A Client-Oriented Solution for Optimistic Replication of Cloud Services

by

Wenbo Zhu, B.Eng, M.Sc.

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

December 20, 2013

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis

“A Client-Oriented Solution for Optimistic Replication of Cloud Services"

submitted by Wenbo Zhu, B.Eng, M. Sc

in partial fulfillment of the requirements for

the degree of Doctor of Philosophy

Department Chair

Thesis Supervisor

External Examiner

Abstract

This dissertation presents a novel approach to enable tunable tradeoffs between performance and consistency for geographically replicated cloud services. The end goal of the approach is to have a solution by which replication is able to improve the overall performance of the system while inconsistency due to optimistic message delivery is bounded in both time and space. We achieve the latter goal with an eagerly executed commit layer that detects and solves conflicts in parallel with the optimistic replication layer.

We believe such a solution not only solves the performance bottleneck of replicated services but also enables a full spectrum of tunability which will further allow applications to choose the best strategy to balance the tradeoff between performance and consistency for their replicated services. The new solution differs from the traditional eventual consistency model by providing a capability to solve conflicts in an online manner and to leverage the explicit role of clients in specifying the consistency requirements.

Experiments on a prototype system in a real cloud environment are described, that show that the *Client Oriented Layered Optimistic Replication* (or COLOR) is feasible, and which evaluate the achievable tradeoffs between performance and consistency on a realistic example system under load, distributed over five replicas in two continents. The prototype shows that COLOR provides a well defined programming model to assist application developers to control the replication of their cloud services without resorting to the otherwise non-guarantee eventual consistency model in face of performance challenges.

Acknowledgements

I would like to dedicate this thesis to my wife, Bing Zhang, for her support throughout the years.

I would especially like to thank my supervisor, Professor Murray Woodside, for his continuous support over the course of last 12 years, and most importantly his trust in me for getting this work done. His guidance has all been instrumental in the success of this work.

I would like to thank my parents and my three lovely boys for their love and help.

I would also like to thank all the people at Carleton University I have had the pleasure to work with, in particular Professor Shikharesh Majumdar, and Professor Dorina Petriu.

Contents

Glossary

C	a client external to replicas under the COLOR system model.
L	a local replica, i.e. a server, in the view of a client C which interacts with L.
S	a sequencer or master in the system.
R	a remote replica that L interacts with, in the view of clients of L.
H	a host node in the system, including S, L, R or C.
T_L	response time seen by L for delivering a message to the replication system.
X	a random variable for a one-way message propagation delay.
$f_X(x)$	density function of X.
F_X	distribution function of X: Prob{X≤ x}, and x is a value of X.
X_C	message propagation delay from C to L.
$OS(j,\{X_1,..,X_n\})$	the j^{th} order statistic of the set $\{X_1,...X_n\}$ of random variables.
P_H	processing (CPU) time at H for a given message.
P_S	processing time at S to sequence a new message and send it to *n* other replicas.
P_L	processing time at L to deliver a message locally.
P_R	processing time at R to acknowledge a message from S, including writing logs.
D_H	delay for all the operations at node H to support a message round.
D_S	delay at S to complete $P_{S.}$
D_L	delay at L, including serialization delays to complete $P_{L.}$
D_R	delay at R, including delays such as writing to a log to complete $P_{R.}$
D_{th}	denotes an intentional delay to achieve a threshold for blocking an operation such as sending or delivering a message.
N_R	total number of replicas in the system.
λ_C	system throughput in messages per second from all clients interacted with L.
λ	system throughput in messages per second for all the replicas: λ_C * $N_{R.}$
μ	failure rate (failures/s) for S, as a Poisson stream, and mean time to failure (MTTF) of 1/μ.
λs	server-message rate generated for all clients interacted with L.
Ack_i	round-trip delay to send a message from S to a remote replica *i*, process it at replica *i* and send the acknowledgment back to S.
$Q(q, N_R)$	$OS(q, \{Ack_1,...,Ack_{NR}\})$: delay to obtain a quorum of size *q* of replies from N_R replicas.

Chapter 1 Introduction and Motivation

This dissertation presents a novel approach to enable tunable tradeoffs between performance and consistency for geographically replicated cloud services. The end goal of the approach is to have a solution by which replication is able to improve the overall performance of the system while inconsistency due to optimistic message delivery is bounded in both time and space. We achieve the latter goal with an eagerly executed commit layer that detects and solves conflicts in parallel with the optimistic replication layer. We believe such a solution not only solves the performance bottleneck of replicated services but also enables a full spectrum of tunability which will further allow applications to choose the best strategy to balance the tradeoff between performance and consistency for their replicated services. The new solution differs from the traditional eventual consistency model with its capability to solve conflicts in an online manner and to leverage the explicit role of clients in specifying the consistency requirements. Experiments in real cloud environments are created to show that the *Client Oriented Layered Optimistic Replication* (or COLOR) provides a well defined programming model to assist application developers to control the replication of their cloud services without resorting to the otherwise non-guarantee eventual consistency model in face of performance challenges.

1.1. Introduction: why replication is hard

The cases for replicating data for Internet based services have been to achieve improved reliability and availability [22] [68]. The former addresses the resilience to failures such as machine or data center failures that may stop a service from returning otherwise success results to its clients. The latter allows a service to continue its operation in the face of network or data center failures, which may prevent clients from accessing services that are running in particular geographic locations.

Replication may also improve the performance of the service in certain cases when data is moved closer to clients that consume the data [109] or when load-distribution is to be enabled safely for instance over read-only data.

However, in most cases, when replicated data is shared by many clients and the state of the data may be modified concurrently by those clients, data replication will not only cost performance but also introduce reliability or availability problems. The key reason behind these costs is the complexity and overhead involved to maintain consistency across multiple geographically distributed replicas; i.e., copies of the same data [68]. In the typical Internet environment when data is replicated across data centers that span 100's to 1000's of miles or even across continents, data replication will significantly increase the response time of reads and writes to the state of the data. Although the underlying solutions may be different, an *n*-way replication effectively turns a single local read or write into a global "transaction" that involves *n* different data sources if one considers each replica as an independent "database" of its own. Even when *n* is relatively small, such as 3 or 5 way replication, the overall performance of the system will be dramatically degraded simply due to the latency (caused by the speed of light) which could be in 10's to 100's *ms*. Moreover, when replicas are geographically distributed, complicated algorithms have to be executed for every single operation in order to reach an agreement that is consistently held across all the replicas. Since replicas may fail on their own

at any random point of time or fail to communicate with each other directly, this so-called distributed agreement problem is very hard to solve and involves non-trivial computation and communication overhead. There is no known solution that is able to offer a timely replication solution that is safe in both normal condition and failure cases.

Given the above challenges, the so-called CAP theorem [95] has become extremely popular, which effectively advocates that consistent data replication is impossible to achieve. However, since data replication is a must for Internet services to tolerate reliability and availability problems, let's forget about safe replication and have applications deal with inconsistently replicated data. Such a solution inevitably turns data replication into a problem that application developers have to solve after the fact with little assistance from the system that manages the replication.

To summarize, the situation every Internet service faces today in managing data replication is an all-or-nothing scenario. There is virtually no solution that allows a service to retain both performance and safety of geographically replicated data.

1.2. Thesis summary

Problem Statement: to create a solution that enables tunable trade-offs between performance and safety of geographically replicated data, with the safety guarantee quantified by taking into account the client-to-cloud interaction semantics and the resulting consistency requirements.

In this dissertation, the overall solution is named as COLOR - Client Oriented Layered Optimistic Replication. COLOR adopts a set of well-known optimistic replication schemes, which allows changes to be applied to the local replica eagerly without waiting for the termination of strict replication protocols, such as a two-phase commit protocol or distributed agreement protocols such as Paxos [91] [19] [28] . Different optimistic schemes are devised that enable different tunable parameters in various forms such as time duration or message ordering.

Moreover, COLOR allows for eager message delivery in both steady-state and fail-over executions.

With the use of optimistic algorithms, COLOR is able to offer performance that is comparable to replication in a local area network or data center. Furthermore, as the optimistic algorithms become more aggressive in scheduling earlier delivery of messages on local replicas, performance of geographically replicated data may even offer a superior performance compared to centralized solutions in that data is distributed closer to clients, which means reduced latency to access the replicated cloud services as well as improved load distribution that often results in better utilization across shared data centers.

Given the safety tradeoffs, COLOR needs to detect and compensate for inconsistency that is caused by the optimistic algorithms. COLOR introduces a novel model that takes into account the explicit role of clients in specifying and resolving inconsistency on both client and server sides. The goal is to ensure inconsistency is detected as early as possible as data is being replicated; and inconsistency is solved in parallel with replication algorithms. To further bound the inconsistency, explicit commits are also made available which will limit the degree of inconsistency. The overall solution for resolving inconsistency adopts a set of eager algorithms, which we also refer to as online optimistic algorithms due to their parallel executions with the strict replication algorithm.

Based on the COLOR model, the thesis creates a set of analytic models that compute the inconsistency cost as probabilistic failures based on different runtime algorithm parameters. The analytic model is not meant to match the actual execution environment, which often involves a lot of runtime variables. The goal of the analytic model is to guide the implementation of COLOR in a typical cloud environment, and as well as to offer a range of tunable parameters as inputs to the runtime evaluation.

A complete implementation of COLOR is created and its effectiveness is evaluated in Google's internal cloud environment. We used a shared whiteboard application as a canonical

use case which covers all the key elements of the target applications of the COLOR solution. Moreover, the application offers an intuitive way to measure the performance and the consistency tradeoff costs. As part of the implementation, we also explored a client-side framework that allows applications to resolve conflicts in real-time. The experimental implementation aims to provide an abstraction that is sufficient for high-level application frameworks to be created, such as those employing “operational transformation” [98] for use cases that represent collaborative online applications.

Lastly, the thesis focuses on one important tunable parameter, i.e., data sharding. We believe this has the best potential to offer a truly continuous tuning space for the COLOR solution. The thesis uses the same whiteboard application to illustrate the role of data sharding, and its performance evaluation is conducted in the same environment as other tunable parameters.

To summarize, this dissertation presents the COLOR solution with the following components:

1. Identify a set of optimistic replication algorithms which cover both steady-state and failure cases.
2. Identify the performance gains of optimistic algorithms.
3. Define a new model under which inconsistency may be detected and resolved in an online manner.
4. Create a set of analytic models to establish parameters that enable tuning of the tradeoff between performance and consistency.
5. Implement the COLOR solution and an experimental application in a real-world environment, along with the necessary client-side framework to support conflict resolution.
6. Study the potential of data sharding as one of the key tuning parameters.

1.3. Background: data replication in cloud services

Replication of data across geographically located data centers has been a challenging task. There are generally three popular models that offer different safety and liveness guarantees, as well as different performance characteristics.

The safety guarantee ensures data is replicated correctly across all replicas at all times. This guarantee is also called reliability guarantee. The liveness guarantee ensures data may be accessed at all times when replication is being executed across the system. This guarantee is also called the availability guarantee [68].

In the first model, e.g. Google's BigTable replication [30], data is replicated as a background job, and the goal of the replication is to merge conflicting updates in some arbitrary order as decided by the replication runtime, and eventually to converge the state of all replicas. Under this model, changes to a local replica may be lost or reordered against changes made by other replicas to the same data entity. On the other hand, since replication is done completely as a background job, there is almost no performance penalty from replication. Furthermore, regardless of how many copies of data needs to be maintained, all reads and writes are executed locally without any replication delay. It is not uncommon to see data being replicated with 10-20 copies across the globe to make data as close to clients as possible and to minimize the access latency. However, due to the lack of any consistency guarantees, only a limited set of use cases may be adopting such a replication solution, notably applications that append log entries from each user. Common examples include blogs, user comments or bulletin board system (BBS) [73].

The second model is the so-called master-slave model [42]. Under this model, all reads/writes got to the same master replica, which may replicate data either synchronously or asynchronously to remote replicas. There is no limit on the number of replicas, and all read-write accesses have to go to the master replica. Read-only accesses may be served by replicas,

such as offline reporting in many transactional systems. When the master replica is down, there will be a down time during which the system becomes unavailable. When the replication to slave is synchronous, i.e., a write to the master will not be completed until all the replicas have been updated with the same write, any slave that fails to accept the write may block the write to the master too. It is a popular solution due to its simplicity as well as inherent scalability when read-only accesses are dominating. However, the master-slave model represents mostly a centralized solution with limited reliability or availability when the master is subject to machine or network failures. When updates are propagated to replicas asynchronously a master failure may cause committed updates to disappear. Likewise, when updates are propagated to replicas synchronously, unreachable replicas may block the master indefinitely even when only a quorum of replicas need be updated for the master to complete the update.

The third model employs a truly distributed model, in which there is no explicit master as far as clients are concerned [151]. Write or read accesses may be accepted by any replica, and consistency is guaranteed globally as if there were a single logical replica which has zero downtime in terms of availability. Replica failures, unless a majority of them failed at the same time, will be completely masked to the client. Such a model therefore represents a truly distributed model with the highest level of reliability and availability, especially when replicas are geographically located over the WAN network. On the flip side, this model has the worst performance due to the underlying algorithm complexity and overhead. Because of the performance issue, only critical applications are adopting such a model at the Internet scale, and for those applications the number of replicas is also limited to 3 or 5 in order to limit the algorithm overhead while still ensuring a sufficient level of availability and reliability.

To summarize, there are three models for data replication in Cloud services. The first model ensures no safety guarantee and replication is done as a best-effort. The second model mostly addresses online data backup and offers limited availability guarantee of what a truly

distributed system offers. The third model is the standard replication model most Cloud services have to choose that offers strict safety guarantee but with significant performance overhead.

1.4. The case for tunable consistency

The safety guarantee offered by each of the three replication models decides the consistency requirement of the underlying replication algorithm. Tunable consistency targets a model under which consistency requirement may be adjusted based on requirements from applications that share the same replication solution. In many ways, the so-called tunable consistency is offered as a hybrid solution that combines the three different replication models into a single system.

Conceivably, for a large system, not all the application data has to be replicated under the same replication model. If we use a collaborative document application as an example, such as Google Docs [59], the actual document has to be replicated consistently when edits may be made by multiple users at the same time. However, user comments made to the entire document may be replicated under the best-effort model. The master-slave model may be used for searching document instances. Indeed, many systems have been adopting such a hybrid solution in order to make the right tradeoff between consistency and system performance.

When the best-effort model is adopted, the system may also be able to put a boundary on the quantitative divergence between replicas. For example, the replication system may adjust the frequency under which updates need be merged and their conflicts need be resolved. Besides time frequency, other thresholds may also be used to manage when updates have to be merged across replicas, such as the numeric value of an update, gaps of update orders relative to each other [153].

Similarly for the master-slave model, it is possible to choose between synchronous and asynchronous delays [61], and for the former it is also possible for the master to block only on a subset of slave replicas, i.e., the so-called quorum update [68]. In this thesis, we are not

interested in the master-slave model, mostly because it does not address the environment when replicas are geographically distributed.

For the distributed replication model, it is also possible to relax the strict algorithm to improve system performance. When a weaker algorithm is applied, inconsistency is bound to happen [9] . It is up to the replication system to constrain the scope of inconsistency. The past two decades of research and practice have seen many solutions being proposed in this direction. However, for geographically distributed replicas, improvements are rather limited if strict consistency has to be satisfied in the view of applications. For instance, temporary inconsistency local to the internal execution of a replica will have rather limited effect on the overall performance. The work of this dissertation is to explore a new model which allows inconsistency to be exposed to clients of replicas, although not necessarily to their users, while the actual replication model is still a strict one that will guarantee availability and reliability at the system level. Under this model, performance of replication will become comparable to non-replicated cases. When data is replicated closer to the client, the actual performance may even be improved over non-replicated cases.

1.5. Thesis contributions

The contributions of this work are:

1. A novel replication model (COLOR) that is specifically designed to improve the scalability of strict data replication over geographically distributed replicas at the Internet scale. COLOR differs from existing solutions in its capability to execute the strict replication algorithm in parallel when temporary inconsistency is detected and resolved (Chapter 3).
2. Optimistic algorithms for COLOR which enable tunable tradeoffs between consistency and performance (also in Chapter 3).
3. Mathematical models for the optimistic algorithms to analyze (1) the performance gain compared to the strict replication model, (2) the resulting inconsistency cost under failure

scenarios and (3) the effects of the tunable parameters under different optimistic replication models (Chapter 4).

4. Implementation of a general-purpose software framework to support the use of COLOR model in a real cloud environment as a proof-of-concept (Chapter 5).
5. Using the framework, a detailed case study has been conducted to evaluate the effectiveness of the COLOR model with a realistic application (an online white-board). The case study also relates the analytic model to the case study measurements and their tradeoffs (Chapter 6).
6. A set of guidelines on how to handle a temporary inconsistency caused by the COLOR algorithms and resolve the conflicts it creates (Chapter 7).

1.6. Novelty of the COLOR solution

Under the COLOR system model, client roles are explicitly defined in how consistency is specified. The algorithms required to detect and resolve inconsistency are client-oriented, and inconsistency is quantified in the view of clients too. No existing solution has explored the client roles in applying weaker replication algorithms and formalizing client-level inconsistency at the same time.

The COLOR solution differs from the traditional Eventual Consistency [41] and its underlying convergence algorithms in that inconsistency is created only as a result of optimistic algorithms that still target strict replication. Furthermore, inconsistency is bounded and solved in real time, while the overall system is running strict replication across all replicas and their clients in parallel with optimistic algorithms. In this respect, we consider the COLOR an online model in how inconsistency is managed. The fact that there are two layers of replication algorithms makes it a layered replication solution, and hence the term “Client Oriented Layered Optimistic Replication”.

The COLOR solution has identified a set of tunable parameters, including a novel way to exploit partial ordering for improved parallelisms in replication of partitioned data sets. Both analytic models and experiments have been created to evaluate the effectiveness of those parameters in a typical cloud environment.

Lastly, the COLOR solution employs an extensible framework under which different strict replication algorithms, optimistic algorithms or client-side recovery algorithms may be applied on top of the core COLOR algorithm. The core algorithm and the interface specification between client and replicated service are to be open-sourced and proposed as IETF standards as well.

To summarize, the thesis creates a novel solution to the performance overhead of strict replication over Internet based cloud services, with the potential to make replication actually improve performance. The solution also brings clarity to the role of the ever popular Eventual Consistency model, which serves mostly as an excuse to ignore the consistency problem as it stands today.

Chapter 2 Background and Related Work

In this chapter, we will discuss the target system model of the thesis, followed by related replication solutions, as well as existing work and their limitations. Many system models have been proposed in the general area of data replication, and in this thesis we choose to adopt a system model that clearly abstracts the target cloud environments and their constraints. We will also be addressing a wide range of related subjects, such as the role of clients, the concept of sessions, and popular client-to-cloud messaging interfaces. We believe that a system model that is closer to the real world environment is crucial to a clear definition of the COLOR solution.

In related work, limitations of the existing work are discussed along with their application background. A wide range of work is discussed in order to cover different aspects of the replication problem which many still view as an unsettled research subject.

2.1. Introduction of the system model

Our target system model is defined to represent the cloud environment, which differs significantly from traditional system models that often involve operating system processes or individual machines.

1. Cluster: a single data center which hosts 100's - 1000's machines and maintains a local copy of application data. Clusters are connected to each other via private or public networks, and are geographically distributed to form a global cloud, such as what is offered by *google.com*, or *amazon.com*.
2. State: the application state that is accessible to clients and servers and is physically stored in clusters. The actual media that stores the state is irrelevant. The state may be replicated internal to a cluster, or may be made durable via multiple storage layers such as disk, flash or DRAM. The storage and durability support are also irrelevant in our model.

3. Replica: the local copy of any replicated state within a single cluster. State may be replicated in multiple clusters, and all the replicated copies are associated with a single logical entity of the replicated application state. Again, replication within a cluster is irrelevant, so is the physical storage of the replicated state within a cluster.
4. Client: the application entity that consumes the replicated state, e.g. on behalf of end-users. Clients represent the application logic that reads or writes the state by interacting with the cluster that hosts the state or its replica. The physical location of a client is also irrelevant in our model, as described in a later section.
5. Server: the runtime entity within a cluster that handles interaction between a client and the cluster. Contrary to traditional system models, the concept of processes does not exist anymore. The physical entity that represents a server may not map to any individual process, and processes do not own the state either. In other words, the server exists only in the abstract form as an interface for the client to access the replica state managed by a cluster.

Figure 2.1 shows how the above core concepts of the system model are related to each other in a typical cloud environment. Specifically individual clients interact with their local cluster which holds the server-side state accessed by the server application. Server state of this local replica is replicated to other remote clusters, each of which will have its own clients too.

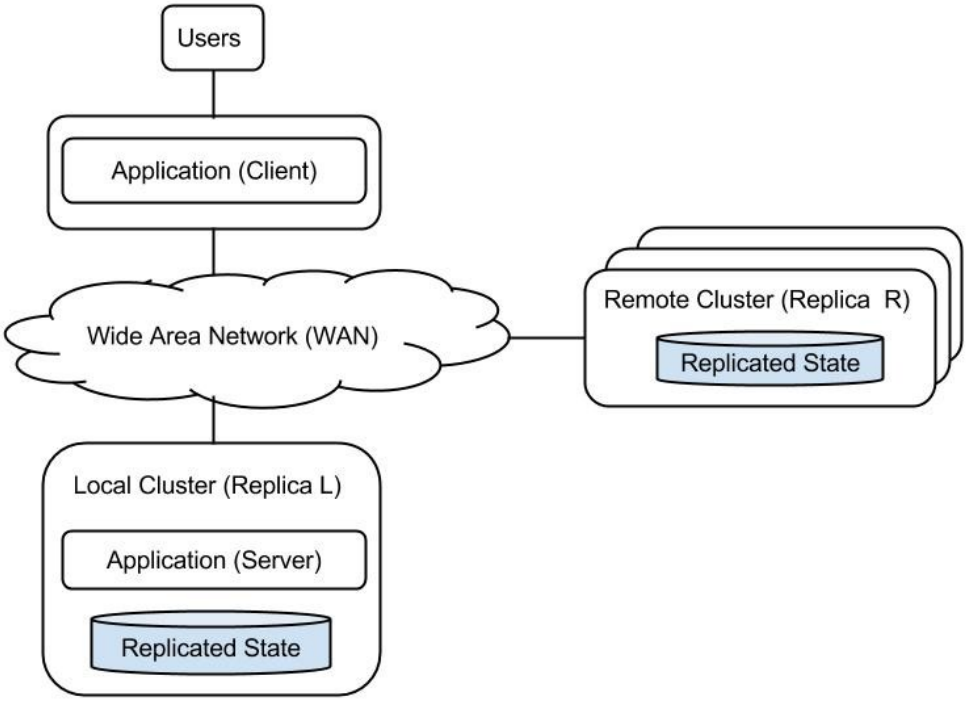


Figure 2.1: Core concepts of the system model.

Under the above model, the concept of clients becomes especially important as clients represent the actual application logic that interacts with each replica and its copy of application state.

To summarize, the key to the role of clients is where actual changes to the state are to be made. Following are two examples for which the role of clients is actually taken by server software or a database layer.

1. A web client that accepts user inputs and passes them to a remote server for actual execution will not make the actual client, whose role will be played by the server-side logic.
2. A server that executes client requests but has to commit changes via a transactional RDMBS (Relational Database Management System) [89] will not be considered as the actual client either. Instead, the centralized RDBMS software is responsible for making

changes to the underlying state, and therefore takes up the role of clients under our system model.

Our definition of the client role is closely related to the State Machine model [128], which relies on the following two requirements for the correctness of state replication.

1. Replica state is decided by the sequence of changes made to the local replica.
2. As long as the sequence of all the changes made to each replica is identical, state of each replica will remain identical to each other.

Under the State Machine model, any non-deterministic logic [17] [72] [119] will have to be treated as that of clients. One typical example is the behavior of a cluster-level RDBMS in scheduling transactions in its local memory. In such a system, the output of the RDBMS, e.g., committed transactions, are considered updates to the replicated state and the RDBMS actually plays the role of clients under this model. Generally, most server-side logic will involve local execution that is non-deterministic and, as a result, the actual client role will always be extended to the server-side logic.

To avoid the pitfalls of identifying clients, we have introduced the following concepts to the system model that specifies the interface between real clients and the server-side state machine.

1. Client messages are what trigger updates to the replicated state. Changes of replicated state are always initiated from client messages, which include autonomous changes which are actions of virtual clients such as server-side robots. Under this model, we do not explicitly model read-only messages, such as queries, as client messages. The messaging pattern, such as RPC [22] or asynchronous messaging, is also irrelevant in our model.
2. Server messages are generated as the result of client messages, or as spontaneous server-to-client notification, or as the result of client-solicited responses, such as

response to a query request. Similar to client messages, server messages also do not concern different messaging patterns.

Although clients represent one logical concept in our model, real cloud applications or systems are often layered; that is, before replicated state is changed, there will be many layers of client-server interactions. Generally, it is of limited value to model only immediate clients of replicated state as represented by software components within a cluster, as state of such components may be trivially recovered with no external effect.

In this model, we consider client messages are always initiated, directly or indirectly, from clients external to the cluster. Given the clients are external to a replica, any inconsistency caused as part of replication will be made externally visible. To simplify the overall model, we consider external clients and cluster-side software components an extended client layer. The extended client layer includes the following two key properties:

1. Changes to the replica state are always initiated from the client layer.
2. Clients directly interface with the replica state, in the form of client messages or server messages.

For typical cloud applications, an extended client layer will include both clients external to the cluster and the server-side software that handles messages from and to the clients. The key to the client role lies in the state of client.

The first case of client state is when external clients hold a cached view of the replicated state. This is the common case for most cloud applications. Since the server-side software is stateless, we can simply ignore the server-side logic insofar as this model is concerned.

The second case involves when the state is mostly on the server-side, e.g., in the form of so-called server-side session state [130]. Under this model, the external clients simply provide a user interface and may also be ignored by the model. Typical examples include computation-heavy jobs submitted by an external client.

The last case involves when there exists both client-side cache and server-side session state. Generally, we will consider the client-side cache state representing the actual client state, when the session state is considered part of the replica state if the session state also needs be replicated (although it is only accessed by a single client). When the session state is not replicated, it is generally recoverable from either the client state or the replica state. For these cases, we still consider the external client playing the role of clients. Server-side session logic and state are important as they allow us to move replication-specific logic from the client-side to the server-side.

With the client role identified in the model, the interaction between the client and the replica may have to go through some middle-layer software that sits between the actual client and the replica; and the middle layer is actually generating all the client messages that update the replica state. In this system model, we assume the interface between the client and the replica will match the underlying state model, commonly following the REST (Representational State Transfer) model [51]. This restriction implies that client-side cache state will match the replica state under the same logical state model, i.e., how entities are named. More specifically, we require client messages to be able to be matched to the client-to-replica interface when middle-layer software exists between external clients and replicas.

To further illustrate the importance of identifying client states, Figure 2.2 shows a full spectrum of client roles, ranging from pure servers, for which the server will effectively play the role of clients at the same time in generating spontaneous state changes, to offline clients, for which the client maintains a local copy of the server state and essentially plays a peer role to the server.

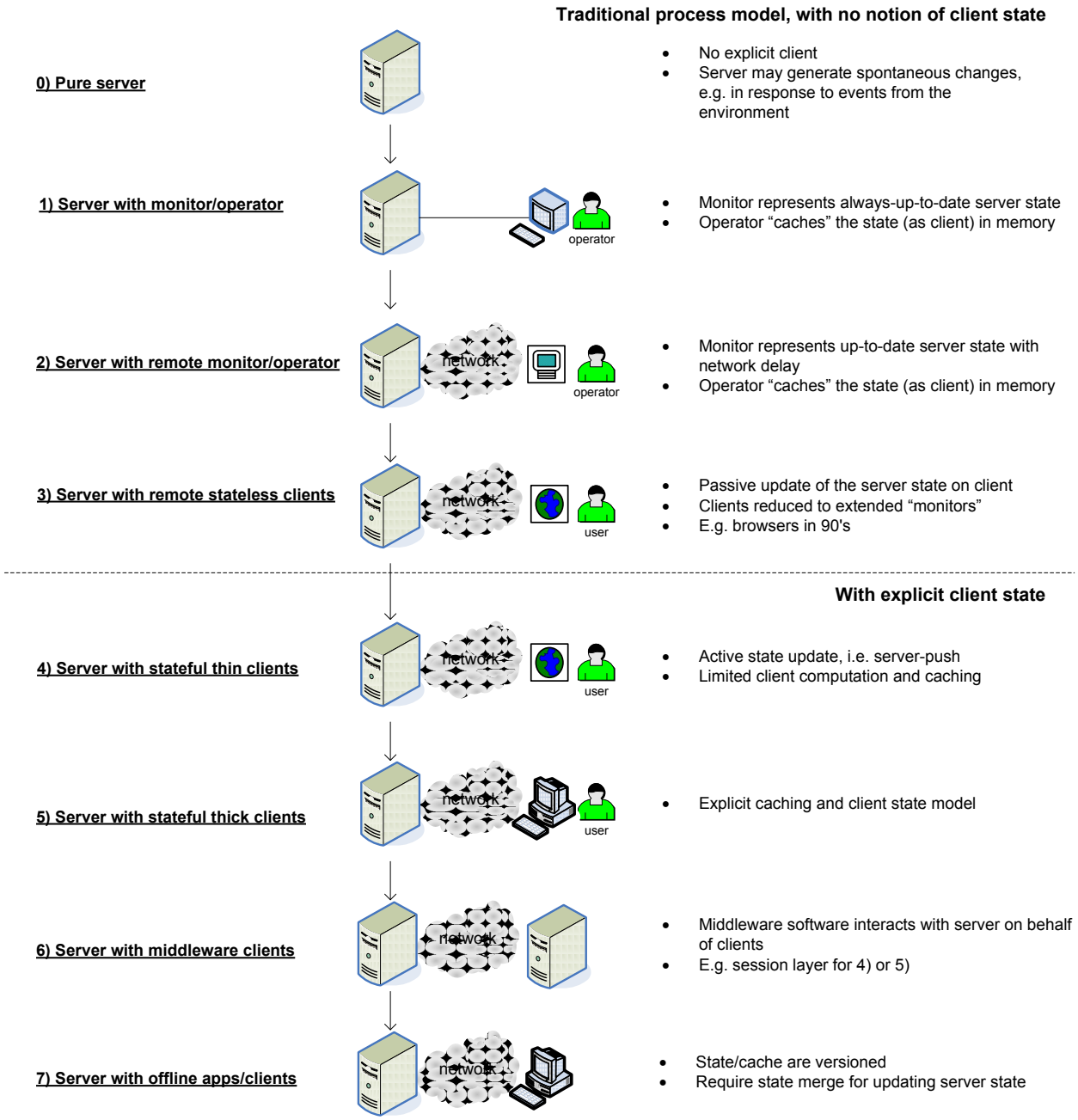


Figure 2.2: Evolution of the role of the client and its state.

2.2. Replica consistency

To meet the requirements of the State Machine model, under the system model defined in Section 2.1, client messages have to be delivered in the same order across all replicas. The

ordering requirement is a well understood consistency requirement for multicast algorithms [25] [43] [75] [80] [81] [93] [101] [117] [152] [23] [120] [121] [122] [123].

The two different total-ordering requirements that are concerned by our system model include: 1) non-uniform total ordering; and 2) uniform total ordering. The difference between the two is that uniform total ordering requires messages to be delivered in the same order on not only correct replicas but also failed replicas. The latter requirement is especially important under the COLOR system model as clients interact with only a single replica; and if a failed replica may deliver messages that are not visible to other replicas and therefore not necessarily applied to the global state, then clients will be exposed with an inconsistent view of the replica state. The cost of such externally exposed inconsistency contributes directly to the overall cost of multicast algorithms.

To understand the actual cost, we have to differentiate between two different system models which are decided by how conditions of networking and communication channels between replicas are modeled.

The first model is the so-called virtual synchrony (VSYNC) model [135] [85], which assumes there is a way to reliably tell if a replica is failed in a timely way. Under such an assumption, failed nodes will be immediately excluded from the multicast while at the same time, when such a decision is reached across all correct replicas, execution of the multicast algorithm is effectively suspended. Under this model, there is no overhead to ensure uniform total-ordering under normal conditions when there is no failed replica. However, its so-called synchrony assumption is generally considered unachievable in a WAN environment because the system cannot distinguish between slow replicas or failed replicas due to unbounded latency of communication channels [29] [126].

On the contrary, under the assumption of asynchrony of networking and communication channels [92], the so-called consensus model [63] [110] ensures uniform total ordering across all replicas by only requiring a quorum of replicas to acknowledge the delivery of each message.

The algorithms are required to tolerate arbitrary communication delays or network partitioning. However, such algorithms will introduce extra overhead for normal executions even when there is no replica failure or communication breakage. Among all the consensus algorithms, Paxos represents the most popular algorithm, with many different adaptations [90] [28] to meet practical requirements or for improved performance.

In this thesis, we assume the asynchrony networking condition [37] [56] [57], and the use of algorithms such as Paxos for uniform total ordering and delivery.

For many cloud environments, data is often replicated in 3-way or 5-way in order to achieve the desired fault-tolerance while keeping the number of replicas low to limit the overhead of multicast.

2.3. Client consistency

First, the problem of client consistency should be modeled under the more general problem of consistency for distributed shared memory [1] [8] [2] [46]. With this assumption and under the State Machine model, client consistency is guaranteed for so long as replica state is consistently replicated. When a client has interacted with a replica with a version of state that is inconsistently replicated, the inconsistency is considered to have been leaked to outside. When inconsistency is leaked to a client that is not part of the replication system, such inconsistency is not recoverable by algorithms.

One could imagine a client being a human that sees an incorrect state from a replica and initiates an action upon such an observation. The action, once initiated, has no trace of the observation of the incorrect replica state and therefore can never be reasoned or corrected. Given such risks, the State Machine model would have to mandate uniform total-ordering for all replicas to avoid introducing any client-visible inconsistency from interacting with any failed replica.

The above requirement is not necessarily true in a synchrony environment [114]. For instance, the so-called open-groups [77] [159] allow external clients to interact with replicas that only run non-uniform total ordering. However, whenever a replica fails, the system will invoke virtual synchrony algorithms which will halt the system and then try to recover any lost or unordered messages based on the message history that is stored on the client that has interacted with the failed replica. Such a recovery algorithm is impossible in any asynchrony environment.

2.4. Problems of strict replication protocols

The main problems with strict replication protocols are performance and availability. First, the performance overhead is directly attributed to the uniform total ordering, such as the Paxos algorithm. The availability issue is an inherent limitation of asynchronous network communication for any geographically distributed systems.

For performance, using Paxos as an example, under the normal conditions, client messages will have to be delivered to the master replica first. Then, a quorum of acknowledgements has to be received from each replica. For clients that are not directly interacting with the master replica, the maximum delivery latency will be t (mean RTT network latency) + t' (latency for reaching the quorum $q - 1$ out of $n - 1$ stochastic variables of t). Typical cross-data center latency is in 50-200 ms, and for a typical configuration, most replicas will be on the same continent, which means non-master reads or writes will suffer from up to several 100ms of latency depending on the number of replicas and their distances. Based on typical fault-tolerance requirements, most replicas will be located on the same continent (such as North America) with one or two replicas located in other continents (such as Europe or Asia). For traffic originated from those regions, the latency will be even higher due to cross-continent communication delay between the local replica and the master.

For strict replication protocols, reads also have to be totally ordered to support consistent reads between clients and their replicas. Therefore data replication is a pure loss in performance as data locality [141] cannot be leveraged for improved read performance either. Multiple lease based algorithms or solutions [10] [27] [35] have been proposed to support local reads, with the assumption that most read-only messages will be directed to a single replica, such as the current master. Even when such an assumption is true, retaining leases will cause a significant delay when a master fails and a new replica has to be elected to be the new master as both the old master and new master have to agree on the master change and the lease hand-off, while in a non-lease based solution a master change only involves a quorum of replicas to agree on the new master. This is especially a problem when the old master is cut off from the network.

A lesser understood performance problem is the response time cost during a failover. For example, when a Paxos master fails and a new master election has to be finished before the normal execution of the Paxos algorithm resumes. Essentially a master election involves two decision phases: 1) a decision to start a new master election, as made by a non-master replica; and 2) a quorum has to be reached by replicas to finish the master election process. The first phase involves failure detection [144], which is known to be subject to network asynchrony and therefore instability; for example, when an aggressively small timeout is chosen. A larger timeout will inevitably cause a longer pause of the replication algorithm which is also undesirable. All these factors contribute to a prolonged, non-deterministic failover process that has serious impacts on the overall system performance in the event of failures of either replicas or networking.

The availability problem becomes an issue for strict replication protocols whenever network partitioning happens. Because the uniform ordering and delivery require a majority of replicas to be responsive, any network partitioning will halt the system and make the data unavailable to the clients.

The solution to the availability problems may include reconfiguration [28] which will reduce the number of replicas to ensure the rest of the system will still remain alive when a subset of replicas are cut off from the network. However, reconfiguration processes are not usually considered an online process, which means that it cannot be executed on the fly as failures occur. Secondary, and as the number of replicas is reduced, the overall fault-tolerance of the system is also reduced.

2.5. CAP theorem and the case for Eventual Consistency

Due to the inherent issues of strict replication protocols, it is now well-known that one has to give up consistency in order to have a scalable and available solution when data has to be distributed and replicated geographically.

The CAP theorem states that it is impossible to have all the following three properties at the same time: consistency, availability and partitioning. Since the latter cannot be avoided in any Internet scale environment, the consistency requirement has to be sacrificed. The direct proof of the CAP theorem is the popularity of the so-called Eventual Consistency.

The key concept of Eventual Consistency is that read and write messages are directly served by local replicas and there is no replication algorithm running in real time to try to prevent or reconcile conflicting messages. Instead, the replica state is merged as a background process by either re-applying the logs or merging the actual state [21] [58]. There is no direct temporal relation between the local replica access and the global process of conflict reconciliation.

Under such a weak consistency model, performance and scalability bottlenecks of strict replication protocols are completely resolved. Moreover, because messages are processed locally the overall system capacity is increased as more and more replicas are added to the system. This is contrary to the strict replication protocols, for which the more replicas the system has the slower the overall system will be operating.

For the background process of conflict resolution, different solutions exist that adopt either pull or push based propagation algorithms [153] [136] [70]. It is also possible to bound the time interval or their quantitative or ordering differences with such algorithms. Among all of the existing work, [153] represents the most notable work on modeling tunable Eventual Consistency, although to our knowledge no practical solutions have been created in the real-world under such a model.

Existing work does not model any client-side impacts due to inconsistency of replication of the server-side state. When client-side cache is used, especially for offline clients, the same model may also be applied to the client-side cache coherence requirements. Overall, the Eventual Consistency model considers conflicts mostly harmless, which is true for typical applications that only append change logs to the replica state, such as Bulletin Boards [73]. However, for real applications to adopt the Eventual Consistency model, the application has to be designed in such a way that conflicts can be tolerated. Short of this, the inconsistency will be exposed to end-users directly in the form of application errors.

2.6. Approaches to optimistic replication

Unlike Eventual Consistency which gives up the consistency requirement entirely, optimistic replication tries different means to explore the possibility of earlier message delivery while retaining effectively the same level of consistency as strict replication protocols.

Most existing work tries to make early delivery possible by assuming conflicts are rare and most of the time it is safe to deliver messages before the global consensus is reached. When the assumption fails, these optimistic algorithms will allow temporary server-side inconsistency to happen, as long as external clients are not exposed to such inconsistency.

Several existing works [83] [60] [146] are restricted to LAN and local cluster, in order to improve parallelism by allowing overlapped execution of message processing and global delivery. We are not particularly interested in this level of optimistic execution as the

predominating cost of distributed replication in a cloud environment is always the actual replication algorithms that are responsible for delivering messages globally.

One notable work is [50], which enables non-uniformed delivery on local replicas but instead requiring the client to receive a quorum of replies from replicas before any response is accepted by the client. While such a solution represents a standard optimistic algorithm, the fact that clients have to participate in the replication protocols make such a solution a so-called "closed group" solution. In essence, when clients have to be involved directly in a replication protocol, the role of clients is effectively changed to a replica. Such a change brings up many practical limitations. For example, clients for typical cloud applications are often standard browsers, which will not be able to run any complicated replication protocols due to either communication or security constraints. Furthermore, although rich web-based clients may support complicated application logic or possess a local cache of state, their interaction with the Web (and therefore remote replicas) is limited to what the HTML5 API [148] or HTTP protocol supports for the Web [52] [53].

For this thesis, our focus is the so-called "Open Group" [32], in which clients generally do not know anything about the replication protocols. Clients simply interact with one replica at any point of time although the physical replica may change over time due to load balancing or locality reasons.

In addition to the above works, a wide range of solutions have been proposed in the past two decades to address different network or application environments. Among them, [118] [129] explored different optimistic techniques for database replication; [131] [134] [112] proposed different optimistic total order multicast algorithms; [84] proposed an optimistic atomic broadcast algorithm for database replication, [125] [85] [44] [54] discussed optimistic consensus algorithms; [67] [65] [24] proposed different probabilistic multicast algorithms; [86] [87] [88] proposed several optimistic solutions leveraging conditions that enable read-only accesses to the replicated state; and [79] [82] [103] [116] [115] [139] [49] [6] discussed different relaxed Group

Communication solutions or specifications, such as those under network partitioning, and their applications in data replication. All these existing solutions are essentially offered as optimization techniques to the otherwise strict consistency problem and therefore are still subject to the general performance and availability problems of strict consistency in asynchronous networks.

Lastly, the only solution to our knowledge that allows earlier delivery but at the same time supports clients as Open Groups is Google Wave [98]. The Google Wave document protocol is able to detect inconsistency due to optimistic message delivery on a local replica. Google Wave clients support real-time online collaborative editing with the use of operational transformation based conflict resolution algorithms. With both techniques, Google Wave is able to support an ideal optimistic solution that this thesis also targets, in which client-side consistency is explicitly modeled to ensure the end-to-end consistency requirements under the general spirit of the End-to-End argument [124]. With clients being exposed to temporary inconsistency as the result of optimistic message delivery, the system is able to break through the performance bottlenecks of truly geographically replicated cloud services without having to resort to the Eventual Consistency model.

Although the Wave solution shares the same goal as the COLOR model, we note that the Wave solution is not actually running any optimistic replication algorithm in parallel on the server side. The inconsistency detection algorithm in the Wave protocol was designed to allow asynchronous storage operations on the server side, mostly because conflict resolution happens to be free for Wave clients.

2.7. Summary

The current state of art remains as an all-or-nothing approach. On the one end, strict replication protocols have been widely deployed with their performance and availability limitations well understood. On the other end, applications have to adopt the Eventual

Consistency model to meet the performance and availability requirements. While optimistic solutions may play a role to improve performance with temporary consistency loss, existing solutions are very much limited in their performance improvements due to the strict consistency required for external clients.

As demonstrated by newly emerging computing models such as Google Wave, we believe that if any viable solution is to have tunable consistency and performance it must include clients explicitly in the consistency model with replication algorithms that are devised to support both earlier delivery and recovery from inconsistent local replica state.

Chapter 3 COLOR - Client-Oriented Layered Optimistic Replication

This chapter describes the COLOR solution that is proposed to solve the performance bottleneck of distributed replication systems in the cloud environment. The solution involves applying optimistic replication algorithms with online detection and recovery on the client side. The solution takes into account the explicit role of clients and their interface with the replicas. The solution works with existing replication algorithms by separating the message delivery layer from the commit layer, and hence it is called multi-layered optimistic replication.

COLOR is presented as a novel approach, compared to the all-or-nothing Eventual Consistency or CAP model. Furthermore, although COLOR allows for temporary client-side inconsistency, it guarantees real-time resolution of conflicts between clients and their replicas, and the end-to-end consistency is maintained across the entire system.

Unlike the popular Eventual Consistency model, we categorize the COLOR model as a type of eager replication algorithm, for which inconsistency is to be bounded in both time and space by the strict consistency algorithm that is running in parallel with the optimistic algorithm.

3.1. State representation and system model

The state representation and system model of COLOR will be specified using the concepts of Object-oriented function and type notation. For a function which does a simple mapping from a key to an instance of some type a shortcut notation will be used with subscripts to map the key to a value given the scope of the container instance.

3.1.1. Overview

Figure 3.1 shows the core concepts of layered state presentation under the COLOR system model, which include the following elements:

1. Global and replica states
2. Messages and message histories
3. Client-side state

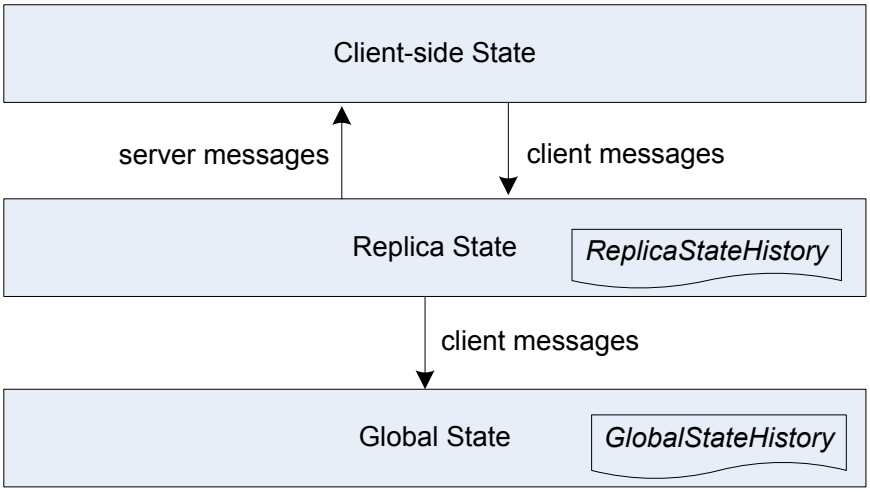


Figure 3.1: Layered state presentation in COLOR.

To further illustrate the above concepts, imagine a shared online whiteboard which allows concurrent users to draw on the same whiteboard and see each other's drawing in real time, then:

1. Global state represents the most recent whiteboard state as agreed by all correct replicas on the server side; and replica state represents the whiteboard state held by each individual replica.
2. A client, such as a browser, may draw on the whiteboard and each drawing operation will generate a list of client messages that include changes to the pixel color of the whiteboard; and in response to each client message, each local replica will generate a list of server messages for all the clients that connect to this replica in order to notify all the clients of the new state of the whiteboard.
3. On the client, the client-side state represents the cache of the server-side replica state of the whiteboard. Clients are able to update the local whiteboard state by drawing; and

will be notified by changes made by other clients, which will refresh the local whiteboard state accordingly.

3.1.2. Global and replica states

The state is a collection of named entities with a global naming scheme, such as provided by a URI. Each of these entities is called an “instance of state” and is represented as an instance of the *State* type. A function *resolveState*() maps an opaque *key* value to an instance of *StateInstance*.

- GlobalState#map(Object key) : StateInstance
- E.g., StateInstance state = globalState[key]

Here the global state *GlobalState* is represented by the namespace that constitutes all distinct *key* instances.

The above representation of state concerns only the logical global state, which may or may not map to any physical representation of state. Physically, each instance of state is represented as part of *ReplicaState.*

- ReplicaState#map(Object key) : ReplicatedStateInstance
- E.g., ReplicatedStateInstance localState = replicaState[key]

Replicated state is also named with the same naming scheme, e.g. using a URI, and the key for a *StateInstance* is identical to the key for a corresponding *ReplicatedStateInstance* as they both address the same logical *State* instance.

- GlobalState#map(Object replicaId): ReplicaState
- E.g., ReplicaState replica = GlobalState[replicaId]
- E.g., ReplicatedStateInstance localState = replica[key]

3.1.3. Messages and message history

There are two types of messages in our model: client messages and server messages. Client messages trigger changes to the local replica state, while server messages reflect the state of the replica and thereby establish the client-side view of the server state. A client message includes the following properties:

- Replica id.
- Key of the target *ReplicaState* instance: Message#stateKey()
- A globally unique message id, which often constitutes a unique client id and a message sequence id: Message#id()

Correspondingly a server message includes the following properties:

- Replica id.
- Key of the source *ReplicaState* instance: Message#stateKey()

The above properties are specified independently of the actual communication protocol between the client and server. For instance, HTTP headers [52] may be used to propagate these properties between the client and server.

For each instance of *State*, there will be a global message history that corresponds to the current value of *StateInstance* according to the State Machine model. Likewise, there will be a replica message history for the *ReplicatedStateInstance*. As for the representation of the global *State*, global message histories may only exist as a logical concept.

- GlobalStateHistory#map(Object key) : MessageHistory
- ReplicaStateHistory#map(Object key) : MessageHistory

Conceptually, *MessageHistory* comprises a vector of message ids whose messages have been delivered against the target global or replicated state. When messages are propagated between the client and server, a serialized format will be used, which will contain the following two fields:

1. The last message id
2. The hashed value of the vector of message ids that constitute the entire message history.

The serialized value of message histories is required to meet the following requirement as mandated by the State Machine model:

- For any two instances of *StateInstance* or *ReplicatedStateInstance*, the physical values of their underlying state are identical to each other if and only if the serialized values of their message histories are identical.

While there may be other details that implementations may choose to include as part of the message histories, the *MessageHistory* type defined here represents the canonical format to impose and satisfy the above requirement.

Under the State Machine model, we often need to check if an instance of *MessageHistory* is a prefix of another instance of *MessageHistory*:

- MessageHistory#prefix(MessageHistory otherHistory) : Boolean

An instance of message history is considered a prefix of another instance of message history if and only if the following conditions hold:

- Both histories are under the same key of *StateInstance* or *ReplicatedStateInstance*.
- Both histories start from the same message id.
- *This* history matches exactly what is after the first message in the *otherHistory* up to the last message in *this* history.

Based on the above conditions, one can conclude that a *ReplicatedStateInstance* is an older version of a *StateInstance* if the former has a history that is a prefix of the global message history of the given *StateInstance*.

3.1.4. Client-side state

For a given *StateInstance,* the client-side view of its *ReplicatedStateInstance* is created out of the interaction between a client and its local replica. Conceptually every server-message that is received by the client will cause changes to the client-side view of the server state.

As discussed earlier, clients may differ in their capabilities to be involved in replication algorithms. Most common clients are rather passive user agents, such as browsers, as decided by their interactions with the server. For now, we will assume the following properties for clients:

- For a given instance of state, a client will only need keep the most recent message history as the current version of its state.
- The message history is only known to the client runtime environment in the serialized format.

3.2. Client access interface

This section will discuss the interactions between the client and server, based on the operations and types specified in the previous section. The goal of the client access interface is to minimize the intelligence required on the client-side to meet the most basic consistency requirements and at the same to minimize the overhead of communication protocols to support such algorithms.

3.2.1. Client messages

Client messages include messages that are sent by the client to the server to update the replica state. Although we are not concerned by read-only client requests that will not affect the replica state, the client is not always aware if the request may or may not affect server-side state. Therefore, it is up to the server to decide if a message from the client should be considered as an update message and hence be included in the message history.

Client messages, as generated by the clients, may or may not explicitly include all the properties that are part of the message history. However, the server must be able to derive from them the following message properties:

1. Replica Id
2. State key

3. Message id

This requirement isolates the clients from involvement in the replication algorithm. As we note, one possible kind of client is a browser, and any intelligence or algorithm state required for such clients to function will create deployment or security issues.

In addition to the above information, each client message will also include context data that is specific to the implementation of the COLOR algorithm, which will be discussed in Section 5.5.1.

3.2.2. Server messages

Every server message is considered to be a state update to the client. As a result, server messages have to identify the following properties:

- Replica id
- State key

These are easily derived from the underlying messaging protocol [34]. For example, in the case of request-response communication such as standard HTTP GETs, the request URI (Unified Resource Identifier) will indicate the state key and an HTTP cookie [52] or an HTTP response header may be used to carry the replica id information.

Unlike the traditional RPC model, there is no correlation between client and server messages. As a result, server messages need not be identified with message ids.

In addition to the above information, each server message will also include context data that is specific to the implementation of the COLOR algorithm, which will be discussed in Section 5.5.1.

3.2.3. State validation

The COLOR consistency model requires the server to be able to detect clients with an invalid view of replica state whenever the server receives a client message. To achieve this

consistency requirement, the client-server interface requires the following extra context to be propagated via each client or server message. For any given state key that is involved in the interaction:

1. Every server message should carry the current replica message history in its serialized format.
2. Every client message should carry the most recent replica message history (or histories) that are known to the client.

The above requirements may be met by passive external clients such as browsers without introducing any significant changes to the underlying messaging protocol. For example, an HTTP Set-Cookie header may be used as the context of server messages; likewise, the HTTP Cookie header may be used as the context of client messages. Most Internet based messaging protocols are equipped with a transparent mechanism to propagate context data between the client and server. The existence of such context data is also transparent to the client-side application, as desired.

Considering the above context data associated with each client or server message, the server handles a client message as follows:

1. The client message is checked against the current replica message history that the server maintains. A client message is considered valid if and only if the client message history is a prefix of the current replica message history.
2. When an invalid client message is received by the server; that is, the client message history is not a prefix of the current replica message history, the server is required to inform the client to recover from its current invalid state.

A typical action to force a client to recover from invalid state will be to issue a restart message that is to be implemented in a way that is specific to the underlying messaging protocol. A common method for browser clients will be for the server to issue an HTTP redirect that will effectively reload the current page and therefore any client-side state.

3.2.4. Client replica binding

A client is typically bound to the same replica for as long as the load distribution system allows. A client starts to interact with a different replica often when one of the following conditions happens:

1. A load balancing decision, to have the client interact with a replica that has less load;
2. A locality decision, to have the client interact with a closer replica;
3. A failover decision, when the current replica fails or becomes unavailable.

The above events usually happen at a rate that is much lower than the message rate when the client is actively interacting with the server.

After a client is bound to a different replica, at the first interaction with the new replica the current client message history will be validated against the replica message history. Following are different conditions that the server may expect:

1. The client message history is a prefix of the replica message history.
2. The replica message history is a prefix of the client message history.
3. The client message history doesn't match the replica message history as in either of the above two cases.

The first condition indicates the client remains in a valid state after a rebinding event. The client should continue interacting with the new replica and the client will start to keep the message histories from the new replica as the client receives new server messages.

The second condition indicates that the client has been interacting with a replica that processes a newer version of message history than the current (new) replica. The client's version may or may not be valid in the view of the new replica and there is no way for the new replica to validate such state at this point. Moreover, the new replica is not in a state to resume interaction with such a client. The general solution to such a condition will be blocking the interaction until the new replica is able to validate the message history, perhaps ending with a

timeout to force a restart of the client. The timeout is important to avoid blocking the client access for too long and hence cause any user perceived delay. Another option is to direct the client to a different replica which may have a newer version of message history; e.g., the master replica often has the more advanced message history compared to non-master replicas.

The third condition simply indicates that an invalid client state has been detected, and therefore a restart should be issued.

A false negative (overly pessimistic) is possible during a rebinding event. For example, the new replica may have a local message history that will be rolled back later due to conflicts between its local commits and global commits. A more detailed analysis is given in Section 3.3 on the commit interface against the global state.

3.2.5. Server-side sessions

For typical Internet clients, a server-side session layer usually exists to maintain the state across the client-server interaction and to move certain client-layer logic from the client to the server. The existence of the session layer may help in many ways to further limit the intelligence and state required on the client side.

In theory it is possible to keep the entire message history context as the server-side session state. In doing so, the client will only be required to possess the session id as the context. However, the server-side session state would then also have to be replicated, which will introduce a separate state key for each client. We consider this mostly an engineering decision and the overall client-server interaction model remains the same whether or not the message history context is to be stored as the server-side session state. When session state is created as part of the application requirement, it is certainly more efficient to not have to propagate message history with each server message.

A second role of the session layer might play arises when the underlying messaging protocol is a stateful messaging protocol as opposed to the stateless RPC protocol. More

specifically when the underlying messaging protocol allows for full-duplex messaging between the client and server, for efficiency purposes we should not propagate the replica message history over each individual server message of the same session. Instead, the session layer only needs to propagate the message history context whenever a new message history version has been committed on the local replica.

Likewise, for the purpose of efficiency, the session layer may choose to batch client messages that are to be committed globally. For example, an online collaborative application such as Google Docs may batch keystroke messages over either a time period or over a buffer limit.

3.2.6. The REST principle

The REST principle [51] [53] is widely adopted by Internet based cloud applications. The COLOR model does not assume REST; however, it benefits from the use of REST in the following ways:

1. The state key may be represented by URI directly, which largely simplifies the overall solution in mapping messages to the state key.
2. The read-only vs. update semantics of client-initiated messages are well defined; i.e., with HTTP methods. This makes it easier to filter out read-only client messages.
3. A REST based client-server interface often implies that the client-side state shares the same data model as the server-side because each client-initiated message may involve only one of the four basic operations: GET, CREATE, DELETE and UPDATE. Such a constraint greatly simplifies the mapping between client messages and the global state.
4. In the common case, REST assumes HTTP is used directly as the client-server messaging protocol. HTTP headers, including Cookies, provide an application-agnostic mechanism to propagate context information between the client and server.

3.3. State commit layer

Changes to the local replica are committed to the global state under some global commit protocol, such as a Paxos based consensus protocol or a two-phase commit protocol. The commit layer for the global state may be chosen independently of the replication protocol and the commit layer is often supported directly by the data persistence layer such as Megastore [10] or Spanner [35]. However, a well defined interface between the local replica and the global commit layer has to exist in order to resolve conflicts from changes that are committed eagerly at the local replica. This section describes the requirements for such an interface.

3.3.1. Local commits

Client messages are committed locally (at its local replica) under various optimistic algorithms. The interface to commit a client message locally is

- ReplicaState#commit(Message) : MessageHistory

The local replica will update its current version of the *MessageHistory* after each client message is committed by appending the new message id to the *MessageHistory* instance, as shown in the following pseudo code:

```
Class ReplicaState {
        // CommitException to handled by the server application to indicate the failure to
        // update the replica and global state
        MessageHistory commit(Message m) throws CommitException {
                this.localCommitLayer.commit(m);
                this.localHistory.append(m);
                return this.localHistory;
        }
}
```

The use of an optimistic algorithm does not concern the above interface, which is expected to be invoked by the server software, such as the session layer, on behalf of the client in processing each client message.

The server-side software should not generate any server message in response to the client as the result of the client message before the change has been committed locally. Furthermore, the server-side software is responsible for producing a serialized format of the local *MessageHistory* for every server message that is to be sent to the client via the following interface:

- ReplicaState#getMessageHistory() : MessageHistory

Sufficient concurrency control is also the responsibility of the server-side software to ensure that the replica *MessageHistory* are generated and propagated, locally, in a serializable order in respect to concurrent client or server messages. This level of concurrency control is not a concern to the replication algorithm because the replication algorithm is executed behind the local commit layer and its interface.

3.3.2. Global commits

At the same time a client message is committed by a local replica, the same message is also to be committed globally via the following interface:

- GlobalState#commit(Message) : MessageHistory

The committed global *MessageHistory* will be used by the server-side software to compare with the local *MessageHistory* to resolve conflicts due to optimistic early delivery, as shown below:

```
Class ReplicaState {
        MessageHistory commit(Message m) throws CommitException {
            this.localCommitLayer.commit(m);
            this.localHistory.append(m);
            // asynchronous invocation of the global commit layer
```

```
            this.globalCommitLayer.commit(this.localHistory);
            return this.localHistory;
        }
    }
```

There is no requirement that the global commit and local commit of the same message have to be processed atomically. That is, the global commit layer may be provided by a different software module than the local commit layer, and the two layers do not have to share any common state in producing their respective *MessageHistories*.

Unlike replica *MessageHistories,* global *MessageHistories* are not propagated directly with the server messages, which are generated from the *ReplicaState* directly; e.g., also in response to read-only client messages.

For the above interfaces to work, both the local commit and global commit layers are expected to share the same data model, i.e., the same state key for identifying the target state of messages.

3.3.3. Message history invalidation and rollback

After a client message has been committed both locally and globally, the global *MessageHistory* is to be compared to the local replica *MessageHistory* in order to detect any conflicts that may invalidate the local *MessageHistory*.

A global *MessageHistory* may be validated against the local replica *MessageHistory* up to a specific message id via the following interface:

- MessageHistory#validate(MessageId, MessageHistory)

The first argument represents the most recent message that is committed globally as known to the local replica. The second argument represents the global *MessageHistory.*

If the above interface returns false for a given global *MessageHistory*, the local *ReplicaState* is to be considered as invalid, and a local rollback should be issued up to the last validated *MessageId* via the following interface:

- ReplicaState#rollback(MessageId lastMessageId) : void

Note we do not replay any client messages that have been received after *lastMessageId*, as it may break the consistency requirement of causality between the client and server interaction. Therefore once the *ReplicaState* has been rolled back, the local replica *MessageHistory* will be identical to the global *MessageHistory*, up to the same *MessageId*.

Upon such a state recovery, any further client interaction may experience a client-side recovery if and only if the client has received any server message that comes with a *MessageHistory* that is newer than the *lastMessageId*.

As an optimization, because the global *MessageHistory*, once committed, will never change, replicas may discard the part of *MessageHistory* that has been committed globally for the local version of *MessageHistory.* To further reduce the space overhead, once a single *MessageId* has become known to all the replicas, all the history up to this *MessageHistory* may be discarded from the global *MessageHistory* too. To avoid hash collision, discarded message history should be replaced with its current serialized value as the initial value of the new *MessageHistory* instance.

3.3.4. Global read consistency

The global commit layer in its rollback role also has to ensure consistent reads. That is, the local replica in detecting an invalid message history, is also reading the most recently committed global history. This is important to allow clients to identify the latest client message that has been proposed to the global commit layer, when the client is loaded with the global state upon a recovery.

3.4. Optimistic algorithms

A set of optimistic replication algorithms may be enabled by the local commit interface. This section will list different types of optimistic algorithms and the key performance issues they are addressing. An informal proof is sketched for each type of optimistic algorithm that shows that the end-to-end consistency is guaranteed under the COLOR system model.

3.4.1. Non-uniform optimistic delivery

Non-uniform delivery requires no quorum based acknowledgement from participating replicas. The basic model involves a sequencer which may be either a fixed replica or a rotating one. The latter model is also referred to as a token based total ordering. The sequence id assigned to a message by the sequencer is distinct from the message id created by the client when the message is originated.

Each client message will be ordered by the sequencer before it is delivered at each of the local replicas. Two different scenarios exist depending on whether the replica is the sequencer itself:

1. For a sequencer replica, the client message will be directly delivered to the local replica without any interaction with its peer replicas. The actual delivery is subject to the concurrency control policy on the local replica and the sequencer will usually introduce no delay in assigning the totally ordered sequence id for the message.
2. For a non-sequencer replica, logically the local replica needs to pass the message to the sequencer for delivery. At the same time when the message is delivered by the sequencer, the actual message and its sequence id will be multicast to all the other replicas, including the original replica when the client message is received.

Figure 3.2 illustrates the above message sequence and each message to or from a replica is annotated with an execution number. Messages with the same execution number are processed as parallel messages.

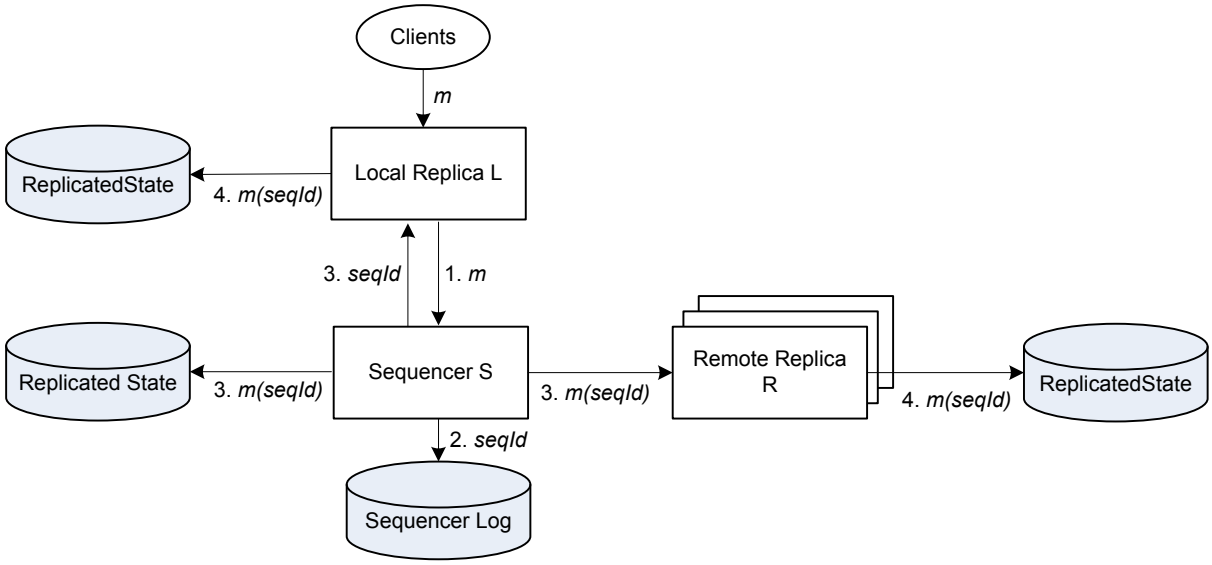


Figure 3.2: Message sequence of the sequencer base non-uniform delivery, with *seqId* being the sequence id assigned to *m* by S.

To enable non-uniform delivery, the local commit layer will invoke the following logic as part of *ReplicaState#commit*.

```
Class LocalCommitLayer {
        // ExternalException to be handled by ReplicaState class which will throw a
        // CommitException to the server application which in turn will generate an error
        // response to the client.
       Message commit(Message clientMessage) throws ExternalException {
              Message deliveredMessage = this.sequencer.deliver(m);
              this.logger.append(deliveredMessage);
       }
}
```

The saving from non-uniform delivery, when compared to uniform delivery, is obvious. When most client messages are dispatched to the sequencer directly, the overall performance of the system is comparable with a non-replicated solution. For many cloud based applications

or network environments, it is not difficult to achieve an optimal binding between clients and a single replica which plays the role of the sequencer.

As a comparison, Figure 3.3 shows a simplified message sequence when *m* is to be delivered via uniform delivery. Messages with the same execution number are processed as parallel messages.

The uniform delivery requires the master to receive a quorum of acknowledges, shown as "5: ack(*gid*)" in the figure, from other replicas. Each acknowledge from a replica also involves writing logs at that replica.

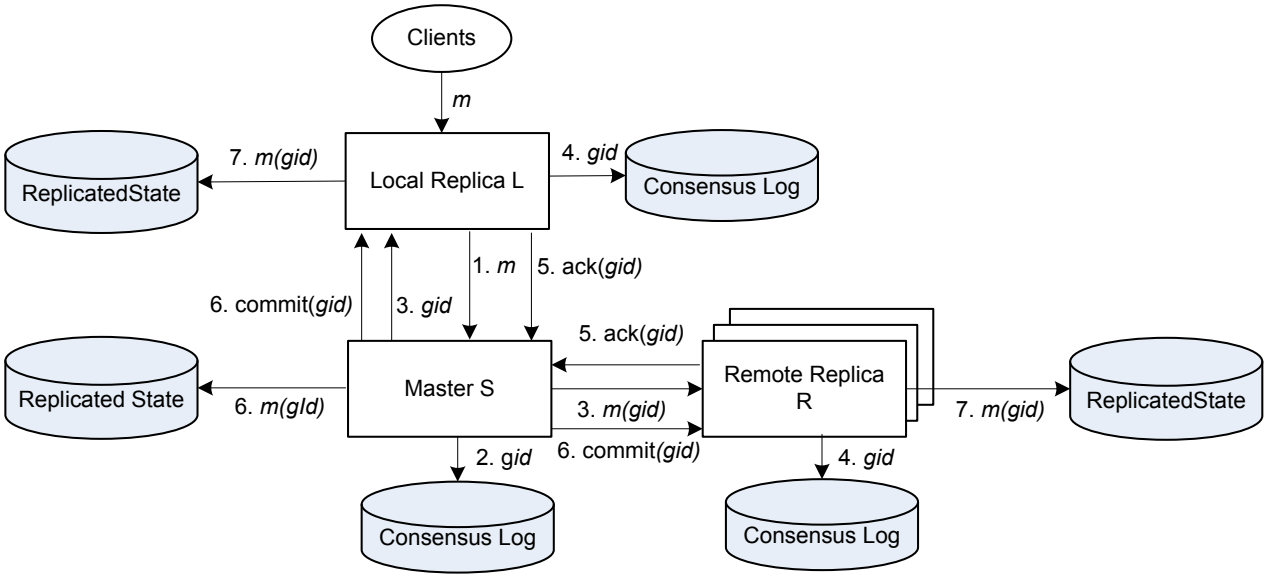


Figure 3.3: Message sequence of the uniform delivery, with *gid* representing the global history version committed for *m* by S.

Under normal conditions when there is no failure on the sequencer, non-uniform delivery introduces no inconsistency on either the server or the client. Because the cost of assigning a sequence id on the sequencer is a very cheap operation, the overall algorithm is a non-blocking algorithm for the sequencer as well as for all other correct replicas.

Inconsistency happens when a sequencer fails in the middle when a message is being replicated. Due to the non-blocking nature of the multicast algorithm, the client message may

simply be lost if the sequencer fails immediately after the message has been delivered by the sequencer itself. It is also possible that the client message has not been multicast to all the replicas when the sequencer fails.

Traditional multicast algorithms often adopt the VSYNC model in the case of any replica failure to recover from any possible loss of messages and to keep all the replicas in sync with each other in terms of all the messages being delivered against different versions of the multicast group (also known as views). This kind of tight synchronization model creates problems in a geographically distributed cloud environment as replicas may fail to be reached in a timely manner due to network or infrastructure reasons.

In typical cloud based environments, we expect a new sequencer to be elected via a consensus layer such as the one used by the global commit layer. Any replica may propose a change of the sequencer to the global commit layer, which will further inform all the replicas that are still correct and therefore connected to the global commit layer. Such a function is to be managed as part of the replication protocol, such as a replica management layer. The use of an external consensus layer will make the sequencer election much more stable in regards to the timely termination even when replicas are suffering from communication delays or network partitioning. Since the majority of failovers are scheduled failovers, it is unlikely that the underlying consensus layer will be subject to a master election of its own at the same time, which typically will suspend the global commit layer for an unpredictable duration.

For optimization purposes the sequencer may choose to be on the same replica that plays the master role of the global commit layer. Such an optimization will improve the performance of the global commit layer, if the location of the sequencer is otherwise insignificant to the performance of the client access. This is often not the case as the master of the global commit layer is often statically assigned while the sequencer may be changed in response to load-distribution of client traffic or changes of network access conditions.

To simplify the overall solution, the COLOR algorithm will simply terminate the optimistic algorithm in face of failure events such as a sequencer failure. This makes it a lot easier to reason about the correctness of the overall solution.

Specifically, in contrast to virtual asynchrony or other optimistic algorithms as discussed in Chapter 2, there is no complicated message recovery algorithm in response to any sequencer or replica failure. This is because all messages will be delivered simultaneously via the global commit layer which will be responsible for the eventual global commit of all received client messages as long as a message has already been proposed to the global commit layer before the sequencer fails. Any undelivered messages that have already been received by any non-sequencer replicas, including the original replica, will simply be discarded. This approach is adopted to ensure the non-blocking nature of the optimistic replication protocol. In contrast, many existing solutions will block the replication execution in such an event such as master or sequencer failover.

Informally, we establish the following properties below to justify to the reader the correctness and effectiveness of the algorithm:

Server Validity

- Under normal conditions when there is no replica failure, the server side state will remain valid due to the total ordering guarantee.
- Any failover event from the optimistic replication may cause invalid local replica state; and such state will be recovered as soon as the global commit layer catches up with the message delivery.

Client Validity

- Under normal conditions when there is no replica failure, the client side state will remain valid against the local replica(s) the client interacts with.
- The client will maintain the basic causality consistency in the event of failover as messages from invalid replica state will require the client to recover too.

Causal Delivery

- Server messages that are generated as the result of client messages will remain causally valid between a single client and its local replica until the replica rolls back its local state.
- Client messages that are lost to the global commit layer will trigger a local replica rollback, which will further invalidate the client state.

Non-blocking Delivery

- The failover of a sequencer will not suspend the delivery of the global commit layer, which delivers messages independently of the optimistic replication algorithm.
- The global commit layer only delivers messages received from the current (i.e., new) sequencer. In other words, new sequencers are not required to recover or reorder messages delivered by the previous sequencer.

3.4.2. Threshold-based optimistic delivery

Thresholds are added to non-uniform delivery to put a tunable limit on the time, message buffer size or quorum level when a consensus based delivery cannot be finished in a timely way. Threshold based delivery is layered on top of non-uniform delivery as the basic total ordering is required to minimize unnecessary conflicts and recovery.

The following thresholds may be enabled for a sequencer replica:

1. Time threshold: block the local delivery until a certain time interval has passed, unless the global commit layer notifies the delivery first;
2. Buffer threshold: block the local delivery until a certain number of messages have been buffered, unless the global commit layer notifies the delivery first;
3. Quorum threshold: block the local delivery until a delivery acknowledgement has been received from a certain size of replica quorum, unless the global commit layer notifies

the delivery first. The threshold quorum size may be as low as 1 which will significantly limit the loss of messages.

The following pseudo code shows how a time threshold may be applied to when a message is committed locally:

```
Class LocalCommitLayer {
    Message commit(Message clientMessage) throws ExternalException {
        // conceptually, the commit is blocked by a timeout period
        Thread.block(timeout);
        Message deliveredMessage = this.sequencer.deliver(m);
        this.logger.append(deliveredMessage);
    }
}
```

For non-sequencer replicas, if the replica is not the original replica for the client message there is no threshold to be enabled because when the message is received, it should be delivered according to the sequence id assigned by the sequencer.

On the contrary, an original replica may have a time based threshold to deliver the message locally when the sequencer is not responsive. At the same time the original replica should initiate a new sequencer election. This kind of non-blocking failover will be discussed in the next section.

The following proof sketch is similar to the one given for non-uniform delivery, and is specified for the sequencer based total-order algorithm. The same proof may be applied to other types of total-order algorithms such as a token based one.

Server Validity

- The server state will remain valid under normal conditions as the sequencer is responsible for deciding the total order of messages.

- The threshold will reduce the probability of message loss in the event of sequencer failures. The existence of the threshold is transparent to non-sequencer as well when a sequencer fails.

Client Validity

- The existence of thresholds will delay the client response for clients that interact with either the sequencer or other replicas. However the client response remains valid under normal conditions.
- In the event of failover the client invalidation and recovery behave the same as in the non-threshold case.

Causal Delivery

- The threshold will not affect the client-server interaction and the causality requirement between the client and server.
- Client messages may still be lost by the global commit layer, and as in the non-threshold case, such an event will trigger a local replica rollback, which will further invalidate the client state.

Non-blocking Delivery

- The threshold injects a minimum timeout to reduce the potential conflicts and the timeout is injected by the sequencer; i.e., the message sender, and as a result the algorithm does not introduce any extra acknowledgement from the receiver and remains as a non-blocking algorithm.
- The threshold will not change the failover procedure, which remains a non-blocking process.

The above algorithm assumes the sequencer is a static sequencer during the normal execution. The author doesn't believe that token based non-uniform multicast algorithms offer any real value in the cloud environment. Token based algorithms rotate the role of sequencer among all replicas and it offers a better performance when the system load is high; i.e., when a

replica gets the token, there are always pending messages to be sent on the replica. This is not really applicable to the COLOR model since messages are originated from clients which tend to interact with dedicated replicas as decided by the load balancing policy.

3.4.3. Non-blocking failover

The process of failover for either the uniform delivery or the non-uniform delivery involves the election of a new master or sequencer. Such a process is by nature a blocking process as the new master or sequencer has to be acknowledged by at least a majority of replicas. In the event of network partitioning, the proposal of a new master that remains connected with a majority of replicas may involve multiple runs of the master election. During a master election the replica state becomes essentially unavailable to clients. In the case of non-uniform delivery, the election of a new sequencer shares the same problems as a master election.

Failovers may also be introduced proactively as a way to enable global load distribution. During such events, a sequencer or a master may be moved to a new replica which is either closer to the clients or has more capacity to handle client traffic.

Specifically, one of the goals of optimistic non-uniform delivery is to be able to serve client messages directly on the sequencer. As a result, a new sequencer may be elected by the replica manager simply for the purpose of optimizing the system performance. In some cloud environments, such an event may happen as frequently as every several minutes. That is, a sequencer may be moved to a different replica for reasons beyond replica failures. We call such events as proactive failovers to distinguish them from failovers that are triggered by actual failures.

The master of the global commit layer may also be deliberately moved for the same reasons, although this is less likely as the master location has less effect on performance than the sequencer location for the non-uniform delivery. When it is actually desired to move the

master of the global commit layer to a different replica in response to load distribution, such events often happen at a much slower pace, such as in hours or days.

Non sequencer or non-master replica failure is not a concern here as such events do not trigger any election of a new master or sequencer. As discussed earlier, the optimistic algorithms do not try to recover messages in the event of failovers. As a result, unlike VSYNC based non-uniform multicast, there is no need to run any synchronous “view” update. A replica may be removed or added to the system as an offline process independently of the local or global commit layer and their algorithms. This strategy is aligned with most Paxos implementations too.

The detail of the non-blocking master election is laid out as following. The first part is the election of a new sequencer for the non-uniform delivery algorithm.

1. When a sequencer failover is suspected or scheduled, the proposal replica does not have to be blocked on receiving acknowledges from the underlying consensus algorithm.
2. The proposal replica may start to deliver messages as soon as the proposal replica and the new sequencer agree on the election of the new sequencer, which may be the proposal replica itself.
3. The old sequencer may not be aware of the new sequencer and as such may continue delivering messages to the global commit layer and to multicast messages to other replicas until the election algorithm is completed and the new sequencer is acknowledged by all replicas.
4. Non sequencer replicas will deliver messages to the local commit layer as messages are received from either the new or the old sequencer.
5. At any point of time a replica should only accept messages from the sequencer that is known to the replica. It is possible that a replica may receive messages from the new sequencer before it has the chance to acknowledge the new sequencer. The solution to such events is left to implementations.

6. Duplicated client messages should be filtered out by the local commit layer and should never be delivered to the global commit layer.

When the non-uniform delivery is not enabled, it is also possible to enable a transient sequencer when the global commit layer is blocked by the master election. The overall behavior is the same as enabling the non-uniform delivery with an unconfirmed sequencer. We will skip the detail of such algorithm as the value of adopting such an optimistic failover is limited since master change is a rare event for the global commit layer and there is no sound reason to not enable the non-uniform delivery under normal conditions since the potential impacts to clients are the same.

The following proof sketch shows how the consistency requirements may be met with the non-blocking failover algorithm, with respect to the algorithm elements listed above.

Server Validity

- There may be multiple replicas that assume the role of sequencers in the system, and each of them may generate different sequence ids for messages that have been received by these replicas. There is no guarantee that any message delivered to the local commit layer will not be rolled back if conflicts are generated by the global commit layer.
- As soon as the new sequencer is agreed upon by all the replicas as determined by the underlying consensus layer, the normal condition resumes and the server state will remain valid.

Client Validity

- The client may receive server responses generated from invalid server state during the failover phase.
- The client will recover from invalid state as messages are delivered via the global commit layer, and as soon as the new sequencer has been determined, the client

state will start to remain valid after the global commit layer catches up with delivering the messages generated during the failover phase.

Causal Delivery

- The basic causality guarantee between the client and server is maintained under the same conditions as in the non-uniform case. However, conflicts become more common during the failover phase due to the absence of global total ordering.
- When invalid server state is rolled back on a local replica that the client connects to, the client state will be invalidated too.

Non-blocking Delivery

- All the replicas will continue delivering messages to the local commit layer as messages are received from either the old or the new sequencer.
- Conflicts due to the existence of multiple sequencers will result in server or client state recovery. The election of the new sequencer will not block the message delivery to the local commit layer. There is no message gap or loss that the local commit layer needs to wait for on either a new sequencer or the old sequencer since conflicts are eventually solved by the global commit layer. Duplicated messages are filtered out before they are proposed to the global commit layer.
- The global commit layer will immediately deliver messages received from either the new or old sequencer, as the global commit layer is not aware of the sequencer failover event at all.

3.4.4. Read-only messages

Unlike traditional consensus based replication algorithms, such as Paxos, reads are never replicated under the COLOR model due to its client-oriented consistency model. The concept of stale reads is meaningless for an external client in the cloud environment. However,

the COLOR model explicitly requires the client-server interaction to maintain the basic causality consistency which is the key to the correctness of distributed client-server programming.

The fact that reads do not have to be replicated also fits the overall optimistic model. Since replica states are not necessarily consistent with each other or with the global state, there is no value to make reads in sync with distributed writes to ensure non-stale reads. As a result, the optimistic solution will also improve the performance of the global commit layer which will see much less throughput since reads are often dominating for most applications.

Moreover, without the optimistic layer, the global commit layer often has to resort to a lease based approach [27] to allow reads to be served locally on the master replica. Such an approach will keep the entire load such as queries on the master replica, which will become the bottleneck. This is because under a non-lease based solution although reads will have to be replicated and ordered with respect to write messages, a read only has to be executed once on a single replica that receives the client message. Furthermore, a lease based model will make the failover slower since the old master has to explicitly acknowledge the release of the lease.

The COLOR model consistency conditions ignore all the read-only client messages and rely on server messages to ensure the consistency requirement. This is unique to the COLOR based optimistic solution.

3.4.5. Explicit commit

A hazard of the COLOR model is the possibility of external inconsistency; i.e., when a client leaks its invalid state to external observers. An example is when the client interacts with an external system via some direct or indirect message passing subsystem during the client's interaction with the replica system. Since messages passed to an external system may have unrecovered side-effects; e.g., a transaction to a payment system, strict consistency must be ensured before the client sends any message to an external system. The same condition is true for human operators. As an example, a commit button, once executed, should never be

cancelled thereafter. To help clients to meet this condition, they can ensure strict consistency through an explicit global commit request.

To meet the above requirements, a client may be able to send explicit global commit messages to the server, which requires the following conditions to be met before the server sends back an acknowledgement message to notify the client that the commit message has been delivered:

1. On receiving a client message requesting an explicit commit, the local replica will check the message history from the client first and if the client state is valid, the replica will then propose the commit message to the global commit layer via the sequencer.
2. The replica will suspend itself from delivering any client messages that are received after the commit request message to preserve the FIFO order. For non-sequencer replicas, the replica will, however, deliver any message received from the sequencer which precedes the commit message at the global commit layer.
3. After the global commit has delivered the commit message, the local replica will wait for the local replica to catch up with a local message history that is a prefix of the global message history up to the commit message.
    a. When such a prefix is caught up, the commit message will be delivered locally. Buffered client messages will be delivered via the sequencer. A server message that acknowledges the commit will be generated by the application and sent back to the client.
    b. It is also possible that an invalid prefix may be detected locally and the client state needs to be invalidated as a result. Since the commit message has also already been delivered, the replica should still deliver the commit message to the local commit layer, followed by a client-invalidation message after the replica state is rolled back to the global state. The local concurrency control should ensure the above events are serialized properly.

c. It is also possible that the global commit layer may never deliver the commit message before a designated time out. This situation is inherent in any distributed systems that are subject to network partitioning, and in such an environment it becomes impossible to ensure the exactly-once delivery semantics between a client and server. In this case, it will be unknown to the local replica that the commit message will be delivered at all. Implementations may choose to retry the commit message again (under the same message id) or simply return an error message to the client to indicate the failure to acknowledge the commit. In the latter case, the application has to ensure that a non-idempotent commit will be committed only once. This is often achievable via a client-initiated retry that employs a read-then-write semantics.

4. It is also possible to make a read-only query message as a commit message to force a client-side read of the latest global state.
    a. Once a read-only message is treated as a commit message, it will be totally ordered across all replicas in respect to all the concurrent client messages (which will update the global state). This makes such read-only messages equivalent to so-called “consistent reads” or “non-stale reads” seen in algorithms that only concern server state.
    b. Consistent reads have limited value under the client-oriented COLOR model. By the time a client receives the response to a read-only commit message, the server state may have already been updated with a newer version. Therefore, an application should avoid relying on consistent reads to ensure client side correctness.

Overall the above algorithm enables a hybrid model that satisfies uniform total ordering for selected messages even while the rest receive non-uniform optimistic delivery.

The following proof sketch shows the correctness of the above approach as well as its effectiveness.

Server Commit Validity

- The commit message will not be rolled back once delivered by the local commit layer. The commit message is delivered at the local commit layer only after the global history has delivered a prefix up to the commit message.
- Although the replica state may be rolled back due to conflicts received from the sequencer before the commit has been received, the commit itself will be delivered either directly or via the refresh of the global state whose message history must have the commit message.

Client Commit Validity

- Any server message that is generated as the result of the commit message remains valid even if the replica state is rolled back due to conflicts against the global commit layer.
- The commit message will only be delivered at the local commit layer after it has been delivered by the global commit layer. Although its delivery order with respect to other messages may be invalidated at the local replica, the commit message itself will be delivered again after the replica state is rolled back to the global state.

Causal Delivery

- Client invalidation remains effective and the commit message delivered locally may be preceded by invalid messages from a sequencer.
- External causality is guaranteed with respect to the commit message itself which will never be rolled back once delivered.

Non-blocking Delivery

- The commit message itself involves a blocking global delivery. However, messages received by a sequencer will be delivered while the local replica waits for the global commit layer to deliver the commit message.

- Client messages received after the commit message will be buffered by the local replica until the commit message is delivered. This is, however, to be considered a client-initiated behavior as a client should expect messages that are sent after the commit message but before the commit delivery is acknowledged to be buffered due to the strict FIFO order for all client messages generated by the same client.

3.5. Client state recovery

In this section we will discuss the interface between the client and server that allows a client to initiate a state recovery in the event of state invalidation. The requirements apply to all different types of clients, ranging from passive clients such as browsers to intelligent clients such as mobiles that are able to handle offline state merge.

3.5.1. Invalidation server message

When the replica executes a local rollback or when the replica detects an invalid client message history, the client has to be notified so the state of the client can be recovered to allow the client to continue interacting with the replica.

The invalidation message is to be generated as a regular server message. For clients that manage the local cache on the client side, such an invalidation message may be sent either as an RPC response message or as a server notification message. As a minimum requirement, in response to an invalidation message the client must reinitialize its client-side state with the current replica, and in doing so the client is expected to reset any local cache of the server state. This process is similar to how a typical online client application would behave in response to a general server error such as HTTP 500 server-error status; e.g., after a server process crashes and is restarted.

For passive clients that rely on a session layer for managing client-side state, the invalidation message will effectively reinitialize the entire server-side session.

The invalidation message may also include the current replica message history, which a capable client may use to recover from any invalid client-side state as discussed in Section 3.5.2.

We note that most stateful clients expect server-initiated state reinitialization even in the non-replication environment. One example is the use of optimistic concurrency control [137], which even has standard HTTP interface support such as E-tag. In a later section we will discuss how to map the invalidation server message to a standard client-server interface such as the REST/HTTP interface.

To summarize the interface for generating the invalidation server message is defined as following:

- ReplicaState#invalidate(Client) : Message

The scope of invalidation is the state key, and the *Client* object represents the current stateful client against which the invalidation is triggered.

3.5.2. Log-based recovery

For capable clients, the invalidation message may carry extra message history data to allow the client to recover the cached state directly on the client side instead of triggering a full re-initialization. The problem with a re-initialization is that the client may lose all pending edits from the user and the new state is non-deterministic in the view of the user. This is especially a problem when the client is failed over to a replica which possesses an older message history due to the lag of its local commit history.

In order for the log-based recovery to work, a new type of validation message has to be communicated to the client:

1. StateValidation#getLastMessageId() : the last message id that is committed globally from the same client.
2. StateValidation#getGlobalPrefix() : the message history that is committed globally.

The above message is to be communicated periodically to the client as a checkpoint message to update the current global commit status. Such an approach allows the client to recover from any invalid state as soon as possible while at the same time the client may still interact with the replica and initiate new messages.

In Chapter 7 we will describe the generic framework to support log based recovery on the client. The concepts here are described to illustrate the basic interface between the client and server to support such frameworks:

1. A client message is logged until a new *StateValidation* message is received that has a *MessageHistory* newer than the last *MessageHistory* saved from previous messages.
2. The client remembers the newest global *MessageHistory* that has been received. No client-side state should be rolled back beyond such a checkpoint.
3. The client also keeps the latest replica *MessageHistory* as in the general case. Whenever the client receives a global *MessageHistory* that invalidates the current replica version of *MessageHistory*, a local recovery is triggered. This condition may also be triggered by an explicit server invalidation message as in the general case.
4. Instead of doing a full state re-initialization, the client should load the latest replica state and then replay all the messages that are not yet confirmed to be committed to the global state.

The following is the proof sketch of several key properties of such a solution indicating that it is correct and effective.

Bounded Client Logging

- The total number of local client messages to be logged is limited as messages will be cleared as soon as they are committed to the global *MessageHistory*.

No Replay Duplication

- When a replica state is rolled back due to conflicts between the local commit layer and the global commit layer, there is no replaying on the server side.
- If the client possesses a replica *MessageHistory* that is older than the latest version of the global *MessageHistory*, only messages newer than the latest message id for the given client will be replayed and there is no risk the same message will be delivered twice to the local replica.
- If the client possesses a replica *MessageHistory* that is newer than the latest version of the global *MessageHistory*, after the re-initialization only messages older than the latest message id for the given client will be replayed. In the meantime, the replica will not accept any messages from the given client since its local replica version of *MessageHistory* has to be invalidated first.

Three-way Merge Compliant

- The overall invalidation and replay mechanism essentially represents a three-way merge algorithm that is popular in offline algorithms and optimistic concurrency control algorithms.
- The server-initiated *StateValidation* enables the client to aggressively solve any conflicts with the global state, and the client-server interface ensures such recovery can be done in a streaming fashion without blocking the underlying replication algorithms on either the replica or the client.
- The global commit layer will remove any duplicated messages that have been delivered by a non-local replica, such as the sequencer.

3.5.3. Conflict resolution

Although log based recovery allows the client to re-initialize to a baseline state and replay all logged messages that are not committed to the global state, the actual replay is

complicated due to the causality requirement between each message and the version of client-state against which the message was generated.

It is often the case that such replays will involve manual conflict resolution. Note the COLOR model employs online conflict resolution and the client messages may be generated based on different versions of replica state, part of whose history may have been rolled back.

To make the state recovery transparent to the end user, automatic conflict resolution has to be employed, in which the client tries to resolve the local state difference with the new replica state that is just committed in a way that will preserve all the client messages and their causal dependencies. In doing so, a client re-initialization will see the replica preserve all the effects from the previous client messages.

We will cover the above approach in more detailed in Chapter 7. Overall the approach requires the client replica to maintain the following information from the on-going client-server interactions:

1. Every client message must be logged until it has been committed by the global commit layer and the corresponding global *MessageHistory* becomes known to the local replica.
2. The *MessageHistory* that comes with every client message must be logged too on the client side. The *MessageHistory* essentially represents the causal dependency between the client message and a particular version of replica state.
3. In resolving a conflict against a global version of *MessageHistory*, all the logged messages that have not been committed globally will be replayed in their causality order with possible changes to the actual message in order to resolve conflicts.
4. Since the client will not know whether any logged message may be delivered at a later time to the global commit layer by a remote replica, any changed message should be transformed into its original client message first, followed by a new message that carries the difference with the original message. The global commit layer will filter out any

duplicated messages from the same client but new messages will be delivered in order as they are generated from the replay operation.

3.6. Sharding and partial ordering

In this section, we will discuss the concept of state sharding and partial ordering, which are two closely related concepts.

State sharding partitions a single state key into multiple disjointed key spaces, which may be organized as a single-level of child state or multiple levels of nested child state. The latter will involve a hierarchical naming space, which fits very well with any URI based naming scheme.

Partial ordering is a different form of optimistic ordering. Instead of ordering all the messages against a large state space, which often do not conflict with each other, messages are ordered only when they are updating the same state partition, which has a much smaller state space size. Partial ordering has been explored in several consensus algorithms, such as Generic Paxos [90], and Generic Broadcast [113]. However, their practical use has been limited due to their complexity and the need to know the ordering relation between messages a priori before messages are actually delivered to the storage layer.

Under the COLOR model, state sharding is exploited as a form of partial ordering in a much simpler and more practical way. Given that state sharding is done statically, it is essentially an optimistic partial ordering solution in which data consistency must be checked and recovered in the following ways:

- Messages for different shards may actually result in different global orders that may conflict with each other.
- Conflicts between shards will only be discovered when messages are actually committed by the global commit layer, which may invalidate the order decided by each replica.
- Messages orders between shards need be rolled back under the above conditions.

3.6.1. Overview

Figure 3.4 shows how sharding affects the original layered state representation, detailed as following:

1. Global state is not affected; and replica state is sharded completely with message histories sharded accordingly too.
2. Client messages are only sharded when they are delivered to the local replica state; and server messages are not sharded. Message histories of the local replica are generated out of the sharded client messages; and sharded replica message histories are propagated to the client together with the non-sharded global message history as known to the local replica.
3. Client state is not sharded either. That is, sharding is transparent to the client-side application and its state cache.

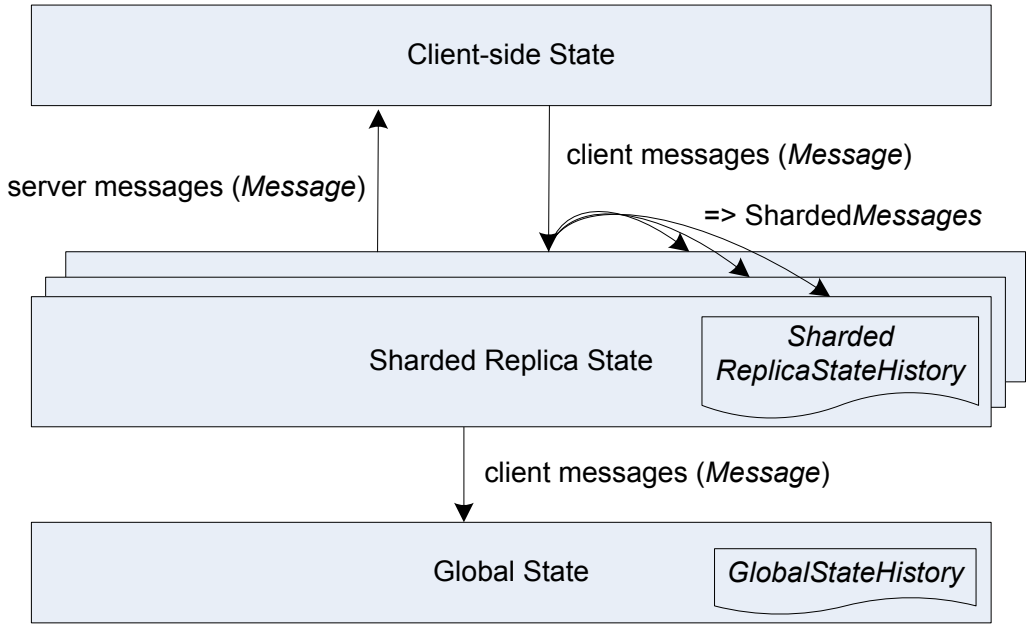


Figure 3.4: Sharded state representation.

To help illustrate the above changes, we again use a shared online whiteboard as an example application, for which the sharding concept will be applied as following:

1. First, the whiteboard state will be partitioned into 10*10 sub-boards, and as a result, the size of each sharded state is only 1% of the original whiteboard.
2. When a client message is received by the local replica, a set of sharded client messages will be generated, each of them will only touch a single shard; i.e., a sub-board that is 1% of the size of the entire whiteboard.
3. Each sharded client message will be delivered independently of other sharded messages and there is no ordering relation for messages against different shards.
4. The original client message will be delivered to the global commit layer as a single message.
5. For server messages, the server-side application will notify the client via messages generated for the entire whiteboard, as opposed to each single shard. However, the server message will include sharded replica message histories as generated by the local replica.

3.6.2. State sharding and scalability

The key benefit of state sharding is to improve the scalability of the replication system by allowing parallel message deliveries, e.g., over Paxos, against one single global state space that is partitioned into individual subspaces.

Figure 3.5 shows how a global state space is partitioned into multiple shards, which then enables a much greater parallelism when messages are being globally ordered and when messages are being delivered at individual replicas.

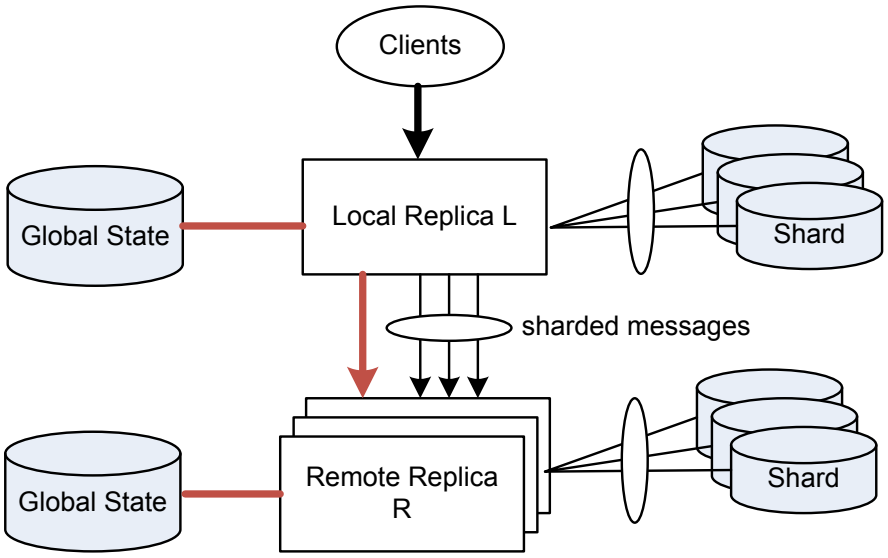


Figure 3.5: State sharding increases the parallelism of message delivery.

For typical cloud applications, a sharding scheme might be based on user accounts. Note that for state spaces that are naturally disjointed, there will be no sharding involved. It is always desirable to have fine grained instances of global state to allow messages that target independent global state to be delivered independently of each other. However, global state may not always be shard-able; e.g., data from different user accounts may be related in some way which means that messages involving different user accounts have to be delivered in a single globally total order.

We also consider the number of shards represents a tunable parameter for the COLOR model. With any hierarchical sharding model, which is very common for large volumes of shared data in the cloud environment, a COLOR based solution is able to tune the granularity of *ReplicaState* to a size that reaches the desired balance between scalability and consistency.

State sharding is an optimistic delivery scheme that is orthogonal to the non-uniform delivery. It is possible to apply both schemes at the same time or individually. While the non-uniform delivery has the capability to minimize the response time to match non-replication cases, state sharding has the potential to scale the replication indefinitely.

With state sharding, the sequencer (for the non-uniform delivery) or the master (for the uniform delivery) will also be sharded effectively, as shown in Figure 3.6. That is, each shard will have its own sequencer or master. Client messages will have to be routed to their corresponding sharded sequence or master based on the state each message touches, as described in Section 3.6.2.

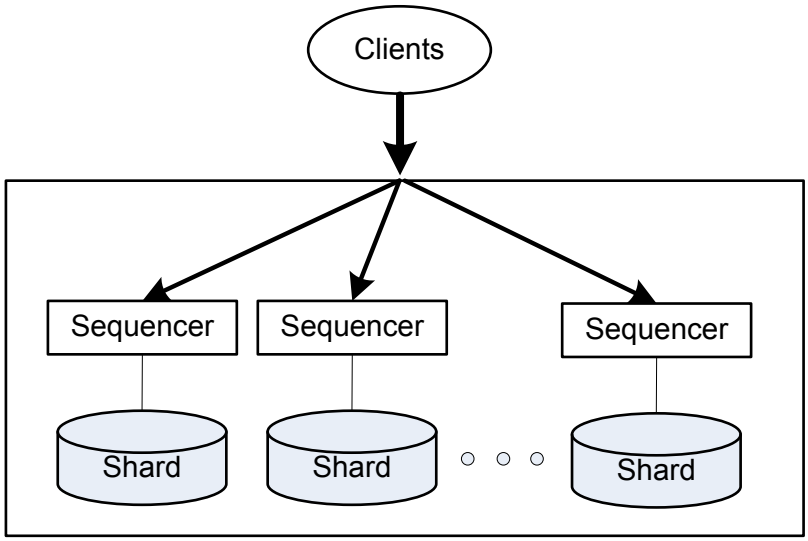


Figure 3.6: Sharded sequencers under state sharding.

3.6.3. Sharded message histories

On the surface, state sharding simply expands the naming space of *ReplicaState*, that is the state key will have a larger value space to name finer-grained state instances that are replicated locally.

Clearly, different shards may be related to each other via implicit or explicit relations, for example via an operation that updates multiple shards atomically. The atomicity requirement is often the result that those individual shards are not to be represented as completely separate state instances. The formal definition of relation between data instances is beyond the scope of this thesis, and we will address the underlying concept with concrete examples in later sections.

From the view of *GlobalState*, a sharded *ReplicaState* instance is regarded as a single instance of *GlobalState*. This means that the global message history is to be generated against the key of *GlobalState.* This is necessary because the global commit layer is not involved in the optimistic state sharding.

From the view of *ReplicaState*, individual message histories for each shard are generated independently as messages are delivered by the local commit layer. Since this is a kind of optimistic delivery, the message history of a sharded state instance may be invalidated by the global commit layer when it identifies a conflict.

Two questions arise on how to invalidate a replica *MessageHistory* over a global *MessageHistory*, given the latter is not aware of the sharding scheme:

1. How to specify messages against multiple shards that are to be delivered atomically to the global commit layer?

2. How to invalidate a sharded *MessageHistory* against the single global *MessageHistory*?

The solution to the first question is an extended *Message* type, defined as *ShardedMessage*, which conceptually holds a map of messages which are keyed by shard keys.

- ShardedMessage[shardKey] : Message

Here *shardKey* is the sub-portion of the global state key space that identifies a unique shard. For each unique shard, there will be a sub message (of *Message* type) that contains only changes made to the given shard. Individual per-shard messages in a *ShardedMessage* are to be delivered as an atomic unit by the global commit layer. The most important property is that individual messages in a *ShardedMessage* have no FIFO order relation.

We note that the above definition of *ShardedMessage* or the actual sharding scheme does not concern the global commit layer as the global commit layer has no knowledge of any message-level sharding when messages against the *GlobalState* are delivered.

With the above extension, the second question may be addressed directly by the global commit layer now that the global *MessageHistory* that is managed by the global commit layer represents the global order of sharded messages too.

- GlobalState#validate(MessageId, MessageHistory)

The above interface takes a local *MessageId*, which is per shard as well as a local *MessageHistory*, and the global commit layer is able to invalidate any local *MessageHistory* should the replica-generated order fail to match the global order in respect to the specified shard.

The following example shows how a local MessageHistory may become invalidated after messages are delivered by the global commit layer.

- Shard instances of two state keys “/state/1”, “/state/2”
- 1st *ShardedMessage*: {“/state/1” : a1; “/state/2” : a2}
- 2nd *ShardedMessage*: {“/state/1” : b1; “/state/2” : b2}
- a1, a2, b1, b2 are message ids generated per each shard
- the two *ShardedMessages* have no causal dependency and therefore are two parallel messages to the same *GlobalState*

The delivery scenario is as follows:

1. Messages against different shards are delivered independently by local replicas. As a result, it is possible to see the following sub *MessageHistory* on local replicas:
    - “/state/1” : {a1, b1}
    - “/state/2” : {b2, a2}
2. At the same time, the two *ShardedMessages* are to be delivered by the global commit layer, which will generate the following global *MessageHistory*:
    - {(a1, a2), (b1, b2)} (it could also be {(b1, b2), (a1, a2)})
3. From this global *MessageHistory*, the *MessageHistory* for “/state/2” will be invalidated and reordered to

- “/state/2” : {a2, b2}

We note here that, assuming sharded states are completely exclusive to each other, one of the common reasons for messages that update two separate sharded states to be in conflict is due to the atomic update requirement. Intuitively, when two shards have to be updated within a single atomic update operation, they become related to each other. For example, data from two separate user accounts may not be seen as two independent state instances as one may think. We will explore the nature of relations between sharded states in Chapter 5 using a real example.

As for how the global commit layer sees atomic changes to multiple sub-states in a single global state space, the global commit layer simply applies these changes as a single update message to its global state. The serializability [18] is always enforced as defined under the COLOR system model which assumes the State Machine model as the underlying consistency model.

To summarize, the following pseudo code shows how the local commit may work when state sharding is enabled:

```
Class ReplicaState {
    MessageHistory commit(Message m) throws CommitException {
        ShardedMessage[] shardedMessages = this.shardClientMessage(m)
        // conceptually, sharded messages are to be committed in parallel
        for(ShardedMessage shardedMsg : shardedMessages) {
            this.localCommitLayer.commit(shardedMsg);
            this.localHistory.append(shardedMsg);
        }
        // global history only sees the original client message
        this.localHistory.append(m);
        this.globalCommitLayer.commit(this.localHistory);
```

```
            return this.localHistory;
        }
    }
```

We also note that even messages that involve only a single shard may generate invalid local message histories due to sharded delivery. For example, the master that is responsible for a particular shard may happen to be located in a different cluster than the global master. This is, however, more of an environmental condition as the result of the optimistic sharded delivery running as a separate commit layer in parallel with the global commit layer in the runtime. We will explore the general problem of sharded delivery in Section 4.5 for the analytic evaluation and Section 6.3 for the experiment result, respectively.

3.6.4. Recovery from optimistic sharding

The detection and recovery algorithms for optimistic sharding are similar to those defined for the non-uniform delivery. The main difficulty is related to how the global *MessageHistory* may be applied to a sharded *MessageHistory*.

The following function reduces a global *MessageHistory* to a per-shard replica *MessageHistory*:

- MessageHistory#getShardHistory(shardKey)

This is useful to apply the global *MessageHistory* to the replica *MessageHistory* whenever an invalid state is detected.

Sharding is transparent to the interface between the client and its local replica. Only when a client message is applied to the local commit layer, it is decomposed into multiple sub-messages against the state shards separately. As a result, the client interacts with the local replica only in the context of the non-sharded replica state, and client messages are always generated as a single atomic message as far as the client state and the client-server interface are concerned.

Compared to the non-sharded case, the replica *MessageHistories* that a client receives will be a set of *MessageHistory*, one for each sharded state instance as keyed by the sharded state key. This difference is transparent to clients that will not attempt any log based recovery as *MessageHistory* is propagated merely as a message context between the client and server.

For clients that are capable of log based recovery, the invalidation message will include the first non-committed client message for which at least one sharded state has an invalid *MessageHistory* that needs be rolled back. In other words, the invalidation message will communicate the original client message id as opposed to message ids included in the *MessageHistories* that are generated per each sharded state. The client message id can be easily generated from the global *MessageHistory* as long as the *ShardedMessage* contains the original client message id as:

- ShardedMessage#getClientMessageId

Likewise, the *StateValidation* message will include the client message id, in addition to individual per-shard message ids, to enable the client-side recovery behavior to apply the latest message ids that have been delivered by the global commit layer. However, the sharding is not completely transparent to the client as the replica *MessageHistory* is always generated per each shard.

Lastly, we note that it is possible that a *ShardMessageHistory* involves only one shard. Even in this case the client message will still address the entire non-sharded state directly because sharding concerns only the local commit layer. When only a single shard is included in a client message it is unlikely that the client message could be rolled back due to sharding reasons, as the message order proposed by the local commit layer will always be respected by the global commit layer unless conflicting proposals are made to the global commit layer from different replicas due to runtime conditions such as when masters of different shards are located in different clusters.

3.6.5. Server messages blocked on FIFO order

Although the FIFO order from a single client or client communication channel is broken with the sharded delivery, the validity of the consistency requirement will still hold. That is, all the replicas will observe exactly the same state changes, except the timestamp order of changes across different shards will be different between replicas.

It is possible to adapt COLOR to respect the FIFO order of the client messages (of a single client), by making the server-side session layer generate server messages in respect to the FIFO order of the client messages that may have been delivered in parallel against different shard instances. The FIFO order may be derived from the message id's supplied by the client. When the temporal order has to be respected, such a requirement may involve delaying of generating server-messages to notify a client of changes made by other clients. This adaptation is not part of COLOR as implemented in the present work.

Cross-shard FIFO order is not a requirement for most applications. For applications that do need to enforce timestamp-based FIFO order, sharding may not be a good choice as such a requirement generally breaks the assumption of sharded state, which assumes messages against distinct shards will not conflict in their delivery orders.

Chapter 4 Performance Models

This chapter describes mathematical performance models for some of the features of COLOR, in order to understand the properties of the proposed COLOR model and its differences from the strict model, and as well to offer insights into implementation strategies. This chapter breaks the performance models into three parts:

- The response time for non-uniform delivery (NU) in Section 4.3, with comparisons to a strict replication strategy (U) in Section 4.3.3 and to the non-replicated case (NR) in Section 4.3.7.
- The probability of client transaction failures in the non-blocking master failover, in Section 4.4.
- The response time for state-sharding delivery (SH) in Section 4.5.

The study of performance models first focuses on the response time, which is always an important metric for cloud-based applications. The response time analysis addresses the steady state performance, when there are no failures in the system that may affect the system performance. The analysis considers the scenario for a message round, from when a client message is generated to when the local commit layer delivers the message and hence is able to generate a response as a server message. Within a message round it considers the critical path, including processing delays which may become bottlenecks that limit scalability. Our analysis also shows that the response time under normal conditions may match the non-replication case (NR in Section 4.7), which represents the lower-bound of any optimistic solution.

The response time of optimistic solutions under the COLOR model is then compared to the strict replication model (such as in Paxos), which represents the common implementation of the global commit layer.

Using elements of the response-time model, the key penalty of the optimistic solution, which is the probability of inconsistency, can be analyzed. The inconsistency is a measure of the cost for adopting the optimistic replication algorithm, and it may be controlled by two tunable parameters, an injected local delivery delay under the non-uniform delivery, and the level of sharding under the state-sharding delivery. Both parameters are continuous parameters, which enable a finer-grained and model-based approach to manage the performance and consistency tradeoffs, in comparison to other optimistic solutions that support only all or nothing approaches.

For each section, the performance analysis will be presented under the following structure:

1. Message scenario.
2. Critical path analysis.
3. Comparison to strict models.
4. Failure scenario.
5. Inconsistency cost.
6. Tunable parameters.

4.1. Performance modeling approach

Modeling performance of distributed algorithms is hard, especially for multicast or total-ordering algorithms that often involve a large number of discrete states. The approach this thesis takes for the analytic modeling of the COLOR algorithm strikes a good balance between simplicity and effectiveness.

We reviewed many existing works on analyzing multicast or replication algorithms, ([26] [33] [36] [38] [96] [97] [105] [127] [138] [143] [145]) and decide to focus on the upper-bound analysis to exclude implementation or environment details from the performance analysis while keeping the results meaningful for predicating performance in practical settings.

At the same time, we choose not to assume any fixed distribution for the network links, and rely on the actual measurement and numeric solution to generate results that will better match the run-time conditions [11]. To the best of our knowledge, no previous work has attempted such an approach.

Lastly, we believe a high-quality analytic model has many benefits for both researchers and implementers of optimistic algorithms. For such a purpose, we have followed the general guidelines of model based performance engineering [15] [74] in how we create the modeling solutions for the COLOR algorithm.

4.2. Notations

The following notation has been defined for specifying the mathematical models in this chapter. The system is viewed from a representative client C attached to its local replica L. For performance comparisons the delay from C to L is beyond the present analysis, so the response time T_L at the local replica L will be determined. In the notation:

- H denotes a host node in the system, such that:
    - H takes value C for the representative client, S for the sequencer or master replica, L for the local replica that the client is connected to, and R for an arbitrary remote replica that L interacts with.
- X denotes a random variable for the one-way message propagation delay between replicas, with density function $f_X(x)$ and distribution function F_X ($F_X(x) = \text{Prob}\{X \leq x\}$). Note that *x* is a value of X. All propagation delays between replicas, including any master or synchronizer node, are taken in this chapter to have the same delay distribution.
    - X_C denotes the propagation delay from the client C to its local replica, with the distribution function $F_{XC}(x)$.

- OS(j,{X_1,...,X_n}) denotes the j^{th} order statistic of the set {X_1,...X_n} of random variables. Thus OS(n, {X_1,...X_n} is max{X_1,...,X_n}, and OS (q, {X_1,...,X_n}) is the q^{th} value ordered from smallest to largest.
- P_H denotes the processing time (CPU time) at node H. In detail,
    - P_S = processing time at S to sequence a new message, and send it to *n* other replicas.
    - P_L = processing time at a local replica L to deliver a message to the local commit layer.
    - P_R = processing time at a remote replica R to acknowledge a message from S, including writing any log for the acknowledgement message.
- D_H denotes the delay for all the operations at node H to support a message round. These delays include processor scheduling delays. In detail:
    - D_S = delay at S to complete P_S.
    - D_L = delay at L, including serialization delays to complete P_L.
    - D_R = delay at R, including commit delays such as writing to a log to complete P_R.
- In addition, D_{th} denotes an intentional delay to achieve a threshold for blocking an operation such as sending a message.
- N_R = total number of replicas.
- λ_C = system throughput in messages per second from all clients interacting with L.
- λ = system throughput in messages per second for all the replicas ($\lambda = N_R * \lambda_C$).
- μ = failure rate (failures/s) for the sequencer S, assumed to be a Poisson stream (exponential time to failure of mean $1/\mu$).
- λ_S = server-message rate generated for all clients connected to L. This rate covers update responses, query responses and server notification messages.

Then define:

- $Ack_i = X + D_{Ri} + X$ = round-trip delay to send a message from S to a remote replica R_i, process it at replica R_i and send the acknowledgment back to S.
- $Q(q, N_R) = OS(q, \{Ack_1,...,Ack_{NR}) =$ delay to obtain a quorum of size *q* of replies from N_R replicas.

Mean values are inadequate for evaluating response time, since percentiles are often specified for QoS. Therefore in this analysis we will focus on the distribution results (displayed as box plots [20] using MATLAB [99] [100]) to demonstrate the effectiveness of the analytic models.

4.3. Between-cluster communication delays

One of the key factors for performance is the communication delay between clusters, represented by the propagation delay X. Performance of geographical replication algorithms is especially affected by the variation of X. This work uses an empirically determined distribution for X, expressed as a histogram. If the relative frequency of the i_{th} cell, ranging from x'_i to x'_{i+1} is p_i:

- p_i = Prob{X is in cell i with $x'_i \leq x < x'_{i+1}$} = (relative frequency measured for cell i)

then the density function $f_X(x)$ is approximated as a uniform distribution over each cell:

- $f_X(x) = p_i / (x'_{i+1} - x'_i)$, for each interval $x'_i \leq x < x'_{i+1}$

For the purposes of this chapter, the distribution of X is modeled from data samples collected for communication between a cluster located in the east coast and a cluster located in the west coast. The sample was taken for a high QoS traffic class to minimize interference from other traffic. Also:

- A short sampling interval (60 min.) was used to avoid slow traffic trends such as daily usage shift across the network while keeping the distribution meaningful for accurately modeling the replication algorithm runs.

- Box plots were generated by MATLAB showing the minimum, median, maximum, 25th percentile and 75th percentile.
- Outliers were removed using the MATLAB box plot outlier functions.
- For comparison with the experimental results, the samples were collected at the same time as the measurement experiments described in Chapter 6, and the empirical distribution was then used in the mathematical models that are compared with measurement.

Figure 4.1 shows one example histogram of X between two clusters across the US. The distribution was collected over a period of 60 minutes, with one RTT sample per second, and using a response-time-sensitive QoS traffic class, which is relatively immune to packet loss.

In certain Monte Carlo evaluations [142] on the mathematical models, random values of X were generated using a discrete approximation over the midpoints values of the cells (that is, $X = (x'_i + x'_{i+1})/2$ with probability p_i)

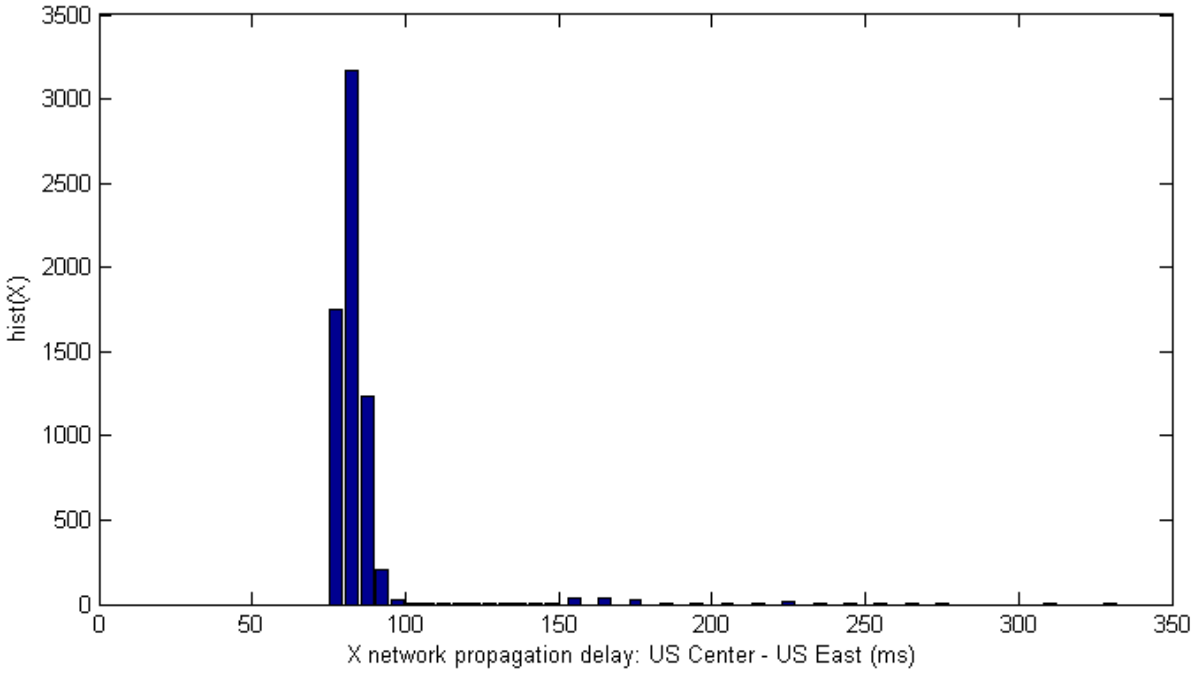
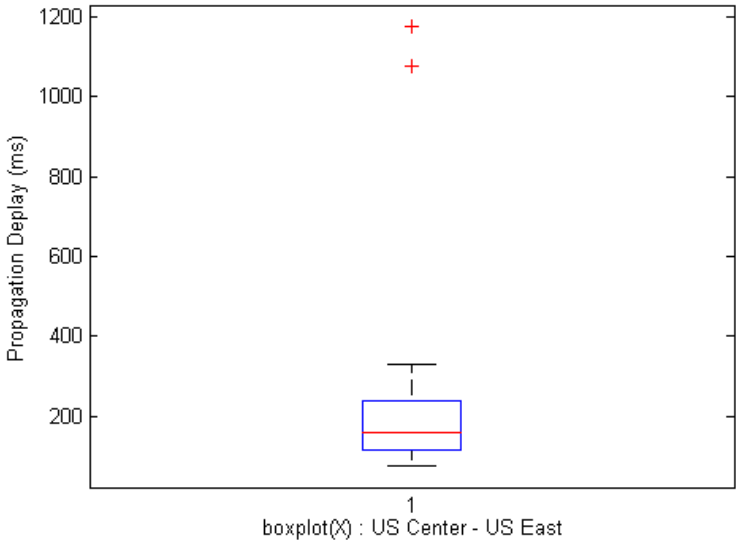


Figure 4.1: The histogram of the network propagation delay between a data center in US Center and a data center in US East.

The Y axis is the total number of samples collected for a given histogram bucket during the entire sampling period. Delays beyond 350ms are removed as outliers.

For the analysis of the rest of this chapter, we will use the MATLAB box plots to decide the range of max, min with outliers excluded by the tool, for summarizing results that involve delays or response times. Figure 4.2 shows how the actual histogram may be presented with the standard box chart, in which the outliers are marked as red crosses.



.

Figure 4.2: Box plot showing the medium, 75th percentile, 25th percentile, max and min of the entire distribution generated from the histogram samples in Figure 4.1.

Outliers are identified using the standard box plot statistics tool provided by MATLAB

4.4. Non-uniform delivery

The non-uniform delivery reduces T_L by eliminating the need for the master replica S to receive a quorum of acknowledgements from all the replicas. The inconsistency cost from the non-uniform delivery is to be measured as probability of invalid states seen by clients when a

sequencer fails. To reduce this probability, the delivery on a local replica is delayed, with a fixed timeout which serves as the deadline of message delivery. This strategy allows the timeout value to be used as a tunable parameter to enable a quantitative tradeoff between the client-perceived response time and the inconsistency cost.

4.4.1. Message scenario

A message round is defined as a one-way message-passing from one replica to another remote replica. Under normal conditions, it will cost two message rounds when a message is delivered on a non sequencer replica, and it will cost zero message rounds if a message is delivered directly by a sequencer replica which happens to the local replica for the client. The latter scenario could be dominating when many clients are collocated in a single geographical area and the load-balancer is able to direct their traffic to a single replica, which would be made the sequencer dynamically in order to have the optimal performance.

The following scenario assumes that L is not S:

1. A local replica L receives a client message *m* against a given *ReplicaState*.
2. L passes *m* to the sequencer replica S, if L is not *S*.
3. S assigns *m* a global sequence *seqId*, and sends *seqId* to L and *m* along with *seqId* to other remote replicas.
4. All replicas, including S and L, will deliver *m* according to the totally ordered *seqId*. This means that *m* will be delivered only after all preceding messages m' (*m'* with *m'.seqId()* < *m*.*seqId*) have been delivered.

4.4.2. Critical Path analysis

Based on the above message scenario, the non-uniform delivery requires no acknowledgement from remote replicas before messages are delivered locally. The only

potential bottlenecks in the system are processing bottlenecks at S and R. Both D_S and D_R will include processor scheduling delays as the system throughput increases.

The CPU resource of the sequencer replica is a potential bottleneck as all messages must use it. To mitigate this, the assignment of message sequence ids can be a lightweight operation if a single process is managing all the operations in its local memory or a remote memory node such as memcache [102]. Note that RPC processing between replicas does not have to be serialized. Likewise, the RPC generated by S to any remote process may be parallelized too in case a remote memory node is used to store the unique *seqId*. The actual storage operations still have to be serialized and therefore D_S remains a factor in the overall scalability of the system.

To increase the fault tolerance level within a cluster a durable storage may be used to generate and store the *seqId*. In this case, D_S will involve the same overheads as D_R. Given such a tradeoff, most practical systems will prefer to keep the sequencer tolerant of any process or memory failure since such a failure will inevitably result in a system level failure due to the nature of single-point failure for the sequencer. Therefore, we should consider D_S as a significant potential bottleneck.

Unlike D_S, D_L includes delivering the message to the local commit layer which is serialized and may have noticeable delay in a range from 1ms to 10ms, as measured from real implementations. The serialized delivery is part of the responsibility of the replication layer which has to ensure the consistency of State Machine replication even though the underlying storage layer may be able to deliver messages in parallel without breaking the serializability requirement.

Given the above discussion, the overall delay (excluding X_C) for delivering a message on a local replica is calculated as:

$$T_L^{NU} = X + X + D_S + D_L \quad (4.1)$$

The sum X + X is a sum of independent random components, so it is not shown as 2*X. Because of host processor congestion, the host delays and thus T_L^{NU} will increase as λ

increases. Assuming that each replica runs on a single core whose entire workload is derived from handling these messages, a simple approximation from the queuing theory [62] is a time-expanded version of the processing demands, expanded by the factor 1/(1 - replica host utilization):

$$D_S = P_S/(1 - \lambda E\{P_S\}); D_L = P_L/(1 - \lambda E\{P_L\}) \quad (4.2)$$

For (4.2), we may conclude that the impact of the long-tailed distribution of D_S is rather limited because of the following two observations:

1. Under the non-uniform delivery the network propagation delay distribution F_X does not affect T_L^{NU} as the number of replicas increases or as λ increases. Note this is not the case for the uniform delivery due to the consensus algorithm, as discussed in the next section.
2. Since D_S, D_L only involve local queuing delays, their distributions do not depend on network condition changes between replicas.

When the hosts are not heavily congested T_L^{NU} will be mostly decided by X. The distribution of X, under normal conditions, will have a smaller variance than the exponential distribution but with a significant tail especially for traffic under a lower QoS class.

In Figure 4.3, we show how the client message rate λ affects the delivery response time based on the above model. Note that we will use the X distribution presented in the previous section to generate the result that shows the box plot range of the delivery response time under different λ and the bottleneck effects thereof.

- $f_x(x)$: the empirical US center-east distribution of network propagation delays between clusters, shown in Figure 4.1.
- $E\{P_S\}$: 5ms, mean processing time for P_S
- $E\{P_R\}$: 5ms, mean processing time for P_R

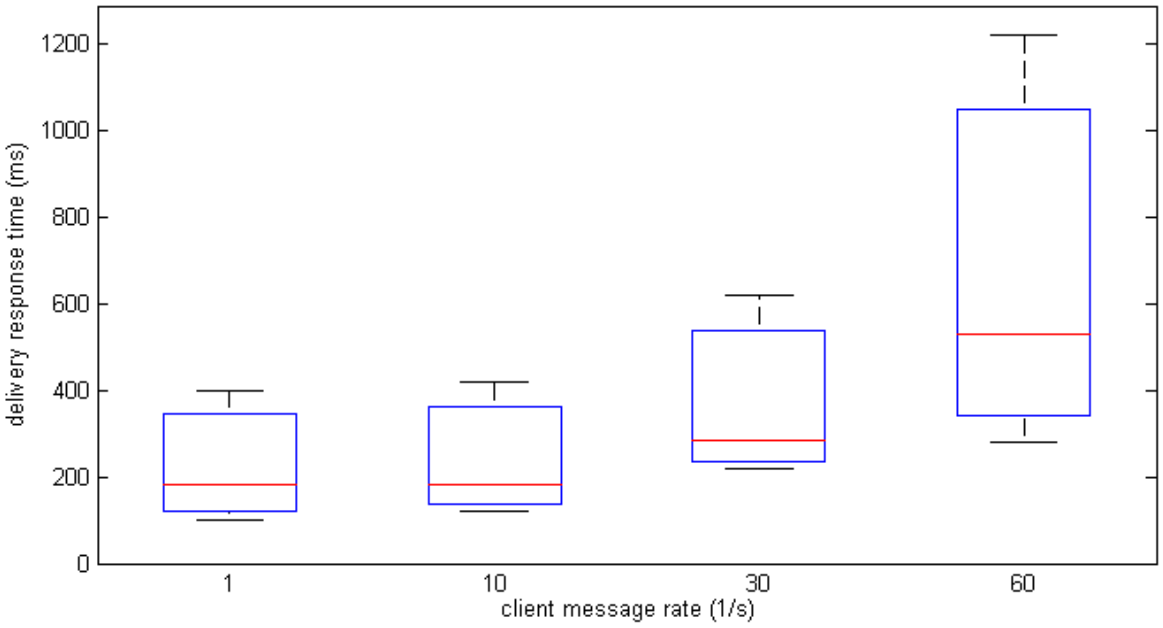


Figure 4.3: The non-uniform delivery response time for different client message rates shown as box plots with outliers removed.

In Figure 4.3 we see that, as the load increases, the increased mean and variance are contributed by the bottleneck on the sequencer.

4.4.3. Comparison to the strict model

The strict model involves running a typical consensus algorithm such as Paxos, over which a quorum of replicas have to acknowledge the delivery of the message. Under normal conditions when there is no replica or network failure, the core message scenario for these strict systems can be summarized as following:

1. *A* local replica L receives a client message *m* against a given *ReplicaState*.
2. L passes *m* to the master replica S, if L is not S.
3. S assigns a globally ordered *seqId* to *m*, and sends *seqId* to L and *m* along with *seqId* to all other remote replicas.

4. All non-master replicas, including L, acknowledge *seqId* to S, often after writing an on-disk log entry for the message.
5. After S receives the acknowledgement for *seqId* from a quorum of replicas, 1) S will deliver *m* under *seqId* locally; and 2) at the same time, the master replica S will send out the confirmation of delivery of *seqId* to all other replicas.
6. All the other replicas besides S, including L, deliver *m* under *seqId* once the confirmation from S has been received.

For simplicity, we assume the messages are delivered individually as they are received by the local replica. This ignores the possibility of message batching.

The above scenario describes a single message delivery. For our analysis, we assume messages will be delivered by an interleaved algorithm such as a multi-paxos algorithm, which allows replicas and the master to run interleaved Paxos instances to deliver newly received client messages without being blocked by the quorum acknowledgements of preceding messages. Such an assumption will make the quorum acknowledgement delay not subject to any extra queuing delay under load, unlike the case for processing delay:

- Messages are passed from one replica to another over a FIFO channel
- Under normal conditions, a single master will be managing all the message passing, and therefore it is not possible to see a message blocked on a local replica because one of preceding messages is still pending for a quorum acknowledgement.

For comparison to the non-uniform delivery, we are making the following two assumptions:

1. The processing of *m* and the assignment of *seqId* on the master are treated the same way as the sequencer operations, and give the same delay D_S for the non-uniform delivery.
2. The processing delay of the final *seqId* on every replica, based on the acknowledge from the master, involves the same delay as $D_{L.}$.

For simplicity (as elsewhere in this chapter), we also assume that the propagation delays and processing delays are homogeneous, that is all X_{ij} and all D_H have the same distribution. Based on the above analysis, for a non-master replica to deliver a message under normal conditions, the total delay of the critical path scenario is:

$$T_L^U = X + Q(q, N_R) + D_S + D_L + X$$

$$= X + OS(q, \{Ack_1,...,Ack_{NR}\}) + D_S + D_L + X, \quad (4.3)$$

in which $Ack_i = X + D_R + X$, and this is taken as the response time for the case of the uniform delivery. So the key difference between the two algorithms is

1. Non-uniform (NU): $T_L^{NU} = X + D_S + D_L + X$
2. Uniform (U): $T_L^U = X + Q(q, N_R) + D_S + D_L + X$

$$= X + OS(q, \{Ack_1,...,Ack_{NR}\}) + D_S + D_L + X,$$

It has been assumed that D_S , D_R and D_L have the same distribution, and therefore they are subject to the same effect due to system congestion under heavy loads.

The same is true of the delays Ack_i. Since, in this case, a lower bound on $OS(q, \{Ack_1,...,Ack_n\})$ is given by Ack_i, a lower bound on the additional response time for the uniform delivery is given by $Ack_i = (X + D_R + X)$. An even simpler (but looser) lower bound on the additional response time, which is independent of the system load, is (X + X).

When the system is under a light load, D_L or D_R may be negligible, and therefore the response time will be increased by at least X + X.

Now let us assume the quorum $q = n/2 + 1$, which usually requires 3 or 5 replicas based on fault tolerance and availability requirements.

Denote the distribution function for an acknowledgement delay Ack_i as $F_A(t)$. Since the q acknowledgements may be considered as n independent trials, each with a probability $F_A(t)$ of "succeeding" by time t, then the probability that exactly q out of n have succeeded up to time t is given by the binomial probability:

$$F_C(t) = [n!/[q!(n-q)!]] F_A(t)^q [1-F_A(t)]^{(n-q)} \quad (4.4)$$

Call this $F_C(t)$, the distribution function of the consensus delay. If the distributions are determined numerically, then the mean consensus delay can be found by differentiating to get the density and integrating.

Since Ack_i involves two propagation delays (X + X) and $D_{R,}$ the distribution $F_C(t)$ may have a significant long-tailed distribution, which will significantly increase the response time variance as observed by the client.

Also note that under a light load, when the master and the local replica are in the same cluster as an optimization of load distribution, the non-uniform delivery will incur a close to zero delivery response time due to the negligible intra-cluster response time. This optimization, however, will not bring the delivery response time for the uniform delivery to zero which will at least incur an X + X latency and will be subject to the variance of X.

For simplicity, we will not discuss the effect when X between different replicas have different distributions including their means. Our goal is to show how sensitive the response time will be to the network propagation delays.

The following charts show the predicted response time given some empirical data on X and D_R.

Overall, we note that the response time for the strict model is much more sensitive to the network propagation delay and the throughput than the non-uniform model. For the former, the number of replicas will be a main factor too. For the latter, the extra D_R introduced by each replica will also play a role in response time as the throughput increases.

- X, D_S, D_L: same inputs as the non-uniform delivery
- D_R : mean 5ms, with exponential distribution
- 5 replicas with a quorum of 3

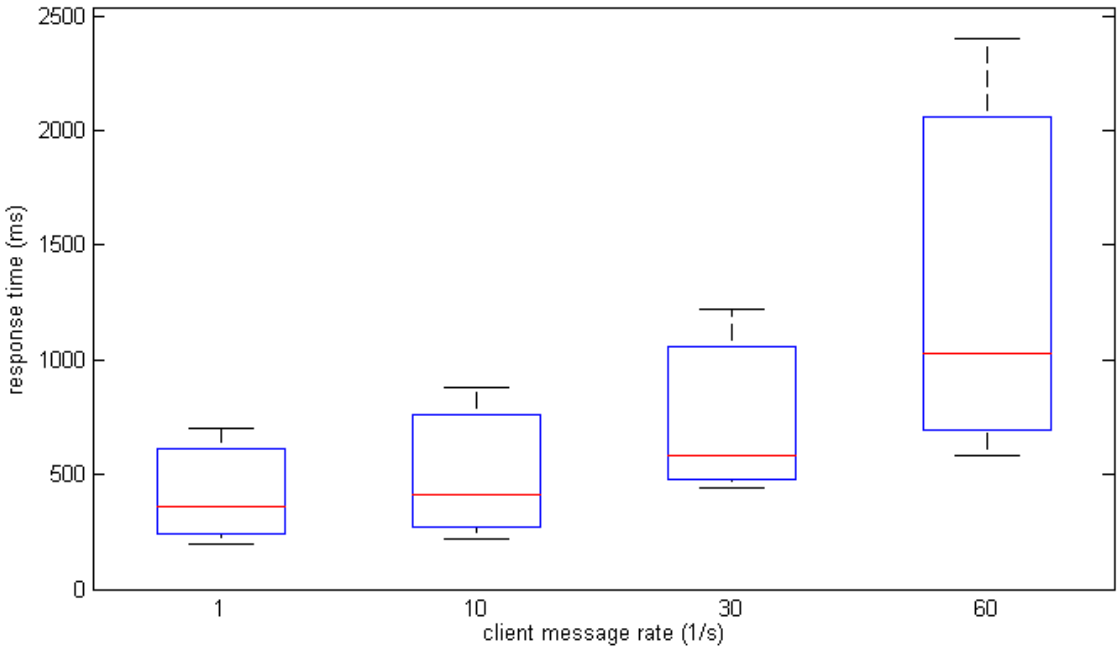


Figure 4.4: Under different client message rates, the strict algorithm delivery response time is shown as box plot charts, with outliers removed.

As in Figure 4.3, when the load increases, the increased mean and variance are contributed by the bottleneck on the sequencer.

As shown in the above chart, the median, 25th and 75th percentiles are higher for the uniform delivery than for the non-uniform delivery, by about 50% at low loads (1 message/sec), up to 100% at 60 messages/sec at L.

Since the bottleneck exists in both cases, and since one of the main sources of bottleneck is the local delivery, the improvement on the scalability of the system is limited with the non-uniform delivery.

4.4.4. Failure scenario

The consequence of the non-uniform delivery is that it is subject to a single-point sequencer failure, which may cause messages to be lost or reordered in respect to the global commit layer.

Although there are different implementation-specific conditions over which the non-uniform delivery will fail to deliver messages as reliably as the strict (uniform) delivery would, the most significant failure event is represented by the following scenario:

1. A message *m* is delivered by the sequencer S locally with *seqId*.
2. S fails before *m* is acknowledged by the global commit layer; i.e., before its order and presence in the global history have been committed. This results in the election of a new sequencer, which does not concern the present analysis.
3. Any server message generated on S after 1) is invalid.
4. One remote replica R receives *seqId* from S after 1) but before 2). As a result, any server message generated on R is also invalid.

For 2), we assume another replica will be elected as the new sequencer S', which will propose and commit a different message history in respect to *m* to the global commit layer.

From the above scenario a worst-case lower-bound can be derived for the inconsistency cost incurred by adopting the non-uniform delivery. In a real implementation, it is possible that *seqId* may be recovered by the remote replica R with the new sequencer if there is no conflicting message when S fails. We will show the practical results with the experiments in Chapter 6.

4.4.5. Inconsistency cost

The inconsistency cost will be measured by the probability of the failure scenario as defined above. The analysis uses the results of the non-failure steady-state analysis as inputs. For a given message *m*, and on any replica *i*:

1. $Q(q, N_R)$ gives the upper-bound of the extra acknowledgement delay introduced by the global commit layer, which runs a consensus algorithm.
2. X gives the upper-bound of propagation of an invalid *seqId* to a remote replica.

A vulnerability window is created, which is approximated by the period between the arrival of the message (upper-bounded by the delay X) and the determination of consensus (upper-bounded by the delay $Q(q, N_R)$). As illustrated in Figure 4.5, the duration of the vulnerability window is denoted as V:

$$V = \max(0, Q(q, N_R) - X) \quad (4.5)$$

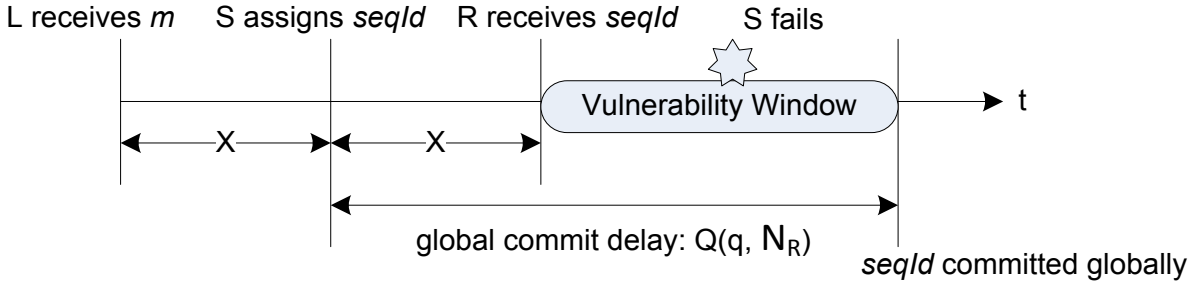


Figure 4.5: Illustration of the vulnerability window as an upper-bound time interval over which an invalid sequence id (*seqId*) may be delivered on a remote replica R.

The actual inconsistency cost is to be measured as the probability of client-visible failures, which will also take into account the server-message rate on any local replica. For our analysis, we assume the number of messages generated on any given replica represents the number of clients that are affected by invalid messages. In other words, the server message rate is measured as the number of clients that may have received new messages during the specified timer interval. Given the large number of clients that are connected to each local replica and the relatively short interval of the vulnerability window we will assume the server message rate follows the Poisson arrival with a given mean as below.

For the local replica that is also the sequencer S, the upper bound for the probability of an invalid message when a sequencer fails is decided as following:

- Vulnerable window and its distribution: $V = Q(q, N_R)$
- Server message rate: λ_S
- Failure rate: μ

- Number of invalidated clients during the vulnerable window: $V * \lambda_S$
- Rate of client failures: $\mu^* Q(q, N_R) * \lambda_S$

For a remote replica that is not the sequencer S, the above parameters are decided as:

- Vulnerable window and its distribution: $V = \max(0, Q(q, N_R) - X)$

On average, the probability that the system is in a vulnerability window is $\mu^* E\{V\}$. Suppose that each server message delivered during a window of vulnerability invalidates one client message. Then the rate of client failures is:

$$\text{Mean rate of client failures} = \lambda_S^* \text{Prob\{vulnerable\}} = \lambda_S^* \mu * E\{V\} \quad (4.6)$$

To compute the mean failure rate, a Monte Carlo approach was used. Random samples of the delays were generated to give samples of X and $Q(q, N_R)$, and thus of V, and the mean of V (denoted Mean{V}) was computed from the samples as an estimate of E{V}. Then

$$\text{Computed mean rate of client failures} = \mu^* \text{Mean\{V\}} * \lambda_S \quad (4.7)$$

We note here that this probability is regarded as an upper-bound, as implementations of the global commit layer may not receive any conflict message history from the new S'.

Figure 4.6 shows an example of such upper-bound under the measured distribution of X and $Q(q, N_R)$ as discussed in previous sections.

- Sequencer mean time to failure: $1/\mu = 48 * 3600$ (s), as a Poisson stream
- $N_R = 5$ replicas with a quorum of $q = 3$

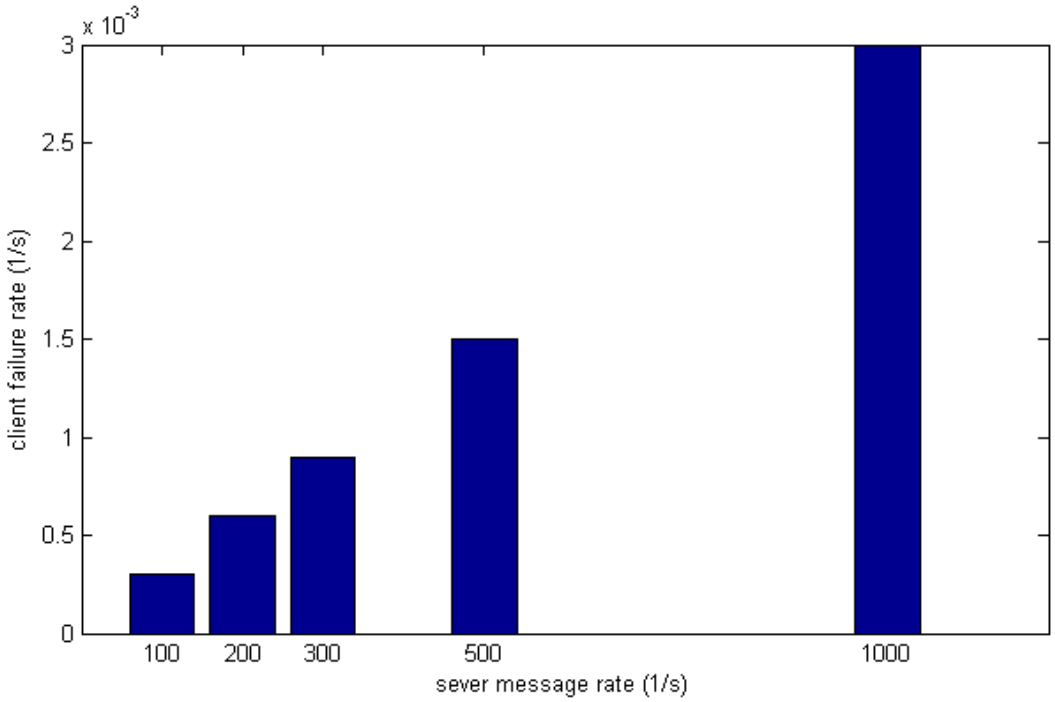


Figure 4.6: Client failure rates as the server message rate increases

Figure 4.6 shows how increased server message rates will increase the client failure rate in proportion, assuming one invalid server message will affect only one client during the vulnerable window.

From the above result, we note here that the client failure rate is merely a function of the server message rate; and the client message rate is not taken into account since the upper-bound result assumes the worst case; that is, when a sequencer fails there is always in-flight client messages that have yet to be delivered by the global commit layer.

4.4.6. Tunable parameters

As described in Section 3.4.2, tunable threshold parameters may be used to modify the consistency/performance tradeoff of COLOR. In effect, they reduce the inconsistency cost by reducing the vulnerability window. Since $Q(q, N_R)$ is controlled by the global commit layer, which is often provided as an external system and is difficult to modify, the simplest tunable parameters to introduce are delays at the client (e.g., at L) and at the sequencer, S.

- Client threshold: To reduce the probability of delivering an invalid message locally, an artificial delay with a maximum value $D_{th,L}$ may be introduced before a local replica delivers a message, after receiving the global *seqId* from the sequencer. This gives the local replica an opportunity to discard an invalid message should the global commit layer disagree with the non-uniform delivery of *seqId* of the sequencer.
- Sequencer threshold: Likewise, it is possible to delay server-messages that are generated as a result of the non-uniformed delivery. A delay up to a maximum value $D_{th,S}$ for server-messages will reduce the rate of propagating invalid server-side state to the client and therefore the overall inconsistency cost.

While both thresholds play the same role in the analysis, in practice it may be more effective to apply the $D_{th,L}$ at the local replica when a client message is committed. This is mostly an implementation concern because the deadline calculation for corresponding server messages will be more accurate at a local replica given the large variance of X.

Other optimization at a local replica may involve applying a longer threshold when the given replica is not generating any immediate server messages for its own clients or when there are no pending client messages from its own clients.

In either case, X will be increased to X' by $D_{th,L}$, or for S, an artificial X is created as X' = $D_{th,S}$.

The response time predictor corresponding to Eq (4.3) becomes:

$$T_L = X + D_R + X + D_{th,L} \tag{4.8}$$

and for the mean rate of client failures, Eq (4.6) is used with V defined by

$$V = \min (0, Q(q, N_R)) - X - D_{th,L}) \tag{4.9}$$

where L is not the sequencer, or $V = \min (0,Q(q, N_R)) - D_{th,S})$ where it is. Since the distribution of $Q(q, N_R)$ often has a long tail distribution which has a relatively small cumulative probability for long delays, the reduction of the vulnerability window will be very effective in reducing the probability of client inconsistency. The delay should have a small effect on the important

measures of the client response time (say, on its 90th percentile) but will considerably reduce the probability of invalid messages.

The following calculations demonstrate this fact. They used these parameter values:

- $D_{th,S}$ = 200ms;
- $D_{th,L}$ = 10ms, 20ms, 30ms, 50ms, 100ms
- λ_S (server message rate) = 100/s
- λ (client message rate) = 10/s

Figure 4.7 shows the reduced client failure rates under the same parameters as the previous section for the non-uniform bottleneck analysis represented by $D_{th,L}$ = 0.

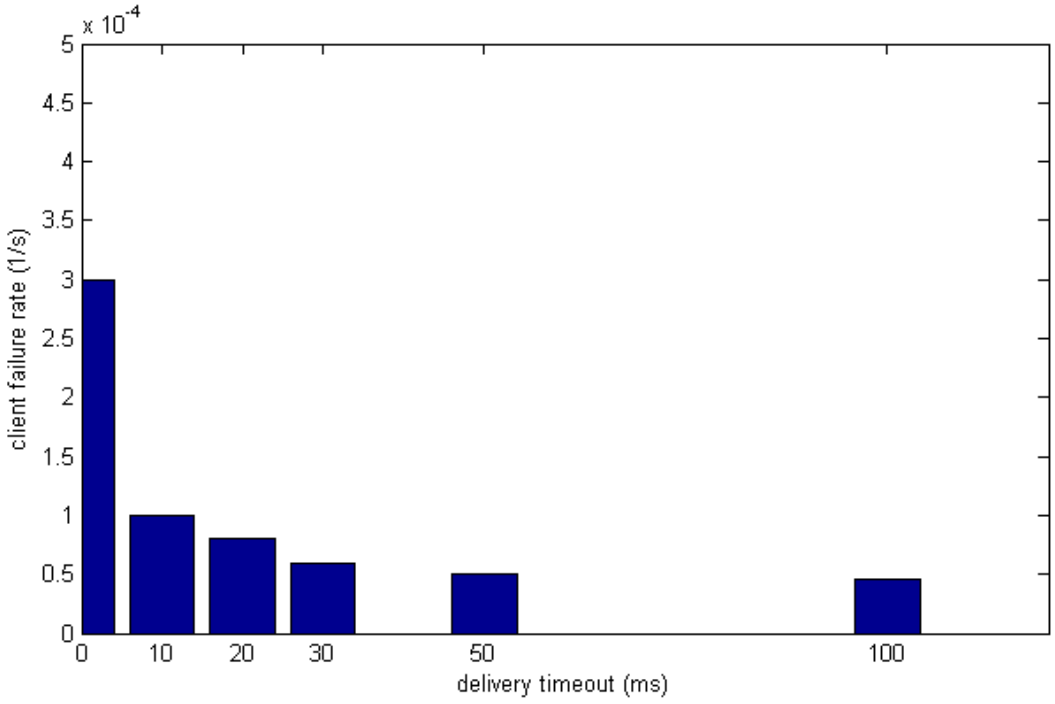


Figure 4.7: Client failure rates for different client delivery timeouts.

Even a small (10ms) delivery timeout (as client-threshold) will reduce the client failure rate significantly. Any further linear increase of client-threshold will have limited effect on reducing the client failure rate due to long tail distributions of other parameters such as the global commit response time.

Figure 4.8 shows the corresponding response time T_L. It is clear that the introduction of $D_{th,L}$ has a small effect on the 75th percentile of response time but significantly reduces the client failure rate. Only when the delay $D_{th,L}$ enters the central range of values of T_L, at 100 ms, is the response time effect noticeable.

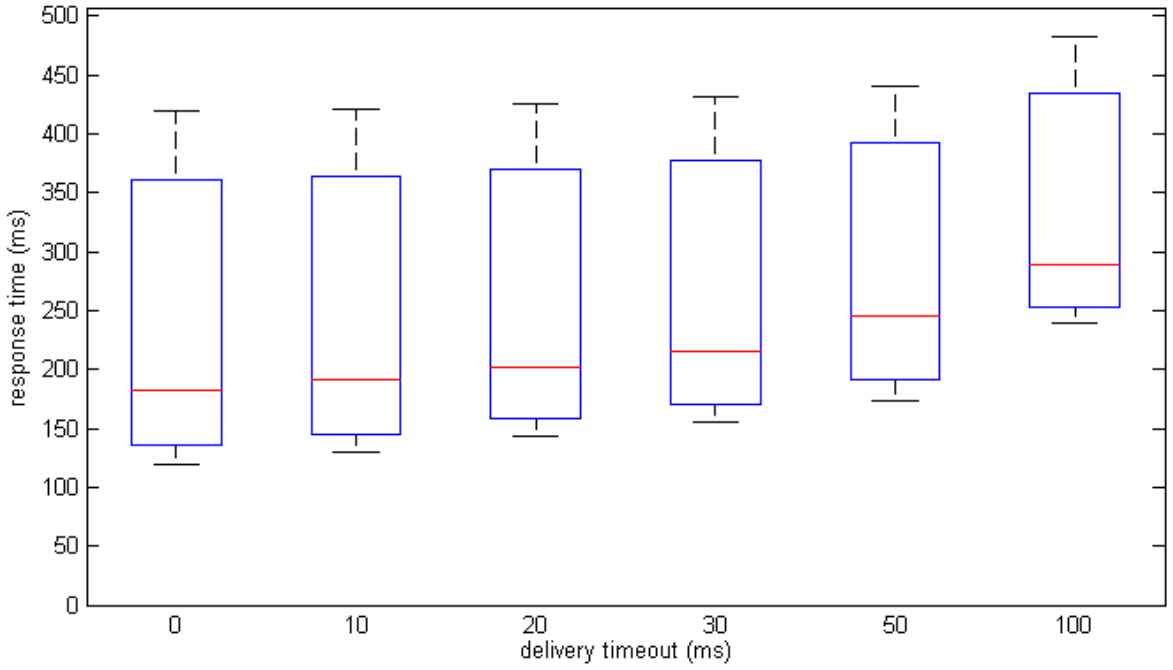


Figure 4.8: Client response time under the non-uniform delivery, as timeout varies.

Figure 4.8 shows that the client response time is increased under the non-uniform delivery by approximately the value of the delivery timeout, which is small relative to the overall delay. Thus, the performance cost of the timeout mechanism is small, while its impact on failure rate is considerable.

4.4.7. Comparison to the non-replicated model

One common optimization for non-uniform delivery is to have all client messages routed to the sequencer replica. This optimization will effectively eliminate any between-cluster

propagation delay from the response time, and as a result S will see the response time expression (4.1) be reduced to:

$$T_L^{NU} = T_S^{NU} = D_S + D_L \quad (4.10)$$

For the non-replicated model (NR), the response time on a local server may be calculated as:

$$T_L^{NR} = D_L \quad (4.11)$$

Therefore, the difference between the NU and NR model is merely D_S, which will be negligible under a light load, approaching P_L, compared to the client-to-server network propagation delay T_C.

As the load becomes higher, the queuing effect of D_S will start to become significant and the response time for NU will be increasingly larger than NR. In practice, because the NR model does not have to ensure serialized delivery, client messages may be delivered to the local commit layer in parallel (e.g., a local storage layer such as a RDBMS), whose local concurrency control will typically perform better than serialized delivery under higher throughputs. In other words, we will see a reduced D_L too for the NR model, and the bottleneck effect seen in NU will not be as significant as in NR.

The calculation of D_L for the NR model involves a detailed analysis of the concurrency control mechanism of the local commit layer. Further, the response time result also depends on the application data model. Therefore, we will not provide any numeric analysis for the response time difference between the NR and NU model.

Nevertheless, it offers important evaluation criteria for practitioners that the lower-bound response time under the NU model should match the NR model under a light throughput when clients are interacting with the sequencer directly.

4.5. Non-blocking failover

Non-blocking failover occurs when a new sequencer or master has to be re-elected due to either failure conditions such as network partitioning or reconfiguration events such as load redistribution across different clusters. This section addresses inconsistencies that may arise while executing the chosen failover strategy.

In a geographically distributed environment, the election of a new master or sequencer is a blocking process under either normal conditions or a failure condition such as network partitioning. Although (in theory) master election in an asynchronous environment is not guaranteed for liveness, practical solutions will often employ bounded clock difference to ensure the process will terminate eventually and as well as being timely. Under the COLOR model, we expect an external consensus layer will be responsible for the master election, e.g. via the global commit layer itself. Such a delegation largely removes the concern of liveness.

We also note that for the general analysis of the non-uniform delivery, managed failover such as reconfiguration due to load distribution events is not considered as failure events that may trigger inconsistency. This is because a coordinated failover is made possible in practice to allow the new sequencer to be elected when at the same time having the old sequencer continue delivering the globally ordered messages. The delay of the reelection is not a concern for modeling the inconsistency cost of the non-uniform delivery.

However, there will be times when an unbounded reelection delay becomes undesirable even for reconfiguration events, and a bound must be imposed. One example is a load redistribution event triggered by denial-of-service attacks. Therefore, in this section we will model both the failure events and reconfiguration events, and analyze the inconsistency cost when we have to enforce an upper bound for the failover process under such events.

4.5.1. Message scenario

When a sequencer fails or is suspected to have failed, the message delivery is blocked until a new sequencer is elected. A non-blocking failover will allow a local replica to decide a new sequencer based on a local decision as follows:

1. Replica L suspects the existing sequencer S has failed, and proposes a new election.
2. Replica L gets a candidate replica, which may be L itself, as the new sequencer S' and starts to wait for the acknowledgement from the replica manager.
3. After a timeout, which may be zero, L starts to communicate with the new sequencer S', which may or may not agree with taking the role of sequencer. If not, L returns to the previous state and tries to get another candidate replica, including L itself.
4. Eventually a new S is confirmed globally. If S' == S, the failover does not cause any extra inconsistency.
5. If S' != S, any message history delivered via S' may conflict with the message history delivered via S as both are proposing conflicting histories to the global commit layer.

Under reconfiguration events, the message scenario will stay largely the same except that before the timeout occurs L will still deliver messages via the old sequencer.

4.5.2. Comparison with the strict model

Under the strict model, the election of a new sequencer will include a delay of $Q(q, N_R)$ to reach a consensus on the new sequencer. The distribution of such a delay will often involve a non-negligible long-tail distribution, such as 90-percentile response time. After a failure, the whole system will be blocked by this delay. Under reconfiguration events, missing deadlines will happen when the consensus on the new sequencer takes longer than the timeout for the reconfiguration process to terminate.

In the case of an external master election, such as via the global commit layer, the above consensus delay will still apply, albeit at a different layer. Furthermore, the replicas have to be notified of the result of the master election, which often involves a global read too.

Overall, our focus here is on the lack of a timeliness guarantee for the failover to terminate. The actual delay is to be treated as a system parameter as such a delay is always implementation specific and is subject to network environment too.

The key difference between the two models is that the non-blocking failover involves a conflict window when the master or sequencer is unknown, while the non-uniform delivery involves a vulnerability window before the new sequencer is known.

4.5.3. Inconsistency cost

The key cause of inconsistency is from what we will call a “conflict window”, which specifies the interval during which multiple sequencers coexist. The duration V_C of the conflict window is

$V_C = \max(0, Q(q, N_R) - D_{th,F})$, in which $D_{th,F}$ denotes the timeout interval for failures, before the local replica resumes operation.

The probability of an invalid replica state depends on the client message rates across all the correct replicas that are involved in the failover to a new master or sequencer. To determine the upper bound of the inconsistency cost, we assume that each correct replica will take the role of the sequencer for itself and all the local messages will be delivered directly on each local replica without involving any remote sequencer. This is often the case when the failover is triggered by network partitioning events over which a remote statically assigned sequencer is not always accessible to local replicas.

The consistency cost will be defined as the overall rate of failure of client messages at a representative replica L. To calculate it the following parameters are defined:

- Vulnerable window = V_C, with mean $E\{V_C\}$. This window applies to the entire system.
- Number of correct replicas during the failure = p
- Failure rate of a sequencer = μ
- Server message rate on a local replica = λ_S

- Client message rate on a local replica = λ_C, and we assume client message load is evenly distributed across replicas during the failover.

The failure rate of client messages will be approximated as the rate of occurrence of the following pattern of events at a given replica L:

1. During a single conflict window, some other replica has one or more client messages, and replica L also has one or more client messages. This may cause the state of replica L to be invalid; it is (pessimistically) assumed that it does.

   The probability of one or more events from a Poisson stream of rate λ occurring during an interval of duration V_C is $1 - \exp(-\lambda * V_C)$. Since the client arrivals are Poisson processes with rate λ_C at replica L, and total rate (p-1)*λ_C at all the (p-1) other replicas, and the arrival streams are all independent,

   Prob { replica L is invalid for this window }

   $$= (1 - \exp(-\lambda c * V_C))*(1 - \exp(-(p-1)\lambda c * V_C)) \quad (4.10)$$

2. During this invalid window, each server message generated at L makes some client fail. Again, this is a pessimistic evaluation of the situation. The mean number of server messages generated at L during an interval of length V_C is $\lambda_S * V_C$, thus the mean number of client failures during one vulnerable window is

   mean client failures per window

   $$= (1 - \exp(-\lambda c * V_C))*(1 - \exp(-(p-1)\lambda c * V_C))*\lambda s * V_C \quad (4.11)$$

3. Considering there are p correct replicas and the rate of occurrence of vulnerable windows is μ/s,

   total mean rate of message failures for the system

   $$= (1 - \exp(-\lambda c * V_C))*(1 - \exp(-(p-1)\lambda c * V_C))*\lambda s * V_C * \mu * p \quad (4.12)$$

This analysis is pessimistic because it ignores batching of messages.

The state sharding model discussed in the next section makes the non-blocking failover much less vulnerable to conflicts in the absence of a global sequencer, by reducing the effective message rates for a given state shard.

Figure 4.9 shows how the client failure rate given by Eq (4.12) increases with the client message rate λ_C on each replica.

- timeout value $D_{th,F} = 0$, so $V_C = Q(q, N_R)$
- p the number of correct replicas during the failure : 5
- $N_R = 5$ replicas with a quorum of q = 3 for the global commit layer
- $\lambda_S = 100/s$
- $\mu = 1/(48 * 3600)$ failures/s

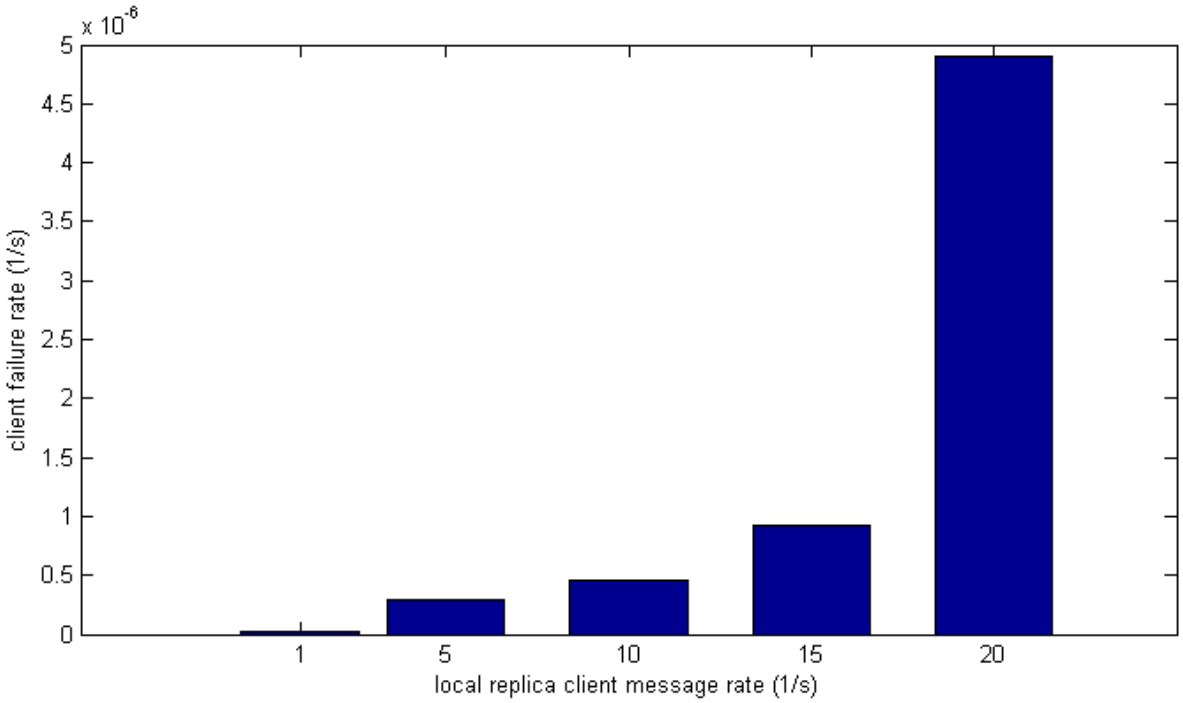


Figure 4.9: Client failure rate vs. client message rate for the uniform delivery.

Non-blocking failover may cause invalid replica state due to conflicting client messages to be delivered on local replicas during the conflict window. As the client message rate increases the probability of invalid replica state also increases, which results in increased client failure rates.

The failure rates are very low. For a client message rate of 30 at each replica, or 100/s overall, there is a failure rate of about 5×10^{-6}, or a failure probability of only 5×10^{-8} per message. Rates are low partly because of the low rate of replica failure assumed here, which is, however, felt to be realistic for real applications (a two-day mean time to failure).

4.5.4. Tunable parameters

The failover delivery timeout $D_{th,F}$ is a tunable parameter. By increasing the failover timeout, the conflict window will be reduced accordingly, which will in turn decrease the probability of invalid replica or client state. With the timeout,

$V_C = \max(0, Q(q, N_R) - D_{th,F})$

Similar to the case of the non-uniform delivery, even a small value of $D_{th,F}$ will reduce the failure rate noticeably. Due to the long-tail distribution of $Q(q, N_R)$, a large $D_{th,F}$ does not have a proportionate effect. The mathematical model can be used to determine the optimal $D_{th,F}$ as a compromise between the failure rate and response time.

Figure 4.10 demonstrates the effectiveness of $D_{th,F}$ for reducing the inconsistency cost, using a numerical solution of the mathematical model. The response time for these cases is approximated by the value for strict (U) delivery without failures (Eq. 4.3 and Figure 4.3), plus the timeout delay. The calculation used the values from the previous figure, with $D_{th,F}$ varying and:

- $\lambda_C = 10/s$

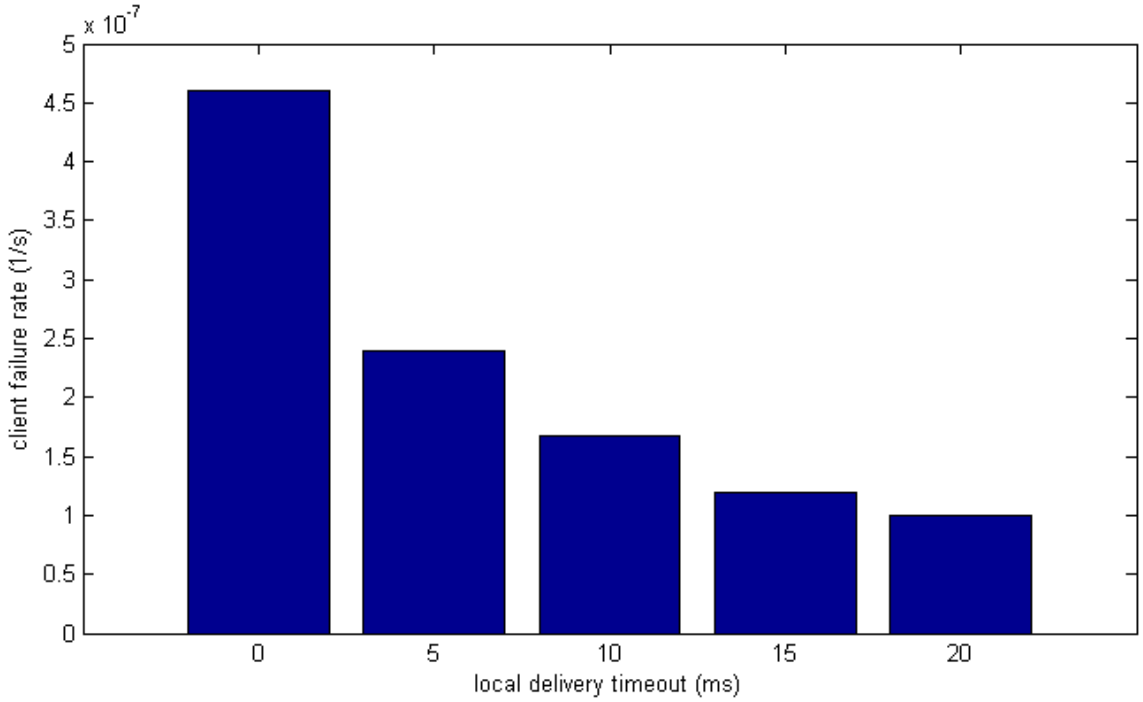


Figure 4.10: Client message failure rates as the delivery timeout increases.

Figure 10 shows that the non-blocking failover has considerably reduced client failure rates with a tunable local delivery timeout (as client threshold). A small timeout will see significant reduction of the client failure rate.

4.6. State-sharding delivery

The main value of state sharding is to reduce the bottleneck of replication algorithms for both the uniform and non-uniform delivery. Without state sharding, messages against the entire state space will have to be totally ordered and delivered in a single sequence across all the replicas. With state sharding there is one sequencer for each shard, and in the optimistic version these sequencers operate independently.

For the uniform delivery, the bottleneck critical path applies to the master in assigning the global order, the acknowledgement phase on each replica and the local delivery phase on

each replica. For the non-uniform delivery, the bottleneck critical path applies to the sequencer in assigning the global order and the local delivery phase on each replica.

In either case, sharding mitigates the bottleneck by allowing a configurable degree of parallelism, by partitioning the state space into a set of smaller state spaces. Under a strict consistency model, the global state space is often huge, for instance when application state associated with a single user or document is often related implicitly or explicitly and therefore individual state instances per a single user or document can rarely be treated as completely independent state spaces. When sharding is applied to the strict uniform delivery, the sharded delivery still involves executing a global consensus for each single message. Therefore the overall response time under a light load will be higher than for the non-uniform delivery.

Sharding also reduces the inconsistency cost under failure events when it is applied together with other optimistic algorithms such as the non-uniform delivery or non-blocking failover. Without sharding, a single failure of the sequencer could invalidate the entire state space at the same time on local replicas and therefore affect all the clients that may have interacted with local replicas during the failure event. Such a condition will especially be a concern for a global cloud service for which one would not want to have a single-point of failure that may affect all the active clients of the system.

Likewise for the non-blocking master failover, without sharding the rate of conflicts will be very high and any local delivery will almost always be forced to be rolled back due to conflicting client messages being delivered on other replicas. The reduced inconsistency costs under sharded delivery are achieved simply from reduced message rates against individual sharded state spaces. The combination of sharded delivery with other optimistic algorithms helps practical systems to function as the state space grows larger.

When it is possible to apply optimistic sharding and the non-uniform delivery at the same time, the present analysis will focus on the effect of sharding against the uniform delivery in

order to allow us to analyze the effect of the sharding as a tunable mechanism, independently of other tuning parameters.

The combination of optimistic sharding and the non-uniform delivery will be evaluated in the experiments in Chapter 6, which will demonstrate how optimistic sharding will affect both the bottlenecks and the inconsistency cost under the non-uniform delivery or the non-blocking failover.

4.6.1. Message scenario and bottleneck analysis

The basic message scenario is the same as a consensus algorithm that ensures the uniform delivery as described in Section 4.2.2. The unique behavior of state sharding is that a single client message is mapped to multiple sharded messages which are delivered independently of each other. Because each sharded *ReplicaState* runs its own consensus instance, masters on which different sharded messages are delivered may be located in different clusters too, and thus fail independently.

1. A client message *m* is received by a local replica L.
2. A set of sharded messages (message m_i for shard *i*) is generated on L and then delivered in parallel to the local commit layer via a consensus algorithm. Only the *u* shards which are touched by the message cause sharded messages to be generated.
3. As m_i is delivered on each local replica, including L, server messages may be generated which will propagate sharded *MessageHistory* to the client.
4. When the *u* parallel sharded messages $m_{1...u}$, all have consensus, the original client message *m* is delivered to the global commit layer, via the current master elected for the global state space. This step has a join delay for the *u* parallel paths [55].

As a note on sharded messages under the COLOR model, concurrency control or local transactions are considered external client-level state in the view of local replicas. Furthermore, under the state machine model, there is no notion of global transactions either. The optimistic

assumption in state sharding is that sharded state instances are not related in any way, whereas the atomic delivery of client messages at the global commit layer implicitly or explicitly reveals such relationships when conflicts are detected between client messages.

We will first consider the (optimistic) case in which only one shard is affected by each message. Then, the strict approach carries out the same update logic on this shard, as it does on a non-sharded state. Since a separate consensus is invoked for each sharded message, the bottleneck analysis for the uniform-delivery still applies. The response time for each message delivery is decided principally by the quorum delay $Q(q, N_R)$, which is the same as with the non-sharded delivery. The effect of sharding in mitigating the bottleneck is achieved by reducing the throughput for a single state instance by the number of shards (see the definition of effective sharding parameter *v* below), and thereby reducing the effect of D_R and D_L. In implementations when a CPU resource also causes a bottleneck, the sharding will also help eliminate the bottleneck with the system load being distributed across the number of shards (also see the parameter v below).

The granularity of shards does not have to be from a uniform partitioning of the global state. When some part of the global state space experiences a high probability of conflicts, then it should be partitioned into fewer shards, or even kept as a single shard. For instance, when a set of online documents have many links to each other, it will reduce the probability of conflicts if we choose not to partition this set of documents; e.g., documents with the same set of labels. Generally a hierarchical naming scheme is often employed to name individual shards, and it is always feasible to control the level of granularity at the instance level, that is to have different granularities assigned for different nodes of the naming tree.

For the rest of analysis, we introduce the notation of sharding level as:

- n: a tunable parameter that decides the number of shards per a single global state space
- v: the effective sharding parameter, which may be statically determined as n / (mean number of shards each client message updates)

Note the actual granularity of the *ShardedState* is not important, and for our analysis we assume each *ShardedState* will bear the same client or server message rates as independent Poisson streams, with the per-shard rate λv /s:

- $\lambda_V = \lambda/v$ = mean client messages per second from all clients for a single shard

With the above sharded throughput, compared to the original uniform delivery model (Eq. 4.3) the response time seen by individual shards will share the same expression as Eq. 4.3:

$$T_L^U = X + Q(q, N_R) + D_S + D_L + X$$

$$= X + OS(q, \{Ack_1,...,Ack_{NR}\}) + D_S + D_L + X, \quad (4.3)$$

However, the replica hosts D_H (including D_S, D_L, D_R) will see $\lambda_V = \lambda/v$ instead, compared to Eq. 4.2:

$$D_H = P_H/(1 - \lambda/vE\{P_H\}) \quad (4.13)$$

As a result, the host capacities are all increased by a factor of v, and bottlenecks are pushed to higher total client throughputs.

The following two figures show how *v* will affect the response time distribution in the case of bottleneck. Figure 4.11 repeats Figure 4.3, and is generated from the non-sharding uniform delivery, which shows how client message rates impact the bottleneck on the master and local commit layer. Figure 4.12 shows the response time with a sharding parameter v = 1000. The response time distribution remains roughly the same as the client message rate increases, even when it exceeds the saturation rate for the non-sharding case.

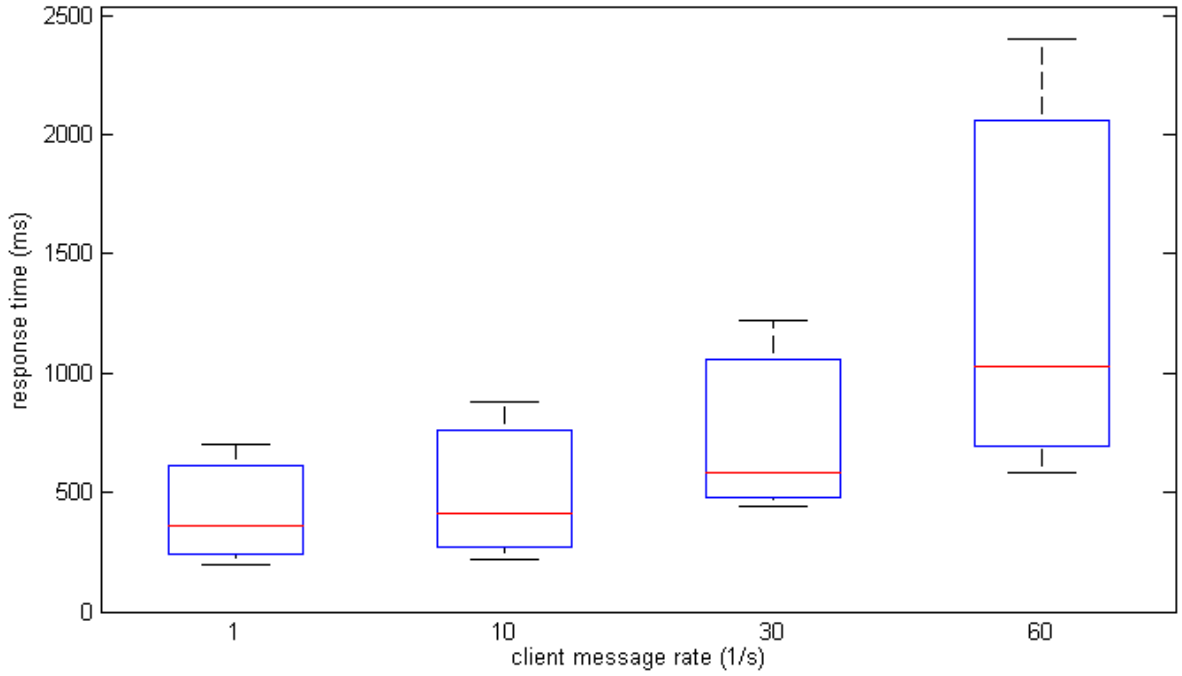


Figure 4.11: Response time with increasing message rates and no sharding.

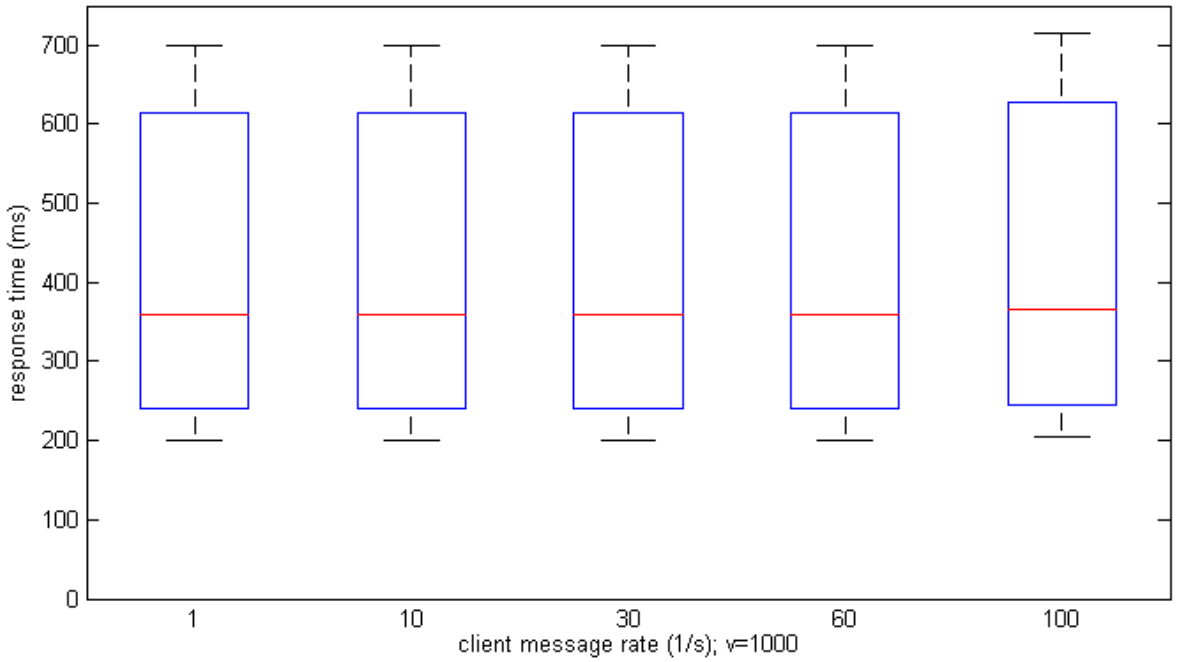


Figure 4.12: Response time with increasing message rates and the sharding parameter v = 1000.

With state sharding (v = 1000), the bottleneck effects of the uniformed delivery are largely eliminated under the same client message rates.

4.6.2. Comparison to the strict model

The strict model will rely on the global commit layer to deliver each client message over the entire global state space. As the state space grows and the arrival rate of client messages increases, the blocking delivery of the consensus algorithm will become the critical path of the replication system, which will limit the overall throughput. The potential bottlenecks due to processing at the replicas also contribute to the distribution of $Q(q, N_R)$ as we note.

As discussed in the previous section, the effect of reduced throughput against each serialized delivery operation will significantly reduce the bottleneck of the system under the strict consensus algorithms.

In this analysis, we are only calculating the response time of delivering a client message. Since sharded messages are delivered independently, the actual delivery of the original client message will be completed after all the sharded messages originated from the client message have been delivered. This so-called join delay [55] over parallel delivery of sharded messages will affect the response time as perceived by the client which may interact with the server via typical RPC style communication.

To evaluate the join delay, we will have the following parameters:

- u: the average number of shards a client message will update, and also v = n / u

We then assume each *ShardedState* will run its own independent consensus instance with identical parameters. Therefore, the join delay will be calculated as:

- Q(u, u): u-order independent but identical distributions of consensus delay

Figure 4.13 shows the impact of different values of u and how the join delay is different from the distribution of the local delivery against a single shard.

- λ_C = fixed client message rate: 10/s

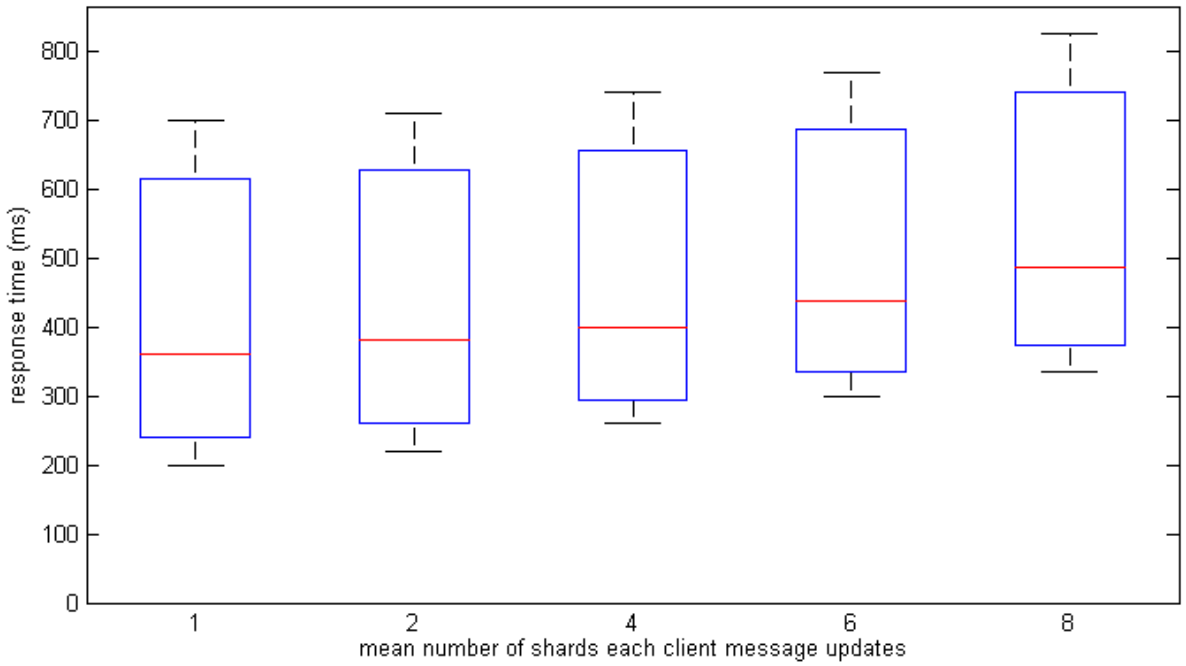


Figure 4.13: Response time as u, the number of shards accessed per client message, increases.

Since a client message may update multiple shards at the same time, the join delay to wait for all the shards to be updated will increase the response time accordingly. However, the effect is relatively small.

We also note that our assumption on independent distributions among different shards in delivering messages originated from the same client message is rather a simplification. In practice, sharded messages may be sent to their respective masters as a batch message. Usually, messages will only be batched for the first propagation delay X when they are sent to the masters that happen to be located in the same cluster. The analysis above is somewhat pessimistic, in ignoring the possible gain from batching some messages.

4.6.3. Failure scenario and inconsistency cost

With state sharding, there will be two parallel consensus executions for each client message.

1. Global commit layer, which will deliver the client message as an atomic unit.
2. Local commit layer, which will deliver a set of sharded messages in parallel with each other.

Generally, the global commit layer will be slower even under normal conditions as it will handle a much larger load as shown in previous analysis.

As client messages are delivered by the global commit layer, the global message history, represented as sharded *MessageHistory*, will be used to validate the local message history which is generated per each state shard. Whenever a local message history is invalidated, any affected clients will be forced to rollback too.

Unlike the non-uniform delivery, there is no timing based parameter. The interval between the local delivery and the global delivery makes no difference to the failure scenario. Conflicts are generated purely as the result of any conflicting client messages that contain overlapped sets of sharded messages. Because the global master is different from masters for individual shards, two client messages may conflict if they include updates to at least one common *ShardedState* instance.

In the simplest case, two client messages may generate a local delivery order that conflicts with the global commit layer if they both include just one sharded message that happens to update the same *ShardedState*. Under the sharded message delivery, the probability of conflicts between the local commit layer and the global commit layer are decided completely randomly by the runtime environment. As one example, one shard may have its master located in a cluster that is different from the global master. As a result, the initially proposed order of the two messages will differ between the two masters even though the two messages share the same local replica.

Based on the above observation, we have defined the following parameter for the analysis:

1. Pc: probability of conflicts between two client messages. This is largely a static parameter as decided by the application data model.
2. Tc: conflicting interval between two messages as they are received by the local replica. Messages arrived within Tc will generate invalid local message histories by the sharded delivery that are different from the global commit layer, as an upper bound. Tc is largely an environment parameter that is decided by the implementation and its runtime.

Under this model, we assume the global commit layer accepts messages directly from local replicas; while, in practice, the global commit layer will accept only message histories committed by masters of sharded state to respect the serialization order decided by each shard. This is important, as masters for different shards may actually be located in different clusters and as a result the global master will be vulnerable to a very large Tc due to variance of network propagation delays X between clusters. With such a conservative model, the only case that the global history will roll back the sharded history is when more than one common shards are involved in two conflicting messages as described in the sharded delivery algorithm. In any case, Pc will be mostly decided by the application interface and can be seen as a static design-time parameter.

Now assuming masters for all the shards are located in the same cluster, and then the Tc is largely a runtime parameter decided by the implementation. For our analysis, we choose an upper bound value for Tc to estimate the probability of invalid replica state.

Now with the above two parameters, we can estimate the actual conflict rate as following:

1. Given a Poisson arrival stream of client messages of rate λ, when a client message arrives, the probability that at least one preceding message falls into Tc is a function of λ, Tc, and Pc that will be calculated by a Monte Carlo method, and will be named P1.
2. The actual rate of messages that will generate invalid replica state due to conflicts is then P1 * λ.

3. Similar to the previous analysis, if we assume each invalid server message will invalidate one client, then the client failure rate will be P1 * λ * λs * Tq. Here Tq represents the global commit delay as an upper bound.

Figure 4.14 illustrates the relation of the above parameters with numeric solutions based on the Poisson arrival stream as the sole input.

- P_c conflicting probability under v=1000 is taken to be: 0.001, 0.002, 0.005, 0.01
- T_c conflicting window: 5ms
- λ client message rate: 100/s
- λ_s server message rate: 1000/s
- T_q the upper bound of global commit: 1000ms

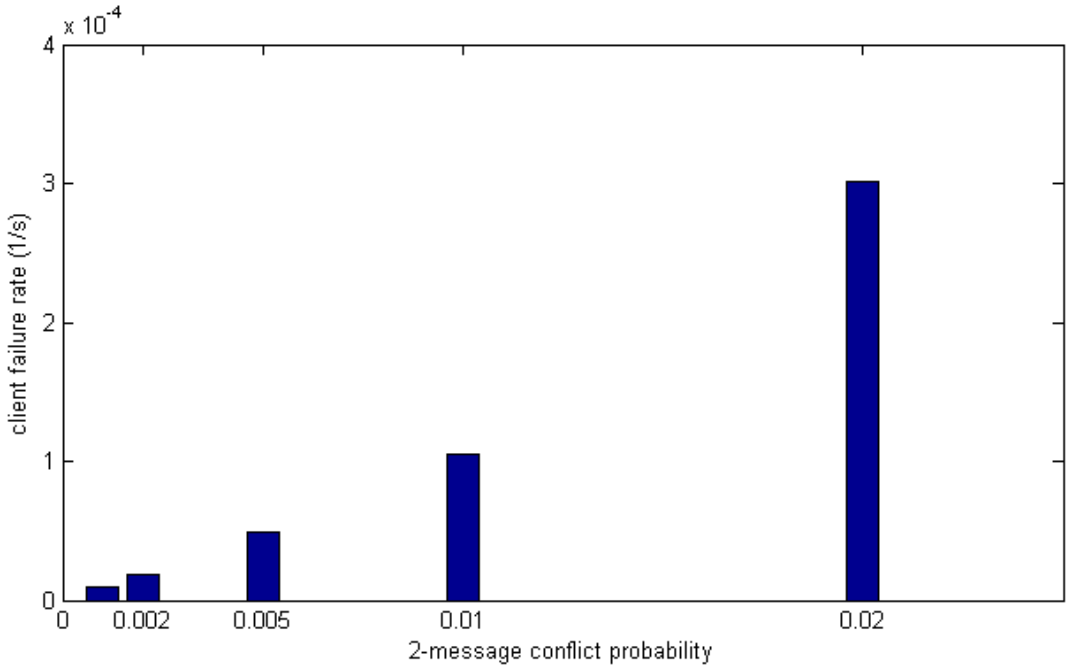


Figure 4.14: Client message failure rate as the probability of conflict increases.

As the static message conflict probability increases (possibly due, for one thing, to a decreased level of sharding), sharded delivery will see a higher client failure rate due to conflicting local message histories.

4.6.4. Tunable parameters

The sharding itself serves as the most important tunable parameter in deciding the inconsistency cost. State-sharding is also a nearly contiguous parameter as the level of sharding is tunable against the entire state space.

By changing the level of sharding, we effectively change Pc, the conflict probability. Since the application access to sharded states is not necessarily uniform, we will use Pc as the tunable parameter for the analysis.

When the number of shards changes, Tc may actually change as well but such a change will be mostly caused by the runtime environment and we will assume it remains as a constant parameter for our analysis.

We also note that similar to database systems when transactions are managed either per tables or per rows [4], over-sharding is in practice undesired either due to the resources required to manage parallel consensus, the logging overhead as well as message overhead due to the need to break a single client message into many sharded messages.

In practice, we will further use the following strategies to manage Pc:

- Scope of client messages: the larger each shard is, the less sharded messages each client message will include, which will further reduce the probability of conflicts between messages.
- Overlapping of client messages: how frequently each client message may overlap with a parallel client message in terms of the state space the client message touches. If the probability of such overlap is high, independently of the message rate, more coarse-grained sharding will reduce the probability of conflicts.

The above input parameters are generally static parameters that can be evaluated at the design time to decide the optimal level of sharding based on the tradeoff analysis between performance and inconsistency cost.

Note the client message rate also plays the role of deciding the actual inconsistency cost. The higher the arrival rate, the higher probability to see conflicting client messages generate sharded message histories that may not be agreed by the global history.

The following two charts show how the tradeoff works as the number of shards increases. The response time improves at the higher client message rates but the client failure rate also increases accordingly. We use the same parameters as in the previous section which has a large client message rate that easily triggers the bottleneck for the non-sharding case.

As discussed earlier, the number of shards will decide directly the conflict probability Pc, and for our analysis, we assume their relation is strictly a linear one. This will generate the same chart as the previous chart, as shown in Figure 4.15.

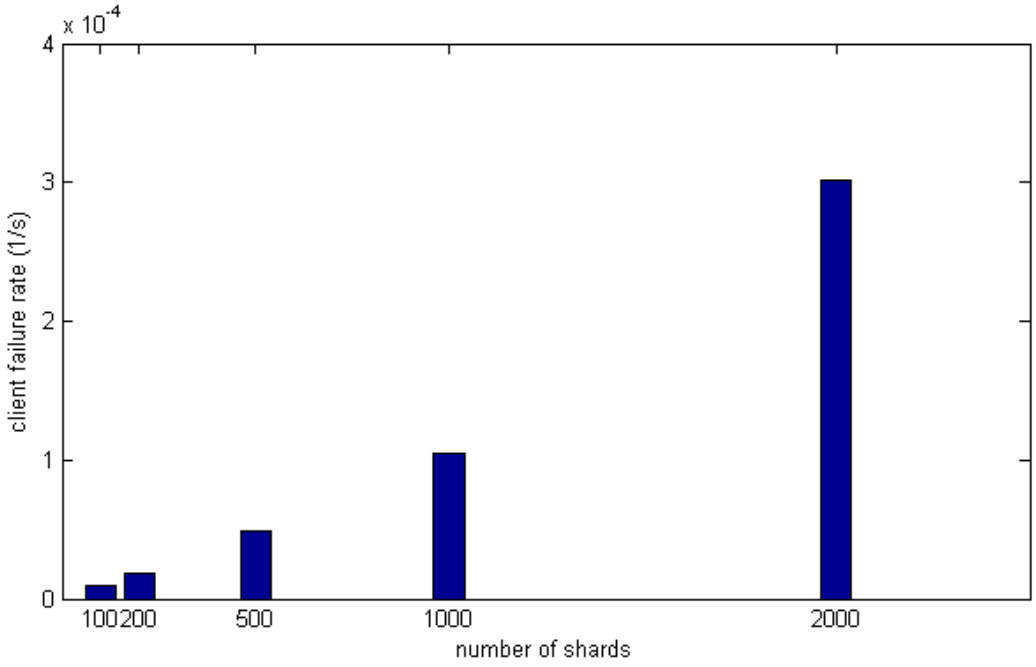


Figure 4.15: Client message failure rates as the number of shards increases.

The increased number of shards (effectively as v) will increase the client failure rate due to the increased message conflict probability.

We then use the number of shards to generate the response time distribution based on the same input parameters as in the uniform delivery case. Figure 4.16 shows the number of shards plays a significant role to reduce the bottleneck and response time of the uniform delivery under a high load. When the number of shards becomes very large, the effect of client message rates on response time is approaching zero.

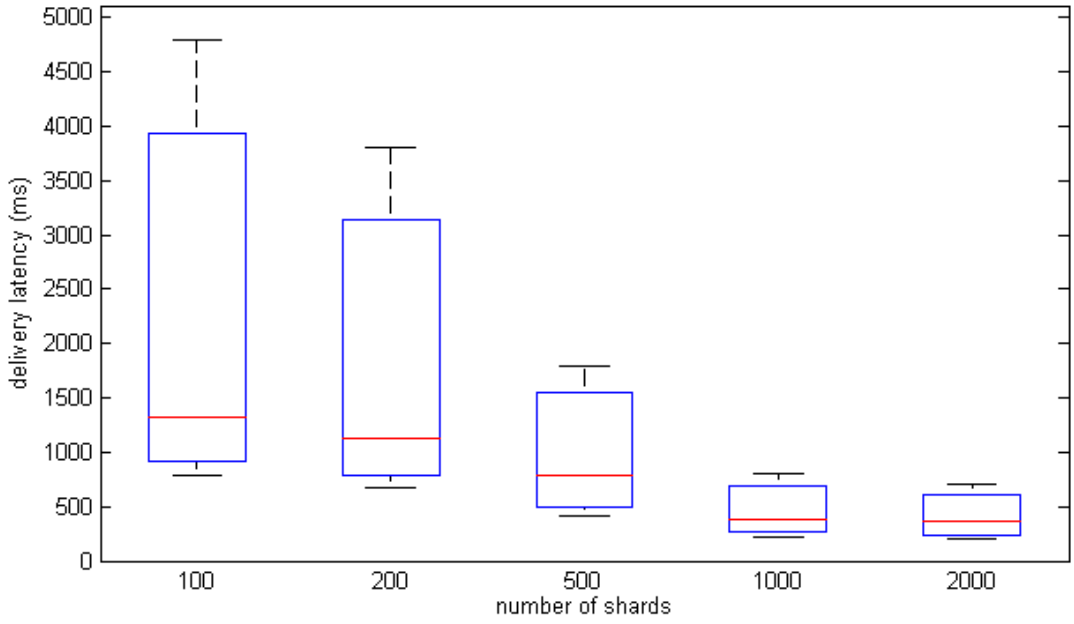


Figure 4.16: Client response time as the number of shards increases.

The increased number shards (effectively as v) will reduce the bottleneck effects of the uniform delivery due to reduced load against each shard. Under a high load, the response time will decrease as the number of shards increases.

In going from 100 to 1000 shards, the mean response time drops by a factor of about three, but the variance drops by a much larger factor of about six. The response is not only faster, but more stable. The cost in increased failures over the same range is a factor of about ten, however the rate is still quite small, about one client message in a million.

Chapter 5 Implementation of COLOR

The COLOR model is implemented as a framework to allow for extensions required to support different external solutions such as different consensus algorithms for the global commit layer or different optimistic algorithms for the local commit layer. The overall solution is implemented with a layered architecture, following the layered system model and interface definition.

The implementation is for the cloud environment over which Google operates its production services. Details specific to Google's internal environment are abstracted away, and we believe that the framework can be easily ported to another public or private cloud environment or to be integrated with related open-source solutions, such as Hadoop [66], which in many ways were initially inspired by published work from Google such as [30].

The client to server interface follows the basic REST principles and all communication is done over HTTP, even for asynchronous or one-way communication. The client-side platform is assumed to be a browser, and the implementation relies on the HTTP cookie to propagate the message histories as context data.

Although client-side recovery solutions are implemented and evaluated, the framework leaves the client-side solution mostly open, which will be further explored in Chapter 7.

The road-map of this chapter is as follows. It begins with the description of the high-level architecture and its layered structure. It then describes each layer in detail along with concrete interface definitions between layers. Lastly, it describes a general-purpose demonstration application that has been created to demonstrate the core concepts of the COLOR solution as well to serve as the test bed for the performance evaluation, which will be discussed in Chapter 6.

5.1. Architecture

The architecture of the COLOR framework adopts multiple layers of abstraction in order to build a solution that may work in different cloud environments and on top of different underlying dependencies such as the consensus provider or the global load balancer.

As shown in Figure 5.1, the overall architecture can be represented by the following layers of abstraction:

1. Global state
2. Local replica state
3. Server-side session state (optional)
4. Client-side state

As messages are passed through the above four layers, they will be intercepted by the COLOR framework in order to apply the optimistic replication algorithms. Layers have strict one-way dependency, that is, a lower layer will not depend on any upper layer, for instance, in the form of callback APIs.

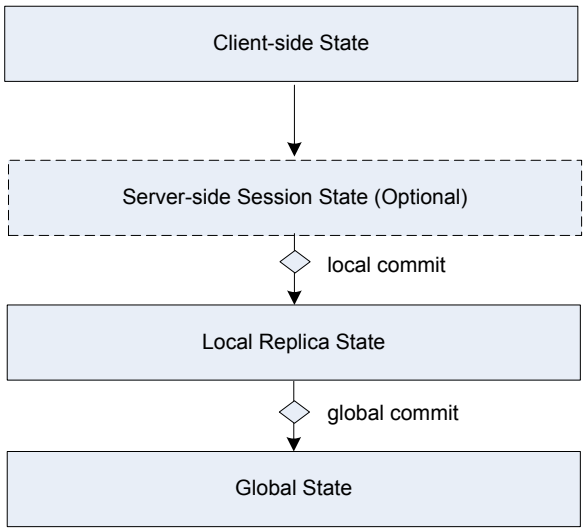


Figure 5.1: Layers of abstraction of the COLOR framework.

5.1.1. Core components

As shown in Figure 5.2, at the core of the framework is the representation of the replica state. The COLOR framework models and supports application-level state in the form of a replicated state machine. The replica state is managed by a replication manager which maintains the algorithm-level state to control the replication as well as the local commit interface. The implementation of the replication manager may depend on other subsystems such as the consensus layer for membership management.

The replication manager also serves as the bridge to the global commit interface, which is supported by an external layer of dependency, such as a global database that offers strict consistency. Mapping between the replica state and the global state, for instance to enable sharding support, is to be handled by the replica state model. The global commit layer will handle the physical mapping between the replica state model and the storage model as necessary.

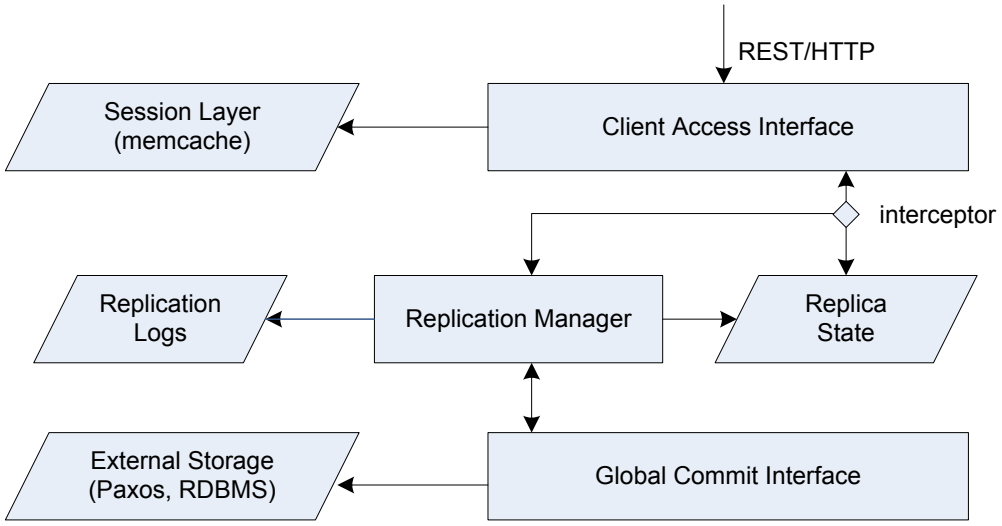


Figure 5.2: COLOR Framework architecture.

The client-access interface manages the generation of client messages and server messages. As discussed in the system model of Chapter 3, a session layer is often expected to

match the state model with the client-side object model against which messages are to be exchanged between the client and server.

How clients are bound to the local replica server is not a direct concern. In a typical cloud environment a global load balancer takes the responsibility for both load balancing and load distribution. The former aims to improve the cluster utilization while the latter tries to move the application data and logic closer to where clients are. Denial of Service (DoS) detection is also another form of load balancing.

Failure detection and failover are to be managed by the replication manager too, with the help of an external subsystem that will notify the replication manager of replica failures.

5.1.2. Framework requirements

COLOR is implemented as an extensible framework to meet the following requirements:

1. Pluggable dependency on external systems that offer standard consensus, failure detection and client interaction modules.
2. Extensible modules to allow internal extensibility over a given layer.
3. Clear APIs for both northbound and southbound interfaces for each layer.

The first requirement is to ensure that COLOR does not have to re-implement any existing solution from scratch, such as a consensus layer based on Paxos. On the other hand, the core of the COLOR solution does not overlap with any existing optimistic solution, and therefore there is a clear boundary on what needs be implemented as part of COLOR versus what should be provided via existing external systems.

The second requirement involves horizontal extension of any given layer, such as the local commit layer which includes multiple internal modules and may be replaced independently with either richer functionality or higher efficiency.

The last requirement follows the standard layered abstraction model to allow for easier vertical extension of COLOR layers. For each layer there will always be two sets of APIs. The

so-called “northbound” interface serves as the public API for other layers in COLOR, while the southbound interfaces allow external systems to be adapted to serve as the dependency of a given COLOR layer, as shown in Figure 5.3.

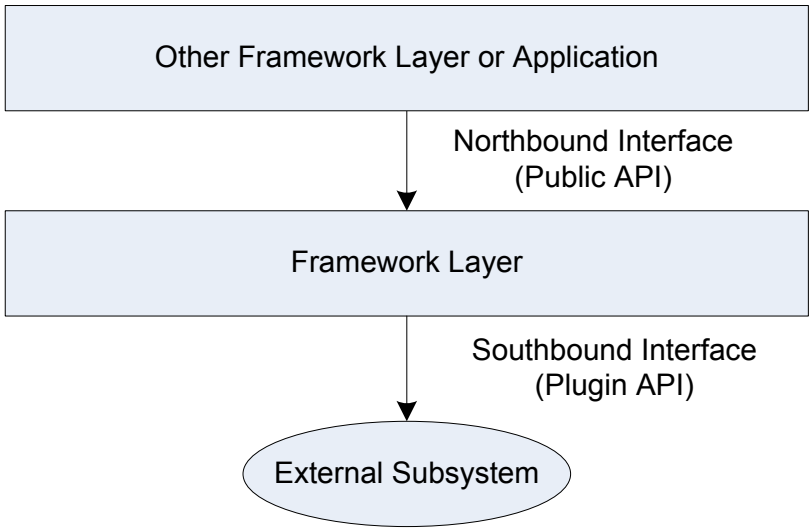


Figure 5.3: API model of the layered COLOR architecture.

With the above requirements, the overall solution is to be offered as a white box which will enable intensive instrumentation and profiling for the purpose of evaluation.

5.1.3. API requirements

Across the framework, different layers share the same meta-form of the API, which uses HTTP as the underlying communication interface with both client and server streaming enabled [158]. HTTP offers several advantages compared to defining some proprietary RPC or streaming protocols:

1. Minimized semantic mapping between internal and external interfaces.
2. Preserve all RPC properties while at the same time allowing streaming in both directions.
3. Extensible metadata as headers.
4. Support cache, session and cookie (context) as part of the protocol.
5. Language independent.

The API is defined in such a way that makes the local or remote interface transparent to clients. We choose a Java interface as the Interface Definition Language (IDL), which will be applied for both local binding (in-process API calls) and HTTP binding (remote API calls). Streaming is enabled by annotations, as in the following example:

```
public interface ReplicaMonitor {
   @streaming(RESPONSE)
   Collection<StatusUpdate> monitorHealth();
}
```

When streaming is enabled for request or response, we use hop-by-hop HTTP chunks as the message framing.

The API follows the basic REST concept in its HTTP binding; e.g., in how HTTP methods are used. The binding is done manually since the number of APIs is limited in the COLOR framework.

The exactly-once semantics is not supported when an API is invoked via the HTTP binding. The API interfaces are designed in such a way that all calls may fail due to runtime conditions, expressed as Exceptions. In some cases, failures may trigger algorithm-level failure events; e.g., a failover event. Such a condition is always handled by the client and is therefore not a direct concern of the API provider itself.

5.1.4. External dependencies

Under the COLOR framework, standard functions are provided by external subsystems as pluggable modules. Each module essentially wraps the underlying external dependency, such as a consensus system, by implementing the southbound API of the pluggable module in question.

For the current COLOR solution, its external dependencies include:

1. Global commit layer backed by a consensus algorithm.

2. Membership service, often backed by a consensus layer.
3. Local replica storage.
4. Session service.

The above dependencies will be discussed in more detail in later sections as we describe the implementation of each module.

The current COLOR framework implementation does not rely on external failure detectors, whose role is to notify the system of suspected failures. It is known that failure detectors are less effective in a geographically distributed system due to network partitioning. That is, out-of-band keep-alive based failure detection is less effective for a geographically distributed system as it is hard to distinguish between network failures and process failures. The other concern is the integration overhead, which will complicate the solution since failure detection may break the layering abstraction.

Instead, in the COLOR framework any remote invocation between two modules may trigger a failure event, which will be reported to the replication manager. A local decision will be made in terms of the actual failure condition, such as a suspected sequencer failure.

Figure 5.4 shows the internal modules of the *ReplicationManager*, as defined with Figure 5.2. The dependency on the external consensus subsystem is required for managing the membership, e.g. election of a new master or sequencer in the event of master or sequencer failures.

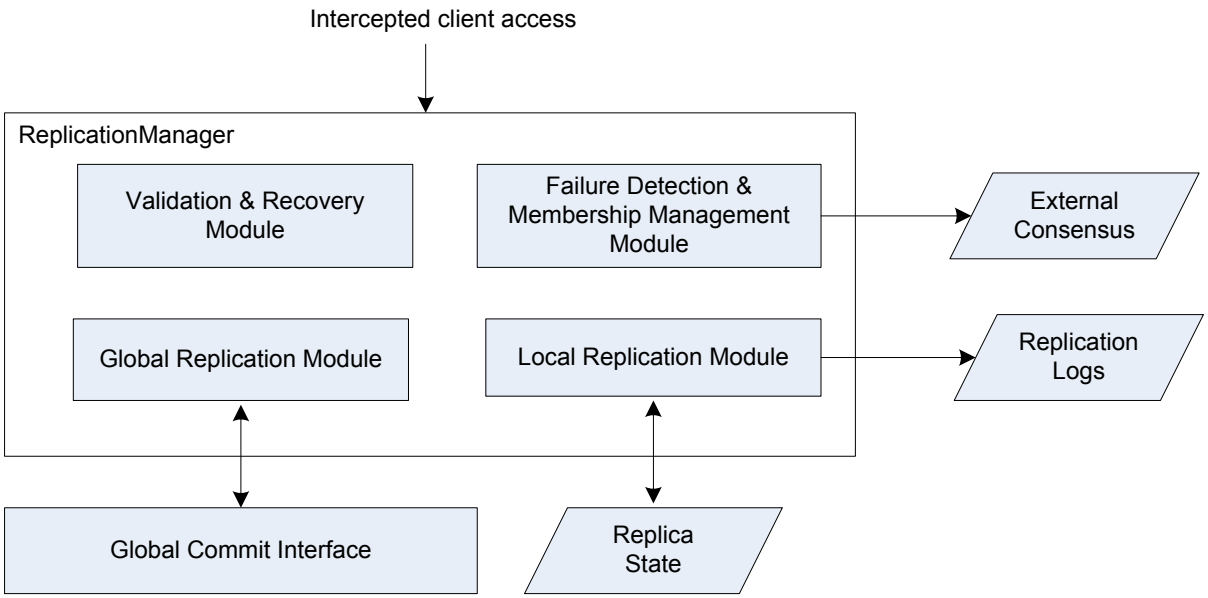


Figure 5.4: Internal modules of *ReplicationManager* and its external dependencies.

5.2. Replica state representation

Replica state represents the target state model that clients are accessing. In the COLOR solution, replica state is represented by nested key-value maps. The model is generic as the entire state can be uniquely addressed with a list of map keys, also known as the state key, to create an arbitrary level of granularity of replica state instances. Such a model also makes state sharding completely transparent to the object model.

Replica state is accessed as if there is a local in-memory database for the process that handles the client requests. The physical storage of replica state is not a concern of the COLOR solution and the local persistent storage is one of the extensible modules.

5.2.1. API specification

The replica state supports REST based operations as following:

- ReplicaState#getState(StateKey key) : Object
- ReplicaState#createState(StateKey key, Object value) : void

- ReplicaState#updateState(StateKey key, Object value) : void
- ReplicaState#deleteState(StateKey key) : void

Logically, replica state is stored as a nested map, and each state key is represented as a String-valued key, formatted as URI paths. A single replica state instance is associated with a single state key, and the state instance is to be treated as an atomic value in the above operations.

Under the above API, the state representation of the local replica may be partitioned at any level, as decided by the state key that the application supplies:

- StateKey#StateKey(String[] keys)
- StateKey#getKeys() : String[]
- StateKey#getPath() : String

For each key, the application may also specify an optional sub-key to enable partial updates to the same replica instance:

- StateKey#StateKey(String[] keys, int subKeyIndex)
- StateKey#getSubKeys() : String[]

Using the concept of sub-key, each of the REST operations may carry only a sub-value:

- Partial insert: #createState(StateKey, Object) : void
- Partial update: #updateState(StateKey, Object) : void
- Partial get: #getState(StateKey) : Object
- Partial delete: #getState(StateKey) : void

The replication system relies on the *StateKey* for generating the message history. However, the above design will make the replica state model transparent to data partitioning.

5.2.2. Design specification

Replica state representation may have different formats depending on how the state is accessed. For external access, different HTTP MIME types may be applied, such as JSON or

XML. Internal access within the server-side cluster requires a language-independent binary or text format. To ease the mapping between internal and external accesses, the implementation adopts the same MIME type for both internal and external accesses. In-memory representation is language specific, and uses regular Java types to represent the state value object.

As discussed earlier, both external and internal accesses rely on HTTP as the base messaging protocol. Therefore, all the APIs support the HTTP binding in addition to their local access interfaces.

To minimize data marshaling overhead, the state is initially stored in its serialized format in local memory. State instances are only deserialized into the local memory representation at the first time when they are accessed by application code.

Advanced queries and full-text indexing may be supported too as a custom layer on top of the local state replica layer. Since reads are served locally, query support is completely a local concern. However, the scope of history is different for a general query than a key-value based map lookup. We will discuss the scope of queries in Section 5.3.

5.2.3. Cloud environment

The physical storage of the local replica needs be made transparent to the API. In a typical cloud environment, the state may be stored in various ways, such as memcache like servers or in a Bigtable like non-SQL storage, or in a full-scale storage layer with its own memory cache. In any case, the access API to the replica state is to be provided as a logical in-process access layer. In the COLOR model, we name these processes as the local replica state processes, although they may be completely stateless with the actual state stored in a remote storage layer.

For efficiency reasons, software that needs to access the local replica state may be a co-process, in which case the local API will be used instead of the remote HTTP based API. We do not expect the co-process or the local access layer to cache any state for optimization

reasons, mainly due to the complicated nature of maintaining cache-coherence and concurrency control, which often requires the local access layer be elected a single master for a given state instance key.

The underlying storage will be responsible for sharding the entire replica state space, which is addressable with the key URI. For the COLOR framework, the default choice of replica state is the memcache, which allows a direct mapping of the replica state model to the actual storage model.

5.3. Local commit layer

The main role of the local commit layer is to manage the delivery of client messages and therefore the generation of local message history. The secondary role of the local commit layer is to decide what message history should be propagated back to the client with each server message.

The local commit layer is the core component of the COLOR framework, and the optimistic algorithms are implemented in this layer.

5.3.1. API specification

The local commit layer and its API is offered as an interceptor layer over the local replica layer. Every client message or server message will be handled by the commit layer following the interceptor pattern [155] [156]. Specifically the commit layer will generate and maintain the local message history for each client message. The state of these histories is stored in the local replica storage.

For each server message, message histories are propagated as context objects associated with their state keys. The HTTP binding will change the context objects to HTTP cookie headers for browser clients or HTTP headers (X-State-History) for non-browser clients.

Since optimistic algorithms are pluggable, the internal interactions between the local commit layer and any external components are not a direct concern of this API specification. For instance when a sequencer is used, each client message will generate a remote invocation to a sequencer to get the message id.

5.3.2. Design specification

The local replica API supports both a synchronous RPC-style interface and an asynchronous one-way client message and server notification interface. For the former, the interceptor will block the RPC as part of the local commit execution. For the latter, the interceptor will be non-blocking. To support both stateful communication, such as a socket like interface, and stateless communication, such as a UDP like interface, the interceptor should function as a stateless interceptor in handling asynchronous client or server messages. Such a design choice relieves the commit layer from maintaining any session state, which will also have to be supported by the local replica API.

As discussed in the system model, query or read-only client messages are not involved in the COLOR algorithm. However, the interceptor layer has to process each query message too in order to decide the scope of message history for any resulting server messages. An example is querying for existence or nonexistence for a given key. In either case, the entire parent scope of the queried key is issued a “read” message.

To enable query support, the query layer has to be made aware of the entire scope of the data to be queried and the scope has to be sent back to the local commit layer as context data along with the server message that represents the query results.

- QueryContext#getScope() : URI

Based on the discussion in Section 4.5, a reverse-sharding scheme may be put into place when queries are dominant in client-server interactions. With a lower sharding parameter v, the conflict probability Pc will be limited for queries that involve a large state space.

Furthermore, it will be very inefficient to propagate message histories of all the individual state keys as part of one query operation. Obviously such a tradeoff will compromise the purpose of sharded message delivery as states are now grouped into more coarse-grained instances.

Note that for supporting queries, the absence or presence of a key has to be modeled too. In other words, each parent URI space itself is a state of its own, similar to a file system directory that contains individual files. Due to the complexity and overhead of maintaining such state as well as the volume of message histories required, query operations may be supported against a snapshot of valid replica state to offload data intensive functions such as reporting or analyzing logs.

5.3.3. Cloud environment

The local commit layer may use a standard sequencer provider such as Chubby [27] in a typical cloud environment. For the COLOR framework, a simple in-memory sequencer was implemented that uses memcache as the cluster-local storage which supports atomic counter operations. For managing the local view of membership, it is possible to use a standard consensus maker such as a Paxos layer as the underlying implementation. The evaluation application uses megastore [10] as the storage layer for the membership, which also involves a master election. Failover state and local master decisions are stored in the local replica storage, so that the interceptor itself is stateless. Similar to the replica access layer, we store no cache for any local cluster-level state.

Generation and propagation of the local message history are distributed operations across multiple processes where local replica state is accessed. Therefore, concurrency control has to be managed carefully. The COLOR framework relies on the storage layer such as memcache or megastore to serialize updates to any local state that is generated by the local commit layer.

For client messages, in the absence of a global sequencer for the non-uniformed delivery, the local replica layer has to manage the concurrency control, similar to how internal state to the COLOR algorithm is to be managed.

For server messages, the interceptor will always pass back the latest message history for given keys as part of the message. The framework relies on the recovery layer or client-side logic to suppress conflicting message histories. In general, there is no synchronization between the state change and the local message history version, as the two may be managed by two different storage providers and any concurrency control between the two will effectively require two-phase commit support between multiple data storage providers, which is not always possible in a cloud environment that adopts heterogeneous solutions.

As stated in the correctness proofs sketched for the COLOR algorithm, the guarantee is that the message history version has to be newer than the actual state version against which a server message has been generated. This guarantee is enforced in the framework by the following sequence of events:

1. A client message is delivered to the local message history first, with a tentative state that indicates that the replica state is yet to be updated.
2. The local replica state is updated.
3. The local message history is cleared of the tentative state, which may block any new message history to be appended for the same state key.
4. Any server message that is generated as a result of #2 will see at least a message history that includes #1.

Since message history changes and local replica state updates are serialized, in the case where step #2 or #3 fails due to the local commit layer failure after #1, the system is able to recover and reach a state that is consistent between the message history version and replica version.

5.4. Global commit and recovery layer

The global commit layer is implemented as another interceptor layer over the replica state access interface. As soon as a client message passes the local commit layer, its message history will be sent to the global commit layer in order to generate the global message history. The actual global commit layer is provided by an external component, such as a Paxos layer or a centralized database.

The global message history may invalidate the local replica history as new messages are delivered to the global message history. The invalidation will roll back both the local message history and the current replica state. Such a rollback may further invalidate clients with an older message history version generated from the local commit layer at an earlier time.

5.4.1. Interface specification

The global commit layer and its API are also offered as an interceptor layer over the local replica layer. The global commit layer is invoked independently after the local commit layer. As soon as the local message history is generated, the client message will be delivered to the global message history.

The actual delivery of a client message to the global message history and any invalidation that follows are invoked in parallel with the local replica operations.

The global commit layer will not affect the server messages, which will always pass back the latest local message history for each state key.

Upon receiving each client message history, a validation step will be invoked as another interceptor before the local commit layer interceptor is invoked. The validation will enforce the core COLOR algorithm to detect any invalid client state and trigger a client state recovery immediately, when an invalid client message is detected.

The response to an invalid client message will have its own HTTP binding, including the HTTP status code and related header to pass back the latest message history that is known to be valid. An invalid message will not be sent to the local commit layer.

Recovery of invalid local message history and replica state involves rollback of the local message history version as well as the replica state version, back to what is known to be committed globally at the last invocation of the validation layer for a given state key.

5.4.2. Design specification

The implementation of the global commit layer is provided by an external subsystem. A few requirements exist for the recovery process which manages the outputs of the global commit layer.

First, concurrency control for the validation phase does not involve any linearizability requirement [69] between the global message history version and the local message history version. The guarantee of the validation phase is that an invalid client will eventually be invalidated, although the design goal is to make such a process as timely as possible. To reduce the response time, the validation may be throttled when the client message rate is too high. The validation phase is synchronous, that is the invalidation response to the client will be generated after the validation phase.

Second, rollback of the local message history and the local replica state have to follow the same process as applying a client message via the local commit layer.

Lastly, delivery of a client message to the global commit layer is completely asynchronous to the rest of COLOR algorithm.

5.4.3. Cloud environment

Both the global message history and global state are delivered by an external consensus based system or a transactional database system. Some solutions may offer client-side caching

and invalidation notification support. However, due to the resource overhead, it is unlikely the local replica will have access to the global message history or state in the same cluster. Therefore, in the COLOR framework client-side validation is done as a background process, to aggregate all the validation requests into a timer-based validation process. This will greatly reduce the response time as perceived by the client.

The commit of the global message history version and the global state must always be done as an atomic transaction. This is generally not a problem for the global commit layer as long as the message history and the global state share the same storage model. In the case when this is not possible; for instance, when the global state has its own storage model which is not part of the COLOR based solution, the COLOR commit layer will adopt the following solution:

1. Commit the global message history first, with the actual message too.
2. Apply messages out of the global message history to the global replica state in a separate process, with the total order decided by the global message history.

The above scheme works with a consensus layer that guarantees exactly-once delivery of a message. Such a guarantee is common in typical solutions for a global commit layer, such as megastore or RDBMS.

5.5. Client interface and session layer

This is the layer that interacts with the external clients as specified in the COLOR system model. The basic COLOR framework assumes a direct HTTP binding for accessing the local replica state by a client. At the same time we also assume a session layer will be present to support server-side logic or per-session state as maintained on the local replica. Remember that session logic and state are treated as client-side concerns under the COLOR system model.

5.5.1. API specification

The client-server interface follows the REST principle and relies on HTTP binding to access the local replica state interface. The message history of the local replica is propagated as context data via cookies or HTTP headers. An invalidation message is issued via a standard HTTP error response status with updated message histories as context data via cookies or HTTP headers.

For efficiency reasons, both partial operations and batching operations are supported. While the former complies with the URI based state key naming scheme, the latter requires encoding of multiple state keys into the URI parameters.

Client-initiated messages that are not generating any state updates are ignored in the COLOR algorithms. However, the client interface needs support for the same read or query access interfaces.

Server-initiated messages are generated either as the result of RPC requests, or as spontaneous server-push messages, which may involve special HTTP binding such as cometd [34] or WebSocket [148].

As part of the COLOR framework, the author is in the process of submitting an I-D draft to standardize the HTTP status and headers, which may be seen as an extension to the E-Tag header.

5.5.2. Design specification

The present COLOR framework is limited to single-client sessions, and the basic consistency requirement for the client-session interface is the so-called session consistency [147], which is trivially enforced when the session state is not replicated.

As stated in the COLOR system model, session state may or may not be replicated as part of the global state. For session state that is not part of the global state, the session layer will effectively hold the client-side state too. When the session layer plays the role of client, the interface between the actual client and the session layer is unspecified by the COLOR

framework. However, in most cases, the existence of a session layer is mostly for optimization purposes, and the client-server interface will remain the same as if the session layer were transparent to the client-side state model. Such a design is often desired such that the client session state does not need be reliably replicated as the global state.

5.5.3. Cloud environment

The COLOR framework relies on standard solutions to provide the front-end layer that allows external clients to access the data in the cloud. A typical web-tier, such as a Servlet Container [71], allows the COLOR framework to be plugged in to enable external HTTP binding, which may involve authentication, authorization or logging requirements, compared to internal HTTP binding which merely uses HTTP as the underlying transport.

When a session is enabled, the framework relies on standard HTTP sessions to manage the session-level state, which uses either cookie or a URI parameter as the session id. The cloud infrastructure has to support standard session based routing, which allows messages to be routed to the same process or cluster where the session state is located. Session state may be stored in-process or in remote storage such as memcache, and the routing is not a concern of the COLOR framework itself.

When a session failover happens; for example, when the process or the cluster that owns the session state fails, the client has to establish a new session. This process itself is not a concern to the COLOR framework components.

Applications may not lose any cached client-side replica state due to a session failover. In case a client has to restart during the session failover, we consider a new client is brought up after a new session is created.

5.6. Demo application - whiteboard service

As part of the COLOR framework, an example application is implemented that serves as a test bed for verification of both the design and implementation. It is a whiteboard with a basic HTML5 canvas-based UI on which the user may use a mouse or touchpad to paint colored and sized lines, as on a real whiteboard. The whiteboard is shared by multiple clients and any drawing by one client will be multicast to all the other whiteboard clients in real time.

The whiteboard application represents a canonical model of shared state machine in the following ways:

1. The state of the whiteboard is shared by multiple clients
2. Painting on a whiteboard generates a set of client messages that have to be totally ordered by the server in order for all clients to see the same whiteboard image in real time.
3. When the whiteboard state is replicated, all the client messages have to be globally ordered for all clients to have a consistent view of the whiteboard.
4. The state of the whiteboard is modeled in an obvious way under the COLOR *ReplicatedState* model, that is, the whiteboard canvas contains a nested map of images.
5. The addressable whiteboard state space can be as small as a pixel, and the whole state space of a replicated whiteboard instance can be easily and arbitrarily partitioned using a URI as the naming scheme.

The whiteboard implementation uses all the layers of the COLOR framework. However, we note that the server-side application logic has little to do with the whiteboard application itself. Therefore, the demo application provides general-purpose COLOR infrastructure with which many similar online collaborative applications may be created, simply by designing a different UI or providing a different state model.

5.6.1. Design specification

The whiteboard application represents one of most typical uses for replicated shared data in the cloud environment that is real-time online collaborative applications. These applications have strong real-time performance requirements, for both steady-state performance and failover performance.

1. A large amount of real-time input data, such as keystrokes or touchpad events, needs be streamed to the server efficiently.
2. The server-side logic has to be finished in a timely manner in order to multicast ordered messages back to all parties, again in a streaming fashion.
3. When a cluster fails over, users should see no pause in their interaction with the server. To meet such a goal, consistent replication has to be done in the background, which is what the COLOR solution tries to provide.
4. Clients are generally able to tolerate a rollback; e.g., see changes made locally being rolled back, as long as such events are very rare or with a bounded probability.
5. The response time requirements or real-time deadlines are tunable based on different application use cases. For instance, a user-visible pause may be limited to 100ms. This represents an excellent motivation for tunable performance and consistency.

To meet the first requirement, the whiteboard has a bidirectional messaging channel which the browser client to send and receive messages much more efficiently. The messaging channel is compatible with the HTTP semantics and uses SPDY [132] as the underlying transport when it is available. The messaging channel supports standard HTTP based headers, including cookies, as optional metadata for each client or server message.

All the channels will be terminated by the web-tier server processes in a local cluster to which messages for a given client are routed. Client messages are delivered against a local replica designated for the cluster where the channel is terminated. The server side logic, after a message is delivered by the COLOR framework on the local replica, will be echoed back in a message to the client to confirm the delivery of messages generated by the client itself. We

point out here that the need for local rendering is very common to most collaborative applications, which is another form of optimistic delivery, commonly known as optimistic concurrency control or client-caching. The local replica will also offer a push based interface which will allow the server-side logic to register for messages from other clients. Any client messages that are delivered either on a remote replica or on the local replica will generate server messages for the client channel, similar to well-known Publish-Subscribe based messaging systems [45].

When the channel has to fail over to a different cluster, the session layer logic supports transparent failover. For a planned failover, a new channel will be established and then messages will be drained from the old channel, which will be closed after the handoff finishes. Similar to the problem of achieving exactly-once RPC, the session failover may risk message loss in rare cases, independently of the COLOR framework, and the current implementation relies on the client-side retry to recover client or server message loss during the session failover. In the absence of any replication-related invalidation condition, the session layer logic will detect any message gap of server notification messages and re-transmit lost messages in order to update the client-side state.

In other words, the whiteboard does not maintain strict session replication to mask unplanned failover events such as server or cluster failures. This is largely a design choice to avoid complicating the evaluation of the COLOR framework.

The client-side rollback involves taking a new snapshot from the current replica state and then resuming applying delivered messages from remote clients or from its own echo. As we will discuss in Chapter 7, a simple three-way merge algorithm allows the client-side implementation to re-apply any local changes should such a rollback happen. This algorithm is adopted to support local rendering too.

Lastly, although we may choose to buffer client-side messages or to relax the need for real-time requirements since user perceived response time is often limited to 100ms, we choose

instead to create an implementation that allows us to evaluate the full spectrum of the tunable space enabled by COLOR algorithms. The basic implementation does not buffer any data and as such the implementation may be transformed to any other application models for which any reduction in response time is important, such as online trading or media processing.

5.6.2. Cloud environment

The demo application is implemented and deployed in Google's cloud environment. Since it is an experimental application, it was put under a sandbox URI namespace to avoid DoS and other security constraints, which may complicate the evaluation process.

The implementation use only open-sourced or published technologies as dependency layers, such as the Paxos layer and membership management. Although the web tier is specifically implemented to deploy in Google's production environment, the session layer design of the COLOR framework is generic enough to be deployed in other public cloud environments.

The whiteboard service has been deployed with replicas on five different clusters across the globe, in order to evaluate the performance of COLOR. This level of replication is rather common for globally accessed shared services. The implementation is also very cautious about the resource cost, especially for cross-cluster network transmission.

5.7. Background of the COLOR architecture

The COLOR framework is in many ways inspired by previous works on layered middleware architectures such as those based on CORBA or JAVA RMI [48] [104] [107] [161] [12] [16] [94], although the actual solution that COLOR adopts is completely different than traditional middleware based replication solutions.

Many previous solutions have also successfully adopted a layered architecture [154] [13] [14] [40] [5] [7], for the same reason that the COLOR framework chooses to have a layered architecture, which is to allow for pluggable modules.

The layered architecture also plays an important role in performance measurement as it allows the COLOR framework to instrument different layers to install failure injectors or to fine-tune processing demands.

Lastly the so-called interceptor pattern enables a very flexible way to implement and deploy different optimistic algorithms on top of existing replication or commit solutions. As part of the thesis work, the author has published a related framework [155][156], whose core idea is also seen in many similar projects [106] [108] [45] [31] that involve distribute object replication.

5.8. Lessons learnt

The author has spent over 6 years building the original open-source framework that supports replicated RPC [155], and the latest COLOR framework that enables an overlay solution on top of existing strict replication solutions.

Overall it is very important for a framework to get the layers of abstraction right, including both the API design and the runtime abstraction. The runtime environment is always complicated and hard to predict. Given the nature of distributed algorithms, it is very important to decide each individual layer or module under a well specified abstraction of its containing environment. Such a requirement is also crucial to reason about the correctness of the implementation.

Failure detection is another challenge that every distributed system has to face. Luckily for the COLOR framework we rely on an external system to manage the result of failures in terms of membership and master election. The actual detection of failure is somehow non-deterministic and passive as we rely on message passing to detect replica failures. However, this is acceptable as the prototype aims to prove the effectiveness of the COLOR solution in any cloud environment, and a dedicated failure detector will make the framework tailored to the choice of the failure detector. Coincidently, failure detectors have never been widely used in the cloud environment in which the evaluation is conducted.

Chapter 6 Evaluation of Performance and Consistency Tradeoffs

Experiments on the COLOR framework and case study system described in Chapter 5 were carried out with two objectives. The first is to verify the mathematical predictions of the models described in the Chapter 4. The second is to show the effectiveness of the COLOR solution in a real cloud environment with a real application context.

We believe both goals are important. The first goal allows us to gain insights on how to tune the specific algorithmic or environmental parameters to manage the tradeoffs between system performance and the consistency requirements. The second goal is even more important to demonstrate the COLOR solution in the real world. Mathematical models abstract from the actual runtime execution environment, which is often affected by many engineering level variables, such as those decided by the implementation details. The experimental results demonstrate the effectiveness of the COLOR solution as well as the soundness of the analytic models.

To make the evaluation useful, the measurements must address the critical path of the system behavior. This requires us to design the evaluation in such a way that the system performance will be affected directly by those key parameters defined in the analytic model. When this condition does not entirely hold, the performance evaluation will be mostly implementation dependent, and therefore will be much less effective.

6.1. Environment

The whiteboard application is the main test bed for the evaluation. As we noted earlier, the whiteboard application is rather generic on the server side. The evaluation environment includes the following layers:

1. Clients of a whiteboard application with instrumentation.
2. A session layer that serves as the front end to the clients.
3. A local replica layer that hosts the shared state machine for the clients. This is essentially the core COLOR framework. Server-side instrumentation is provided from this layer.
4. A global commit layer that provides Paxos based storage. The Paxos layer also allows us to measure the performance of the uniform delivery.

6.1.1. Whiteboard application

The web-based whiteboard client application is designed to be run from the Chrome browser, mostly in order to access the browser's internal measurement APIs. The whiteboard application uses a fixed-sized canvas to generate the client message traffic to the server. A robot is created on each emulated client to produce client messages; i.e., drawing actions, at a constant rate in order to measure the mean response time and other metrics at steady state.

The whiteboard session layer has a public API for implementing non web based clients. We use this API to create a header-less client program so that it can be run from Google's cluster network to emulate different network communication conditions between the client and server. This also allows us to inject access-level network conditions such as those for mobiles over cellular networks. Under the COLOR model, we actually use the session layer to simulate network conditions because the session layer plays the client role in the view of the COLOR framework.

6.1.2. Methodologies

With the goal of studying the tradeoffs between performance and consistency, we have followed the mathematical models in devising the evaluation test cases. This allows to compare the analytical results against the actual production systems, and to show how effective the

analytic model is in predicting the results. Because of differences between the assumptions of the mathematical model and the experimental environment, the goal is more qualitative than quantitative. It is hoped to predict the trends and scalability limits, and demonstrate the tradeoff factors and the effectiveness of the tunable parameters, rather than predict the actual delays observed in the experiments.

Overall we have created two groups of test cases.

The first involves only steady-state performance, with different client or server-side variables we need be able to measure and assess the target performance metrics defined in the analytic models. In general, we are not trying to prove the accuracy of quantitative analytic results with the evaluation results. The goal is to study key trade-off factors that may affect the system performance such as response time and scalability.

The second group of test cases involves transient events such as failovers. This evaluation is often very difficult due to the nature of being state-dependent. We rely on injected failures to simulate system events such as cluster failures or network timeouts. For this evaluation, we are more interested in the effectiveness of the COLOR solution; e.g., the rate of state convergence after a failover event, the stability of the system, and the lack of outliers in response to massive failures. We believe most of these problems are fundamentally engineering problems due to the excessive number of variables involved in the system behavior.

We used MATLAB for the graphic report of evaluation results. CSIM [39] is used for load generation from either simulated clients or the session layer.

6.1.3. Network topology and cross-cluster latency

The test bed uses 5 Google clusters with three of them located in the US and two of them in Europe. This is a typical setting for internet scale replicated services that offer both availability and fault tolerance in the face of failures that affect entire clusters or data links between two continents.

The three clusters located in US are named as: “center” (Center US), “east” (East US) and “south” (South US) respectively in the experiment of results. The two clusters located in Europe are named as “center-eu” (Center Europe) and “west-eu” (West Europe). In our experiments, we always designate “center” as the master or sequencer replica unless there is a deliberate master failover.

Figure 6.1 shows propagation delay distributions between the “center” cluster and each of the other four clusters.

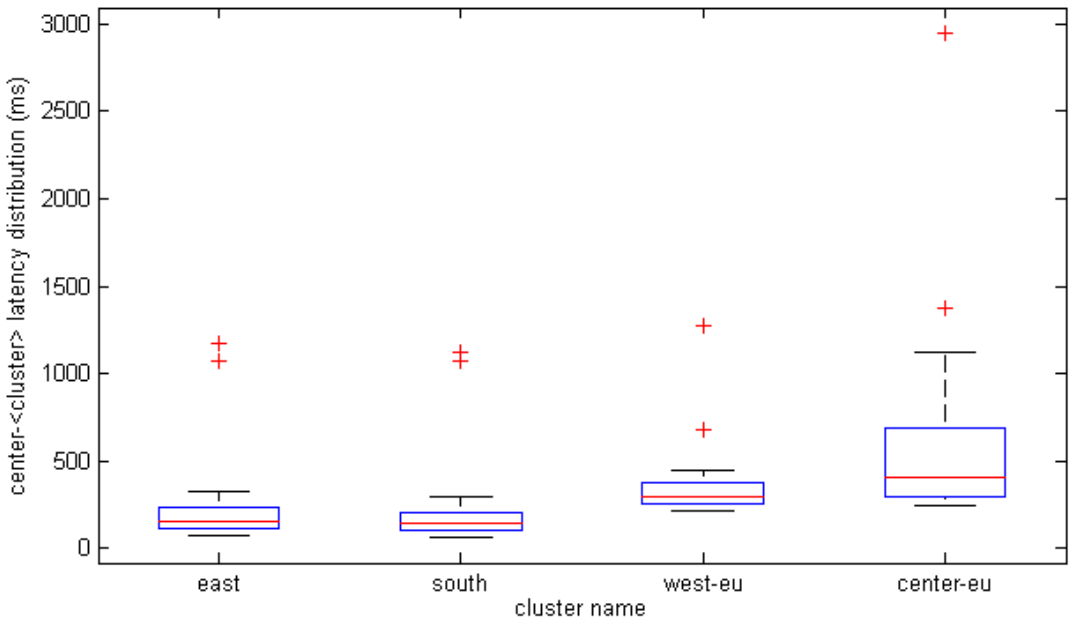


Figure 6.1: Distributions of propagation delays of network links between the “center” cluster and four other clusters, indicated by box plots.

From the above box plots, we notice that the network link latency distribution has a wide variance, especially for the two links between “center” and the two clusters located in Europe.

It was observed that the latency distribution is subject to time of day and traffic classification. For the experiments, most test cases were run for 2 hours using the “assured

forwarding" traffic class, which is commonly selected for running data replication algorithms, and which has a relatively stable and low level of packet loss.

Because the latencies are so much lower within the US, we can expect that the quorum needed for uniform delivery will usually be reached based only on acknowledgements from the three replicas located in the US, including the "center" itself as the master replica. This will limit the consensus delays for clients attached to local replicas located in the US. For other clients that need to interact with one of the two replicas in Europe, due to the master being located in US, those clients will be subject to the much larger latency over the inter-continental links.

In practice, for replicated state whose traffic is mostly originated from Europe, a dynamic load distribution scheme might be activated to move the master from the US (e.g., the "center" cluster) to a cluster in Europe. To avoid this, we chose not to measure any client traffic initiated from Europe as it might involve a different master or sequencer setting, and delays due to intentional or dynamic migration of the master.

6.1.4. FIFO channels

The COLOR performance models assume messages between clusters are sent via a FIFO channel, similar to traditional process-oriented replication models. In fact, messages can arrive out of order, and then have to be re-ordered. For the experiments, the out-of-order message propagation between two replicas was measured and found to range from 1% to 10%. This is due to an implementation detail. Specifically, the local replica that handles a client message is often not a persistent process in a cloud environment. When the local replica passes the message to the sequencer or master replica located in a different cluster, due to the network condition and process scheduling conditions on both ends, the FIFO ordering guarantee may be violated. Such a violation will introduce an extra join delay over the original empirical distribution of X. The analytic model ignores this extra delay. From the run time

statistics gathered from the system, the reordering delay seems to play a non-negligible role as part of X in the performance of the COLOR algorithms.

One way to look at this is the temporal stability of X. Without detailed time-series data for X, we cannot tell if the reordering results from network conditions or from scheduling jitter at either end of the message link. However, an artificial FIFO channel over the experiment cloud environment was not created as it would make the results less representative of modern cloud applications.

The other factor that could affect the FIFO channel property is the effect of batched delivery. While batching remains an important optimization for practical systems, for these experiments all batching was disabled in order to make the propagation delay X of individual messages an independent random variable.

6.1.5. Load-generation and reporting

For the evaluation simulated clients are running from the same cluster as the local replica against which client messages need to be sent to. Since the client-to-replica latency is not a concern for the evaluation, such a strategy will allow us to avoid any network bottleneck between clients and their local replica. We also use a load generator developed in house to generate the require client message rates, under Poisson arrivals. The load generate is required not to saturate the client itself with the load by running a sufficiently large number of parallel processes with enough memory and CPU allocated. In most cases we configure the load generator to run 10 processes across 10 different nodes with each process simulating up to 100 clients to generate the required message rates.

The response time is reported by individual clients and then aggregated into a single counter as managed by the production cloud environment in which the experiment is deployed. Similarly failures are reported and aggregated as they are detected by the client. Since clients are simulated, physical state recovery is not required when a client receives an invalidation

message. A client will simply resume operation upon any invalidation event, as if a new client just joined.

6.1.6. System parameters

Unlike the network links, for which real-time samples are recorded from the environment in order to generate the empirical distribution as inputs to the mathematical models, approximated mean processing times such as P_S and P_L are chosen based on what's observed in real systems. The mathematical models then apply an exponential distribution for those processing time parameters.

The number of clients is chosen arbitrarily merely to ensure there will be no bottleneck on the client side and the load generator is able to generate the required message rates.

The client or server message rates are chosen based on the results of the mathematical models, to allow us to compare the experiment results with the prediction results. In generally we try to generate enough loads to demonstrate the effects of the COLOR algorithms without causing too much runtime load which may trigger uncontrolled events such as cluster-level throttling events or excessive logging overhead.

Failures are all simulated. The failure rates of sequencers are also chosen based on the results of the mathematical model, in order to generate enough client failures in a reasonable simulation window such as several hours to one day. Failure intervals are fixed, mostly to avoid overlapping between two consecutive failure events. Such an overlap will cause complicated race conditions in reporting. We note that failures are rare events in the real environment, in which MTTF would be in days as minimum. Therefore, such a decision will not affect the effectiveness of the evaluation results.

Lastly the number of replicas and their topology are chosen to represent a typical set up in the target cloud environment. Although the number of replicas may affect the results in significant ways, we believe the choice of 5 replicas allows us to achieve the main goals of the

evaluation, that is, to demonstrate the effectiveness of COLOR algorithms in an environment that resembles what is faced by real-world cloud applications.

6.2. Non-uniform delivery

The non-uniform delivery reduces the response time as well as its variance by reducing the quorum based consensus algorithm to a single acknowledgement from the sequencer. Furthermore, because the sequencer itself involves little computation, the critical path of generating a response from the local commit layer is reduced to a single RTT between the local replica and sequencer under a light load.

6.2.1. Test bed design

As discussed earlier for the network topology, the local replicas are deployed over five different clusters, with three in US (“center”, “east” and “south”) and two in Europe (“center” and “west”).

The following tests were conducted with the sequencer located in different regions:

1. Response time distribution with the non-uniform delivery.
2. Response time distribution with the uniform delivery.

The uniform delivery is implemented using an interceptor which waits for the delivery of the global commit layer before passing the local commit layer.

As in the performance models, the client-to-cloud latency is excluded from the metrics by running the client in the same cluster as its local replica. This makes the results comparable with the analytic model results. In a real-world environment, response time seen by the user; e.g., from a browser client, will often be decided by the latency of the client-to-cloud access network, rather than Google’s production environment and the COLOR algorithm.

Response time is instead always measured at the session layer of the local replica. For the experiment, simulator clients are used to generate client messages from the same cluster as the local replica (as defined by the test case).

We also note that even if it is appealing to run clients outside the cluster network, the actual results will be affected by the client-access network latency as well as the client operating system, both of which are often less stable. Furthermore, the COLOR model treats session layers as real clients for what the COLOR algorithm is concerned. Therefore, the physical location of the client that initiates all the client messages is actually not relevant.

In the experiments, the response time is measured as the end-to-end delay of an RPC message that includes a single client message, and a single server message as the result of the local delivery of the corresponding server message. For the experiments, RPC was always used as the underlying messaging protocol even for one-way messages, mainly to allow their response time measurement. RPC processing overhead is included in the host demand for all the operations.

The second part of the evaluation involves the bottleneck analysis. In this test, we generate a significant load against the same whiteboard instance in order to show that the non-uniform delivery has a much better scalability.

The last part of the evaluation involves the inconsistency cost. On the server side, we randomly “restart” the sequencer by failing all the requests. When the sequencer is down, the local replica will see replication failure and will resort to global delivery. When the sequencer is up again, the local replica will start to communicate with the sequencer again. We evaluate the inconsistency cost by the number of client restarts during the failure window of the sequencer.

To quantify the impact of inconsistency, we inject the failure as a Poisson arrival process and calculate the client failure rate accordingly, measured as the mean time to failure for a single client. A client will report invalid messages as soon as the client is able to detect such messages, and the client will re-initialize its state from the server and resume operation. Similar

to the analytic model, we use many clients to generate the overall message rate, and the message rate for each client is kept much lower than 1/s so that we do not run into races in measuring invalid client events.

6.2.2. Evaluation result

1. End-to-end response time differences between uniform and non-uniform delivery

- Client message rate λ = 1/s

1.1. Clients from "east"

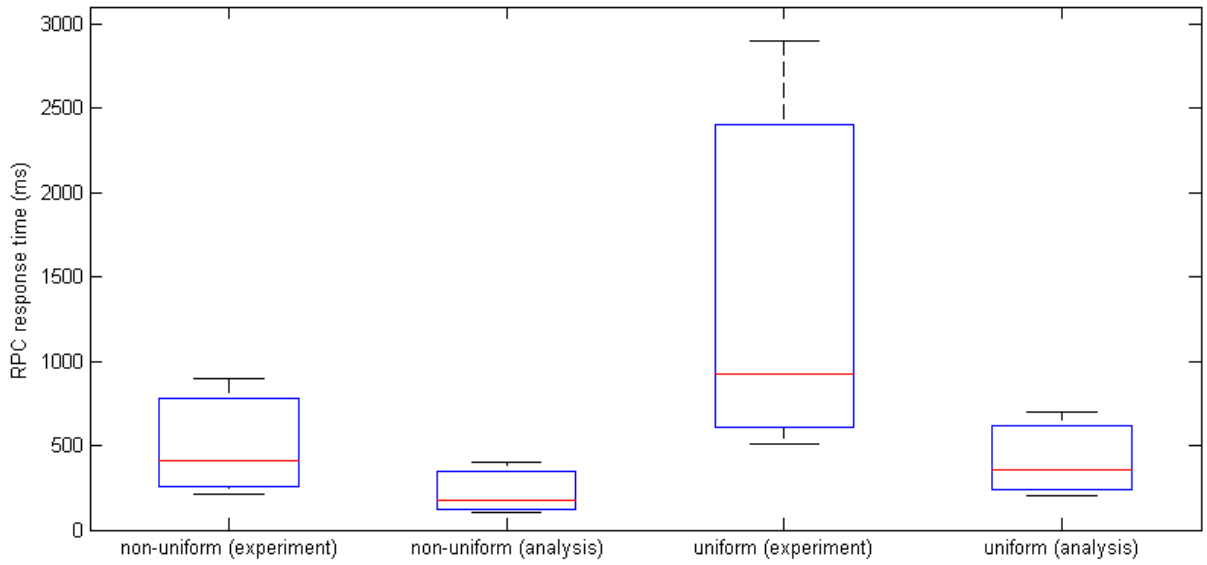


Figure 6.2: RPC response time distribution plotted with box plots for the non-uniform and uniform delivery with both experimental and analytic results. Clients are from "east" and the sequencer is from "center".

Overall the above result shows that the non-uniform delivery is able to reduce the response time significantly, with a much smaller response time variance than the uniform delivery.

We note here that the analytic results use the "east-center" link latency distribution for the network propagation delay X of all the links. In the experiment, the two clusters in Europe

will experience a much longer delay than “east”. As discussed in Section 6.1.4, the FIFO channel assumption is also not guaranteed in the actual experiment, which generates a much larger variance than the analytic results.

1.2. Clients from “west-eu”

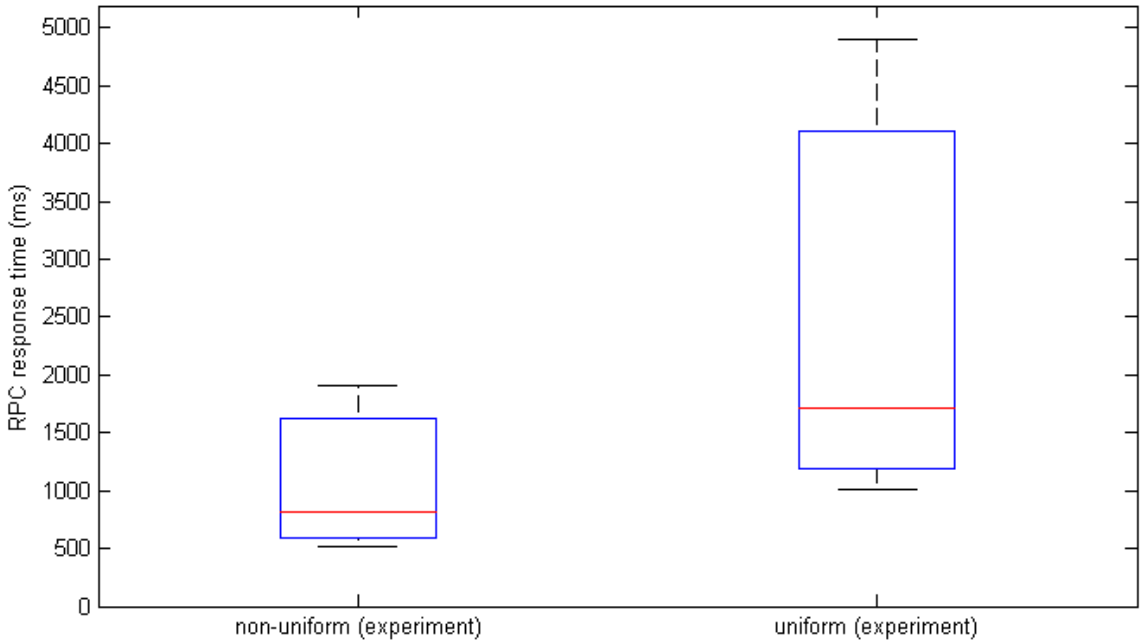


Figure 6.3: RPC response time distribution plotted with box plots for the non-uniform and uniform delivery. Clients are from “west-eu” with the sequencer being “center”.

1.3. Conclusion

The observed mean values of the response time for the non-uniform delivery are about half of those for the uniform delivery, and the 75th percentiles are about a third (thus, the improvement in the tail is greater than that in the mean).

The results match the qualitative predictions of the analytic model although the response time values are much higher, due to the much greater network latencies for clients from “west-eu”.

The results for clients from “center”, where the sequencer is, were not shown. For those clients, the response time approaches zero with this test bed.

Overall, non-uniform offers a much improved response time, especially considering the variance of end-to-end response times. The smaller variance is especially valuable to applications as it will significantly reduce the response time as perceived by clients, as human users will only be able to identify response time differences beyond a certain threshold, such as 200ms.

2. End-to-end response time under heavy load

- Client message rate: 50/s, 100/s
- Clients from “east”

2.1 Non-uniform delivery under increased loads

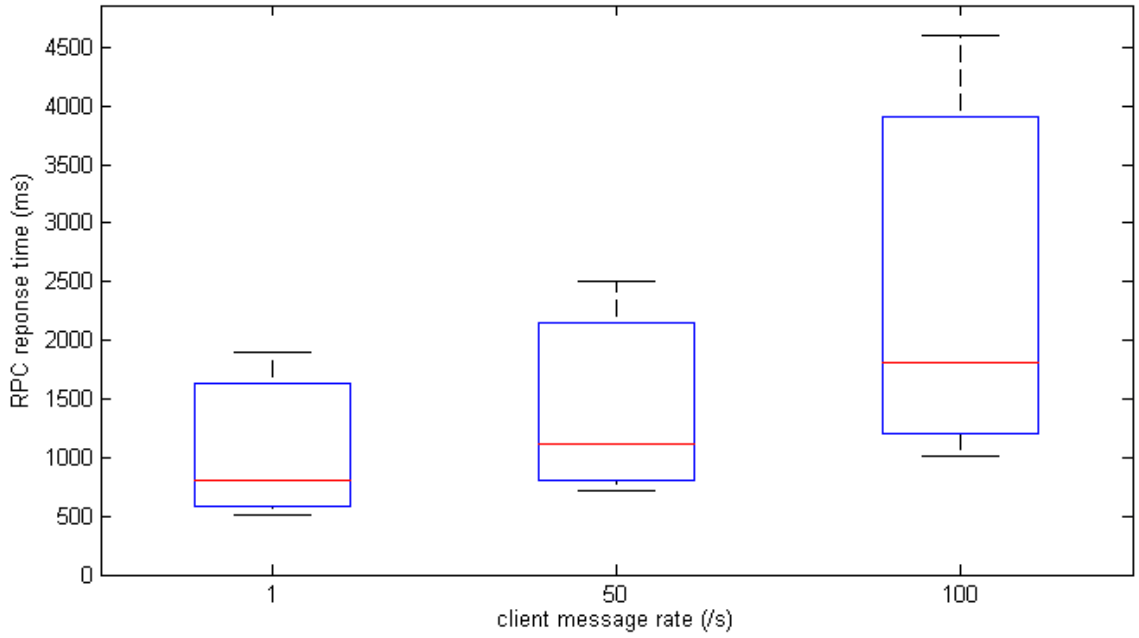


Figure 6.4: RPC response time distributions plotted with box plots under different client message rates. Non-uniform delivery with clients from “east”.

Comparing this to the analytic result in Figure 4.3, we can see the same trend of how the response time increases as the message rate increases. The increases are due to the bottleneck of the sequencer operation and the increasing local delivery response time.

2.2 Uniform delivery under increased loads

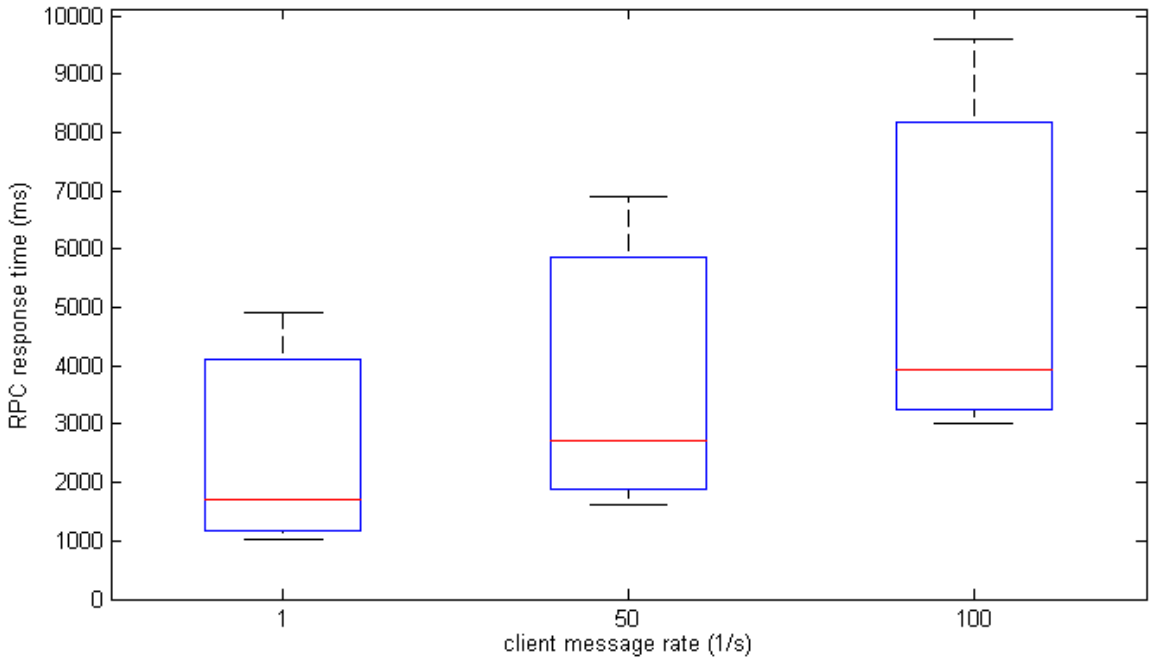


Figure 6.5: RPC response time distributions plotted with box plot under different client message rates. Uniform delivery with clients from “east”.

In Figure 6.5, cases with a higher load have a much higher variance in the response time, as well as a higher mean value. Comparing this to the analytic result in Figure 4.4, we can see that in both cases, the response time increases as the client message rate increases.

2.3 Conclusion

Both the non-uniform and uniform delivery are subject to the bottleneck of serialized operations on local replicas and on the master or sequencer replica. The bottleneck effect

seems to set in at about the same load levels. The uniform delivery will have a wider variance under a high load due to the two-phase commit processes which involves more log operations across all replicas.

3. Client failure rate when the sequencer fails

- Sequencer mean time to failure: 60s.
- Fixed failure intervals for the experiment to avoid overlapped failover runs.
- Total measurement time: 8 hours (480 sequencer failures)
- Client message rate: 10/s, 50/s, 100/s
- Due to RPC, the server message rate is equal to the client message rate

3.1 Clients from "east"

Client message rate (1/s)	10	50	100
Experiment client failure rate (1/s)	0.011	0.052	0.12
Analytic model client failure rate (1/s)	0.0864	0.432	0.864

Table 6.1: Client failure rates measured under different message rates.

For the experiments, S fails with a fixed MTTF, with clients from "east". For the analytic result, the same mean time to failure (60s) is used. For the experiments, we have to use a much smaller MTTF value (60s) in order to generate a sufficient large number of failure events in the real environment to calculate the client failure rate.

We note that the analytic result has a much higher failure rate due to the upper-bound analysis. In the actual experiment, conflicts are generated only under certain conditions that are controlled by the runtime environment, such as local scheduling and failure detection.

3.2 Clients from "west-eu"

Client message rate (1/s)	10	50	100
Experiment (west-eu) client failure rate (1/s)	0.008	0.042	0.09
Experiment (east) client failure rate (1/s)	0.011	0.052	0.12

Table 6.2: Client failure rates measured under different message rates.

In Table 6.2, the sequencer fails under a fixed MTTF of 60s, with clients from “west”. Due to the longer network propagation delay, client failure rates are reduced compared to those shown in Table 6.1.

3.3 Conclusion

From the above evaluation results, it is easy to see that the message rate will affect the client failure rate, as shown in the analytic model.

We note that clients from “west-eu” are less vulnerable to the sequencer failure due to the longer X, which will allow a higher probability for a new *seqId* to be acknowledged by the global commit layer between the time when it is delivered locally and the time when the sequencer dies.

We also note that the failure rate is much lower than the upper-bound given by the analytic model. This is because the timing requirement for an acknowledged *seqId* to be completely denied by the global commit layer is, in practice, much less than the entire assumed vulnerability window $Q(q,N_R)$. The underlying messaging passing system also plays a role in reducing the vulnerability window; e.g., when the sequencer fails, in-flight messages may still be able to be propagated to the target replica as if the sequencer had not failed.

4. Reduced client failure rate with a tunable delivery threshold parameter

- Sequencer mean time to failures: 60s
- Fixed failure intervals for the experiment.
- Total measurement time: 8 hours (480 sequencer failures)

- Client message rate: 100/s

4.1 clients from "east"

Delivery threshold (ms)	0	10	50	100
Experiment client failure rate (1/s)	0.12	0.092	0.074	0.063
Analytic model client failure rate (1/s)	0.864	0.314	0.284	0.271

Table 6.3: Client failure rate measured under a fixed client message rate but with different delivery threshold values.

In Table 6.3, we can see that increased threshold value sees reduced client failure rates, as predicted in the analytic result.

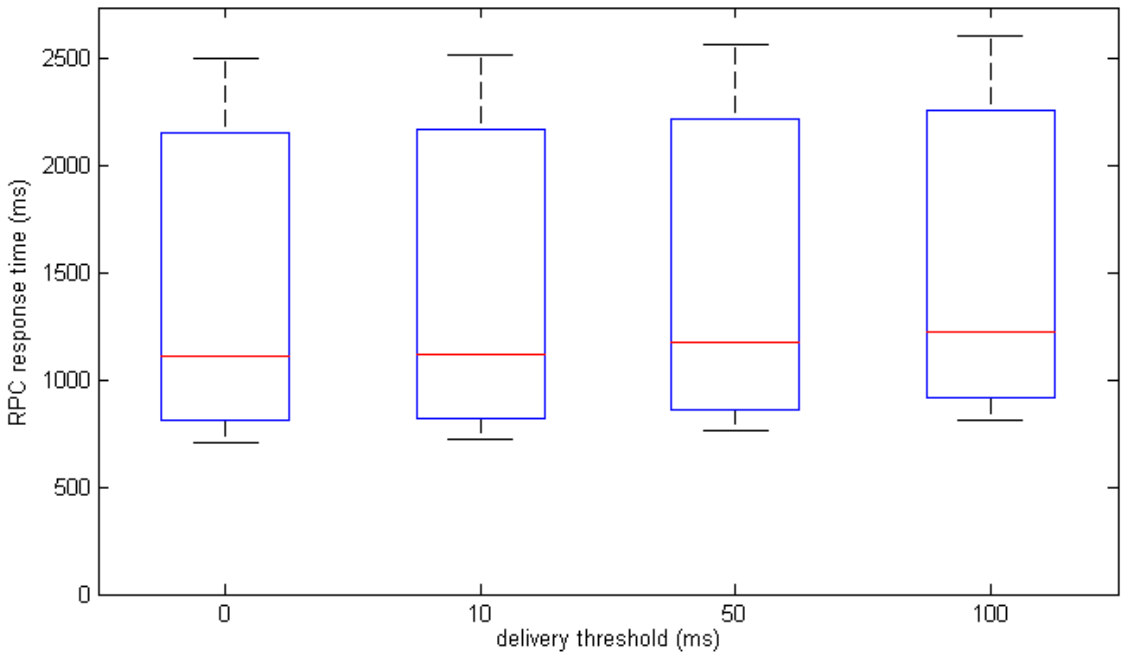


Figure 6.6: RPC response time distribution under different tunable threshold values.

In Figure 6.6, we can see that due to the large variance of response time under zero threshold value, the fixed threshold value has little effect on the actual distribution of RPC response time due to the relatively small threshold value (up to 100ms).

4.2 clients from "west-eu"

Delivery threshold (ms)	0	10	50	100
Experiment ("west-eu") client failure rate (1/s)	0.090	0.081	0.073	0.069
Experiment ("east") client failure rate (1/s)	0.12	0.092	0.074	0.063

Table 6.4: Client failure rate measured under a fixed client message rate but with different threshold values, with clients from the distant "west-eu".

In Table 6.4, we also copy the result for "east" and the result for "west-eu" shows that increased threshold value sees reduced client failure rates, comparable with what's for "east".

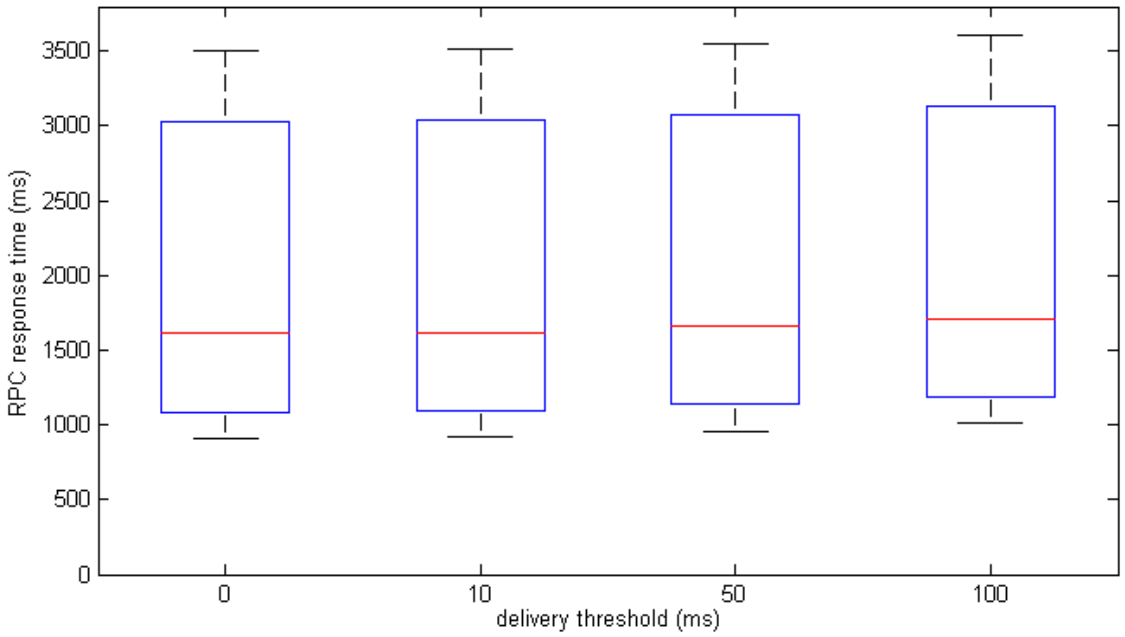


Figure 6.7: RPC response time distribution under different tunable timeout values, with clients from the distant "west-eu" cluster.

In Figure 6.7, we also identify that due to the large variance of response time under zero timeout, the fixed threshold value has little effect on the actual distribution of RPC response time.

4.3 Conclusion

From the above result, we can see that a small delivery threshold value will noticeably reduce the client failure rate. We note that there is no upper bound on the limit of threshold in reducing the failure rate. However, in practice we will limit such threshold to under 100-200ms in order to avoid any client-perceived response time increase under normal conditions.

As a trade-off, the artificial threshold adds a corresponding fixed increase to the RPC response time. On the contrary, the effect on the long-tail outliers of response time is rather small due to the fixed threshold.

6.3. State-sharding delivery

The goal of state sharding is to address the bottleneck issue. Unlike the non-uniform delivery, the state sharding by itself retains the uniform delivery and therefore is not subject to inconsistency cost from cluster or network failures. The purpose of the evaluation is to validate the effectiveness of such an approach.

6.3.1. Test bed design

We implement the sharding state algorithms of the COLOR framework for the whiteboard application. Note the sharding is transparent to the client application and therefore the performance and consistency evaluation methods remain the same.

The partitioning of the whiteboard state is modeled by dividing the canvas into multiple equal-sized sub-sections. For the purpose of this evaluation, there is only one level of partitioning. The size of partition is configured statically on the server side and the user is not aware of the sharded key space.

The whiteboard application allows the drawing pen to be sized into any arbitrary width. Therefore every sampling of the drawing action will generate a set of updates which is grouped into an atomic change set, as a single client message which contains all the pixels that have been updated by the last client action. Without state sharding, client messages are ordered at

the action level, which will show what one would expect on a shared whiteboard; e.g., each drawing action is contiguously painted on the canvas, and any overlap actions on the same position of canvas will not cause mixed colors at the pixel level.

On the replica side, we also choose to assign a different sequencer process for different sections of the canvas randomly. This emulates the real world scenario when it is desired to assign sequencers to replicas where most client messages will arrive.

6.3.2. Evaluation result

The evaluation of state sharding delivery involves no failure events. A set of clients is created and they are assigned to different replicas where sequencers are also located. Each simulated client will generate a random set of painting actions with different size of canvas coverage. The scope of these actions may overlap with each other over different partition boundaries.

The evaluation was carried out for both the uniform delivery and the non-uniform delivery. The latter allows us to enable and evaluate two optimistic approaches at the same time.

1. End-to-end response time comparison, between single-shard and multiple shards

- Client message rate: 100/s
- Clients from “east”

1.1. Uniform delivery

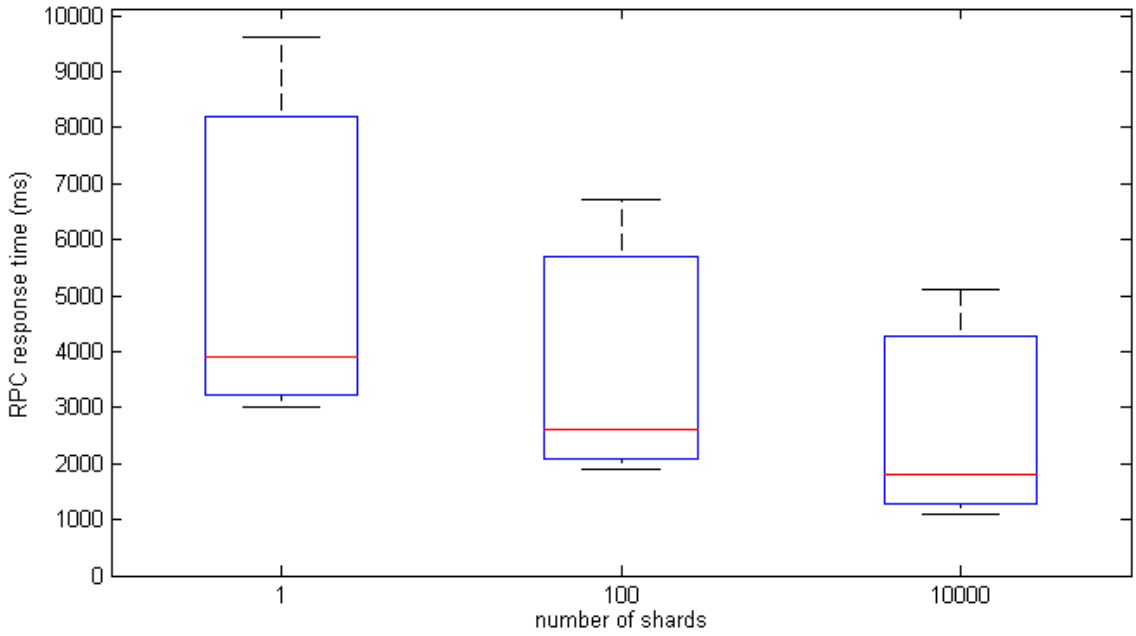


Figure 6.8: RPC response time distributions under different number of shards for the uniform delivery.

In Figure 6.8, we show how under a high client message rate, the sharding will reduce the bottlenecks and therefore improve the response time.

1.2. Non-uniform delivery

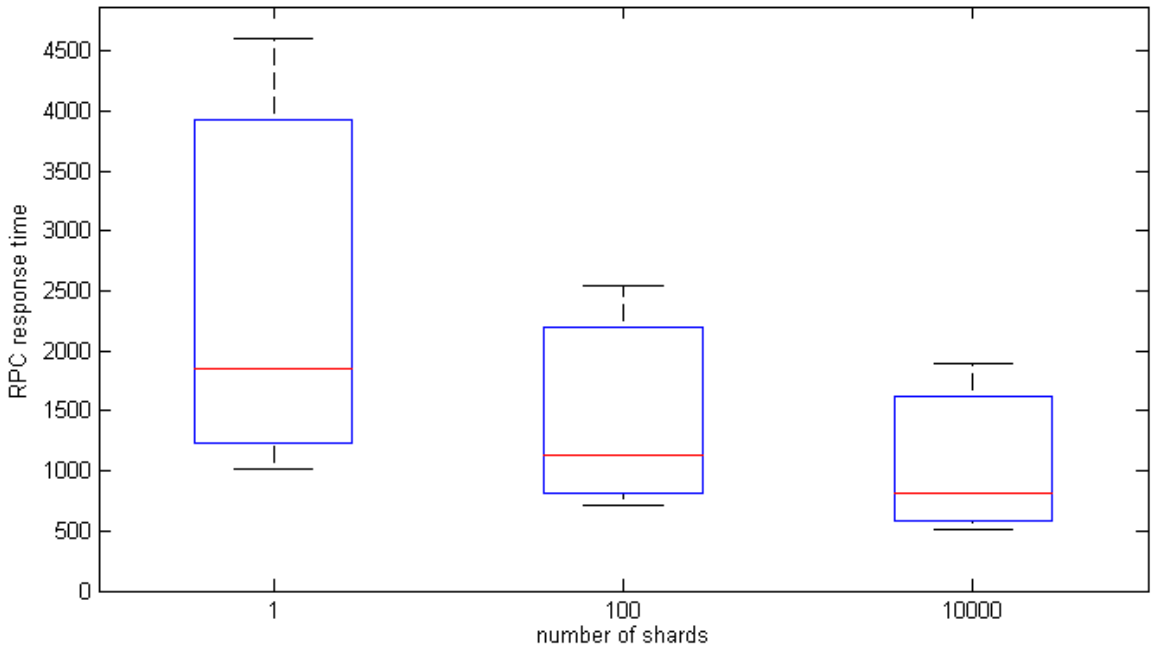


Figure 6.9: RPC response time distributions under different number of shards for the non-uniform delivery.

In Figure 6.9, for the non-uniform delivery the response times are much smaller than for the uniform delivery, but they have the same trend as sharding increases. Under a high client message rate, the sharding reduces the sequencer bottleneck and therefore improves the response time.

1.3 Conclusion

The above results show that sharding significantly reduced the bottleneck when the replication system was under a high load. With a large degree of sharding the response time under a high load is approaching the response time under a light load for both the uniform and non-uniform delivery.

2. Client failure rate due to overlapped messages

- Client/server message rate: 100/s

- Clients from “east”
- % of state that each message updates: 1-10% with uniform distribution.
- With 100 shards, each message updates 1-10 (contiguous) shards with uniform distribution
- All masters or sequencers are located in “center”; i.e., no per-shard load distribution

2.1. Uniform delivery

Number of shards	1	100	1000	10000
Uniform delivery client failure rate (1/s)	0	0.0055	0.063	0.221

Table 6.5: Client failure rates under different numbers of shards for the uniform delivery.

In Table 6.5, we can see that, as the number of shards increases, the rate of conflicts also increases due to concurrent local delivery on different shards.

2.2. Non-uniform delivery

Number of shards	1	100	1000	10000
Non-uniform delivery client failure rate (1/s)	0	0.0045	0.059	0.211

Table 6.6: Client failure rates under different numbers of shards for the non-uniform delivery.

In Table 6.6, we can see that the non-uniform delivery will experience approximately the same conflict rates as the uniform delivery, and both cases will see increased conflict rates as the number of shards increases.

2.3. Conclusion

The above results show the effectiveness of trade-offs via the number of shards to balance the inconsistency cost and the system bottleneck.

We note that the actual result is also highly tied to the run time environment. For instance, the rather small difference between the non-uniform delivery and uniform delivery is likely caused by the two different implementations of master and sequencer processes.

On the other hand, we are able to apply the sharded delivery to both the uniform and non-uniform delivery in an orthogonal way. When the sequencer remains correct, the conflicts caused by optimistic sharding is completely caused by the fact that the *seqId* is generated by two different logical processes within the same cluster.

The evaluation does not address the effect of having the per-shard sequence or master located in different clusters. Such a setting will likely increase the conflict rate. It also does not address the effects of sharding on the client message failure rate when the sequencer fails or in the event of the master or sequencer fail over. The subject of using sharded delivery to reduce the failure rate due to master or sequencer failover will represent an interesting experimental topic for future work.

6.4. Non-blocking master election

The non-blocking master election addresses the failover delay, which is not bounded when a sequencer fails under the non-uniform delivery. Since the goal of the non-uniform delivery is to improve the response time, especially the long-tail response time outliers, a bounded master election is essential when the network condition is not stable.

6.4.1. Test bed design

The test bed is the same as the basic test bed designed for the non-uniform delivery. For this evaluation, the sequencer is modified to block the delivery of local commit messages for a

certain duration. When the local replica cannot get an acknowledgement from the sequencer, it will start to deliver messages locally.

For each cluster, a set of clients was assigned to generate messages randomly to their respective local replicas at a fixed rate. When the sequencer fails, locally delivered messages may conflict with each other, and when the sequencer comes back a rollback may be triggered immediately.

6.4.2. Evaluation result

Unlike the steady-state response-time evaluation, the results from this test bed have limited value in verifying the analytic results. We are mostly interested in gaining insight into the design and implementation to guide the use of this kind of optimistic algorithm.

The following metrics were evaluated to show that the COLOR framework is capable of handling transient failover states.

- Client message rate: 50/s + 50/s, with the same number of clients from “east” and “west-eu”
- Failover mean time to failures: 60s
- Fixed failover intervals.
- Total measurement time: 8 hours (480 failures)

1. Client failure rate during a master failover

Delivery threshold (ms)	0	10	50	100
Experiment client failure rate (1/s)	0.086	0.081	0.063	0.046
Analytic model client failure rate (1/s)	0.122	0.101	0.076	0.062

Table 6.7: Client failure rates under different tunable threshold values, for the master failover under the uniform delivery.

In Table 6.7, we can see that an increased delivery threshold is able to reduce the client failure rate (due to message conflicts) significantly. The experimental result also matches the analytic result, in how effective the threshold is in reducing the client failure rate.

2. Client failure rate during a sequencer failover

Delivery threshold (ms)	0	10	50	100
Non-uniform client failure rate (1/s)	0.085	0.081	0.063	0.046
Uniform client failure rate (1/s)	0.086	0.081	0.063	0.046

Table 6.8: Client failure rates under different tunable timeout values, for the sequencer failover under the non-uniform delivery.

In Table 6.8, we can see that the failure rates of a sequencer failover under the non-uniform delivery and of a master failover under the uniform delivery are about the same. The reason behind this is that the client failure rate is decided by the message conflict probability between two local replicas. Therefore, under the same client message rate, both cases will see the same message conflict rate, within the same conflict window that is decided by the delay of electing a new master or sequencer.

3. Conclusion

The above results show the effectiveness of the delivery timeout in reducing the failure rate due to non-blocking master or sequencer failover.

The difference between the uniform and non-uniform cases is expected to be very small. This is understandable as the nature of conflicts generated during the sequencer or master failover is from the local replica. Therefore, the invalid message rate is mostly decided by the conflicts between two correct replicas; in our case between “east” and “west-eu” replicas.

Since messages are delivered locally, the response time increase due to the local delivery timeout will be a constant parameter. For the sake of brevity, we omit the charts that show the constant response time increase that matches the threshold value.

6.5. Insights from the experiments

The experiment results allow us to verify the predictions of the mathematical model as well as the effectiveness of the solution in a real cloud environment. In addition to these basic goals, the evaluation has also offered the following insights, which may prove useful for future users of the COLOR solution.

6.5.1. Verification of correctness

It's well known that distributed algorithms are hard to prove correct and they are also hard to implement correctly in a real environment which often involves many variables than the abstract system model against which the algorithms are specified.

Since the goal of COLOR algorithms is to ensure end-to-end consistency between the client and the global commit layer, in measuring the inconsistency cost we have also had the opportunities to verify that the COLOR optimistic algorithms and their implementations are correct under the experiment conditions. The following statements are verified for every experimental run:

1. No client failure was recorded under the non-uniform delivery when the sequencer was not failing.
2. No client failure was recorded under the sharded delivery when client messages did not overlap.
3. No client failure was recorded under the non-blocking failover of a sequencer or master when there was no conflicting message from other remote replicas.

The above observations assume the global commit layer is always correctly implemented, which should have been the case since the global commit layer runs an external Paxos system.

Assuming the local message history is always generated correctly, those invalidation messages generated in the experiment also indirectly verified those state-validity properties established in Section 3.4 and 3.6 for the three optimistic COLOR algorithms.

Lastly the experiment result also verified the non-blocking property of the COLOR algorithms as RPC is used for generating server messages in our experiment and any timeout of a server-generated response message will cause unexpected runtime errors to be reported.

To summarize, with the use of an external, proven Paxos implementation as the global commit layer, we believe that the current experiment results are sufficient to verify the correctness of the COLOR algorithms as implemented in the current prototype.

6.5.2. Choice of the tunable value

Based on the results from the experiment, in comparison to the prediction of the corresponding mathematic models, the following procedure is suggested to choose the tunable value, such as the delivery threshold or the number of shards, will work as follows:

1. Make the delivery threshold as long as possible, consistent with the end-to-end response time requirement and the runtime measurement. For example, if the end-to-end requirement is 1000ms and the average response time under normal conditions, as measured in the real environment, is 500ms, then a 50ms-100ms value would be a good candidate which offers a reasonable reduction on failures while limiting the response time cost to only 10%.
2. Choose the number of shards based on the static analysis of the data model and the client-to-server state access interface. Similar to database design, the goal should be to partition the data model based on the understanding of the data relation

underlying the data model. In generally loosely coupled data sets should be in different shards, when highly coupled data sets should not.

3. Always rely on the monitoring to provide useful feedback on the current setting. Optimizing the real-world environment is not necessarily a science; and one has to take into account the dynamics of the environment and create a system in which results of tunable parameters are accurately monitored and new values can be easily configured and pushed into the runtime. Automation of the above process is however beyond the scope of this work.

Chapter 7 Programming against Temporary Client Inconsistency

In this chapter, we will discuss how applications may recover from invalid state under the COLOR model without resorting to a full restart every time an invalid state is detected.

We believe a framework such as the COLOR framework may play a significant role in assisting applications to design and implement a client-side data model that is tolerant of temporary inconsistency.

We also point out that the problem to deal with inconsistent view of server states is not unique to the COLOR optimistic replication, and therefore a generalized solution would benefit any type of cloud based applications that have to access server states over the Internet regardless of how server states may be replicated.

7.1. Programming guidelines

The core of the programming guidelines for coping with inconsistency is the introduction of the concept of tentative states. Any update issued by the client, even after a response from the server has been received for the original update message, is to be marked as a tentative update. As a result, the state that has been updated by such a message will be marked as a tentative state on the client side. Likewise, any notification or query response from the server will also have such a tentative status against the local state on the client side, which carries the client view of the server state. The tentative status of a state is to be changed to the committed status after the client receives a global message history context that includes the last message (*m_id*) known to the client; i.e., after *m_id* is validated against the global message history for the given state.

When the client-side state has an explicit tentative vs. committed status, any rollback due to optimistic algorithms will be treated just like a traditional three-way merge [149], in order not to lose any client-side messages, such as edits made by the user.

First, we note that resolution of any conflicts between concurrent server changes and edits made by the local client may have to involve some out-of-band mechanisms such as operational transformation [98] or even manual conflict resolution. Either solution is highly dependent on the application semantics, and needs be applied as an external module to the framework. This situation is no different than any solution that employs optimistic concurrency control to allow updates to the cached server state on the client side.

Second, one unique property of the COLOR algorithm is that server messages may carry tentative states too. This is new compared to other existing solutions in which tentative states are introduced only on the client side. We argue, however, that this should not introduce any semantic difference on the client side in how local state cache is to be managed. During a three-way merge to reapply local edits to the latest global version, it is possible that some replayed client messages may depend on messages from other replicas that appear in the previous local message history but are not in the latest global message history; e.g., due to replica failures, as shown in Figure 7.1.

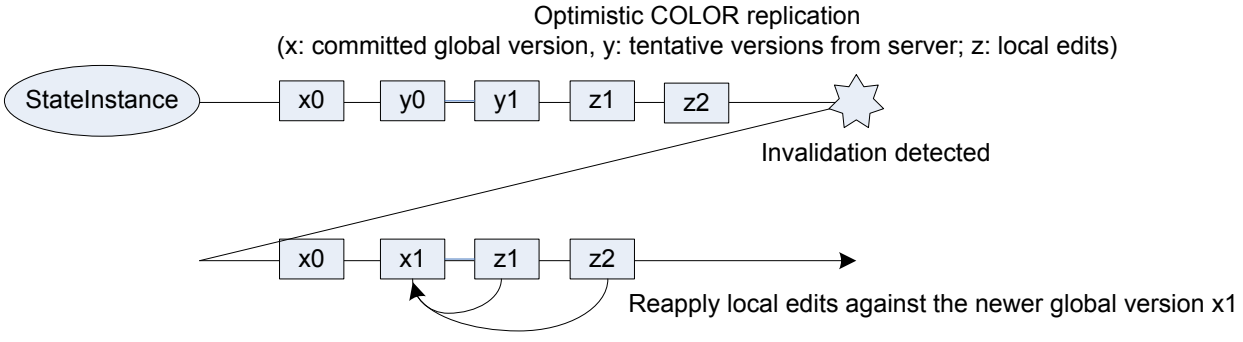


Figure 7.1: Upon an invalidation message, the client-side framework reapplies the local edits *z1*, *z2* against the newer global version *x1*, which misses the two tentative updates *y1, y2* that the local edits may depend on.

The above situation is unique to COLOR since the standard three-way merge algorithm will only replay client messages generated on the local client. However such a situation does not really violate the assumption of any three-way merge algorithm, which merely requires a common base version to reapply local edits against a newer message history committed concurrently by the server; e.g., due to updates from other clients. In other words, implicit causal relation between a local edit and a specific message history version may still be broken under a traditional three-way merge model, such as the one for managing optimistic concurrency control of the client-side cache, as shown in Figure 7.2.

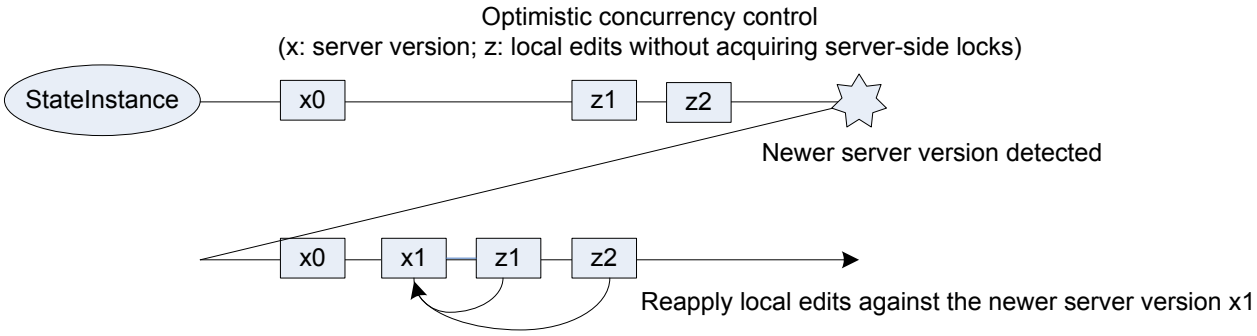


Figure 7.2: Under a typical optimistic concurrency control model, a newer server version will require reapplying local edits against the new server state and resolve any conflicts thereof.

In the above example, if *x1* masks any specific value of *x0* against which *z1*, *z2* is generated, then the causal relation between *z1, z2* and *x0* will be broken after *x1* is loaded. Resolving such missing causal dependencies will always require knowledge specific to the application data semantics or a manual resolution will have to be conducted.

To further unify the COLOR model with traditional models, we may transform a missing dependency message *y* (due to server rollback) into a faked client message *y'* in the global message history (as seen by the client) that simply undoes the tentative message *y* that failed to be committed to the global history. With such an approach, we now have a unified model between COLOR and any standard three-way merge model, as shown in Figure 7.3.

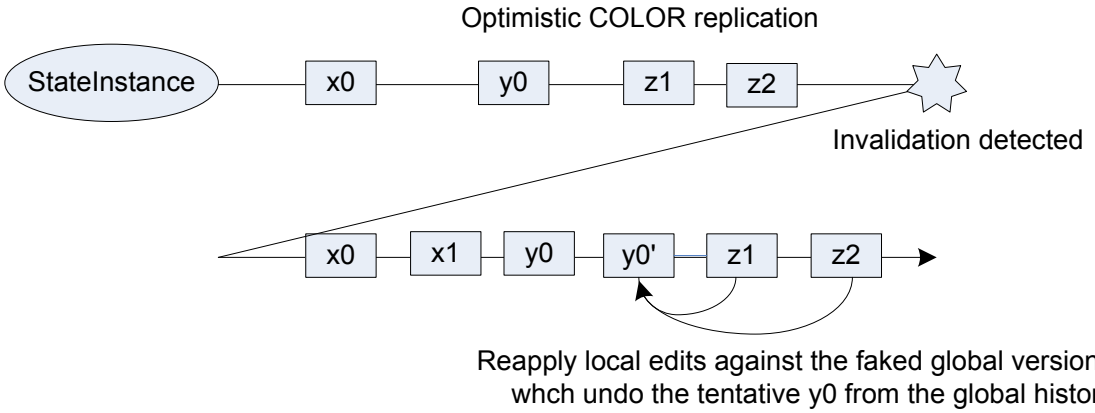


Figure 7.3: Transform the COLOR model which involves tentative server versions to a traditional three-way merge model for which only client-side tentative versions are possible.

Lastly, for the COLOR model it is possible to indicate to the server that only globally committed messages should be generated for the client; e.g., when an RPC is to be committed globally in a blocking fashion. The support of explicit commit messages, as defined in Section 3.5, allows a client-side framework to treat certain messages specially and to control the extent of tentative states; e.g., by creating consistent check-points or snapshots of the server state.

7.2. Framework support

There are popular client-side frameworks or solutions that already support the notation of operational transformation or the general three-way merge model. The latter is becoming especially popular with emerging solutions that aim to offer mobile platforms a consistent data model to develop both online and offline applications.

For the COLOR framework, the present author has created a simple client-side, javascript-based state representation model which manages the client-side cache with the following extra context data for each state key:

1. The tentative status
2. The replica message history, including the latest globally committed version
3. The client generated local message log.

In response to any invalidation message, which comes with the latest global version known to the local replica, the following algorithm is applied to the three kinds of context data on the client side:

1. Change the current replica state from the tentative status to the committed status.
2. Replace the replica message history with the latest version from the invalidation message.
3. Using the last known global version prior to the invalidation as the baseline, and the new global version as the online version and the local message history, include their dependencies from other clients as the offline version, reapply any missing messages to the server, with their original message ids, when conflicts are not detected.

We note that on the server side, the COLOR framework will filter out any redundant messages generated by the client, at both the local and global commit layer.

The detection of any data conflicts using the above conflict resolution approach is the difficult part, and we rely on an external resolver such as [133] [149], to report and resolve any conflicts. At the current stage, we choose to have the user to resolve all local pending messages if they conflict with the global version that invalidates the local message history.

7.3. Real-world considerations

The above programming model aims to preserve any user generated messages in the event of client invalidation as triggered by the COLOR algorithm, instead of forcing a client

restart. This has many direct benefits for applications that are user-facing and receiving user generated edits. However, due to the causal dependency on messages from other clients that may have also been lost during the replica failures, a completely automated conflict resolution is often impossible.

We also note that when a new global version is loaded the client must assume that all prior knowledge of previous server state may be invalidated. That is, a “memory-less” model has to be adopted and the client has to think that the new version is the latest version of the replica state although it may be older than the latest tentative message history version the client has seen. This behavior forces a tracking back to history, and the use cases for such a model have to be state dependent rather than message dependent.

Lastly, for clients that do not interact with human users directly, a restart is often desired as it allows a much faster and cleaner recovery. A typical use case may involve a stateless software agent that is doing some computation against some replicated state in the cluster. When failures actually happen, restarting the affect software agent may be the best choice rather than attempting any recovery.

7.4. The common case for tentative server states

We believe the requirement to roll back a conflict on the client side due to optimistic delivery is not unique to optimistic replication. There are several other use cases whereby a client-side application has to handle server states that are only tentatively committed, and may be rolled back in future.

Therefore, we are positioning this experimental client-side solution as a unified approach for all the three uses defined below:

1. Optimistic concurrency control based on versioning
2. Operational transformation for offline edits
3. Optimistic replication as proposed by the COLOR framework.

Given the wide adoption of mobile based clients, the first two scenarios are increasingly common. In other words, we do not believe the COLOR solution introduces a unique challenge to applications in dealing with inconsistency, although the first two have the property that the clients are more aware of tentative delivery on the client side, since they are explicitly introduced on the client side directly.

Chapter 8 Conclusions

8.1. Summary of contributions

The COLOR solution gives a satisfying answer to the question of how to make tradeoffs between consistency and performance for distributed replication systems. By introducing the notion of online validation and state convergence, the COLOR solution differs from the popular eventual-consistency solution in its novel way to bound the performance degradation while minimizing the duration of transient inconsistency on either client or server side.

The COLOR solution enables a list of tunable parameters precisely because the COLOR algorithms run both the optimistic and strict algorithm side by side. In doing so, the solution addresses not only the steady-state performance but also the failover performance; and at the same time offers different tunable parameters to reach both response-time and scalability goals of the system.

The COLOR solution is able to significantly improve the performance of the replication system, as shown in the analytic results and experiments. In the absence of failures, the response time performance of the COLOR solution was much better than that of a strict implementation that used Paxos-based software to maintain strict consistency. The response time will involve only 1 between-cluster RTT when clients are interacting with a replica that is remote to the sequencer, while the Paxos-based solution will involve two between-cluster RTTs. Furthermore, the overall response time will see a much reduced variance with the COLOR solution. The COLOR solution is also much more scalable due to reduced bottlenecks under a heavy load. The response times were longer in the presence of failures than without failures, but still comparable with that of the strict approach because the COLOR optimistic algorithm will always fall back to the strict approach during any failover events. The inconsistency cost due to

failures, which is not incurred by the strict approach, was very small. Probabilistically only a small fraction of clients may be affected by invalid states.

The prototype implementation of the COLOR solution uses a whiteboard application which places typical consistency demands on its distributed data. The evaluation result based on the implementation and the whiteboard application demonstrates the feasibility and effectiveness of the overall approach. Since most of the work is done in a real-world cloud environment, we are in the process of publishing the COLOR framework software as a reusable solution for any cloud applications that involve replicated state representation and a REST-based client-server state access interface.

8.2. Future work

Based on the understanding created in this thesis, of the tradeoff between performance and consistency under the COLOR model, it may be possible to further simplify the overall solution as well as its underlying abstractions, to promote its adoption as a mainstream solution for the important Eventual Consistency use cases. For this purpose, the following future research could significantly extend the current scope of the COLOR solution.

8.2.1. Cloud programming model

The new programming model to deal with state rollback on the client side could be further formalized. More importantly the API between the state representation model and the conflict resolution needs be further specified. The latter itself is an active research area too.

The current model does not clearly address the causal dependency on invalid or tentative state versions generated by other clients. Although many heuristic solutions exist in real-world implementations to solve such a problem, it may be possible to find a solution that is easier to understand at the design phase. For that, we need identify more real-world use cases to verify the design and programming model.

8.2.2. Transactions and sharding

The general problem of the relation between transactions and replication has been studied intensively [63] [77]. However, there is still a lack of well-understood models to unify transactions (including distributed transactions) with data replication (and the state machine model).

The state sharding approach has a tight relationship with scheduling transactions across multiple replicas. A unified system model between distributed transactions and data replication will help to clarify the value of state sharding.

Our current approach mainly uses the atomicity requirements as a way to represent relationships between shards in order to identify conflicts between client messages. While we believe these relationships represent the most fundamental nature of transactions, there is still a lack of a system model to clearly establish this.

8.3. Shortcomings of the COLOR model

While the COLOR model offers a novel solution to the performance problem of geographically replicated cloud services, the overall solution is not without shortcomings.

In implementing the whiteboard example with the prototype COLOR framework, we noticed that the framework could be very difficult to use for developers without a deep understanding of the application, especially in how to handle inconsistency when it arises. Furthermore, the client-side recovery imposes a significant overhead for client-side developers, for whom the concept of tentative server messages has to be incorporated into the design at the very beginning to actually benefit from the optimistic COLOR model.

The COLOR model, by introducing an extra layer of optimistic replication on top of the strict replication, will also incur certain run-time overhead, which includes extra CPU cost for managing the two-tier commit algorithm, sharded delivery, and state validation. In the case of

sharded delivery, the complexity of the overall system will also increase due to the large number of parallel sequencers that have to be managed in the runtime, even though we consider this part of the system external to the core of the COLOR framework.

Bibliography

[1] Adve, S., Gharachorloo, K.: Shared Memory Consistency Models: a Tutorial. J. Computer vol. 29-12 pp. 66-70 (1996)

[2] Ahamad, M., Neiger, G., Burns, J., Kohli, P., Hutto, P.: Causal Memory: Definitions, Implementation and Programming. J. Distributed Computing, vol. 9-1 pp. 37-49 (1995)

[3] Ahamad, M., Raynal, M.: Ordering vs. Timeliness: Two Facets of Consistency? In: Future directions in distributed computing pp. 73-77 (1999)

[4] Allcock, B., Bester, J., Bresnahan, J., Chervenak, A., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S., Foster, I.: Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In: Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies pp. 13 (2001)

[5] Amir, Y., Caudy, R., Munjal, A., Schlossnagle, T., Tutu, C.: N-Way Fail-Over Infrastructure for Reliable Servers and Routers. In: Proceedings of International Conference on Dependable Systems and Networks pp. 403-412 (2003)

[6] Amir, Y., Chockler, G., Dolev, D., Vitenberg, R.: Efficient State Transfer in Partitionable Environments. Institute of Computer Science, The Hebrew University (1997)

[7] Amir, A., Danilov, C., Miskin-Amir, M., Schultz, J., Stanton, J.: The Spread Toolkit: Architecture and Performance. (www.spread.org) (accessed in Dec. 2013)

[8] Anderson, T., Breitbart, Y., Korth, H., Wool, A.: Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. J. ACM Transactions on Database Systems vol. 18-3 pp. 460-486 (1993)

[9] Anderson, T., Breitbart, Y., Korth, H., Wool, A.: Replication, Consistency, and Practicality: Are These Mutually Exclusive? In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data pp. 484-495 (1998)

[10] Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, L., Leon, J., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: Proceedings of the Conference on Innovative Data system Research pp. 223-234 (2011)

[11] Baldoni, R., Cimmino, S., Marchetti, C., Termini, A.: Performance Analysis of Java Group Toolkits: A Case Study. In: International Workshop on Scientific Engineering for Distributed Java Applications pp. 49-60 (2001)

[12] Baldoni, R., Marchetti, C.: Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems. J. IEEE Transaction on Computers vol. 53-5 pp. 497-511 (2004)

[13] Baldoni, R., Marchetti, C.: Three-tier Replication for FT-CORBA Infrastructures. J. Software Practice & Experience vol. 33-8, pp. 767-797 (2003)

[14] Baldoni, R., Marchetti, C., Termini, A.: Active Software Replication through a Three-Tier Approach. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems pp. 109 (2002)

[15] Balsamo, S., Marco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. J. IEEE Transactions on Software Engineering vol. 30-5 pp. 295-310 (2004)

[16] Baratloo, A., Chung, P., Huang, Y., Rangarajan, S., Yajnik, S.: Filterfresh: Hot replication of Java RMI Server Objects. In: Proceedings of the 4th conference on

USENIX Conference on Object-Oriented Technologies and Systems vol. 4 pp. 5-5 (1998)

[17] Basile, C., Kalbarczyk, Z., Iyer, R.: A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks pp. 22-25 (2002)

[18] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A Critique of ANSI SQL Isolation Levels. P. ACM SIGMOD international conference on Management of data pp. 1-10 (1995)

[19] Bolosky, W., Bradshaw, D., Haagens, R., Kusters, N., Li, P.: Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In: 8th USENIX Symposium on Networked Systems Design and Implementation pp. 11-11 (2011)

[20] Box Plot (http://en.wikipedia.org/wiki/Box_plot) (accessed in Dec. 2013)

[21] Breitbart, Y., Komondoor, R., Rastogi, R., Silberschatz, A., Seshadril, S.: Update Propagation Protocols For Replicated Databases. In: ACM SIGMOD pp. 97-108 (1999)

[22] Birman, K.: Building Secure and Reliable Network Applications. Prentice Hall pp. 292-302 (1997)

[23] Birman, K.: The Process Group Approach to Reliable Distributed Computing. In: Communications of the ACM vol. 36-12 (1993)

[24] Birman, K., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal Multicast. In: ACM Transactions on Computer Systems vol. 17-2 (1999)

[25] Birman, K., Schiper, A., Stephenson, P.: Lightweight Causal and Atomic Group Multicast. In: ACM Transactions on Computer Systems vol. 9-3 (1991)

[26] Budhia, R.: Performance Engineering of Group Communication Protocols. Dissertation (1997)

[27] Burrows, M.: The Chubby Lock Service for Loosely-Coupled Distributed Systems. In: Seventh Symposium on Operating System Design and Implementation pp. 335-350 (2006)

[28] Chandra, T., Griesemer, R., Redstone, J.: Paxos Made Live: An Engineering Perspective. In: Proceedings of the twenty-sixth annual ACM Symposium on Principles of Distributed Computing pp. 398-407 (2007)

[29] Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. In: Journal of the ACM vol. 43-2 (1996)

[30] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A Distributed Storage System for Structured Data. In: Seventh Symposium on Operating System Design and Implementation vol. 7, pp. 15-15 (2006)

[31] Chereque, M., Powell, D., Reynier, P., Richier, J.-L., Voiron, J.: Active Replication in Delta-4. In: Twenty-Second International Symposium on Fault-Tolerant Computing (1992)

[32] Chockler, G., Keidar I., Vitenberg, R.: Group Communication Specifications: A Comprehensive Study. In: ACM Computing Surveys vol. 33, iss. 4 (2001)

[33] Coccoli, A., Urban, P., Bondavalli, A., Schiper, A.: Performance Analysis of a Consensus Algorithm Combining Stochastic Activity Networks and Measurements. In: Proceedings of International Conference on Dependable Systems and Networks pp. 551-560 (2002)

[34] Cometd Protocol. (http://cometd.org/) (accessed in Dec. 2013)

[35] Corbett, J., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y.,

Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's Globally-Distributed Database. In: Proceedings of Tenth Symposium on Operating System Design and Implementation pp. 251-264 (2012)

[36] Cristian, F., Beijer , R., Mishra, S.: A Performance Comparison of Asynchronous Atomic Broadcast Protocols. J. Distributed System Engineering vol. 1-4 (1994)

[37] Cristian, F., Fetzer, C.: The Timed Asynchronous Distributed System Model. In: IEEE Transaction on Parallel and Distributed Systems vol. 10-6 pp. 642-657 (1999)

[38] Cristian, F., Mishra , S., Alvarez, G.: High-Performance Asynchronous Atomic Broadcast. J. Distributed Systems Engineering vol. 4-2 (1997)

[39] CSIM: Development Toolkit for Simulation & Modeling. (http://www.mesquite.com/) (accessed in Dec 2013)

[40] Dabholkar, A., Dubey, A., Gokhale, A., Karsai, G., Mahadevan, N.: Reliable Distributed Real-Time and Embedded Systems through Safe Middleware Adaptation. In: IEEE 31st Symposium on Reliable Distributed Systems pp. 362-371 (2012)

[41] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles vol. 41-6, pp. 205-220 (2007)

[42] Defago, X., Schiper, A.: Specification of Replication Techniques, Semi-Passive Replication and Lazy Consensus. In: IEEE International Symposium on Reliable Distributed Systems (1998)

[43] Defago, X., Schiper, A., Urban P.: Totally ordered broadcast and multicast algorithm: A comprehensive survey. In: ACM Computing Surveys vol. 36, iss. 4, pp. 372-421 (2004)

[44] Défago, X., Schiper, A.: Semi-Passive Replication and Lazy Consensus. J. Parallel and Distributed Computing Archive vol. 64-12 pp. 1380-1398 (2004)

[45] Dumitraş, T., Narasimhan, P.: Fault-Tolerant Middleware and the Magical 1%. In: Proceedings of the ACM/IFIP/USENIX International Conference on Middleware pp. 431-441 (2005)

[46] Ellis, C., Gibbs, S.: Concurrency Control in Groupware Systems. In: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data pp. 399-407 (1989)

[47] Elnozahy, E., Alvisi, L., Wang, Y., Johnson, D.: A Survey of Rollback-recovery Protocols in Message-passing Systems. J. ACM Computing Surveys vol. 34-3 pp. 375-408 (2002)

[48] Fault-tolerant CORBA Specification. In: Catalog of CORBA/IIOP Specifications (2005)

[49] Fekete, A., Lynch, N., Shvartsman, A.: Specifying and Using a Partitionable Group Communication Service. J. ACM Transactions on Computer Systems vol. 19-2 pp. 171-216 (2001)

[50] Felber, P., Schiper, A.: Optimistic Active Replication. In: ICDCS-21 Proceedings of the 21st International Conference on Distributed Computing Systems, pp. 333-341 (2001)

[51] Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Doctoral Dissertation (2000)

[52] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol -- HTTP/1.1 (http://www.ietf.org/rfc/rfc2616.txt) (1999)

[53] Fielding, R., Taylor, R.: Principled Design of the Modern Web Architecture. J. ACM Transactions on Internet Technology (TOIT) vol. 2-2, pp. 115–150 (2002)

[54] Fox, A., Brewer, E.: Harvest, Yield and Scalable Tolerant Systems. In: Proceedings of the Seventh Workshop on Hot Topics in Operating Systems pp. 174 (1999)

[55] Franks, G.: Performance Analysis of Distributed Server Systems. Doctoral Dissertation Carleton University (2000)

[56] Fromentin, E., Raynal, M., Tronel, F.: On Classes of Problems in Asynchronous Distributed Systems with Process Crashes. In: Proceedings of 19th IEEE International Conference on Distributed Computing Systems pp. 470-477 (2000)

[57] Glässer, U., Gurevich, Y., Veanes, M.: Abstract Communication Model for Distributed Systems. J. IEEE Transactions on Software Engineering vol. 30-7 pp. 1-15 (2004)

[58] Golding, R.: A Weak-Consistency Architecture for Distributed Information Services. Technical Report University of California at Santa Cruz (1992)

[59] Google Inc. Google Apps (http://en.wikipedia.org/wiki/Google_Apps) (accessed in 2013)

[60] Gopal, A., Strong, R., Toueg, S., Cristian, F.: Early-Delivery Atomic Broadcast. In: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing pp. 297-309 (1990)

[61] Gray, J., Helland, P., O'Neil, P., Shasha, D.: The Dangers of Replication and a Solution. In: Proceedings of the ACM SIGMOD International Conference on Management of Data pp. 173-182 (1996)

[62] Gross, D., Harris, C.: Fundamentals of Queuing Theory. J. 3rd Ed. John Wiley & Sons (1998)

[63] Guerraoui, R., Schiper, A.: The Generic Consensus Service. J. IEEE Transactions on Software Engineering vol. 27-1 pp. 29-41 (2001)

[64] Guerraoui, R., Schiper, A.: Transaction Model Vs Virtual Synchrony Model: Bridging the Gap. Theory and Practice in Distributed Systems Lecture Notes in Computer Science vol. 938 pp. 121-132 (1995)

[65] Gupta, I.: Building Scalable Solutions to Distributed Computing Problems using Probabilistic Components. Doctoral Dissertation (2004)

[66] Hadoop Apache (http://hadoop.apache.org/) (accessed in Dec. 2013)

[67] Hayden, M., Birman, K.: Probabilistic Broadcast. Technical Report Cornell University (1996)

[68] Helal, A., Heddaya, A., Bhargava, B.: Replication Techniques in Distributed Systems. Kluwer Academic Publishers (1996)

[69] Herlihy, M., Wing, J.: Linearizability: A Correctness Condition for Concurrent Objects. J. ACM Transactions on Programming Languages and Systems vol. 12-3 pp. 463-492 (1990)

[70] Holliday, J., Steinke, R., Agrawal, D., Abbadi, A.: Epidemic Quorums for Managing Replicated Data. In: Proceedings of Nineteenth IEEE International. Performance, Computing, and Communications Conference pp. 93-100 (1999)

[71] JSR 340: Java Servlet 3.1 Specification. (http://jcp.org/en/jsr/detail?id=340) (accessed in 2013)

[72] Jiminez-Peris, R., Patino-Martinez, M., Arevalo, S.: Deterministic Scheduling for Transactional Multithreaded Replicas. In: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems pp. 164-173 (2000)

[73] Jiménez-Peris, R., Patiño-Martínez, M.: Towards Robust Optimistic Approaches. In: Future Directions in Distributed Computing pp. 45-50 (2003)

[74] Jogalekar, P., Woodside, M.: Evaluating the Scalability of Distributed Systems. J. IEEE Transactions on Parallel and Distributed Systems vol. 11-6 pp. 589-603 (2000)

[75] José, P., Luís, R., Rui, O.: Enforcing Strong Consistency with Semantically View Synchronous Multicast. Technical Report University of Lisbon (2001)

[76] JSR 4: ECperf Benchmark Specification. (http://jcp.org/en/jsr/detail?id=4) (accessed in 2002)

[77] Karamanolis, C., Magee, C.: Client Access Protocols for Replicated Service. J. IEEE Transaction on Software Engineering vol. 25-1, pp. 3-22 (1999)

[78] Keidar, I.: Large-Scale Transaction Replication. In: ACM SIGACT vol. 44-3 Distributed Computing Column 51 pp. 72-72 (2013)

[79] Keidar, I., Dolev, D.: A Totally Ordered Broadcast in the Face of Network Partitions. Exploiting Group Communication for Replication in Partitionable Networks. In: Dependable Network Computing ch. 3 Kluwer Academic Publications (2000)

[80] Keidar, I., Khazan, R.: A Client-Server Approach to Virtually Synchronous Group Multicast: Specifications, Algorithms, and Proofs. In: Proceedings of the The 20th International Conference on Distributed Computing Systems pp. 344 (2000)

[81] Keidar, I., Sussman, J., Marzullo, K., Dolev, D.: A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. Technical Report University of California at San Diego (1999)

[82] Keleher, P.: Decentralized Replicated-object Protocols. In: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing pp. 143-151 (1999)

[83] Kemme, B., Alonso, G.: A New Approach to Developing and Implementing Eager Database Replication Protocols. In: ACM Transactions on Database Systems vol. 25-3 (2000)

[84] Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing Transactions over Optimistic Atomic Broadcast Protocols. In: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems pp. 424 (1999)

[85] Khazan, R.: A One-Round Algorithm for Virtually Synchronous Group Communication in Wide Area Networks. Doctoral Dissertation MIT (2000)

[86] Krishnamurthy, S., Sanders, W., Cukier, M.: Performance Evaluation of a QoS-Aware Framework for Providing Tunable Consistency and Timeliness. In: Proceedings of the International Workshop on Quality of Service (2002)

[87] Krishnaswamy, V., Ahamad, M., Raynal, M., Bakken, D.: Shared State Consistency for Time-Sensitive Distributed Applications. In: Distributed Computing Systems pp. 606-614 (2001)

[88] Krishnaswamy, V., Ganev, I., Dharap, J., Ahamad, M.: Distributed Object Implementations for Interactive Applications. In: Proceedings of Middleware '00 IFIP/ACM International Conference on Distributed Systems Platforms pp. 45-70 (2000)

[89] Kumar, V.: Performance of Concurrency Control Mechanisms in Centralized Database Systems. In: Prentice Hall 1st Edition (1995)

[90] Lamport, L.: Generalized Consensus and Paxos. In: Microsoft Research Technical Report MSR-TR-2005-33 (2005)

[91] Lamport, L.: The part-time parliament. J. ACM Transactions on Computer Systems vol. 16-2, pp. 133-169 (1998)

[92] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. J. Communications of the ACM vol. 21-7 pp. 558-565 (1978)

[93] Levine, B., Garcia-Luna-Aceves, J.: A Comparison of Reliable Multicast Protocols. J. Multimedia Systems vol. 6-5 pp. 334-348 (1996)

[94] Li, Z., Ma, X., Wang, Y.: Extension of RMI with Group Communication. Tech Report Cornell University (2005)

[95] Lynch, N., Gilbert S.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. P. ACM SIGACT News vol. 33, iss. 2 (2002)

[96] Malhis, L., Sanders, H., Schlichting, D.: Numerical Evaluation of a Group-Oriented Multicast Protocol Using Stochastic Activity Networks. In: Proceedings of the Sixth International Workshop on Petri Nets and Performance Models pp. 63-72 (1995)

[97] Maihöfer, C., Eberhardt, R.: A Delay Comparison of Reliable Multicast Protocols: Analysis and Simulation Results. Kommunikation in Verteilten Systemen (KiVS) Informatik aktuell pp 271-282 (2003)

[98] Wang, D., Mah, A., Lassen, S.: Google Wave Operational Transformation (http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html) (accessed in Dec. 2013)

[99] MATLAB (http://www.mathworks.com/help/matlab/) (accessed in Dec. 2013)

[100] MATLAB Box Plot (http://www.mathworks.com/help/stats/boxplot.html) (accessed in Dec. 2013)

[101] Mayer, E.: An Evaluation Framework for Multicast Ordering Protocols. In: SIGCOMM '92 Conference Proceedings on Communications Architectures & Protocols pp. 177-187 (1992)

[102] Memcached: A Distributed Memory Object Caching System (http://memcached.org/) (accessed in Dec. 2013)

[103] Mena, S., Schiper, A., Wojciechowski, P.: A Step Towards a New Generation of Group Communication Systems. In: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware pp. 414-432 (2003)

[104] Moser, L., Melliar-Smith, P., Narasimhan, P.: A Fault Tolerance Framework for CORBA. P. the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing pp. 150 (1999)

[105] Mostafa, W., Singhal, M.: Performance Analysis of a Reliable Multicast Session Protocol for Collaborative Continuous-Feed Applications. In: Proceedings of the 5th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems pp. 213 (1996)

[106] Narasimhan, N., Moser, L., Melliar-Smith, P.: Transparent Consistent Replication of Java RMI Objects. In: Proceedings of the International Symposium on Distributed Objects and Applications pp. 17 (2000)

[107] Narasimhan, P., Moser, L., Melliar-smith, P.: Strongly Consistent Replication and Recovery of Fault-Tolerant CORBA Applications. J. Computer System Science and Engineering vol. 2-1, pp. 20-33 (2002)

[108] Natarajan, B., Gill, C., Gokhale, A., Cross, J., Fernandez, S.: Adding Fault-Tolerance to Real-Time CORBA Systems. In: OMG Real-Time and Embedded DOC Workshop (2002)

[109] Nicola, M., Jarke, M.: Performance Modeling of Distributed and Replicated Databases. J. IEEE Transaction on Knowledge and Data Engineering vol. 12-4, pp. 645-672 (2000)

[110] Ongaro, D., Ousterhout, J.: In Search of an Understandable Consensus Algorithm. Stanford University (2013)

[111] Ozkasap, O., Xiao, Z., Birman, K.: Scalability of Two Reliable Multicast Protocols. Technical Report Cornell University (1994)

[112] Pedone, F., Schiper, A.: Optimistic Atomic Broadcast. J. Theoretical Computer Science - Special issue: Distributed Computing Archive vol. 291, iss. 1 (2003)

[113] Pedone, P., Schiper, A.: Generic Broadcast. In: DISC-99 13th. Intl. Symposium on Distributed Computing, pp. 94-108 (1999)

[114] Pereira, J., Rodrigues, L., Oliveira, R.: Reducing the Cost of Group Communication with Semantic View Synchrony. In: International Conference on Dependable Systems and Networks pp. 293-302 (2002)

[115] Peris, R., Martinez, M., Alonso, G., Kemme, B.: Are Quorums an Alternative for Data Replication? In: ACM Transactions on Database Systems, vol. 28, iss. 4 (2003)

[116] Pertet, S., Prof, A., Narasimhan, P., Reader, S., Koopman, P.: Proactive Fault-Recovery in Distributed Systems. Carnegie Mellon University (2004)

[117] Rodrigues, L., Fonseca, H., Veríssimo, P.: Totally Ordered Multicast in Large-Scale Systems. In: Proceedings of the 16th International Conference on Distributed Computing Systems (1996)

[118] Saito, Y., Shapiro, M.: Replication: Optimistic Approaches. Technical Report Internet Systems and Storage Laboratory HP Laboratories (2002)

[119] Slember, J., Narasimhan, P.: Living with Nondeterminism in Replicated Middleware Applications. In: Proceedings of the 7th ACM/IFIP/USENIX International Conference on Middleware pp. 81-100 (2006)

[120] Renesse, R., Birman, K., Friedman, R., Hayden, M., Karr, D.: A Framework for Protocol Composition in Horus. In: Proceedings of the fourteenth annual ACM Symposium on Principles of Distributed Computing pp. 80-89 (1995)

[121] Renesse, R., Birman, K., Maffeis, S.: Horus: A Flexible Group Communications System. In: Proceedings of the The 20th International Conference on Distributed Computing Systems pp. 344 (1995)

[122] Renesse, R., Hickey, K., Birman, K.: Design and Performance of Horus: A Lightweight Group Communications System. In: Cornell University Tech Report (1994)

[123] Rodrigues, L., Katherine, G., António, S., Renesse, R., Glade, B., Veríssimo, P., Birman, K.: A Dynamic Light Weight Group Service. J. Parallel and Distributed Computing vol. 60-12 pp. 1449-1479 (2000)

[124] Saltzer, J., Reed, D., Clark, D.: End-to-End Arguments in System Design. J. ACM Transactions on Computer Systems vol. 2-4 pp. 277-288 (1984)

[125] Schiper, A.: Early consensus in an asynchronous system with a weak failure detector. J. Distributed Computing archive vol. 10-3 pp. 149-157 (1998)

[126] Schiper, A.: Failure Detection vs. Group Membership in Fault-Tolerant Distributed Systems: Hidden Trade-Offs. In: PAPM-PROBMIV, LNCS 2399 (2002)

[127] Schiper, A., Urban, P., Urbán, P.: Comparing the Performance of Two Consensus Algorithms with Centralized and Decentralized Communication Schemes. Japan Advanced Institute of Science and Technology (2004)

[128] Schneider, F.: Implementing fault tolerant services using the state machine approach. A tutorial. J. ACM Computing Surveys vol. 22-4 pp. 299-319 (1990)

[129] Shapiro, M., Saito, Y.: Scaling Optimistic Replication. In: Future Directions in Distributed Computing pp. 164-168 (2002)

[130] Slember, J., Narasimhan, P.: Nondeterminism in ORBs: The Perception and the Reality. In: IEEE 17th International Workshop on Database and Expert Systems Applications pp. 379-384 (2006)

[131] Sousa, A., Pereira, J., Moura, R., Oliveira, R.: Optimistic Total Order in Wide Area Networks. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems pp. 190 (2000)

[132] SPDY Protocol. (http://www.chromium.org/spdy) (accessed in Dec. 2013)

[133] Strom, R., Banavar, G., Miller, K., Ward, M., Prakash, A.: Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects. J. IEEE Transactions on Computers vol. 47-4 pp. 458-471 (1998)

[134] Sussman, J., Keidar, I., Marzullo, K.: Optimistic virtual synchrony. In: Proceedings The 19th IEEE Symposium on Reliable Distributed Systems pp. 42 (2000)

[135] Tarashchanskiy, I.: Virtual Synchrony Semantics: Client-Server Implementation. Technical Report MIT (2000)

[136] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. P. ACM SIGOPS Operating Systems Review vol. 29, iss. 5 (1995)

[137] Thomasian, A.: Concurrency Control: Methods, Performance, and Analysis. J. ACM Computing Surveys vol. 30-1 pp. 70-119 (1998)

[138] Thomopoulos, E., Moser, L., Melliar-Smith, P.: Analyzing the Latency of the Totem Multicast Protocols. In: Proceedings of the 6th International Conference on Computer Communications and Networks pp. 42 (1997)

[139] Torres-Rojas, F., Ahamad, M., Raynal, M.: Timed Consistency For Shared Distributed Objects. In: Proceedings of the Eighteenth Annual ACM symposium on Principles of Distributed Computing pp. 163-172 (1999)

[140] TPC-E, an On-Line Transaction Processing Benchmark. (http://www.tpc.org/tpce) (accessed in Dec. 2013)

[141] Triantafillou, P., Taylor, D.J.: The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed Systems. In: IEEE Transaction on Software Engineering vol. 21-1 pp. 1-18 (1995)

[142] Trivedi, K.: Probability and Statistics with Reliability, Queuing, and Computer Science Applications. 2nd Ed. John Wiley & Sons (2002)

[143] Urban, P., Defago, X., Schiper, A.: Contention-Aware Metrics for Distributed Algorithms: Comparison of Atomic Broadcast Algorithms. In: Proceedings of the 9th IEEE International Conference on Computer Communications and Networks pp. 582-289 (2000)

[144] Urban, P., Hayashibara, N., Schiper, A., Katayama, T.: Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm. In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems pp. 4-17 (2004)

[145] Urban, P., Shnayderman, I., Schiper, A.: Performance Study of Two Atomic Broadcast Algorithms. In: Proc. International Conference on Dependable Systems and Networks pp. 645-654 (2003)

[146] Vicente, P., Rodrigues, L.: An Indulgent Uniform Total-order Algorithm with Optimistic Delivery. In: Proceedings of Reliable Distributed Systems pp. 92-101 (2002)

[147] Vogels, W.: Eventually Consistent. In: ACM Queue October 2008 (2008)

[148] W3C: HTML5 Specifications. (http://www.w3.org/html/wg/) (accessed in Dec. 2013)

[149] Weihl, W.: Commutativity-Based Concurrency Control for Abstract Data Types. J. IEEE Transactions on Computers vol. 37-12 pp. 1488-1505 (1988)

[150] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. In: Proceedings of the 5th symposium on Operating systems design and implementation vol. 36 pp. 255-270 (2002)

[151] Wiesmann, M., Pedone, F., Schiper, A.: A Systematic Classification of Replicated Database Protocols based on Atomic Broadcast. In: Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems (1999)

[152] Wilhelm, U., Schiper, A.: A Hierarchy of Totally Ordered Multicasts. In: Proceedings of the 14th Symposium on Reliable Distributed Systems pp. 106-115 (1995)

[153] Yu, H., Vahdat, V.: Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. J. ACM Transactions on Computer Systems vol. 20, iss. 3, pp. 239-282 (2002)

[154] Zhao, W., Moser, L., Melliar-Smith, P.: Unification of Replication and Transaction Processing in Three-Tier Architectures. Proceedings of the 22nd International Conference on Distributed Computing Systems pp. 290 (2002)

[155] Zhu, W.: RMI multicast and replication. JGroups – A Tool Kit for Reliable Multicast Communication (http://www.jgroups.org/javagroupsnew/docs/success.html) (accessed in Dec. 2013)

[156] Zhu, W.: Service-context propagation over RMI. Javaworld 01/17/2015 (http://www.javaworld.com/javaworld/jw-01-2005/jw-0117-rmi.html) (accessed in Dec. 2013)

[157] Zhu, W.: Tunable Optimism for Group-Communication-based Fault-tolerant Replication. Technical Report Systems and Computer Engineering Department, Carleton University (2007)

[158] Zhu, W., Jennings, M.: Implications of Full-Duplex HTTP. IETF Internet Draft (http://tools.ietf.org/html/draft-zhu-http-fullduplex) (2011)

[159] Zhu, W., Woodside, M.: Optimistic Open Groups for Fault-Tolerant Replication. In: The International Conference on Dependable Systems and Networks Fact Abstract (2006)

[160] Zhu, W., Woodside, M.: Optimistic Scheduling with Geographically Replicated Services in the Cloud Environment (COLOR). In: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) pp. 735-740 (2012)

[161] Zinky, J., Bakken, D., Schantz, R.: Architecture Support of Quality of Service for CORBA Objects. J. Theory and Practice of Object Systems vol. 3-1 (1997)