Probabilistic Sequential Consistency in Social Networks

Priyanka Singla
Computer Sc. and Automation
IISc Bangalore, India
priyanka.singla6@gmail.com

Shubhankar S. Singh
Computer Sc. and Engg.
IIT Delhi, India
shubhankar@cse.iitd.ac.in

K. Gopinath
Computer Sc. and Automation
IISc Bangalore, India
gopi@csa.iisc.ac.in

Smruti R. Sarangi
Computer Sc. and Engg.
IIT Delhi, India
srsarangi@cse.iitd.ac.in

***Abstract*—Researchers have proposed numerous consistency models in distributed systems that offer higher performance than classical sequential consistency (SC). Even though these models do not guarantee sequential consistency; they either behave like an SC model under certain restrictive scenarios, or ensure SC behavior for a part of the system. We propose a different line of thinking where we try to accurately estimate the number of SC violations, and then try to adapt our system to optimally tradeoff performance, resource usage, and the number of SC violations. In this paper, we propose a generic theoretical model that can be used to analyze systems that are comprised of multiple sub-domains – each sequentially consistent. It is validated with real world measurements. Next, we use this model to propose a new form of consistency called social consistency, where socially connected users perceive an SC execution, whereas the rest of the users need not. We create a prototype social network application and implement it on the Cassandra key-value store. We show that our system has $2.4\times$ more throughput than Cassandra and provides 37% better quality-of-experience.**

***Keywords*-Hot-spot; social; load balancing; quality-of-experience; distributed; consistency violations.**

I. Introduction

The CAP theorem creates a tradeoff between consistency, availability, and tolerance to partitions. For example, it is not possible to create a system that is sequentially consistent, and at the same time is highly available and immune to network partitions. As a result in large commercial deployments [1], [2] engineers do not explicitly aim for sequential consistency (SC). Instead, they use a consistency model that is somewhere between sequential and eventual consistency [3], [4]. For example in Dynamo [1], the authors try to ensure that no writes are lost from the shopping cart at the time of checkout (eventual consistency); however, it is possible to see different views of the cart throughout the shopping process.

The field of non-SC models that have different correctness properties is very extensive. Most of the related work has focused on models that behave as SC given the unique constraints of the problem [5], or SC is relaxed under known and predictable conditions [1]. In this work, we look at a third line of work known as *probabilistically bounded staleness* [6]. In this model, we can place bounds on the probability of SC violations for a given window of time. Researchers have mainly looked at such models in the context of key-value stores. The line of reasoning is that some applications can tolerate violations in SC; therefore, the aim is to limit the number of such violations so that the users can perceive a good quality-of-experience. Our aim in this paper is to extend this line of work and introduce a new consistency model and accompanying theoretical fundamentals. We call our model *social consistency*.

We focus on social networking applications that try to provide a sequentially consistent view on a best-effort basis. These applications have dynamic workloads and often observe spikes in the incoming traffic [7], [8], resulting in hot-spots. For example, a particular post on a social-networking site by a celebrity [7] can result in a deluge of requests, possibly overwhelming the system. Current systems handle this by either giving up performance or consistency. In such cases, our model tries to group users based on social ties and creates minimally overlapping subsets of users. Users within each such subset perceive sequential consistency; however, users across groups see an eventually consistent execution. In particular, an object is initially stored on a single server and is accessed by all the users, who see a consistent value of the object. Upon overload, this hot object is cloned and stored on another server, and the user set is partitioned into 2 groups (clusters) on basis of social relationships among the users. These clusters are one-to-one mapped to cloned servers. The clones diverge as time progresses and we call them *splits.* If the request rate increases further, new splits are created resulting in more sub-clusters. Only when the request rate decreases, these clusters are combined back and the *splits* are merged in an application dependent manner [1], [9].

In addition, for a given system we outline a modeling methodology where we can estimate the number of SC violations as a function of the number of requests and make our system react accordingly.

We argue about the acceptability of social consistency on the basis of the following observation in existing social networking sites like Facebook/Twitter. We are primarily interested in the updates that our friends are posting, and in general, we would like to see our friends' posts, followed by the comments on it. This is typical SC behavior. However, when we are seeing the Facebook wall of a stranger, and we see a comment to a post without seeing the post, this

is a violation. We wish to minimize such incidents. We argue that because of the natural limitations imposed by the CAP theorem this situation is not completely unavoidable. However, if we can estimate such occurrences and adapt our system, then we can control the quality of experience to a reasonable extent.

In specific, our main contributions are:

- We define a new model called *social consistency*, which offers a client-centric view of consistency.
- We present a split-merge algorithm to handle a simple scenario: single object overloads.
- We introduce a new consistency metric, *quality*, and use it to compare our model with various existing systems using a basic prototype of a social networking service.
- We develop the mathematical foundations for simulating such a system and create a framework that allows us to reason about the behavior of such systems using simple Monte Carlo simulations. The results of these simulations can be used to tune our system to reduce the number of SC violations.

The paper is organized as follows: Section II discusses the related work. Section III provides a detailed description of our social consistency model. Section IV presents an insight into the mathematical framework for computing the SC violation probability. Section V presents the architecture of the system and then describes the split-merge protocols. Section VI implements and evaluates a prototype based on our model, and finally, Section VII concludes the paper. Please note that the terms clients and users have been used interchangeably throughout the paper.

II. Background and Related Work

Achieving low latency along with strong consistency is difficult, and the situation becomes acute in an overloaded system. Various approaches proposed to deal with such systems can be categorized as follows:

Admission Control and Load Balancing Techniques: Randles et al. [10] use work-stealing based techniques to steal work from overloaded nodes. Authors in [11] rely on random sampling for distributing jobs to servers and performing load balancing. These approaches completely ignore the relationships between clients, thus it is tough to ensure a good QoE for the clients.

Data or Metadata Partitioning, and Replication: Amazon Dynamo [1] and Cassandra [9] use consistent hashing to ensure a uniform key distribution. However, this goal of uniform distribution might fail if the access distribution is highly skewed, resulting in a single key getting overloaded; accesses to this key cannot be partitioned across the nodes.

Distributed file systems such as NFS [12] suffer from hot-spots when multiple clients simultaneously open the same file. Weil et al. [13] handle overloads by file system subtree migration and replication. They [14] extend this work and perform load balancing by randomly forwarding metadata requests within a metadata server cluster. These methods of random replication and forwarding lead to inefficient resource utilization. In contrast, our approach carefully chooses the split-merge servers and handles client requests on the basis of social relationships among them. Similar to our approach, Bayou [15] has per object split-merge, however it does not form clusters based on social relationships.

Relaxed Consistency Models: Lloyd et al. [4] propose a novel system with low latency and causal consistency. However, the main drawback of this work is that it is very specific, and it is tough to enforce causality all the time, and in all the scenarios. Some systems [5], [16] rely on the commutativity of operations for ensuring different kinds of consistency (strong and weak) at different times. We, on the other hand, provide two different levels of consistency by partitioning clients into clusters based on their social distance: clients within the cluster are strongly consistent while across the clusters are eventually consistent.

Partitioning the Set of Users: Although the idea of partitioning users and identifying clusters in social networks has been explored in a significant body of research [17], these works focus on other goals, and unlike us, they do not propose to handle overloads. For example, SPAR [17] uses social partitioning to minimize replication and in storing the entire data of clients in a single cluster. Our work, on the other hand, is independent of client placement and follows an object based lightweight split at runtime (during overloads). Walter [18] is a geo-replicated key-value store, offering linearizability for replicas deployed in the same data center and weak consistency across different data centers. This can be considered as a special case of our architecture, where consistency is dependent on physical proximity. Our model is more generic.

Bounding the Eventual consistency: Various relaxed consistency models provide bounds on sequential consistency violations. Bailis et al. [6] proposed a Probabilistic Bounded Staleness (PBS) model to quantify the bounds on the time (t-visibility) when a particular update becomes visible to all the clients. This work also provides bounds on the version (k-staleness) of the object which will be visible to the clients at any time. Liu et al. [19] present a formal probabilistic model to quantify the consistency guarantees achieved by Cassandra. Their formalization is not generic to any eventual consistency model but is specific for Cassandra. Also, they do not provide any analysis/model for other metrics like performance. In contrast, our model provides a mathematical framework for any eventually consistent system along with performance and quality metrics.

III. Social Consistency

A weak consistency model can be obtained from a strong consistency model by applying various program order relaxations for operation pairs accessing different locations [20].

Table I: Valid relaxations

Relaxation	Y/N
W → R/W Order	Y
R → R/W Order	Y
Read Own Write Early	N
Read Other's Write Early	N
Read Own Cluster's Write Early	Y
Read Other Cluster's Write Early	N

Table II: Example of social consistency

Time	P1	P2	P3
$t=0$			$w_3(x)$ = 3
$t=1$	$w_1(x)$ = 1		
$t=2$		$w_2(x)$ = 2	
$t=3$	$r_1(x)$ = [1,2]	$r_2(x)$ = [1,2]	$r_3(x)$ = [3]

Different models include relaxations from the following set of relaxations: a) Write to Read b) Write to Write c) Read to Read d) Read to Write e) Read own write early f) Read others' write early. For our socially consistent system, we introduce a new relaxation: any client can read the writes within its cluster earlier as compared to the clients in the other clusters (*Read own cluster's write early*). Table I lists the relaxations that are valid in our social consistency model.

We describe these relaxations by an example for a generic system defined as a tuple Γ=$\langle C, O, r, w, \varrho \rangle$:

- C=$\{c_1, c_2, ..., c_n\}$ is a finite set of n clients in the system.
- O=$\{o_1, o_2, ..., o_k\}$ is a finite set of k objects on which operations are performed. The object's value is a $list$ and is denoted by a square bracket [].
- r denotes a read and $r_c(o)$=V represents a read request issued by a client c to an object o, which fetches the entire state V (list of values) of the object o, atomically.
- w represents a write operation and $w_c(o)$=v appends the value v to the list V maintained by object o.
- *A partition function* $\varrho : C \times O \rightarrow Cl$, where Cl is the set of all cluster ids ($\{Cl_1, Cl_2, ..., Cl_k\}$). The function maps the client id to the corresponding cluster depending on the overloaded object.

Example: Let x be a shared object in a system with three clients $\{P1, P2, P3\}$ accessing it. The three clients issue a write request to the object x (see Table II); they write (1), (2), and (3) respectively (at different times). Now while reading, the clients P1 and P2 read the value [1,2], while $P3$ gets [3]. As all the clients do not read a sequentially consistent value, the system is not sequentially consistent. However, this result is valid in the case of social consistency: if we consider the clients $P1$ and $P2$ to be in a single cluster and client $P3$ to be in the other cluster. Then, both $P1$ and $P2$ will be able to read the writes of each other early, but they cannot read the updates from the other cluster containing $P3$. It should be noted that in the socially consistent model, all the three clients will eventually read the same value, which happens upon a merge.

A. Formal Definition

Let the set of socially consistent executions for a given program be E. An execution $e \in E$ is socially consistent if the following conditions hold:

Condition 1: Within a cluster, all the writes are atomic. Writes are not visible atomically to nodes outside of a cluster.

Condition 2: All writes are *eventually* visible across all clusters.

Condition 3: Within a cluster, we obey sequential consistency, where nodes within the cluster can read the writes of another node (in the cluster) before they are visible to nodes in other clusters.

B. Partitioning the Set of Clients (ϱ)

The set, C, of clients accessing the object is represented as a weighted graph G=$G(C, E)$, where E represents the set of weights (connectivity strength among the clients). A larger edge weight represents strongly connected clients. The edge weights broadly have two types of components: *static weight (SW)*- provided by the application, and *dynamic weight (DW)*- computed during execution depending upon the intensity of communication among clients. Along with weighted edges, we also consider the graph to have weighted nodes, with only a dynamic component (*dynamic node weight–DNW*) which indicates a node's contribution to the overall load of the system.

Upon interaction between two clients, we have:

$$DW_{new} = 1 + f(\Delta t) * DW_{old} \quad (1)$$

where DW_{old} represents the previous dynamic edge weight, which loses its contribution with time according to a damping function, $f(\Delta t)$. Δt is the time difference between two consecutive interactions between these two clients. The value of this function should be close to 1 for small Δt and should decrease with an increase in Δt; we choose a variant of the Sigmoid function, $f(\Delta t) = 1/(1 + e^{\Delta t - b})$, where b is a tunable parameter. This pattern appropriately captures (observed empirically) the client behavior such that if Δt is high, then the corresponding clients are not interacting much and hence their edge weight should be small.

There can be some clients who send a disproportionate number of requests. Such clients should be placed in separate partitions; otherwise, the partition with these clients will remain overloaded, thus nullifying the overall effect of partitioning. Our system supports this by trading off social consistency for load balancing. We assign weights to the nodes, **dynamic node weights** (DNW), which vary with time and represent each node's contribution in overloading the system. Whenever a client *writes*, her DNW is computed by using a formula similar to Equation 1, but with a different value of b.

We use a weighted matrix, $\mathcal{W}$, to represent the edge weights and the node weights. A diagonal entry, $\mathcal{W}[c_i c_i]$, represents the node weight of client c_i, and a non-diagonal

entry, $\mathcal{W}[c_i c_j]$, represents the total edge weight between clients c_i and c_j. These weights are computed as follows:

$$\mathcal{W}[c_i c_j] = \alpha * SW + \beta * DW \quad (2)$$

$$\mathcal{W}[c_i c_i] = \gamma * DNW \quad (3)$$

where α, β, and γ are the normalization parameters and are tunable. They are provided by the application according to its requirement.

After a partition, the clients across the clusters do not see each other's data. But there might be certain clients who would wish to see all the data at the cost of higher latency (e.g. the owner of the post) and would like to be in both the clusters. We call such clients as *"global nodes"*, and in our current prototype implementation, the application specifies this set of global nodes. These global clients are omitted from the graph while performing partitioning and later added to all the clusters. The following conditions have to be satisfied by the partitioning algorithm:

1) *Maximize social consistency and load balancing:* $| \sum_{\forall \mathbf{c_i}, \mathbf{c_j} \in \mathbf{Cl_1}} \mathcal{W}[\mathbf{c_i c_j}] - \sum_{\forall \mathbf{c_k}, \mathbf{c_l} \in \mathbf{Cl_2}} \mathcal{W}[\mathbf{c_k c_l}] | < \epsilon_1$, where $\sum \mathcal{W}[c_i c_j]$ denotes the combined weight of all the edges and nodes in a partition. This condition states that the sum of edge and node weights in two partitions are roughly equal. (ϵ_1 is a relatively small number).
2) *Minimize the loss of social connectivity:* $\forall c_i \in Cl_1 \wedge c_i \notin Cl_2 \wedge c_j \in Cl_2 \wedge c_j \notin Cl_1$, $c_i c_j \in E : \sum \mathcal{W}[c_i c_j]$ is minimized at all points of time. This condition states that the partition should be a min-cut, minimizing the total weight of edges across the partitions.

Please note that we defined the conditions considering only a pair of clusters (Cl_1 and Cl_2), though these can be generalized to any number of clusters.

We try to provide social consistency on a best-effort basis. Our graph partition captures social consistency and concomitantly performs load balancing; we prioritize load balancing when the load becomes prohibitive.

IV. Sequential Consistency Violation due to Inter-Cluster Communication

As per our definition of social consistency, the nodes within the clusters are sequentially consistent, while nodes across clusters (inter-cluster) are eventually consistent. This means that if clients seek the value of an object from clients in other clusters, then sequential consistency can be violated. For example, consider a scenario with two clusters Cl_1 and Cl_2. A client from Cl_1 reads the value of an object from Cl_2 at some time t_1. If the client in Cl_1 re-reads the object's value from Cl_2, at some future time $t_2 : t_2 \geq t_1$, then it should receive either the previously read value (if the cluster Cl_2 did not write to the object in the period (t_1, t_2)) or a later value (due to Cl_2's write). However, if the value read is older than the value read at t_1, it implies a violation in sequential

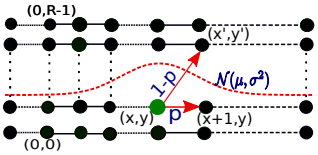

Figure 1: Feasible movements in XY space

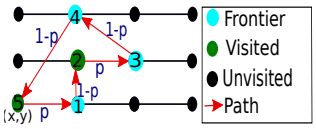

Figure 2: SC Violation

consistency. This example can be extended to more number of clusters.

Let us consider a system with R clusters, where each cluster can progress independently, and can have either inter-cluster or intra-cluster communications. The term intra-cluster *communication* is used for all the read/write accesses to the shared object within the cluster. This is because these accesses result in information transfer among the clients of the same cluster. Let us define a single step as a read or a write. As is common while analyzing concurrent systems [21] let us assume each cluster to be a process that takes only one step at a time. Furthermore, assume that all the processes are in lock-step and make a single movement (send a single read/write message either within or outside the cluster) at discrete times: steps 1, 2, 3, 4, ... (similar assumption as [21]). Assume that we send a message within the cluster with probability p, and we send a message to a node in another cluster with probability $1 - p$.

This can be modeled as a random walk in a 2-dimensional Cartesian system (see Figure 1). We consider a matrix of points where each point is a read or write event. Since the read/write events (in each cluster) are sequentially consistent, we can arrange them in a linear order. Hence, we represent them as a row of points arranged sequentially (one row for each cluster). Hence, the y coordinate in this Cartesian system represents a particular cluster, and it ranges between $[0, R-1]$. The x coordinate represents the number of the time step and has a range from $(-\infty, \infty)$. We use this formalism to develop a mathematical model to bound the probability of violating SC.

Initially, we start at $x = 0$ for all the clusters. Let us consider a point (x, y) in our Cartesian space as shown in Figure 1. From this point, we can either take 1 step forward (intra-cluster message with probability p) and reach $(x + 1, y)$, or we can move to another row (inter-cluster message with probability $1-p$), and reach any point $(x', y') : y' \neq y$. A move from the point (x, y) to (x', y') where they are in different rows (clusters) happens when the event at (x, y) and the event at (x', y') have a happens-before relationship ($read \rightarrow write$, $write \rightarrow read$, $write \rightarrow write$). In other words, every movement from point A to point B represents a happens-before relationship between the events at points A and B. Assume that there is a directed edge between any two points that have a happens-before relationship: two points on the same row, and two points connected across clusters.

To summarize, we can model the process of reads and

writes in a distributed system (with the assumptions made by Attiya et al. [21]) as a random walk in a matrix. We can either move one step to the right in the same row (intra-cluster access), or move to any other column on a different row (inter-cluster access, which is not in SC). We know that the probability of going to another row is $1-p$. We assume that the other row is uniformly distributed, and the value $x' - x$ follows a given distribution $\mathcal{D}$.

SC Violation Condition: We consider that every co-ordinate (x, y) in this system has a state, $S(x, y) \in \{visited, unvisited\}$. Initially, all the coordinates are in the unvisited state. When the walk begins, the state of points being traversed is changed to visited (for example, in Figure 2, as point (x, y) is traversed its state becomes visited). If during the walk, we reach a particular coordinate (x', y'), such that $\exists\ (x'', y') : x'' \geq x' \wedge S(x'', y') = visited$, then such a situation is called a violation.

This is tantamount to a cycle in the graph of happens-before relationships. To compute the violations, for each cluster ($y \in Y$) we define a function, $Frontier(y) = Max\{x : S(x, y) = visited\}$. Formally, a violation happens when:

$$(x, y) \xrightarrow[\text{to}]{\text{transition}} (x', y') : x' \leq Frontier(y').$$

Referring to Figure 2, node 1 is initially unvisited. Once it is traversed, it becomes visited and also serves as the new frontier for its row. As the walk continues, the transition from node 4 to node 5 result in a violation, since node 5 is before the frontier of the corresponding row.

The probability of violation depends upon the following parameters:

- Inter-cluster communication probability $(1-p)$.
- The distribution $\mathcal{D}$.
- Number of clusters or rows (R).
- Number of steps taken (k), $k = k_f + k_a$. k_f is the total number of forward steps, and k_a is the total number of inter-cluster steps.

To find the dependence of the violation probability on these parameters, we performed Monte-Carlo simulations. Following observations have been made from the experiments as shown in Figure 3(a-d). Here $\mathcal{D}$ is a discrete normal distribution $\mathcal{N}(\mu, \sigma)$ that has been shifted (validated empirically).

The violation probability:

- Decreases with an increase in p. This is because a high value of p means a low number of inter-cluster messages, and hence a low violation probability.
- Decreases with an increase in R. This is because we can make a lot of moves in many more clusters.
- Increases with an increase in the variation of the standard deviation, σ. This is because as the variance increases, the dispersion increases; thus increasing the probability of returning to an older value.
- Increases with an increase in k.

Let us denote each configuration by a tuple $\langle p, \sigma, R, k\rangle$. Given a bound on the probability of SC violations (P_v), and bounded values of some parameters in the tuple, the best possible values for the remaining parameters can be found by using our model. Let us describe two criteria.

- **Criteria CLUST_SIZE:** Assume that we wish to have an SC violation on an average once every k steps. In other words, for let's say a million reads/writes, we want to bound the number of SC violations. In such a scenario we need to choose the minimum number of clusters R in such a way that we can meet our target. Note that R and p are inter-dependent; however, the distribution $\mathcal{D}$ is not necessarily dependent on R. It is a function of the hardware and network latencies.
- **Criteria RUN_LENGTH:** In another situation, we can fix the number of clusters R (and consequently p), and try to run the system until we accumulate a maximum number of SC violations. To stop the quality of experience from worsening, we can either introduce a period of quiescence or rearrange the clusters.

A. Empirical Validation of our Model

Our model is generic and can be tuned to represent any application. Let us consider SC violations in a popular key-value store, Cassandra, as described by Liu et al. [22]. Figure 4 shows three consecutive operations, $W1$, $W2$ and $R3$, respectively, where $L1$ and $L2$ are the issuing latencies between them. This scenario satisfies SC if $R3$ reads the value of $W2$. For achieving SC, the propagation delay (D) among the replicas should be less than $L2$, else $R3$ will either return $W1$ (if $L2 \leq D \leq (L1 + L2)$) or some older value, thus violating sequential consistency. To compute the probability of achieving sequential consistency we have used a version of Cassandra with support for Probabilistic Bounded Staleness [19]. Figures 3(e,f), represents the no-violation probability as a function of the issuing latency for 4, and 5 node systems, each with a replication factor of 2, and 3, respectively. The graph shows that as the issuing latency increases, the probability of violation decreases. This happens because the updates can be propagated to the replicas before the read request arrive. Our mathematical model can be adapted to represent this system as described in the following subsection.

1) Adapting our Formulation to Model Cassandra: The probability of SC violations in Cassandra depends upon the number of replicas (R_{rep}), the propagation delay among the replicas (D), and the latency distribution between *writes and subsequent reads*. Please note that the number of clusters (R) in our model is different from the number of replicas (R_{rep}) in Cassandra, since the replicas communicate at a very high frequency to remain consistent while our clusters communicate minimally. So we consider two rows to model Cassandra; one row represents all the replicas and the other row represents a cluster of clients issuing requests (there

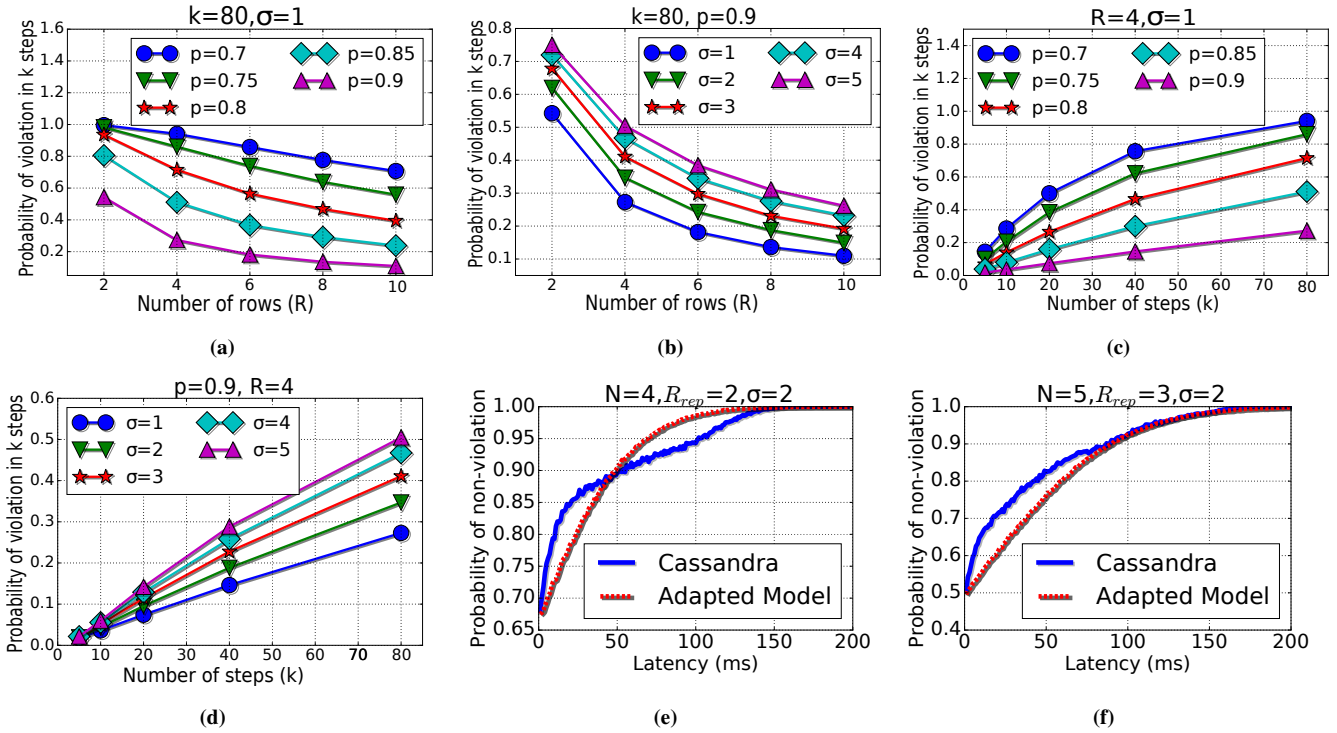

Figure 3: Probability of violation vs (a-d) **(a)** Number of rows at different values of forward probability p **(b)** Number of rows at different values of σ **(c)** Number of steps at different values of forward probability p **(d)** Number of steps at different values of σ **(e)** No violation probability vs issuing latency (R_{rep}=2, N = 4) **(e)** No violation probability vs issuing latency (R_{rep}=3, N = 5).

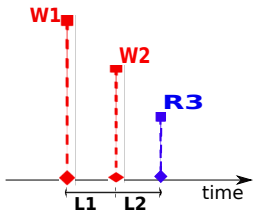

Figure 4: Experimental scenario for checking Sequential Consistency (adapted from [22])

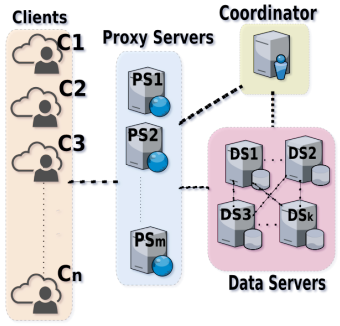

Figure 5: Architecture of the system

is a total order between them). The standard deviation of this model σ', depends on the number of replicas and the propagation delay among them as:

$$\sigma' = C * \sigma * R_{rep}^{r}, \quad (4)$$

where C is some constant of proportionality.

A high value of issuing latency means that the probability of reading a correct value is high. This means that in our model the probability of reading a value before the frontier of the replica row should decrease with increase in issuing latency. We capture this by continuously taking forward steps corresponding to the value of issuing latency, before taking an inter-cluster step. In particular, we consider that for $1ms$ of issuing latency, we move one step forward. Thus in the adapted model, our forward steps are a function of issuing latency rather than a probability p. After performing issuing latency equivalent number of steps, we take an across step which returns a value according to the discrete Normal distribution $\mathcal{N}_d(x, \sigma'^2)$. In summary, the adapted model has the following parameters for our 4-tuple $< [1|0], \sigma', 2, k >$ configuration, where the first component means that for k_f number of steps we move forward with probability 1, and the remaining steps are across steps, hence the corresponding p is zero. We performed Monte-Carlo simulations on our adapted model. Figures 3(e,f), show that our adapted model has similar non-violation probabilities as that of Cassandra. For all the experiments with $\sigma = 2$, the σ' value which best reflected Cassandra's behavior was $40 * \sqrt{R_{rep}}$, implying that if $\sigma = 2$, we have $C = 20$, and $r = 1/2$.

V. Split and Merge Protocols

Figure 5 presents the architecture of our system, where $C_1, C_2, \ldots, C_n$ represent the clients. The data servers, $DS1, DS2, \ldots, DS_k$, form the storage layer for all the objects. Initially, in a no-overload situation, the data servers store different objects, and each object is stored on a single server. Whenever there is an overload we initiate the *split* protocol: add more servers to store the replicas. We can also dynamically reduce the number of replicas (*merge* protocol).

The *Coordinator* maintains the mapping between the client ids and the cluster ids (which are uniquely mapped to the data servers), for different objects in the form of a partition function. Upon any split or merge, the coordinator provides this mapping information to the proxy servers as routing functions. The proxy servers, $PS1, PS2, \ldots, PS_m$, add the clients to the system with help of the coordinator.

Proxy Servers act as mediators between the clients and the data servers; they route the client requests according to the routing information provided by the coordinator.

A. The Split Protocol:

Each data server tracks the incoming request rates of its objects, and when the request rate for a particular object goes beyond a predefined threshold, the overloaded data server sends a split request to the coordinator, who then performs the user partition and generates new routing functions.

Consider a social graph of users where their connections are captured by the weight matrix, $\mathcal{W}$, as described in Section III. Before the split, assume that the social graph for a given object is split into R separate sub-graphs (one for each cluster). Now, we need to increase R to R' ($R' > R$) to reduce the load on each server. Once we choose R', we can then split the graph into R' sub-graphs, where the number of accesses for each group of users is roughly the same (see Section III for the details). To choose R' we can use either the criteria *CLUST_SIZE* or *RUN_LENGTH* as described in Section IV. This depends on the application's requirements and can be conveyed to the runtime system by the developer.

We envision a small software routine that runs a short profiling run for different values of R (1s for each configuration in our experiments, less than 1% of the execution time). In the profiling run, we record the parameters that are required in our theoretical model: p and $\mathcal{D}$. Then we run a short Monte Carlo simulation (in ms) to find the relationship between the number of steps and the probability of error. An astute reader may question the need for a Monte Carlo simulation when we have a profiling run; however, note that finding SC violations is hard in most systems, and secondly just to observe p and $\mathcal{D}$ we need very short profiling runs with minimal instrumentation (timestamping of messages). Additionally, this profiling can only be done once in the lifetime of an application, and the results of the Monte Carlo simulations can be stored in memory. This will make the overhead negligible. To summarize, at this point we have a relationship between the number of clusters, number of steps, and the probability of SC violations.

Now, the load per server decreases linearly with the number of servers; however, having more servers is expensive. For the *CLUST_SIZE* criteria, we need to minimize R' subject to the constraints: (1) $R' > R$, (2) if L is the request rate for the object, then $L/R' < \tau$, (3) the probability of SC violation is bounded by $\mathcal{P}$ for every k steps and (4) $R' < R_{max}$, where R_{max} is the maximum number of servers we can allocate for the object. $\mathcal{P}$ and τ are user-defined thresholds. We have similar equations for the *RUN_LENGTH* criteria. Note that (3) is trivially satisfied for a shifted normal distribution; however, this need not be the case for other kinds of distributions. This optimization problem can easily be solved using a greedy approach (if it is convex), or by exhaustive enumeration. The time penalty for this phase is negligible.

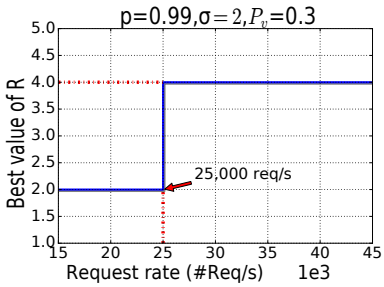

Figure 6: Best R as a function of Request rate

Figure 6 shows the results of Monte Carlo simulation corresponding to the *CLUST_SIZE* criteria. The simulation uses $p = 0.99$ (determined from the profiling run of a Cassandra workload), and the distribution $\mathcal{D}$ is chosen to be Normal with $\sigma = 2$, which is the same as obtained from PBS version of Cassandra in Section IV. The bound on probability violation is assumed to be 0.3. The value of τ, i.e., the system threshold, is approximately 8000 req/s (experiments computing this system threshold value are described in Section VI).

B. The Merge Protocol:

Akin to the split protocol, the merge protocol is invoked when the average load per server falls below a lower threshold. In this case, we need to choose a new value of the number of clusters, R', where $R' < R$. The rest of the analysis is the same as the split protocol.

VI. Evaluation

A. Experimental Setup:

We implemented a mock social network application, GLEE (**G**ood qua**L**ity **E**xperience and p**E**rformance), over the widely used Cassandra [9] database and stored posts as key-value pairs: the post-id is the key, and the comment-list is the corresponding value. In our experiments, each write request is saved as a comment, which is appended to a post and each read request returns a list of the latest 50 comments for that post. We have focused on only one post to show the impact of hot-spots. We use Gpmetis [23], a graph partitioning tool, to partition the social graphs. Our experiments are based on the *CLUST_SIZE* criteria, i.e., we target on choosing R, while k depends on the incoming workload.

We ran our experiments within a single data center equipped with Intel Xeon, Core $i3$, 2^{nd} generation processors, each having 4 cores, 16 GB RAM, 3 TB Seagate HDDs (7200 rpm) and connected by Gigabit Ethernet (GbE). The servers ran Ubuntu Linux 16.04. In each experiment, we use three dedicated machines for proxy servers, one for the coordinator, three for all the clients, and one, two or four for data servers (depending on the number of clusters created after splitting). The data servers run Cassandra 3.9 in the backend

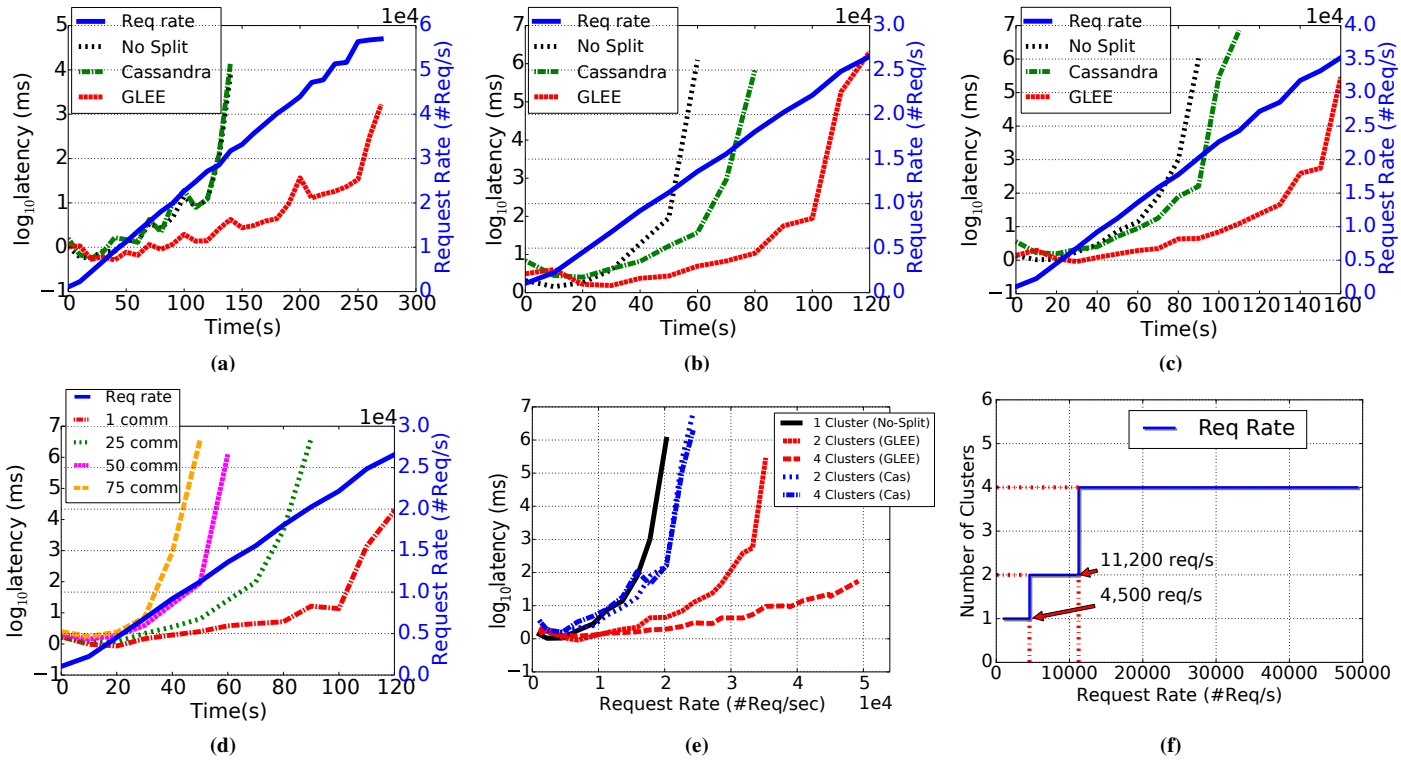

Figure 7: **(a)** Write-Only workload **(b)** Read-Mostly workload **(c)** Mixed workload **(d)** Latency with number of comments read **(e)** Multi-Split (1, 2 and 4 clusters) **(f)** Number of clusters at different request rates during the execution.

and store data without replication, with a consistency value of one (i.e., reads/writes are performed at any one server before returning). Initially, there is a single data server to handle the incoming workload. Upon overload, another database with the same schema is created (on another data server), and the object data is copied. These two servers form their clusters, without any replication (replication factor:1) and consistency value:1. Upon subsequent overloads, this process is repeated without altering the replication factor and consistency value.

We evaluated the performance of our system in terms of latency and request throughput and compared it with Cassandra. Cassandra is used in Facebook and is its default load balancer; hence, it is one of the best candidates for comparison. In the experiments, Cassandra initially has a replication and consistency value of one. Upon any subsequent overload, the replication factor is incremented by one (after adding a data server), but the consistency value remains one throughout, unlike our model. We get the data for social relationships for 360 users from Twitter [24], and we use both synthetic data sets and data sets from Twitter and Facebook.

B. Experiments and Inferences:

We initially evaluated our system on a synthetic benchmark where the request rate increases in steps, i.e., considering that the time has been divided into intervals, request rate is constant during a time interval, and increases or decreases in steps at different intervals. A split is performed when the incoming request rate reaches 60% of the system threshold (determined empirically to be the best point for a split in terms of performance).

Figures 7(a-c) show the request rate (plotted in the second y-axis) and corresponding latencies (log_{10} scale) for three types of workloads: i) Write-Only ii) Read-Mostly (95% Reads, 5% Writes) and iii) Mixed (50% Reads, 50% Writes). Each latency graph has three curves representing the base case (no-split, i.e., the system has only one cluster), Cassandra and GLEE, both with one split each (i.e., the system has two clusters). After the split, both Cassandra and GLEE use double the resources. Figures show that GLEE handles $(1.7 - 1.9)\times$ more workload in comparison to the non-split configuration, and $1.6\times$ workload in comparison to Cassandra. Cassandra has a negligible performance gain, $(1 - 1.2)\times$, showing that Cassandra does not perform effective load balancing when there is a single hot object. Cassandra sends more messages across clusters, whereas GLEE has a better clustering scheme.

Figures 7(a-b), show that GLEE can handle $(2 - 2.5)\times$ more of the Write-Only workload than the Read-Mostly workload. This is because in the Read-Mostly workload, the performance is dependent on the number of comments that are read from the server in response to a read request. The latency of the read request increases as the number of comments read by it, increases. We confirmed this by running a non-split experiment (Figure 7d), varying the number of comments to be read. Thus, the number of comments read can be tuned, and in our experiments, we read 50 comments as this value is realistic and provides reasonable performance.

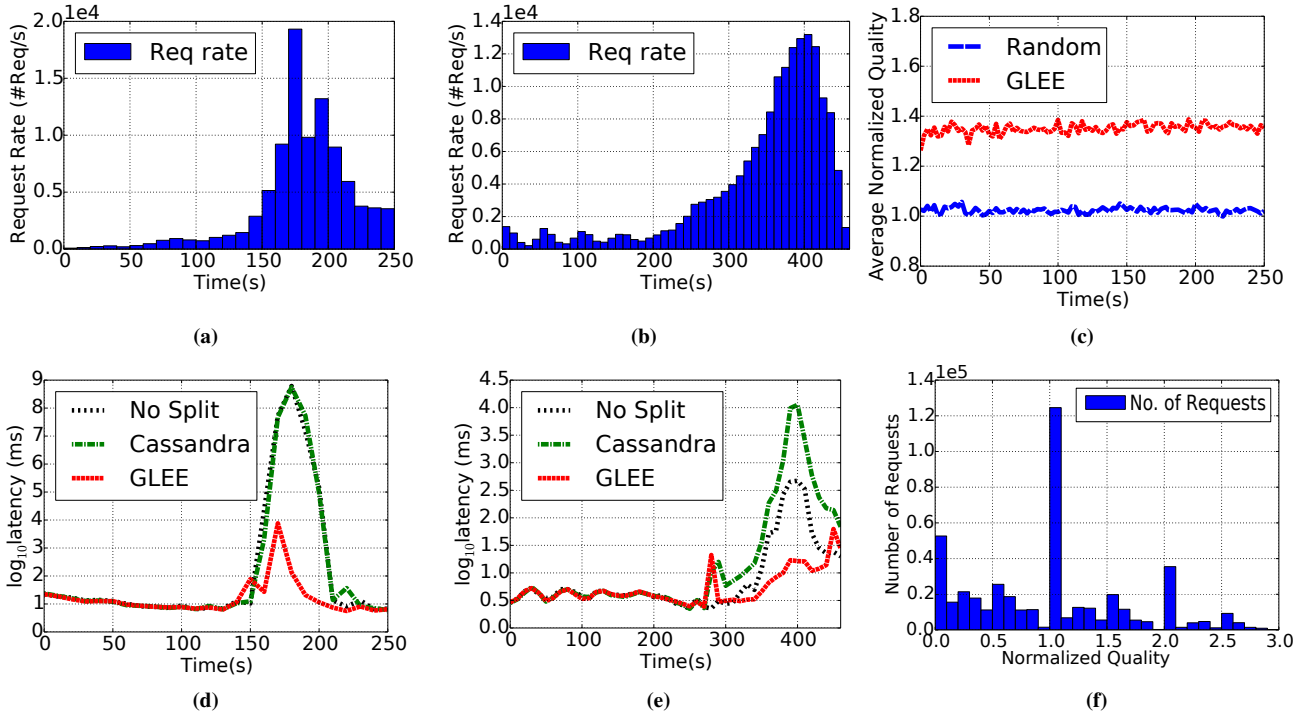

Figure 8: **(a)** Request rate for Twitter Dataset **(b)** Request rate for Facebook Dataset **(c)** Quality curve for Twitter Dataset **(d)** Latency for Twitter Dataset **(e)** Latency for Facebook Dataset **(f)** Clients QoE statistics

Henceforth, we use the Mixed workload for all our future experiments as the other workloads show the same pattern predictably. Figure 7(e) presents a multi-split scenario: initially, there is only one cluster in the system, and after subsequent splits, the number of clusters increases to two and four. Our experiments demonstrate that GLEE surpasses Cassandra by handling higher workload: $1.6\times$ more by performing one split and $2.4\times$ more by performing multiple splits. Figure 7(f) shows the number of clusters used at different request rates during the execution of a multi-split scenario.

We also present the results with two real-world datasets collected using i) Twitter: Higgs dataset [24] and ii) Facebook's API: comments on a famous live video, shown in Figures 8(a,d) and 8(b,e) respectively. The graphs show that the results are in line with our previous results with synthetic workloads. For clarity, we have shown the request rate and latency in separate graphs. The spike in latency curves 8(d) is due to queuing, as the request rate spontaneously increases from 9000 req/s to 18000 req/s (at the time interval of 160-170s). GLEE quickly handles the situation (in $\sim 20s$) and becomes stable after splitting, while Cassandra provides poor performance throughout and performance is improved solely when the request rate decreases.

Apart from load balancing, we demonstrate the impact of social partitioning, and claim a good Quality of Experience (QoE) along with performance. Quality is loosely correlated with violation probability (P_v), since good quality means that the clusters formed have good social connectivity and hence low probability of inter-cluster communication, thus a low value of P_v. To quantify the QoE received by the clients, we introduce a new consistency metric: "***Quality***", which can be computed for the response of each read request. Quality represents the ratio of the number of friends' comments received ($Friends\_Writes_{received}$) to the total number of comments done by the user's friends ($Friends\_Writes_{done}$). Formally, the quality of a read request, R_i, is:

$$Quality_i = \frac{Friends\_Writes_{received}}{Friends\_Writes_{done}} \quad (5)$$

In the experiments, we compute the normalized quality of a split system w.r.t a non-split system, i.e

$$NormalizedQuality_i = \frac{(Quality_i)_{split}}{(Quality_i)_{no\_split}} \quad (6)$$

If there are n requests in total, the average normalized quality, $AvgNormalizedQuality$, can be computed as:

$$AvgNormalizedQuality = \frac{\sum_i^n (NormalizedQuality_i)}{n} \quad (7)$$

Figure 8(c) compares the quality of GLEE against another load balancing system model; we call it '$Random$'. $Random$ performs client partition randomly (similar to [11], [14]), thus handling only overloads while completely ignoring consistency. Figure 8(c) presents the $AvgNormalizedQuality$ of each system plotted relative to the non-split case, and show that the quality index of GLEE is 1.37 times higher than Random. Since we read only

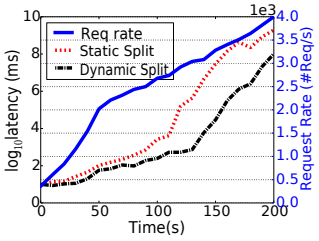

Figure 9: Static vs Dynamic Performance

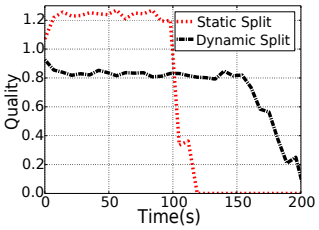

Figure 10: Static vs Dynamic Quality

the latest 50 comments, the probability of reading friends' comments can be higher in a split case than in a non-split case, leading to a quality value higher than one. Figure 8(f) presents the histogram of the number of requests issued by clients versus the quality of each request for GLEE. The graph shows that maximum requests have a quality index of 1, implying that maximum clients get a high-quality response. Only 30% of requests have a quality reduction by more than 50%, meaning that only a small percentage of clients do not get good quality. Please note that Figure 8(f) presents the quality of all requests 'individually', unlike Figure 8(c), which shows the average.

To corroborate the importance of dynamic weights in social partitioning, we evaluate our system in both the settings: only static weights (denoted by Static Split in the graph), and both static and dynamic weights (represented as Dynamic Split). Figure 9 shows that dynamic split provides better performance, though the quality offered (Figure 10) in this case is slightly lower than the case of static. This reduced quality is because our system tries to provide both excellent load balancing (leading to better performance) and good social consistency (leading to better QoE) wherever feasible, but it favors better performance by compromising upon social consistency in case of critical situations. Figure 10 shows that the quality in the static case suddenly drops to 0. This sudden drop is because as the request rate increases, the system becomes overloaded with extremely high latency, and hence no requests complete.

VII. Conclusion

In this paper, we proposed a new model for analyzing a system with multiple sequentially consistent sub-domains, and then used it to implement a new consistency model called *social consistency*. The class of applications that do not require very strong consistency such as social networking, online reviews (Amazon), and collaborative editing (Google docs) are very well suited for social consistency. We implemented a prototype social application, GLEE, and evaluated it using synthetic and real-world datasets. We compared GLEE against the Cassandra key-value store and showed that GLEE outperforms Cassandra by handling $1.6\times$ and $2.4\times$ more workload (by performing one and three splits). GLEE provides this performance along with an average 37% better quality of experience.

References

[1] G. DeCandia *et al.*, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, pp. 205–220, 2007.

[2] "Project voldemort," http://project-voldemort.com/.

[3] W. Lloyd *et al.*, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *SOSP*, 2011.

[4] ——, "Stronger semantics for low-latency geo-replicated storage," in *NSDI*, 2013, pp. 313–328.

[5] C. Li *et al.*, "Making geo-replicated systems fast as possible, consistent when necessary," in *OSDI*, 2012.

[6] P. Bailis *et al.*, "Quantifying eventual consistency with pbs," *The VLDB Journal*, pp. 279–302, 2014.

[7] NDTV, "Ellen degeneres' selfie crashes twitter," http://www.ndtv.com/world-news/ellen-degeneres-selfie-crashes-twitter-552579, 2014.

[8] I. Ari *et al.*, "Managing flash crowds on the internet. modeling, analysis, and simulation of computer systems," in *International Symposium on, 0*, 2003, p. 10.

[9] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, pp. 35–40, 2010.

[10] M. Randles *et al.*, "Cross layer dynamics in self-organising service oriented architectures," *Self-Organizing Systems*, pp. 293–298, 2008.

[11] A. Rao *et al.*, "Load balancing in structured p2p systems," *Peer-to-Peer Systems II*, pp. 68–79, 2003.

[12] B. Pawlowski *et al.*, "Nfs version 3: Design and implementation." in *USENIX*. Boston, MA, 1994.

[13] S. A. Weil *et al.*, "Intelligent metadata management for a petabyte-scale file system," in *2nd Intelligent Storage Workshop*, 2004.

[14] ——, "Dynamic metadata management for petabyte-scale file systems," in *ACM/IEEE conference on Supercomputing*, 2004.

[15] D. Terry *et al.*, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *SIGOPS Operating Systems Review*, 1995, pp. 172–182.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[17] J. M. Pujol *et al.*, "The little engine (s) that could: scaling online social networks," *ACM SIGCOMM Computer Communication Review*, pp. 375–386, 2010.

[18] Y. Sovran *et al.*, "Transactional storage for geo-replicated systems," in *SOSP*, 2011.

[19] P. Bailis, "Using pbs in cassandra 1.2.0," http://www.bailis.org/blog/using-pbs-in-cassandra-1.2.0/, 2013.

[20] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, pp. 66–76, 1996.

[21] H. Attiya and D. Hendler, "Time and space lower bounds for implementations using k-cas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, pp. 162–173, 2010.

[22] S. Liu *et al.*, "Quantitative analysis of consistency in nosql key-value stores," in *QUEST*. Springer, 2015, pp. 228–243.

[23] G. Karypis *et al.*, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[24] D. Domenico *et al.*, "The anatomy of a scientific rumor," *Scientific reports*, p. 2980, 2013.