A LAZY DIRECTORY-BASED IMPLEMENTATION OF CONSISTENT REPLICATED STORAGE

BY

PHILBERT R. LIN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Nitin H. Vaidya

Abstract

Recent work in replicated datastores has focused on making availability and low latency the primary requirements. We present two directory-based datastores which make strong application semantics a primary requirement based on ideas from cache coherence in multiprocessors and distributed shared memory. The two datastores are implemented and evaluated against the Apache Cassandra distributed datastore in a simulated geo-distributed environment with workloads generated by the Yahoo! Cloud Serving Benchmark. We find that while there are cases where our systems perform with better average latency, the variability in the request latencies is undesirable due to expensive synchronization operations.

To my parents,

who have done more for me than I could ever do for myself

Acknowledgments

I would like to thank Dr. Nitin H. Vaidya for his continued support and guidance over the past year and a half. This work was supported in part by Boeing and I would like to thank them. Additionally conversations with Larry Kai, Adrian Djokic, Danyang Zhuo, Stephen Abraham and Ghazale Hosseinabadi have been helpful. I would also like to thank Brian Cho for teaching me how to effectively work with systems research. Last but not least, I would like to thank my family and friends for their continued support.

Table of Contents

Chapter 1

Introduction

With the rise of big data and cloud computing the amount of information that backend infrastructure must handle with high performance is growing rapidly. The issue of replicated data storage has received increasing attention in recent years [1]–[12].

A replicated datastore consists of multiple physical copies of an object and logically one or more copies of application clients. Replication can improve performance, availability and fault-tolerance at the cost of maintaining those replicas. The most concise characterization of the complex design trade offs in any replicated storage system was described by Brewer [13] and is known as the CAP theorem. The CAP theorem says that a system can simultaneously provide at most two of the following three attributes: consistency, availability, and tolerance to network partitions.

Consistency in this context refers to whether or not the replicas are all matching. Availability describes how often the data can be read or written to. And tolerating a network partition means that the system can continue to function even if part of the system cannot be contacted or fails.

Systems were designed to tolerate network partitions and prioritize either consistency [1, 2] or availability [3]–[7]. Although the CAP theorem stated that systems could only have two of the three properties, it was clarified in [14] that the former statement is misleading. Systems *can* provide all three, but not 100% of each trait simultaneously. Other systems were developed which provided a better balance [8]–[12]. Recent systems have also started looking at the problem of *geo-replicated* storage [2], [9]–[12]. Geo-replicated storage is when objects are replicated across datacenters located over many geographic locations. This is also known as *wide-area* replication.

Besides the issues covered by the CAP theorem, there are other factors involved when we consider building a highly scalable, distributed replicated datastore. Some of these issues include application semantics, system coordination, how the updates are propagated through the system, how and where data is logically and physically placed, and how to handle fault-tolerance. These additional factors will influence the CAP characteristics and performance of the system.

These recent systems have been looking at making the worst case scenario as best as possible, attempting to minimize the latency that 99.9% of requests experience [3]. Typically this has been done by compromising the consistency of the system. In this thesis we discuss potential ways to maintain strong consistency semantics to the client while also keeping the *average* of the request times as low as possible in a geo-replicated setting. We present and evaluate two different methods using ideas from cache coherence in multiprocessors and distributed shared memory.

The remainder of the thesis is organized as follows. In Chapter 2, we present a survey of the existing literature on replicated datastores structured around how each of the core issues are addressed. Chapter 3 presents related background ideas from multiprocessor cache coherence, distributed shared memory, and details on the Cassandra datastore. In Chapter 4, we present the design of two directory-based datastores. In Chapter 5, we evaluate the two directory-based datastores along with Cassandra. Finally in Chapter 6 we conclude the work and discuss future work.

Chapter 2

Related Work

There has been ample work in the theory and building of replicated datastores. This chapter will describe how various systems have tackled the problems that distributed replicated datastores face. The core issues are what semantics to present to the application, which consistency model to use, how should the system coordinate among its replicas, how should updates be propagated through the system, how and where data is logically and physically placed, and how faults are handled.

2.1 Application Semantics

The main purpose of a datastore is to allow a client to read and write persistent information. There are various ways of presenting this interface to the client and here we present a few. In order of increasing complexity:

- **Key-Value:** The simplest abstraction is a key-value store, similar to a map data structure. A client provides a key and value when writing, and then can retrieve the value with a key. This abstraction does not have logical order or grouping to it but keeps the design simple.

- **File System:** A file system, similar to one encountered on a modern operating system, provides a hierarchy of files which can be manipulated. In a simplified sense, a file system is a key-value store with the path and name of a file as the key and the contents as the value. Although a file system provides additional organization, it does not lend itself naturally to manipulating multiple objects.

- **Database:** A database provides an easier interface to manipulate structured data. Typically a query language is used to semantically express what information the client would like to read or write. A common type of database is a relational database which emphasizes the relationships between records.

None of these abstractions need to be implemented in a distributed or replicated fashion, but doing so increases the performance and fault-tolerance of the system at the cost of increased complexity.

Amazon's Dynamo [3] is primarily a key-value store. The Google File System (GFS) [6] is not surprisingly a file system, but departs from the standard POSIX API to improve performance. Some of the interesting design choices were to make the default block size 64MB and make most of the files append-only. PNUTS [1], BigTable [5] and Cassandra [4] are in between a key-value store and database. In BigTable and Cassandra clients can access their data with multiple attributes. Data is primarily indexed by a row key but clients can also optionally specify which column they would like to access. Each row has various column elements. Google's Spanner [10], Megastore [2] and recent versions of Cassandra are most similar to a relational database and support SQL-like queries. The additional functionality and overhead of maintaining structure of the data may come at the cost of lower performance. Many applications only need a datastore similar to a key-value store [3], however, as application complexity grows the benefit from having a richer data model increases as well [10]. Systems can layer together these abstractions. One example is that BigTable is built on top of GFS.

Ideally when data is replicated the client is unaware of the fact. It is most intuitive for an application when there is one logical copy of the data and one or more physical copies. However, to improve performance the system can ask the application for assistance regarding the multiple physical copies, thus increasing the number of logical replicas. This is necessary during optimistic operations when state diverges and the application has more knowledge of how to recognize and resolve conflicts. This technique is used in Bayou [7]. A less intrusive method of giving the application is allowing the client to specify a timestamp. This method is used by Spanner, BigTable, and PNUTS. This approach lets applications potentially receive a stale copy of the data, but it will be correct in the version history unlike what could happen in an eventually consistent system where reads may return inconsistent values [1].

2.2 Consistency

Consistency of data in a distributed system is an inherent problem due to occasional ambiguity in the ordering of events between processes [15]. In order to correctly keep each replica consistent, the concurrent operations on them need to be ordered. Another option is to relax the ordering and allow the state to diverge. While there are many consistency models, we will only cover a few here. Ordering them approximately from strongest to weakest:

- **Linearizability:** also known as atomic consistency, is the strongest guarantee that a replicated system can provide. It guarantees 1) the system's execution of the operations is consistent with the real finishing times and 2) any read of an object will occur after its corresponding write in the interleaving [16]. Linearizability provides the illusion that each operation takes effect instantaneously at some point between its invocation and its response [17].

- **Sequential Consistency:** is a slightly weaker version of strong consistency. It approximates linearizability and guarantees 1) the system will be in the same state for any interleaving of individual processes and 2) any read of an object will occur after its corresponding write in the interleaving [16]. It is similar to linearizability but without the real finishing time constraint. One way to intuitively understand sequential consistency is to imagine that the shared memory can only be connected to one of the multiprocessors at any time. All processors see the same memory state and have their actions completed in the order they specified.

- **Causal Consistency:** maintains the ordering of relative operations which are *causally related* [18]. In this model reads respect the order of causally related writes. An operation is potentially causal with 1) previous operations performed by the same thread of execution, 2) operations that wrote the value this operation has read, and 3) operations which are causally after an operation from rule 1) or 2) [11]. One weakness of how causal consistency is typically implemented is that operations can only be ordered based on the actions observable by the system.

- **Eventual Consistency:** is a form of weak consistency where if there are no new updates to an object, then eventually all replicas will have the same copy [19]. Although an upper bound to the inconsistency window can be approximated for a system, this model does not dictate one. In this model updates can reach replicas in different orders.

The consistency model and guarantee of a system depend on how and when the data is replicated from one server to another. The data can be replicated either synchronously or asynchronously whenever there is a client request. Synchronously replicating data requires that a group of servers will agree on a value before replying to the client while asynchronously replicating data takes that action off the critical path, resulting in better performance.

2.2.1 Strong Consistency

Megastore uses Paxos to replicate user data across datacenters for every write to achieve snapshot consistency, which provides sequential consistency within an entity group (a collection of data) [2]. Paxos [20] is a well known consensus protocol. It is an expensive operation consisting of a couple rounds of messages to a quorum, but it is currently the best known technique to achieve consistent fault-tolerant ordering between a set of replicas. Although many systems use Paxos to synchronize values, Megastore was the first at the time to use it to replicate data on the critical path in wide-area storage. MDCC [12] also uses Paxos and a few of its extensions such as Generalized Paxos.

Spanner similarly uses Paxos during client updates, however by using extra hardware (atomic clocks and GPS) the system is able to globally order operations *across* datacenters with even better performance. The extra precision from their TrueTime API allows the timestamps to be useful globally, thus achieving linearizability. Additionally, Spanner provides consistent read-only snapshots of the system at any previous timestamp.

Azure [8] provides strong consistency at an object granularity by coupling their stream layer and partition layer together. The append-only stream layer provides high availability while the partition layer asynchronously replicates the data. The stream manager within the stream layer uses Paxos, but mainly for fault tolerant purposes.

PNUTS [1] achieves per-record timeline consistency (sequential consistency for each record) by designating one of the replicas of an object the master and funneling all of the write operations through it. The system allows for reads off any of the replicas if stale information is acceptable, otherwise the master is contacted for the latest update.

Azure, PNUTS, and BigTable provide strong consistency within a datacenter but Spanner with TrueTime and Paxos is able to totally order operations across datacenters.

2.2.2 Weak Consistency

Bayou [7] allows a client to write to the system in almost all conditions, even when the client is off-line and completely partitioned from the rest of the system. However, this design can lead to many conflicts. Bayou tackles this problem by presenting the application with an interface to detect and merge the conflicts. Another limitation of their design is that previously written updates may be modified by the system before the data is labeled committed.

Dynamo [3] and Cassandra [4] use a quorum-like technique whenever updates or reads are made. Clients can configure the number of replicas which must synchronously respond to reads or writes. Cassandra calls this tunable consistency. If we let N be the total number of replicas, W be the number of replicas which acknowledge a write, and R be the number of replicas that are accessed on a read operation, then setting $W + R > N$ results in strong consistency while $W + R \leq N$ results in eventual consistency. However in the background there is asynchronous replication to reach the other N nodes in charge of the data.

Eiger [11] uses causal consistency in order to achieve better performance in a geo-distributed setting. All operations from clients are served from the local datacenter from which they arise. Operations are asynchronously pushed to remote datacenters but committed only after all causally dependent operations have been previously committed.

2.2.3 Mixed Consistency

Gemini [9] takes a different approach and categorizes operations as either red or blue. Each color has a different consistency guarantee, but both are executed together in the system. Blue operations are executed locally and quickly while the red operations require the coordination of other servers.

2.2.4 Consistency Discussion

Systems such as PNUTS, Megastore, Azure, and Spanner provide stronger consistency but do not perform as well as systems which provide weaker consistency guarantees such as GFS, Dynamo, and Cassandra.

The stronger consistency stores become unavailable if the network becomes partitioned in order to preserve the consistency among the replicas. If some of the replicas were able to proceed then the data could diverge. This unavailability is the cost of keeping the state consistent. However since a system like Spanner is spread across datacenters, it is unlikely that a majority of the datacenters will go down and prevent the system from making progress.

The performance benefits from eventual consistency come at a cost of less intuitive semantics and increased application concern. The optimistic replication allows state to diverge and lead to conflicts. Bayou requires the application to define how to both detect and merge conflicts, while Gemini also requires modifications to the application to support the classification of operations. Dynamo uses vector clocks to detect conflicts but prefers client side resolution. Replication is no longer transparent to the application since it needs to prepare itself to receive different copies of the data. General methods of resolving conflicts include letting the "last-writer win" based on a timestamp or requiring manual intervention.

Note that some centralized designs are meant for replication within a datacenter while others are meant to take care of replication between datacenters. Typically the intra-datacenter system is built on top of the inter-datacenter system as in the case of BigTable and Megastore, Colossus (the successor of GFS) and Spanner, and the two layers of Azure. Typically the intra-datacenter layer provides a weaker consistency model but higher availability while the inter-datacenter layer provides stronger consistency guarantees.

2.3 System Coordination

Section 2.2 glossed over the details about how the replicas were asynchronously updated and how servers communicated among each other. In this section we discuss how the information is disseminated throughout the system. This is challenging in an asynchronous system, such as over the Internet, because it is impossible to tell whether a replica is slow, or has failed [21].

Bayou, Dynamo, and Cassandra use Gossip-like protocols in order to spread data such as the current membership view, detect failures, and update replicas in the background [3, 4, 7]. A Gossip-like protocol will occasionally send information to a random partner to reconcile their state. The advantage of gossiping is high availability and scalability. The architecture avoids a single point of failure while allowing each replica to contact a few others without each replica having a consistent membership view. This method of communication contributes to the eventually consistent nature of the systems.

PNUTS uses Yahoo! Message Broker (YMB) as a publish and subscribe (pub/sub) system to accomplish asynchronous replication while the master based system for each record takes care of synchronous replication. The pub-/sub system works by replicas publishing updates to YMB and then letting YMB deliver the update to all the subscribers. PNUTS chose to use the pub/-sub system over gossip because it can be optimized for geographically distant replicas and the replicas do not need to know the location of other replicas [1]. The broker can maintain a consistent membership view with occasional heartbeat messages. However, using a single broker for each datacenter introduces a single point of failure and potential performance bottleneck even though it is off the critical update path.

BigTable and Megastore both use Chubby to maintain a consistent view of the group [22]. Chubby is a distributed lock service based on Paxos. BigTable also uses Chubby for any coordination needed between replicas (such as leader election) and to store metadata such as access control lists. Megastore is also built on top of BigTable, so any communication used by BigTable is inherently used by Megastore. This extra layer makes Megastore slower than a system like Spanner which is only built on top of a file system.

GFS has a master node in charge of information about the chunk servers which hold data [6]. It keeps track of data locations with regular heartbeats and the data is spread through the system sequentially from one chunkserver to another.

The main difference between the design choices here is having a completely distributed architecture vs. having a centralized server in a hierarchy. When information is completely decentralized and lazily spread with gossip the system is highly available but not suited for real-time applications [23]. Funneling requests through a master or using a consensus protocol like Paxos allows the system to order the request among replicas and keep more consistent state.

2.4 Data Partitioning and Load Balancing

Another key issue is determining where to place the data across the replicas so that certain replicas do not become overburdened and slow down the system. Knowing where data is located is also an important aspect to request routing.

2.4.1 Distributed Hash Table

Dynamo and Cassandra both use a variant of consistent hashing to partition the key space [3, 4]. The output to a hash function can be treated as a position on a circle. Servers are assigned a position on the circle and designated the primary for all the keys that fall clockwise between its position and the next one. Given an object's key the corresponding replicas in charge of the value can be found in constant $O(1)$ time. In Dynamo however, servers are assigned multiple positions ("tokens") and treated as virtual nodes along the circle to evenly balance the load while Cassandra reassigns the position of lightly loaded nodes to resolve the same problem. In the current systems, the load balancing is done manually by sending commands to the replicas.

2.4.2 Centralized Master

PNUTS, GFS, BigTable, Spanner and Azure have centralized servers which keep direct mappings of the data to the server. They occasionally load balance according to metrics such as disk space and traffic. This process is automated unlike what is the current norm when using consistent hashing.

Having a single master allows the system to make globally optimal decisions when choosing data placement and rebalancing, although it introduces a potential bottleneck in performance. The bottleneck is typically alleviated by introducing another layer of servers that serve client requests while the master holds metadata.

2.4.3 Physical Location of Data

To improve performance, data can be placed to take advantage of 1) data locality, 2) geographic locality, and 3) network topology.

Spanner allows the application to configure which datacenter holds its data, how far data is from the users (affecting read latency), how far replicas are from each other (affecting write latency) and how many replicas there are [10]. Additionally, clients can define locality relationships between multiple tables so that Spanner knows what data is typically accessed together [10].

Another consideration to the physical placement of replication is fault-tolerance to natural disasters and other incidents such as a switch going off-line. GFS creates three replicas for each object. One in the same server, one in the same rack, and then one in another rack [6] while systems such as Spanner focus on spreading the data across datacenters.

2.5 Fault-Tolerance

Fault-tolerance refers to the system's ability to function correctly even in the presence of faults. Failures are treated as the norm due to the underlying commodity hardware that many of these systems use [6]. However there is a separation of concerns. Lower levels can primarily address the common hardware faults while higher levels can be abstracted away from those problems. One example of the lower level is GFS, while the higher level is BigTable.

Fault-tolerance is inherently provided by having multiple replicas of the data. The systems which use Paxos can tolerate up to f failures in $n = 2f+1$ where n is the total number of replicas. Typically $f = 1, n = 3$.

The number of faults that Dynamo and Cassandra can tolerate depend on the values of W, R and N as discussed in Section 2.2.2. They optimize for high availability by using hinted handoff and sloppy quorums during temporary failures [3]. If a replica does not respond immediately, another replica may hold a “hint” until it recovers and still quickly respond to the client [3].

To recover from crash failures, Dynamo, Cassandra and PNUTS synchronize the replicas as discussed in Section 2.3. GFS, BigTable, and Azure use replicated commit logs to replay the operations and bring the replica up to date.

2.6 Summary

Depending on the factors that the system wants to optimize for, there are many different design choices available. Spanner, Azure, Megastore and MDCC have shown that keeping data consistent with Paxos is an option that performs well while Dynamo and Cassandra make availability and performance the primary concern. Eiger manages to find a good balance point between performance and consistency. Systems such as Bayou and RedBlue show that giving the application more responsibility and not making the replication transparent can lead to improved performance as well.

Chapter 3

Background

This chapter will present related background material from cache coherence in multiprocessors and distributed shared memory. Then Cassandra is covered in more detail. These components contribute to the design and understanding of the systems introduced in Chapter 4.

3.1 Cache Coherence and Distributed Shared Memory

Caching is a form of replication typically used to improve performance in systems, specifically to reduce the latency of data access. Multiprocessors have had to deal with keeping their caches and memory consistent with each other for a long time. There are numerous techniques for doing so. The protocols for cache coherence depend on whether the underlying network is a bus supporting broadcast messaging or a different architecture which only supports message passing [24]. Snoopy protocols are used for broadcast networks while directory-based protocols are used when message passing is used. Blocks within the cache are typically associated with a state so that the protocol knows how to handle requests appropriately. An example is the MSI protocol [25]. The protocol defines the *Modified*, *Shared*, and *Invalid* states. These states indicate to the processors which blocks are clean or dirty and can be used to fulfill the current request or need to contact main memory or other cache controllers.

Multiprocessors have also benefited from having shared virtual memory when their physical memory is distributed. This notion is known as *distributed shared memory* (DSM). [26] simulated shared virtual memory over physically distributed memory. [26] uses a manager to maintain memory coherence. The manager in this context is very similar to a directory in directory-based cache coherence. The manager keeps track of the *state*, *owner*, and *copy_set* of a page. They considered both a centralized and distributed manager taking care of either statically located pages, or dynamically located pages. The main idea to satisfying coherence is that a processor is only allowed to update data when no other process is reading or modifying it, however, multiple processors can simultaneously read data.

The DASH shared-memory multiprocessor [27] used a directory-based cache coherence protocol in hardware, and also introduced the notion of *release consistency*. The idea is that memory that is within a critical section does not need to be synchronized with other processors until the end of the region since other processors may not access that memory simultaneously. This section is surrounded by a call to two synchronization operations, `acquire()` and `release()`. [28] implemented release consistency in software for the Munin distributed shared memory system. However, they were the first system to support *multiple* consistency protocols. Previous systems only supported sequential consistency. Munin allows programmers to annotate all shared variables with their expected access pattern to improve performance. [29] reduced the number of messages and amount of unnecessary data exchanged by using a lazy synchronization method (also known as optimistic replication). They propagate updates at `acquire()` rather than `release()`.

Maintaining consistency across a wide area network (WAN) in a distributed system has very similar issues to those faced in memory coherence and DSM. In Chapter 4 we will use similar ideas to those presented here to design systems which present strong application semantics.

3.2 Cassandra

The Cassandra project was originally started by Facebook as a way to power their inbox search [4] and was eventually opened sourced to the Apache foundation. It is a distributed datastore without a single point of failure and meant to provide high availability in the face of failures while being highly scalable. The backend architecture is very similar to Amazon's Dynamo while the data model is similar to Google's BigTable. This section will give a brief overview to better understand Cassandra and how client requests are processed.

3.2.1 Data Model

Cassandra uses the column-family data model [4, 5]. This is a multidimensional map indexed by a key. We will refer to this data structure as a *row*. The first mapping is from the row key to a set of named *column families*. The next layer of mapping is from each column families to another set of named columns, each of which may optionally have a value. An additional *super column* mapping can be added to wrap a set of columns. The column-family data model provides more structure than a simple key-value store and can be used to effectively describe the information used by many applications such as social networks to a web crawler [5, 11].

3.2.2 Client API

To manipulate the data within Cassandra, clients can use the Thrift interface or the Cassandra Query Language (CQL). This section will talk about the Thrift interface. Thrift provides remote procedure calls (RPC) to read, write and query system state. A client can read data by using `get` to retrieve a single value from a column while multiple columns and rows can be accessed with `get_slice` or `multiget_slice`. `insert` and `batch_mutate` provide similar functionality for updates. Cassandra does not differentiate between inserts and updates, both are treated the same way. More information can be found at [30].

3.2.3 System Architecture

Each node in Cassandra operates as a coordinator for client requests in a logical ring. There is a logical ring for each datacenter. Rows are stored based on a consistent hash function which allows for each node to determine which node is responsible for that row. The replication factor of the row is configurable based on the reliability, availability and storage costs desired. Let the number of total copies of the row be N. For each request, clients can choose a consistency level depending on the values W and R, the number of replicas which must respond to a write and read request respectively. When a request comes to any node, this coordinator node ensures that W or R replicas have responded before replying to the client. If $W + R \leq N$ then the system will be eventually consistent with concurrent operations being ordered by a timestamp provided by the client. If $W + R > N$ then the system will be strongly consistent. This is a quorum system which guarantees that there will always be an up-to-date replica common in any two requests. Regardless of the values of W or R, all requests are sent to all replicas, the values only represent the number which must have responded to proceed in the operation.

Instead of specifying the actual number for W and R, Cassandra offers a few consistency levels such as: `ONE` and `QUORUM` and additionally `LOCAL_QUORUM` and `EACH_QUORUM` when there are multiple datacenters. A quorum is defined as $\lfloor RF/2 \rfloor + 1$ where RF is the replication factor for a datacenter. When an operation is set to `LOCAL_QUORUM` only the quorum from that datacenter must respond to the coordinator before returning to the Client, while `EACH_QUORUM` must wait for a quorum from *each* datacenter. Currently the best way to guarantee strong consistency across multiple datacenters is to write to `EACH_QUORUM` while reading from `LOCAL_QUORUM`.

Figure 3.1 shows an example when $N = 3$, $R = 1$ and $W = 2$. Note that this configuration only guarantees eventual consistency. The Client issues a read request to node 7 (shown as step 1 in Figure 3.1). Node 7 is now the coordinator node which forwards the request to all replicas based on the distributed hash table (shown as step 2). Since R is 1, the coordinator responds back to the client after only one reply from node 5 (shown as step 3 and 4). If the client issued a write request then the coordinator would respond back to the client after two replies.

When Cassandra is configured to operate across datacenters, there are additional parameters the system must take care of. The system allows the user to adjust the number of replicas *per* datacenter, along with W and R for local or remote datacenters. Figure 3.2 depicts this situation when the replication factor is three for both datacenters and `EACH_QUORUM` must respond for the coordinator node to proceed with the client write request.

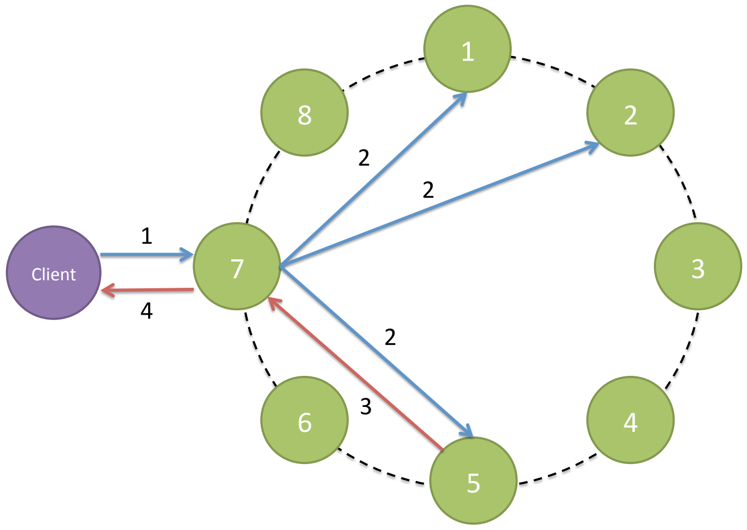

Figure 3.1: Cassandra client request within a datacenter. Figure is based on DataStax client request documentation.

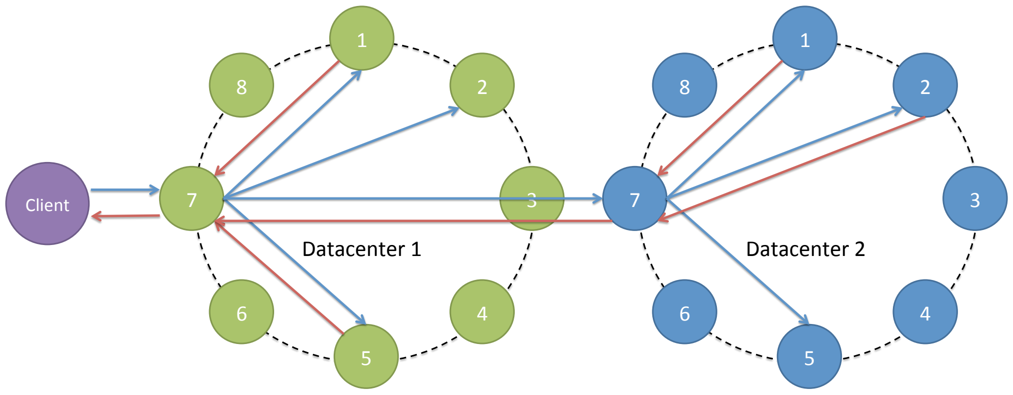

Figure 3.2: Cassandra client request across datacenters. Figure is based on DataStax client request documentation.

Chapter 4

System Design and Implementation

This chapter will describe the goals of our system design and then talk about the various design choices made before going into more detail about two directory-based system designs. One uses a directory-based scheme similar to directory-based cache coherence [31] to achieve strong consistency. The other uses the directory to achieve release consistency [27]. Both systems manage consistency using a two-level hierarchy in order to provide strong application semantics. The chapter will conclude with implementation details.

4.1 Goals

The primary goal of our datastore is to provide strong application semantics while keeping client requests as low-latency as possible in a geo-replicated environment. We use strong semantics here to denote that a client should read the most recent write to the shared object when using the system correctly.[1] While recent datastores have focused on optimizing the latency times for a single request for a client, a directory-based approach can lead to lower *average* request completion times when contention on a shared object is not frequent. Scalability is treated as a secondary goal in our work, although various design choices can be altered to provide better scalability at a cost of other parameters.

To summarize, our goal is to construct a datastore that presents clients with strong semantics and can fulfill client requests as quickly as possible while a directory maintains consistency of the objects across datacenters. Within datacenters we assume there exists a linearizable storage layer that we have access to which takes care of intra-datacenter consistency. The Di-

[1]In the presence of concurrent operations, a read can see the most recently completed write or a concurrent write.

rectory and the Frontends work together to maintain inter-datacenter consistency.

4.2 Directory-Based Strong Consistency

Typically datastores have kept their data coherent by keeping the actual object strongly consistent and using pessimistic replication. Instead of keeping the actual data consistent between all of the replicas we use a directory-based approach [31] to keep the *metadata* always up-to-date. The location of the most recently written copy is the key metadata to track so that future reads can be directed to it. There are various design choices to make when designing a system based around a directory-based consistency protocol. The most important factors are:

1. Object states
2. Update propagation
3. Centralized or distributed directory
4. Object granularity

The next few subsections will go into more detail about how each design choice was addressed in order to meet our goals from Section 4.1.

4.2.1 Object States

The states of the shared objects in the datastore are similar to states in the MSI cache coherence protocol [25]. Objects can be either *Modified*, *Shared* or *Invalid*. Below we define what each state signifies:

- **Modified:** denotes that this copy of the object is the only valid copy of data because state has been changed but not yet updated to the other replicas. When a replica has this state it can be considered the owner of the object. After synchronizing the object with other replicas the state will return to *Shared*.

- **Shared:** denotes that there are multiple valid copies of this row in the system. A replica can consider this a read-only copy and can serve read requests locally.

- **Invalid:** denotes that the current object is out of date and needs to synchronize with the most up to date copy. A possible choice is to allow a client to request stale data, however, we do not consider that here.

4.2.2 Update Propagation

Updates can be propagated to other replicas through either write-update or write-invalidate in order to maintain strong consistency in a DSM system [16]. A write-update approach is similar to how Cassandra handles write requests when $W = N$.[2] This write-update approach is pessimistic in the sense that other replicas will update their copy regardless of there being a demand to read that object. This adds extra latency to the completion of the request and is unnecessary when another write comes subsequently before any client reads the other copies.

Instead we use a write-invalidate approach with lazy replication to propagate the changes across replicas. This is beneficial in our context of geo-replicated storage. When a datacenter is performing operations on an object without concurrent accesses from other datacenters, there is at most one invalidation message sent out. The rest of the operations can be performed locally to the datacenter. This will drastically reduce latency of every operation at the cost of higher initial access times when a remote datacenter does want to access the object.

4.2.3 Centralized or Distributed Directory

As mentioned in Chapter 2 many systems have chosen to make their components as distributed as possible. This is typically done to remove the single point of failure and allow for the scalability of the system. However, this complicates other aspects of the system. Since scalability is not our primary

[2]Refer to Section 3.2.3 for more detail.

concern, we have chosen to use a centralized directory to manage the metadata. Using a centralized directory has various benefits when considering 1) data partitioning 2) consistency guarantees and 3) simplifying the design.

If the directory was distributed we would have to deal with properly partitioning the data across them. Methods such as consistent hashing used in [3, 4] are not ideal for this situation because of the geo-distributed nature of the datacenters. A replica looking to fulfill numerous requests may have to contact the numerous directories and face variables delays resulting in unpredictable performance. By using a centralized directory the latency to the directory will remain consistent and the replica could even batch numerous objects' requests into one message.

Another benefit of a centralized directory is that we can be more flexible with the consistency guarantees that the system offers. Since we are serializing the operations through a single logical point, we can achieve linearizability if desired or relax the ordering to provide a weaker model.

Lastly a centralized directory simplifies the design. Simplicity should not be overlooked since it eases the pain when having to implement, debug, and reason about normal and edge case behavior.

4.2.4 Object Granularity

Our system uses the column-family data model[3] used by Cassandra, Apache's HBase and Google's BigTable [4, 5]. We choose to use the row key within one column family as the granularity that the metadata tracks. This level of granularity is similar to the decision made in [1] and maintains the appropriate balance between fine granularity and performance overhead.

4.2.5 System Overview

Figure 4.1 shows an overview of the system architecture. We imagine a Frontend and Storage component representing a datacenter.[4] The underlying Storage components are reliable and strongly consistent replicated datastores such as Apache Cassandra or Apache HBase. These Storage com-

[3] Refer to Section 3.2.1 for more detail.

[4] The terms Frontend and datacenter will be used interchangeably when referring to the process that a Client contacts.

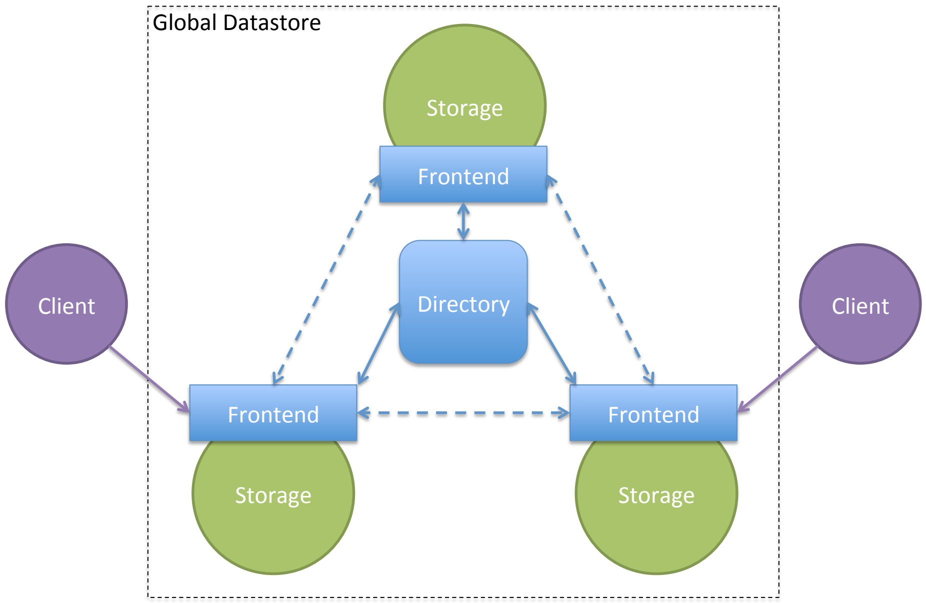

Figure 4.1: Overview of system architecture

ponents maintain consistency within the datacenter and among the Clients who contact the same Frontend. The Frontend components are processes which wrap the underlying datastores and coordinate with the Directory to maintain strong consistency across datacenters. Frontends will message each other when they need to synchronize data. They are also the only processes which interact with Clients. The Directory is the central entity which tracks the metadata of all the objects in the system.

4.2.6 Directory State

The centralized Directory has to hold the metadata for every object in the system. The Frontends need to query the Directory whenever they are unsure whether they can access the shared object. It acts as a serialization mechanism which enforces strong consistency among datacenters, however, no data flows through the Directory. The Directory provides pointers to other Frontends whom they request to synchronize the data from. This process is also responsible for invalidating copies of the data, although this logic could be pushed to the requesting Frontend. The **object state** and **list of replicas** is maintained for each row key. Each replica in the list is a Frontend associated with a Storage component. The replicas in the Directory metadata do not refer to the actual replicas that hold the data at the Storage level, but only to the Frontends that are responsible for each datacenter.

4.2.7 Frontend State

The Frontend processes each hold a local directory with entries for keys that are in their Storage component and their current state. Based on the local state of the key, they will know whether they need to contact the Directory or not. The Frontends act as a serialization mechanism for each datacenter since each Client only communicates through them. This is sufficient for strong consistency because the system always maintains up-to-date meta-data. Unlike the Directory, the Frontends do not need to know about all of the keys in the system. Additionally the Frontends do not need to keep track of the replica list, unless we want them to take care of the invalidation.

4.2.8 Client Requests

Clients only contact Frontend processes. We assume they will contact the geographically closest one. The Frontend exposes an interface similar to a subset of Cassandra's 1.0 Thrift interface. Essentially this allows the Client to read and update the columns of a row when given the row key. More specifically we support the following operations: `get`, `get_slice`, `insert`, and `batch_mutate`. One limitation of our operations is they currently operate only on one key at a time. This is different from Cassandra's interface which allows for multiple keys to be accessed at a time (however, the modifications are not atomic or transactions).

To discuss the roles that the Directory and Frontend have during the request protocol we will simplify the discussion to `get` requests which encapsulate all reads and `put` requests which function as inserts and updates.

Get Requests: When a Client wishes to read a key it will send the request to a Frontend. The Frontend checks the local state and will return the value from his Storage component if the local state is either *Modified* or *Shared*. If the state is *Invalid* then the Frontend will need to request access from the central Directory and block until receiving a response. On a successful response the Frontend will synchronize with a replica given from the Directory, write the value into his local Storage component and return to the Client. The Frontend must use a lock locally for each row in order to prevent different Clients from seeing different row states at this Frontend. Algorithm 4.1 gives the read pseudocode for the Frontend.

Algorithm 4.1 Frontend code for Read requests

```
procedure READ on receipt of $get(Row)$ from Client
    lock(local directory[Row])
    local state := local directory[Row]
    if local state = Modified or Shared then        ▷ Value is up-to-date
        return Value from Storage component[Row]
    else if local state = invalid then
        send request $(Row, Shared)$ to Directory
        block until response $(success, replica)$ from Directory
        if response = success then
            local directory[Row] := Shared
            send row synchronize message $(Row)$ to replica
            block until response (new value) from replica
            perform write to local Storage component[Row]
            return Value from Row
        else
            throw failure exception
        end if
    end if
    unlock(local directory[Row])
end procedure
```

When the Directory receives a *shared request* from a Frontend one of three events may happen. At the end of normal behavior the first two possible events result in the row state being changed to *Shared* and the requesting Frontend being added to the replica list.

- If the state at the Directory is *Shared* then the requesting replica can be added to the list of replicas without needing to contact other Frontends. The Directory would then send back a message indicating its success along with a Frontend replica that it needs to synchronize with and get the data from.

- If the state at the Directory is *Modified* then the behavior is similar to the *Shared* case except that an *invalidation message* is sent to the current owner of the row telling it that it should reduce the state of the row in its local directory from *Modified* to *Shared*. The Directory will need to block until an ACK is received.

- If the state at the Directory is *Invalid* or there is not yet an entry for the row then the entry has either been deleted, or was never in the datastore. The Directory can respond with a failed response and note that this entry does not exist.

When there is a concurrent request to the row, the Directory will fail the later requests in order to avoid a deadlock at the later requesting Frontends. The *Processing* state allows the Directory to keep track of when concurrent accesses occur. The Frontend blocks on the request to the Directory, but also requires exclusive access to the row when handling the invalidation message from the Directory. The failed request will allow the Frontend to begin processing other operations on that row, such as the invalidation, which allow other Frontends to proceed. The Frontend who has failed can attempt to access the row again afterwards until it succeeds. Algorithm 4.2 gives the Directory pseudocode for both read and write requests.

Put Requests: When a Frontend receives a write request from a Client the behavior is very similar to a `get` request. The only difference is that the data synchronization does not occur and the state that the replica requests from the Directory is *Modified*.

The Directory behavior is similar to the second case of the `get` request where the Directory needs to send out invalidations. In this case the Directory will send an invalidation to either the owner of the row if the state is *Modified* already or will need to invalidate all the replicas with a read-only copy if the state is *Shared*. The invalidation messages require that the Frontends change their local row state to *Invalid*. The Directory will block until receiving an ACK for each invalidation before responding to the Frontend. In the case that the state at the Directory is *Invalid* or missing an entry, the Directory can set the state to *Modified* and return success to the Frontend without sending any invalidations. The Directory will locally lock the row on each request to prevent data races on the state of the row. The code for the Directory is shown in Algorithm 4.2.

Algorithm 4.2 Code for Directory, D

```
$directory = \{x \in (Row) \rightarrow Metadata)\}$, initially := $\varnothing$
$Metadata = (State, Replica\,List)$
$State = \{Modified, Shared, Processing, Invalid\}$

procedure on receipt of $request(Row, New\_State)$ from Frontend $F$
    lock(directory[Row])
    current state := directory[Row]
    if current state = $Processing$ then                    ▷ Avoid deadlock
        unlock(directory[Row])
        return send ($failure$) to $F$
    end if
    directory[Row] := $Processing$
    unlock(directory[Row])
    if New_State = Modified then
        $\forall r \in Replica\,List$ send Invalidation(Row, Invalid)
        block until received success response $\forall r$
        lock(directory[Row])
        directory[Row] := $(Modified, F)$
        unlock(directory[Row])                    ▷ $F$ is only valid replica now
        return send ($success$) to $F$
    else if New_State = Shared then
        if current state != Shared then
            $\forall r \in Replica\,List$ send Invalidation(Row, Shared)
            block until received ACK $\forall r$
        end if
        replica_sync := pick_replica(Replica List)     ▷ Choose $r$ somehow
        lock(directory[Row])
        directory[Row] := $(Shared, F \cup Replica\,List)$       ▷ Add $F$ to list
        unlock(directory[Row])
        return send $(success, replica\_sync)$ to $F$
    end if
end procedure
```

4.2.9 Fault-Tolerance

One of the main criticisms of traditional approaches to DSM is their poor tolerance of faults. Our system improves on the fault-tolerance of some of the older DSM systems by splitting the coherence protocol into two hierarchal levels, similar to [27]. The lower level of replication is within a datacenter, while the higher level of replication is across datacenters. The underlying storage system, such as Cassandra, provides its own (possibly tunable) consistency protocol while the Frontend manages the wide area replication protocol. This means that the system can tolerate a certain amount of faults within each datacenter before affecting inter-datacenter behavior. Each of the Frontends or Directory can be considered a single point of failure but state machine replication can be used to make them reliable [20].

In the case that Frontends become partitioned from the rest of the network, they may still be able to fulfill requests that only depend on local state. For example, writes can proceed if the local state is *Modified* and reads can be fulfilled in either *Modified* or *Shared* state. This provides a limited set of operations for disconnected operation while still maintaining strong consistency.

4.2.10 Drawbacks

One of the undesired behaviors of the system is when there is thrashing. This occurs when there is frequent contention of the same object and invalidation messages will constantly be sent between the requesting parties. One solution is to fail the requests which see concurrent behavior and have them wait using an exponential backoff [32] before trying again. Another solution is to add "global locks" to each object and have replicas acquire and release them before other replicas can access them. This will change the consistency model to release consistency and will be discussed in Section 4.3.

4.3 Release Consistency

One variation on the strong consistency model used in Section 4.2 is to use release consistency to provide strong application semantics [27]. The system still uses a two-level protocol but we introduce two synchronization operations `acquire` and `release`. Both of these operations are blocking calls. Each operation takes a row key as an argument and essentially “locks” and “unlocks” global access to the shared object for a Frontend (which may still serve more than one Client). Within this critical region, Frontend replicas do not need to push out the updates done to the object until the next `acquire` operation. Objects now only have two states, owned and not owned. The intuition here is that only one Frontend process may be in the critical region at once so the shared variables are only being used by one datacenter at any moment. Within the datacenter the Storage component maintains strong consistency among the clients. Additionally, synchronizing lazily at the next `acquire` rather than on the `release` reduces unnecessary messages [29].

4.3.1 System Overview

The architecture does not need to change when we change the consistency model. Figure 4.1 still clearly represents the system. However, the state at the Directory and Frontend have changed. Also we require the Client to acquire rows they wish to access from the Frontend and releasing them when finished. The Frontend must acquire the rows from the Directory. The Frontend aggregates the `acquire` and `release` operations for the clients at one datacenter and will send its own `fe_acquire` and `fe_release` when appropriate. The Directory only allows one datacenter to access a row, while the underlying Storage layer will take care of keeping access to that row consistent within that datacenter.

Now the only requests that flow through the Directory are the `fe_acquire` and `fe_release` operations. More specifically it is the first `acquire` and the last `release` that a Frontend receives from its Clients that require communication with the Directory. Intuitively each Frontend is acquiring the row for the datacenter so that multiple Clients in the same area may concurrently operate on that row. However, Clients which want to access the same row from a different datacenter through a different Frontend must wait until the holder of the row lock has released the row and the Directory has granted the Frontend admission into the critical region.

At the Frontend there is an atomic counter for each row. On each `acquire` from different Clients the counter is incremented while each `release` decrements the counter. Only on the first Client `acquire` does the Frontend send a `fe_acquire` to the Directory and a `fe_release` on the last Client `release`. This is necessary because there may be multiple Clients contacting each Frontend.

At the Directory a FIFO queue is held for each row. The Directory enqueues the Frontend on `fe_acquire` and dequeues it on `fe_releases`. The Frontend is notified when it is at the head of the queue and can enter the critical region. The Frontend will then respond to all of the Clients blocking on `acquire`.

4.3.2 Client Request Example

This section will walk through an example scenario where three clients are attempting to access the same key. Two of the Clients are accessing the same datacenter and can operate on the row concurrently while the third Client needs to wait until the others have released access. Figure 4.2 depicts this situation. Each number represents at least one sent message. The following bullets explain what message is sent and the associated events which occur with it.

1. Client 1 and 2 both are attempting to access key k. They both send an `acquire` to Frontend 1 and block until they receive a response.

2. Frontend 1 only sends one `fe_acquire()` message to the Directory and also blocks until a response. In this case since no Frontend is holding the lock for k the Directory can immediately respond with a success message. The Frontend will respond back to both Client 1 and 2 and they can proceed to perform their operations. The Storage component still maintains the strong consistency within the datacenter when processing both clients' operations.

3. Client 3 attempts to access k by sending an `acquire` to Frontend 2 before Client 1 and 2 have released their lock on k.

4. Frontend 2 asks the Directory for the lock on k with a `fe_acquire`. The Directory enqueues it on the queue for row k. Frontend 2 will wait

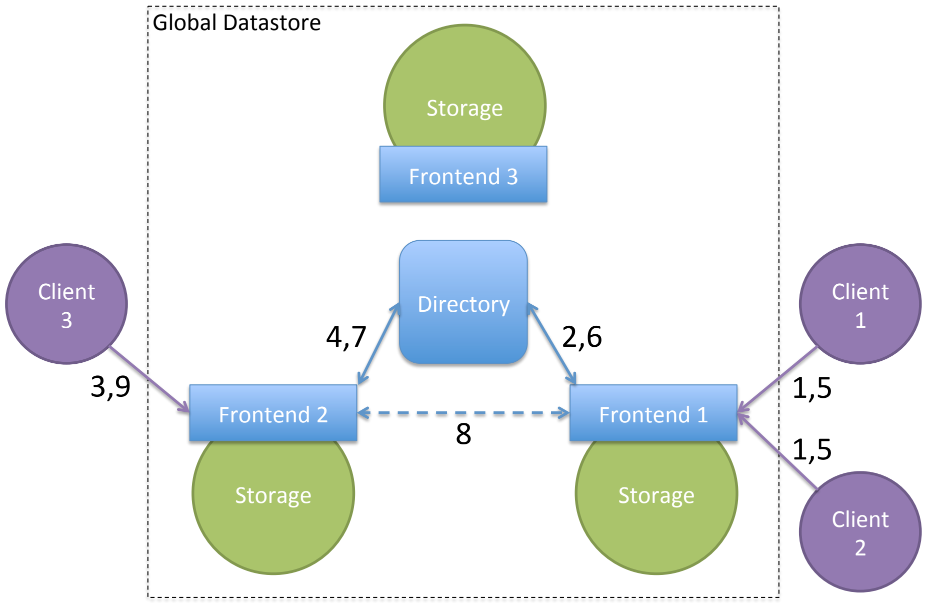

Figure 4.2: Client Request Example for Release Consistency. Each number is a message event and described in detail in Section 4.3.2.

until it gets to the front of the queue. Additional Clients who wish to access k through Frontend 2 will also block.

5. Once both Clients 1 and 2 have finished their access to k they will send the `release` messages to Frontend 1.

6. Only after receiving all of the releases on k will Frontend 1 ask the Directory to dequeue itself by sending a `fe_release`.

7. The Directory will dequeue Frontend 1 and notify the head of the queue it can enter the critical section along with which Frontend it can receive the most up-to-date version of k from. In this case Frontend 2 will receive a message from the Directory informing it Frontend 1 has the most up-to-date version of k.

8. Frontend 2 will ask Frontend 1 to synchronize on k by sending it a copy of k.

9. Frontend 2 will respond back to Client 3 who may now access k.

4.3.3 Drawbacks

Like with other locking schemes there are a few potential drawbacks with this approach. The largest drawback is that deadlocks are possible whenever a `release` is not called appropriately. Other Clients will be blocked indefinitely until the holder of the lock gives it up. One possible solution to this is a heartbeat and timeout mechanism [23]. If a process does not respond saying that it is still actively accessing the row, then the Directory could force it to release the lock.

Another inherent problem is when replicas are waiting to access a row there is an unbounded amount of time they have to wait. However, if all processes are behaving properly then we expect progress to be maintained in a timely fashion.

4.4 Implementation

We have chosen to implement the Frontend and Directory in both system designs as separate processes which communicate with the underlying Storage component. The alternative was to integrate the two systems tightly but this design allows for more flexibility in how the Storage component will replicate the data within the datacenter.

The Frontend and Directory processes build on the message passing and Staged Event Driven Architecture (SEDA) [33] of Cassandra and added only around 2800 lines of code. To interface with the Cassandra backend we use Netflix's Astyanax driver. This driver provides connection pooling to Cassandra and is easier to work with than the Thrift interface.

Building on the same source code as Cassandra gives us numerous benefits. Some of the benefits were not having to recreate the Client Thrift interface, having the well structured SEDA architecture and the possibility to use the JMX monitoring. Currently the Directory and Frontend both store their state only in memory, but with the embedded Cassandra database layer it would not be difficult to store it on disk. Besides the implementation advantages there are also benchmarks for Cassandra that are easily adapted to our datastore such as the Yahoo! Cloud Serving Benchmark.

Chapter 5

Experiments

In this chapter we evaluate the performance of multi-datacenter Cassandra, our directory-based strongly consistent system, and our system using release consistency. For convenience we will abbreviate Cassandra as C*, the strongly consistent system as SC and the release consistency system as RC. Our evaluation will focus on the factors and parameters that influence latency. Since the network delay is considerably higher in a geo-replicated setting than other environments, throughput is constrained by that delay. For example, if the average delay is 200ms, then there can only be five requests per second regardless of processing time. Additionally since both directory-based systems are using Cassandra as the underlying storage layer we do not need to focus on low level details such as I/O cost and how rows are stored physically. Those costs will be reflected in all measurements. This chapter will cover the experimental setup and then discuss the overall results before going into more detail on various factors.

5.1 Experimental Setup

The experiments were conducted using seven computers in the Illinois Cloud Computing Testbed (CCT) [34] and two other computers within the University of Illinois network. Each physical machine in the CCT has dual 64-bit quad-core processors with 16 GB of RAM, gigabit Ethernet and CentOS 5.9 with Linux Kernel 2.6. The servers in CCT use shared storage accessed through Network File System (NFS). The other two computers have dual 64-bit six-core processors with 24 GB RAM, 2x1 TB disks, gigabit Ethernet and CentOS 6.3 with Linux Kernel 2.6. The CCT servers are running the datastore systems while the other computers are running the Yahoo! Cloud Serving Benchmark (YCSB) 0.1.4 client [35]. We run Cassandra 1.1.10 in its

Table 5.1: Sample ping latencies (round-trip delay) from CCT to Destination (ms)

Destination	Mean	Min	Max	Std. Dev
China (220.181.111.86)	352.9	350.7	469.2	10.5
Australia (139.130.4.5)	215.2	214.2	215.9	0.6
France (87.98.182.37)	102.7	101.7	107.9	0.9
US West (64.37.174.140)	73.4	72.2	101.7	2.6
US East (199.108.194.38)	27.1	25.2	144.2	10.2
Simulated Datacenter	201.2	190.6	244.2	9.9
Simulated Directory	101.3	95.6	129.6	5.6

multi-datacenter mode when evaluating Cassandra and in single-datacenter mode when used as the backend to the other systems.

To evaluate the performance of the systems in a geo-distributed environment we use emulated delays in the network layer. More detail is provided in Section 5.1.1.

5.1.1 Delay Simulation

In order to simulate the behavior of a real geo-replicated datastore, delays were added to the network layer in the Linux kernel using the `tc` tool. `tc` uses the Network Emulator, `netem` component to add delays to packets for preconfigured destinations. More information on the tools and configuration can be found in Appendix A.

The given parameters to use for the delays were chosen based on pings to servers in various geographic locations. Table 5.1 shows some sample latencies based on 200 pings from the Illinois CCT to the specified destination along with the two simulated delays in the last two rows.

We chose to use a one way 100ms delay between datacenters and a 50ms delay from the Directory to each datacenter for the experiments. To model the variation of network delays, each one is augmented with an additional delay that has a Pareto distribution with `tc` parameters 8ms and 4ms respectively, as shown in Appendix A. 200ms RTT was chosen to simulate a large enough practical delay that shows the cost of geo-replication.

5.1.2 Multi-Datacenter Cassandra

To evaluate the performance of Cassandra in a multiple datacenter configuration, six CCT servers were divided into two groups of three. Each three-server cluster acts as a datacenter and has a remote delay to the other one. The client writes to Cassandra with consistency level `EACH_QUORUM` and reads at consistency level `LOCAL_QUORUM` in order to maintain strong consistency. The keyspace is configured to have three replicas of each object in each datacenter. This results in six total copies of the object with four replicas having to respond to writes (two per datacenter), and only two for reads (from the local datacenter). The writes will have to wait on both datacenters while reads can be served locally. Refer to Section 3.2 for more information on Cassandra.

5.1.3 Directory-Based Systems

To evaluate the performance of the SC and RC systems, six CCT servers were divided into two groups of three. Each group forms a Cassandra cluster configured for a single datacenter and a Frontend process is run on one of the servers in each group. A Directory process is running on a separate CCT server. Each Cassandra cluster has a delay to the other one. Both servers running the Frontend have a delay to the Directory. The Frontends and Directory can be configured to be in either the strong consistency or release consistency mode. This setup is depicted in Figure 5.1. The dotted lines surrounding processes shows the physical boundaries of the machine. The underlying Cassandra storage layer is unaware of the Directory and other Cassandra clusters. They only contact the Frontend associated with that cluster. Underlying Cassandra clusters are operating at `QUORUM` for both reads and writes to maintain strong consistency at the storage level.

5.2 Workloads

We use Yahoo! Cloud Serving Benchmark (YCSB) to evaluate the performance of the datastores. YCSB is a framework that allows users to benchmark various datastores. The system predefines various workloads and distri-

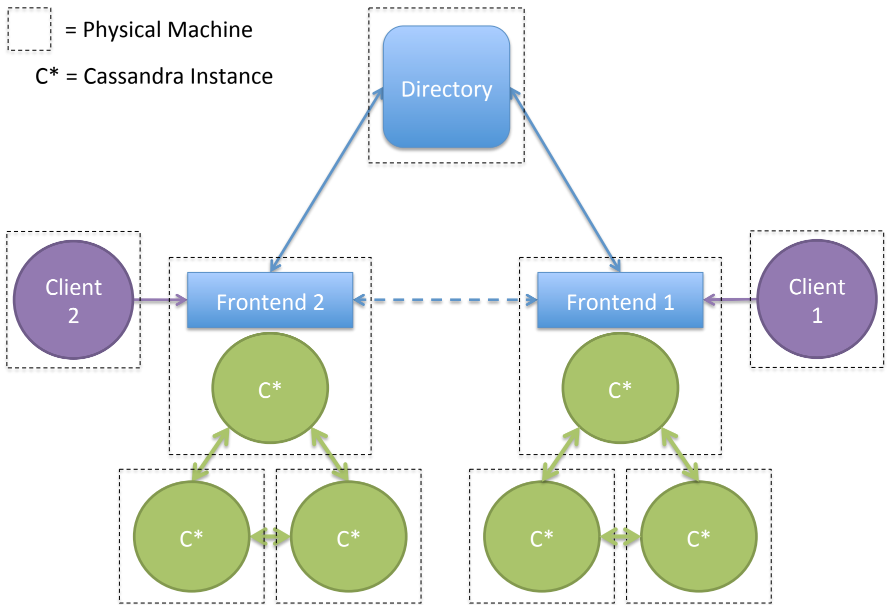

Figure 5.1: Experimental Setup for Directory-based Schemes

butions to randomly generate the client load [35]. Among the choices, we run update heavy workload A (50% writes), read heavy workload B (95% reads), read only workload C, and read latest workload D. The distributions that were used include Zipfian, Uniform, and Latest. The Zipfian distribution is one where a small amount of rows are being accessed most often (the head of the distribution) while the remaining rows are infrequently accessed (the tail of the distribution) [35]. In the Zipfian distribution the head has most of the area while its tail is long. The Latest distribution is like the Zipfian distribution but the most recently inserted rows are in the head of the distribution and have the highest access frequency. A uniform distribution has equal probability of accessing any of the rows. We additionally modify various parameters such as grouping more operations into the read-modify-write operation for workload F and a workload which performs only writes. To use the client we have had to slightly modify the existing Cassandra driver in YCSB to support multi-datacenter strong consistency and also to conform to the release consistency interface.

The benchmarking process is composed of two parts, the *load* phase and the *transaction* phase. The load phase inserts all of the records in the datastore while the transaction phase performs requests to the system based on the workload distribution and given parameters. The load phase is typically

Table 5.2: **Overall latency metrics (ms)**

	50%	90%	95%	99%	Mean	Std Dev.
Read						
Strong Consistency	2	812	823	845	253.42	211.60
Release Consistency	150	319	333	400	207.67	112.01
Cassandra	1	2	3	4	2.56	32.46
Write						
Strong Consistency	197	211	216	228	122.81	61.05
Release Consistency	293	319	334	400	226.65	122.65
Cassandra	198	211	220	292	212.96	29.43

only run once while the run phase is repeated multiple times with different workloads and parameters. The metrics here mainly concern the transaction phase. Unless otherwise specified each workload consisted of 60,000 records with 5,000 operations from each of two clients. Each workload was performed at least 5 times and the overview of the combined results are presented in Table 5.2 and Figures 5.2 and 5.3. Table 5.2 gives precise measurements that show the characteristics of the read and write operations for all the systems. Table 5.2 shows the latency that $N\%$ of requests finished under and lists the mean and standard deviation of all the requests. Figure 5.2 conveys the mean and standard deviation visually. Figure 5.3 shows the cumulative distribution (CDF) of latencies for the overall system when read and writes are combined. Note that the CDFs are plotted on a logarithmic scale due to the high variance of the latencies. One artifact of using the CCT is the occasional high delays from NFS writing to all the servers. This is one of the factors which contribute to the high variance, but affects all of the systems.

In general C*'s performance is much more predictable and consistent than SC or RC. C*'s behavior does not depend on the current system state nor the state at other datacenters while SC and RC are more complicated. C* has quick local reads while writes have higher latency from the remote datacenter messages. SC can perform local operations very quickly if the Frontend has the appropriate state for the Client request but suffers heavily from a "row miss" and must contact the Directory and potentially other Frontends. RC suffers from the overhead of `acquire` operations and must wait until no other Frontend is accessing the row. Additionally the `acquire` operation forces us to synchronize with the Frontend for the latest copy of the row if necessary.

This is apparent in Figure 5.3. None of the requests can finish faster than the delay to the Directory, which is around 100ms, but also a tail to around 400ms when it must retrieve the row from the remote datacenter. Figure 5.3 also shows the two spikes in each system which is the result of quick local operations vs. slower remote operations. These factors lead SC and RC to have much more variance in their latencies, but can outperform C* in certain circumstances. For example, from Table 5.2 we can see the average SC write latency is 122.81ms while C*'s is 212.96ms. C* must wait for the remote datacenter to respond on every write request whereas SC can perform many local operations if there is no contention on the row. However, once there are other datacenters attempting to access that row, there are more messages passed between the Directory and Frontends, along with the time it takes to synchronize the row between Frontends.

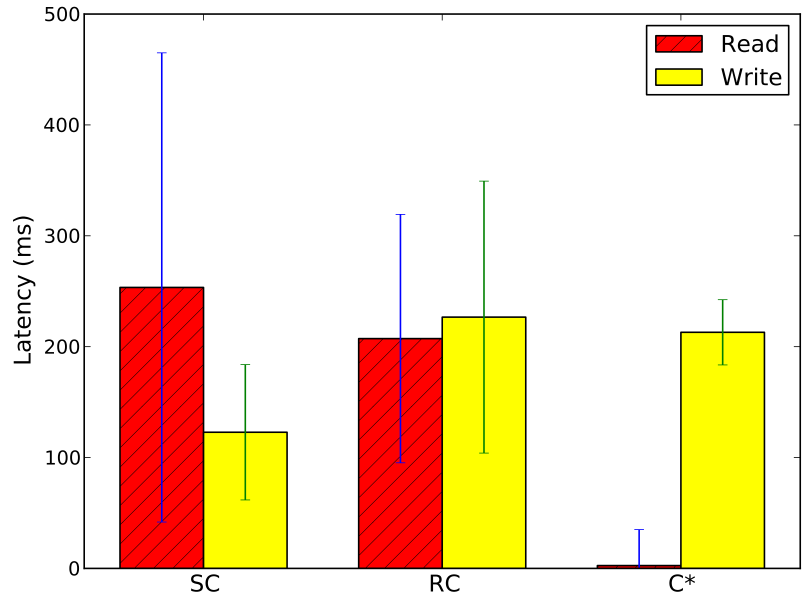

Figure 5.2: Average read/write latency with stddev. (ms)

One issue when evaluating the performance of SC is that the latency depends on the current key state. The numbers used in this section are the average of the best and worst case scenarios. We present more detail in Section 5.2.2. When evaluating RC, we came across a similar problem because Clients may only access a row after they have acquired it. While the `acquire` is an expensive operation, Clients may quickly access the row afterwards numerous times before the `release`. The measurements for RC in this

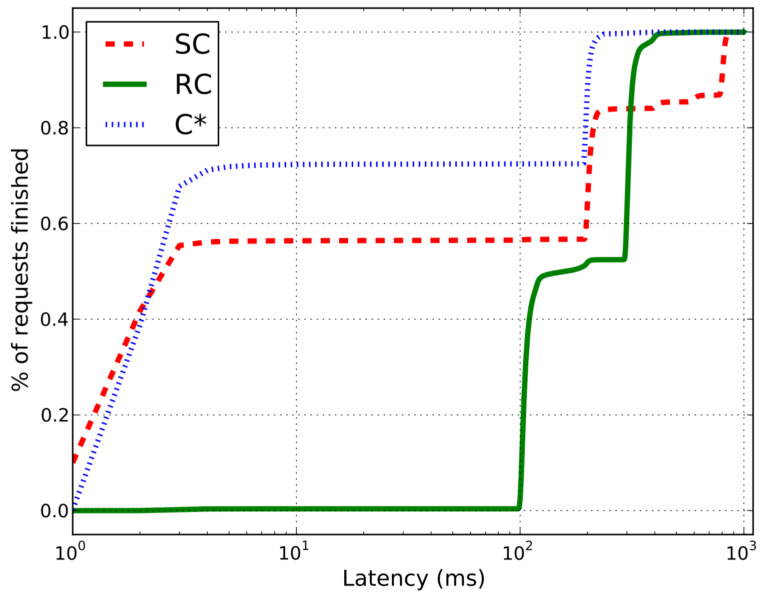

Figure 5.3: CDF of request latency

section are the result of accessing the row once between the synchronization operations. More detail is presented in Section 5.2.3.

5.2.1 Read-Write Ratio

C* writes to remote datacenters for every write request but reads only the local datacenter. SC optimizes for writes by the same datacenter or many reads to shared objects. SC should outperform C* whenever C* is pessimistically[1] contacting the remote datacenter. RC should not be affected much by the read-write ratio since it has to `acquire` before it attempts to read or write to a row.

Figure 5.4 shows the average latency as the fraction of writes increases for all of the systems. SC-1 and SC-2 represent the average latency that Client 1 and Client 2 experience when the system is in SC mode. SC is the average of those latencies. SC-1 is the Client that loaded all of the rows into the datastore so the Frontend that it is contacting has all of its rows in the *Modified* state initially. This means that it can serve both read and write requests locally. SC-2 however must contact the Directory on practically every

[1]Pessimistic in this context means that the operation was not necessary at that moment to maintain consistency.

request and often synchronize its row state with the remote datacenter. As the fractions of writes increase, all of the system average latencies approach around 200ms delay. C* must contact the remote datacenter on every write, and SC-2 does not need to synchronize its data because it is writing, but still must wait for the Directory to change the state of the row to *Shared* at the other Frontends. As expected, RC's performance is independent of the read-write ratio.

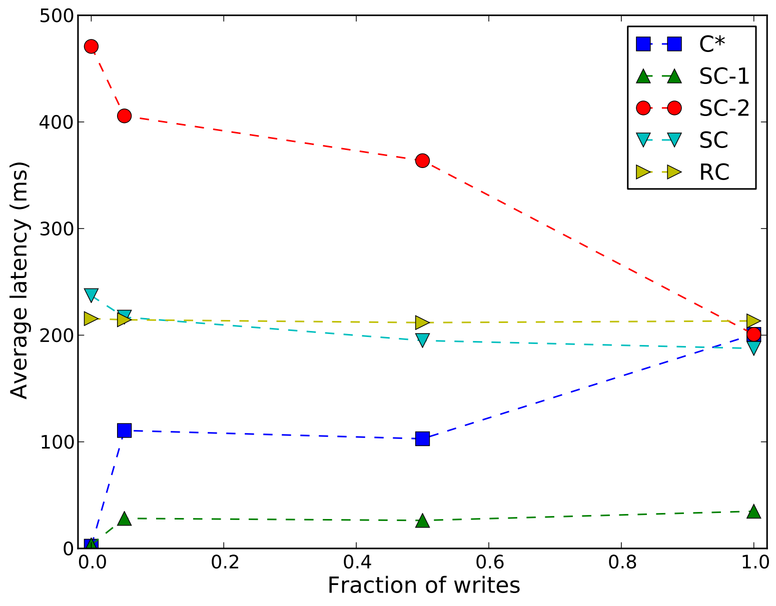

Figure 5.4: Average latency as write fraction increases. SC-1 and SC-2 are each Clients while SC is the overall system latency.

5.2.2 State Dependency

The performance of SC depends heavily on the state of the rows accessed as we have seen in Figure 5.4. If an operation can be served locally in either the *Modified* or *Shared* state then the request will finish two orders of magnitude faster than an operation which requires remote messaging. The average local operation can finish in the single digit microseconds while an operation which requires remote accesses can take from 100 to 600ms depending on if the Frontends need to synchronize to the most up-to-date version. When a Client request cannot be served locally, we call that a *row miss*. Such misses are one of the reasons why the variation in latencies is high for SC.

Figure 5.5 shows a time series trace of SC performing workload C with a Zipfian distribution with a linear regression line fitted. Each data point is the average of the requests within a five second interval. This workload is entirely reads. As more of the rows at the Frontend become *Shared*, latency decreases since reads can be made locally instead. The regression line shows this trend. However, the Zipfian distribution has a very long tail, so the Frontend still must synchronize on rows it has not accessed before. Additionally, if there are other Frontends attempting to write to those rows then there would need to be more expensive synchronization operations occurring.

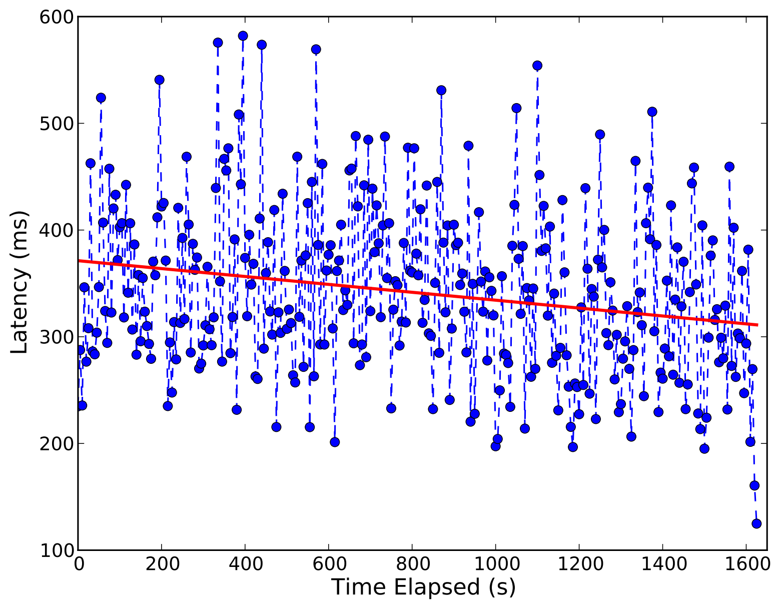

Figure 5.5: Time series trace of SC running workload C. The fitted linear regression line shows latency dropping as time progresses since the Frontend has more rows with Shared state.

To account for this variation, all runs in this chapter were done in the following way for SC. One Frontend runs the load phase of YCSB, thus changing all rows to the *Modified* state for that Frontend. Then both Frontends run the run phase of YCSB. One Frontend will be able to serve most requests locally while the other Frontend will need to perform expensive row synchronizations on almost every update. However, if eventually there are no more writes to the shared rows, then both Frontends will have *Shared* access to those rows and enable them to serve reads locally. After many operations the state should become steady with Clients just reading the rows locally.

The throughput of the YCSB client is defined as the number of operations completed per second [35]. The YCSB client allows us to specify a *target throughput* which allows us to control the rate of which requests are generated. When the client cannot reach the target throughput, requests are generated immediately after the previous operation completes. In the case that the throughput is higher than the target throughput, the YCSB client waits for the appropriate amount of time before issuing the next request. The waiting time is not included in the latency of the request. The target throughput of the Client at the Frontend who performed the load phase is throttled so that both Clients are accessing the datastore simultaneously for the entire duration of the run phase (otherwise one Client would finish much earlier than the other). The average of the runs from both clients is what is used in most of the results. However, the cost of a row miss is extremely high and we see that they can quickly dominate the advantage of local operations. The variation of misses accounts for the large variation in latencies.

5.2.3 Batch Size

Most of the YCSB workloads are tailored toward single operations which have no state dependency between operations. For all of the results presented in Section 5.2 Clients issued only one request for each pair of `acquire` and `release` when accessing RC. However, with release consistency we should have very high performance on the acquired row amortized across many operations. The expensive `acquire` operation will not matter as much per operation if there are many requests before the following `release`. We call the number of operations between `acquire` and `release` the *batch size*. We can model the average cost of each operation as such: Let D be the round-trip delay from each of the datacenters to the Directory, R be the delay it costs to perform the read synchronization, L be the round-trip delay from the client to the local datacenter, B be the batch size. The average time for each operation is then roughly

$$\overline{t_{op}} = \frac{D + R + BL}{B} \tag{5.1}$$

Figure 5.6 shows Equation 5.1 with $D = 100$, $R = 200$, and $L = 0.45$ along with the experimental values. The experimental values were attained

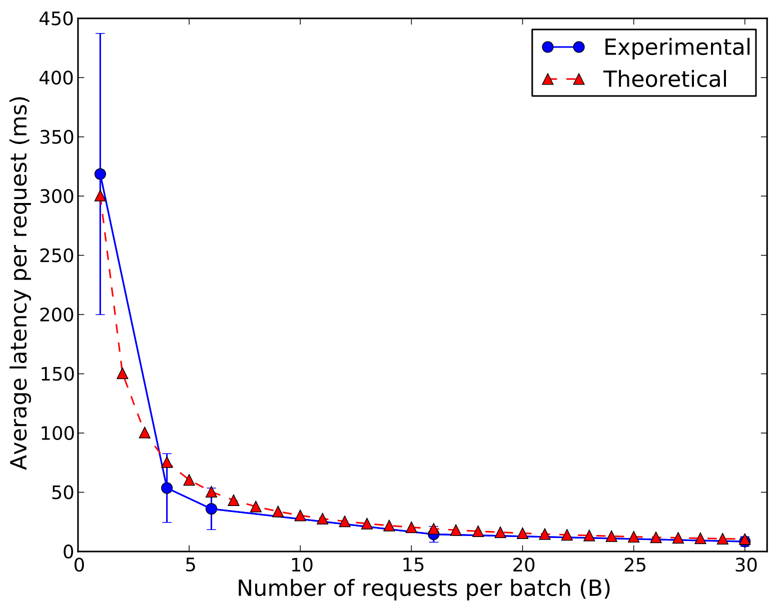

Figure 5.6: Experimental and theoretical performance of RC using Equation 5.1 with $D = 100$, $R = 200$, and $L = 0.45$. Experimental values are averages and error bars are standard deviation.

by modifying YCSB workload F to only use read-modify-write operations and performing B reads and writes per batch operation. For each B we used at least 2,000 batch operations, which means that each run had at least $2 * 2,000B$ requests between both clients. The latency of the batch operation is divided by B to calculate the average *per request*. The average per request latency is plotted in Figure 5.6 with the standard deviation as error bars.

The benefits from RC are not as apparent in the normal YCSB workloads since we are not performing many operations within each critical section. The default behavior is to `acquire` and `release` for every request. Although the theoretical result presented in Figure 5.6 does not take into account the waiting time associated with getting into the critical section it seems that the experimental results actually perform slightly better than predicted. The reason is that Equation 5.1 is pessimistic and assumes that every batch operation will need to perform a read synchronization between the Frontends. When the Frontend was the previous replica to `release` the row, then no synchronization is needed. Additionally we could also extend the implementation of `acquire` to gain access to more than one row at a time. This would allow us to reduce the overhead of `acquire` at the cost of introduced complexity.

5.3 Row Contention and Workload Distribution

The performance of both SC and RC suffers when there is row contention between datacenters. For SC, alternating reads and writes between datacenters will force the Frontend to contact the Directory and possibly synchronize the row from the remote Frontend. Contention in RC will cause a Frontend to wait whenever another Frontend is currently holding the lock for a row.

The probability of row contention should be dependent on the workload distribution. The workload distributions used in our experiments are Zipfian, Latest and Uniform. Zipfian and Latest distributions model realistic workloads [35]. The Zipfian distribution is one where the head has most of the area but its tail is long. The Latest distribution is where the most recently inserted rows have the highest access frequency. A uniform distribution has equal chance of accessing any of the rows. Figure 1 in [35] shows an example of the distributions that are used in YCSB.

If the workload behaves Zipfian then the most common records will be accessed most of the time. This can work to our advantage in SC if all processes are reading the data, but will be disadvantageous if multiple parties are attempting to modify the row. This distribution should not be good for RC because multiple parties will need to wait for the lock to be released. For uniform workloads contention should be relatively lower.

C*'s performance does not depend on the row contention while there is possibility for write thrashing and expensive synchronization for SC and waiting on rows for RC.

Figure 5.7 shows the effects of the distributions on average read and write times for all three systems for workload B with the request distribution as either Zipfian, Latest, or Uniform. As expected the choice of workload distribution does not influence C* much. SC is heavily affected by the Zipfian distribution since writes to the common rows will cause expensive read synchronizations to occur. With Latest and Uniform this effect is mitigated slightly, but we still see high variation in the latencies. The workload distribution does not seem to affect RC very much, but the batch size was set to one. We expected to see higher delays in Zipfian while the Frontend is waiting for access, but the effect would probably be more observable if the batch size were higher so each Frontend held the row longer.

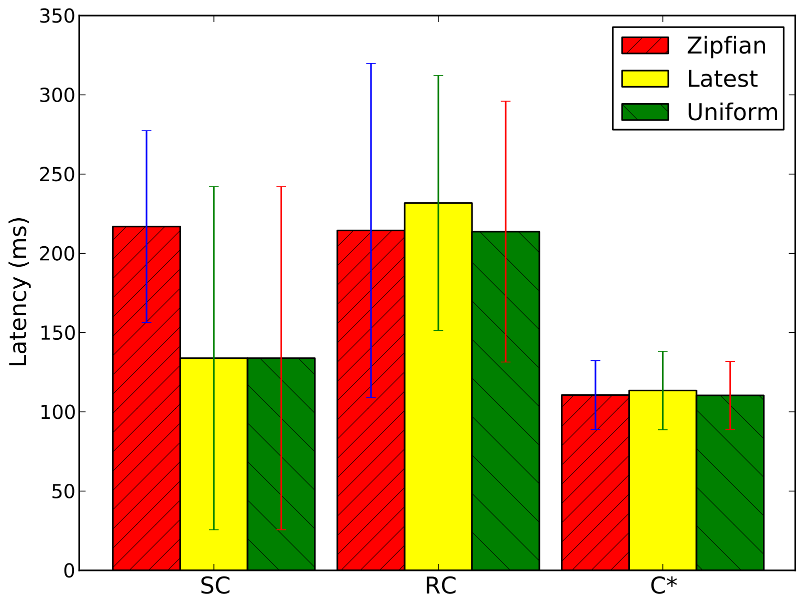

Figure 5.7: Average combined read/write latencies for workload B with various workload distributions. Standard deviation is plotted as error bars.

5.4 Directory Placement

Both directory-based systems can give priority to selected datacenters by placing the Directory server geographically closer to those datacenters. This is effectively giving priority to those regions since the communication delay between the Directory will decrease. However, the latency between datacenters will remain the same so the cost of synchronizing data will still be high. This approach is simpler than setting priorities through scheduling or quality of service but is harder to change dynamically.

5.5 Summary

Through experimental evaluation we have seen that the latency for SC and RC is affected significantly by the expensive synchronization operations that both systems must perform in order to maintain their respective consistency guarantees. While there are scenarios where SC and RC have better performance than C*, such as write-heavy workloads where there are not many row conflicts, the variability of the latency requests leads to unpredictable behavior where the average latency depends heavily on the workload the system faces. However, if users of the system know that the workload that the datastore will face has many read operations to shared rows while the rows that are modified are not accessed by geographically separated datacenters, often then SC could be an appropriate system choice. If there is low row contention and many operations are issued within a critical section then RC could be a preferred system design.

Chapter 6

Conclusions and Future Work

In this thesis, we have presented two directory-based datastores which support strong application semantics. Their designs are based on ideas from the multiprocessor and distributed shared memory literature which has dealt with memory coherence problems in a different environment. When these techniques are used to maintain consistency in a geo-distributed environment the performance suffers for workloads commonly encountered in replicated datastores. The costs of synchronization are high. However, there are situations in which the directory-based systems do provide good performance. One of these situations is when there is not much row contention between geographically separate datacenters.

Replicated datastores will become increasingly important as the demands for accessing data grow. In particular, better ways for handling geo-distributed data need to be developed. Perhaps expanding the consistency models offered by the system to fit the needs of the application will be one of the correct decision choices [9, 11]. Another useful primitive that has been developed by recent replicated datastores is transactions [10, 11]. Transactions have been used for decades by relational databases but have been difficult to implement efficiently in replicated datastores. The complex interactions of a distributed replicated datastore bring together the problems faced in both the distributed systems and database community and present new challenges in the future.

Appendix A

Network Delay Simulation in Linux

This appendix will cover some background of the command line tools and commands used in the experiments of this thesis in order to simulate a geo-replicated environment for the datastore systems.

The `netem` kernel component provides Linux with the ability to emulate various network functionality. Some of the features in the current version include variable delay, loss, duplication and re-ordering. The module is controlled by the command line utility `tc` by creating `qdisc`s, which are queues which hold packets and can optionally do extra processing. The kernel enqueues a packet onto an interface's `qdisc` whenever it wishes to send a packet to that interface. `netem` can be used by creating a `qdisc` and specifying netem as one of the parameters. As an example, the following command will add an additional 100ms of delay to all outgoing traffic on interface eth0.

```
$ tc qdisc add dev eth0 root netem delay 100ms
```

For the remainder of the appendix we assume that the network interface that we want to manipulate is eth0. We may also specify additional parameters to modify the variability of the delay and the distribution of the variation. The following command adds 100ms ± 20ms of delay with a normal distribution to all outgoing traffic on eth0.

```
$ tc qdisc change dev eth0 root netem delay 100ms 20ms
    distribution normal
```

However, we may not want to always add the delay to all traffic, or we may wish to specify different delays for different hosts. In order to do so we must use `tc filter`s. We will present the commands used in the experiment and then explain them line by line. The commands used in the experiments on one server are presented below in order to add a 100ms ± 8ms delay where the variations follow a Pareto distribution.

```
$ tc qdisc add dev eth0 root handle 1: prio
$ tc qdisc add dev eth0 parent 1:1 handle 2: netem delay 100ms
    8ms distribution pareto
$ tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32
   match ip dst 172.22.28.99/32 flowid 1:1
```

The first line creates a classful `qdisc` that contains three classes with the assigned handle number 1:. The `PRIO qdisc` is a special `qdisc` that automatically creates these three classes for us. Classes are ways for `qdisc`s to group behaviors. The next line attaches a `netem qdisc` underneath one of the automatically created classes with the specified parameters. The three generated classes are by default called 1:1, 1:2, and 1:3. In this example we choose to attach to the first one. The last line of the example creates a filter attached to the root `PRIO qdisc` and puts any packet with an outgoing IP address of 172.22.28.99 into the 1:1 class (which we have added our delay behavior to). By using this same method we can now add various delays to different hosts. To add additional hosts to the already created `netem qdisc` we just make additional filters. To add a different delay we need to create another `netem qdisc` and also create filters for it (if we do not want all other outgoing traffic to end up in this `qdisc`). One pitfall is that an additional `qdisc` with zero delay and a filter for all other IP addresses may be necessary to avoid having other packets being enqueued onto the `qdisc` with delays. An example of a command which generates such a `qdisc` is below.

```
$ tc qdisc add dev eth0 parent 1:3 handle 4: netem delay 0ms
$ tc filter add dev eth0 protocol ip parent 1:0 prio 2 u32
   match ip dst /0 flowid 1:3
```

Additional information can be found in the `man` pages for `tc` along with information at `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem` and `http://lartc.org/howto/lartc.qdisc.classful.html`.

References

[1] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1454159.1454167

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data System Research (CIDR)*, 2011, pp. 223–234.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281 pp. 205–220.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," vol. 44, no. 2. New York, NY, USA: ACM, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922 pp. 35–40.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323 pp. 15–15.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/945445.945450 pp. 29–43.

[7] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224056.224070 pp. 172–182.

[8] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. U. Haq, M. I. U. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043571 pp. 143–157.

[9] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguia, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," ser. OSDI '12. ACM, 2012, pp. 265–278.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Googles globally-distributed database," ser. OSDI '12. ACM, 2012, pp. 251–264.

[11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482657 pp. 313–328.

[12] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465363 pp. 113–126.

[13] E. Brewer, "A certain freedom: Thoughts on the CAP theorem," in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835701 pp. 335–335.

[14] E. Brewer, “CAP twelve years later: How the “rules” have changed,” *Computer*, vol. 45, pp. 23–29, 2012.

[15] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[16] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 4th ed. USA: Addison-Wesley Publishing Company, 2005.

[17] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, July 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[18] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, “Causal memory: Definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF01784241

[19] W. Vogels, “Eventually consistent,” *Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[20] L. Lamport, “Paxos made simple,” *SIGACT News*, vol. 32, no. 4, pp. 51–58, Dec. 2001. [Online]. Available: http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf

[21] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” in *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, ser. PODS ’83. New York, NY, USA: ACM, 1983. [Online]. Available: http://doi.acm.org/10.1145/588058.588060 pp. 1–7.

[22] M. Burrows, “The Chubby lock service for loosely-coupled distributed systems,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298487 pp. 335–350.

[23] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. USA: Addison-Wesley Publishing Company, 2011.

[24] J. Sustersic and A. Hurson, “Coherence protocols for bus-based and scalable multiprocessors, internet, and wireless distributed computing environments: A survey,” ser. Advances in Computers. Elsevier, 2003, vol. 59, pp. 211 – 278. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245803590052

[25] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[26] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '86. New York, NY, USA: ACM, 1986. [Online]. Available: http://doi.acm.org/10.1145/10590.10610 pp. 229–239.

[27] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: ACM, 1990. [Online]. Available: http://doi.acm.org/10.1145/325164.325132 pp. 148–159.

[28] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '91. New York, NY, USA: ACM, 1991. [Online]. Available: http://doi.acm.org/10.1145/121132.121159 pp. 152–164.

[29] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992. [Online]. Available: http://doi.acm.org/10.1145/139669.139676 pp. 13–21.

[30] "Cassandra Thrift API," http://wiki.apache.org/cassandra/API, accessed: 2013-05-16.

[31] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, ser. AFIPS '76. New York, NY, USA: ACM, 1976. [Online]. Available: http://doi.acm.org/10.1145/1499799.1499901 pp. 749–753.

[32] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993. [Online]. Available: http://doi.acm.org/10.1145/165123.165164 pp. 289–300.

[33] M. Welsh, D. Culler, and E. Brewer, “SEDA: An architecture for well-conditioned, scalable internet services,” in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’01. New York, NY, USA: ACM, 2001. [Online]. Available: http://doi.acm.org/10.1145/502034.502057 pp. 230–243.

[34] “Illinois cloud computing testbed (CCT),” accessed: 2013-05-16. [Online]. Available: http://cloud.cs.illinois.edu/

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC ’10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152 pp. 143–154.