Handling Overloads with Social Consistency

A Thesis

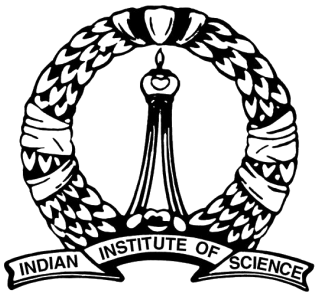
Submitted for the Degree of

Master of Science(Engineering)

in the **Faculty of Engineering**

by

Priyanka Singla



Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

JUNE 2018

DEDICATED TO

My Family, Teachers and my Friends

Signature of the Author: .....

Priyanka Singla
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Signature of the Thesis Supervisor: .....

K. Gopinath
Professor
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Acknowledgements

I would like to take this opportunity to express my gratitude to my guide Prof. K. Gopinath for his exemplary guidance and encouragement throughout the course of this work.

I would also like to thank Prof. Lorenzo Alvisi and Prof. Allen Clement with whom I started working on this project. I am greatly indebted to Dr. Smruti Sarangi for his help, guidance, and support which was very crucial during the tough times of the program. I also acknowledge my colleague, Shubhankar Singh, for great and wonderful discussions.

I am extremely grateful to my family and friends for being my strong support pillar and motivating me to work more efficiently. This accomplishment would not have been possible without them.

Priyanka Singla,
IISc

Abstract

Cloud computing applications have dynamic workloads, and they often observe spikes in the incoming traffic which might result in system overloads. System overloads are generally handled by various load balancing techniques like replication and data partitioning. These techniques are effective when the incoming bursty traffic is dominated by reads and writes to partitionable data, but they become futile against bursts of writes to a single hot object. Further, the systems which use these load balancing techniques, to provide good performance, often adopt a variant of eventual consistency and do not provide strong guarantees to applications, and programmers. In this thesis, we provide a solution to this single object overload problem and propose a new client based consistency model – *social consistency* – that advocates providing a stronger set of guarantees within subsets of nodes (*socially related* nodes), and providing eventual consistency across different subsets. We argue that by using this approach, we can in practice ensure reasonably good consistency among the clients and a concomitant increase in performance.

We further describe the design of a prototype system, BLAST, which implements this model. It dynamically adjusts resource utilization in response to changes in the workload thus ensuring nearly constant latency, and throughput, which scales with the offered load. In particular, the workload spikes for a single *hot* object are handled by cloning the object and partitioning the clients according to their social connectivity, binding the partitions to different clones, where each partition has a unique view of the object. The clones and the client partitions are recombined when the spike subsides. We compare the performance of BLAST to Cassandra database system, and our experiments show that BLAST handles 1.6× (by performing one split) and 2.4× (by performing three splits) more workload. We also evaluate BLAST against another load balancing system and show that BLAST provides 37% better QoE.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There has always been a trade-off between performance, flexibility, and programmability in distributed consistency models. For example, sequential consistency is an excellent model for the programmer; however, implementing it in practice is difficult, and most of its implementations have a very low performance. On the other hand, a more relaxed model such as eventual consistency provides high performance; however, it is hard to provide accurate correctness guarantees or ensure a minimum quality of experience. Hence, finding a consistency model somewhere in between has been the focus of a lot of current work [1, 2]. Most of the relaxed consistency models are more restrictive than eventual consistency, and depending upon the business case, they prioritize a given set of operation orders. For example, the memory model in Amazon's Dynamo [3] prioritizes writes over reads, because the aim here is to ensure immediate logging of all the additions to the shopping cart. Similarly, in the case of an e-mail server, it is to be ensured that the reply is visible only after the original e-mail.

In this thesis, we look at a novel consistency model for cloud computing environments particularly for the scenarios that involve large-scale data storage. These applications have dynamic workloads and often observe spikes in incoming traffic [4, 5], which result in hot-spots. For example, a simple reference to a web page in a popular news feed or a particular post on a social-networking site by a celebrity [6], or any major worldly event [7] can result in a deluge of requests, which might possibly overwhelm the system. Such scenarios are particularly pernicious for interactive systems [8], and it is important to ensure that most of the clients get a minimum Quality of Experience (QoE).

Our main insight is as follows: Most of the social networking sites have fairly lax QoE guarantees which are reflected in their consistency models and methods of operation [9, 10]. They try to ensure a good quality of experience on a best-effort basis. Given the fact that they are not dealing with mission-critical workloads, it is possible to ensure an equitable trade-

off between power, performance, and reliability. We leverage such scenarios and propose a novel consistency model that is particularly germane to load balancing. We observe that there might be certain objects (e.g. popular Twitter feed) that are heavily accessed and a situation might arise where the system gives up on performance or consistency. However, none of the clients like to have a degraded performance or quality of experience. To handle such situations, we propose a novel consistency model – *social consistency*, which provides good quality of experience without any performance degradation. The word is derived from the fact that a set of *socially* related clients are grouped together and can view each others' updates without any decrease in performance. Several such subsets can be formed; within each subset clients see strongly consistent data with respect to each other, and for the clients across subsets, we opt for a more relaxed consistency such as eventual consistency.

In this thesis, we provide a theoretical definition of social consistency. Along with this,

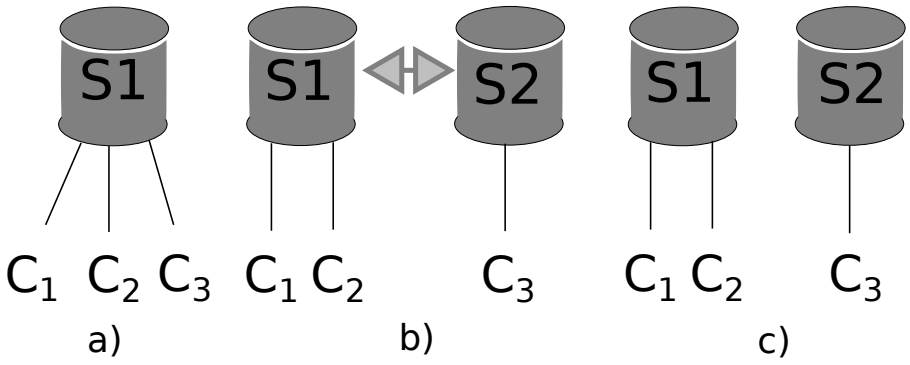


Figure 1.1: A system with social consistency

we describe a practical realization of the idea and name it BLAST (**B**est-effort **L**atency **A**nd **S**calability **T**ogether). BLAST takes into cognizance the dynamic nature of workloads and it provides different consistency types, adapting to different workloads. It spans the consistency continuum from strong consistency to eventual consistency, with load on the system being the tuning knob. In particular, initially an object is stored on a single server and is accessed by all the clients, who see a consistent value of the object, i.e all the clients in the system are strongly consistent (Figure 1.1-a: the object is stored on server $S1$ and is accessed by clients C_1, C_2, and C_3). With the passage of time, as the object starts to get heavily accessed, this hot object is cloned and stored on another server (server $S2$ in figure 1.1-b), and the client set is partitioned into 2 groups (clusters) on basis of social relationships among the clients (Figure 1.1-b: sets $\{C_1, C_2\}$ and $\{C_3\}$). Hereafter, each set of clients accesses different clones, and the clones do not communicate (Figure 1.1-c) until the workload spike gets over. 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 [3, 11]. From the consistency perspective, all the clients accessing the data of a particular split see a consistent value and hence are strongly consistent, but the system as a whole is eventually consistent where the clients (all the clients of the system) see consistent data only after the merge. In essence, initially the system as a whole is strongly consistent, however, as the load increases it moves towards weak consistency. Though at different times during the execution, the overall system might be strongly or weakly consistent, but the clients always see consistent data with respect to other socially related clients (within the same cluster). This is summarized in Table 1.1.

	System consistency (w.r.t an object)	Client consistency
No split exists	Strong	Social
Split exists	Weak	Social

Table 1.1: Different consistency types during an execution

1.1 Motivating Example

We argue about the acceptability of social consistency on the basis of following observation in existing social networking sites such as Facebook, Quora and Goodreads [9, 12, 13]. The comments (on any post) posted by a user's friend are displayed on the user's Facebook wall. Updates are displayed as "your friend A commented on post X" or "your friend B liked post Y" on a user's wall. (Figure 1.2 represents a combined screenshot from social sites like Facebook, Quora and Goodreads, showing how these sites display updates from our social connections on various posts.)

Typically, seeing a friend's name, the user comments on that post or clicks to see the content associated with the post. Thus these sites use social relationships to show only the data that would be more relevant to each user. Similar to this, users are generally interested in the information provided by their friends. For example, while deciding to visit a particular place (using a travel website like TripAdvisor [14]) or while purchasing a new mobile phone on an e-commerce website [15, 16], if a user sees a review by any of her friends, then her decision about going to that place or buying the same phone would be greatly enhanced. Even in figure 1.2 (bottom part of the figure, referring to *Goodreads* data), a user's decision to read a particular book will be affected by the reviews provided by her friends. Thus even if these websites present a lot of data, users mostly prefer to see the data related to their friends. So

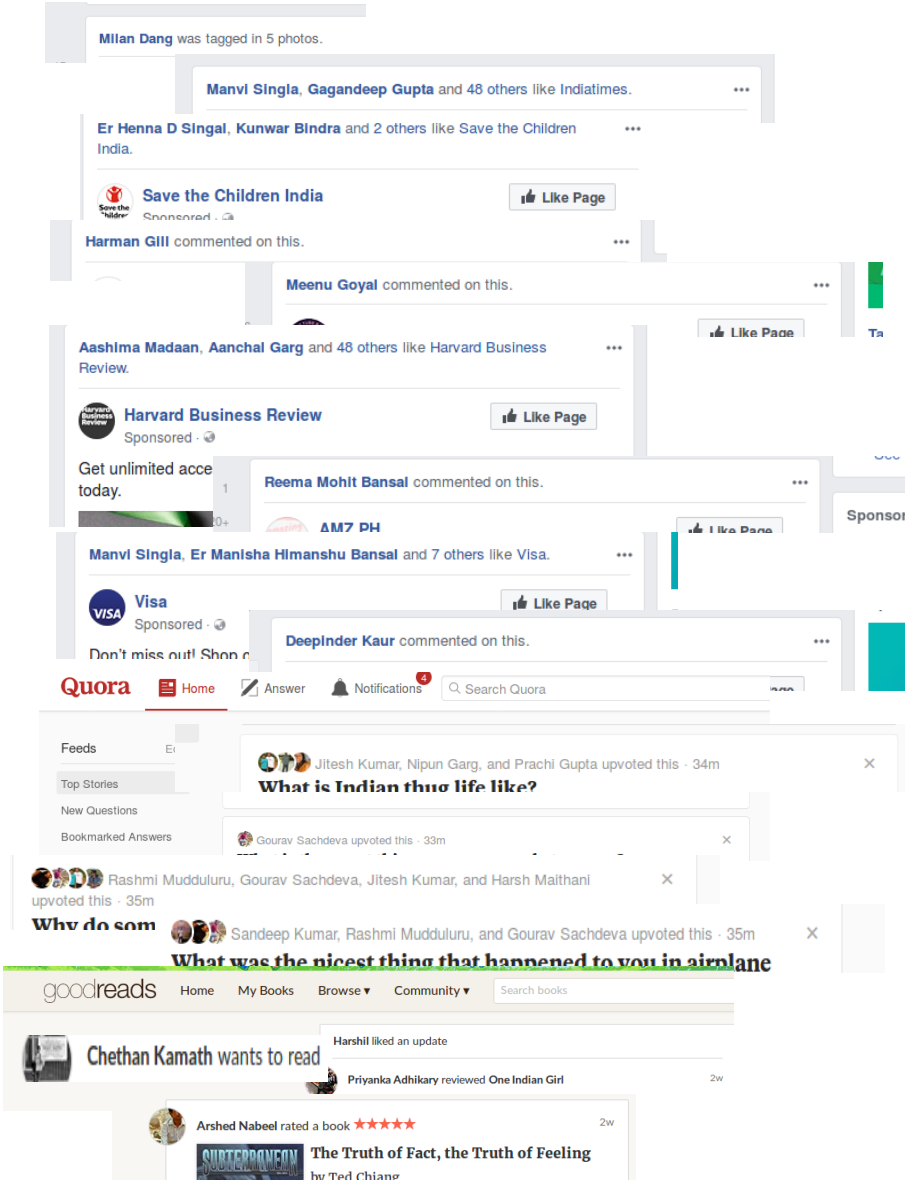


Figure 1.2: Data displayed by Facebook, Quora and Goodreads

for all practical purposes, users mostly interact with other users, who are socially connected to them. Our mechanism of overload handling uses the same idea, as the users are partitioned based on their social relationships. Since a user and other users in her social connection, all belong to the same partition and can see each other's data, they are not dissatisfied if they do not see the data of users who are not in their social circle. Thus showing only the desired data, rather than the entire data and hence, ensuring consistency within social circles of each user is

sufficient for consistency with respect to that user. Note that is not a general statement but is particular to certain use cases such as social media.

Our Contributions in this thesis are:

- We define a new model of consistency called *social consistency*, which offers a client-centric view of consistency in comparison to the existing object-centric consistency models, namely strong and weak consistency.
- We formally define strong and weak consistency and subsequently formalize social consistency.
- We present a split-merge algorithm to handle single object overloads.
- We introduce a new consistency metrics, *quality of response*, and use it to compare our model with various existing systems.
- We implement a basic prototype of a social networking service to simulate single object overloads and present results which demonstrate the efficacy of our system regarding performance and quality of user experience.

The thesis is organized as follows: Chapter 2 discusses the related work. Chapter 3 provides a detailed description of our social consistency model. Chapter 4 presents the architecture of the system. Chapter 5 implements and evaluates a prototype based on our model. Chapter 6 presents some discussion on a few concepts referred in the thesis, and then provides a list of various applications that can be modeled by using our approach. Finally, Chapter 7 concludes the thesis. Please note that the terms clients and users are used interchangeably in this thesis.

Chapter 2

Background and Related Work

The CAP theorem [17] states that consistency, availability, and tolerance to partition cannot be achieved together in a distributed system. Although if a network with good connectivity and no partitioning is assumed, achieving high availability and strong consistency, thus good performance, might seem trivial. However in practice, maintaining a good performance is not an easy task. There can be situations where a workload might overload the system, thus severely hampering the performance and resulting in high latency. Several approaches have been proposed in the past to handle the overload issue. Some of these approaches provide trade-off solutions, in which they compromise consistency to provide low latency, while a few other approaches provide non trade-off solutions. For example, admission control techniques manage overloads by handling requests of only certain clients, while rejecting others. Admission control techniques do not trade-off consistency, but they treat clients differently and do not provide same performance for all of them. Following we present both types of solutions (trade-off and non trade-off) in detail, beginning with the former.

Replication and Caching: In this approach, the overloaded data is replicated to other servers, which then handle the incoming requests, thus reducing the overload. The replicas can be synchronized according to the application's requirement (see the survey by Wang et al. [18]). A few of the major proposals in this area include Bayou [19] distributed database, which uses selective replication and provides weak consistency guarantees. It provides a per-update dependency check and a merge procedure to detect and resolve conflicts. This work resembles our work in performing per object split-merge, but they do not form clusters based on social relationships. There are various other systems which perform replication [20, 21], but most of them either trades-off performance or consistency thus providing a poor quality of experience. Our system, on the other hand, provides good QoE without afflicting performance.

Data Partitioning: Along with replication, many systems use data partitioning techniques, in which the data is partitioned either horizontally or vertically, and different partitions are handled by different servers [3, 22, 23]. These systems perform an object based partitioning by using methods like hash-partitioning, range-partitioning or round-robin. In contrast, our system uses client-based partitioning, i.e rather than partitioning the objects in the system, the clients (who are accessing a particular object) are partitioned. This partitioning strategy helps to provide good quality of experience to the clients.

Amazon dynamo [3] and Cassandra [11] use consistent hashing to uniformly partition a key-space across the servers. However, a uniform key distribution does not imply a uniform workload distribution. There can be situations where the access distribution of the keys is highly skewed, for example, when a single key gets overloaded. This situation cannot be handled by partitioning and it deteriorates the system performance. We have quantified the impact on performance by evaluating our system against Cassandra (in Chapter 5), and show that Cassandra's performance is extremely poor in such workloads.

Metadata Partitioning: It is similar to the data partitioning technique, and is used for overload handling in the file systems. Large scale file systems need to efficiently manage their metadata [24, 25, 26], as metadata operations make up approximately 50% of the total file system operations. Most of the distributed file systems (NFS [27], Sprite [28]) use approaches like directory subtree partitioning (DSP) and hashing as their metadata partitioning strategies. However, despite these strategies, hot-spots can be formed in a file system. For example, the DSP technique suffers hot-spot issue when multiple clients open the same file simultaneously or a large number of users create multiple files in the same directory. Weil et al. [24] address these issues by performing subtree migration and replication. Weil et al. [25] further extend this work by integrating it with a caching scheme to adapt to the changing workload. Authors minimize the hot-spots by limiting the number of clients, who know from where to get the requested metadata. The incoming requests are randomly sent and forwarded within the metadata server cluster, which can result in unexpected long response times. This method of random replication and forwarding leads to inefficient resource utilization. In contrast to random replica selection, our approach wisely chooses the split and merge servers depending upon the loads on them and further handles client requests on the basis of the social relationships among the clients.

Wujuan et al. [26] present a Weighted Partitioning and Adaptive Replication (WPAR) scheme in which metadata servers are weighted on the basis of parameters like CPU power, memory capacity, network bandwidth, etc., and replicas are managed according to the incoming variable workload. Concretely, the replicas are created when the request rate increases beyond

a particular maximum threshold and are subsequently deleted when the request rate falls below another minimum threshold. This approach of creating and deleting replicas is similar to our split merge technique, however like the above approaches, WPAR also handles the requests independent of the relationships among the clients. Our work also presents the trade-offs related to consistency and performance, which are not discussed by the previous works, and we believe that they are important from an overall system perspective.

Relaxed Consistency Models: There have been various attempts to define systems that are generic, scalable, and possibly stronger than eventual consistency [29, 30, 21]. These systems either provide different consistency at different times, or depending upon the application's requirement, they prioritize a given set of operation orders. For example, Shapiro et al. [30] propose CRDTs, that rely on commutativity of the operations to ensure system consistency in presence of concurrent updates, without performing any synchronization or consensus. There is another work [21], which provides two different levels of consistency (strong and weak), based on the commutativity. Particularly, if the operations are commutative, they are allowed to execute concurrently, thus providing fast responses but eventual consistency. If the operations are not commutative, then the system is slow and strongly consistent. Our system, on the other hand, provides two different levels of consistency based on the social relationships among the clients and the frequency of their operations, rather than commutativity between the operations. Our work partitions the clients into different clusters based on the social distance between them, and the clients within the cluster are strongly consistent while across the clusters are eventually consistent.

Though the idea of user partitioning and cluster identification in social networks has been explored in a significant body of research [31, 32, 33], these works focus on other goals and unlike our work, they do not propose to handle overloads. For example, SPAR [31] uses the concept of social partitioning to minimize replication and focuses on storing the entire data of social clients in one group. Our work, in contrast, is independent of the client placement and follows an object based lightweight split at runtime (during overloads). To the best of our knowledge, none of the previous works concentrate on the relationships between the clusters and the impact of shared data accesses on system performance.

Some prior works, like WALTER [34], include various geo-replicated key-value stores that offer linearizability for replicas that are deployed in the same data center and weak consistency across the replicas in different data centers. This can be considered as a special case of our architecture, where consistency is dependent on physical proximity. Our model is far more generic and has an abstract social parameter. If the location of the replicas is chosen as the social

parameter, it would result in the above-mentioned system. Similarly, different applications can have other social parameters. For example, Chapter 3 describes a use case where friendship and the intensity of communication between the clients are considered as the social parameters.

Eiger [2] is another system which focuses on providing good consistency (restrictive than eventual) along with low latency. It proposes a novel system that provides causal consistency, which is relaxed in some cases. However, the main drawback of such works is that they are very specific, and it is hard to enforce causality at all times, and in all scenarios.

Next we discuss the solution which does not trades-off performance or consistency.

Admission Control and Load Balancing Techniques: Various approaches based on these techniques have been proposed in the past. In one of the approach, non-critical requests are dropped to limit the amount of load entering the system. The remaining requests are optimally distributed among different servers, such that all the servers perceive an equitable load. Another approach favors certain clients over others [35]. Randles et al. [36] present an extensive survey of load balancing algorithms in the cloud computing environments. They classify the design space of algorithms into two categories: coordinated and uncoordinated. The former class of algorithms [37, 38] uses a variety of work-stealing based techniques to steal work from the overloaded nodes. The other class of algorithms, uncoordinated load balancing, primarily rely on random sampling. For example, Rao et al. [39] rely on simple Bernoulli sampling for distributing jobs to the servers. All of the above approaches perform load balancing primarily considering the workload on the servers and are entirely oblivious to social relationships among the clients, which might result in poor satisfaction of the clients. Our approach keeps a track of social strengths among the clients and the the response contains updates by a client's social connections, resulting in high client QoE.

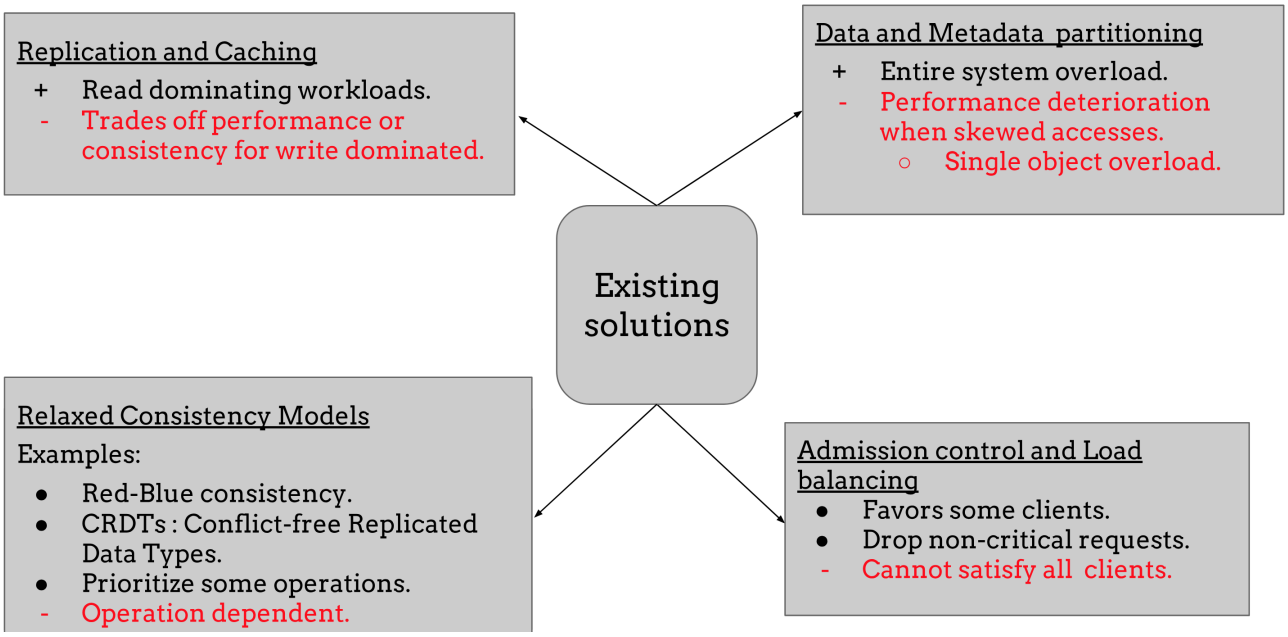


Figure 2.1: Existing overloading solutions and issues with them

Figure 2.1 summarizes all the solution categories, and also mention the main problem with each of them. In conclusion, the existing solutions *can't handle write dominated single object overloads along with providing good quality of experience to all the clients*. Our proposed model of social consistency handles these issues and aims at providing good consistency among the clients without any performance degradation. In the next chapter we begin by formalizing our model, followed by a detailed description of social consistency.

Chapter 3

Social Consistency

This Chapter introduces a formal definition of social consistency. We begin by formalizing the existing forms of consistency: strong and weak, in order to exhibit the association of social consistency with them.

In a distributed system, the data is replicated and any update should propagate to all the replicas to maintain consistency. Since the propagation takes time, so depending upon the application's latency requirements, the response to the clients can be sent either after updating all the replicas (synchronous replication, high latency) or immediately after updating the local replica (other replicas are updated asynchronously, low latency). These two replication mechanisms respectively result in strongly and weakly consistent distributed systems.

A 4-tuple, $\Gamma=\langle C, O, w, r\rangle$, is used to define a generic system:

- $C=\{c_1, c_2, c_3, ..., c_n\}$ is a finite set of n clients in the system.
- $O=\{o_1, o_2, o_3, ..., o_k\}$ is a finite set of k objects, on which operations are performed by the clients.
- w represents a write operation issued by a client. Concretely, $w_c(o,t)=v$, denotes that a value v is written to object o at time t by client c.
- r denotes a read operation, and $r_c(o,t)=v$, represents a read request issued by a client c to an object o at time t, and it returns a value v.

Note: For simplicity of notation, we assume that operations occur instantaneously, i.e., for strong consistency, the request issuance, propagation to replicas and return to the client, all happen instantaneously at time t (assuming propagation delay is zero). In weak consistency,

the request issuance and return happen instantaneously at time t (but with asynchronous propagation, hence there is no need to assume zero propagation delay). In weak consistency, if the propagation (asynchronous) to all replicas does not finish and another update request for the object comes, then the two updates are considered to be concurrent.

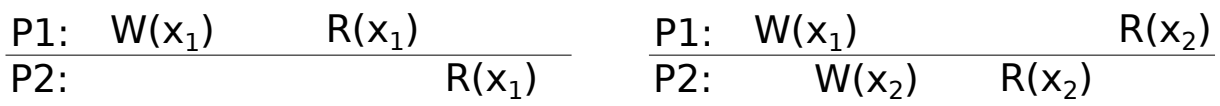
Below we formalize the strong and weak consistency systems.

3.1 Strong Consistency

It states that after the update completes, any subsequent access will return the updated value. Thus we have the following set of formal conditions that should be satisfied by strongly consistent systems:

- $\forall c_i, c_j, c_k \in C,\ c_i \neq c_j,\ \forall t_1, t_2, t' : t_2 > t_1, t' \in (t_1, t_2)$:
  $w_{c_i}(o, t_1){=}v \wedge \nexists\ w_{c_j}(o, t_1){=}v' \wedge \nexists\ w_{c_k}(o, t'){=}v'' \Rightarrow$
  $r_{c_k}(o, t_2){=}v$.
  When there are *no concurrent* updates, then after an update returns to the client, any subsequent access will return the updated value, provided there is no new update between the current update and its corresponding read. Figure 3.1a presents a pictorial view of the definition.

- $\forall c_i, c_j, c_k, c_l \in C,\ c_i \neq c_j,\ \forall t_1, t_2, t' : t_2 > t_1, t' \in (t_1, t_2)$:
  $w_{c_i}(o, t_1){=}v_1 \wedge w_{c_j}(o, t_1){=}v_2 \wedge \nexists\ w_{c_k}(o, t'){=}v' \Rightarrow$
  $\bigvee\ r_{c_k}(o, t_2){=}v_1 \wedge \nexists\ r_{c_l}(o, t_2){=}v_2$
  $\bigvee\ r_{c_k}(o, t_2){=}v_2 \wedge \nexists\ r_{c_l}(o, t_2){=}v_1$
  In case of *overlapping* updates, there is a non-determinism in the values observed by clients. However, all clients will see the same value (corresponding to the updates), provided there is no new update in between. Figure 3.1b shows that both the clients read the value x_2. Although, both of them could also have read the value x_1.



(a) No concurrent updates (b) Concurrent updates

Figure 3.1: Example: Strong consistency

We further formalize eventual consistency as this form of weak consistency is guaranteed by major cloud applications [3, 40, 41, 42].

3.2 Weak Consistency (Eventual Consistency)

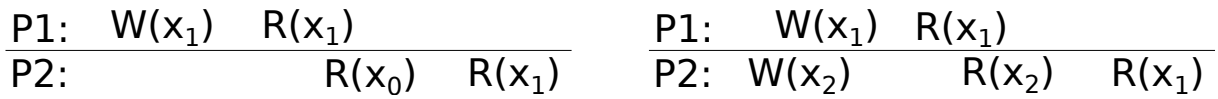
It guarantees that if no new updates are made to an object, eventually all the reads will return the last updated value. Thus we introduce the following formal conditions:
Assuming v_0 to be the initial value of the object o before any update.

- $\forall c_i, c_j, c_k \in C,\ c_i \neq c_j,\ \forall t_1, \exists t_2 : t_2 > t_1, \forall t' \in (t_1, t_2)$:
  $w_{c_i}(o, t_1){=}v \wedge \nexists\ w_{c_j}(o, t_1){=}v' \nexists\ w_{c_k}(o, t'){=}v'' \Rightarrow$
  $r_{c_k}(o, t_2){=}v \bigwedge \vee\ r_{c_k}(o, t'){=}v$
  $\vee\ r_{c_k}(o, t'){=}v_0$

  When there are *no concurrent* updates, then after an update returns to the client, any subsequent access can return either the updated value or an old value, but eventually, all the accesses will return the updated value, provided there are no new updates (Figure 3.2a).

- $\forall c_i, c_j, c_k, c_l \in C,\ c_i \neq c_j,\ \forall t_1, \exists t_2 : t_2 > t_1, \forall t' \in (t_1, t_2)$:
  $w_{c_i}(o, t_1){=}v_1 \wedge w_{c_j}(o, t_1){=}v_2 \wedge \nexists\ w_{c_k}(o, t'){=}v' \Rightarrow$
  $\bigwedge r_{c_k}(o, t'){=}v_1 \vee r_{c_k}(o, t'){=}v_2 \vee r_{c_k}(o, t'){=}v_0$
  $\bigwedge \bigvee r_{c_k}(o, t_2){=}v_1 \wedge \nexists\ r_{c_l}(o, t_2){=}v_2$
  $\bigvee r_{c_k}(o, t_2){=}v_2 \wedge \nexists\ r_{c_l}(o, t_2){=}v_1$

  In case of *concurrent* updates, the subsequent accesses can return either the updated value or the old value, but eventually, all the clients will see the same value (by either of the updates), provided there are no new updates (Figure 3.2b).



(a) No concurrent updates

(b) Concurrent updates

Figure 3.2: Example: Weak consistency

3.3 Social Consistency

Social Consistency is a hybrid of the above two consistency models. In social consistency, upon an object overload, the clients in the set C are divided into different clusters based on a partition function $\mathcal{Q}$: $C \times O \rightarrow Cl$, where Cl is a set of all cluster ids ($\{Cl_1, Cl_2, ..., Cl_m\}$), and O is the set of objects. This function maps the client id to the corresponding cluster depending

on the overloaded object. In a socially consistent system, all the clients within a cluster Cl_i are strongly consistent and follow the conditions defined in section 3.1. The clients across the clusters Cl_i and Cl_j form an eventually consistent system, in which after some time all the clients, $c \in (Cl_i \cup Cl_j)$, see the consistent data upon merge, thus satisfying the conditions mentioned in section 3.2.

We append the partition function ϱ to our system definition defined at the beginning of this Chapter 3. Thus, the updated system definition is a 5-tuple, $\Gamma'=\langle C, O, w, r, \varrho\rangle$. The read and the write definitions for this system are also modified slightly. We consider that the objects maintain lists rather than variables, and any *write (value v)* operation to the object o, *appends the value* v to the list maintained by this object. Similarly, any *read* operation to the object o returns the entire list of values, maintained by the object.

Following we present the modified definition of our system, $\Gamma'=\langle C, O, w, r, \varrho\rangle$, where

- C and O represents the finite set of clients and objects respectively (same as above).
- w represents a write operation, and $w_c(o,t)=v$ appends the value v to the list maintained by object o.
- r denotes a read request, and $r_c(o,t)=[v]$ represents a read request issued by a client c to an object o, which fetches the entire state of object o, atomically. Please note that we have denoted a *list* of values by a square bracket, [].
- A partition function ϱ defined above (described in detail in section 3.4).

Our system provides different types of consistencies within and across the clusters, and like any weak consistency model, it can be obtained by applying various relaxations to a strong consistency model [43]. These are basically the program order relaxations for operation pairs accessing different locations. Already existing weak consistency systems include the following relaxations: a) Write to Read b) Write to Write c) Read to Read and 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 compared to the clients in the other cluster (*Read own cluster's write early*). Table 3.1 lists the relaxations that are valid in our social consistency model.

We describe these relaxations by the following example. Let x be a shared object in a system, with three clients $\{P1, P2, P3\}$ accessing it. The clients first issue a write request to the object x (see Table 3.2); they write (1), (2), and (3) respectively, but at different times.

Then 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.

Relaxation	Yes/No
W → R/W Order	Yes
R → R/W Order	Yes
Read Own Write Early	No
Read Other's Write Early	No
Read Own Cluster's Write Early	Yes
Read Other Cluster's Write Early	No

Table 3.1: Valid relaxations in the social consistency model

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

Table 3.2: Social consistency example

However, this result is valid in the case of social consistency: if the clients P1 and P2 are considered to be in a single cluster while the client P3 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.

Theoretically, 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 the nodes outside of the cluster.

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

Condition 3: Within a cluster, the following additional properties are obeyed: monotonic reads and writes, and read your own reads/writes early. Please refer to the following reference [44] for a deeper exposition of these topics.

3.4 Partitioning the Clients into Clusters

This section describes the partition function 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 edge weights (i.e., the strength of the connections among the clients). A larger edge weight represents strongly connected clients. Along with the weighted edges, the graph also has weighted nodes.

Gpmetis [45], a graph partitioning tool is used, which partitions the graph balancing both types of weights (edge weights and node weights). The edge weights broadly have two types of components: *global* and *per-object dynamic*. The global component reflects the general measure of social strength among the clients, and is independent of any object, whereas the *per-object dynamic* component is specific to the overloaded object and it varies with time, depending upon the intensity of communication between the clients. In contrast to edge weights, the node weights have only a *per-object dynamic* component. Hereafter, for simplicity, we refer to '*global component*' as '*static weight*' and '*per-object dynamic component*' as '*dynamic weight*'. Following we describe these weights in detail:

- ***Static Weight of the edges (SW):*** This weight depends upon various parameters (provided by the application), each of which reflects a social behavior. Each parameter, i, has a boolean value, $x_i \in \{0, 1\}$, and an associated priority, $p_i \in$ [0,1], which reflects its importance relative to other parameters (higher value of priority means more importance). It is assumed that the application provides these priorities (though in general other scenarios are possible which are discussed in the Chapter 'Discussion').

  The static weight of an edge e, SW_e, is computed as:

$$SW_e = \sum_{i=1}^{s} p_i * x_i, \tag{3.1}$$

  where s is the number of static parameters. To understand this definition, let us consider an example of a Facebook-like application, where the parameters can be friendship, working place and interest. If an application assigns the following priority values: friendship: $p1 = 0.5$, working place: $p2 = 0.3$, interest: $p3 = 0.2$, then for any two clients, depending upon the values of these parameters, the corresponding edge weight can be calculated. For instance, an edge's weight would be $(0.5 * 1) + (0.3 * 0) + (0.2 * 1) = 0.7$, if the corresponding clients are friends ($x_1 = 1$) with same interests ($x_3 = 1$) but work at different places ($x_2 = 0$).

  It is to be noted that the parameters should reflect social behavior and hence contribute in enhancing the client-client relationships. If a particular parameter represents anti-social behavior, it would not be considered in the computation, i.e. its priority would be set to zero. For example, if two clients are "friends" but like "solitude", i.e., they like to be alone, then the latter parameter will not be considered in the computation. Thus such parameters *won't have any impact* on edge weight. This is because if the clients like

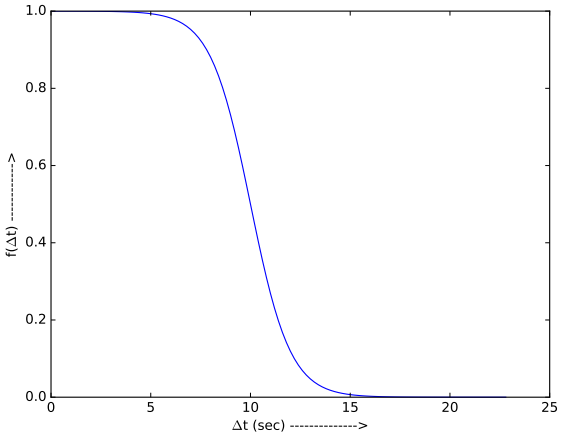


Figure 3.3: Damping function

solitude, it is highly unlikely that they would be accessing the object. They would be oblivious towards their placement in the clusters, and hence can be placed in any cluster (same or different). So considering only the 'friendship' parameter for their placement is sufficient.

- ***Dynamic Weight of the edges (DW):*** This weight captures the impact of the client-client interaction on the total edge weight. If two clients are interacting a lot on an object, they should continue to do so, even after partition. Hence upon partition, they should not be separated, and should be placed in the same cluster. This situation is practically realized by increasing their corresponding edge weight, thus increasing the probability of these clients being in the same cluster.

  Calculation of Dynamic weights: Our calculation is based on the assumption that if two requests arrive consecutively at the server, then the corresponding clients are interacting and their dynamic edge weight, *(DW)*, is updated. Initially $DW = 0$ for all the edges. Upon an interaction between two clients, their DW is updated as:

$$DW_{new} = 1 + f(\Delta t) * DW_{old} \tag{3.2}$$

  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 of these two clients. The value of this function should be close to 1 for small Δt and should decrease with an increase in Δt and approach 0 asymptotically (resembling the graph in Figure 3.3).

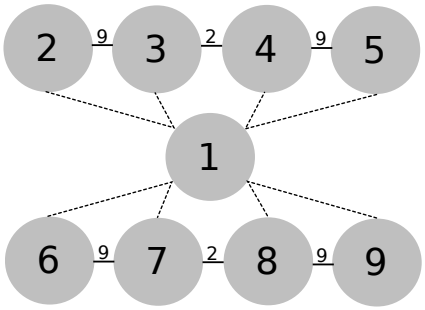
A variant of the Sigmoid function, $f(\Delta t) = 1/(1+e^{\Delta t - b})$ where b is a tunable parameter, is chosen as the damping function. This pattern appropriately captures (observed empirically) the client behavior such that if Δt is high then the corresponding clients are not interacting much and hence their edge weight should be small.

We use a weighted matrix, wt, to represent the weights, and $wt(c_i c_j)$ represents the total weight of an edge between clients c_i and c_j. These weights are computed as follows:

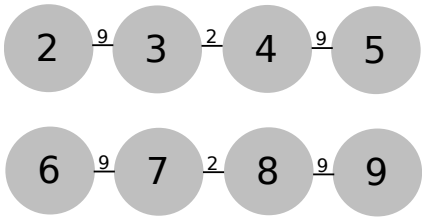
$$wt(c_i c_j) = \alpha * SW + \beta * DW \tag{3.3}$$

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

Upon overload, this graph is partitioned into different subgraphs by using Gpmetis [45]. After the partition, the clients across the clusters do not see each others' 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. Such clients are called as *"global nodes"*, and in our current prototype implementation, the application specifies this set of global nodes. Since these global clients would be present in both the partitions, they can be omitted from the graph while performing partitioning, and can be later added to both the clusters. We present an example, with 9 clients ($\{1, 2, ..., 9\}$ in Figure 3.4a) to explain the formation of clusters. The edge weights shown in the graph represent the combined weight of the static and dynamic components.



(a) Client Connectivity



(b) Connectivity after removing the owner

Figure 3.4: Social graph

Figure 3.4a shows that the client pairs $\{(2,3),(4,5),(6,7),(8,9)\}$ are strongly connected. In the graph, node 1 is the owner of the post and hence is connected to all other nodes. Initially, when there is no overload, all the clients can see each others' data, and are strongly consistent. After some time when the post becomes viral and causes a system

overload, the split operation is invoked (after removing the owner, Figure:3.4b), which partitions the clients into two clusters: $\{2, 3, 4, 5\}$ and $\{6, 7, 8, 9\}$, based on the min-cut algorithm. Later the owner is added to both the partitions.

- It is possible that the partition set created above might not be optimal. For example, when there are clients who send a disproportionate number of requests. Specifically, let us consider a situation in which the clients $\{2, 3, 4, 5\}$ are the hot clients and they are contributing the maximum to the load, while the remaining clients send very small number of requests. Our current partitioning mechanism does not offer any benefit as the data server with this partition is still overloaded, while the data server with other partition is lightly loaded, nullifying the effect of partitioning. This situation is handled by trading off social consistency with load balancing. So in such cases when there is a choice of favoring social consistency or overload handling, our system favors the latter. This is done by assigning weights to the nodes, called **Dynamic Node Weights** (DNW), which vary with time and represent each node's contribution in overloading the object. Dynamic node weights are computed similarly to dynamic edge weights, i.e., whenever a *client comments*, it's node weight is updated as follows:

$$DNW_{new} = 1 + f(\Delta t) * DNW_{old} \tag{3.4}$$

  Here, Δt represents the time difference between two consecutive comments by a client, and $f(\Delta t)$ is the same function but with a different value of b. Thus the node weight is based on the frequency of a client's comments and also captures the recency of comments.

  Similar to α and β for edge weight normalization, node weight has γ as its normalization factor, which is also provided by the application. Thus, upon normalization the node weight becomes $\gamma * DNW$.

In summary, the partition function ϱ, which is described above, should satisfy the following conditions:

(1) *Maximize social consistency and load balancing:*
$|\sum_{\forall \mathbf{c_i}, \mathbf{c_j} \in \mathbf{Cl_1}} \mathbf{wt}(\mathbf{c_i c_j}) - \sum_{\forall \mathbf{c_k}, \mathbf{c_l} \in \mathbf{Cl_2}} \mathbf{wt}(\mathbf{c_k c_l})| < \epsilon$, where $\sum wt(c_i c_j)$ 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 is a relatively small number).

(2) *Minimize the loss in social connectivity:*

$\forall \mathbf{c_i} \in \mathbf{Cl_1}, \mathbf{c_i} \notin \mathbf{Cl_2}, \mathbf{c_j} \in \mathbf{Cl_2}, \mathbf{c_j} \notin \mathbf{Cl_1}, \mathbf{c_i c_j} \in \mathbf{E}$: $\sum \mathbf{wt}(\mathbf{c_i c_j})$ is minimized at all points of time. This condition states that the partition should be a min-cut, thus minimizing the total weight of edges across the partitions.

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

Please note that the partition should provide social consistency and should concomitantly perform some load balancing, but it should prioritize load balancing when the load becomes prohibitive. Referring to our running example, the set of possible partitions can be $\{(2,3,6,7),(4,5,8,9)\}$ or $\{(2,3,8,9),(4,5,6,7)\}$ (assuming that the clients (2,3,4,5) are contributing equally to the overload and hence have same node weights). Then, the partitions in which (2 and 3) or (4 and 5) are in different clusters are not chosen, as it would result in a socially weak cluster. Later, if any of the partitions, let say, $\{2,3,6,7\}$ gets overloaded due to the clients 2 and 3, then social consistency is traded-off for load balancing, and $\{(2,6),(3,7)\}$ or $\{(2,7),(3,6)\}$ are the possible partitions. Thus, our system tries to provide good consistency on a best effort basis.

Chapter 4

System Design

Our system is based on the log data-structure, and adapts to the incoming workload by performing split and merge protocols. This Chapter presents the design and implementation of the system. Initially, the system architecture is described, followed by the split-merge protocols.

The system (Figure 4.1) comprises of:

(1) *The Clients,* which are added to the system via the client initialization messages.

(2) *The Data Servers,* which form a persistent storage layer. They monitor the incoming request rate and initiate a split protocol during overload, and subsequently initiate a merge protocol as the request rate reduces.

(3) *The coordinator,* which coordinates the split and the merge protocols. During the split operation, it chooses a least loaded helper server among the available servers. During the merge protocol, when there exists several split data servers with the reduced load, the coordinator chooses a set of data servers that are most efficient to merge. It also generates the routing information corresponding to the object's state (split-merge), which is used by the proxy servers for redirecting the client requests to the appropriate data servers.

The coordinator also stores the information of all the clients in the form of a social graph. Upon a split, this graph is partitioned into different social clusters, and the coordinator performs a reverse operation of combining the graphs during the merge protocols.

(4) *The Proxy Servers,* that act as a mediator between the clients and the data servers, and route the client requests according to the routing information provided by the coordinator. They also add the clients to the system, with the help of the coordinator.

The clients have a many-to-one mapping with the proxy servers, and are connected to them in a round-robin manner. The proxy servers are further mapped to the back-end data servers with a many-to-many mapping (the connections are not shown to keep the figure simple). The entire communication between the data servers and the clients happens via the proxy servers. This design is opted in contrast to the design, where the clients contact the proxy servers only to find the corresponding data servers, and then directly contact the particular data server for the data exchange. This alternate design decision would require the clients to maintain the object-to-dataserver (object's location) mappings. Upon every reconfiguration (split-merge), these mappings would change, and the clients would need to be informed about the updated mappings. These factors might result in the scalability issues. Thus to avoid these issues, the proxy servers are used as the mediator for all the data exchanges. Furthermore, since the proxy servers and the data servers both belong to the same data center, they have a small network latency which has a minimal impact on the performance. To prevent the proxy servers from becoming the bottleneck, multiple proxy servers are installed.

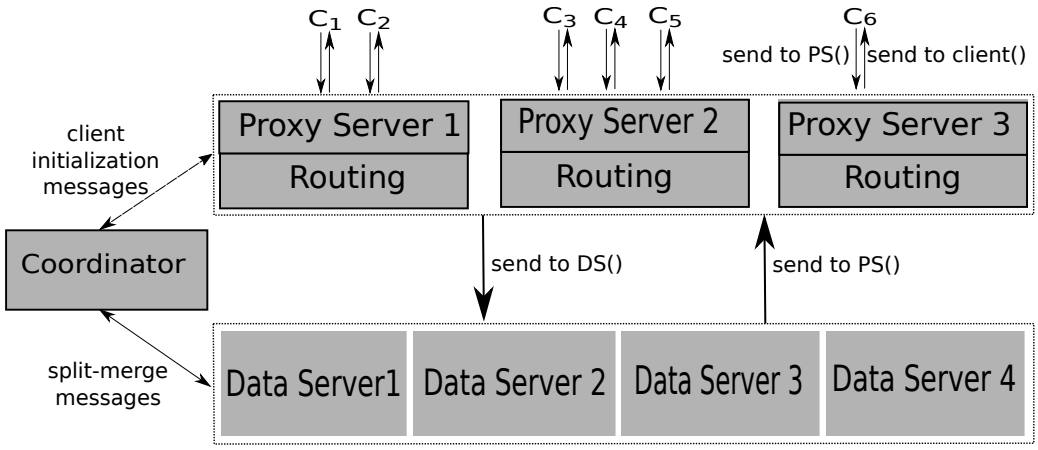


Figure 4.1: System Architecture

4.1 System APIs and the Split-Merge Protocols

The following algorithms show, how clients interact with proxy servers (PSs), and how the proxy servers further communicate with the data servers (DSs).

Algorithm 1: Write Client	**Algorithm 2:** Read Client
1 $send\_to\_PS(c\_id, W, data, o\_id)$;	1 $send\_to\_PS(c\_id, R, o\_id)$;
2 $recv\_from\_PS(Ack, o\_id)$;	2 $recv\_from\_PS(data, o\_id)$;

In the Algorithms 1 and 2, *c_id* denotes the client id, *W/R* denotes the request type (write/read), *Ack* is the acknowledgment received, and *data* is the information sent/received,

for the object whose id is *o_id*.

Algorithm 3: Proxy Server

```
1 while true do
2     recv_from_client(c_id, cmd, data, o_id);
3     cl_id = get_client_cluster(c_id, o_id);
4     s_id = get_server_id(cl_id, o_id);
5     send_to_DS(s_id, cmd, data, o_id);
6     recv_from_DS(s_id, ret_val, o_id);
7     send_to_client(ret_val, o_id);
8 end
```

Algorithm 4: Data Server

```
1 while true do
2     recv_from_PS(cmd, data, o_id);
3     if cmd = R then
          send_to_PS(data, o_id);
4     if cmd = W then
          update_value(data, o_id);
5      send_to_PS(Ack, o_id);
6 end
```

Similarly in the Algorithms 3 and 4, *cmd* represents a read/write request type, *cl_id* denotes the cluster id, *s_id* is the server id and is retrieved from the routing information stored at the proxy servers using $get\_server\_id(cl\_id, o\_id)$, *ret_val* represents either the data or the Ack.

In Algorithm 3, $get\_client\_cluster(c\_id, o\_id)$ returns the id of the cluster to which a particular client belongs.

The next section describes the split and the merge protocols.

4.2 The Split-Merge Strategy

During an overload, the object-based splitting is performed, which affects only the clients who are accessing the hot object. Upon a split, the clients belonging to different clusters are not allowed to access each others' data. However, all the clients can still interact over any other non-split object. Object-based splitting is favored rather than an entire system split, as the latter is expensive and also prevents the user interactions on different objects.

4.2.1 The Split Protocol

In the following algorithm, only a single split is considered, and the split of an object creates two sub-objects which can further overload, resulting in another split. This splitting can continue to any level. The protocol is described below (Figure 4.2):

(1) Each data server continuously tracks the incoming request rates on its objects. When the request rate for a particular object, let say "*o*", goes beyond a predefined threshold, the overloaded data server (Data Server1 in Figure 4.2), initiates the split protocol by requesting the coordinator for a split (represented by 1 in the Figure) and this data server

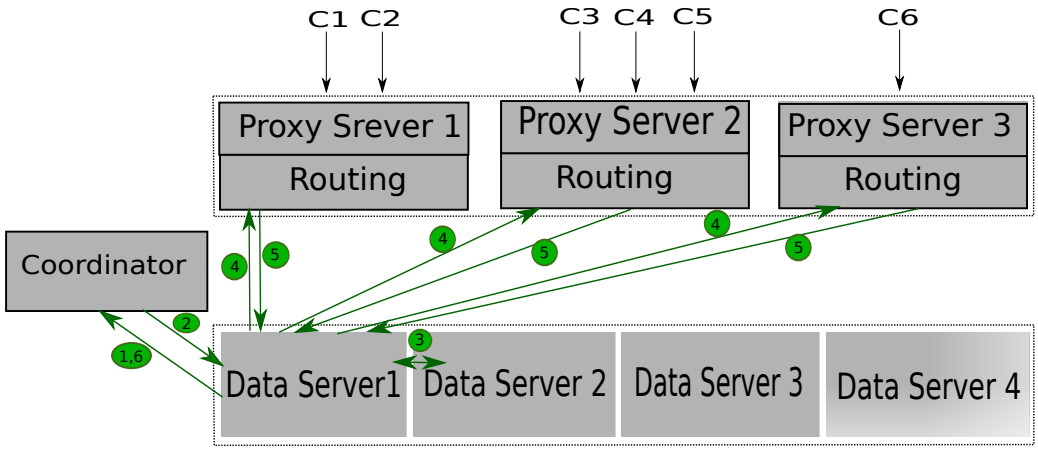


Figure 4.2: Split protocol

($DS1$) is referred as the master server.

(2) The coordinator, upon receiving the split request assigns a helper data server $DS2$ (referred as slave server), from the list of available data servers. It then partitions the existing set of clients into two clusters, and each cluster is bounded to one server, master and slave, respectively. This information is encapsulated in the form of a routing function and is sent back to $DS1$. In the case of unavailability of free data servers, the coordinator is unable to fulfill the split request, and rejects it.

(3) When $DS1$ receives the reply from the coordinator, it connects to the slave server ($DS2$), and iteratively transfers the data object to it. During the last iteration of the transfer, when the amount of data to transfer is sufficiently small, $DS1$ pauses its incoming requests and transfers the remaining data. After the transfer completes, $DS2$ sends an ACK to $DS1$.

(4) Upon receiving the ACK, $DS1$ resumes the previously paused requests, and asynchronously informs the proxy servers about the updated routing information. Asynchronous updates to the proxy servers do not impact the performance, since DS1 itself has the routing information. So, if a stale proxy server (following the old routing function) sends a request to $DS1$, which should have been handled at $DS2$ as per the new routing information, then $DS1$ forwards the request to $DS2$.

(5) After updating the routing information, the proxy servers reply to DS1 with End of Epoch (EoE) messages, indicating that all the subsequent requests will be routed according to the new routing function. Upon receiving the EoEs from all the proxy servers, the master

disconnects from the slave.

(6) Finally, $DS1$ informs the coordinator about split completion.

Figure 4.3 and Table 4.1 summarizes the protocol.

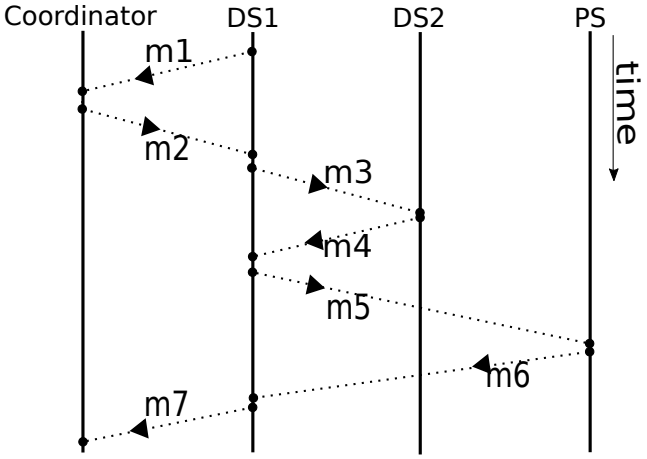


Figure 4.3: Time diagram for split protocol

Message	Content
m1	Split request
m2	New routing information and $slave_{id}$
m3	Cloned data
m4	Ack
m5	New routing information
m6	EoE
m7	Split done

Table 4.1: Messages in the Split protocol

4.2.2 The Merge Protocol

Figure 4.4 represents the merge protocol.

1) Every DS monitors the incoming request rate on its split objects, and report it to the coordinator when it falls below the merge threshold. As shown in the figure 4.4, $DS1$ and $DS4$ detect a low request rate and inform the corresponding rates to the coordinator.

2) The coordinator then sorts the received request rates (for a particular split object), in ascending order. Then the servers for whom the summation of request rates is less than a

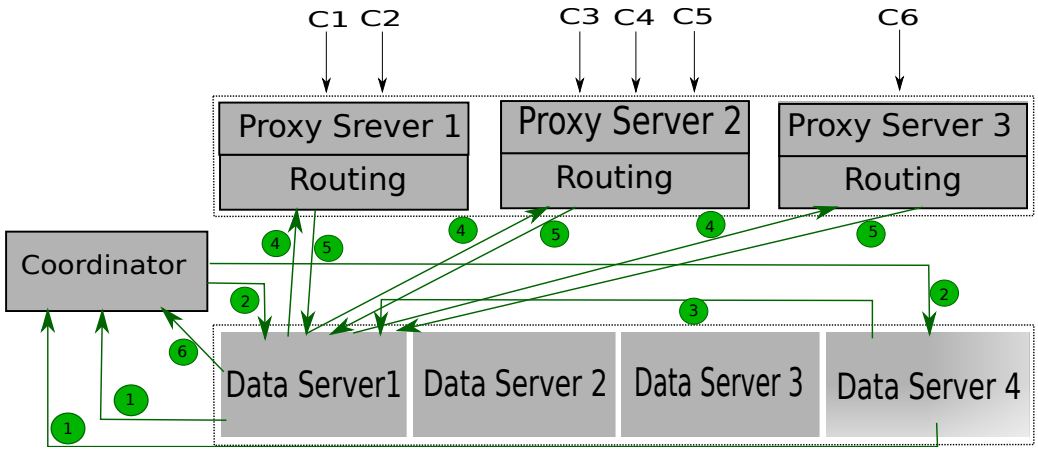


Figure 4.4: Merge protocol

specified global threshold, are considered for merging and are called slaves. One of them is chosen as the merge-master (DS1 in the example), on which the merged data will be stored. All the slaves are notified to begin the merge (represented by 2 in the figure).

3) Upon receiving the merge notification, these slave data servers ($DS1$ and $DS4$) iteratively transfer their data to the master ($DS1$). Master performs an iterative merge (merge-sort) on the incoming data. When the amount of data to be transferred is very less (δ), then the slaves block the incoming requests and transfer the state atomically to the master. Upon reception of these δ requests, the master temporarily pauses the requests on this object and merges them. Iterative merging increases the system availability as the system is blocked only during the last iteration which has a small amount of data to be merged. It also has minimal impact on performance since the merge protocol is initiated when the incoming load is small.

4) Once the merging is done, the master asynchronously informs all the proxy servers about the new routing function.

5) Proxy servers update their routing information and reply with EoE messages. After receiving all the EoEs, the master closes the connection with the slaves.

6) Master informs the coordinator about the merge completion.

Figure 4.5 and Table 4.2 summarizes the merge protocol. Please note that the current example considers only two servers, however multiple servers can be merged. Also, the current implementation performs merging only when the spike subsides. Though a background merger can be started, which will speed-up the merging.

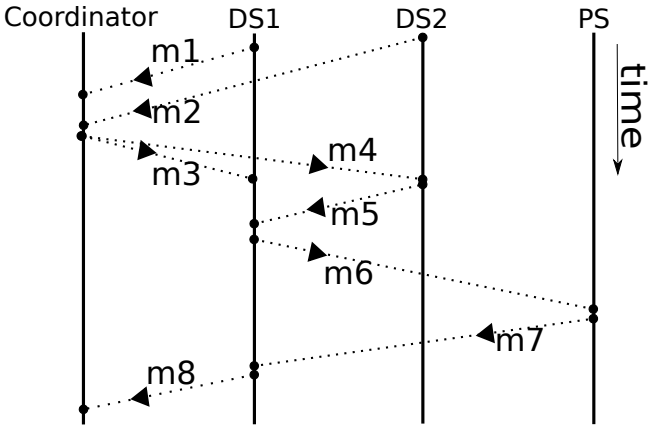


Figure 4.5: Time diagram for merge protocol

Message	Content
m1, m2	Merge request
m3, m4	New routing information, $master_{id}$ and $slave_{ids}$
m5	Data
m6	New routing information
m7	EoE
m8	Merge done

Table 4.2: Messages in the Merge protocol

System Unavailability time: Since both the protocols perform an iterative data transfer, the incoming requests are paused only during the last iteration, to perform an atomic transfer. The corresponding unavailability time for split (U_{split}) and merge (U_{merge}) protocols can be represented as follows:

- During the split protocol, the system is unavailable when the data for the last iteration is sent from $DS1$ to $DS2$. This time includes reading the data at $DS1$, sending it to $DS2$ (network delay), writing data to $DS2$ and finally returning to $DS1$. U_{split} in Table 4.3 represents this, where RTT denotes the Round Trip Time, $Read$ corresponds to the time taken to read on $DS1$ and $Write$ is the time taken to write on $DS2$.

Unavailability time	Value
U_{split}	1 Read + 1 RTT + 1 Write
U_{merge}	1 Read + 1 RTT + 1 Write + δ time

Table 4.3: System unavailability time

- During the merge protocol, the system is unavailable when the slaves send the data of their last iteration, and also when the δ requests of this iteration are merged. Correspondingly, U_{merge} is defined in Table 4.3 , where *Read* is the time taken to perform reads at the slaves (all done in parallel), $Write$ is the time taken to write on the *master server* and δ is the time spent in merging the data of the last iteration.

Logical Split Tree: The formation of various clusters during the split protocol follows a tree-like pattern, as shown in Figure 4.6. The leaf nodes of the tree structure (at any point in time) represent the existing clusters (at that time), and each cluster is mapped to a data server. Referring to the figure, initially all the clients belong to cluster 1, which is mapped to $DS1$. Upon the first split, cluster 1 is partitioned to cluster 2 and 3 which are mapped to $DS1$ and $DS2$, respectively. The process of splitting can continue to any level, with each split following a hierarchical structure.

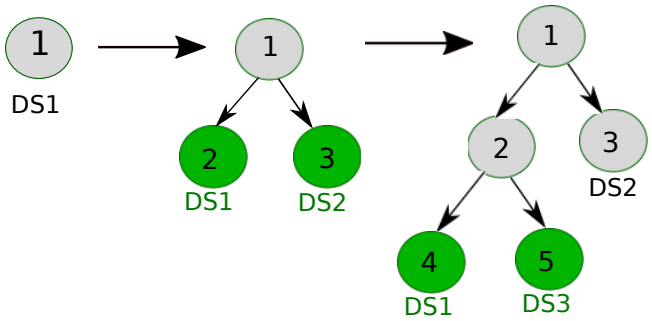


Figure 4.6: Hierarchical split

On the other hand, a flat merge technique (Figure 4.7), irrespective of the split structure, is followed. Particularly, after any split, all the clusters in the system are considered to be at the same logical level and merging is performed on any set of leaves (clusters), depending upon the workloads on them.

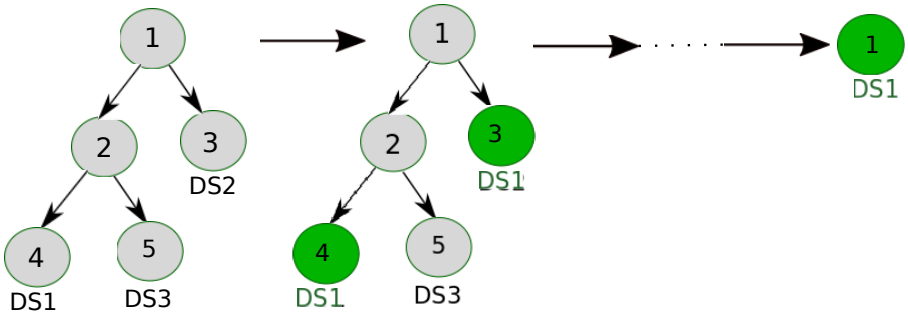


Figure 4.7: Flat merge

As shown in figure 4.7, the split state has three clusters: 3,4, and 5, mapped to data servers $DS2$, $DS1$, and $DS3$, respectively. During a merge, it is not necessary to merge the clusters 4 and 5 (i.e., $DS1$ and $DS3$), but the clusters 4 and 3 (or 5 and 3) can also be merged, if the workload on these servers is low. This merged data is then stored on any one of the two servers (let it be $DS1$). Further, when the workload on $DS3$ reduces, it can be merged with the previously merged data server ($DS1$), and finally, the entire data can be stored on $DS1$. This approach provides more flexibility as an arbitrary set of under-utilized versions can be merged to reduce the resource usage. However, this flexibility comes at the cost of merging a large amount of data. Precisely, if a hierarchical merge is done (i.e merging clusters 4 and 5 in Figure 4.7), it would require merging only the data differences which originated after cluster 2 was split, i.e the data created during the paths [2 - 4] and [2 - 5]. But in the case where 3 and 4 are merged, the logical paths [1 - 2 - 4] and [1 - 3] are merged. Later when another merge happens (from cluster 5), then the part [1 - 2] from path [1 - 2 - 5] is re-merged (though it was already considered in the previous merge), which is redundant. However, since the request rate is quite low during a merge, this additional work has a minimal impact on performance.

4.2.3 System Goals and Semantics

- *Adaptability and Scalability:* The system is self-optimizing, and efficiently adapts to the variations in the workloads. It is scalable and dynamically adds more servers when the workload increases, and removes additional servers in the case of reduced load.

- *Semantics:* As mentioned in section 3.3, the system provides the following guarantees:

  - *Read your own reads/writes:* The clients can read their previously read/written data in the same order. Upon a split, a clone of the object is created and transferred, so irrespective of the client's cluster, it can read all of its past data. Similarly, after a merge, it can view its entire data in the same order. Though there can be some additional data as a result of merging, however, the client's order remains intact.

  - *Monotonic reads/writes:* If a client reads/writes a value v from/to an object, any successive read operation on that object by the same client will always return, either the value v or a more recent value.

Chapter 5

Evaluation

The proposed system is favorable for the applications that are based on log data structure (for example, a user's Facebook wall, a Twitter page, or a list of comments for a social media post). So, this thesis mainly focuses on social-networking applications and implements a mock social application, BLAST (Best-effort Latency And Scalability Together), based on the architecture described in chapter 4. BLAST targets to handle overloads and achieve scalability, and it supports multiple clients, who issue read/write requests to the posts and comments. A post combined with its comments forms a log-structured object. The system provides the following APIs:

- Add-Me: Connects the clients to the system and assigns unique ids to each of them.
- Write-Post: Issued by the clients to create a new post. In the current implementation, a post is considered as an object.
- Write-Comment: Allows the clients to comment on the existing posts. A comment is not considered as an object in itself but is appended to the corresponding post object.
- Read-Post: Reads the post along with its 50 latest comments from the data server.

The prototype is implemented over the open-source Cassandra [11] database and it stores the posts as key-value pairs: post-id is the key, and the comment-list forms the corresponding value.

In this chapter, we present the cost-benefit analysis of the proposed paradigm with respect to performance and social consistency. BLAST is compared against Cassandra to show the performance benefits. For comparing the benefits of social partitioning, BLAST is evaluated against another load balancing system which performs random partitioning of the clients.

The experiments presented in this chapter initially evaluate the system on a synthetic benchmark with three types of workloads: i) Write-Only, ii) Read-Mostly (95% Reads, 5% Writes) and iii) Mixed (50% Reads, 50% Writes) for a single split. Next, they show the results for a multi-split scenario, pointing towards the scalability of the prototype. Further, the results against two real-world datasets are presented. The datasets are collected using i) Facebook's API: comments on a famous live video and ii) Twitter: Higgs dataset [46], which also provides the connectivity information among the clients in terms of "who-follows-whom" relationship. This who-follows-whom relationship is considered as bidirectional, i.e if a client A follows client B, it is assumed that B also follows A, and the two clients are considered as friends. This friendship parameter is used as a static parameter. Only one static parameter (i.e. friendship) is considered in the social graph, and its priority is set to one. So in this case, the static edge weight between A and B is set to one. The request rates of these datasets were low in comparison to that required for the system saturation, so the request rates are scaled to cause overloading.

After the performance evaluation, the experiments comparing the quality of the system are performed. Finally, the experiments demonstrating the quality-performance trade-off, while using the dynamic weights scenario, are presented. In the experiments, Gpmetis [45] (graph partitioning package), is used to perform the social split. The input to this tool is a weighted graph, whose static edge weights are provided by the application while dynamic node and edge weights are computed during the execution.

Experimental Setup: The experiments are performed within a single data center equipped with Intel Xeon, Core $i3$, 2^{nd} Generation processors, each having $four$ cores ($eight$ with hyper-threading), 16 GB RAM, 3 TB Seagate HDDs (7200 rpm) and are connected by Gigabit Ethernet (GbE). Average ping latency among the machines is $\approx 0.17ms$. In each experiment, three dedicated machines are used for proxy servers, one for the coordinator, three for all the clients, and one, two, or four for the data servers (depending on the number of clusters created after splitting).

The data servers run Cassandra 3.9 in the backend, and store data without replication, with a consistency value of one (i.e. the reads/writes are performed on any one server before returning). Initially, when there is no split, a single data server handles the incoming workload, thus allowing all the clients see a consistent value with respect to each other, and makes the system strongly consistent. Upon an overload, another database with the same schema is created (on another data server), and the object data is copied. These two servers act as independent clusters, without any replication (replication factor:1) and consistency value:1, i.e.,

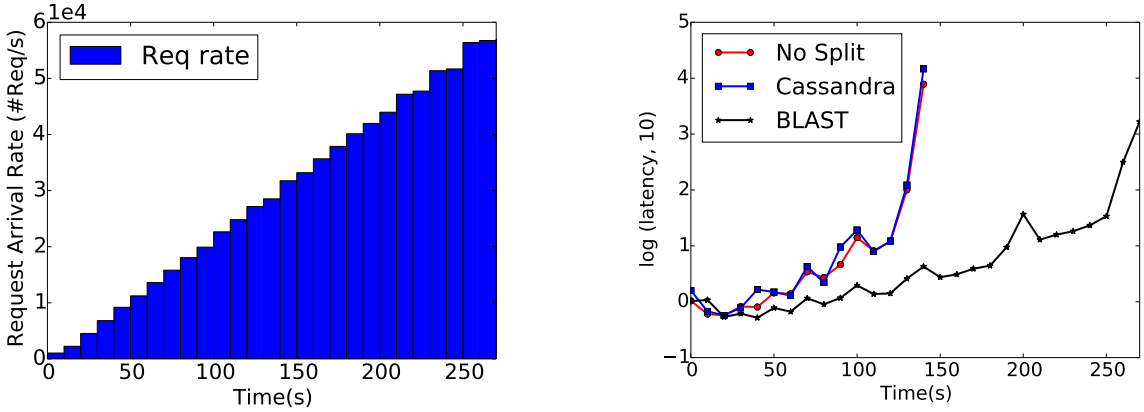
they do not communicate. Upon subsequent overloads, this process is repeated without altering the replication factor and consistency value. The new clusters formed are strongly consistent within themselves and the system as a whole is socially consistent. As the workload decreases the clusters are merged resulting in an eventually consistent system.

The performance of the system is evaluated in terms of latency and throughput, and is compared 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 equal to 1. Upon any subsequent overload, the replication factor is incremented by 1 (after adding another data server to the cluster), but the consistency value remains 1 throughout, unlike BLAST. This means that the request is satisfied from any one server rather than multiple servers. The data is replicated to other servers asynchronously at the backend, which results in an eventually consistent system.

In the synthetic workload, the request rate increases/decreases in steps. In particular, the clients continuously issue requests at a rate that is approximately constant for some time interval, and the rate changes in the next interval. Each write request appends a comment to a post, and each read request reads a list of the latest 50 comments from that post and returns them back to the client. The experiments performed consider only one post, in order to show the impact of hot-spots. Please note that the system is an open system, i.e. the request arrival rate at the system is independent of the system performance and there is no feedback mechanism to control the request rate.

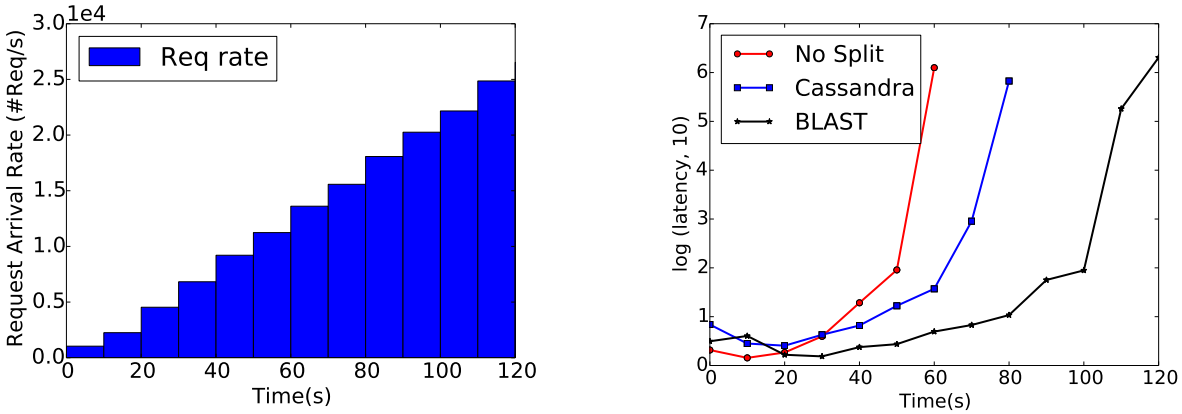
Experiments and Inferences: Initially, the system threshold is calculated for all types of workloads, by disabling the split protocol. Then the experiments are performed with split enabled, and the split is done when the incoming request rate reaches $x\%$ of the system threshold, where x is tunable, depending upon the workload. Experiments with different values of x have been performed and it is found that both Cassandra and BLAST finish their split protocol before the system overloads when the value of x is 60%. Thus, this value is chosen in all the experiments.

Figures 5.1a, 5.2a, 5.3a show the arrival request rate vs execution time for a *Write-Only, Read-Only and Mixed* workloads, respectively. Their corresponding latency graphs (plotted on log_{10} scale) are represented in Figures 5.1b, 5.2b, 5.3b. Each latency graph has three curves, representing the base case (non-split, i.e., the system has only one cluster), Cassandra and BLAST, both with one split each (i.e., the system has two clusters in each case). After a split, both Cassandra and BLAST use double system resources to handle the incoming requests. Figures show that BLAST handles $(1.7 - 1.9)\times$ more workload in comparison to the base case,



(a) Request rate (b) Latency curve

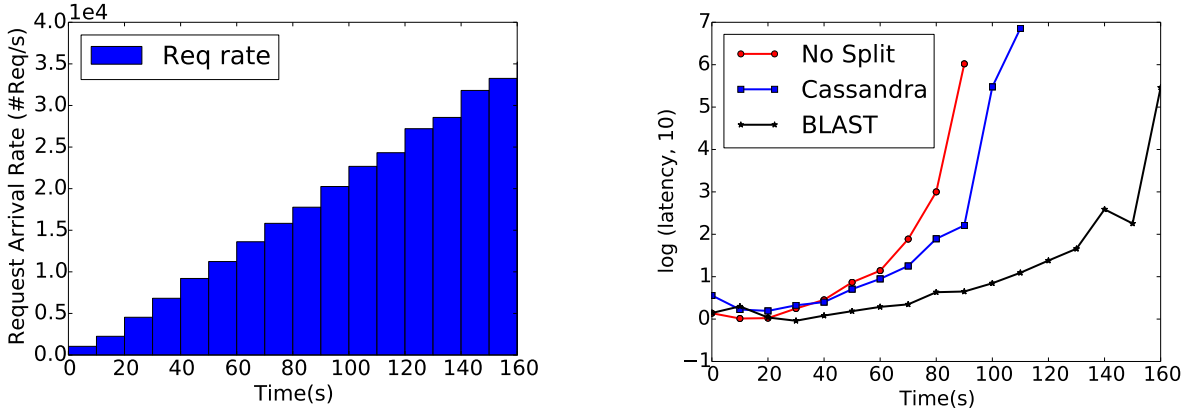
Figure 5.1: Write-Only workload



(a) Request rate (b) Latency curve

Figure 5.2: Read-Mostly workload

and 1.6× more workload in comparison to Cassandra. Cassandra has a negligible performance gain, $(1-1.2)\times$, showing that it does not perform effective load balancing when there is a single hot object.



(a) Request rate

(b) Latency curve

Figure 5.3: Mixed workload (1 split)

If the load is not adequately balanced between the data servers, then these data servers have to communicate a lot among themselves, to correctly respond to the incoming client requests. An experiment has been performed, to quantify the impact of load balancing on performance, in which the communications (that happen when the client requests arrive at the data servers) between the data servers are tracked. In particular, the total number of read and write requests that **arrive** (directly from the clients) at each data server is tracked. Along with this, the total number of reads and writes which **actually happened** at them are also tracked. The difference between the two values represents the corresponding number of internal communications. The results in Figure 5.4, show that for every 1000 incoming requests, Cassandra performs more than 450 communications (in case of both the read and write requests). In contrast, BLAST data servers handle the requests locally without any communication. This huge amount of difference in the number of communications reflects the difference in the performance numbers.

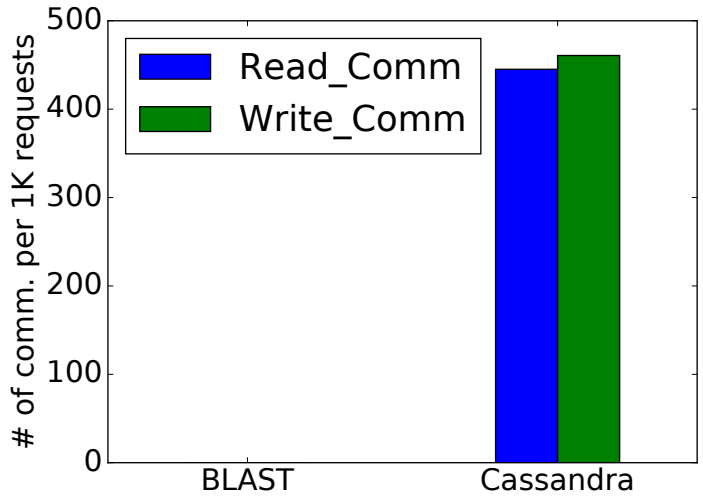
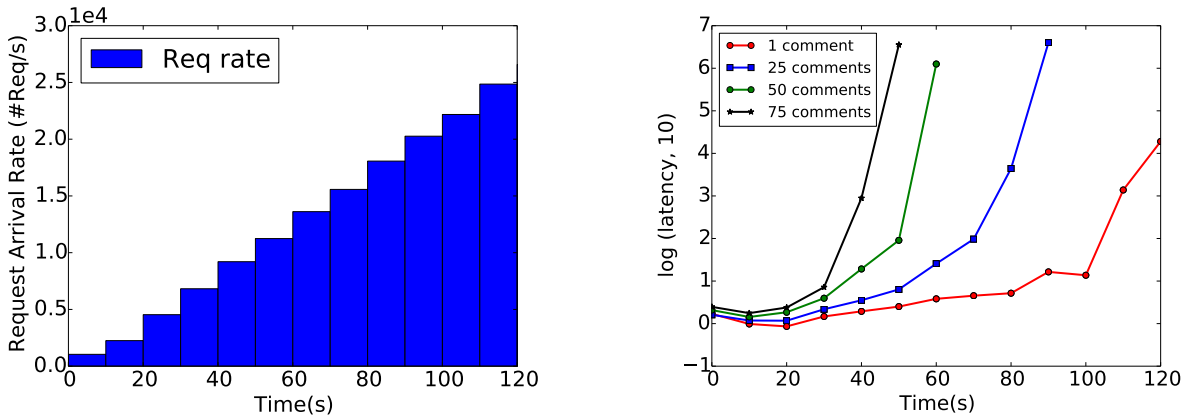


Figure 5.4: Number of communications per 1000 incoming requests (1 split)



(a) Request rate

(b) Latency curve

Figure 5.5: Read latency with number of comments (Read-Mostly workload)

Figures 5.1 and 5.2 show that BLAST handles $(2 - 2.5)\times$ more Write-Only workload than Read-Mostly workload. The intuition behind this is that in Read-Mostly workloads, the performance depends upon on the number of comments that are read from the server, in response to the issued read request. The latency of a read request increases with increase in the number of comments read by it. To verify this, a non-split experiment with the Read-Mostly workload is conducted, in which the number of comments read from the server are varied in the set $\{1, 25, 50, 75\}$. The results in Figure 5.5 confirms the explanation. The read latency depends on how the underlying storage handles the incoming requests. Since Cassandra is used

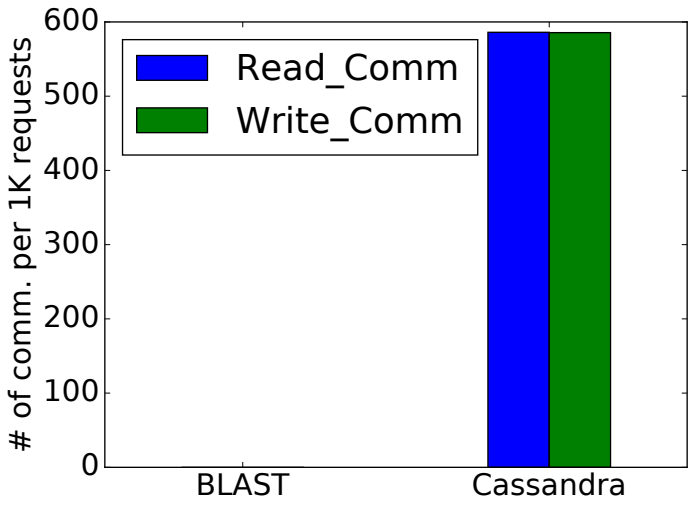
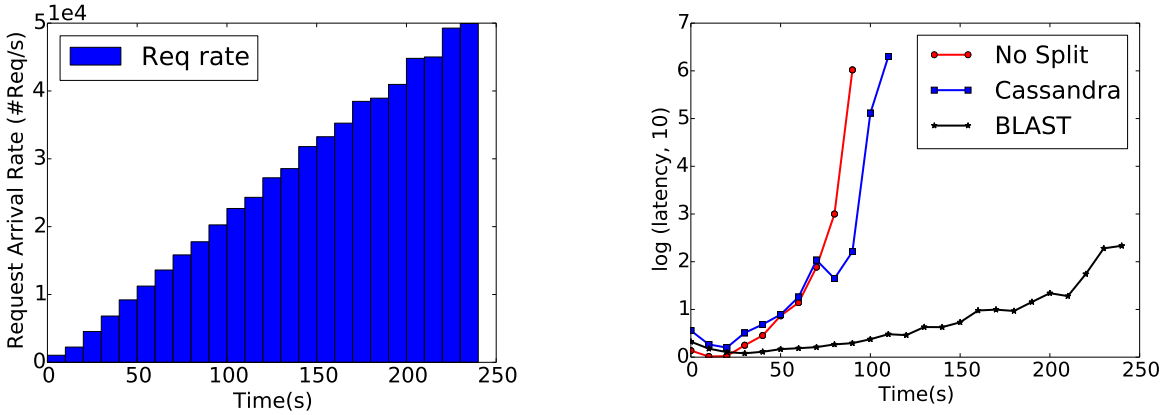


Figure 5.6: Number of communications per 1000 incoming requests (3 splits)

at the backend, so this work does not focus on the details of how read requests are internally handled by Cassandra. For all the future experiments, the number of comments to be read is set to 50, as this value is realistic and provides a reasonable performance.

Henceforth, the Mixed workload is used for all the future experiments as the other workloads show the same pattern predictably.



(a) Request rate

(b) Latency curve

Figure 5.7: Mixed workload (3 splits)

Figure 5.7 presents the scalability perspective of the system by showing the results of a multi-split scenario: initially, there is only one cluster in the system, and after subsequent splits, the number of clusters increases to four (Figure 5.7). The experiments demonstrate that BLAST surpasses Cassandra by handling 2.4× more workload after performing multiple splits.

The number of communications across the clusters in this case follow a similar pattern as in the single split scenario (Figure 5.6), thus inferring poor performance for Cassandra.

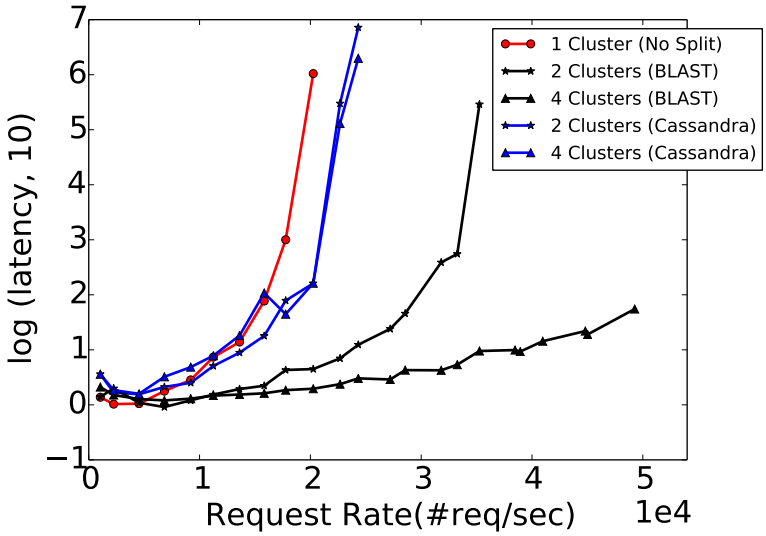
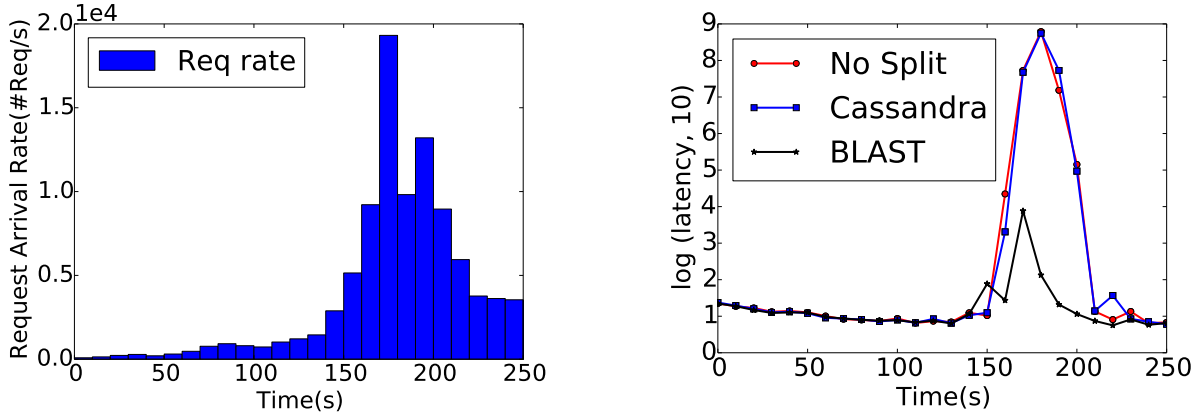


Figure 5.8: Summarizing all 3 splits

Figure 5.8 summarizes the performance of all the systems under the mixed workload, and clearly shows that Cassandra is unable to handle single object overloads. In contrast, BLAST scales with the offered load and surpasses Cassandra by handling $(1.6 - 2.4)\times$ more workload.

Further, the experiments are conducted with two real-world datasets. Figure 5.9 presents the arrival request rate and the corresponding latency graph, considering the *Twitter dataset* [46]. The graphs show that the results are in line with the previous results obtained in the synthetic workload scenario. The graph shows a spike in the latency curves, which occurs due to queuing, as the request rate spontaneously increases from 9000 req/s to 18000 req/s (at the time interval of $160 - 170s$). BLAST quickly handles the situation (in $\sim 20s$) and becomes stable after splitting, while Cassandra provides poor performance throughout, and the performance is improved solely when the request rate decreases.



(a) Request rate

(b) Latency curve

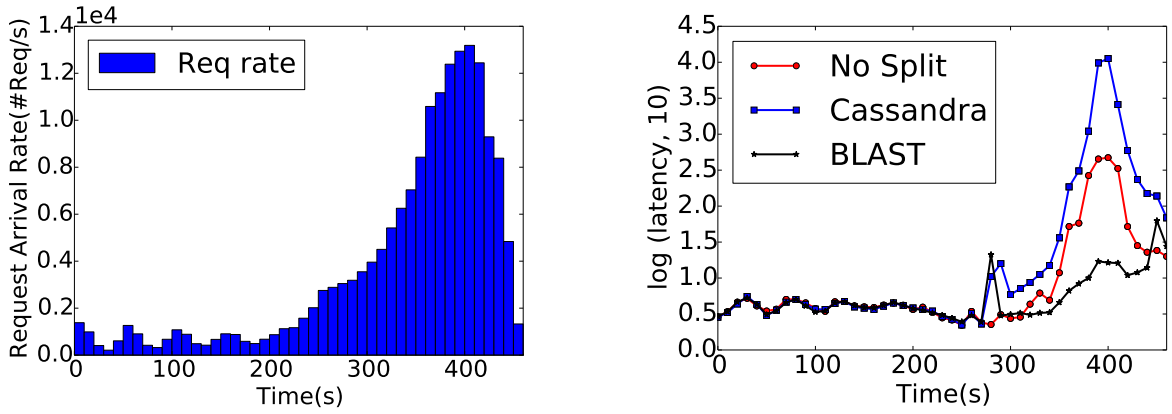
Figure 5.9: Twitter dataset

Similarly fig. 5.10 shows that BLAST outperforms Cassandra on the *Facebook dataset*. In this dataset, Cassandra's performance (despite a split) is even worse than the non-split scenario.

All the above experiments show the impact of load balancing and clearly demonstrate that BLAST performs better than Cassandra. Further, the thesis presents the experiments which demonstrate the impact of social partitioning and claim that BLAST provides a good Quality of Experience (QoE) to its clients. To quantify the quality experienced by the clients, a new consistency metric:"***Quality***" is introduced, which can be computed for the response of each read request. Theoretically, the quality of a read response is high, if the response contains all the previous writes done by the client's social connections. Thus, "Quality" is defined as the ratio of number of friends' comments (writes) received ($Friends\_Writes_{Received}$), to the total number of comments done by client's friends ($Friends\_Writes_{Done}$). Formally, the quality of a read request, R_i, is:

$$Quality_i = \frac{Friends\_Writes_{Received}}{Friends\_Writes_{Done}} \tag{5.1}$$

An experiment determining the quality is performed to test the effectiveness of the par-



(a) Request rate

(b) Latency curve

Figure 5.10: Facebook dataset

titioning algorithm. This experiment (Figure 5.11), uses the Twitter dataset and compares BLAST against another load balancing system model. The new model handles the overloads by partitioning the clients into clusters, as done in BLAST. However, unlike BLAST, it partitions the clients randomly rather than on the basis of social connectivity. This system is named '*Random*', and it focuses only on handling the overloads while completely ignoring consistency, as is the case with general load balancing algorithms. In the experiments, the "Quality" of a split system (for both BLAST and Random) is computed and then normalized to the quality obtained for a 'no split' system (Equation 5.2). In particular, initially the number of friends' comments received in a 'no split' system are computed. Next, with the same incoming request rate trace, a partitioning algorithm is applied to compute the number of friends' comments received.

$$NormalizedQuality_i = \frac{(Quality_i)_{split}}{(Quality_i)_{no\_split}} \tag{5.2}$$

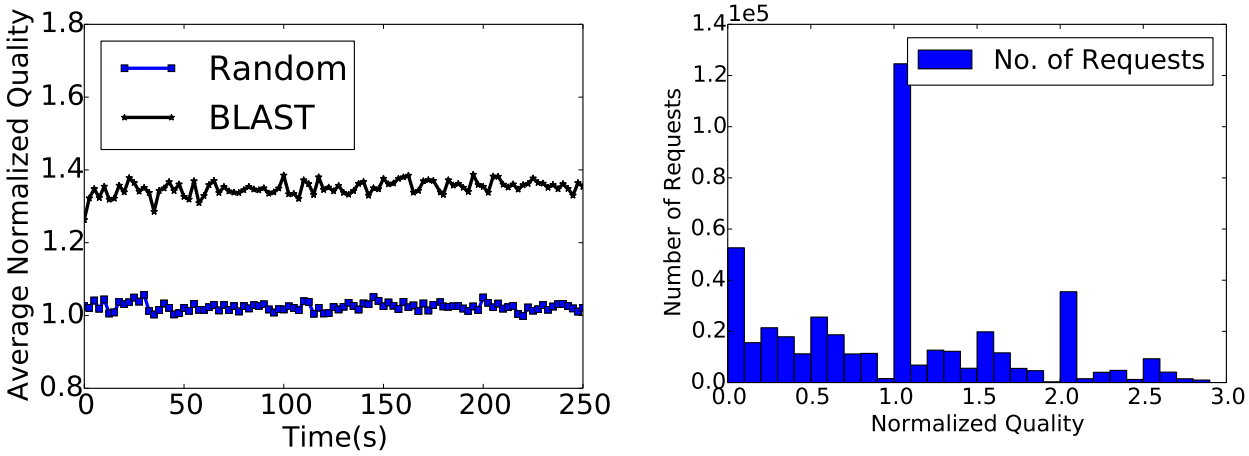
Since the normalized quality in equation 5.2, is defined for each read request, the average normalized quality for all the requests issued during some time interval can be computed. The $AvgNormalizedQuality$ for an interval (with n requests) is defined as:

$$AvgNormalizedQuality = \frac{\sum_i^n (NormalizedQuality_i)}{n} \tag{5.3}$$

Figure 5.11a shows that the average normalized quality of BLAST is 1.37× higher than Random.

Figure 5.11b presents the number of requests that receive a particular quality in BLAST.

The quality values are plotted on the x-axis and the number of requests are plotted in the y-axis. The graph shows that the maximum number of requests have a quality value of 1, implying that maximum clients get high quality of response. The quality for all requests is in [0,3]. A value greater than one implies that the clients are able to see more of their friends' comments in comparison to the number of friends' comments seen in the base case (no split). This happens due to social partitioning, which replaces the received comments of non-friend clients with the comments from friends. In contrast, a value smaller than one indicates that the quality of BLAST is less than that of 'no split' scenario. This is because BLAST provides social consistency on a best effort basis, and in case of worse situations (when load is very high), BLAST favors load balancing. Thus the friends are partitioned to different clusters to effectively balance the load, and this reduces the quality. Figure shows that only 30% of the requests have a quality reduction by more than 50% which means that only a small percentage of clients do not get a good quality.

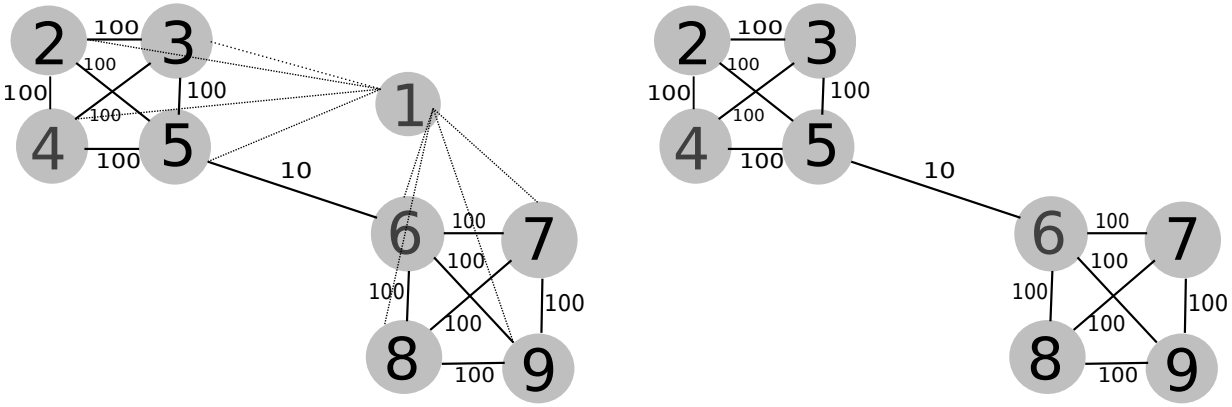


(a) Average quality

(b) Clients QoE statistics

Figure 5.11: Quality of response

Finally, this chapter presents the quality-performance trade-offs in the static and dynamic splits, i.e., BLAST is evaluated in two settings: considering only the static weights (static split), and considering both the static and dynamic weights (dynamic split). Figure 5.12 shows the input social graph which is used in the experiment. In the graph, client 1 refers to the owner of the post and hence is a "*global*" node. The graph partitioning algorithm is performed after removal of this node (fig. 5.12b). The weights shown in the graph are static, and the nodes {2,3,4,5} contribute maximum in the system overload. Since there are very few clients, they could not overload the system so we reduced the number of cores to 1 (from 4 in our previous



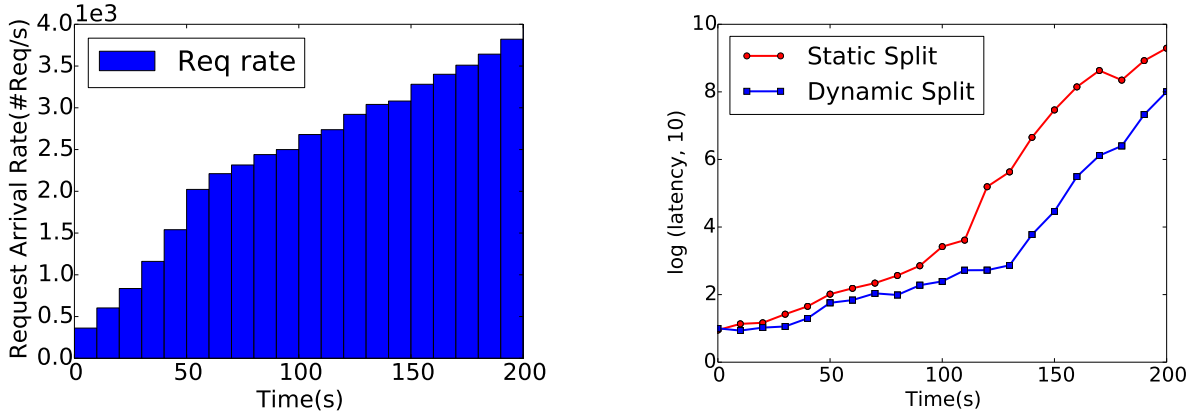
(a) Input social graph

(b) Graph after removing the owner

Figure 5.12: Social graph

setup), thus decreasing the maximum achievable request rate to 4000 req/s. (see Figure 5.13)

If only static weights are considered, then the partitioning algorithm leads to the partitions {2,3,4,5} and {6,7,8,9}. However, considering the dynamic weights, the partitioning algorithm can lead to any partition which balances the number of nodes which are causing the maximum overload. For example, {2,3,8,9} and {4,5,6,7} is one such possibility. The partition will also depend on the dynamic edge weights, and hence might change depending upon the amount of interaction among the clients.



(a) Request rate

(b) Latency curve

Figure 5.13: Static vs dynamic performance

Figure 5.13 shows that the dynamic split provides better performance, though the quality offered (Figure 5.14) in this case is slightly lower than the static case. This reduced quality is because BLAST tries to provide both excellent load balancing (leading to better performance) and good social consistency (leading to better QoE) wherever feasible, however it favors better performance by compromising upon social consistency in case of critical situations.

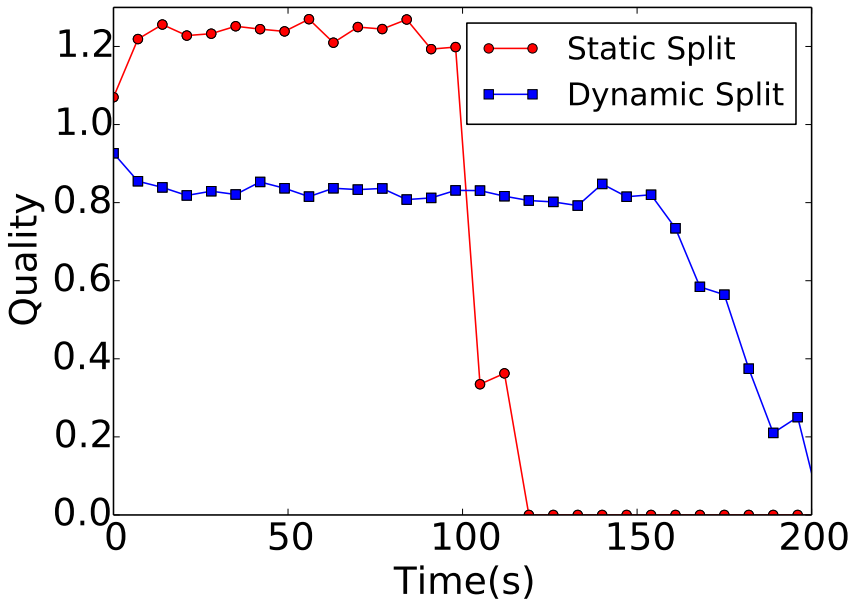


Figure 5.14: Static vs dynamic quality

Figure 5.14 shows that the average normalized quality (as defined in equation 5.3) of the static split is 1.2 for some time, and then it suddenly drops to 0. This sudden drop is because as the request rate increases, the system gets overloaded, which results in extremely high latency and hence no requests complete. Dynamic split increases the system performance, and the system runs for a longer duration. Summarizing, the static split provides good quality, while the dynamic splits provide a slightly reduced quality but at a higher performance.

Chapter 6

Discussion

This chapter initially presents some discussion related to the priority order of features which are used in static weights computation (as referred in 3.4). Later, the chapter discusses various types of systems which are possible by using the social partitioning technique, and these systems vary in performance and consistency. Then, the significance of the log data structure is discussed and the chapter emphasizes on how most of the data objects can be represented by its operation log. Finally, the chapter briefly describes the various real-world applications that can use the approach of social consistency.

6.1 Possibility of different preference orders

Section 3.4 describes the computation of the edge weights, based on the priorities assigned to the features. In the experiments, it has been assumed that the application provides this preference order, though other scenarios with different preference orders are possible, and multiple preference orders can exist. For example, it is possible that the clients provide their own preferences, thus each client prioritizes the features differently. In this setting, as each preference order is a client's personal choice, it should only impact her social strength with her peers without affecting the social strength between other clients. The model proposed in this thesis can easily incorporate this situation as explained below:

The application provides a list of features and seeks clients for their preferences. The features have a rank range (decided by the application), of the form $[a, b]$ with $a < b$, higher rank means more importance for that user. The range has been set to normalize the features, so that no feature can dominate the other.

Ranks provided by the users are then directly incorporated as edge weights using the same formula, $Edge\ weight = \sum_{i=1}^{s} p_i * x_i$, where p_i is the rank of the feature and x_i is

the value of the feature. For example, consider two clients $(c1, c2)$ and location as a feature (range [0-1]), with c_1 giving a rank of 1 while c_2 giving rank 0.2. Then the combined rank will be the mean of these (i.e., $(1 + 0.2)/2 = 0.6$), and hence the edge weight will be : $0.6 * (value_{if\_they\_belong\_to\_same\_location\_or\_not})$.

Apart from this scenario, another scenario is possible in which the application asks the owner of the post to give her preference about the way she wants her post to be seen by different clients. Though this is not a practical scenario as an application uses preference orders to partition the social graph only when an overload situation arises. Overloading is caused because the application is unable to properly manage the incoming load. Thus the problem is at the application side and the onus is completely on the application. The owner of the post should not be bothered to give her preference to handle a situation which arises due to application's inability. Moreover, the owner has no incentive to give her preference. Thus this situation is highly unlikely from a practical perspective.

However, if such a hypothetical case is considered where, upon an overload the application asks the poster to give her preference, then there would be two possible priority orders (one by the application and other by the owner) and a global order is to be chosen. According to the classical Arrow's Impossibility Theorem [47], this assignment of global ranking over parameters is not possible. Though there can be some alternatives by which Arrow's theorem can be avoided, this work does not go further into the details. In such cases, it is assumed that the application uses the owner's preference order rather than assigning its own order. Thus in summary, only the following three situations are possible and can be easily handled by the proposed system:

(1) The application provides the preference.

(2) Clients accessing the post can provide their own preferences.

(3) Owner of the post can provide her preference (not very practical).

6.2 Significance of Social Partitioning

The core idea of the social consistency model is the client-client relationships, on the basis of which the clusters are formed. Clustering helps to quantify the per client consistency and per client QoE, depending on the data a client would desire to access. Although this thesis uses the concept of social partitioning to handle overloads, this technique can also be used in general,

to develop systems with different performance and different types of consistency among the clients.

For example, a system targeting high overall performance can have strongly consistent (SC) clients within the cluster and eventually consistent (EC) clients across the clusters. This is similar to the system focused in the thesis, but unlike our system where the partition happens only upon an overload, this system will have such partitions throughout the execution. Another possible system is where all the clients (even in different clusters) are strongly consistent, however, this is achieved by trading off performance for some clusters. In particular, the clusters in this new system are prioritized and the maximum priority cluster has the highest performance. A recent article [48] describes that the National Stock Exchange (NSE) has a similar model, where the brokers form a cluster with high priority (priority is on the basis of the location of their machines from the exchange trading system), benefit a lot from lower latencies and faster execution of trades.

The cluster priority can be an important parameter for deciding the performance of the system and the consistency among the clients. If the clusters have different priorities, then two types of systems are feasible : i) strongly consistent throughout, ii) strongly consistent within a cluster and eventually consistent across the clusters. On the other hand, if the clusters have same priorities, they will have similar performance. However, in such a setting, it is not possible to provide strong consistency to all the clients in the system, and the consistency across clusters has to be traded-off to maintain high performance. This results in a system with strong consistency within a cluster and eventual consistency across the clusters. Table 6.1 summarizes all such feasible systems under different cluster priority and performance setting. (In the table, SC refers to strong consistency and EC is eventual consistency).

	SC within, EC across	SC throughout the system
Different priority clusters (different performance)	Feasible	Infeasible
Same priority clusters (high performance for all)	Feasible	Feasible

Table 6.1: Different feasible models based on social partitioning

It is also possible to have a composite system in which different approaches can be adopted at different times during the system execution. For example, when a system gets overloaded for the first time, the clusters can be created on the basis of priority. Further, if a cluster with the highest priority gets overloaded then it can be partitioned into same priority clusters, providing high performance and social consistency. Thus, depending upon the application's requirement a corresponding model can be adopted.

6.3 Modeling Object's State by its Operation Log

In the proposed model, the object is assumed to be in the form of a list. However, the approach can still be applied to a non log structured object, if the object's state can be modeled by its operation log. In particular, if the state of an object can be easily recognized by parsing and replaying its operation log, then the object can be completely replaced with its log, and can be scaled out by splitting and merging the log.

Logging of the client requests makes merging of the object states extremely easy. Even when an object is in the split state and has workloads across different data servers, a global logger can be used to log and forward the requests to the data servers. During merge, this global log can be read and reconciliation can be done depending on the application.

If rebuilding of the object state is expensive, the object can be maintained in memory as a view, with an operation log as back-end. Merging can be performed by only replaying the logs that have been created during split [49].

Further as an optimization, instead of logging the entire request in the global logger, a global sequencer can be used to assign unique sequence numbers to the request, and perform the logging locally at the data servers. In the current implementation, timestamps have been used as sequence numbers. Since the application is assumed to run in a single data center (4), the clock synchronization of data servers can be easily done using NTP (Network time protocol) [50].

6.4 Extending Splittable Logs to Generic Applications

The approach of social consistency is generic, and can be applied to various applications other than Facebook and Twitter:

- Goodreads: An online catalog of books, where the users can review the books, and based on the reviews, other users decide if they want to read them too. The launch of a new book can result in a burst of reviews by users which creates the hot-spot. The social consistency can be applied here, and only the reviews of user's friends can be shown to a user, as the reviews will have more impact on the decision.

- Shared word documents: When a large number of users are simultaneously editing a large online document. Then the users can be partitioned on the basis of the *region* of the document they are accessing. In this, the social parameter is the 'same region'.

- Online Auction: Clones with the same initial value can be created upon the increase in request rate. As time progresses the clones handle the requests independently and are

allowed to diverge. Occasionally the clones can be synced. Once the load decreases, they can be made consistent and the winner can be declared. Since it is an online system, the end users will get an impression of someone bidding a higher price suddenly.

- Popular movie ticket: Simple application partition strategy can work here, where the object (movie tickets) can be fragmented. For example, if there are 100 movie tickets, they can be divided onto 4 systems (25 tickets at each), each system handling the users independently.
- Youtube/Quora: Similar to the Facebook/Twitter example described in the thesis.
- Online Shopping Site: Consider an example in which a new phone is launched in the market, and its review page gets overloaded due to enormous reviews by the users. As in Goodreads, But a particular user's decision will be greatly impacted by the positive/negative reviews of her friends. Thus in this overload situation also, the users can be shown only partial reviews.

Chapter 7

Conclusion

In this paper, we proposed a new distributed consistency model based on social relationships between the clients. 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, BLAST, and evaluated it using synthetic and real-world datasets. BLAST provides extremely low latencies as compared to the state of art techniques while giving a good QoE to most of the clients. We have leveraged the min-cut graph partition algorithms to find the best partitioning of the clients based on static and dynamic features. We compared BLAST to Cassandra database system and our experiments show that BLAST outperforms Cassandra by a handling higher workload, particularly 1.6× (upon one split) and 2.4× (upon three splits). BLAST provides this performance along with 37% better quality of experience, unlike many state-of-the-art systems. Our system provides consistency as a best effort and favors load balancing in critical situations where we have to choose between load balancing and social consistency.

7.1 Future Work

Currently, BLAST handles single reconfiguration at a time. We intend to extend this work to handle several reconfigurations simultaneously so that multiple hot objects can be handled efficiently. We also plan to look at different failure scenarios and deal them efficiently by using replication. In the current version, we have used static threshold values to perform split-merge

operations. We plan to implement machine learning techniques to have a model which will learn and predict the values for better performance. We intend to explore the various other possible applications which can benefit from the concept of social consistency which includes IoT devices, energy harvesting devices used in smart cities, smart grids and health monitoring.

Bibliography

[1] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 401–416. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043593

[2] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger Semantics for Low-latency Geo-replicated Storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2013, pp. 313–328. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482657

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[4] T. Stading, P. Maniatis, and M. Baker, "Peer-to-Peer Caching Schemes to Address Flash Crowds," *Peer-to-Peer Systems*, pp. 203–213, 2002. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45748-8_19

[5] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. Long, "Managing Flash Crowds on the Internet," in *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*. IEEE, 2003, pp. 246–249. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1240667

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

[7] W. LeFebvre, "CNN.com: Facing a World Crisis," in *Invited Talk on 2002 USENIX Annual*

Technical Conference, 2002. [Online]. Available: https://www.usenix.org/conference/lisa-2001/cnncom-facing-world-crisis

[8] J. Brutlag, "Speed Matters for Google Web Search," 2009. [Online]. Available: http://services.google.com/fh/files/blogs/google_delayexp.pdf

[9] "Facebook," https://www.facebook.com/.

[10] "Twitter," https://twitter.com/.

[11] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[12] "Goodreads," https://www.goodreads.com/.

[13] "Twitter," https://www.quora.com/.

[14] "Tripadvisor," https://www.tripadvisor.in/.

[15] "Ebay," https://www.ebay.in/.

[16] "Snapdeal," https://www.snapdeal.com/.

[17] E. Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6133253/

[18] J. Wang, "A Survey of Web Caching Schemes for the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, pp. 36–46, 1999. [Online]. Available: http://doi.acm.org/10.1145/505696.505701

[19] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM, 1995, pp. 172–182. [Online]. Available: http://doi.acm.org/10.1145/224056.224070

[20] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 335–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298487

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 265–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387906

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[23] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855742

[24] S. A. Weil, S. A. Brandt, E. L. Miller, and K. T. Pollack, "Intelligent Metadata Management for a Petabyte-scale File System," in *2nd Intelligent Storage Workshop*, 2004. [Online]. Available: https://www.ssrc.ucsc.edu/Papers/weil-isw04.pdf

[25] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic Metadata Management for Petabyte-scale File Systems," in *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2004, p. 4. [Online]. Available: https://doi.org/10.1109/SC.2004.22

[26] W. Lin, Q. Wei, and B. Veeravalli, "WPAR: A Weight-based Metadata Management Strategy for Petabyte-scale Object Storage Systems," in *International Workshop on Storage Network Architecture and Parallel I/Os*, 2007, pp. 99–106. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4438054/

[27] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3: Design and Implementation," in *In Proceedings of the Summer USENIX Technical Conference*, 1994, pp. 137–152. [Online]. Available: https://www.usenix.org/publications/library/proceedings/bos94/full_papers/pawlowski.ps

[28] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *Computer*, vol. 21, no. 2, pp. 23–36, 1988. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=16

[29] P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond," *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, 2013. [Online]. Available: http://doi.acm.org/10.1145/2460276.2462076

[30] 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. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-642-24550-3_29

[31] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The Little Engine(s) That Could: Scaling Online Social Networks," in *Proceedings of the ACM SIGCOMM 2010 Conference*. ACM, 2010, pp. 375–386. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851227

[32] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining Email Social Networks," in *Proceedings of the International Workshop on Mining Software Repositories*. ACM, 2006, pp. 137–143. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138016

[33] D. Greene, D. Doyle, and P. Cunningham, "Tracking the Evolution of Communities in Dynamic Social Networks," in *The International Conference on Advances in social networks analysis and mining*. IEEE, 2010, pp. 176–183. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5562773

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 385–400. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043592

[35] K. Li and S. Jamin, "A Measurement-Based Admission-Controlled Web Server," in *19th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, 2000, pp. 651–659. [Online]. Available: https://ieeexplore.ieee.org/document/832239/

[36] M. Randles, D. Lamb, and A. Taleb-Bendiab, "A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing," in *24th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 2010, pp. 551–556. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5480636

[37] M. Randles, A. Taleb-Bendiab, and D. Lamb, "Cross Layer Dynamics in Self-Organising Service Oriented Architectures," *Self-Organizing Systems*, pp. 293–298, 2008. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-92157-8_28

[38] M. Randles, A. Taleb-Bendiab, and D. Lamb, "Scalable Self-Governance using Service Communities as Ambients," in *World Conference on Services-I*. IEEE, 2009, pp. 813–820. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5190726

[39] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," *Peer-to-Peer Systems II*, pp. 68–79, 2003. [Online]. Available: https://pdfs.semanticscholar.org/0a79/a9692768c5b71f5eeceb1aed83936748960d.pdf

[40] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. "O'Reilly Media, Inc.", 2013.

[41] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide: Time to Relax*. "O'Reilly Media, Inc.", 2010.

[42] "Riak," http://basho.com/riak/.

[43] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=546611

[44] A. S. Tanenbaum, *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2008.

[45] G. Karypis and V. Kumar, "A FAST AND HIGH QUALITY MULTILEVEL SCHEME FOR PARTITIONING IRREGULAR GRAPHS," *Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://epubs.siam.org/doi/pdf/10.1137/S1064827595287997

[46] "Higgs Twitter Dataset." [Online]. Available: https://snap.stanford.edu/data/higgs-twitter.html

[47] K. J. Arrow, "A DIFFICULTY IN THE CONCEPT OF SOCIAL WELFARE," *Journal of political economy*, vol. 58, no. 4, pp. 328–346, 1950. [Online]. Available: http://www.jstor.org/stable/pdf/1828886.pdf

[48] "NSE ex-staffer reveals how some brokers got 'preferential access' to servers." [Online]. Available: https://economictimes.indiatimes.com/markets/stocks/news/nse-ex-staffer-reveals-how-some-brokers-got-preferential-access-to-servers/articleshow/58136897.cms

[49] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed Data Structures over a Shared Log," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 325–340. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522732

[50] D. L. Mills, "Network Time Protocol," *Network*, 1985. [Online]. Available: https://tools.ietf.org/html/rfc958