however, with some new ideas, we can significantly reduce the time complexity of the PBFT [Castro et al., 1999] algorithm inspired by it. The key idea is similar. We shall use cryptographic signatures that cannot be forged.

6.3.1 Fault Model and Objective

Let us understand the fault model. We consider a generalized asynchronous distributed system where the network is not trustworthy. It can drop messages, duplicate messages or delay them. For correct nodes, the network cannot indefinitely delay messages. Faulty nodes can suffer from Byzantine faults. The only restriction is that nodes fail independently.

The system relies on cryptographic techniques. Every node has a public key and a private key. The public keys of all the nodes are known, whereas the private key is a node-specific secret. For every message, we create a message authentication code (MAC), which is an encrypted hash. A collision-resistant hash function converts a message of arbitrary length to a 128-bit or 160-bit string. The probability of two random pieces of text having the same hash is vanishingly low: 2^{-128} for 128-bit hashes. This hash is encrypted using a shared symmetric key. This secure MAC acts as a signature of the message. It is possible for every other node to verify the ID of the signer of the message. Note that signatures of non-faulty nodes cannot be forged.

The aim is to provide both safety and liveness when less than a third of the nodes are faulty ($\leq m$ faulty processes in a system with $3m + 1$ processes).

6.3.2 Overview

Similar to systems such as Chubby (refer to [Burrows, 2006]), there is one primary server and there are a set of replicas known as *backup* servers. Let us assume that there are $\mathcal{R}$ replicas in total (primary and all backup servers). This configuration is called a *view*, which is an important concept in distributed systems. Let each view have a unique number (its sequence

number). Every view is linked to the primary server. If the primary server fails then the view changes. The next view comes into force with an incremented view number. The primary server in a view can be chosen using leader election, or can simply be a mathematical function of the view number.

The basic structure of the algorithm is as follows. Each replica has a state machine. The aim is to receive messages from client machines and apply exactly the same set of transitions to the state machines of all the correct nodes. They need to pass through exactly the same set of states given that they start from the same initial state.

1. A client sends its request to the primary server. The aim is to effect a distributed state machine transition.
2. The primary multicasts the client's request to all the backup servers.
3. All the replicas including the primary execute the request. This means that they apply the transition to their state machines. After doing so, they send a reply back to the client. If the operation produces a return value, then the value is sent along with the reply.
4. The client waits till it gets $\lceil \mathcal{R}/3 \rceil$ replies from replicas with the same value.

The objective is to ensure that all non-faulty nodes perceive the same sequence of state machine updates. Given that all of them start from the same initial state, and all transitions are deterministic, the final states should be the same.

Messages

A client C sends a message of the form $\langle \sigma_{\mathtt{C}}(\mathtt{REQ}, \mathtt{p}, \mathtt{t}, \mathtt{id}) \rangle$ to the primary server. `REQ` indicates that it is a request. `p` is the payload of the message that indicates the details of the operation that needs to be performed. `t` is a monotonically increasing timestamp, and `id` is the ID of the client. This message is signed with the digital signature of the client ($\sigma_C(\ldots)$). It can thus always be verified.

Upon receipt of this message, the primary server multicasts it to the rest of the replicas. They apply the requested operation on their respective state machines and send the return values to the client. The replicas send a digitally signed message of the form $\langle \sigma_{\mathtt{i}}(\mathtt{REP}, \mathtt{v}, \mathtt{t}, \mathtt{c}, \phi, \mathtt{i}) \rangle$ (σ_i is a private-key-based encryption function for replica i). `v` is the return value, `t` is the

timestamp of the original request, $\mathtt{c}$ is the number of the client, ϕ is the view number and $\mathtt{i}$ is the identifier of the replica. All that the client needs to do is wait for $\lceil \mathcal{R}/3 \rceil$ messages from replicas. Note that all of them need to have the same return value. The messages that have a different value cannot be counted. Furthermore, all the digital signatures need to match. If the client does not receive the requisite number of replies on time, it tries once again by sending the request to all the replicas. If a replica has already sent a reply, it sends it back again. However, if the replica hasn't gotten the original request from the primary server, it tries to ping the primary server and check if it is alive. If the primary server is not alive, then there is a need to elect a new one.

Strict ordering is maintained between requests (from the client's point of view). A client waits for a request to be fully processed before issuing the next request. Fully processing a request means that the client has received at least $\lceil \mathcal{R}/3 \rceil$ replies with the same return value and the update has been applied to the distributed state machine.

Let us now focus on the details.

6.3.3 Normal-Case Operation

Let us focus on the process that is used by the primary server to broadcast messages to the rest of the replicas. The protocol has three phases: pre-prepare, prepare and commit.

The pre-prepare phase works as follows. A short message is sent to establish message ordering by the primary server. These messages are sent to all the backup servers. This message is of the following form: $\langle \sigma_{\mathtt{P}}(\mathtt{PRE\text{-}PREPARE}, \phi, \mathtt{n}, \mathcal{H}(\mathtt{m})), \mathtt{m} \rangle$. $\mathtt{P}$ is the primary server. ϕ is the number of the view, $\mathtt{n}$ is a monotonically increasing sequence number given to the message, $\mathcal{H}(\mathtt{m})$ is the hash of the message $\mathtt{m}$. Each backup server processes these PRE-PREPARE messages. It checks if the extracted $\mathcal{H}(\mathtt{m})$ is indeed the hash of $\mathtt{m}$. Its current view should also be ϕ – the view embedded in the message. It should not have received any message in the same view with a different hash. The sequence number has to be between two thresholds: L and U (lower and upper). This ensures that a faulty primary server does not send a message with a random sequence number that is way out of bounds in the hope of exhausting the space of sequence numbers.

A backup server needs to accept a pre-prepare message. This message is accepted if all the aforementioned conditions are true. After successful acceptance, the backup server subsequently sends a prepare message to the rest of the replicas. It is of the form $\langle \sigma_{\mathtt{i}}(\mathtt{PREPARE}, \phi, \mathtt{n}, \mathcal{H}(\mathtt{m}), \mathtt{i}) \rangle$. It adds

the message to its log. Each backup server accepts the prepare message by performing the same set of checks that are done for a pre-prepare message.

The PREPARED($\mathtt{m}, \phi, \mathtt{n}, \mathtt{i}$) predicate is true if the following conditions hold:

1. Assume $|\mathcal{R}| = 3f + 1$. f is the maximum number of faulty nodes. If we have more faulty nodes, Byzantine agreement is not possible.

2. It has received a `pre-prepare` message with view ϕ and sequence number n.

3. It has received at least $2f$ prepare messages that match the pre-prepare message. Two messages are said to *match* if they have the same view number, the signatures are correct and the sequence number is between L and U.

The final aim of both these phases is to establish a total ordering of all the messages in a view. The following lemma will take us one step forward in this direction. It will prove that for two correct replicas `i` and `j`, if the PREPARED predicate holds at both of them for the same message sequence number `n`, then they have "prepared themselves" for the same message (with sequence number `n`).

Lemma 13

If PREPARED($\mathtt{m}, \phi, \mathtt{n}, \mathtt{i}$) is true, then PREPARED($\mathtt{m}', \phi, \mathtt{n}, \mathtt{j}$) is false if $\mathcal{H}(m) \neq \mathcal{H}(m')$.

Proof: This lemma basically means that two non-faulty nodes `i` and `j` agree on the global ordering of requests. It is not possible for different messages to have the same sequence number in the same view. This is ensured as follows.

The fact that replica `i` is "prepared" means that it has received $2f + 1$ matching prepare and pre-prepare messages. This is easily seen given the fact that it must have received one pre-prepare message from the primary server and $2f$ prepare messages from replicas. Out of these $2f + 1$ messages that are in the log of replica `i`, at most f can be generated by faulty nodes. Let us assume the worst case and assume that indeed f replicas are faulty. This means that there are $f + 1$ non-faulty replicas that have sent messages to node i. Let this set be S. $|S| \geq f + 1$.

Now assume that PREPARED($\mathtt{m}', \phi, \mathtt{n}, \mathtt{j}$) is true ($\mathtt{i} \neq \mathtt{j}$). Then, the same analysis can also be applied here. $2f + 1$ nodes must have sent messages to

node $\mathtt{j}$. The claim is that at least one of them must be from S. If this is not the case and S is disjoint, then we have at least $(2f + 1 + |S|)$ $(\geq 3f + 2)$ nodes in the system. This is not correct because we have only $3f + 1$ nodes. Hence, there has to be at least one non-faulty node that has sent a message to $\mathtt{j}$.

This node seems to have sent different messages to different replicas. It sent $\mathtt{m}$ to $\mathtt{i}$ and $\mathtt{m}'$ to $\mathtt{j}$. Given that it does not suffer from Byzantine faults, this is not possible. This completes our proof by contradiction. ■

Lemma 13 establishes an ordering of messages in a view. All non-faulty nodes agree with this ordering. For a given sequence number in a view, there is only one message that is sent (as believed by non-faulty nodes).

After getting prepared, a replica sends a commit message to the rest of the replicas (including itself). Its format is $\langle \sigma_{\mathtt{i}}(\mathtt{COMMIT}, \phi, \mathtt{n}, \mathcal{H}(\mathtt{m}), \mathtt{i}) \rangle$. Once replicas receive this message in their commit phase, they insert it into their logs if the message matches messages in their logs that have the same view and sequence numbers (resp.).

Let us now proceed to define a few more predicates based on the committed state.

The $\mathtt{committed}(\mathtt{m}, \phi, \mathtt{n})$ predicate evaluates to true if $\mathtt{prepared}(\mathtt{m}, \phi, \mathtt{n}, \mathtt{i})$ is true for at least $f + 1$ replicas that do not suffer from any faults. Let us now define one more predicate $\mathtt{committed\text{-}local}(\mathtt{m}, \phi, \mathtt{n}, \mathtt{i})$ that evaluates to true if and only if $\mathtt{prepared}(\mathtt{m}, \phi, \mathtt{n}, \mathtt{i})$ is true and $\mathtt{i}$ has $2f + 1$ commit messages in its logs. Needless to say, all these messages need to match the original pre-prepare message in the log of each server.

Let us now prove several lemmas.

Lemma 14

If $\mathtt{committed\text{-}local}(\mathtt{m}, \phi, \mathtt{n}, \mathtt{i})$ is true for some node $\mathtt{i}$, which is not faulty, then it implies that for the entire system $\mathtt{committed}(\mathtt{m}, \phi, \mathtt{n})$ is true.

Proof: Let us start with the assumption: $\mathtt{committed\text{-}local}(\mathtt{m}, \phi, \mathtt{n}, \mathtt{i})$ is true for node $\mathtt{i}$. This means that node $\mathtt{i}$ has received $2f + 1$ commit messages. All of them match its corresponding pre-prepare message.

We know that any node $\mathtt{j}$ sends a commit message if it is prepared. Here, we loosely use the term "prepared" to indicate that $\mathtt{prepared}(\mathtt{m}, \phi, \mathtt{n}, \mathtt{j})$ is true. This means that $2f + 1$ nodes are in the prepared state because $2f + 1$ commit messages hvae been received.

Note that we can at most have f faulty nodes. This means that at least $f + 1$ correct nodes are in the prepared state. This is precisely the definition

of the system-wide committed state. ■

Let us try to understand the implication of Lemma 14. It essentially means that a local property ensures a global property. If any node satisfies the `committed-local` predicate, then a statement can be made about the entire system. We can say that the entire system satisfies the `committed` predicate.

Once `committed-local` evaluate to true, the corresponding replica knows that the protocol for it has ended. It can apply the state transition in the message to its state machine. Subsequently, it sends a reply back to the client.

Summary of the Normal-Case Operation

Let us now quickly summarize the flow of events (refer to Figure 6.8).

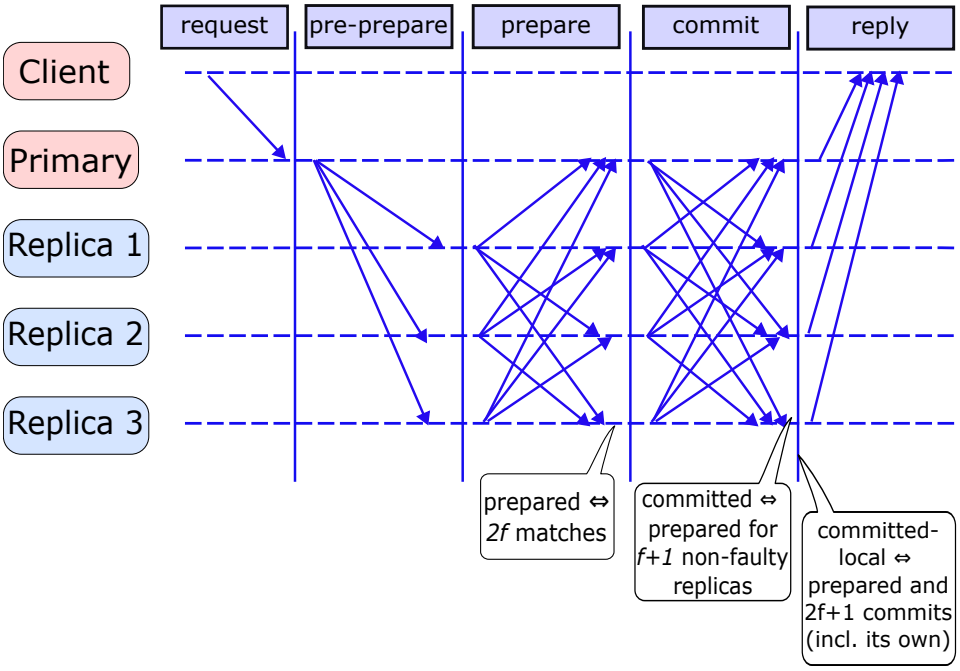


Figure 6.8: Messages sent in the normal case. Adapted from Figure 1 in [Castro et al., 1999]

The classical Byzantine agreement algorithm had an exponential number of rounds. In this case, there are only five rounds. Other than the request

and reply phases. There are three rounds. The prepare phase involves one-to-many communication: primary server sends messages to the rest of the replicas. In the pre-prepare and commit phases, we have an all-to-all message pattern unless some messages are dropped in the previous phases. These stages ensure that a local property (one node commits) translates to a global property (all the correct nodes ultimately commit the same value).

6.3.4 Garbage Collection

A lot of messages are sent in the PBFT algorithm. All of these messages are logged. This means that the logs shall grow endlessly unless additional steps are taken. There is a need to perform garbage collection periodically and clean the logs.

Recall that the primary server assigns a number to every request. Some of these requests are successfully applied to the state machine. We shall prove in the next section that all the non-faulty nodes agree on the order of such requests. Now, assume that a replica falls behind. It either goes down or its connecting links delay a lot of messages. It will end up missing a lot of the communication that is happening in the network of nodes. There is a need for it to come up to speed, once it is up again. It needs to learn about all the requests that have been successfully applied to the state machine, and it needs to start participating in the protocol for the set of requests that are currently active.

To enable this, it is necessary to periodically collect checkpoints of the system. A *checkpoint* is a point that a large number of processes agree with. We will shortly quantify what the term "a large number" means. From a qualitative standpoint, any information contained in the checkpoint should not change in the future. A checkpoint should represent *stable information* that is not subject to change in the future. We have already seen something of this nature happen in the commit phase that required $2f + 1$ commits. Similarly, in this case, if $2f+1$ replicas agree on the content of the checkpoint, then we can say that no other conflicting checkpoint can be collected in the future. Let us mathematically analyze this statement.

Consider a set of $2f + 1$ replicas. Let it have k faulty replicas. Then the number of non-faulty replicas in the set is $2f + 1 - k$. We know that $k \leq f$. Hence, $2f + 1 - k \geq f + 1$. Consider the remaining f replicas $((3f + 1) - (2f + 1))$. Assume it has k' faulty replicas. Note that $max(k, k') \leq f$ (basic assumption in Byzantine fault tolerance). We can derive the following relationships.

$$
\begin{aligned}
& f+1 > max(k, k') \\
\Rightarrow & f+1 > k - k' \\
\Rightarrow & 2f+1 > f+k-k' \\
\Rightarrow & 2f+1-k > f-k'
\end{aligned}
\tag{6.7}
$$

Let us refer to the set of non-faulty replicas that agree on a checkpoint as the *agreement set*. Its size is at least $2f + 1 - k$. Equation 6.7 implies that the size of the agreement set is greater than the number of correct replicas that do not agree on the checkpoint. The number of such replicas that disagree is limited to $f - k'$. Given that the LHS is greater than the RHS, we can conclude that there are more nodes that agree on the contents of the checkpoint than those that disagree with it. In other words, we have a simple majority among the non-faulty replicas in terms of agreeing with the checkpoint.

Taking inspiration from this result, we can design the following protocol. Let a message $\langle \sigma_{\mathtt{i}}(\mathtt{CKPT}, \mathtt{n}, \mathcal{H}(\mathtt{state}), \mathtt{i}) \rangle$ be sent to the replicas by the replica that wishes to initiate the checkpoint protocol. $\mathtt{n}$ is the sequence number of the latest request that was successfully committed. Let $\mathcal{H}(\mathtt{state})$ be the hash of the current state (of the state machine). Once $2f + 1$ such messages are received with the same contents of the hash, this checkpoint can be accepted. These messages will be sent by $2f+1$ replicas that will also include the current replica. After the checkpoint has been accepted, each node can discard all the messages that precede the checkpoint, i.e., have a sequence number less than or equal to $\mathtt{n}$. Let $\mathtt{n}$ be the sequence number of this checkpoint.

The high and low watermarks can change appropriately. The low watermark can be the number of the checkpoint. The high watermark can be a maximum limit on the number of requests that the system expects to receive before it takes the next checkpoint. This will ensure that the space of sequence numbers is compact. Given that we always increment the sequence number by 1, the primary server cannot skip sequence numbers (if it is malicious). If it decides to skip sequence numbers, this event can be detected.

6.3.5 View Changes

Let us next consider situations in which the primary server fails. This can be detected by the rest of the replicas when the request sequence numbers

are outside the range specified by the watermarks. This event can also be detected when a client times out because it has not gotten the requisite set of replies for its request. It starts contacting replicas directly. If replicas figure out that they have not received the pre-prepare messages from the primary server, they initiate leader election. This process elects a new primary server and initiates a view change.

Once a backup server realizes that there is a need to change the view, it broadcasts a request to the rest of the replicas to change the view. The message is of the following form: $\langle \sigma_{\mathtt{i}}(\mathtt{CHANGE\text{-}VIEW}, \phi + 1, \mathtt{n}, \mathtt{C}, \mathtt{P}, \mathtt{i}) \rangle$. $\mathtt{n}$ is the sequence number of the last checkpoint that process $\mathtt{i}$ is aware of. $\mathtt{C}$ is the set of valid $2f+1$ messages that established the checkpoint. Recall that we need at least $2f+1$ messages to collect a checkpoint that is *valid*. Finally, $\mathtt{P}$ is a set of sets. Each set in $\mathtt{P}$ corresponds to a request that was prepared at node $\mathtt{i}$ with a sequence number greater than $\mathtt{n}$. Each such set contains a pre-prepare message and $2f$ matching prepare messages. Let us explain the key idea here. We want to collect all the requests that have been prepared after the last checkpoint. Given that they have reached some maturity (i.e., they are prepared) and they are not part of a checkpoint, it is necessary to treat them in a special manner. They simply cannot be discarded. There is a need to process them in the subsequent view.

Next, consider the new primary server of view $\phi + 1$. Once it receives $2f$ view-change ($\langle \mathtt{CHANGE\text{-}VIEW} \rangle$) requests from other replicas, it is ready to initiate the creation of the new view. This means that it needs to inform all the replicas about this development. Consequently, it multicasts the following message: $\langle \mathtt{NEW\text{-}VIEW}, \phi + 1, \mathtt{V}, \mathtt{R} \rangle$ message. $\mathtt{V}$ is the set of valid view-change requests that it has received. Computing $\mathtt{R}$ is more interesting. We shall see that this set has something to do with all the oustanding prepared requests that are not a part of any checkpoint.

Remark 6.3.1

We would have noticed that while initiating a view change and while creating a new view, the concerned replica sends all the messages that it has received. For example, it is not just sufficient to report the requests that have reached the prepared state – it also sends the messages that made it arrive at this conclusion. Similarly, in the case of starting a new view, it is also necessary for the new primary server to send all the view change requests that allowed it in the first place to start a new view. This means that no replica will just believe what another replica claims. Adequate proof needs to be supplied

in terms of the received messages. Then only another replica will believe the claim.

The reason for doing this is quite simple. We are creating a protocol to provide fault tolerance in the presence of Byzantine faults. We simply cannot take the word of any replica. We need to see some proof. The reason the proof can be trusted is because this protocol relies on strong cryptographic assumptions. It assumes that digital signatures cannot be forged. This means that replicas can still lie about themselves, but one replica cannot masquerade as another one. This is the crux of this protocol.

Construction of the Set R

First, consider set V. It contains all the view-change requests that have been received. Recall that each such request has the details of the latest checkpoint. In set V, let us find the smallest and largest sequence numbers. Recall that each checkpoint has a number associated with it. Let these two numbers be V_{min} and V_{max}, respectively. We can thus safely conclude that all the processes that are a part of the new view have seen all the state that is a part of V_{min}. Let us consider the rest of the checkpoints in the range $(V_{min}, V_{max}]$.

The new primary server needs to ensure that all the messages with these sequence numbers are multicasted to the rest of the replicas (in case, they have been missed earlier). Hence, it iteratively considers every number in the range $[V_{min} + 1, V_{max}]$. For sequence number k, it first checks if it is present in the set P of any view-change message that it has received (including its own). If it is present, then it means that the corresponding message has been prepared by some node in the past. If there is only one mention of a request with sequence number k, then we do not have a choice. However, if mtuliple replicas report preparing a message with sequence number k, then we choose the message from the view with the highest view number. Clearly, there cannot be two different prepared messages with sequence number k in that view as per Lemma 13. Let the contents of this message be m.

In such a situation, the following procedure is followed. The primary server p creates a message of the form $\langle \sigma_{\mathtt{p}}(\texttt{pre-prepare}, \phi + 1, \mathtt{k}, \mathcal{H}(\mathtt{m})) \rangle$ and sends it to the rest of the replicas. It is as if this request is being multicasted from the primary server for the very first time.

Next, consider the second case, where the request is not present. In this case, a pre-prepare message is also sent; however, the hash of the message is set to null. All these pre-prepare messages are added to the set `R`. It serves as a proof that adequate steps were taken to update the state of all the members in the view.

This is like providing a proof of stability. It convinces the rest of the replicas that the primary has the most up to date information insofar as all the prepared messages are concerned. If they were prepared at some replica that acknowledged the view change request, then the primary will ensure that the rest of the replicas also have a copy of it.

Whenever a replica receives the set `R` from the primary, it proceeds in a similar fashion. If it does not have a request it in its log, then it sends the prepare message to the rest of the replicas. It is important to understand that this entire process is being done to come up to speed with the fastest replicas. The client should be blissfully unaware of this process. This means that no client request should be re-executed. If a reply has already been sent to a client for a certain request, then there is no need to send another reply.

Furthermore, given that $2f+1$ acknowledgements are necessary to change a view, we can say for sure that at least $f + 1$ replicas in this set are correct. This means that if some request or information from some checkpoint is missing, then any primary or even any backup replica can obtain this information out of all the replicas that acknowledged the view change. This is one of the advantages of sending `V` in the view change message. Recall that `V` is the set of all the view-change messages that have been received.

6.3.6 Proof of Correctness

There are two aspects to correctness: safety and liveness.

Safety

Safety implies that all non-faulty replicas agree on the order of requests that are applied to the state machine. In other words, it is the same order of local commits.

Sanctity of the Prepared State and Local Commitment

We need to understand that the prepared state has some sanctity. Consider two non-faulty replicas. For the same sequence number, it is not possible for two such replicas to prepare two different messages (Lemma 13). This

means that once a request is prepared, it represents a state of permanence. No other request with the same sequence number can get prepared if its contents are different.

Next, consider Lemma 14. It proves that if a replica has committed a sequence number locally (received $2f + 1$ $\langle$`commit`$\rangle$ messages), then the sequence number has committed globally. At this point, the request can be applied to the state machine subject to the fact that requests with smaller sequence numbers have already been applied.

Thus, it is obvious that if there is no view change, then all replicas agree on the order of messages. It is not possible for two non-faulty replicas to agree on a different order of requests. Let us now consider view changes. It is not possible for a new replica to accept a $\langle$`pre-prepare`$\rangle$ message from a future view with sequence number $(\phi + 1)$ without getting a view change request first.

Guarantees of Safety Properties when the View Changes

Consider the view change process. Consider sequence number n. If a request has been committed, then it means that at least $f + 1$ non-faulty replicas have prepared the request. Let all these replicas be a part of S_1. Now, consider another set S_2 that sent valid view-change messages. It has at least $2f + 1$ entries. $\mid S_1 \mid + \mid S_2 \mid > 3f + 1$. This means that they have at least one replica in common. In other words, some view-change message will have information about this common replica. Once a request has been committed, it will always be a part of any subsequent view-change message. Hence, the information about request commitment will never get lost. This is the key property that ensures correctness. It establishes the *finality of commitment*.

Let us look at some details. It is possible that the request got prepared in a previous view and is propagated as a part of the view-change request. If this is a part of a checkpoint, nothing needs to be done. However, if it is not a part of a checkpoint, there is a need to send a fresh $\langle$`pre-prepare`$\rangle$ message and initiate all three phases of the protocol. Ultimately, this request will be covered by a checkpoint, and there will be no additional need to track it.

Liveness

The system should not enter a livelock where messages are being sent, but there is no progress. Now, it is clear that if the primary server goes down, then this event will be detected. The client will let the replicas know that

it is has not gotten the adequate number of replies. They will detect the non-availability of the ⟨`pre-prepare`⟩ message and start the process to elect a new primary server.

The key question is how long should the new primary server wait to receive $2f + 1$ view-change messages before it can change the view. If the waiting time is too short, then the protocol will again time out. A need will arise to change the view once again. On the other hand, if it is too long, then the protocol will appear to be unresponsive to node failures. It is important to adapt to the system and prevailing network delays, and properly calibrate the time-out intervial. In general, an exponential backoff-based strategy is followed. The timeout is first T, then $2T$, $4T$ and so on.

If the primary server is faulty, it can deliberately delay messages and induce view changes. There is a need to thus rotate the primary server among the replicas across views. It will ensure that no single replica can delay and derail the process indefinitely.

6.4 The Paxos Algorithm

We know that we are naturally constrained by the FLP result when it comes to designing a consensus protocol in an asynchronous setting with at least one faulty process. However, it is important to understand that an asynchronous setting with one or more faulty processes mimics most real-world scenarios. Consider the case when we are trying to book an airline ticket. We clearly need a consensus between the app running on the phone, the airline, the travel agency, the credit card company and the bank that issued the card. Only if all of them agree, then only the transaction can go through and a ticket can be booked. Otherwise, even if one of them disagrees, then the transaction needs to be aborted. Here again, there needs to be complete agreement about aborting the transaction. In other words a consensus needs to be reached.

Many times it so happens that money has been deducted from a bank account, but the ticket has not been booked. This is a sad reality of a lot of travel sites that do not implement consensus protocols properly. This creates a lot of distress among passengers, and then they find themselves requesting the travel site on the phone to either give them a ticket or refund the money. It is thus important that we design consensus protocols for such settings, which at least work most of the time.

The FLP result basically says that we cannot guarantee both *safety* and *liveness*. In this case, safety basically means that the processes do not end up

deciding different values. Liveness means that for every process, a consensus decision is made within a finite or preferably bounded amount of time. Because of the FLP result, it is clear that we need to sacrifice either safety or liveness. Clearly safety cannot be compromised, hence liveness needs to be the casualty here. This means that in a rarity of cases, the protocol will not terminate. We will just keep on trying to achieve a consensus over and over again, yet a consensus will never be reached. This scenario is also known as a *livelock* where every process continues to take internal steps but is never successful.

Definition 6.4.1 Livelock

A *livelock* is defined as a situation in a concurrent system where every process continues to make progress by taking internal steps, however no process is ever successful in achieving its desired outcome, i.e., achieving consensus. Despite the seemingly constant progress, it is possible that for indefinite time no process will ever reach the consensus state (finally decide on a value). This is not the same as a deadlock, where processes are not able to make any progress at all and there is a cyclic dependency between them.

6.4.1 Basic Concepts

Let us now proceed to discuss the basic concepts underlying the classical Paxos algorithm that was originally proposed by Leslie Lamport [Lamport, 1998]. The algorithm was fairly complex and a lot of people did not understand it. As a result, its adoption was initially quite difficult. Later authors including Leslie Lamport himself published simplified versions of the Paxos algorithm that were far more straightforward and easier to understand. The version of the algorithm that we will be presenting in this chapter is a simplified version of Paxos; it was published in 2001 and is referred to as simplified Paxos [Lamport, 2001].

Here, the overall idea is quite simple. We have three kinds of processes in the system: proposers, acceptors and learners. The job of a proposer is to propose values that will be considered by the consensus protocol. The job of an acceptor is to temporarily accept values and ensure that the protocol terminates. A learner joins the system much later and tries to learn the value that has been chosen as the consensus value.

6.4.2 Conditions

Let us define a set of conditions that must hold at all points during the operation of the algorithm. They may look like being overly conservative, however they are needed to realize a fast, simple and efficient consensus protocol in an environment that is asynchronous as well as contains slow/faulty processes.

Definition 6.4.2 C1: First-Accept Condition

An acceptor does not know how many proposals are currently there in the system. She has a limited view of the system. Hence, she needs to *accept* the first proposal that she gets

Let us understand the rationale behind C1. Many values can be simultaneously proposed by different proposers. Other processes will not be aware of all of them. This is because in a concurrent system, a process has a very limited view of the system. It will only be aware of its own state and have a sketchy view of the global state based on the messages that it has received. This means that an acceptor process needs to possibly accept multiple proposals. In this case, proposal acceptance is a temporary step. This is not the same as finally *choosing* or deciding on a proposal. It is a provisional acceptance.

Now, consider the first proposal that an acceptor gets. She does not know if this will be the last proposal message or not. It very well may be the last proposal that she gets. Hence, she has no choice but to accept it.

Before introducing the next condition, let us outline the structure of a proposal. It is a tuple of a number and a value.

$$proposal \rightarrow (number, value)$$

The proposal numbers can be lexicographically ordered – this makes all the proposal numbers unique. Let us now come to the second condition, which is known as the Consensus condition (C2).

Definition 6.4.3 C2: Consensus Condition

If a proposal (n, v) is *chosen*, then every proposal that is chosen with a higher number, needs to choose the value v.

C2 basically establishes consensus. It essentially establishes the finality of consensus. Once a value is chosen, then no other value can be chosen. We can look at this another way. Take the least proposal number that is

chosen. Note that all the proposal numbers are comparable here, hence the notion of the "least number" exists. Now, every higher-numbered proposal needs to choose the same value as per the definition of consensus.

To satisfy C2, we need to define two sub-conditions, which are more restrictive.

Definition 6.4.4 C2a: Acceptance Condition

If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor also has value v.

Condition C2a makes things more restrictive. It basically puts a restriction on proposal acceptance as well. It says that after a proposal is *chosen* by the system, no higher-numbered proposal that conflicts with the chosen value can be accepted. This means that if we have chosen v, henceforth, for any higher-numbered proposal we can only accept v. No other value can be accepted.

Now, this seems to violate condition C1, which is the First-Accept condition. Assume a process was sleeping. It suddenly wakes up. By the First-Accept condition it needs to accept the first proposal that it gets. This could very well be a proposal that conflicts with the chosen value. Given that the process that just woke up does not have any additional visibility, it needs to accept the proposal. Even though this satisfies condition C1, it violates condition C2a.

We thus need to add one more condition to ensure that this does not happen. This will basically make matters even more restrictive.

Definition 6.4.5 C2b: Issuance Condition

If a proposal with value v has been chosen, then every higher-numbered proposal issued by any proposer can only have value v.

In this case, we are placing a restriction on issuing proposals. We are saying that no process can even issue proposals with other values once a consensus decision has been made. This means that the sleeping process in the counter-example that we just discussed for condition C2a will never get a proposal that has any value other than v. Hence, it can happily use the First-Accept condition (C1) to accept the first proposal that comes to it. This will not cause a problem.

Basically, C1 is a necessity because every process needs to make forward progress – in a certain sense it ensures liveness. However, for consensus, we need to ensure condition C2b.

6.4.3 The Algorithm

The algorithm is very short and simple. It is quite surprising that it works. The algorithm has two phases. Phase 1 (run by the proposer) is shown in Algorithm 42. It starts with a call to the method PROPOSE.

Every process has an internal variable called `cnt` (count) and `val` (value). The count is incremented when the PROPOSE method is invoked with argument `v`. First we initialize the process variable `val` to `v`. `val` is the value that the given process is going to propose if it is allowed to do so.

There are two more internal variables: `maxPrep` and `maxAccept`. `maxPrep` is the number of the highest-numbered proposal received by the current process. `maxAccept` is a proposal, which is a (number,value) tuple. It is the highest-numbered proposal accepted by the current process.

Algorithm 42 Paxos: Phase 1

```
1:  procedure PROPOSE(v)
2:      ▷ Called by the proposer
3:      cnt ++
4:      val ← v
5:      Send ⟨prepare(cnt)⟩ to a majority of acceptors
6:  end procedure
7:
8:  procedure RECEIVEPREPARE(n)
9:      ▷ Called by the acceptor
10:     if n > maxPrep then
11:         maxPrep ← n
12:         return maxAccept
13:     end if
14: end procedure
```

Whenever a process wants to propose a new value, it increments `cnt` and sends a ⟨`prepare`⟩ message with `cnt` to a majority of acceptor processes (referred to as the *quorum* henceforth). The reason that we use a *majority* here is that if another process does the same, then there will at least be one acceptor in common. We will see that the common acceptor plays an important role in the proof.

When a process receives a ⟨`prepare(n)`⟩ message, it calls the RECEIVEPREPARE function. This is where it compares the number of the proposal with the internal variable `maxPrep`, which is the highest proposal number that it has received. It wishes to consider only higher-numbered proposals. It

checks if `n` > `maxPrep` in Line 10, and enters the body of the *if* statement only if `n` is greater. This means that it always wishes to consider newer proposals. In this case, it sets `maxPrep` to `n` and returns a message with the value of the internal variable `maxAccept`, which is actually a proposal – a tuple of the proposal number and proposal value. We will henceforth refer to them as `maxAccept.n` and `maxAccept.v`, respectively. Note that here `maxPrep` is a monotonically increasing variable. `maxAccept` is the highest-numbered proposal that the current process has accepted. Note that if no proposal has been accepted as yet, then a null value needs to be returned.

There is a fine print here that is not easily visible. The first point to note is that if `n` < `maxPrep` then the message is pretty much absorbed, nothing is returned. However, if `maxAccept` is returned, then it is returned along with a *promise*. The promise is that no proposal with a number less than or equal to `maxPrep` will ever be entertained in the future. This promise is important because it helps us clean up lower-numbered proposals from the system and ensures the monotonicity of `maxPrep`, which is important for guaranteeing consensus.

Once the messages with `maxAccept` proposals are received from all the acceptors, the algorithm proceeds to the second stage (shown in Algorithm 43). Note that if a process sends a proposal with a low number, it will simply not get replies from many acceptors because the check `n` > `maxPrep` will fail. We shall proceed to Phase 2 only if a majority of acceptors reply with their `maxAccept` proposals.

Remark 6.4.1

Summary of Phase 1: Phase 1 of the algorithm achieves two things for the proposer: First, it extracts a promise from each acceptor that it will never respond to a proposal with a number less than the proposal number n. Second, it gets the highest-numbered proposal accepted by each acceptor (null if no proposal has been accepted).

When a process receives replies from a majority of acceptors, it transitions to Phase 2. This informally means that the number of its proposal is high, which is why it got all the replies. It also means that when each of the acceptors had gotten the current proposal in Phase 1, they had not replied to any higher-numbered proposal. Hence, entering Phase 2 indicates the recency of the proposal. It also indicates that each of the responding processes will not accept a lower-numbered proposal.

Algorithm 43 Paxos: Phase 2

```
1: procedure receiveMaxAccepts(maxAccepts[])
2:     ▷ Called by the proposer
3:     ▷ received maxAccept messages from a majority of acceptors
4:     (n, v) ← max(maxAccepts)
5:     if (n, v) = φ then
6:         (n, v) ← cnt, val
7:     else
8:         (cnt, val) ← (n, v)
9:     end if
10:    send ⟨accept(n, v)⟩ to all the acceptors in the quorum
11: end procedure
12:
13: procedure receiveAccept(n, v)
14:    ▷ Called by the acceptor
15:    if n ≥ maxPrep then
16:        maxAccept ← (n, v)
17:        accept the proposal (n, v)
18:        send a ⟨response⟩ to the proposer
19:    end if
20: end procedure
21:
22: procedure receivedAllResponses
23:    ▷ Called by the proposer
24:    choose val
25: end procedure
```

In the function receiveMaxAccepts, we first find the maximum numbered proposal out of all `maxAccept` proposals received (from the majority of acceptors). We put all the `maxAccept` proposals received in the array `maxAccepts` and then find the maximum. Let us refer to this proposal as $(\mathtt{n}, \mathtt{v})$ (number,value). If all the received proposals are null because no proposal has been accepted till now, then the proposer uses the value that it is proposing $(\mathtt{cnt}, \mathtt{val})$. Note that this is the only instance, when the proposer gets to propose its own value. Condition C2b says that once a decision has been made we need to stop proposals with other values even from getting issued (sent to the other processes). This is the point at which this is enforced. A process only gets to *propose* if no other process in its quorum has accepted any proposal. The next step is to send $\langle \mathtt{accept}(\mathtt{n}, \mathtt{v}) \rangle$ to all the

acceptors in the quorum.

When an acceptor gets this ⟨`accept`⟩ message, it invokes the function RECEIVEACCEPT. The arguments are the proposal number `n` and the proposal value `v`. The acceptor compares `n` with `maxPrep` again (see Line 15). Recall that in Phase 1, this comparison was made for the first time (Line 10 in Algorithm 42). The reason for comparing for a second time will be clear in the proof. However, informally it means that no other proposal was received in the intervening period with a higher number.

There is a general point to be made here. Many distributed algorithms work on the principle of the "period of quiescence". This means that they rely on a small period of time when no other message has been sent or received – a period of quietness or *quiescence*. The existence of such a period can of course be checked with checking a variable twice as we are doing in Line 10 in Algorithm 42 and Line 15 in Algorithm 43. We shall see that having such a period of quiescence has one advantage and one disadvantage. The advantage is that we can prove that in the distributed system some global state must have been reached – all or majority of the nodes must have agreed on some property. However, the disadvantage is that we may never have such a period of quiescence and the algorithm will just continue forever. This, in fact, is a negative side of consensus algorithms that guarantees safety but not liveness. Recall that this is a direct consequence of the FLP result.

Now, let us restart our discussion from Line 15 (Algorithm 43). If the `n` $\geq$ `maxPrep` check succeeds, then we set `maxAccept` (internal variable of the process) to the received proposal (`n`, `v`). This is tantamount to **accepting** the proposal (`n`, `v`). A response can be sent to the proposer informing it about the same.

Once all the responses are received, the proposer can end Phase 2 by invoking the function RECEIVEDALLRESPONSES. The value `val` (value that was proposed in the ⟨`accept`⟩ message) can now be chosen as the consensus value.

We need to appreciate how the First-Acceptance condition (C1) and the Proposal Issuance condition (C2b) are being ensured. Consider the first message that an acceptor gets. It is simply accepted unless another higher-numbered proposal has contacted it between Phase 1 and Phase 2. If there is a period of quiescence where no such higher-numbered proposal is received, then the first proposal is accepted. A subsequent proposal can also be accepted by an acceptor as long as it satisfies the condition `n` > `maxPrep`.

Once a value is chosen, it is clear that a majority quorum has set its `maxAccept` values. This is because at the end of Phase 2, the ⟨`accept`⟩

message (with the value that will be used to set `maxAccept`) is broadcast to a majority of the processes, and we wait for their responses. Once the responses are received, we can be sure that they have set their respective `maxAccept` values. Now, the claim is that no new proposal can be issued or in other words enter the system after this (C2b). This is clearly going to happen because in the beginning of Phase 2, the proposer asks the processes in the majority quorum for their `maxAccept` values. After a proposal is chosen, this set will have at least one non-null value. Thus, the condition for adding a new proposal to the system is not getting satisfied here. A new proposal can only be added only if all the `maxAccept` values received are null, which will not be the case here.

Hence, we see that this algorithm is obeying both the conditions: C1 and C2b.

Remark 6.4.2

Summary of Phase 2: The proposer receives a set of `maxAccept` proposals from the majority quorum. It finds the highest-numbered proposal and if all are null, then it makes a proposal. This is then multicasted back to the majority quorum. If no higher-numbered proposal has arrived in the time being, then an acceptor **accepts** the value that is being sent to it by the proposer. This involves setting the value of `maxAccept` and sending a response back. Once the processes in the majority quorum send their responses back, the value sent in the ⟨`accept`⟩ message can be chosen.

6.4.4 Analysis of the Paxos Algorithm

Consider two proposals P_1 and P_2. Consider an acceptor A that receives P_2's ⟨`prepare`⟩ message after P_1's ⟨`accept`⟩ message. Then we say at A, P_1 precedes P_2 or alternatively $P_1 \prec P_2$. This means that pretty much in this two-proposal setting, the ⟨`prepare`⟩ message is received after the period of quiescence at A.

Now, let's say P_1's ⟨`prepare`⟩ message is ignored (does not pass the check in the *if* statements). Assume that A receives P_2's ⟨`prepare`⟩ message after ignoring P_1's ⟨`prepare`⟩ message. Then also we say that at A, $P_1 \prec P_2$.

It is possible that both the ⟨`prepare`⟩ messages are *concurrent*. This means that neither $P_1 \prec P_2$ nor $P_2 \prec P_1$ is true (at A). Then we say that $P_1 \bowtie P_2$ (at A). This is equivalent $P_1 \not\prec P_2$ and $P_2 \not\prec P_1$ (at A). We say that $P_1 \bowtie P_2$, if $P_1 \bowtie P_2$ at any acceptor.

Let us now prove a bunch of lemmas regarding the properties of this protocol.

Lemma 15

If $P_1 \bowtie P_2$ and $P_1.n < P_2.n$, P_1 will not pass Phase 2 at the common acceptor.

Proof: Given that these messages are sent to a majority quorum, there must be a common acceptor. This is because an intersection of two majority sets is always non-empty. Let us thus assume that there is a common acceptor A. Assume that it got the $\langle$`prepare`$\rangle$ message first from P_1. Given that $P_1 \bowtie P_2$, P_2's message will arrive before P_1 crosses Phase 2. P_1 will thus either set the value of `maxPrep` or `maxPrep` will be greater than $P_2.n$ (also $P_1.n$). In the first case, P_1 will not cross the second phase because $P1.n < P2.n$, and P_2 would either end up setting the value of `maxPrep` or `maxPrep` would be greater than $P2.n$ (and also $P_1.n$). In the second case `maxPrep` $> P_2.n > P1.n$ and thus P_1 will not pass the check in Phase 2. In either case `maxPrep` will be greater than $P_1.n$ at the time of the checking in Phase 2, and thus P_1 will fail the check in Line 15 (Algorithm 43). It will not pass Phase 2 (if at all it crosses Phase 1).

Now assume the other case where it first gets the $\langle$`prepare`$\rangle$ message from P_2. In this case, `maxPrep` $\geq P_2.n$. As a result, P_1 will fail the check where `n` is compared with `maxPrep`. It will thus fail Phase 1. Hence, the question of passing Phase 2 does not arise. ■

Lemma 16

If $P_1 \prec P_2$ and $P_1.n > P2.n$, P_2 will not pass Phase 1.

Proof: Let us again consider the common acceptor A. P_1 would have either set the value of `maxPrep` in Phase 1 or found out that `maxPrep` is greater than its proposal number. This means that when P_2 arrives, it will definitely find `maxPrep` to be greater than its proposal number in Phase 1. It will thus fail Phase 1. ■

Let us understand what the lemmas are giving us in terms of results. Lemma 15 says that it is not possible for two proposals to pass both the phases at all acceptors if they are concurrent. Lemma 16 further refines this notion. It says that the proposal numbers should obey the same relationships as the precedence relation. If $P_1 \prec P_2$, then $P_1.n < P_2.n$. This basically means that if two proposals P_1 and P_2 succeed (pass Phase 2 at all acceptors) then we can assume without loss of generality that $P_1 \prec P_2$, which implies

$P_1.n < P_2.n$. We now want to prove the main theorem, which is that for all such pairs of proposals, they end up *choosing* the same value.

Theorem 6.4.1

If $P_1 \prec P_2$ and both of them succeed in passing Phase 2 at all acceptors, then they must choose the same value.

Proof: We know that $P_1.n < P_2.n$ (from Lemma 15 and Lemma 16).

Assume that they choose different values. P_1 chooses v_1 and P_2 chooses v_2. P_1 and P_2 must have a common acceptor A. After A received P_1's $\langle$`accept`$\rangle$ message, it must have received P_2's $\langle$`prepare`$\rangle$ message. It must have sent it the proposal containing value v_1.

However, for some reason P_2 did not choose it. This is basically because it must have a gotten a higher-numbered `maxAccept` proposal with value v_3 from another process in its quorum. Let the proposer of v_3 be P_3.

The question is, how did P_1 miss v_3 (proposed by P_3)? Since P_3's proposal could beat P_1's proposal, it means that $P_3.n > P_1.n$. Given that P_1 succeeded, it means $P_1 \prec P_3$. P_3 must have solicited `maxAccept` messages from a majority quorum. It would have clearly seen a non-null value because by this time P_1's accept message would have reached a majority quorum. This means that the proposer of P_3 could not have proposed a new value, i.e., P_3's value. This is because a new value can only be proposed if all the `maxAccept` messages are null, which in this case is not going to happen. As a result, P_3's value cannot be issued as a proposal (in line with Condition C2b).

Since P_3's value was never circulated, it could not have been accepted by P_2. There is thus a contradiction here.

Hence, P_1 and P_2 will choose the same value. In fact, once a value is chosen, no other value can be chosen. ■

6.5 The Raft Consensus Protocol

6.5.1 Overview

The Paxos protocol that we just described is basically for a single consensus round. If we want multiple rounds, then it starts becoming more complex and starts approaching the original Paxos protocol that Leslie Lamport had proposed [Lamport, 1998]. Also, the Paxos protocol is a classical concurrent algorithm where a majority of nodes need to periodically participate

in the consensus process. To make a decision, a period of quiescence is required. The community found this class of algorithms to be very difficult and thus the acceptability of such complex protocols has always been quite low. Hence, for a long time, people wanted to design a much simpler protocol, which is very easy to implement, understand and verify. We need to bear in mind that if a protocol is complex, then coding and verifying it also becomes a very onerous task. Furthermore, its code does not remain maintainable because most developers find the algorithm and its code very hard to understand. Keeping all of these requirements in mind, Raft [Ongaro and Ousterhout, 2014] was proposed in 2014.

The Raft protocol operates like a distributed ledger (similar to many cryptocurrencies). There are many client and server processes. One server process is elected as a *leader process*. Each server maintains a state machine. The state machines across the servers are expected to be synchronized. Client machines send their requests (proposals in Raft) to the leader server. A request performs some operation on the state machine and changes its state. All the servers need to agree on the global order of the requests. If they do (consensus condition), then they can compute the current *state* by applying all the requests one after the other to each state machine. Given that all the servers already agree on the order of requests, the final state will be the same in each state machine, even though all the servers compute it separately. This is a standard paradigm in distributed systems. The current state is a result of a sequence of transitions applied to a state machine that is initialized to a given state.

Observation 6.5.1

In a distributed system, the current state of any process is a result of applying a sequence of transitions to the initial state. Every process acts like a state machine. If two processes agree on the initial state and the state transition sequence, then their current states are the same.

Let us outline the role of a *leader*. It accepts requests from clients and multicasts them to the rest of the servers. Furthermore, it establishes a global order of client requests and conveys this global order to rest of the servers. The servers then know the order in which these requests need to be processed (applied to the state machine). This will ensure that all the state machines remain *synchronized*.

There is a possibility that the leader fails. In this case, a new leader needs to be elected. The new leader may fail again. It is also possible that a

leader that was hitherto known to have failed suddenly wakes up. All sorts of tricky situations are possible. There is thus a need to divide time into *terms*, where each term has a unique leader. To ensure that an old leader that had gone to sleep does not mistakenly assume that it is still a leader, we associate a monotonically increasing sequence number with each term. This helps us identify leaders from previous terms. It is possible to make them realize that they are not leaders anymore.

Before we proceed further, a disclaimer is due. The Raft protocol works in spite of node failures, network packet losses and network partitions. However, it is not resilient to Byzantine failures. This means that it assumes that processes can fail and be slow, however they do not lie.

6.5.2 Safety Properties

Property	Explanation
Election safety	Every election produces a *single* leader.
Leader append-only	A leader's existing log is immutable – no entry is deleted or overwritten. New entries can only be appended.
Log matching	Assume that two logs contain the same entry at a given index, and they belong to the same term. The Log Matching property ensures that the logs are identical till that index.
Leader completeness	This deals with the finality of commitment. If an entry is *committed* by the leader in term T, it is present in the logs of the leaders of all terms with numbers greater than T.
State machine safety	There needs to be a consensus on the contents of the logs before they are applied to the state machine. Specifically, if a log entry at a certain index is *applied* to the state machine by the leader, then all the servers can also apply it to their copy of the state machines.

Table 6.2: Safety Properties in Raft

Let us understand each of these safety properties. We divide time into a set of *terms*, where each term can have at most one leader. This is known as *election safety*. In a certain sense, we are creating a centralized algorithm.

However, we are accounting for the fact that a node may fail, and thus it would be necessary to elect a new leader. In this case, we create a new term and elect a new leader. The log is basically a linked list that comprises all the changes that need to be applied to the shared state machine. This is immutable in many senses. In this context, this property basically means that it is not possible for a leader to overwrite or delete entries in the log – it can only *append* new entries.

There is a need to reconcile the logs across servers especially when there are leader changes. Hence, for each entry we also add an index, which is pretty much its sequence number in the log (starting from 0). In essence, we are treating a log as an array where the index can be interpreted as the array index. The *log matching* condition states that if two logs contain the same entry at a given index, then the logs are identical till that index. This can be thought of as a consensus condition on the logs. We shall see later that the algorithm guarantees a far stronger condition, which says that if two servers have j and k entries respectively in their logs where $j < k$, then the larger log needs to contain all the entries in the smaller log.

Raft distinguishes between propagating a state machine update to all servers and committing it. Whenever a client makes a request, it is sent to the leader server. The leader adds it to its log in an uncommitted state and then broadcasts it to all the servers. Only after getting a response from a majority of servers does it actually *commit* it. In this case, commitment means that the entry is permanently added to the log. Logs thus have a certain sense of immutability, when it comes to committed entries. Raft guarantees that if an entry is *committed* in a given term, it will be present in the logs of the leaders of all successive terms. This is precisely implied by the notion of commitment. This property is known as *leader completeness*.

The final condition *state machine safety* is the consensus condition. It says that if a server has *applied* a log entry to the state machine – made it transition according to the entry – then all the other servers will also apply the same entry (stored at the same index) to their respective state machines. All the servers need to basically see a common view of the log. Let us elaborate.

Periodically, the leader server sends messages to the rest of the servers. It ensures that all of its committed entries reflect as committed entries in other servers as well. If an entry is committed in a log, then we assume that all of its previous entries (with lower indexes) are committed as well. It further means that all the servers will eventually see the same order of committed entries in their logs. A log entry can then be *applied* to a state machine. Note that we can apply a log entry only if it is committed.

6.5.3 Overview

Figure 6.9 shows an overview of the scheme. A Raft cluster typically contains 5 or more servers. A server has three states: leader, follower and candidate. The names are self-explanatory.

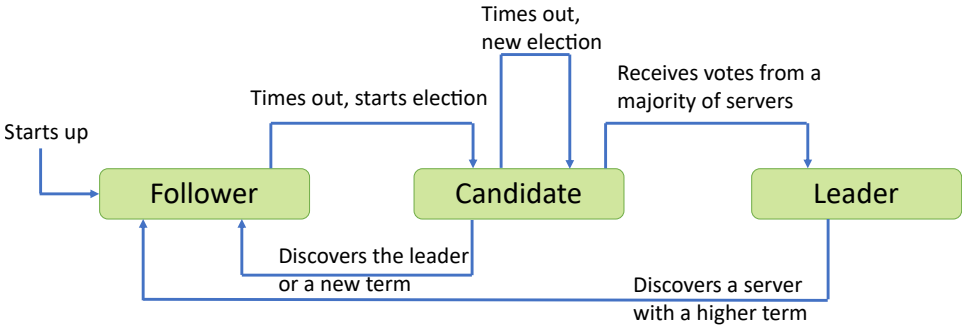


Figure 6.9: Raft state diagram

Let us now take a deeper look at Figure 6.9 and explain all the state transitions. The system starts in the follower state and then leader election starts. This is when an interested follower becomes a candidate. If a candidate discovers a new leader, then it realizes that the election has already finished. It thus relegates itself and becomes a follower. On the other hand, if there is a timeout and no leader is elected, then a new election is started. Once a candidate gets the majority vote, it becomes a leader and remains a leader until the next term. From the leader state, the only transition that is possible is to the follower state. This happens when another server is discovered, which has a higher term. This server could also be the new leader that has replaced the current leader unbeknownst to it.

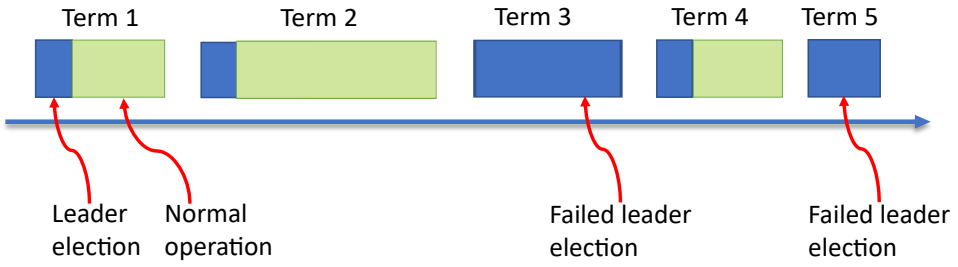


Figure 6.10: Division of time into terms

Figure 6.10 shows how we divide time into *terms*. As we can see, every term is numbered. These numbers are maintained by each server. A term begins with leader election, then there is normal operation of the protocol where log entries are added to the log. Finally, when a term ends, a new term begins. It is possible that the leader election process fails, and no leader is elected. In this case, normal execution cannot happen in that term.

The term number needs to be attached to every message. This is akin to timestamping every message. Using this method we can find messages that belong to old terms or messages that have simply been delayed. If a server with a higher term number receives a message that is stamped with a lower term number, then it drops the message. This should be done because we do not want any server to take cognizance of messages that were sent in previous terms. These are old and outdated messages and thus should not be processed. Assume we have the reverse case – the server receives a message with a higher term number. In this case, it needs to upgrade its term number and set it equal to the higher term number. This is very similar to Lamport clocks.

The term number also plays an important role in the election process. If a candidate or a leader process finds that its term is *stale*, which basically means that it has received messages with higher term numbers, then it relegates itself and becomes a follower once again. We basically want the system to remain up to date at all points of time: this includes normal operation as well as the leader election process.

6.5.4 Leader Election

All the servers start in the follower state. They periodically get ⟨`heartbeat`⟩ messages from the leader that also contain its term. This indicates that the leader is alive and there is no network partition. If a server does not get a ⟨`heartbeat`⟩ message for a certain duration, then it is clear that something is wrong. It times out and begins the process of leader election.

The process of beginning an election is quite simple. The server increments the current term number, transitions to the candidate state, votes for itself and sends a ⟨`RequestVote`⟩ message to the rest of the servers. Note that a server can only vote for one candidate in a term. This is similar to a regular election. Any server process will obviously vote for the candidate that sent it the first ⟨`RequestVote`⟩ message because it does not know how long it needs to wait if it wants to vote for the second ⟨`RequestVote`⟩ message that it receives (if at all). Recall that we had something similar in Paxos. We called it the *First-Accept* condition.

Now, there are three possible outcomes for a server in the election process.

Wins the election: In this case, a server gets the majority of the vote. It clearly knows that it is the leader. It thus goes on to declare itself as the leader and starts sending ⟨`heartbeat`⟩ messages to the rest of the servers. Since we are not assuming Byzantine faults, it is not possible for two servers to declare themselves as leaders in the same term.

Receives an ⟨`AppendEntries`⟩ message from a server: This is a regular message to append entries into the log. If the term in the message is greater than the current term, then it is necessary to transition to the follower state and recognize the server that sent the message as the leader. ⟨`AppendEntries`⟩ messages can only be sent by the leader. No other server can send such messages.

No leader is elected This means that a situation was reached when the server did not get a majority of the votes. Such a situation is known as a *split vote*. There is no option but to attempt the leader election once again with an incremented term number. The backout or quiescence period can be randomized to ensure that conflicts are reduced and at least one candidate has a better chance of winning the election.

6.5.5 Managing the Logs

After a leader has been elected, it makes its presence known to the clients and other servers by sending messages. The entire system knows who is the current leader unless there are network partitions. Clients send requests to the leader to append entries to the log. The leader then sends entries to the rest of the servers using ⟨`AppendEntries`⟩ messages. A server can either drop the message, request for reconciliation of logs or acknowledge the message by sending a response.

Let us now look at the structure of a log. It is a list of entries, where each entry stores a term number and a command. This command is meant to be *applied* to the synchronized state machines. Furthermore, each entry has an index that indicates its position in the log. It is like an array index – a monotonically increasing integer.

An entry is considered *final* or an irremovable part of the log when it is *committed*. An entry is said to be committed when it has been sent to a majority of the servers, and it has been accepted by them. Committing one entry is tantamount to committing all the preceding entries as well

(implications of this discussed later). This is enforced by attaching the highest committed index along with every message. This tells each server about all the indexes that can be committed.

Log Matching

Let us now look at the Log Matching property in some detail. We wish to state two sub-properties.

S1 If two entries in different logs have the same index and term, then they store the same command.

S2 If S1 holds for a given index, then all the preceding entries of the logs are *identical*.

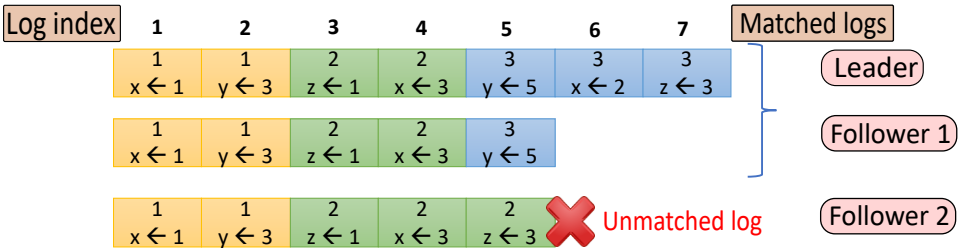


Figure 6.11

This property is ensured as follows (proven later). After the leader creates an entry at a given index and term, it broadcasts it to its followers. To ensure S1 and S2, the leader also appends the index and term of the latest entry in its log (before adding the current entry) with each ⟨`AppendEntries`⟩ message. Each follower needs to check the latest entry in its log. If its term and index match the index and term sent by the leader, then it means that its logs are in sync with the leader. As per the Log Matching property, we expect the rest of the entries to also be identical.

This is shown in Figure 6.11. The logs of the leader and Follower 1 match. However, for the third log at the bottom corresponding to Follower 2, we see an unmatched log entry.

Reconciling Log Entries

Let us now consider the case when there is a mismatch in the index and term. The leader needs to fix the entries in each follower to match its own

because the leader never deletes or overwrites entries. To do this, the leader maintains a *nextIndex* pointer for each follower (see Figure 6.12). This pointer is initialized to one plus the index of the last entry in the leader's log. This means by default it assumes that all the entries match unless proven otherwise.

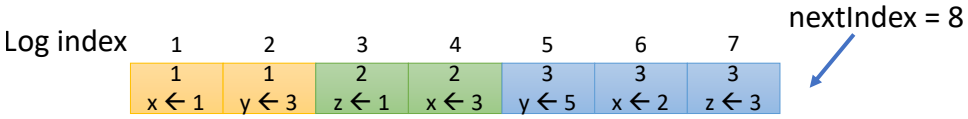


Figure 6.12: A log with a *nextIndex* pointer

For some followers there may be a log mismatch. This can be detected by simply comparing the latest entries of the logs of the leader (prior to adding the new entry) and a follower as discussed earlier. Note that we are throughout assuming the Log Matching property, which says that if two entries at the same index match, then all the entries in earlier (smaller) indexes also match. This will be proven later on.

Now, after receiving a message from the leader, a follow may find that the respective entries at $nextIndex - 1$ do not match. This means that the logs are not consistent up till that point. We need to find the latest point (largest index) at which the logs begin to diverge. The leader continues to decrement *nextPointer* for the follower and continues to try. Ultimately, a point will arise when the logs match. Till this point, both the properties S1 and S2 are satisfied – not beyond it. We thus can conclude that all the entries in the follower's log beyond the point of match are invalid. They are thus removed.

The leader can then send the remaining log entries that succeed the point of match to the follower. The follower needs to simply add these entries to its log. It will end up replicating the leader's log by following this procedure.

It is important to note that it is the follower that is forced to delete/overwrite entries in its log. The leader never overwrites/deletes entries; it only appends.

6.5.6 Dealing with Server Crashes

Leaders will continuously keep changing in this protocol. However, the new leader is bound by the same commandments as the previous leader – it cannot delete or overwrite any entries. Furthermore, they need to preserve

all the committed entries that have possibly been committed by previous leaders.

The Leader Completeness property is enforced as follows. For a candidate to win an election, it needs to ensure that it contains at least all the committed entries (Leader Completeness). The ⟨`RequestVote`⟩ message includes information about the candidate's log and has a count of the number of committed entries. Needless to say it should be at least as up to date as the log of each voter. The check to find if a candidate's log is up-to-date or not is quite simple. There are two conditions.

- Check the last entries of the logs. The log with the entry with the higher term number is more up to date.
- If the terms are the same for the last entries, the log with more entries is more up to date.

Remark 6.5.1

A server does not vote for a candidate unless the candidate's log is at least as up-to-date as its voter.

Now, assume that a leader crashes. Consider an entry e. If the leader server crashes before *committing* e in a majority of servers, then we have a complex situation in our hands. The log of the new leader has to be as up-to-date as a majority of the servers (the voters). Assume that the new leader has e in its log, then there is no problem. It will ultimately commit it and send it to all the servers. However, if it does not have the entry, then the new leader has a choice. It can either proceed to commit or ignore it. It is a much better idea if it chooses to ignore it – after all it is an uncommitted entry from a previous term.

Let us now consider the case when a follower or candidate crashes. Raft continuously tries to send ⟨`AppendEntries`⟩ and ⟨`RequestVote`⟩ messages to all the servers. This includes followers and candidates that may have crashed. Given that all these messages are idempotent (can be sent multiple times), there is no problem if these messages are sent many times in case the feeling is that they are getting lost.

There is however a timing requirement for the protocol to execute reasonably well. The time needed to broadcast a message should be much lower than the election timeout time, which should ideally be much lower than the mean time between failures.

6.5.7 Proof

Given that we have described the algorithm, we need to now prove that the five properties mentioned in Section 6.5.2 hold.

Election Safety

In any term, we can only elect a single leader, or we will have a situation with split votes (no majority). This is because in a given term, a server can vote only once, and we need a simple majority to elect a leader. Hence, two leaders cannot be elected.

Leader Append-Only

By design, the leader does not append or overwrite any entries in its log. This is trivially ensured.

Log Matching

Let us prove that the described algorithm maintains the Log Matching property. We will prove a series of lemmas to reach our final goal.

Lemma 17

If server A has N_A entries in its log for term T and server B has N_B entries in its log for the same term T, and $N_A \geq N_B$, then A contains all the entries that B contains for term T.

Proof: Consider the time period when T was the current term. Recall that whenever the leader sends a new entry, it also sends the index and term of the previous entry.

Consider the first log entry that was added in the term. Either there was a $\langle term, index \rangle$ match or not. If there was a match, then the entry sent by the leader is added to the log. This becomes the first entry in the term. If there is a mismatch, then the leader repairs the logs of the follower till both match till that point (beginning of term T). After that the first entry in term T is added. In both cases, for A and B the same entry is added as the first entry of term T.

Let us now use mathematical induction. Assume that the first k entries are the same (in term T). Consider the $(k+1)^{th}$ entry. It will be sent to both A and B by the leader. Both of them cannot add the entry unless their k^{th} entry in term T matches that of the leader. If there is a mismatch, then

it means that there must be lost messages. In such a case the logs have to be reconciled. After reconciliation the $(k+1)^{th}$ entry will be added. It will still be the same entry. Hence, the first $min(N_A, N_B)$ entries in term T will be the same for A and B because this sequence will match the corresponding sequence of the leader.

We can thus conclude that at the end of term T, if we find that A has more entries than B in term T (i.e., $N_A \geq N_B$), then we can say for sure that A has all the entries that B added. It must be that B missed some messages. ■

Lemma 17 is a restricted version of the Log Matching property. It only discusses matching in a given term. We now need to extend it to hold across terms. This is where we need to consider the way in which we conduct elections. Recall that the log of a candidate needs to be at least as up to date as the log of each voter. This means that its last entry must either have a higher term or a higher index. Let us consider the next lemma.

Lemma 18

If two logs have the same entry at the same index, then the logs match till that index. The entry here comprises the command and the term.

Proof: Consider the first term. This is true given the results of Lemma 17.

Now, consider the second term and the first entry that is added in it. Assume that the election was successful.

Consider how the leader of the second term got elected. It must have had at least as many entries in its log as a majority of the servers. This is a straightforward conclusion from the election restriction. Once it has been elected as a leader, it will bring all the logs into sync with it before they can add the first entry in the second term. If some server has additional entries that the leader does not have, then it will be forced to delete them. Ultimately, for the first term, they will all have the same number of entries (also same entries) before they can add the first entry in the second term.

Now, let us extend this logic to the induction case. Assume that the Log Matching property holds for the first k terms. Consider the $(k+1)^{th}$ term. Here also a server cannot add its first entry unless the logs match with the leader for the first k terms. Only after that the first entry in the $(k+1)^{th}$ term can be added.

The only case that we have not considered up till now is the case when the election fails. This means that no server gets a clear majority. In this case, we pretty much waste a term. We move to the next term and again try to elect a leader until a leader is successfully elected. The wasted terms

don't add messages to the log. The only modification that we need to make to the proof is that if the leader is elected in term k', then we find the latest term k that has a valid leader. The rest of the proof remains exactly the same.

We have thus proven by mathematical induction that the Raft algorithm preserves the Log Matching property. ■

Given that the Log Matching property holds, we need to now prove that if an entry is committed by a leader it remains as a committed entry in the logs of all future leaders. This is, in principle, different from the Log Matching property because it brings in the notion of commitment, which we have hitherto not discussed.

Leader Completeness

This property establishes the finality of commitment. This means that once an entry is committed, it remains present and committed in the logs of all future leaders. It is never forgotten in the system. Let us prove a lemma.

Lemma 19

If a leader commits an entry, it is present in the log of the next leader.

Proof: If an entry e is committed, then a majority of servers have stored it in their logs and also acknowledged the same. Consider the election of the next leader. It needs to have a log for the current term that is at least as long as each of its voters. Given that the intersection of two sets that comprise a majority is non-null, there will be at least one voter that will have entry e in its log. This automatically implies that the same entry e needs to be present in the log of the new leader. Otherwise, it would not have been chosen to be a leader in the first place given the election restriction.

Hence, once an entry is committed in a term, it will be present in the log of the next leader. ■

Now, let us prove that no subsequent leader can delete an entry that has already been committed in a previous term.

Lemma 20

If a leader commits an entry, then it is present in the logs of all subsequent leaders.

Proof: We have already proven in Lemma 19 that if a leader commits an entry, then the next leader has it in its log. We now need to show that no

leader is elected that does not have the entry.

Let us say that the leader of term k (L_k) commits the entry e. Without loss of generality, let us assume that term $k+1$ has a valid leader L_{k+1}. The reason that we can make this assumption is that if no leader is elected, then nothing happens in the term. Let us basically skip all the terms after term k in which no leader is elected. Ultimately, a leader will be elected. There is no harm in considering it to be term $k+1$, from the point of view of correctness in the proof. Now, as per Lemma 19, L_{k+1} will also have entry e in its log. Again without loss of generality let us assume that L_{k+2} is a valid leader elected with a majority vote. The claim is that it will also have entry e in its log. The reason is as follows.

Given that L_{k+1} has e in its log, it will not remove it. Instead, it will ensure that the rest of the servers also add it to their logs if they don't have it already. Recall that a leader never deletes entries from its log. We know that a majority of servers at this point already have e in their logs because it was committed by L_k and not subsequently removed. Of course, this process may not be fully successful because L_{k+1} may crash in the middle.

Now, consider L_{k+2}. If it already has e then there is no issue. Is it possible for it to not have e in its log? This is not possible because it will need a majority vote. There will at least be one server that has e in its log, and thus it will not allow any candidate to get elected without having e. Now, L_{k+1} did not remove e from any log and thus the status quo was maintained. As a result, for the same reason that was used for L_{k+1}, L_{k+2} will also have e in its log.

We can thus prove by induction that all successive leaders will have the entry e in their logs. This means that once an entry is committed, it cannot be removed by any leader. ■

Lemma 20 proves the Leader Completeness property. We thus observe that once an entry is committed, it cannot be removed, and it remains in the logs. We can prove an interesting corollary here.

Corollary 6.5.1

If an entry is committed, then all the previous entries in the log are also implicitly committed.

Proof: Lemma 20 proves that once an entry is committed, it remains in the logs of all leaders and also a majority of servers. Let us now use the Log Matching property. This means that in a majority of the servers including the leader, all the entries preceding any committed entry are identical. Given that a committed entry cannot be removed (as we have just seen), and

the process of removing entries starts from higher indexes and moves to lower indexes, we can happily conclude that no entry that has a lower index that than of a committed entry will ever be removed. It is also present in a majority of servers. This basically means that it is *committed* and has achieved a degree of finality – it cannot be overwritten or removed. ■

State Machine Safety

Given the previous lemmas that prove that Log Matching and Leader Completeness hold, proving state machine safety is quite trivial. It basically says that if a server has applied a committed entry present at a certain index in its log, then no other server will apply any other entry at the same index. This is true because if an entry at a certain index is committed, then all the servers will eventually store the same entry at that index. A majority of the servers already do, which is why the entry is committed in the first place. Even the rest of the servers that are somewhat slow cannot store any other entry at that index. This will violate Lemmas 18 and 20. Hence, a committed entry can be safely applied to a state machine. All other servers will eventually do the same. Given that we are not considering Byzantine failures, the servers cannot do anything else.

6.5.8 Miscellaneous Issues

Cluster Membership Changes

In a practical system, servers can be added and deleted. This means that the Raft cluster membership has the potential to continuously change. A naive solution is to stall the system, change the system configuration and then restart the system. If a new server is added, then there is a need to copy the logs of the leader to the new servers that are added.

The advantage of Raft is that it does not suffer from any downtime. There is no need to pause or stall the system. It continues to work even when servers are added or removed from the system. The leader gets a request to change the configuration of the system. Let the old configuration be C_{old}, and let the new configuration be C_{new}. This mechanism also allows us to join two Raft clusters.

Several problems can happen. While a larger cluster is being created, it is possible for two leaders to be elected. This is because the entire cluster has not joined yet to form one cohesive unit. All the servers may not even be aware how large the new cluster is, and they may not know what the majority mark is.

Figure 6.13: A joint consensus mechanism

A joint consensus mechanism is used by Raft to solve all such problems that arise when a new cluster is forming. Temporarily, a short-lived joint configuration is created called $C_{old,new}$. There is a need to replicate all entries to all the servers that belong to both the configurations. This is done as follows.

In the joint consensus mechanism, there is a need to elect or choose a leader. The leader of any configuration – C_{old} or C_{new} – may work as the overall leader. For any kind of majority-based agreement for elections and AppendEntry operations, there is a need to get votes from a majority of servers from both the old and new configurations, C_{old} and C_{new}, respectively.

The joint consensus mechanism is shown in Figure 6.13. Note that the joint consensus state is a valid state. At this point, client requests can be added to the log and can be serviced. Let us discuss the details.

Joint Consensus Mechanism

The leader of configuration C_{old} receives a request to convert it to C_{new}. It creates a new temporary configuration called $C_{old,new}$. This information is sent to all the servers as other regular messages such as heartbeat messages. At this point, the logs are copied from existing servers to new servers. Next, the entry $C_{old,new}$ is added to the logs of each server. Once this entry is committed, this becomes the current configuration. During this period if there is a leader crash, the choice of a new leader is dependent on which configuration is used. It depends on the configuration that the candidate belongs to: C_{old} or $C_{old,new}$.

Once, the joint consensus has been set up, and the entry $C_{old,new}$ has been committed, it is time to elect a leader for the combined configuration. All the servers that have $C_{old,new}$ in their logs are eligible to vote. The temporary leader of the configuration $C_{old,new}$ then initiates the procedure to add and commit a new entry C_{new} that corresponds to the new configuration. Once this entry is committed and is sent to all the servers, the new configuration fully takes effect.

Log Compaction

Logs keep growing over time. It is important to ensure that the logs do not grow indefinitely. When a log reaches a predetermined size, we can take a *snapshot*. We can use the Log Matching property here. We consider an old index, record it and the store the rest of the logs that precede it in stable storage. Subsequently, we can discard the logs that were stored in the rest of the servers. No information is lost. The logs match anyway. Hence, storing one copy is just enough.

Sometimes the leader can transfer its snapshot to followers that are far behind. This fast-forwards them and brings their logs in sync with the leader.

Client Interaction

Raft provides linearizable consistency, which means that every operation appears to complete at a unique point between its start and end (more details in Section 7.1.2). Moreover, it is possible to quickly realize read operations. There is no need to create a new log entry. The only restriction here is that only committed values can be returned.

6.6 Summary and Further Reading

6.6.1 Summary

Summary 6.6.1

1. The consensus problem is a fundamental problem in distributed systems. There are multiple processes – each proposes a value. One out of those proposed values is ultimately chosen and accepted by all. All the processes choose a single value.

2. A large number of problems in distributed systems can be reduced to solving a consensus problem. Hence, this is a problem of fundamental importance.

3. The Fischer, Lynch and Paterson (FLP) result proves that in any asynchronous system with at least one faulty process, it is not possible to design an algorithm that reaches a consensus state all the time. This result holds for situations, where it is

not possible to distinguish between a dead (crashed) and a slow process.

4. If we can guarantee that no process will suffer from a failure after the execution has started, then it is possible to design an algorithm that achieves consensus all the time. There could be processes that are dead to begin with (initially dead); however, the caveat is that no process is allowed to fail after the consensus algorithm has started.

5. In the case of synchronous systems, it is possible to achieve agreement in spite of faulty processes. The Byzantine General's problem has one distinguished process known as the commander and a set of other processes (lieutenant generals) that need to mutually agree on the value that the commander has sent. In practice, this problem is quite challenging because the commander himself could be traitorous. This problem is typically solved in exponential time using a recursive construction.

6. It is not possible to achieve Byzantine agreement if the number of disloyal generals is equal to or exceeds a third of the total number of generals.

7. If the generals are allowed to use signed messages where the signature of a loyal general cannot be forged, then it is possible to achieve consensus regardless of the number of disloyal generals.

8. The Practical Byzantine Fault-Tolerant Algorithm (PBFT) uses digitally signed messages to achieve Byzantine fault tolerance. It replies on several phases of messages and cryptographic mechanisms to achieve consistency. Given that the identity of any server cannot be forged because of its unique and verifiable digital signature, it is easy to achieve agreement using $O(n^2)$ messages. The real complexity lies in accommodating a view change when the primary server fails.

9. The Paxos algorithm is one of the simplest algorithms for achieving consensus in an asynchronous setting that does not assume Byzantine faults. It has proposers, acceptors and learners. It works in two phases. The first phase tries to propagate

the proposed value to the rest of the nodes. However, if a decision has already been made, then the value does not propagate any further. Ultimately, the value that is sent to the later phases of the protocol is the consensus value. This prevents any other value from being decided (finally chosen) after a consensus is arrived at.

10. The Raft consensus algorithm is a far simpler consensus algorithm and is in the same class as Paxos. It divides time into non-overlapping terms. Each term has a leader whose job is to decide the consensus value. There are elaborate mechanisms to elect a new leader in case a leader goes down, and propagate this information to the rest of the follower nodes.

6.6.2 Further Reading

Out of all the consensus protocols, Paxos and its variants stand out. It provides the strongest possible safety guarantees in an asynchronous environment. Sadly, the generalized version of the Paxos protocol is quite complicated. Hence, over the years several simpler variants have been proposed. Raft is one such variant. Another important variant is EPaxos [Moraru et al., 2013]. It requires a simple majority for a consensus decision and it requires at most two communication rounds. Up till now we have discussed consensus protocols that require all the nodes to participate. The Stellar family of consensus protocols [García-Pérez and Schett, 2019,Mazieres, 2015] require only a subset of nodes to participate in the protocol. As long as a designated set of nodes (referred to as the *quorum*) agrees on the consensus value, a decision can be made.

Off late, different versions of Byzantine fault tolerant protocols like PBFT have become quite popular. We have covered PBFT in this chapter. Some of the other variants are HotStuff [Yin et al., 2019] and SBFT [Gueta et al., 2019]. These are highly responsive and scalable protocols that are near-linear time in the common case. Consensus protocols have a special place in the world of cryptocurrencies and blockchains. They form the bedrock of blockchain technology. The Algorand [Gilad et al., 2017] and Streamlet [Chan and Shi, 2020] consensus protocols are simple protocols that can be used to construct a blockchain, which is nothing but a linked list of elements that all the processes agree on. We shall touch upon this topic in detail in the chapter on blockchains. It is also possible to create

a DAG of transactions, where the aim is to immediately buffer them such that they are not lost. An exact ordering of transactions can be created later [Danezis et al., 2022].

There are a large number of consensus protocols. The paper by Aublin et al. [Aublin et al., 2015] proposes a standard framework for generating Byzantine Fault-Tolerant Protocols (BFT). It is possible to tradeoff consistency requirements (introduced in Chapter 7) with the responsiveness of the protocol (refer to Zeno [Singh et al., 2009]).

Consensus protocols are fairly complex, as of today, and have numerous constraints. There is thus a need for formal verification. IronFleet [Hawblitzel et al., 2015] and the paper by Chand et al. [Chand et al., 2016] are important references in this area.

Questions

Question 6.1. Answer the following questions regarding the Paxos algorithm.

1. How can the Paxos algorithm enter a livelock?
2. Why do we need proposers?
3. What is a learner? How does it learn the consensus value?

Question 6.2. Assume that all the nodes in a distributed system wish to execute the same sequence of steps as a part of a state machine. For example, it is possible that Node 1 might propose to add two numbers as the third step, and Node 2 might propose to subtract the same pair of numbers as the third step. However, both the nodes need to carry out the same operation as a part of their third steps. How can the Paxos algorithm be used here?

Question 6.3. Does the Paxos algorithm conflict with the FLP result? Why or why not?

Question 6.4. We wish to prevent read-write conflicts and write-write conflicts in a quorum based system. If there are two concurrent writes, then the system needs to order one write before the other. Assume that an ordered multicast primitive is available across the servers. Assume there are N servers. While reading, we wish to read from N_R severs, and while writing, we wish to write to N_W servers. What is the relationship between N_R, N_W and N? (Note that $N_R = N_W = N$ is a trivial solution, and is not allowed

Question 6.5. Prove the following in the context of Byzantine fault tolerance.

(a) Prove the correctness of the exponential time Byzantine fault tolerant protocol as described in class.

(b) How does it not violate the impossibility result of Nancy-Lynch-Patterson?

Chapter 7

Consistency

> *Consistency is the true foundation of trust.*
> *Either keep your promises or do not make them.*

Roy T. Bennett

Every distributed system has a large number of processes. They either read the state or make changes to the global, distributed state. We can thus classify operations into two broad types: read updates that do not make state changes and write updates that change the state. Hence, it makes sense to refer to operations as reads or writes. Let us consider the behavior of such read and write operations in the context of distributed systems. A large distributed system is supposed to be visible as a single machine to the external world. Furthermore, the latency should be low, the throughput should be high and there should be strict specifications regarding the values that are returned by these operations.

If a distributed system has thousands of machines, it is not possible to easily provide all these guarantees. Most such systems maintain copies of data or replicas at all the machines (also known as servers). Any write operation needs to update at least all the replicas, or at least most of the replicas such that the updated data is available to future read operations. Similarly, read operations should ideally return the latest value that has been written. However, this may not be possible given that updates may not have propagated to all the replicas. Hence, there is a need for creating sophisticated protocols that ensure that reads get the latest value, which might not always be practical. In such a case, there is a need to relax the requirements and fetch a recent update, need not be the latest one.

The behavior of reads and writes can be characterized by their outcomes. This behavior needs to adhere to specifications, which is known as the *consistency model*. Any execution in a distributed system is said to be consistent with these specifications. There can be strict consistency models, which make the distributed system appear to behave more like a sequential system. However, they typically have a high latency because of a lot of messages need to be exchanged between the processes to ensure this behavior. Hence, if a lower latency is desired, it is a wise idea to sacrifice strict consistency and adopt a more relaxed form of consistency.

Next, if we consider faults, then there is a trade-off between a system's responsiveness and consistency. Let us say we want a system to always be available, which in means that it is responsive regardless of the faults that its constituent processes suffer. This is fine from an external perspective. However, in such a system it is not possible to provide strict consistency guarantees. For example, if we want every read to fetch an updated value, then it may not be possible. The update should have propagated to the server that is servicing the read. It is easier to ensure this in a fault-free environment. The classic CAP theorem and the subsequent PACELC theorem capture such constraints and trade-offs.

Whenever there are replicas, the issue of ensuring consistency will always arise. The CAP theorem establishes a trade-off between consistency, availability and partition tolerance. Different systems can operated at different points in the space of trade-offs. Apache ZooKeeper is a very popular data management service that makes it easy to implement a wide variety of primitives. It provides strict consistency guarantees for writes and relaxes the consistency requirements for reads. In general, reads are on the critical path and it is necessary to reduce their latency. The PACELC theorem says that there is a trade-off between latency and consistency if there are no faults in the system. The designers of Apache ZooKeeper chose strict consistency for writes and relaxed consistency for reads.

All such protocols require updates to be broadcast to all the servers. This is not a very scalable approach. Modern approaches center around multicasting updates to a subset of processes. Such a subset is known as a quorum. Quorum-based approaches require operations to be sent to all the members of the quorum. This is often enough to ensure correctness. For example, consider a consensus problem where the aim is to ensure that all the processes in a quorum agree on the same value. If the quorum intersection property holds, i.e., any two quorums intersect at a non-faulty node, then it can be guaranteed that the system will never enter a state where two different values are decided. It is possible to extend this simple idea to consider

quorums that are dynamically decided or those that guarantee consistency on a probabilistic basis.

Organization of this Chapter

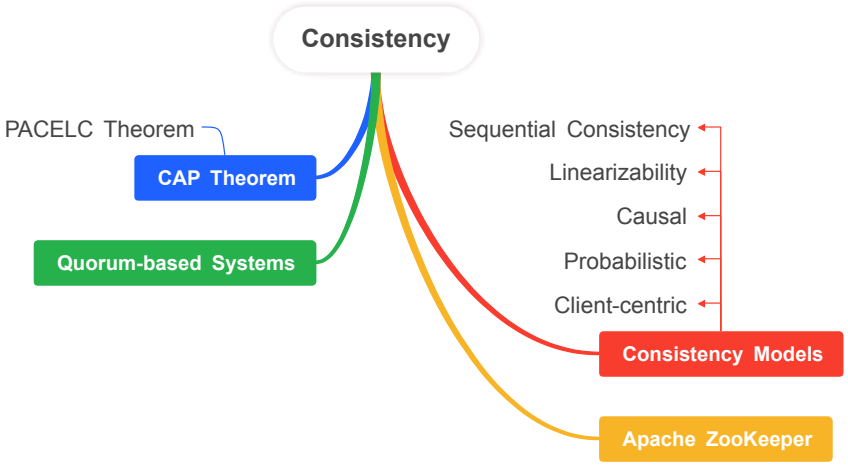


Figure 7.1: Organization of this chapter

Figure 7.1 shows the organization of this chapter. We will start with discussing different consistency models. In the field of distributed systems, linearizability and sequential consistency are the gold standards. They are close to a purely sequential system. Often it is impractical to implement them in practice. This is why more relaxed consistency models are used in practice such as causal consistency and eventual consistency. These are all system-level consistency models. They take a system-level view and look at the execution of all the operations in the system. In comparison, there are consistency models that specify the behavior only from the point of view of an individual client machine.

After covering consistency models, we shall look at the classical CAP and PACELC theorems. They explore the trade-offs between consistency, availability, partition tolerance and latency. It is not possible to guarantee everything and trade-offs need to be made.

We shall find uses for such theoretical results in the next section, which discusses the popular Apache ZooKeeper system. It adopts different kinds of

consistency models for reads and writes. Reads are often on the critical path and thus their latency should be low. On the other hand, the write order is important for correctness. Consequently, in this case, strict consistency is important.

In the last section, we shall focus on making the process of creating distributed data stores that respect a given consistency model more efficient. It is not necessary to broadcast writes to all the servers. There are protocols that multicast writes to a subset of servers known as the quorum, which we shall in detail in this section. Quorums can be considered in many different ways, they can be dynamic and can also provide probabilistic guarantees. We will study all these variants in this section.

7.1 Consistency Models in Distributed Systems

Recall the definition of a distributed system – a set of loosely coupled machines that appear to be "one" machine. Here "loosely coupled" means that the machines do not share memory and are connected over the network. This simple definition now needs to be further nuanced.

Let us consider a large distributed system that hosts the website of a newspaper. Assume that a user comments on an article, and then another user responds to the comment. It is possible that the original comment and its response initially go to different distributed nodes. Now, when a third user tries to read the post, she should be able to see the comment and its response as well. This will involve sending a series of messages to fetch all the comments and responses for a post. Recall the discussion that we had while introducing the FLP result. It is possible that a node appears to be slow because it is overloaded, the network is congested, or the node has simply crashed. Thus, it may be possible that the distributed application fetches the post and the response, but not the original comment that led to the response. This will appear to be a very non-intuitive situation, however in any practical implementation it is indeed possible unless additional measures are deployed. It is important to note that these "additional measures" will involve sending additional messages, which will slow down the system yet provide the vital correctness property: *If a response is visible, then the original post and comment should also be visible.*

Let us now look at a situation in which such behavior may be acceptable. Assume that we have created a large distributed system to conduct an opinion poll. An opinion poll is typically a phone-based survey where volunteers ask the public at large about their opinions on a certain issue.

Given that the call volume will be pretty large, we can design a distributed system where different sets of volunteers will log their data in different physical servers. Periodically, the servers will exchange their contents and remain up-to-date. Let us assume that there is a server failure. If its data has not been backed up, it may get lost either temporarily or permanently. Now, if the final outcome of this survey was just a count of the number of people who are for or against a certain issue, then these counts will get distorted. This can make the data inaccurate because when all the counts are collected at the end of the polling, some counts may be missing. However, given the approximate nature of the task, often there is a margin of error. It is $\pm 3\%$ in most cases, which basically means that the agency conducting the polling announces a priori that the counts will have an error that will not exceed $\pm 3\%$. In such a situation, if one or two servers crash and the error remains within this bound, the situation is acceptable. Even though there is a small error, it is within bounds (as defined earlier) and there is no need for an expensive protocol to ensure that data is very frequently backed up. Given that we have accepted a certain margin of error, our data backup and exchange protocol can be tailored accordingly. We have an opportunity to minimize the number of messages here.

We thus see that the same situation that is not acceptable in one case, may be acceptable in another case. There is thus no global definition of the correctness of a distributed system. It all depends upon the *context*. In the first case, the context needed a high degree of preciseness. It enforced a causal relationship between the original post, the comment and the response. It is not possible to see the response without seeing the original comment. In the latter case, the error can be defined in terms of the deviation of the count that is visible at the end of the opinion poll and the "real count". We need to limit it to $\pm 3\%$. We could define many more contexts, where the loss of messages has a minimal effect. For example, if we are doing e-shopping, and we are adding a lot of items to the shopping cart, then even if a few items get dropped, it is not a very serious issue. This can always be detected at the time of checking out and the items that were dropped because the servers storing them crashed, can always be added back. This situation is even more tolerant to message loss if we are willing to bear the brunt of the user's anger when she realizes that she is checking out but a few of the items that she added to her cart are not there anymore.

Hence, if we were to ask, "Is there a notion of right and wrong in a distributed system?" The answer is NO. It is all context dependent. It is much better to define a set of specifications. For example an error margin of $\pm 3\%$ is a specification, not missing out on at least half the items added to

the shopping cart is one more example of a specification. The only concrete decision that can be made is whether a given execution is compliant or consistent with a specification or not. This admits a concrete Yes/No answer and can be used to decide whether a distributed system does its job or not. Note that instead of an absolute notion of correctness, we have adopted a more nuanced notion, where there is a specification, and the only thing that matters is whether a given execution of a distributed system adheres to the specification or not. An execution is defined as basically a run of the distributed program – it includes all the outputs produced by the distributed system including any intermediate results of interest.

Definition 7.1.1 Consistency

In distributed systems, it is hard to define an absolute criterion for correctness. Instead, we check if a distributed system's execution is *consistent* with specifications. This property is referred to as *consistency*.

There are many different kinds of consistency models corresponding to different kinds of specifications. We can categorize them into different types.

Inspired from Sequential Programs: We intuitively understand sequential programs. There are a large number of consistency models that are inspired from establishing an equivalence with sequential programs to different extents. Distributed programs can be shown to behave like sequential programs under specific conditions.

Dependence-based Models: In another class of consistency models, we do not care about the execution of the full distributed system (program). Instead, we care about the order of execution of specific events that have dependencies between them. They thus cover the execution *partially*.

Numerical and Probabilistic Criteria: This criteria of models are limited to executions that produce a numerical outcome. The specifications specify a certain maximum error or some kind of error distribution.

Client-Centric Criteria: In this case, we look at the outcomes of a distributed system from the point of view of a client (user of the distributed system). The client looks at the updates that she has made and the responses that she gets. Any specification in this space, constrains the responses that can be obtained for a given set of inputs.

Along with these we have many more hybrid models such as data-centric consistency where different sets of data are associated with different consistency models. We shall look at such hybrid models later. Let us look at the basic models first.

7.1.1 Sequential Consistency

Multithreaded Programs

Let us consider the most basic distributed object – a register. A *register* [1] is the same as a variable in regular sequential programs. We can write to the register and read from a register. Consider a regular single-threaded C program and a variable named `x`. We can either write to `x` or read its value. The semantics is straightforward. A read retrieves the *latest* value that is written. A write overwrites the value that is already present, and can optionally return an acknowledgement. Now, if we consider multiple threads, then the same set of operations can be done on a shared variable. Note that some extra complexities are introduced here because different cores on a multicore machine may store different physical copies (replicas) of `x`. However, logically `x` needs to appear to be a single variable that is possibly shared across cores. Any write needs to update all or most of the replicas, whereas a read operation typically accesses only one replica – the one that is closest to the core issuing the operation. Such a shared variable that appears logically to be a single variable is a *register* in a concurrent program.

Different threads can access `x` and perform read/write operations. Of course, in this case, things become more complicated because `x` now represents a variable that is defined logically (or virtually). In reality, it corresponds to many physical locations across cores. There is a high-level access protocol that keeps the replicas in sync. It makes the register appear to be a single variable that is stored at a single location. However, this is just an abstraction.

Because of many pipeline-level optimizations, it is possible that different cores in a multicore machine may not agree on the order of reads and writes. This is because messages to either read the nearest replica or update all the replicas need to traverse the on-chip network; they can get reordered or get significantly delayed. Hence, defining the notion of consistency in multithreaded programs is tricky. If we use the same notion that every read

[1] The term *register* should not be confused with CPU registers. That is a different concept.

retrieves the value of the latest write, then it will not make a lot of sense because in a concurrent system, we are not sure what is the "latest write". Multiple write operations may have been issued at the same time. Given this overlap, it is hard to define a *total order* across writes to the same location.

Let us define some terminology first before proceeding with examples. Let `Rx3` indicate that register `x` was read, and its value was found to be 3. Similarly, let `Wx4` indicate that `x` was set to 4 by a write operation. Let us further assume that all temporary registers (internal to a node) start with 't' and all registers that are globally visible (such as `x`) are initialized to 0.

Let us explain this space of complexities with examples. Consider the code snippet shown below. Assume that x and y are shared global variables. They are initialized to 0. The question is whether the following two reads are possible: $Ry0$ and $Rx0$. Intuitively, it may appear that this is not possible.

Thread 1	Thread 2
Wx1	Wy1
Ry0	Rx0

This is because we are setting $x = 1$ and $y = 1$ first. We expect at least one of the reads to return a non-zero value. This counter-intuitive outcome can be justified as follows. When x is set to 1, the corresponding write is sent to the memory system. In modern systems, cores have write buffers that may temporarily buffer the writes to reduce the load on the caches. In such a case, the write remains confined to Core 1 (running Thread 1). It is not immediately visible to Thread 2 (assumed to run on Core 2). As a result Thread 2 will return $Rx0$. Similarly, Core 1 (running Thread 1) may miss the update to y before reading it. The reason is again the write buffer on Core 2. Hence, both the cores may read x and y to be zero (resp.). The conclusion here is that a seemingly trivial execution hides a lot of complexities when exported to a concurrent setting. Specifically, we are violating program order, which means that the instructions are not *appearing* to execute in the order that they appear in the program. It appears that $Rx0$ and $Ry1$ are executing before their previous instructions. This is an example of program order violation, which is sadly visible on most modern multicore processors as of today. This is allowed for performance reasons.

Let us consider one more example.

Thread 1	Thread 2
Wx1	Rx2
Wx2	Rx1

Can we get the outcome $Rx2$ followed by $Rx1$? Again, this appears to be non-intuitive. Thread 1 perceives the order of writes to x as follows: $x \leftarrow 1$ and then $x \leftarrow 2$. If the outcome is $\langle 2, 1 \rangle$, Thread 2 perceives the writes to be in the reverse order. In general, all multicore systems will prevent this outcome because it breaks the notion of a shared memory location. The cache coherence protocol will prevent such outcomes.

An astute reader will ask, "How come the earlier outcome with write buffers is allowed, but this outcome is not allowed?" Before we answer this question, we need to ask ourselves what we expect from a register in a concurrent system. The accepted standard in multicore processors is that all the processes (or threads) should agree on the order of writes to the register. This means that if x was first set to 1, then to 2, and so on, all the participating concurrent entities should agree on this order. Nobody can report a different order. This is known as the Write-Order axiom. This means that for any given register, the writes to it are *totally ordered*.

In the earlier example, we had multiple locations and the outputs in question were $Rx0$ and $Ry0$. However, on closer inspection they do not violate the Write-Order axiom. x was first 0, and then it was set to 1. We just happened to read x before it was set to 1. The same holds for y. Hence, this situation is acceptable because the outcome does not break the notion of a register in a distributed system.

However, in the next example, the output sequence $Rx2 \rightarrow Rx1$ is incorrect because it violates the Write-Order axiom. Both the threads are supposed to see x first getting set to 1 and then to 2.

We are not in a situation to fully understand the sanctity of the Write-Order axiom. Once we have discussed sequential consistency and linearizability, we will be in a better position to appreciate its significance. Nevertheless, let us bear in mind that on multicore systems this axiom is sacrosanct and threads are not allowed to violate it. Otherwise, the notion of a register breaks down.

Definition 7.1.2 Write-Order Axiom

Write-Order Axiom: The writes to a register in a distributed system are *totally ordered*.

Distributed Systems

Now, let us generalize this concept to distributed systems. We can pretty much do the same. We can define a distributed variable, which is known

as a register. It is logically a single shared variable. However, it may be stored in different physical locations across the nodes in a distributed system. Primarily, two operations are supported: read and write. Reading involves reading one or more replicas and returning a value. The result of the read operation is a function of all the values that are read. Similarly, a write operation needs to ideally update all the replicas. Sometimes this rule is relaxed and only a majority of replicas are updated. The outcome of the write operation is typically *void* or a Boolean value, in case there is a possibility that the write operation may not be successful.

The Non-Intuitive Nature of Outcomes in a Distributed System

Let us revisit our examples. Consider the following execution.

Process 1	Process 2
`Wx1`	`Wy1`
`Ry0`	`Rx0`

In this case, we write 1 to two registers, `x` and `y`, respectively. Then, when each process tries to read the value of the other register, it reads 0. This seems non-intuitive because somehow it breaks some notion of correctness, as we had argued earlier. We might not be able to articulate what exactly seems broken at the moment, but something does not look right. One may want to argue as follows. One of these two operations, `Wx1` and `Wy1`, must have happened earlier. The read operation associated with the other thread should have seen the write and returned 1. This argument however has flaws when we look at it in the context of distributed systems.

Both the writes are effected by sending messages to the nodes that contain the replicas corresponding to the variables `x` and `y`, respectively. These messages can get *delayed* by the network. In this case, we cannot blame the processes if they return 0 for both the reads. Recall that all variables are initialized to 0. This means that in a practical situation, this execution is indeed possible. However, we somehow do not like it because if one of the writes happens first, the corresponding read (to the same variable) should return the value that was written (as per our understanding). This is not happening because the reads are appearing to overtake the writes. They are not appearing to execute in program order – the order of operations that will be observed by a hypothetical machine that executes one operation at a time, waits for it to complete, and then issues the next operation.

Let us consider the other example that is equally vexing in a distributed setting.

Process 1	Process 2
`Wx1`	`Rx1`
`Wx2`	`Rx2`

In this case, we write to the same variable `x`. However, there seems to be a problem. The order of reads does not match the order of writes. We write 1 first and then 2. However, we read them in the reverse order. This means that processes 1 and 2 somehow see the writes to be happening in different orders. This again can happen quite easily in practice. It is possible that the message to write the value 1 is delayed, whereas the second message (to write 2) reaches the replicas sooner. In this case, the replicas get updated in the reverse order. Unless we have additional safeguards such as waiting for an acknowledgement of the first write (from a majority of the replicas) before proceeding with the second write, this situation can indeed happen. Here there is a clear violation of program order. Note that the same outcome was disallowed in the case of multicore processors, however, it may be allowed in distributed systems depending on the use case. We can conclude that in different situations different rules prevail. Something strictly disallowed in one context may be allowed or acceptable in a different context.

Let us now look at yet another example that highlights a different kind of problem.

Process 1	Process 2	Process 3
`Wx1`	`while` (`x` $\neq 1$) {}	`while` (`y` $\neq 1$) {}
	`Wy1`	`Rx0`

In this case, Process 2 waits for `x` to be 1. Once, it observes the write, it proceeds to set `y` to 1. Process 3 waits for `y` to be set to 1. It then reads `x`. We would expect some causality here. `x` was set to 1 $\rightarrow$ it was read to be 1 $\rightarrow$ then `y` was set to 1 $\rightarrow$ `y` was read to be 1 $\rightarrow$ `x` was then read (by Process 2). Common sense would suggest that Process 2 should read `x` to be 1. However, that was sadly not the case. The reason can actually be quite simple. The message that was sent to update `x` reached Process 2 (and other nodes that Process 2 must have consulted for the value of `x`) but did not reach Process 3 and its associated nodes.

Here, there is no violation of program order. But we are violating another condition called *atomicity*, which means that an operation appears to execute instantaneously. The write to variable `x` is not happening atomically. It is visible to Process 2 but not to Process 3. This basically means that it is appearing to execute at different points of time to different processes, and the execution is non-atomic.

Definition 7.1.3 Atomicity

An operation is said to be *atomic* if it appears to execute instantaneously. Its partially executed state is never visible to any process.

From these three examples, it should already be quite clear that both reading and writing are complicated processes and are heavily influenced by the relative order and delays incurred by messages. Furthermore, what is allowed and what is not allowed is dependent upon the context. Such wild environments often need to be disciplined by constraining the set of outcomes for a given input. This is where the role of a *specification* comes in. It precisely defines the set of possible outputs for a given distributed program and inputs. Note that there can be multiple outputs given the nondeterministic nature of message transmission and arrival. Given that different distributed nodes can get delayed for numerous reasons such as intermittent crashes and context switches, we typically assume that the delay between any two consecutive instructions/operations issued by a process can have an indefinite delay between them. For example, there can be a context switch in the middle, and the program may be restarted a long time later. Even here also, the output should be as per specifications or alternatively the *consistency model*.

Definition 7.1.4 Consistency model

A consistency model specifies the acceptable outputs of a distributed system given a distributed program and a set of inputs. It deems certain outputs to be "inconsistent" with specifications.

Definition of Sequential Consistency

Let us try to understand what exactly we find non-intuitive in these examples. We are used to thinking *sequentially*. This means that we expect that one statement executes, then the next one (after it in program order), so on and so forth. In a certain sense program order is a part of thinking sequentially. In the example shown below, clearly the order in which the statements are executing is not the order in which they are specified in the program – the execution is not in program order.

Process 1	Process 2
`Wx1`	`Wy1`
`Ry0`	`Rx0`

Let us consider all possible executions where program order is preserved – the instructions in each process execute one after the other (as they are specified in the program). One of the instructions `Wx1` or `Wy1` needs to execute first. Without loss of generality, let us assume that the instruction `Wx1` executes first. Let us consider all possible execution orders where we are not allowed to violate program order.

- `Wx1` $\rightarrow$ `Wy1` $\rightarrow$ `Ry1` $\rightarrow$ `Rx1`
- `Wx1` $\rightarrow$ `Wy1` $\rightarrow$ `Rx1` $\rightarrow$ `Ry1`
- `Wx1` $\rightarrow$ `Ry0` $\rightarrow$ `Wy1` $\rightarrow$ `Rx1`
- `Wx1` $\rightarrow$ `Rx1` $\rightarrow$ `Wy1` $\rightarrow$ `Ry1`
- `Wx1` $\rightarrow$ `Rx1` $\rightarrow$ `Ry0` $\rightarrow$ `Wy1`
- `Wx1` $\rightarrow$ `Ry0` $\rightarrow$ `Rx1` $\rightarrow$ `Wy1`

Regardless of the particular execution order, we observe that `x` has to be read as 1. We cannot read `x` to be 0. Similarly, if `Wy1` executes first, then `y` has to be read as 1. Hence, the outcome `x` $=$ `y` $=$ `0` is not permissible.

We would ideally like to have program order such that the executions are intuitive (such as this case). Moreover, we would not like to have situations where one process can see a write operation, but another process cannot. This means that we also need atomicity. An astute reader will observe that if program order and atomicity are guaranteed, then all the examples that we have been seeing up till now will yield intuitive outcomes where all instructions appear to execute instantaneously (atomicity) and instructions within a process do not "appear" to be reordered.

This consistency condition is known as *sequential consistency*. We assume that all read and write operations execute atomically – appear to execute instantaneously. Second, program order is not violated. Sequential consistency is thus a combination of two properties: atomic execution and per-process program order. This is its *specification*.

Definition 7.1.5 Sequential Consistency

Sequential consistency comprises two properties: atomicity and program order. Atomicity means that every operation appears to execute instantaneously. No intermediate state is observed, and once an operation completes from the point of view of one process, it completes from the point of view of all processes. Second, all the operations issued by a process complete in *program order*. This means that any given external observer will observe all the operations of a process to take effect in program order – the order in which they are specified in the program run by the process.

Let us now look at the definition another way. Consider a valid sequentially consistent execution.

Process 1	Process 2
`Wx1`	`Wy1`
`Ry0`	`Rx1`

Given that all the operations appear to execute atomically (at a single instant), we can arrange all the operations (of all the processes) in a single linear sequence. In this sequence, if we extract all the operations issued by a single process, there should be no violation of the program order. Otherwise, operations issued by a single process will appear to overtake each other, which in this case is not allowed. Furthermore, this sequence needs to be *legal*, which means that every read needs to fetch the value of the latest write to the same register. This is a common sense condition for correctness.

Let us apply this learning to the aforementioned example. The equivalent linear sequence is `Wx1` $\rightarrow$ `Ry0` $\rightarrow$ `Wy1` $\rightarrow$ `Rx1`.

We can view sequential consistency as an execution model, where a single hypothetical process simulates all the actual processes. It picks the next operation from a process, fully executes it, then picks the next outstanding operation from another process, executes it to completion, so on and so forth. Given that this hypothetical process waits for an operation to complete before initiating the next one, the execution is clearly atomic. Second, given that no operation is skipped, program order is maintained. Hence, the existence of such a sequence acts as a proof of sequential consistency. In fact, the name "sequential consistency" arises due to the existence of such a hypothetical process that executes all the operations in *sequence*.

Let us thus summarize our learning and propose an alternative definition of sequential consistency.

Definition 7.1.6 Alternative Definition of Sequential Consistency

If an execution is sequentially consistent, it is possible to interleave the operations of all the processes, and create a single sequence that contains the operations of all the processes. This sequence needs to have the following properties.

- It needs to have all the operations of all the processes issued in a given time interval. No operation can be omitted, and no extra operation can be added.
- It needs to be a *legal* sequence – every read needs to fetch the value of the latest write (to the same register).
- All the operations issued by the same process need to appear in this sequence in program order.

If it is possible to create such a sequence, the execution is said to be sequentially consistent. The converse is also true.

7.1.2 Linearizability

Motivation

Sequential consistency is considered to be the gold standard in multiprocessor systems, especially hardware systems. It is however considered impractical because it enforces too many orders. This prohibits pipeline and cache-level optimizations. Let us now come to general concurrent systems. Here also sequential consistency has a special place ever since it was proposed in the late 70s. Here also, it is regarded as an impractical model for two reasons. The first reason is the same as traditional hardware systems – there are too many constraints and orders. Enforcing all these orders is difficult in a large distributed system. However, the second constraint is interesting. There are cases when we want stricter consistency especially adherence to absolute time, which sequential consistency completely ignores. Let us consider the following example, which seems to be trivial, but has a hidden message.

Process 1	Process 2
`Wx1`	`Rx1`

Process 1 writes 1 to x and then Process 2 reads x and obtains the value 1. There is a simple ordering here: $Wx1 \rightarrow Rx1$. This is a sequentially consistent execution. Given that we do not record the actual times at which these events occurred, it is possible that $Wx1$ was initiated 2 years after $Rx1$. The outcome will still be a valid sequentially consistent outcome; however, it would make little sense. We can clearly see that it is not possible. This is thus not a desirable model for a concurrent system, where the actions are more clearly visible to the user and thus a certain alignment with the real world is required. Given such considerations, the notion of *linearizability* was born. It is essentially sequential consistency plus real-world constraints and create a new consistency condition. There are many ways of defining this new condition – linearizability.

Definition of Linearizability

Definition 7.1.7 Linearizability

Every operation has a start time and an end time. It appears to instantaneously complete at a point of time that is between its start and end.

Definition 7.1.7 states a very concise and compact definition of linearizability. It mandates that every operation needs to appear to complete in an instant. This means that the execution is atomic. Let us refer to this instant as the point of linearizability (PoL). If one operation ends before another operation begins then the PoL of the first operation is bound to be before the PoL of the second operation. This is because the PoL is constrained to lie within the start and end times of each operation. Note that in sequential consistency, we did not have such a constraint. This is how some anomalous outcomes were possible; however, linearizability prevents them. If we consider the previous example, then linearizability will not allow the outcome $Rx1$. It will instead be $Rx0$. This appears to align with common sense.

Given the definition of linearizability, let us answer a couple of questions.

1. Is every linearizable execution sequentially consistent?
2. Is the converse true?

We know that every sequentially consistent execution is not linearizable from our running example. This is because the outcome $Rx1$ is not allowed in a linearizable execution. Let us thus concern ourselves with the first question. This is indeed true.

Relationship with Sequential Consistency

Theorem 7.1.1

Every linearizable execution is sequentially consistent.

Proof: Sequential consistency is program order + atomicity. Every operation in a linearizable execution has a point of linearizability at which it appears to execute instantaneously. Hence, by definition, the execution is atomic.

Let us look at program order now. Consider any process. It is clear that it cannot start a new operation before the previous operation has ended. The PoL of the previous operation is before its respective end time. The order is thus as follows. Consider two operations A and B issued by the same process. Let $A \rightarrow B$, which means that B started after A ended, which will always be the case. Two operations issued by the same process can never overlap. Now, let A_{st}, A_{pol} and A_{end} be the start time, point of linearizability and end time, respectively, for Process A. The orders are thus as follows:

$$A_{pol} \rightarrow A_{end}$$
$$A_{end} \rightarrow B_{st}$$
$$B_{st} \rightarrow B_{pol}$$
$$\therefore A_{pol} \rightarrow B_{pol}$$

We thus observe that the points of linearizability respect program order.

We have thus proven that linearizability respects both program order and atomicity, hence, every linearizable execution is sequentially consistent. ■

We can thus say that linearizability is a *stricter* consistency model because every linearizable execution is sequentially consistent, but the converse is not true. An astute reader may want to ask if sequential consistency was found to be too impractical and restrictive, how come linearizability is more desirable in regular distributed systems?

This is indeed an interesting question that crosses the mind of every student at least once. There are several answers to this question. The first is that it serves the role of a golden model. Even if it cannot be achieved, it is still a good target or a good reference. Other consistency models can be evaluated in the backdrop of linearizability. The second is that it is not always impractical. The reason that sequential consistency and by implication

linearizability was discarded in tightly coupled multiprocessor systems because those systems are very latency critical. Memory operations are very frequent and an additional 50 nanoseconds per memory request can spell disaster. However, in a distributed system operations are relatively more infrequent and thus some degree of delay for a stricter level of consistency may be desirable. For example, if a system sends a message once every 100 milliseconds, then a few additional milliseconds makes little difference. Consider an e-commerce platform. Users wouldn't really mind a delay of even a few seconds. However, losing items that have been added to the shopping cart is a far more serious issue. Given that many distributed systems are far less latency-critical than multicore processors, it is sometimes a wise idea to provide the strictest form of consistency, i.e., linearizability and make the execution far more intuitive. It is much easier to verify the execution also.

Moreover, it is also easy to implement linearizability. An operation ends when an acknowledgement is received from the underlying storage system indicating that the operation has been fully completed. This is easy to realize in practice. Some software module needs to keep monitoring the status of the operation and finally end the operation once all the changes to permanent state have been made. This does increase the latency of the operation because there is a need to wait till all the changes have been made. In many cases, this is tolerable given the potential benefits.

Linearizability and Concurrent Execution

Let us take a second look at linearizability. Let us say that operation A precedes operation B ($A \prec B$) when $A_{end} \leq B_{st}$. These operations may be issued by different processes.

Clearly, given the definition of linearizability $A \prec B$ implies that $A_{pol} < B_{pol}$. This is why linearizability respects the real-time order. Let us consider the more complicated case where two operations are overlapping. They cleared need to belong to two different processes. This means that $A \not\prec B$ and $B \not\prec A$. They are concurrent, i.e., $A \mid\mid B$. In this case A and B are *overlapping*.

In this case, there is no fixed rule for the points of linearizability. Both the options are possible: either $A_{pol} < B_{pol}$ or $B_{pol} < A_{pol}$. In other words, in terms of a happens-before order both the following orders are possible: $A \rightarrow B$ or $B \rightarrow A$.

Remark 7.1.1

In a system that ensures linearizability, overlapping operations can be ordered in any manner.

There are some caveats here. All the processes need to see the same order of operations. For example, if the system has ordered A before B, then all the processes must agree on this. They also must agree on the points of linearizability.'

Example with Concurrent Queues

Recall that we had explained sequential consistency with basic read and write operations on registers. In this case, we can explain linearizability with more generalized distributed objects. Let us consider one such popular example namely concurrent queues. A concurrent queue supports two operations: `enqueue` and `dequeue`.

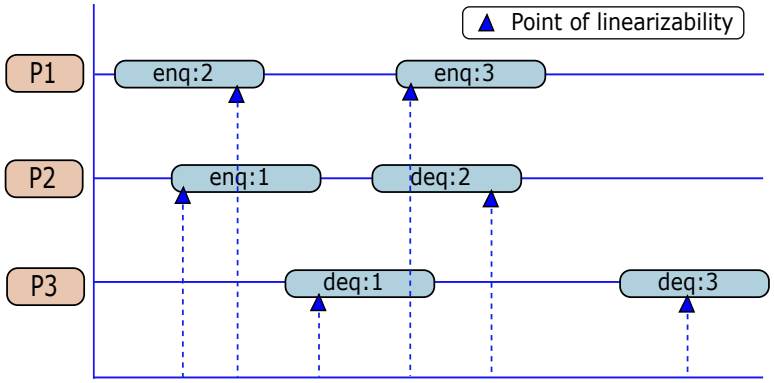


Figure 7.2: Execution of a concurrent queue. Note the ordering between overlapping operations. A linearizable execution is free to order them in any way as long as all the processes agree on the ordering.

Let us use the following terminology. $enq : 4$ means that the number of 4 was enqueued. Similarly, $deq : 3$ means that the number 3 was dequeued. Assume that these are blocking calls – a dequeue operation always returns with a value. Consider the sequence of operations shown in Figure 7.2. We observe a set of operations issued by multiple processes. The start time and end time of each operation is clearly visible. It isn't always necessary for the entire system to be aware of the points of linearizability of all the processes.

Instead, when an execution is analyzed later on, it should be possible to find unique points of linearizability for each operation. Furthermore, these points should lie between the start and end point of the operation. We can always arrange these points on a sequential timeline and make a single process simulate the entire distributed execution. The outcomes should be the same. Needless to say the outcomes should be *legal*, i.e, they should be semantically correct and should match the outcomes obtained in a single-threaded setting if the operations are scheduled in accordance with their points of linearizability.

Summary 7.1.1

Linearizability, in essence, is a specification. It specifies the set of valid outcomes of a parallel execution of processes. The key concept is the point of linearizability that needs to lie between the start and end of each operation. The entire operation "appears" to execute at that point instantaneously. It is stricter than sequential consistency and is a gold standard in distributed systems. It is applicable for analyzing all kinds of distributed systems that have clearly defined operations with start and end times for each. The points of linearizability need not be known or advertised; however, it should be possible to find such unique points by performing a post-facto analysis of an execution's outcomes.

7.1.3 Snapshot Isolation

Linearizability is without doubt the golden consistency criterion. However, it is sometimes hard to implement in practice mainly because of its restrictive in nature. In this case, we desire a weaker criterion that is still very useful in many scenarios. Let us consider one such criterion that is quite useful for updating database tables, traversing linked lists and dealing with large data structures in general.

Consider multiple overlapping concurrent operations. Assume that they are working on different parts of a large data structure like a linked list. They may be adding, modifying or deleting entries. Is there a need to sequentialize (serialize) them – one after the other – and ensure that they execute in order? This would be tantamount to obtaining a lock on the entire linked list and using that to serialize the operations. This is not needed because they are after all working on different disjoint components of the linked list. The operations can do their job in parallel. This will

enhance the performance of the overall system.

Let us define snapshot isolation (SI) formally. We define an operation as a sequence of instructions initiated by the same thread. Examples include enqueue and dequeue operations on a concurrent queue and add, remove, and modify instructions on a linked list. They need to appear to execute atomically if we consider strict consistency conditions like linearizability, which hold for individual instructions and multi-instruction operations alike. An operation in the case of conventional databases is a transaction (refer to Chapter 9). However, we shall keep the treatment generic in this chapter and use the term *operation*. The key features of SI are as follows.

- Just before an operation begins, it takes a snapshot of the entire data structure. Note that there is often no need to create a copy of the entire data structure. This can be done conceptually.
- The execution of the operation proceeds on the private copy of the data structure – the snapshot.
- Once the operation is done, all the writes are atomically committed – the modified variables are updated in one go.
- The operation succeeds if no variable that has been modified by another operation after the snapshot was collected. Otherwise, the operation aborts. It appears to an external observer that the operation never began. Most systems retry the operation once again.

Remark 7.1.2

The snapshot isolation consistency model is ideally suited for a system with multiple processes that independently work on disjoint parts of a large data structure. They can operate in isolation and need not see the changes made by concurrent processes/threads. At the time of writing the modifications to the permanent state (committing), the entire operation is atomically committed if no conflicting writes were committed by a concurrent operation.

Snapshot Isolation vs Linearizability

Let us explain with an example. This pattern is known as *write skew*.

Thread 1	Thread 2
`begin`	`begin`
Rx0	Ry0
Wy1	Wx1
`end`	`end`

In this case, Threads 1 and 2 execute two different operations each. The start and end of each operation are demarcated with the keywords `begin` and `end`, respectively. The first operation (initiated by Thread 1) reads the value of x that was initialized to 0, and then writes 1 to y. The second operation does the reverse. It reads the value of y to be 0, and then it writes 1 to x.

Consider sequential consistency and linearizability. Both require the operations to appear to execute instantaneously at a single point of time. This means that either the operation of Thread 1 is ordered before the operation of Thread 2, or vice versa. In either case, one of x or y should be read as 1. It is not possible that at the end of these two operations, we conclude that $x = y = 0$. This behavior is forbidden by both the consistency models.

Now, let us consider snapshot isolation. We assume that just before the operations begin, each thread takes a conceptual snapshot of the entire memory space. It works on a private copy of the snapshot and atomically commits all the changes. If we assume that both the operations start at the same time and execute in lockstep, then we can clearly see that both of them will begin with reading $x = 0$ and $y = 0$. Then, they will set $y = 1$ and $x = 1$, respectively. Finally, they will try to commit their changes. The commit operation will be successful because there is no write-write conflict. This basically means that there is no write to a variable by another thread/process after the snapshot has been taken. Read-write conflicts where another thread or process may have read a value after the snapshot do not lead to failed operations and aborts (failure to commit). This case exemplifies the latter, where there is a read-write conflict but no write-write conflict. As a result, both the operations go through and the outcome $x = y = 0$ is valid under snapshot isolation.

Utility in Modifying Large Data Structures

The benefits of this consistency model are accrued when we consider large data structures such as very long tables or linked lists. Ignoring read-write conflicts is often a good idea in such scenarios. Consider the linked list shown in Figure 7.3.

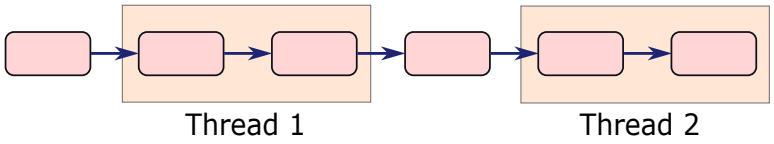


Figure 7.3: Two threads trying to make modifications to a linked list. They operate on disjoint parts of the linked list.

Consider the threads 1 and 2 that are operating on disjoint parts of the linked list. It is possible for Thread 2 to cross the region that Thread 1 operates on and move to its region of interest in the linked list. Thread 1 can then make its changes. Both the threads can operate in parallel and make their modifications. The changes can then be committed without any issues because the same linked list element is not modified by both the threads. This means that there is no write-write dependence. Hence, as per the rules of snapshot isolation, both the sets of modifications can go through and get committed (finalized). Note that Thread 2 reads the linked list elements that Thread 1 subsequently modifies in a concurrent operation. However, this is allowed in snapshot isolation.

7.1.4 Causal Consistency

Let us focus on the $write \rightarrow read$ dependence further. Consider the example of a forum where there are comments and replies. If comment A is referenced in another comment B, then it should never be possible for a user to observe B before observing A. It is absolutely fine if an earlier unrelated comment C is not visible. However, there is no violation of causality here, which means that no cause-effect relationship is being violated here. This means that it should never be possible that the effect is visible, but the cause that led to the effect is not visible. For instance, in our example, comment B was the effect, and A was the cause.

To enforce causal consistency, we need a communication path from every cause to every event. This means that some message should be sent with the timestamp of the cause to the server that executes an event that is *dependent* on the cause (the effect). Assume an online collaborative document-editing platform. User A writes a sentence. User B writes another sentence to serve as a reply to the sentence written by user B. It should not be possible for a third server to read B's sentence before reading A's sentence. The main objective is to store the *backward slice* of an event. The backward slice com-

prises the set of all events that *happen before* the event, their predecessors, and so on. This can be achieved as follows.

Let us outline a method to achieve causal consistency. Let us use vector clocks (refer to Section 2.3.2) that guarantee that if $t(e_1) < t(e_2)$ then $e_1 \rightarrow e_2$. Since causal consistency is to be ensured, we need to ensure that if an event e with vector timestamp v_e is received, all the events that it "causally" depends on have also been received. Consider an example. $v_e = [3, 0, 1, 2]$. This means that all the vectors that are less than it should have been observed by the process. For example, it should have seen $[2, 0, 0, 1]$. In fact before event e is processed, all the events with vector timestamps that precede $[3, 0, 1, 2]$ should also have been observed. If any event has not been observed, then there is a need to wait.

This method will ensure causal consistency; however, it is difficult to realize in practice – a lot of vector timestamps need to be stored. It is possible to optimize this process by removing entries that are redundant. Consider two timestamps: $[0, 0, 2, 0]$ and $[0, 0, 1, 0]$. There is no need to store $[0, 0, 1, 0]$ because it is *dominated* by $[0, 0, 2, 0]$. In spite of such optimizations, the storage space requirement is high. If there are additional guarantees on network delivery times, then the storage requirements can be limited. However, if messages can be delayed for a long time or even dropped, then storing a list of "dropped" messages may end up taking a lot of space. We can alternatively do the reverse, and store all the gaps (missing messages/events) in the backward slice of an event. Regardless of what is actually stored, there is a need to design an efficient mechanism for tracking the completeness of the backward slice.

7.1.5 Approximate Consistency

Let us discuss approximate consistency models. In this case, some amount of error is tolerated in the output subject to pre-specified bounds. The reason for tolerating some degree of error is that it is possible to create a more efficient implementation. We need to understand that the reason for not preferring strict consistency models such as linearizability and sequential consistency is that they are very hard to realize in practice. To adhere to all the ordering requirements, it is necessary to delay a lot of messages, add extra acknowledgement messages and in the case of causal consistency, maintain a lot of additional state. Such approaches do not scale to a large number of nodes. It becomes hard to create web-scale systems that follow such consistency models. Hence, it makes sense to simplify the system and once in a while compromise on the consistency.

A natural question that arises is that if the consistency can be relaxed once in a while, why cannot it be relaxed all the time? After all, any deviation from the consistency model leads to a violation of correctness. The results can be catastrophic, and it is possible that the distributed program may crash. Hence, in general, relaxing consistency occasionally is not advisable. It may cause unrepairable errors. However, there are specific cases in which consistency can be relaxed sometimes. The deviations in correctness may be tolerable. Typically, in such systems, lapses in consistency happen when the system load exceeds a threshold. A trade-off is thus established between high-throughput operation and impaired consistency.

Let us thus look at specific approximate consistency models and the situations in which they are found to be useful.

Bounded Staleness

Consider a distributed object that is modified by processes in the distributed system. Every time a write is made, assume that a new version is created. Any read operation should return the latest version (written by the latest write). Assume that it is allowed to return one of the last k versions. This means that the value returned by a read may not be the most recent, but it is an intact version of the data that existed in the recent past. This model relies on bounded staleness and is clearly not suitable for most situations. However, there are some situations where this is acceptable. It does end up causing some discomfort to the user of the system once in a while, but it does not cause system failures.

Consider an e-commerce website. Assume the user enters a search term. She is shown a list of products that match the search term. Each entry can be thought of to be a distributed object. The version shown on the screen matches a certain version. It may not be the most recent version because the server that has prepared the website may not have it. It may be showing a slightly stale version of the product. It is possible that there might be a change in the product's description, photograph or price. In this case, the main focus is on showing results to the user as soon as possible. Most of the information should be correct and accurate. Given that product versions do not change very often, the servers can afford to lazily synchronize themselves. The flip side is that occasionally some product versions that may be shown to users may be slightly outdated (stale). This is not expected to cause major discomfort. It may so happen that the user is not interested in the product at all. Even if she is interested, she may not care about the specific details that have changed. In any case, most users of e-commerce sites are

well aware that the view that they see is subject to change at any point of time. When she clicks a certain product's icon, she is directed to a new page that shows all the details. This action does not require a lot of information to be fetched. The user is prepared to wait a little more to see accurate information. This information can be shown to the user after a slight delay, and then she can make a choice.

The operative point here is that minor disruptions of the kind described, where a slightly older version of the product's description is shown, are not expected to be perceived to be very problematic. If this can lead to a large reduction in the number of messages that the servers need to send between each other to maintain strict synchronization, then this slightly relaxation of synchronization is a beneficial idea.

Numerical Consistency

Let us now consider another example where consistency is relaxed differently. Assume we have sensors to monitor an oil pipeline. This is a distributed system where a network of sensor nodes and gateway nodes collaborate to produce a "distributed snapshot". In this case, a distributed snapshot is a set of all the values read by all the sensors. They could for instance sense pressure or temperature. Depending upon the readings, appropriate action can be taken.

If the system has very strict guarantees of consistency, then it is guaranteed that all point of time, we will get an up-to-date system of the entire network. However, this requires a lot of messages and is hard to achieve when the nodes are running on intermittent sources of power such as solar energy. Hence, any practical system tries to reduce the number of sent messages. There are several ways of doing this. Internal nodes may compute the average of readings and just report the mean value as opposed to all the individual readings. Furthermore, the sensor nodes may not have sensed their values at exactly the same point of time. If the protocol follows loose synchronization, then the time duration between different sensed values may be large. For a combination of such reasons, we can conclude that the picture of the network presented to the user will have some degree of inaccuracy.

The level of inaccuracy in this case can be traded off with higher message traffic. The more messages we send, the more accurate will be the values and vice versa. This is an example of relaxed numerical consistency, where the inaccuracy is computed in terms of the deviation of the value shown to the user and the ideal value.

Probabilistic Consistency

Probabilistic consistency is a special kind of bounded staleness where different replicas in a distributed system can diverge and different versions of an object can be served subject to the degree of staleness obeying a probability distribution. For example, a distribution could be of the following form: latest version (90%), 2^{nd} last version (7%) and 3^{rd} last version (3%). This means that statistically 90% of the last time users will be able to see the latest version of an object. In this case, the quality of service is specified probabilistically.

Consider a content distribution network (CDN) that hosts web pages. CDNs are very popular because the developer of a popular website may not have the resources to host the site and serve it to millions of users. Her server may crash. It is a wiser idea to use a third-party service that hosts websites and scales with demand. This turns out to be much cheaper because an economy of scale can be achieved. A CDN typically hosts sites for thousands of clients. Often their peak periods of usage do not overlap. This ensures an optimal use of server resources. A CDN achieves scalability by using thousands of individual servers that serve copies of the website. If the website is updated, then the content provider needs to inform the CDN. The primary server of the CDN needs to get all the replicas updated. This can take variables of time and is highly dependent on the load placed on the CDN. Hence, it is possible that users may see different versions of a site in a short span of time.

However, this is not something that any website developer would like. It is possible that it may be a news site. There is a possibility that the public may get mislead or incorrect scores for a game of cricket may be displayed. This is where there is a need to establish a strict QoS (quality-of-service) guarantee where the degree of staleness is governed by a probability distribution. If the developer is comfortable with let's say a 5% of chance of not seeing the latest data, then she may be fine with it.

7.1.6 Eventual Consistency

This is a very weak form of consistency that only makes sense for very large systems and that too not all the time. There are no guarantees on the degree of staleness, the time to propagate updates or resolve conflicts. It can take a long time to propagate the updates to all the nodes. However, if the system remains quiescent for a long enough period where there are no writes, then all the replicas are guaranteed to contain the same state. This is where the

term “eventual” comes from.

It is a form of consistency where the short term is prioritized. The system should always be available and responsive even though the values that are being sent to users may have different degrees of staleness. The only correctness criterion is the final convergence guarantee. A period of quiescence without writes produces the same state across replicas.

Distributed key-value stores such as Amazon Dynamo [DeCandia et al., 2007] typically use eventual consistency. Items such as a user's profile or shopping cart are stored across replicas distributed worldwide. It is possible that a search query may return stale data because the replica that it reaches hasn't received the update. This can continue to happen for a short duration of time. Given that the framework uses a reliable message passing protocol, the updates will ultimately reach the replicas. At that point of time, the replicas will be updated and ultimately all the replicas will receive the update. This is the guarantee of eventual consistency. If the system is allowed to stabilize then an update is received by all the replicas and all issues arising out of inconsistency are resolved.

7.1.7 Client-Centric Consistency

Let us now focus on a different type of consistency models that focus on the experience of a single client as opposed to taking a systemwide view. Clients look at the values of reads and writes from their own perspective. They need not be aware or even take into consideration the state changes happening at other nodes.

Any client-centric model ensures that individual clients are provided some guarantees. From their point of view, some correctness conditions should hold regardless of the system state. Note that different requests made by the same client may reach different replicas in the distributed system. However, clients are not concerned about this. They simply look at the values that they have read or written.

Definition 7.1.8 Client-Centric Consistency

A client-centric consistency model determines the behavior of the system from the point of view of any individual client. A client is a regular user who sends requests to the distributed system. It is processed by either by one replica or a set of replicas. Different requests made by the same client can be processed by different sets of replicas. The client is oblivious of this and is only concerned with the values that it reads and writes.

There are numerous client-centric consistency models. Let us look at some of the major variants in this space.

Monotonic Reads

If a client has seen the newer version of an object, it cannot subsequently read an older version. This implicitly means that it is possible to *order* the versions of an object seen by a client.

Consider the case of an email inbox. Assume that no mail is deleted. If the client observes 100 messages in the mailbox first and then observes 95, it means that the client observes a new version of the mailbox first and then observes an older version. This is not allowed. The client should never see stale data (from its point of view). Note that this consistency model is not dependent on the state of the system and messages received by other nodes. The only criterion here is perceiving monotonic reads, which means that no client should feel that it is regressing in time.

Monotonic Writes

Consider a sequence of writes issued by the same client. Let them be ordered one after the other. All processes perceive the same order of writes. Assume that a client issues write w_1 and then write w_2. All the processes perceive w_1 to have happened before w_2.

The guarantee of monotonic writes is typically provided in collaborative systems that maintain sessions. Every client maintains a sequence number for its updates. It is incremented by 1 for every subsequent update. Each replica maintains state for each session akin to Lamport clocks. Assume it has received all messages till sequence number 7 from a given client. Next, it receives a message with sequence number 9. It buffers the update and does not apply it. It waits for a message with sequence number 8 to come from the client. After that it applies the updates in the messages with sequence

numbers 8 and 9 in order. This ensures that all replicas respect the order of updates of each client.

Consider collaborative document editing. In this case monotonic writes translates to all users seeing the same sequence of updates made by a given user. Regardless of the synchronization between the users, this is a reasonable correctness criterion – no client sees a lost update. Otherwise, this will lead to non-intuitive behaviors. Client 1 will see Client 2's later writes before its earlier writes. The shared document will stop making sense at a lot of points of time.

Read Your Writes

Let us now combine reads and writes. If a client performs a write, all subsequent reads by the same client are always guaranteed to see the write operation.

Sadly, this property can be violated on most practical systems. The write is sent to one replica. Subsequently, the client sends read request to other replicas that haven't received the update. In this case, this consistency model is violated. Consider the case when a user updates her profile picture. A while later when she accesses the same website, she does not see her latest profile picture. Instead, she sees the older version of her profile picture. This means that the client is not reading her own writes.

To ensure that the client always observes her latest write, it is necessary to implement write sequence numbers as we had proposed in the case of monotonic writes. Every time a read operation is sent to a replica, the write sequence number should also be sent. If the replica does not have the update with that sequence number, it needs to first fetch it from another replica. Subsequently, it needs to reply to the read operation. This will ensure that the latest write by a client is always visible to it, even if she switches the replica.

Writes Follow Reads

Assume that a client first reads the value of a given object that has many fields. Subsequently, it writes to some fields of the object. In this case, there is a causal relationship between the read and the subsequent write. Now, at a later point of time, if the client's request goes to another replica, it should never be the case that the client can see the result of her write operation but cannot see the values that she has read or newer values.

Let us explain with an example. Consider a discussion thread. A client

sees a comment and replies to the comment. It should never be the case that the reply is visible, yet the original comment is not visible. This will be a violation of causality and should not be allowed. From the client's point of view, writes are not following reads. The reply to the comment is a write, which has strictly happened after *reading* the original comment. Hence, either just the original comment should be visible on another replica or the comment and its reply (both) should be visible.

Ensuring this is simple. Every reply to a comment needs to have a pointer to the original comment. If the original comment has not been received, then the reply should be buffered until the original comment is received. This will ensure the property that a subsequent write is never observed before an earlier causally-related read.

Session Consistency

Consider a typical email session. The client logs in to the website of the email service provider such as `gmail.com`. A typical email session may involve reading and writing a lot of mails. The client may additionally do other activities such as access her calendar or list of tasks. All of them may be a part of the same login session where these features are a part of the email site. In a long browsing session, the client may be directed to numerous replicas. It is expected that normally all the four aforementioned client-centric consistency properties will be honored – this is known as session consistency. They will together ensure that a client will never read stale data, never miss data, all its writes in the same session will be observed to be in the same order globally, and it can only observe data that has been read before or a newer update. This will ensure that the behavior of the system is on expected lines.

The advantage of session consistency is that it optimizes the overall client experience. It ensures that regardless of whatever is happening to other replicas and clients, the experience of no single client is compromised.

Definition 7.1.9 S

ession consistency provides all four client-centric consistency guarantees namely monotonic reads, monotonic writes, read your writes and writes follow reads. They maximize the client experience and ensure that no single client perceives non-intuitive behavior. They however do not provide system-wide guarantees quite unlike linearizability or snapshot isolation.

7.2 The CAP Theorem

Eric Brewer proposed a conjecture in 2000, which said the following: we cannot design a protocol that achieves some form of strong consistency, service availability and network partition tolerance at the same time. At that point of time, this was an intriguing hypothesis. The community thought that we can indeed design a protocol that should be resilient to all kinds of failures. Of course, there are many definitions of these terms. A simple definition of consistency here is that a read fetches the value written by the latest write, and all operations appear to execute instantaneously. The term *availability* means that every request is replied to in a finite amount of time. Partition tolerance refers to the fact that even if the network is partitioned into two, the system continues to work. Gilbert and Lynch [Gilbert and Lynch, 2002] were able to formally prove this conjecture in 2002. The PACELC theorem first proposed in 2010 (and formally proven in 2018) built on this basic result. It adds an element of the consistency-latency trade-off during normal operation, which CAP did not cover. A lot of this thinking has been incorporated in state-of-the-art commercial designs.

For any real-world web service, we would expect that all these properties hold. We would ideally not want to sacrifice any of these properties. However, life is not always that rosy.

7.2.1 Basic Definitions

Before we start, a disclaimer is due. The term atomic consistency has multiple definitions, and it varies from author to author. The particular definition that we shall use is by Nancy Lynch [Lynch, 1996], where she defines *atomic consistency* to be a property that is the same as linearizability [Herlihy and Wing, 1990] (see Section 7.1.2). Consider a distributed object such as a simple shared variable (also known as a register), a database or a complex distributed data structure. All of them can be virtualized as a concurrent object that supports a set of methods that operate on it, change its internal state and return a value.

The system is said to provide *availability* if every request leads to a response unless the node generating the request fails. This basically means that the system cannot become unresponsive. Finally, we have *partition tolerance*. This means that even if the network is partitioned and a pair of nodes (across the partitions) become mutually unreachable, the system remains operational. Some part of it continues to provide responses to requests.

7.2.2 Impossibility Results in Asynchronous Settings

Theorem 7.2.1

A basic read/write register in an asynchronous system cannot guarantee all the following properties, when network partitions are allowed:

- Availability
- Atomic consistency
- Fair execution (all the requests are processed in a bounded amount of time)

This is known as the CAP theorem, because the statement of the theorem implies that it is not possible to provide consistency (as per our earlier definition), availability and partition tolerance at the same time.

Proof: Assume that there is an algorithm A that provides all the three CAP guarantees. Consider a point of time, where the network is partitioned. The set of nodes V in the network is partitioned into two non-overlapping subsets: V_1 and V_2 (resp.). Any message sent from node $u \in V_1$ to node $v \in V_2$ is dropped.

Consider a distributed read/write object x. If a write operation occurs in u ($\in V_1$) and later on the same object is read in v ($\in V_2$), then the write will surely be missed. This is because no message can be sent from any node in V_1 to any node in V_2. As a result, the information regarding the write cannot be transmitted. Let us look at this situation more formally (on the lines of the original proof).

Let the value x_0 be the initial value of the read/write object. Consider an execution (sequence of operations) that only contains a write to x by node $u \in V_1$. This write sets the value of x to x_1 ($\neq x_0$). Let this execution be e_1. This write needs to complete because it cannot stall indefinitely (fair execution) and it needs to generate an acknowledgement or response (availability requirement).

Let us next consider an execution e_2 that only consists of node $v \in V_2$ reading the value of x. We can assume that the execution terminates after this read operation.

Assume that there are no additional client requests after the respective operations in e_1 and e_2 finish. Let us now consider the combined execution

$e = e_1 \mid e_2$ (first e_1 then e_2 after a long time). Given that the network is partitioned, the nodes in V_2 will not be able to distinguish between e and e_2. This is because they will not receive any message arising from the write in e_1. If we only consider e_2, then the read should return x_0 (the initial value). However, if we consider the combined execution e and the atomic consistency property, then the read operation should return the value of the last write, which is x_1.

This situation is clearly contradictory in nature. We don't know what to return – x_0 or x_1. If we were to return x_0, then it will violate atomic consistency. If we return x_1, then it will violate causality because the value x_1 was never sent to the nodes in V_2.

Hence, it is not possible to have any such algorithm A. This proves the CAP theorem. ■

We clearly cannot guarantee/provide all three properties at the same time. It turns out that we can prove a very interesting corollary. Even if no message is lost, yet the network is asynchronous, it turns out that we cannot guarantee availability and atomic consistency together.

Corollary 7.2.1

The CAP properties cannot be guaranteed in an asynchronous network even if we assume that no messages are lost.

Proof: Recall the FLP result where we had proved that it is not possible to differentiate between a scenario in which a message is just "delayed" or it is lost. Now assume that an algorithm exists that guarantees atomic consistency and availability in scenarios in which no message is lost. It would mean that the algorithm will work in all scenarios in which messages are arbitrarily delayed. This is because there is no way to distinguish between a message loss and an arbitrary delay. If arbitrary delays are also covered, then this makes the system partition tolerant as well because we can simply treat any message dropped because of a partition as just "delayed".

It appears that we can provide all three CAP guarantees now, which is impossible. Hence, such an algorithm cannot exist. ■

Solutions in Asynchronous Settings

Given that we cannot provide all three CAP guarantees, let us look at providing only two.

Assume that the condition of availability can be relaxed. In this case, we can have a single centralized node that responds to all requests. It can

clearly provide atomic consistency (as defined in the CAP theorem) and the algorithm is also partition-tolerant. The reason for this is that the nodes in the partition that has the centralized node continue to get responses to their requests. The rest of the nodes need not get responses, which is fine, because availability is not guaranteed. Of course, many may have an issue with such a system because some partitions will become unresponsive. We are not making any guarantees to nodes that are in those partitions. Nevertheless, from the point of view of an academic discussion, atomic consistency and partition tolerance can indeed be provided if we relax availability.

Let us now relax partition tolerance, which means that if there are partitions, the system will simply go down. Let us consider the execution when there are no partitions and the system is up and running. We can use the same centralized algorithm. Every node in the distributed system sends a message to the centralized node. This node processes requests one after the other, sequentially and atomically, and responds to every single message. We observe that both atomic consistency and availability are provided.

Let us now consider the last combination, which is partition-tolerance and availability. In this case, atomic consistency is being relaxed. Every partition can have a cache (similar to a web cache), which can store the results of the most common queries/requests and their corresponding responses. In case, a request is not there, some old value (response of some other request to the same register) can be given or even a default response can be returned. For instance, if we are trying to simulate a distributed register, we can always return the initial value as a fallback mechanism. This will break atomic consistency, but the system will still be partition-tolerant and available. Every request will be matched with a response, even thought the response may contain a value that is old and outdated (non-atomic).

The summary of this section is that it is always possible to provide two out of the three guarantees. However, such systems will seldom be practical and usable.

7.2.3 Partially Synchronous Networks

Let us now consider partially synchronous systems. In this case, we assume that every node has a clock, yet the clocks across nodes are not synchronized. They however increase at the same rate. This means that there is some loose synchrony between them, but we cannot assume that they return the same value at the same instant of time. We can nevertheless have some strict guarantees in such systems: every message is delivered within a bounded amount of time, and it is processed within a certain amount of time.

Theorem 7.2.2

In a partially synchronous system, it is not possible to implement a distributed register that guarantees atomic consistency, availability and partition tolerance in executions where messages may be lost.

Proof: Consider a network that is partitioned into two parts: G_1 and G_2. In the first component G_1 we implement a write operation that is successful. Assume we write the value x_1 to a distributed register x. Let x be initialized to the value x_0. Let us now wait for a long time.

This time duration should at least be several times the maximum message transfer and processing latency. Then, let us try to read the register x in the partition G_2 by invoking a read operation. Given that there is a strict partition, there is no way that the value x_1 could have been communicated to any node in G_2. As a result, the read operation (in G_2) will return x_0. Note that because of the availability property, every request has to return with a response. This clearly violates atomic consistency in a partially synchronous setting because the read request starts long after the write request has completed. Given that we have defined atomic consistency to be the same as linearizability here, we expect the real-time nature of the consistency model to be respected. This means that the read in G_2 should return x_1 instead of x_0. However, because of the partition, this is not possible. We thus observe that atomic consistency is not being provided.

This proves the CAP theorem for partially synchronous settings. ■

Corollary 7.2.2

If messages are not lost, then all three guarantees can be provided in a partially synchronous system.

Proof: Here, the results of the FLP theorem will not hold because this is a partially synchronous system, not an asynchronous system.

If messages are not lost, there are no partitions and thus partition tolerance is per se not required. We can design a centralized system as discussed before, which can guarantee availability.

Let us next discuss consistency. Every request will be matched with a corresponding response and the central node can implement a queue that ensures that messages are processed and responded to as per their order of arrival. This will guarantee atomic consistency and adherence to real-time specifications. Each node will wait for a response to its outstanding request before sending the next request. This will guarantee program order. Hence,

we have all the three conditions of linearizability (atomic consistency in this case) satisfied.

Hence, all the three CAP guarantees can be provided. ■

Summary 7.2.1

When messages are lost or when message loss cannot be detected, atomic consistency cannot be guaranteed – we may return stale values. This is the case in asynchronous systems where it is not possible to differentiate between a message loss and a message delay. In partially synchronous systems, it is possible to provide all three guarantees when messages are not lost.

7.2.4 The PACELC Theorem

The CAP theorem has its inherent limitations. Many of the trade-offs are relatively obvious. If there is partition tolerance, then it is obvious that either consistency or availability need to be sacrificed. Messages from the first partition cannot reach the second partition, and thus clients that access the second partition will never get updates sent to the first partition. Either servers in the second partition never send their replies sacrificing availability or return stale values (sacrificing consistency).

Moreover, note that the CAP theorem is for failures. It does not handle the normal case when there are no failures, yet the system might be running slow because of a high request rate. The user of a distributed system is mostly concerned about latency. This metric does not figure in the CAP theorem. Also note that failures such as network partitions occur quite rarely. Modern networks are quite reliable and thus a large distributed system getting partitioned into two parts where the clients accessing one part cannot reach nodes in the other part, is rare.

Traditional data storage systems make ACID guarantees. ACID stands for atomicity, consistency, isolation and durability. These properties sadly do not figure in the CAP theorem. Hence, there is a need to extend the CAP theorem to incorporate the trade-off between latency and consistency. We can think of availability as a metric that is closely related to latency. If a system's latency is very high, then it is perceived to be non-available. Similarly, if some nodes have gone down, then the latency is perceived to increase. This is because in this case, data needs to be fetched from other replicas or there may be a need to reconstruct data. If the system provides strict consistency, then there is a need to send a lot of messages between

the replicas and ensure that the data that is served to the client is as fresh as possible. Hence, the latency of such systems is expected to be high. We can infer that stricter the consistency, higher the latency. There is thus a trade-off between them, which needs to be formally explored.

Definition and Use Cases

Given that the CAP theorem only conveys a part of the story, the community felt that there is a need to expand its scope. In 2012, Abadi proposed the PACELC theorem [Abadi, 2012]. It tries to answer the following questions. If there is a network partition (P), how is a trade-off achieved between availability and consistency? Else (E), in the normal case when there are no failures, what is the trade-off between latency and consistency? In such a circumstance, which is the normal case, the trade-off between latency and consistency is the key question. Given that a no-partition system is easy to create given today's highly fault-tolerant and redundant networks, the normal case is more common and will be the focus of our subsequent discussion.

Definition 7.2.1 PACELC Theorem

If there is a network partition (P), then there is a trade-off between availability (A) and consistency (C). Else (E), there is a trade-off between latency (L) and consistency (C).

Let us thus proceed assuming partition tolerance. There are four possible choices: PA/EL, PA/EC, PC/EL and PC/EC. PA/EL means that partition tolerance and availability are provided *else* between latency and consistency, latency is chosen.

Let us understand the choices that modern systems make. Consider large DHTs that are often PA/EL systems. This means that if there is a partition, then between consistency and availability, they choose availability. Consistency is sacrificed, which means that stale data can be returned. This makes sense in a large distributed system because users know that data will be sometimes inaccurate. This is the price of scalability. However, there will be a need for rectification and reconciliation later. The important thing is that the system at all times should be responsive. Between latency and consistency, they choose latency because they would like their system to respond quickly. This means that stale, inconsistent data can be returned even in normal operation. However, the users will still observe a very responsive system. Note that in both the cases – with and without failures –

the same choice is made, i.e., consistency has a lower priority.

Now, focus on PA/EC systems. This means that if there is a failure, consistency is sacrificed. Any system that has a moderate amount of replication will ultimately have to sacrifice consistency if there is a high failure rate and the system gets partitioned. Either the system can become unresponsive (no availability) or decide to serve users with whatever data it has. Many such systems become eventually consistent upon a failure. However, in the normal case, they provide a high degree of consistency and relax latency constraints. Such a design decision makes sense when failures are expected to be rare and operation with eventual consistency as the model is deemed to be acceptable till the system recovers. Data storage systems with multiple replicas typically have such a mode of operation. They inform the users that they are running in degraded mode, when there is a partition. In the "degraded mode", they provide eventually consistent semantics. The moment all the components of the system are up, strict consistency is provided.

Next, consider PC/EL, which at first may seem counter-intuitive to many astute readers. If there is a partition, then clearly consistency is preferred. A partition does not lead to eventually consistent results. This basically means that for a large part of the system, accesses are "frozen" (not allowed to proceed), lest inconsistent results are returned. However, during normal operation, consistency is sacrificed in favor of low latency. This model indeed seems to be an anomaly. The system appears to be getting more consistent when there is a failure, especially a partition. We need to thus interpret this model slightly differently. In the normal case, the consistency is kept above (stricter than) a baseline level. Hence, priority is given to low latency. However, if there is a failure, for keeping consistency above a baseline level, there is a need to sacrifice availability. We thus are pursuing the same objective in both the cases. A typical master-slave system follows this mode of operation. When there is a failure, the primary server or master needs to serve all the requests. For the rest of the nodes that are unreachable, availability is sacrificed. Hence, consistency is provided. If there are no partitions, then there is a baseline level of consistency because the primary server (or master) is reachable from all nodes. Hence, here the target is low latency.

Finally, consider PC/EC systems. Such systems always prioritize strong consistency. Traditional databases that provide full ACID semantics fall in this category. Banks and financial institutions use such systems. For them, latency and availability are goals that can be sacrificed. This is because at no point of time do they wish to return stale data to the user – there will

be adverse legal consequences.

Proof of Correctness

Unlike the CAP theorem, the complete PACELC theorem does not have a proof. The CAP part of the PACELC theorem is provable; however, the trade-off between latency and consistency is something that aligns with common sense and is easily observed. We can say that it is "observationally correct". However, there is no proof given that the *latency* aspect is not well-defined. We never specified what exactly is low-latency and what is high-latency. Nevertheless, some basic results are provable. A theorem in reference [Golab, 2018] states that the sum of the read and write latencies for a distributed object is at least as large as the message delay. However, this looks at a specific subcase.

7.3 Apache ZooKeeper

In a distributed system, there are many standardized services that typically need to be provided regardless of the underlying application. Think of these as common services that are always used in some form of the other. This means that akin to the standard C library on standalone systems, there is a need to provide a similar API on distributed systems.

The Apache ZooKeeper [Junqueira and Reed, 2013] service was created with a similar objective in mind. It provides a service that globally orders operations, provides wait-free linearizable writes and maintains per-client FIFO ordering. The system automatically scales with the number of nodes and thus provides a relaxed consistency model.

It can seamlessly be used to store global configuration-related information, group membership status, leader and follower information, locks and state related to distributed barriers. Pretty much wherever global information is required, ZooKeeper is needed. ZooKeeper maintains the consistency of the data and ensures that all the nodes can easily access such information.

Note that the key objective is not to store large amounts of data. It is not a "big data" engine. The quintessential use case is storing configuration-related global information. These are small parcels of information that need to be accessed by all the nodes in the network. The configuration information can be related to distribute nodes, the network, message formats, passwords and session keys. Given that this information is intricately related to the operation of the system, consistency guarantees have to be

quite strictly specified and enforced such that this information reaches all the nodes in a timely, predictable and controlled manner.

Another use case is leader election and maintaining information about the current leader. The ID of the current leader can be thought of to be a part of the configuration information. On similar lines, this service can be used to implement locks and barriers. The status of the lock and the ID of the lock owner should be globally-visible information. Another use case is the maintenance of views such as all the currently active processes. We have seen in a lot of protocols such as PBFT that maintaining accurate information about the current view is crucial to the operation of the protocol. All the participating processes should be aware of the constituents of the current view.

ZooKeeper can also be used to implement event-driven architectures. If there is some infrequent event in some node, then this information can be relayed to the rest of the nodes. Such events include node failures, node addition and removals. Such anomalies can be quickly relayed to the rest of the nodes such that they can undertake corrective action.

The interesting feature here is that all of this information about the system can be stored in a very structured format. It can be versioned (multiple versions) and serialized (read or written from data files). Runtime logs of the activity can be collated and centralized in one place. This can be made visible to administrators via a centralized console. Furthermore, many add-on services can be attached to ZooKeeper. They can work on the data that is being produced by ZooKeeper's clients.

7.3.1 Overall Architecture

Let us now look at the overall architecture of ZooKeeper. Let us start with explaining the basic components. A *client* is an independent process that uses the ZooKeeper service. It is not a part of ZooKeeper's architecture. The ZooKeeper system has *servers* and *znodes*. A server is a machine that coordinates with other servers to provide the ZooKeeper service. A *znode* functions as a data node. Note that such znodes are organized as a tree. This tree is known as a *data tree*. *znodes* act as generic data objects. They can be used to store any kind of information. Guarantees are only with respect to the timely completion of operations on znodes and their consistency semantics. The guarantees are oblivious of the data stored in znodes.

ZooKeeper uses standard Unix semantics to manage znodes. They are arranged as a tree and the path from the root to a znode is its path. For example, the path of a znode can be of the form "/folder-1/folder-

x/znode_3". Some znodes are regular (explicitly created by clients) and some are ephemeral (can be deleted automatically). In addition, every znode can be associated with a sequence number, which is guaranteed to monotonically increase. Furthermore, the sequence number is more than that of the sequence number of any child created under its parent in the past. In a certain sense, this sequence number uniquely identifies the znode, and differentiates it from znodes under the same parent that may have existed in the past. They could have had the same name. However, the sequence number can eliminate the confusion between them. Along with a sequence number, a client can associate a *watch* event with a znode. This means that if the watch flag is set in a read operation, the client is notified when the data stored in the znode changes. Note that a watch is a "one-time trigger". This means that the client is notified only once, not repeatedly. Moreover, once the session finishes, all such watches are removed. We can think of the data tree as the registry in Windows. It is a hierarchical database that stores a lot of configuration related information with respect to the applications installed on the system, users and the system. ZooKeeper thus provides a hierarchical distributed database that clients connect to, establish a session, read/write data (in the form of accesses to znodes) and disconnect. Note that a client here is not a single machine. It refers to a user who maintains the same session across machines.

Operations in ZooKeeper

Let us now go over the operations supported by ZooKeeper. ZooKeeper supports the usual set of calls for creating znodes, associating data with them, reading/writing them, deleting them, checking their existence and traversing the tree of znodes. The operation worth mentioning is the *sync* operation that It waits for all the outstanding updates to reach the ZooKeeper servers and get acknowledged. This is an important operation, insofar as maintaining consistency is concerned.

Let us now touch upon some of the key design decisions.

In general, in an operating system, when a file is opened for reading or writing, the kernel associates a file handle with the opened file. In this case, it is not necessary to use the path of the file to repeatedly access it. A path is used only once, when a handle is associated with a file. Subsequently, the handle is used for all accesses, and finally the open file is closed using its handle. This is a very good idea for a single system, however, it introduces too much of state in a distributed system like ZooKeeper where in a single session the client can be accessing the ZooKeeper service from different

physical machines. This is why, the designers chose simplicity and mandated that every operation in ZooKeeper needs to have the path of the znode as a mandatory requirement. It is true that this does introduce some overhead in terms of locating the znode; however, the simplicity of the design makes it a viable option.

Next, we need to understand that there are two classical paradigms: synchronous and asynchronous execution. In the former case, the caller waits for the response. In general, in distributed system software, it is not a good idea to wait for the function to return, unless the data is so critical that no other parallel task can be started. It is often a better idea to opt for asynchronous execution, where the caller can move on to doing other work. Whenever a reply is received, the client (the function invoker) can be notified via a software interrupt.

Because ZooKeeper decided to use the path as an argument in every function call, we need to open up to the possibility of the original znode being deleted and no znode being created in its place with the same name. This is possible to easily ensure using the sequence number of the original znode. ZooKeeper has calls that use such sequence numbers as version numbers. If there is a mismatch, then the function is not executed.

7.3.2 Consistency Guarantees

There are two consistency guarantees that are provided by ZooKeeper.

Linearizable writes All the write operations are linearizable (definition in Section 7.1.2). This definition is however slightly different. It assumes that client requests can be overlapping, which means that at a point of time different requests issued by the same client can be alive. It is necessary to have this assumption because in this case a client is a moving entity that can frequently switch machines. The second condition addresses the issue of consistency among these requests.

FIFO order All the requests from a client are executed in order. This means that if a client issues request A before request B, then the ZooKeeper service executes them in the same order.

These two guarantees are the core of ZooKeeper's consistency model. Note that read operations are processed locally. We need to understand the reason for having these primitives.

Consider an application where we are implementing a broadcast service. The leader process is an OTT platform, which broadcasts the list of shows

to subscriber processes. All of them are clients of the ZooKeeper service. Consider an update operation, where the list of shows for the coming week are uploaded to ZooKeeper by the OTT server. This involves doing multiple write operations where the entire list is uploaded to a group of znodes. Consider this process from the point of view of another subscriber process. It deserves to see an "atomic" view of the entire system. Either an empty list is visible with a message that it will be updated in the future, or the new and complete list is visible. The fact is that znodes are updated incrementally and a client can always view the partial update. However, this is where the consistency guarantees are useful. The OTT server can create a fresh list that is initially unlinked. Once it is fully populated, the pointer to the list of shows can be updated. We can assume that the pointer is stored in a single znode and thus it can be updated atomically. Given ZooKeeper's consistency guarantees this code should work seamlessly.

Next, let us assume that the OTT server issues multiple concurrent requests. The modified definition of linearizability allows this. However, the order in which they appear to execute is the same as the order in which they were issued by the client, which in this case is the OTT server. If the OTT server wants to maintain a strict order of changes seen by the rest of the client, it is quiet easy to achieve because ZooKeeper preserves the FIFO order of requests.

Let us look at a corner case where the clients use separate out-of-band channels. In this case, a client could observe a fresh value and convey the same to another client. The other client may not be able to see the update. This is possible if channels of communication that do not go through ZooKeeper. These are known as out-of-band channels. An easy way to extend consistency to such scenarios is to use the *sync* operation. It will flush all the outstanding updates to the ZooKeeper system. This will ensure that updates are observed globally.

7.3.3 Implementation Details

ZooKeeper relies on an architecture similar to that used by Chubby and Raft. These protocols have a *leader* server that takes requests from clients and atomically broadcasts them to other *follower* servers. They are kept consistent using consensus protocols.

Similar to Raft, each server maintains a list of updates that need to be made to its state machine. The lists do not diverge; this means that at the same index, the same entries are stored. This means that the same update can be applied twice. This is because in the logs an entry is just overwritten

by the same entry. Such updates are idempotent.

ZooKeeper nodes do not suffer from Byzantine faults. They either crash and stop or work correctly. This means that a majority quorum is required to accept an update. There is no need for the quorum to contain at least two-thirds of the nodes as is the case in a Byzantine fault-tolerant system.

Next, note that in the replicated state machine model, the current state can be thought of as a combination of the initial state and a list of state changes (updates). This update log is stored locally at each server. If there is a crash, this log can be used to recover from a fault. It makes little sense to store the entire log and then replay all its entries to arrive at the current state during crash recovery. It is instead a much better idea to store periodic snapshots. This will also shorten the size of the log. It can terminate at the latest snapshot, and can be much shorter. All the servers process write operations in the order that they are sent by the leader. This preserves the FIFO property. Furthermore, linearizability is naturally guaranteed because a single leader broadcasts all the writes.

Fact 7.3.1

Note that a snapshot should not necessarily correspond to a valid state. It needs to be a state where the current state can be obtained after applying all the updates in the update log. Such a snapshot, which is not necessarily valid is known as a *fuzzy snapshot*. It is easy to collect a fuzzy snapshot by just doing a depth first traversal of the data tree. ZooKeeper ensures that by applying the updates in the outstanding update log the current state (which is valid) can be obtained.

Read Requests

Read requests are sent to the servers individually. It is important to understand that the are serviced by the local cache of the servers. Hence, read requests are fast. Note that there is no guarantee of ordering and consistency for reads. It is possible that reads return stale values. If any client wishes to get an up-to-date value, then the correct method is to call the *sync* operation. It ensures that all the pending updates are sent to the server that is connected to the client. Thus the server that is servicing the read has all the values. Any subsequent read returns fresh values. However, *sync* is an expensive operation with high latency and is thus meant to be used sparingly.

There is a need to check if clients or servers have become significantly inconsistent. This is easily achieved by associating a monotonically increasing id with every transaction issued by the leader server. Every follower server maintains the id of the last transaction sent by the leader. This ID can be used to find if different follower servers have updated information or not, or if there is a discrepancy between a client and a server. In that case, the node that has fallen behind can request updates from the node that has more updated information. The FIFO property of writes is beneficial here. If an update is present, then it means that all its previous updates are also there. It is thus easy to transfer logs.

Heartbeat Messages

The leader sends heartbeat messages periodically to the rest of the servers. This ensures that the followers know that the leader is alive. Otherwise, there is a need to elect a new leader. Similarly, clients send heartbeat messages to the servers that they are connected to. If a client goes down, then servers don't receive their heartbeat messages. After a timeout period, the client's session is terminated.

7.4 Quorum-Based Systems

In all the protocols that we have seen up till now, there is an atomic broadcast primitive. The leader server broadcasts all updates to the rest of the servers. The idea is to implement a replicated state machine, which needs to be kept consistent across the servers. All the state machines start from the same initial state. Servers just apply an update to each state to arrive at the next state. A list of updates or rather a log of updates is kept consistent across the servers. This automatically ensures that the final state is the same across the servers. This is a very popular model; however, it is not very scalable. This means that if there are a large number of servers, the protocol may slow down prohibitively.

Hence, there is a need to design mechanisms that avoid broadcast-based communication. The crux of these approaches is to effect updates by multicasting them to a limited set of nodes, known as a *quorum*. It is still possible for an agreement to be established in such a setting. For example, if a majority of nodes are required to decide a value, then no other value can be decided. Two different majority sets can never decide different values because they will have a node in common. This node cannot decide two values. Hence, a majority, in this case, suffices. The size of a quorum is

thus larger than one more than half the number of nodes. Let us further generalize this.

Next, we can define a read quorum and a write quorum. This means that for different kinds of operations, we can have different types of quorums. For writes, if we wish to establish a global order, then it is clear that the size of the write quorum needs to have a majority of the nodes. Then all the nodes will agree on the same order of writes. Next, we would like to create read quorums. The aim is to fetch the value written by the latest write. This can be easily guaranteed if there is at least one node in common.

7.4.1 Quorum Intersection Property

Let us capture this description formally. Let the size of the read quorum be R and the size of the write quorum be W. If the total number of nodes is N, then we need to ensure that $R + W > N$. This will guarantee that there is one node in common between the read and write quorums. If both R and W are at least as large as a majority, i.e., $\lceil \frac{N+1}{2} \rceil$, then this guarantee is trivially satisfied. In another simple case, $W = N$ (broadcast to everybody) and $R = 1$ (read from any server).

There are several complexities here. What if the common node suffers from a fault? It need not suffer from a Byzantine fault. If it just crashes or becomes unresponsive, then it is not possible to guarantee consistency, i.e., a read may not fetch the value of the latest write. This is why there is a need to choose quorums wisely. Nodes can be classified on the basis of their probability to fail. Nodes that are more robust can be placed at the intersection of different quorums. Another approach is to compute the size of quorums dynamically. This means that if nodes fail, quorum sizes can be increased such that there is at least one correctly functioning node in the intersection of any two quorums.

Remark 7.4.1

The quorum intersection property ensures that there is at least one correct node in the intersection of any two quorums.

Unconventional Quorums

Recall the Maekawa's algorithm 4.1.3. Here also quorums were used to ensure that the request sets of two different nodes (processes) have an overlap size that is at least 1. In this case, the processes were arranged as a grid.

A quorum comprises all the processes in the same row and same column. This arrangement ensures that every quorum intersection has at least two processes.

Another way of creating quorums is to designate a few nodes as high-priority and a few nodes as low-priority. High-priority nodes are sent writes immediately and low-priority nodes can spend more time catching up. This means that read operations that are sent to high-priority nodes shall always return fresh values. However, consistency cannot be guaranteed to reads sent to low-priority nodes.

7.4.2 Probabilistic Quorums

In a probabilistic quorum, the quorum intersection property holds probabilistically [Malkhi et al., 1997]. This means that the probability that two quorums intersect at a node is not 100% guaranteed. They have a non-null intersection with a high probability.

Let the set of nodes in the system be $\mathcal{V}$. Let us define a set of quorums $\mathcal{Q}$, where each element is a set of nodes in $\mathcal{V}$. For any two quorums Q_1 and Q_2 in $\mathcal{Q}$, a regular (or strict) quorum system ensures that $Q_1 \cap Q_2 \neq \phi$. This means that the quorum intersection property holds. In a probabilistic quorum system we associate a probability distribution with $\mathcal{Q}$. For $Q_1 \in \mathcal{Q}$, let its probability be $p(Q_1)$.

The following relationship holds.

$$\sum_i f(Q_i) = 1 \tag{7.1}$$

Next, let us define the *load* induced on node v. Let us define $L_f(v) = \sum_{v \in Q_i} f(Q_i)$. If a server (node) v' is shared across a lot of quorums, especially quorums associated with a high probability, $L_f(v')$ will be high. Next, let us extend this definition to all the nodes. $L_f(\mathcal{Q}) = max_{v \in \mathcal{V}} L_f(v)$. It is basically representing the probability of reaching the busiest server. This is clearly dependent on the choice of the function of f. We clearly do not want to create a hot spot or a point of contention. Hence, it is in our best interests to choose a function f that minimizes the load. Hence, let us define the load of the quorum system $\mathcal{Q}$ as $L(\mathcal{Q}) = min_f L_f(\mathcal{Q})$. This definition makes the load independent of the specific choice of f. It is only dependent on the structure of the quorum system.

Next, let us quantify *fault tolerance*. We need to recall that fault tolerance is a fundamental concept in the design of quorum systems. The quorum

intersection property at its simplest mandates a non-null intersection. However, this is useless if the nodes in the intersection of two quorums have crashed. Hence, they should be fault-free. Let us define the available set $A(\mathcal{Q})$ as the smallest set of nodes that intersects every quorum in the quorum system. If the entire available set fails, then every quorum is impacted in the system. This means that if at least one node from every quorum goes down, no quorum is complete. The system thus cannot arrive at an agreement. The *fault tolerance* is defined as the size of the available set.

Relationship between Load and Availability

Let us now look at classical results that establish the relationship between load and availability in strict quorum systems (not probabilistic) (refer to [Naor and Wool, 1994]). Based on results derived from linear programming and relaxation arguments, Naor and Wool proved the following result. Let n be the total number of nodes ($|\mathcal{V}|$).

$$L(\mathcal{Q}) \geq max\left[\frac{1}{s(\mathcal{Q})}, \frac{s(\mathcal{Q})}{n}\right] \tag{7.2}$$

Let $s(\mathcal{Q})$ be the size of the smallest quorum in $\mathcal{Q}$. The authors have derived a few more results.

$$L(\mathcal{Q}) \geq \frac{1}{\sqrt{n}} \tag{7.3}$$

Trivially, $|A(\mathcal{Q})| \leq s(\mathcal{Q}) \leq nL(\mathcal{Q}))$. Hence, we can conclude that $|A(\mathcal{Q})| \leq nL(\mathcal{Q})$. Intuitively, as we decrease the load, the fault tolerance should increase because there is more sharing. The minimum possible load is $\theta(\frac{1}{\sqrt{n}})$.In this case, the fault tolerance (size of $A(\mathcal{Q})$) is $O(\frac{1}{\sqrt{n}})$. This is the best that we can get if we stick to strict quorum systems.

Relationships for Probabilistic Quorum Systems

Let us define a probabilistic quorum. Consider all pairs of quorums Q_1 and Q_2 that have a non-null intersection. Mathematically,

$$\sum_{Q_1, Q_2 : (Q_1 \cap Q_2 \neq \phi)} f(Q_1).f(Q_2) \geq 1 - \epsilon \qquad (0 < \epsilon < 1) \tag{7.4}$$

It is easy to see that in a strict quorum where all pairs of quorums are guaranteed to have a non-null intersection, the value of this summation will exactly be 1. However, in this case, a relaxation is done and the sum is

allowed to go down to $(1 - \epsilon)$. We can interpret this expression to mean that some pairs of quorums are missing. In other words, their intersection is null, which is not allowed in a strict quorum system.

Construction of Probabilistic Quorums

In this system, the construction of a probabilistic quorum is quite straightforward. If there are n nodes in the full system, the quorums are all the sets that contain $\alpha\sqrt{n}$ nodes. α is a constant here. It is possible to extend the mathematical analysis that is used to analyze the classical birthday paradox to prove the following. The probability that no node is present in the intersection of two randomly chosen quorums of size $\alpha\sqrt{n}$ is strictly less than $e^{-\alpha^2}$. Higher the value of α, lower the probability of a null intersection. There is an exponential relationship.

Let us now state a few results without proof. The details are given in reference [Malkhi et al., 1997].

- The load is $O\left(\frac{1}{\sqrt{n}}\right)$.
- In such a quorum system, the failure probability $F_p(\mathcal{Q})$ is defined as the probability of having at least one faulty server in every quorum. This makes the entire system unresponsive. In this case, $F_p(\mathcal{Q}) = e^{-\Omega(n)}$.
- The fault tolerance (size of the available set) is $\Omega(n)$. $\Omega(n)$ does not mean that the size of the available set is more than the number of nodes. It just means that its rate of growth is greater than n.

7.5 Summary and Further Reading

7.5.1 Summary

Summary 7.5.1

1. A distributed system runs many processes in parallel. The synchronization between them is weak given that primitives such barriers and mutual exclusion are not meant to be used frequently.
2. Hence, a distributed system can have many different outcomes. Some of these outcomes may be non-intuitive and may not be allowed in strictly sequential programs. Given that it is hard

to enforce strict operation-level synchronization in large distributed systems, some such outcomes are often allowed in the interest of performance.

3. A consistency model specifies the set of valid outcomes. Every distributed system has an associated consistency model. It needs to conform with its specifications.

4. Linearizability and sequential executions are the strictest models in this space given that they are the closest to a sequential execution on a single thread. Linearizability requires every operation to appear to complete at a unique point between its start and end. Whereas, sequential consistency preserves intra-process order and atomicity (appearance of execution at a unique time instant).

5. Snapshot isolation and causal consistency are weaker in the sense that they allow more outcomes. A good use case of the former is traversing a linked list and then making an update. Causal consistency is good for capturing happens-before events.

6. There are approximate consistency models where the degree of staleness is bounded. Numerical consistency models allow a certain degree of deviation from the ideal value.

7. Eventual consistency is one of the weakest models, where an update eventually reaches the rest of the servers. No guarantees can be made regarding how fresh the values are. All of these consistency models take a system-level view where they look at the outcomes of all the processes.

8. Client-centric consistency models specify the behavior with respect to any individual client.

9. There is a trade-off between the strength of the consistency model, availability (responsiveness) and partition tolerance – it is not possible to achieve all three. This is the essence of the CAP theorem. If there are no faults, then there is a trade-off between latency and consistency.

10. Apache ZooKeeper is a very popular data storage system that guarantees linearizability and per-client FIFO order for writes.

It supports fast reads that are serviced from the local cache. Hence, for reads the consistency guarantees are much weaker. Sequential consistency for reads is typically provided.

11. Most protocols broadcast writes to all the servers to maintain consistency. However, this is not strictly required. Multicasting a read or write operation to a subset of nodes is often good enough. Such a subset is called a quorum. There are many quorum-based protocols for managing data stores that provide different degrees of consistency.

12. In general, if R is the size of the read quorum and W is the size of the write quorum, then $R + W > N$. Here, N is the total number of nodes (or processes) in the system.

13. Quorum-based systems observe the quorum intersection property that states that any two quorums have at least one node in common. Sometimes an additional caveat is added – at least one of the nodes that lie in the intersection needs to be fault-free.

14. There are systems that define quorums probabilistically. They can make better stochastic guarantees about fault tolerance as compared to regular quorum-based systems.

7.5.2 Further Reading

Consistency requirements are rigorously defined in transactional systems such as databases. However, in generic distributed systems, consistency guarantees need to be provided for all operations. The survey paper by Viotti et al. [Viotti and Vukolić, 2016] provides a comprehensive overview of numerous consistency models. Most of the client-centric consistency models are presented in the paper by Terry et al. [Terry et al., 1994]. For continuous consistency models and the limits of eventual consistency, readers can refer to the paper by Yu et al. [Haifeng and Amin, 2000] and Peter et al. [Bailis and Ghodsi, 2013], respectively.

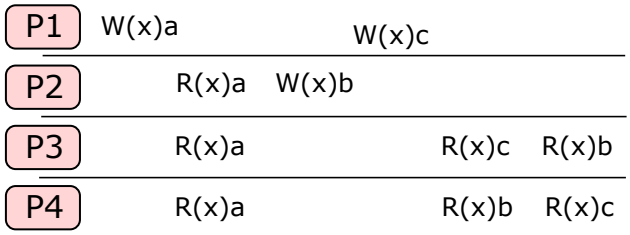
In general, consistency models are defined for the entire system where all the nodes participate. However, it is possible to design quorum-based systems where operations need to performed only by the quorum (subset of servers). If all quorums have a non-null intersection at a correct node, then consistency guarantees can be made. It is possible to design probabilistic

quorum systems that have a bounded probability of inconsistency (refer to [Malkhi et al., 1997]). Refer to Naor et al. [Naor and Wool, 1994] and Stellar [García-Pérez and Schett, 2019] for methods to construct quorum systems.

Let us now discuss references related to applications. CockroachDB [Taft et al., 2020] shows the design of a large distributed SQL layer that provides consistent transactions. Readers can also refer to Spanner [Corbett et al., 2013], Cosmos DB [Nuriev and Lapteva, 2024] and BookKeeper [Junqueira et al., 2013], which are important proposals in this space. BookKeeper provides strong consistency with write-ahead logging and quorum-based replication. Azure Cosmos DB provides five different consistency levels. In other words, its consistency guarantee is tunable.

Questions

Question 7.1. Is the following sequence of events allowed with a sequentially-consistent data store? What about a causally-consistent data store? Explain your answer.



[W(x)a means that we write a to location x, and R(x)b means that we read the value b from location x]

Question 7.2. What is the difference between data-centric and client-centric consistency models.

Design Questions

Question 7.3. We want to implement a large distributed computer consisting of many individual nodes. The nodes are connected by ethernet links. We want all the nodes to see the same view of memory. Clearly, we cannot make any changes in the hardware. At a software level, we need to ensure that we create an illusion of a distributed shared memory. Discuss the following points:

(a) The design of the distributed shared memory.

(b) Coherence and consistency requirements.

(c) Scalability

(d) Extend the scheme to give all processes the same view of I/O devices.

Bibliography

[Abadi, 2012] Abadi, D. (2012). Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42.

[Alon et al., 1986] Alon, N., Babai, L., and Itai, A. (1986). A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583.

[Arora et al., 2000] Arora, A., Kulkarni, S., and Demirbas, M. (2000). Resettable vector clocks. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 269–278.

[Aublin et al., 2015] Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2015). The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):1–45.

[Awerbuch et al., 1989] Awerbuch, B., Goldberg, A. V., Luby, M., and Plotkin, S. A. (1989). Network decomposition and locality in distributed computation. In *30th Annual Symposium on Foundations of Computer Science*, pages 364–369. IEEE Computer Society.

[Awerbuch and Peleg, 1990] Awerbuch, B. and Peleg, D. (1990). Sparse partitions. In *31st Annual Symposium on Foundations of Computer Science*, pages 503–513. IEEE Computer Society.

[Bacon et al., 2017] Bacon, D. F., Bales, N., Bruno, N., Cooper, B. F., Dickinson, A., Fikes, A., Fraser, C., Gubarev, A., Joshi, M., Kogan, E., Lloyd, A., Melnik, S., Rao, R., Shue, D., Taylor, C., van der Holst, M., and Woodford, D. (2017). Spanner: Becoming a SQL system. In Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., and Suciu, D., editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 331–343. ACM.

[Bacrach et al., 2019] Bacrach, N., Censor-Hillel, K., Dory, M., Efron, Y., Leitersdorf, D., and Paz, A. (2019). Hardness of distributed optimization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, page 238–247.

[Bailis and Ghodsi, 2013] Bailis, P. and Ghodsi, A. (2013). Eventual consistency today: Limitations, extensions, and beyond: How can applications be built on eventually consistent infrastructure given no guarantee of safety? *Queue*, 11(3):20–32.

[Balliu et al., 2019] Balliu, A., Brandt, S., Hirvonen, J., Olivetti, D., Rabie, M., and Suomela, J. (2019). Lower bounds for maximal matchings and maximal independent sets. In Zuckerman, D., editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 481–497. IEEE Computer Society.

[Barenboim and Elkin, 2010] Barenboim, L. and Elkin, M. (2010). Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Comput.*, 22(5-6):363–379.

[Barenboim et al., 2016] Barenboim, L., Elkin, M., Pettie, S., and Schneider, J. (2016). The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45.

[Beame et al., 2013] Beame, P., Koutris, P., and Suciu, D. (2013). Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 273–284. ACM.

[Borthakur et al., 2008] Borthakur, D. et al. (2008). Hdfs architecture guide. *Hadoop apache project*, 53(1-13):2.

[Brady and Cowen, 2006] Brady, A. and Cowen, L. J. (2006). Compact routing on power law graphs with additive stretch. In Raman, R. and Stallmann, M. F., editors, *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*, pages 119–128. SIAM.

[Burrows, 2006] Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350.

[Carlsson and Gustavsson, 2001] Carlsson, B. and Gustavsson, R. (2001). The rise and fall of napster-an evolutionary approach. In *International Computer Science Conference on Active Media Technology*, pages 347–354. Springer.

[Castro et al., 1999] Castro, M., Liskov, B., et al. (1999). Practical byzantine fault tolerance. In *OSDI*.

[Chan and Shi, 2020] Chan, B. Y. and Shi, E. (2020). Streamlet: Textbook streamlined blockchains. In *Advances in Financial Technologies*.

[Chand et al., 2016] Chand, S., Liu, Y. A., and Stoller, S. D. (2016). Formal verification of multi-paxos for distributed consensus. In *International Symposium on Formal Methods*, pages 119–136. Springer.

[Clarke et al., 2001] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2001). Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies: international workshop on design issues in anonymity and unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*, pages 46–66. Springer.

[Corbett et al., 2013] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W. C., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. (2013). Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press.

[Cowen, 2001] Cowen, L. (2001). Compact routing with minimum stretch. *J. Algorithms*, 38(1):170–183.

[Cristian, 1989] Cristian, F. (1989). Probabilistic clock synchronization. *Distributed computing*, 3:146–158.

[Danezis et al., 2022] Danezis, G., Kokoris-Kogias, L., Sonnino, A., and Spiegelman, A. (2022). Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220.

[Demers et al., 1987] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12.

[Dijkstra, 1965] Dijkstra, E. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569. Cited by: 636; All Open Access, Bronze Open Access.

[Dijkstra and Scholten, 1980] Dijkstra, E. W. and Scholten, C. (1980). Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4.

[Douglas, 2004] Douglas, G. (2004). Copyright and peer-to-peer music file sharing: the napster case and the argument against legislative reform. *Murdoch University Electronic Journal of Law*, 11(1).

[Dwork et al., 1988] Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323.

[Ellingson and Kulpinski, 1973] Ellingson, C. and Kulpinski, R. (1973). Dissemination of system time. *IEEE Transactions on Communications*, 21(5):605–624.

[Eugster et al., 2003] Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P., and Kermarrec, A.-M. (2003). Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374.

[Facebook, 2020] Facebook (2020). https://engineering.fb.com/2020/03/18/production-engineering/ntp-service/.

[Fidge, 1988] Fidge, C. (1988). Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66.

[Fischer et al., 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382.

[Forum, 2024] Forum, G. D. (2024). The annotated gnutella protocol specification v0.4. `https://rfc-gnutella.sourceforge.net/developer/stable/`. Accessed on 26^{th} June 2024.

[Foster and Kesselman, 1998] Foster, I. and Kesselman, C. (1998). The globus project: A status report. In *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*, pages 4–18. IEEE.

[Fowler and Zwaenepoel, 1990] Fowler, J. and Zwaenepoel, W. (1990). Causal distributed breakpoints. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, pages 134–141. IEEE Computer Society.

[Gallager et al., 1979] Gallager, R. G., Humblet, P. A., and Spira, P. M. (1979). *A distributed algorithm for minimum weight spanning trees*. Citeseer.

[Gallager et al., 1983] Gallager, R. G., Humblet, P. A., and Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77.

[García-Pérez and Schett, 2019] García-Pérez, Á. and Schett, M. A. (2019). Deconstructing stellar consensus (extended version). *arXiv preprint arXiv:1911.05145*.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman.

[Geng et al., 2018] Geng, Y., Liu, S., Yin, Z., Naik, A., Prabhakar, B., Rosenblum, M., and Vahdat, A. (2018). Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94.

[Ghaffari, 2019] Ghaffari, M. (2019). Massively parallel algorithms. `http://people.csail.mit.edu/ghaffari/MPA19/Notes/MPA.pdf`.

[Ghaffari et al., 2021] Ghaffari, M., Grunau, C., and Rozhon, V. (2021). Improved deterministic network decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2904–2923. SIAM.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43.

[Ghodsi, 2006] Ghodsi, A. (2006). *Distributed k-ary system: Algorithms for distributed hash tables*. PhD thesis, KTH.

[Gilad et al., 2017] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. (2017). Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68.

[Gilbert and Lynch, 2002] Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59.

[Golab, 2018] Golab, W. (2018). Proving pacelc. *ACM SIGACT News*, 49(1):73–81.

[Goodrich et al., 2011] Goodrich, M. T., Sitchinava, N., and Zhang, Q. (2011). Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer.

[Gueta et al., 2019] Gueta, G. G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D.-A., Tamir, O., and Tomescu, A. (2019). Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE.

[Gusella and Zatti, 1983] Gusella, R. and Zatti, S. (1983). Tempo: A network time controller for a distributed berkeley unix system. Technical report, Berkeley.

[Hadzilacos, 2001] Hadzilacos, V. (2001). A note on group mutual exclusion. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 100–106.

[Haifeng and Amin, 2000] Haifeng, Y. and Amin, V. (2000). Design and evaluation of a continuous consistency model for replicated services. In *4th Symposium on Operating Systems 4th Symposium on Operating Systems Design and Implementation*.

[Hawblitzel et al., 2015] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. (2015). Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17.

[Herlihy et al., 2020] Herlihy, M., Shavit, N., Luchangco, V., and Spear, M. (2020). *The art of multiprocessor programming*. Newnes.

[Herlihy and Wing, 1990] Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

[Howe, 2000] Howe, A. J. (2000). Napster and gnutella: a comparison of two popular peer-to-peer protocols. *Universidade de Victoria*, 11:29.

[Islam et al., 2014] Islam, M. T., Akon, M., Abdrabou, A., and Shen, X. (2014). Modeling epidemic data diffusion for wireless mobile networks. *Wireless Communications and Mobile Computing*, 14(7):745–760.

[Izal et al., 2004] Izal, M., Urvoy-Keller, G., Biersack, E. W., Felber, P. A., Al Hamra, A., and Garces-Erice, L. (2004). Dissecting bittorrent: Five months in a torrent's lifetime. In *Passive and Active Network Measurement: 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004. Proceedings 5*, pages 1–11. Springer.

[Jard and Jourdan, 1994] Jard, C. and Jourdan, G. (1994). Dependency tracking and filtering in distributed computations. Technical Report 851, IRISA: Publications internes: Institut de Recherche en Informatique et Systèmes Aléatoires.

[Jayanti and Toueg, 1992] Jayanti, P. and Toueg, S. (1992). Some results on the impossibility, universality, and decidability of consensus. In *International Workshop on Distributed Algorithms*, pages 69–84. Springer.

[Jelasity et al., 2007] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and Van Steen, M. (2007). Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8–es.

[Junqueira and Reed, 2013] Junqueira, F. and Reed, B. (2013). *ZooKeeper: distributed process coordination.* " O'Reilly Media, Inc.".

[Junqueira et al., 2013] Junqueira, F. P., Kelly, I., and Reed, B. (2013). Durability with bookkeeper. *ACM SIGOPS operating systems review*, 47(1):9–15.

[Jurdziński and Nowicki, 2018] Jurdziński, T. and Nowicki, K. (2018). Mst in o (1) rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2620–2632. SIAM.

[Kaashoek and Karger, 2003] Kaashoek, M. F. and Karger, D. R. (2003). Koorde: A simple degree-optimal distributed hash table. In *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003. Revised Papers 2*, pages 98–107. Springer.

[Karger et al., 1997] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663.

[Karloff et al., 2010] Karloff, H. J., Suri, S., and Vassilvitskii, S. (2010). A model of computation for MapReduce. In *Proc. ACM SPAA*, pages 938–948.

[Klauck et al., 2015a] Klauck, H., Nanongkai, D., Pandurangan, G., and Robinson, P. (2015a). Distributed computation of large-scale graph problems. In Indyk, P., editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 391–410. SIAM.

[Klauck et al., 2015b] Klauck, H., Nanongkai, D., Pandurangan, G., and Robinson, P. (2015b). Distributed computation of large-scale graph problems. In Indyk, P., editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 391–410. SIAM.

[Kothapalli and Pemmaraju, 2011] Kothapalli, K. and Pemmaraju, S. V. (2011). Distributed graph coloring in a few rounds. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing*, pages 31–40. ACM.

[Kothapalli et al., 2006] Kothapalli, K., Scheideler, C., Onus, M., and Schindelhauer, C. (2006). Distributed coloring in $O(\sqrt{logn})$–bit rounds. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*. IEEE.

[Kulkarni et al., 2014] Kulkarni, S. S., Demirbas, M., Madappa, D., Avva, B., and Leone, M. (2014). Logical physical clocks. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Proceedings*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. Springer.

[Lamport, 1974] Lamport, L. (1974). A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453 – 455.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[Lamport, 1987] Lamport, L. (1987). A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11.

[Lamport, 1998] Lamport, L. (1998). The part-time parliament. acm transactions on computer systems.

[Lamport, 2001] Lamport, L. (2001). Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58.

[Lamport et al., 2019] Lamport, L., Shostak, R., and Pease, M. (2019). The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226.

[Lenzen, 2013] Lenzen, C. (2013). Optimal deterministic routing and sorting on the congested clique. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 42–50. ACM.

[Lesniewski-Laas and Kaashoek, 2010] Lesniewski-Laas, C. and Kaashoek, M. F. (2010). Whanau: A sybil-proof distributed hash table. In *NSDI*.

[Lewin, 1998] Lewin, D. M. (1998). *Consistent hashing and random trees: Algorithms for caching in distributed networks*. PhD thesis, Massachusetts Institute of Technology.

[Li et al., 2013] Li, T., Zhou, X., Brandstatter, K., Zhao, D., Wang, K., Rajendran, A., Zhang, Z., and Raicu, I. (2013). Zht: A light-weight

reliable persistent dynamic scalable zero-hop distributed hash table. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 775–787. IEEE.

[Linial, 1992] Linial, N. (1992). Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201.

[Liskov and Ladin, 1986] Liskov, B. and Ladin, R. (1986). Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 29–39.

[Lougheed and Rekhter, 1989] Lougheed, K. and Rekhter, Y. (1989). Rfc1105: Border gateway protocol (bgp).

[Luby, 1986] Luby, M. (1986). A simple parallel algorithm for the maximal independent set problem. *SIAM J. Computing*, 15(4):1036–1053.

[Luk and Wong, 1997] Luk, W.-S. and Wong, T.-T. (1997). Two new quorum based algorithms for distributed mutual exclusion. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 100–106. IEEE.

[Lundelius and Lynch, 1984] Lundelius, J. and Lynch, N. (1984). An upper and lower bound for clock synchronization. *Information and control*, 62(2-3):190–204.

[Lynch, 1996] Lynch, N. A. (1996). *Distributed algorithms*. Elsevier.

[Malkhi et al., 1997] Malkhi, D., Reiter, M., and Wright, R. (1997). Probabilistic quorum systems. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 267–273.

[Marzullo, 1984] Marzullo, K. A. (1984). *Maintaining the time in a distributed system: an example of a loosely-coupled distributed service (synchronization, fault-tolerance, debugging)*. PhD thesis, Stanford University.

[Mathur et al., 2022] Mathur, U., Pavlogiannis, A., Tunç, H. C., and Viswanathan, M. (2022). A tree clock data structure for causal orderings in concurrent executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 710–725.

[Mattern et al., 1988] Mattern, F. et al. (1988). *Virtual time and global states of distributed systems*. Univ., Department of Computer Science.

[Maymounkov and Mazieres, 2002] Maymounkov, P. and Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *International workshop on peer-to-peer systems*, pages 53–65. Springer.

[Mazieres, 2015] Mazieres, D. (2015). The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45.

[Mills, 1985] Mills, D. (1985). Network Time Protocol (NTP). RFC 958.

[Moraru et al., 2013] Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372.

[Naor and Wool, 1994] Naor, M. and Wool, A. (1994). The load, capacity and availability of quorum systems. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 214–225. IEEE.

[Nuriev and Lapteva, 2024] Nuriev, M. and Lapteva, M. (2024). Facilitating efficient energy distribution and storage: The role of data consistency technologies in azure cosmos db. In *E3S Web of Conferences*, volume 541, page 02003. EDP Sciences.

[Ongaro and Ousterhout, 2014] Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319.

[Ousterhout et al., 1988] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. (1988). The sprite network operating system. *Computer*, 21(2):23–36.

[Özkasap et al., 2010a] Özkasap, Ö., Çağlar, M., Yazıcı, E. Ş., and Küçükçifçi, S. (2010a). An analytical framework for self-organizing peer-to-peer anti-entropy algorithms. *Performance Evaluation*, 67(3):141–159.

[Özkasap et al., 2010b] Özkasap, Ö., Çağlar, M., Yazıcı, E. Ş., and Küçükçifçi, S. (2010b). An analytical framework for self-organizing peer-to-peer anti-entropy algorithms. *Performance Evaluation*, 67(3):141–159.

[Panconesi and Srinivasan, 1996] Panconesi, A. and Srinivasan, A. (1996). On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):356–374.

[Pandurangan et al., 2016] Pandurangan, G., Robinson, P., and Scquizzato, M. (2016). Fast distributed algorithms for connectivity and MST in large graphs. In Scheideler, C. and Gilbert, S., editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 429–438. ACM.

[Peleg, 2000] Peleg, D. (2000). *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, USA.

[Pittel, 1987] Pittel, B. (1987). On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223.

[Pouwelse et al., 2008] Pouwelse, J. A., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D. H., Reinders, M., Van Steen, M. R., and Sips, H. J. (2008). Tribler: a social-based peer-to-peer system. *Concurrency and computation: Practice and experience*, 20(2):127–138.

[Pozzetti and Kshemkalyani, 2020] Pozzetti, T. and Kshemkalyani, A. D. (2020). Resettable encoded vector clock for causality analysis with an application to dynamic race detection. *IEEE Transactions on Parallel and Distributed Systems*, 32(4):772–785.

[Raynal and Taubenfeld, 2022] Raynal, M. and Taubenfeld, G. (2022). A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, 919:47–65.

[Reed and Dongarra, 2015] Reed, D. and Dongarra, J. (2015). Exascale computing and big data. *Communication of the ACM*, 58(7):56–68.

[Rowstron and Druschel, 2001] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*, pages 329–350. Springer.

[Rozhon and Ghaffari, 2020] Rozhon, V. and Ghaffari, M. (2020). Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 350–363. ACM.

[Sanders, 1987] Sanders, B. A. (1987). The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems (TOCS)*, 5(3):284–299.

[Sarangi, 2023] Sarangi, S. R. (2023). *Next-Gen Computer Architecture*. White Falcon, 1st edition edition.

[Scharf, 2011] Scharf, N. (2011). Napster's long shadow: copyright and peer-to-peer technology. *Journal of Intellectual Property Law & Practice*, 6(11):806–812.

[Schmuck, 1988] Schmuck, F. B. (1988). The use of efficient broadcast protocols in asynchronous distributed systems. Technical report, Cornell University.

[Singh et al., 2009] Singh, A., Fonseca, P., Kuznetsov, P., Rodrigues, R., Maniatis, P., et al. (2009). Zeno: Eventually consistent byzantine-fault tolerance. In *NSDI*, volume 9, pages 169–184.

[Singhal, 1993] Singhal, M. (1993). A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101.

[Singhal and Kshemkalyani, 1992] Singhal, M. and Kshemkalyani, A. D. (1992). An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1):47–52.

[Singhal and Shivaratri, 1994] Singhal, M. and Shivaratri, N. G. (1994). *Advanced concepts in operating systems*. McGraw-Hill, Inc.

[Sivasubramanian, 2012] Sivasubramanian, S. (2012). Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 729–730. ACM.

[Stallings, 2006] Stallings, W. (2006). *Cryptography and network security, 4/E*. Pearson Education India.

[Stoica et al., 2003] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32.

[Taft et al., 2020] Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., et al. (2020). Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509.

[Tanenbaum et al., 1990] Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G. J., and Mullender, S. J. (1990). Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63.

[Tel, 1999] Tel, G. (1999). *Introduction to distributed algorithms*. Cambridge university press.

[Terry et al., 1994] Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M. J., Theimer, M. M., and Welch, B. B. (1994). Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE.

[Thain et al., 2005] Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356.

[Thorup and Zwick, 2001] Thorup, M. and Zwick, U. (2001). Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA*, pages 1–10. ACM.

[tous droits réservés, 2023] tous droits réservés, B. (2023). Bureau International des Podis et Mesures. https://www.bipm.org/en/.

[Valiant, 1990] Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111.

[Van Renesse et al., 2008] Van Renesse, R., Dumitriu, D., Gough, V., and Thomas, C. (2008). Efficient reconciliation and flow control for anti-entropy protocols. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–7.

[Viotti and Vukolić, 2016] Viotti, P. and Vukolić, M. (2016). Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):1–34.

[Walls and Gagnepain, 1992] Walls, F. and Gagnepain, J.-J. (1992). Environmental sensitivities of quartz oscillators. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 39(2):241–249.

[Wattenhofer, 2014] Wattenhofer, R. (2014). Chapter 7: Maximal independent set. https://disco.ethz.ch/courses/fs14/podc/lecture/chapter7.pdf. Accessed: October 23, 2024.

[Wuu and Bernstein, 1984] Wuu, G. T. and Bernstein, A. J. (1984). Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242.

[Yin et al., 2019] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. (2019). Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM symposium on principles of distributed computing*, pages 347–356.

Index