INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Geo-Replication in Large Scale Cloud Computing Applications

Sérgio Filipe Garrau dos Santos Almeida

Dissertation for the Degree of Master of
Information Systems and Computer Engineering

Jury

President:	Prof. Doctor João Emílio Segurado Pavão Martins
Advisor:	Prof. Doctor Luís Eduardo Teixeira Rodrigues
Member:	Prof. Doctor José Orlando Roque Nascimento Pereira

November 2012

Acknowledgements

I would like to issue a special thanks to my advisor Professor Luís Rodrigues and to João Leitão for the opportunity of doing this thesis and also for all the help and motivation during this period of time.

I am also thankful to my coleagues Oksana Denysyuk, João Paiva, and Miguel Branco for the fruitful discussions and comments during the execution of all the work and preparation of this thesis. I am also thankful to my family and friends for all the support during the execution of this thesis.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/ EEI/ LA0021/ 2011, and via the project HPCI (PTDC/ EIA-EIA/ 102212/ 2008).

Lisboa, November 2012

Sérgio Filipe Garrau dos Santos Almeida

For my family and friends,

Resumo

O equilíbrio entre coerência, disponibilidade e escalabilidade, nomeadamente em sistemas que suportam Geo-replicação, é um dos maiores desafios na construção de sistemas de base de dados distribuídos para aplicações baseadas em *computação em núvem*. Desta forma, várias combinações entre garantias de coerência e protocolos de replicação têm sido propostos nos últimos anos.

O trabalho descrito nesta tese tenta avançar esta área de investigação através da proposta de uma nova arquitectura para um sistema de base de dados distribuído. Esta arquitectura permite oferecer garantias de coerência *causal+*, tolerância a faltas, escalabilidade e um elevado desempenho. Introduzimos uma nova técnica de replicação baseada na replicação em cadeia, esta última oferece garantias de coerência atómica e elevado desempenho de uma forma simples. A nossa abordagem permite evitar os pontos de estrangulamento associados a soluções que oferecem garantias de coerência atómica e oferece um desempenho competitivo com soluções que oferecem garantias mais fracas (coerência eventual). Para mais, o ChainReaction pode ser instalado num único ou em vários centros de dados distribuídos geograficamente.

Os benefícios do ChainReaction foram avaliados experimentalmente através da utilização do Yahoo! Cloud Serving Benchmark para testar um protótipo da nossa solução em comparação com o Apache Cassandra e o FAWN-KV. Por fim, a nossa solução também oferece uma primitiva transaccional que permite a um cliente obter o valor de vários objectos de uma forma coerente. Resultados experimentais mostram que esta extensão não resulta num impacto negativo no desempenho do ChainReaction.

Abstract

Managing the tradeoffs among consistency, availability, and scalability, namely in systems supporting Geo-replication, is one of the most challenging aspects of the design of distributed datastores for cloud-computing applications. As a result, several combinations of different consistency guarantees and replication protocols have been proposed in the last few years.

The work described in this thesis makes a step forward in this path, by proposing a novel distributed datastore design, named ChainReaction, that offers *causal+* consistency, with high performance, fault-tolerance, and scalability. We introduce a new replication technique based on Chain Replication, a replication technique that provides linearizability and high performance in a very simple way. Our approach avoids the bottlenecks of linearizability while providing competitive performance when compared with systems merely offering eventual consistency. Furthermore, ChainReaction can be deployed both in a single datacenter and Geo-replicated scenarios, over multiple datacenters.

We have experimentally evaluated the benefits of our approach by applying the Yahoo! Cloud Serving Benchmark to a prototype deployment that includes our own solution as well as Apache Cassandra and FAWN-KV. Finally, our solution also provides a transactional construct that allows a client to obtain a consistent snapshot of multiple objects. Experimental results show that this extension has no negative impact on the performance of ChainReaction.

Palavras Chave
Keywords

Palavras Chave

Gestão de dados

Sistemas de armazenamento de dados chave-valor

Tolerância a faltas

Geo-Replicação

Coerência Causal+

Keywords

Data-management

Key-value datastores

Fault-tolerance

Geo-Replication

Causal+ Consistency

Contents

List of Figures

List of Tables

Acronyms

ACID Atomicity, Consistency, Isolation, Durability

ACL Access Control List

API Application Programming Interface

CAP Consistency, Availability, Partition Tolerance

COPS Cluster of Order-Preserving Servers

CRAQ Chain Replication with Apportioned Queries

DHT Distributed Hash Table

FAWN Fast Array Of Wimpy Nodes

FAWN-DS FAWN Data Store

FAWN-KV FAWN Key Value Store

FIFO First In, First Out

NTP Network Time Protocol

PSI Parallel Snapshot Isolation

RAM Random Access Memory

RDBMS Relational Database Management System

ROWAA Read One Write All Available

RPC Remote Procedure Call

SI Snapshot Isolation

WAN Wide Area Network

YCSB Yahoo! Cloud Serving Benchmark

1 Introduction

This work focuses on providing a data storage service for distributed applications. The need for distributed storage has been increasing in the last few years as the services are being moved to the Internet and require to serve a large number of users. Moreover, with the emergence of the Cloud Computing, applications built according to this paradigm are able to provide services for a large number of users worldwide. To support this kind of operation, systems that provide data storage to these kind of applications usually need to offer consistency of the data, high service availability and scalability in order to provide a good user experience.

One way to provide availability and scalability is by the use of *replication*, which places multiple copies of the data in different servers in order to survive to server failures (availability) and to distribute the request load (scalability). On the other hand, the existence of multiple data replicas introduces a challenge which is how to maintain replicas consistent. *Distributed Consensus* (Lamport 1998) is a fundamental technique that allows to maintain replicas consistent, however this technique exhibits low performance when used with a large number of servers (exhibits low scalability). Moreover, the CAP Theorem (Brewer 2000) states that is not possible to achieve both consistency, availability and partition tolerance in a distributed system. This way, one has to choose to offer two of the three properties described (CA, CP, AP). This led to a research effort in trying to find the best tradeoff among these properties and also to maintain the scalability required by such data storage systems.

1.1 Motivation

Whit the aim of providing storage for distributed applications, in the last few years several datastore systems were proposed (Lakshman & Malik 2010; Andersen, Franklin, Kaminsky, Phanishayee, Tan, & Vasudevan 2011; Hastorun, Jampani, Kakulapati, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007; Cooper, Ramakrishnan, Srivastava, Silberstein, Bohannon, Jacobsen, Puz, Weaver, & Yerneni 2008; Terrace & Freedman 2009; Sovran, Power, Aguilera, & Li 2011; Lloyd, Freedman, Kaminsky, & Andersen 2011). These systems implement replication protocols that offer different combinations of consistency, availability and scalability guarantees. Some of them choose to offer weak consistency guarantees (for example, eventual consistency) to achieve the desired level of availability, partition tolerance and scalability, even if this imposes a burden on the application programmer as she will be aware of the existence of multiple replicas. On the other hand, other solutions offer strong consistency guarantees, such as linearizability, providing a better programming model but suffering from scalability issues.

There is a lack of solutions that try to offer a consistency model between the strongest model (linearizability) and the weaker model (eventual consistency), providing useful guarantees to the programmer while maintaining the availability and scalability required by distributed applications. Moreover, with the emergence of the Cloud Computing, there are new scenarios in the operation of distributed systems. An application developer can choose to deploy an application among multiple datacenters (that offer the cloud platform) in order to provide better access latency to local clients. However, most of the existing data storage systems are not prepared to cope with data management in such scenario. So it would be an interesting challenge to provide a system that offers a solution for both problems, which would offer a useful consistency model over a Geo-replicated deployment.

In this work we answer to this challenge by presenting a new replication technique that implements the recently formalized *causal+* consistency model (Lloyd, Freedman, Kaminsky, & Andersen 2011), which offers the causal ordering of operations while maintaining the availability and scalability required by Cloud Computing applications. We also show how this technique can be applied in a datastore system and ported to a Geo-replicated scenario.

1.2 Contributions

This work addresses the problem of providing data storage to Geo-replicated applications built according to the Cloud Computing paradigm. More precisely, the thesis analyzes, implements and evaluates techniques to provide replication while offering *causal+* consistency, high performance, and high availability. As a result, the thesis makes the following contributions:

- New replication technique, based on chain-replication, that offers *causal+* consistency;
- Architecture of a data storage system that incorporates the developed replication technique;
- Extension of the architecture for a Geo-replicated scenario;

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A prototype implementation of the described architecture;

- An experimental evaluation of the new replication technique by applying the Yahoo! Cloud Serving Benchmark to our prototype and also to Apache Cassandra (Lakshman & Malik 2010), FAWN-KV (Andersen, Franklin, Kaminsky, Phanishayee, Tan, & Vasudevan 2011), and an emulation of COPS (Lloyd, Freedman, Kaminsky, & Andersen 2011).
- An improved version of the FAWN-KV system;

1.4 Research History

This work was performed in the context of the HPCI project (High-Performance Computing over the Large-Scale Internet). One of the project main goals is to develop solutions for the large-scale distributed systems over the Internet.

In the beginning, the main focus of this work was to study the different existing consistency models and examples of replication techniques that implement them. As a result of that study this work would produce a new replication technique that would improve an existent one. However, during the bibliographic research a new consistency model (causal+) was introduced in (Lloyd, Freedman, Kaminsky, & Andersen 2011). We found this model very interesting since it offers useful guarantees to the programmers while maintaining the availability and scalability of the system, also this model could be deployed in a Geo-replicated scenario counting with multiple datacenters. During that time, we were also investigating the details behind the Chain Replication technique, which offers linearizability in a simple way while offering high performance. This way, the main idea behind this work emerged from the combination of the causal+ consistency model and the Chain Replication technique.

During my work, I benefited from the fruitful collaboration with the remaining members of the GSD team, namely João Leitão, João Paiva, Oksana Denysyuk and Miguel Branco.

A previous description of this work was published in (Almeida, Leitão, & Rodrigues 2012).

1.5 Structure of the Document

The rest of this documents is organized in the following chapters:

Chapter 2: Introduction of the main concepts related to this work. Description of the main consistency models and replication techniques. Examples of existing data storage systems.

Chapter 3: Description of the architecture of the ChainReaction system. Detailed explanation of the new replication protocol and operation on a single and multiple datacenter scenarios.

Chapter 4: Insight on the implementation details of the prototype with some details about the original implementation of FAWN-KV and its improvements.

Chapter 5: Results of the experimental evaluation of our prototype against other systems in different scenarios.

Chapter 6: Conclusion of the document and future work.

2 Related Work

This chapter surveys the main concepts and systems that are relevant for our work. We start by describing key concepts to understand the rest of the document. Then we will introduce different data consistency models and replication techniques. Finally, we describe a number of relevant systems that implement different variants and combinations of these models and replications techniques. These systems can also be classified as suitable for a single datacenter scenario or for both a single and a Geo-replication scenario.

2.1 Concepts

In this section we introduce the concepts that are required to understand the work that was developed. We start by describing the concept of a Key Value Store (or Datastore). Then we will make an introduction to the concept of Concurrency Control and Replication Management. Finally, we will present the description of Geo Replicated applications and the challenges of offering Concurrency Control and Replication mechanisms in this type of applications.

2.1.1 Key Value Stores

Key Value Store systems result from the NoSQL approach that was motivated by the fact that the traditional RDBMS (Relational Database Management System) could not achieve the scalability and performance required by the products of companies like Google, Amazon, and Facebook. In traditional SQL systems the data is structured using complex schemas, that allow for arbitrary dynamic queries over the data. The complex structure of data and of the queries requires a large amount of processing resulting in some overhead on operations. This overhead is even higher in a scenario where the data is spread among different machines, resulting in poor performance and low scalability of SQL systems. Also these systems are not adaptable for applications built according to the object-oriented paradigm resulting in additional effort from the programmer to use them. Therefore, the Key Value Stores are able to store data in a schema-less way, avoiding the complexities and overhead imposed by SQL. The data is stored in "key - value" pairs in which the value usually represents an object (of any type) and the key is usually a string that uniquely identifies the object. The basic operations of a Key Value Store are updates and queries. An update writes the value corresponding to a certain key, while a query allows to retrieve the value identified by a key. There are also some Key Value Stores that support different types of operations that allow for more complex semantics of even transactional semantics.

These systems are highly optimized for basic operations (updates and queries) achieving much better performance and scalability than SQL systems for object-oriented data models. Also, they can be easily adapted for a distributed scenario in which data is maintained in a redundant manner on several servers. Moreover, these systems can scale easily by adding more servers, which also provides fault-tolerance in case of a server failures. However, these systems usually do not offer the ACID (Atomicity, Consistency, Isolation, Durability) guarantees of traditional RDBMS, offering weaker guarantees and transactions that only include a single data item.

2.1.2 Concurrency Control

Two operations on a given object execute in serial order if one starts after the other terminates. Otherwise, the operations are said to execute concurrently. While the outcome of a serial execution of operations is often clear to the programmer, the result of a concurrent execution may not be intuitive. For instance, consider an empty FIFO queue object and a method *insert*. If "INSERT(b)" is executed *after* "INSERT(a)", it is clear that '*a*' should be at the head of the queue and '*b*' at the tail. But what are the legal outcomes when both operations are executed concurrently?

The problem is exacerbated when we consider sequences of operations, typically known as *transactions*. The concurrent execution of two transactions may yield many different interleavings of their individual operations. Many of these interleavings may result in outcomes that are not intended by the programmer of the transactional code (which assumed that her code would be executed in isolation or could not foresee all possible interleavings).

One way to avoid the problems above is to avoid concurrency, by forcing all operations and transactions to execute in serial order. Unfortunately this strategy may result in extremely long waiting times when the system is subject to high loads. Furthermore, it would prevent the infrastructure from fully using existing resources (for instance, by preventing operations or transactions from running in parallel in different cores or machines). This is unacceptable, in particular because many operations access independent data structures and so, their concurrent execution does not mutually interfere. Therefore, any practical system must support concurrent execution of operations and implement mechanisms to ensure that, despite concurrency, operations still return the "expected" outcomes.

A *data consistency model* defines which outcomes of a concurrent execution are deemed correct and which outcomes are deemed incorrect. The mechanisms required to prevent incorrect outcomes are known as *concurrency control mechanisms*. As we have noted before, it is often intuitive to programmers and users to consider that the system behaves *as if* all operations and/or transactions were executed in *some* serial order. This has led to the definition of data consistency models such as *linearizability* (for isolated operations) and *serializability* (for transactions). These models offer what is often called *strong consistency*. We will describe these models in detail later in the text.

Finally, to allow concurrent access by different threads, a system must provide some mechanism to coordinate them. These coordination mechanisms can be of two types: *pessimistic* or *optimistic*.

2.1.2.1 Pessimistic Concurrency Control

The most common way to implement pessimistic concurrency control is through the use of locking. In this approach all operations made over an object require the acquisition of a lock to the object. This grants to the thread exclusive access to a certain object, blocking the other threads that are trying to access that object. Unfortunately, the process of acquiring a lock can lead to a situation of *deadlock* in which all processes are blocked. This problem is aggravated when in distributed settings where nodes can fail, leaving objects locked. The latter issue is usually due to the complexity associated with the use of locks in programs. In order to avoid deadlocks, a programmer has to verify all possible scenarios for the execution, ensuring that two or more threads are not waiting for each other to release a certain object (*i.e.*, all threads are ensured to progress in the execution).

2.1.2.2 Optimistic Concurrency Control

Optimistic concurrency control offers a better utilization of the computational resources by allowing the execution of concurrent operations over the same objects without requiring the usage of locking mechanisms. This concurrency control model is usually applied in transactional systems and in order to achieve the consistency guarantees of serializability, the threads must check at commit time if there are no conflicts between the issued transactions. As described before, a conflict occurs in the two following situations: i) if two concurrent transactions write the same object (write-write conflict); ii) one transaction reads an object that was written by a concurrent transaction (read-write conflict). If there is a conflict, then one of the transactions must rollback its operations (*i.e.*, it is aborted). Still, the transaction that was aborted is repeated later. Unfortunately, if conflicts happen frequently, the repetition of aborted transactions hinders the performance of the system (*i.e.*, computational resources are wasted in executing operations that will be discarded). Therefore, in optimistic concurrency control, it is assumed that different threads will perform operations that do not affect each other.

2.1.3 Replication Management

Replication serves multiple purposes in a large-scale system. On one hand, it allows to distribute the load of read-only operations among multiple replicas, at the cost of requiring inter-replica coordination when updates occur. Since many workloads are read dominated, replication often pays off from the performance point of view. On the other hand, replication may increase the reliability and availability of data. If a given machine fails, data may survive if it is also stored on a different node. If data is replicated on different datacenters, it may even provide tolerance to catastrophic events that may destroy an entire datacenter (such as an earthquake, fire, etc). Additionally, by keeping multiple copies of the data in different locations, one also increases the probability that at least one of these copies is reachable in face of failures in networking components that may, for instance, partition the network.

The main challenge when using replication is how to maintain the replicas consistent. Again, as with concurrency, ideally it would be possible to hide replication from the programmers. In that case, the access to a replicated data item would be similar to the access to a non-replicated item, something known as *one-copy equivalence* (Bernstein, Hadzilacos, & Goodman 1987). Naturally, this requires that copies coordinate during write (and possibly read) operations. A fundamental coordination abstraction is *distributed consensus* (Lamport 1998), that allows to implement *state-machine replication* (Schneider 1990), a well-established strategy to keep replicas consistent.

As noted before, an essential characteristic of Geo-replicated systems is that network connections among datacenters are characterized by high latencies. This makes coordination across datacenters very expensive. Therefore, to operate with acceptable latencies, a system has to provide weaker guarantees than one-copy equivalence. Furthermore, wide area networks are subject to network partitions. In a partitioned network, replicas in different partitions cannot coordinate, and may diverge if updates are allowed. The observation of this fact has motivated the *CAP theorem* (Brewer 2000), that states that it is sometimes impossible to offer both consistency, availability, and partition tolerance. The problems underlying the CAP problem are thus another reason to offer weaker consistency guarantees in Geo-replicated systems.

2.1.4 Geo Replicated Applications

Due to the expansion of distributed applications and to the appearance of Cloud Computing, clients of a certain application or service are no longer located in a single geographic location. Applications like Facebook, Google, and Amazon Services are used by a large number of clients that may access data from different locations, thus replication allows to place data copies closer to the users, and allows them to access data with smaller latencies. It is therefore no surprise that replication is an important component of cloud computing. For example, if Google only had a datacenter on one continent the clients of distant continents would experience high latency when accessing Google services.

The use of replication and the support for concurrency brings several benefits to this type of applications. This strongly suggests that a Geo-replicated system must provide both features. On the other hand, the high-latencies observed in *Wide Area Networks* (WANs) that connect different datacenters and the occurrence of network partitions, make it difficult to support strong consistency and one-copy equivalence, as it requires tight coordination among datacenters. Therefore, one has often to impose on the programmers (and users) more relaxed consistency models that can be implemented at a smaller cost. Unfortunately, the use of weaker consistency models makes programming harder, as the allowed set of outcomes is larger, forcing the programmer to take into account a wider set of legal executions.

Moreover, the systems that are built to operate on a Geo-replicated scenario usually need to support a large number of users. These users, combined, issue thousands of requests per second that must be answered with a reasonable short delay in order to provide a good user experience. This stresses the infrastructure to operate at a very large scale while providing low latency operations. These applications require an infrastructure that allows them to manipulate the data with such low latencies. In order to satisfy these requirements, the infrastructure must employ very efficient algorithms, often resorting to the weakening of consistency guarantees.

Additionally, applications are willing to trade consistency for performance because any increase in the latency of operations could mean the loss of a percentage of their users. Therefore, a key aspect in the development of cloud-computing infrastructures is to find data storage protocols that offer a good trade-off between efficiency and consistency. This is a very active area of research, to which this work aims at contributing.

2.2 Data Consistency Models

The following sections describe various consistency models that are related with the work developed. We start by discussing models that do not consider transactions,*i.e.*, where no isolation guarantees are provided to sequences of operations. Instead, the first models only define which outcomes of individual operations are legal in face of a given interleaving of individual operations executed by different threads. Consistency models for transactional systems are surveyed in the subsequent section. Models are described by showing the outcomes of the operations that are legal under each model. Unless otherwise explicitly stated we assume that the shared data items are simple *registers*, that only export READ and WRITE operations. In a register, any update operation (*i.e.*, a WRITE) always overwrites the effect of the previous update. However, in some cases, the system also supports higher level objects with complex semantics. For instance, consider a *set* object, where items can be added using a ADD operation. In this case, the ADD operation updates the state of the object without overwriting the previous update.

As will be discussed in the following sections, some consistency models allow concurrent update operations to be executed in parallel at different replicas. As a result, replicas will have an inconsistent state if operations are executed in an uncoordinated manner. The procedure to reconcile the state of different replicas is often called *conflict resolution*. Conflict resolution may be *automatically* handled by the system or it may require the intervention of some external component. In the latter case the system may only detect the conflict and handle the different states to the external component that will be in charge of generating the new reconciled state of the object (ultimately, this may require manual intervention from the end-user).

Automatic conflict resolution is, in the general case, impossible. The most common way of performing automatic conflict resolution is to use the last-writer-wins rule (Thomas 1979). This rule states that two concurrent writes are ordered, giving the impression that one happened before the other (the last write overwrites the value written by the first). This policy is reasonable for a register object but may yield unsatisfactory results for more complex objects. For instance, consider again the *set* object. Consider a set that is initially empty and two concurrent updates add different items (say 'a' and 'b') to the set. If the set is seen as black box, the last-writer-wins rule would result in a set with only one item (albeit the same in all the replicas). However, if some operation semantics aware scheme exists, it could recognize that some operations are commutative, and merge the operations in a way that results in the same final state. In our example, the ADD operation would be recognized as commutative.

2.2.1 Non-Transactional Data Consistency Models

2.2.1.1 Linearizability

The Linearizability or Atomic Consistency model (Herlihy & Wing 1990) provides the strongest guarantees to applications. As a matter of fact, this consistency model is the one that is closer to the idealized abstraction of memory where each operation occurs instantaneously and atomically. Moreover, the model assumes that operations may take some time to execute: their duration corresponds to the interval of time between its invocation and completion. However, the system behaves as if the operation took effect instantaneously, in an atomic moment within this interval.

This means that the results of a WRITE operation are necessarily visible to all other READ operations, at most when the WRITE completes. Also, if a WRITE operation is concurrent with a READ operation, the outcome of the WRITE may or may not be visible to READ. However, if it becomes visible, then it will be visible to all other read operations that are linearized after READ. This is illustrated in Figure 2.1(a).

Furthermore, linearizability has a property known as composability (or locality). The latter states that a system is linearizable if it is linearizable with respect to each object. This means that a complex system built by composing several linearizable objects is also linearizable.

Unfortunately, to build a replicated fault-tolerant system that is linearizable is very expensive. One way to enforce linearizability is to resort to locking to perform writes, preventing concurrent reads while a write is in progress. This may avoid non-linearizable runs but may lead to poor performance and create unavailability problems in the presence of faults. Non-blocking algorithms exist to implement linearizable registers, but most of these algorithms require the reader to write-back the value read, effectively making every read as expensive as a write.

2.2.1.2 Sequential Consistency

Sequential consistency (Lamport 1979) states that the execution of multiple threads should be equivalent to the execution of a single thread which combines all the operations of the multiple threads in some serial interleaving, as long as this interleaving respects the partial order defined by each individual thread (i.e, the serial order must respect the order of the operations as they were defined in the program).

Unlike linearizability, sequential consistency is not composable. A system where operations, made over an individual object, respect the local serial consistency is not necessarily globally sequentially consistent.

Sequential consistency is weaker than linearizability, as it does not require the outcome of a write operation to become immediately visible to other read operations (*i.e.*, read operations may be serialized "in the past"), as can be seen on Figure 2.1(b). Still, it requires operations to be totally ordered, which is also very expensive in a Geo-replicated system.

2.2.1.3 Per-record Timeline Consistency

Per-record timeline consistency (Cooper, Ramakrishnan, Srivastava, Silberstein, Bohannon, Jacobsen, Puz, Weaver, & Yerneni 2008) is a relaxation of sequential consistency that implicitly makes the existence of replication visible to the programmer/user. In the first place, it only ensures sequential consistency on a per object basis. In this model, the updates made to an object have a single ordering, according to a certain timeline, that is shared by all its replicas (similar to sequential consistency). This ordering is imposed by a single replica that works as a serialization point for the updates.

This model also guarantees that the versions of the object present in a certain replica always move forward in this timeline. This way, read operations, made under this model, will return a consistent version from the ordering described above. However, and unlike sequential consistency, a read from the same thread may move back in the timeline with respect to previous reads. This may happen if the thread needs to access a different replica.

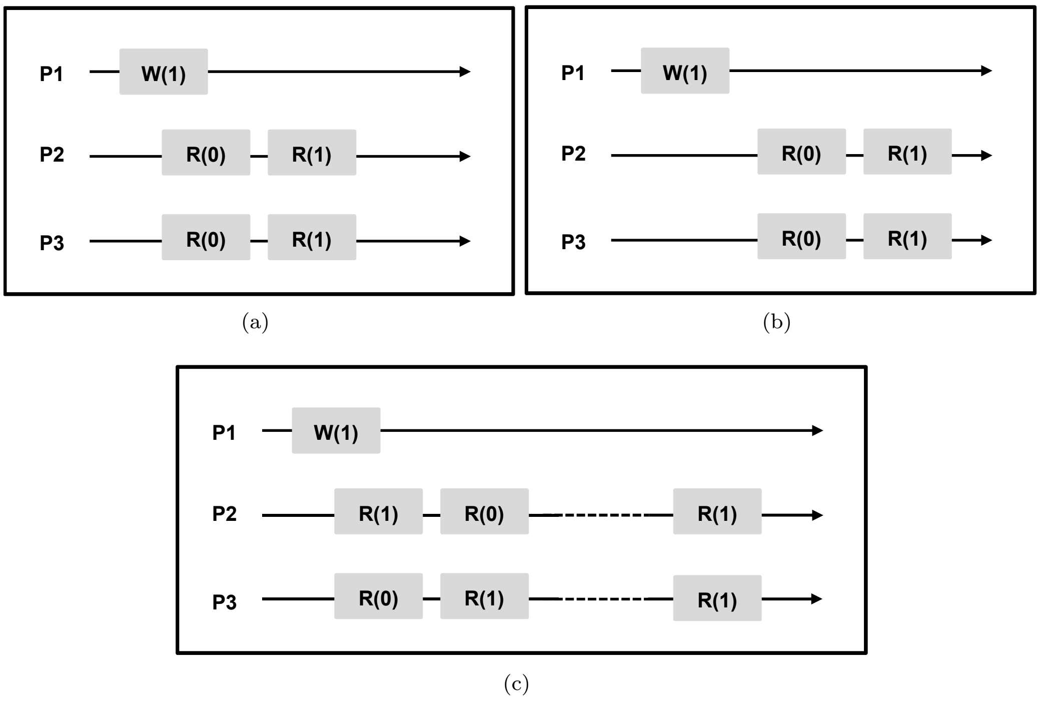


Figure 2.1: Sequences of operations that are correct (assuming that the initial value of the register is 0) under: a) Linearizability; b) Sequential Consistency (not linearizable); c) Eventual Consistency (Neither linearizable nor sequentially consistent).

The rationale for this model is that most applications access a single object and often remain connected to the same datacenter. Therefore, in most cases users will not observe the "anomalies" allowed by the model. On the other hand, the weakening of the consistency model and the absence of conflicts (*i.e.*, updates are serialized at a single node) allows for updates to be propagated in a lazy fashion among replicas.

2.2.1.4 Causal Consistency

The causal consistency model (Ahamad, Neiger, Burns, Kohli, & Hutto 1995) ensures that the observed outcomes of operations are always consistent with the "happened-before" partial order as defined by Lamport (Lamport 1978). For completeness, we reproduce here the definition of the happened before order:

- Considering that A and B are two operations that are executed by the same process, if A was executed before B then they are causally related. There is a causal order between A and B , where A *happens before* B.

- If A is a write operation and B is a read operation that returns the value written by A, where A

and B can be executed at different processes, then A happens before B in the causal order.

- Causal relations are transitive. Considering that A, B and C are operations, if A happens before B and B happens before C, then A happens before C.

Note that, as with sequential consistency, threads are not required to read the last value written, as read operations can be ordered in the past, as shown in Figure 2.2. Furthermore, and contrary to sequential consistency, concurrent writes (*i.e.*, two writes that are not causally related) may be observed by different threads in different orders

2.2.1.5 Eventual Consistency

Eventual consistency (Vogels 2009; Gustavsson & Andler 2002) is a term used to designate any consistency model that allows different threads to observe the results of update operations in different orders during some (possibly long) period. Although all threads are guaranteed to eventually read the same values, if write operations stop being executed. The time interval between the last write and the point when all replicas see that write is called *inconsistency window*. The length of this window depends on various factors like communication delays, failures, load of the system, network partitions, and *churn*. The latter refers to the phenomenon that happens when there are nodes joining and leaving the network at a very high rate. An example of a sequence of operations that is acceptable under eventual consistency can be observed in Figures 2.1(c) and 2.2.

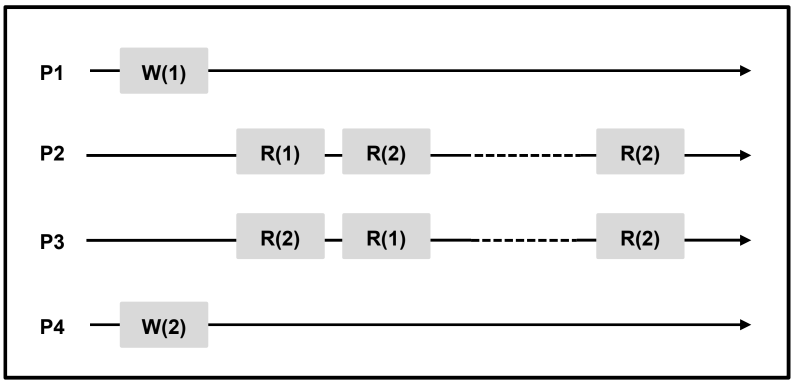


Figure 2.2: Sequence of operations that are correct (assuming that the initial value is 0) under Causal Consistency (and Eventual Consistency), although they are neither linearizable nor sequentially consistent.

2.2.1.6 Causal+ Consistency

Causal consistency guarantees that the values returned by read operations are correct according to the causal order between all operations. However, it offers no guarantees on the ordering of concurrent operations. This leads to scenario where replicas may diverge forever (Ahamad, Neiger, Burns, Kohli, & Hutto 1995). This happens in face of conflicting writes, which can be applied in different orders at different nodes.

Causal+ consistency appears as a model that aims at overcoming the drawback described above by adding the property of *convergent conflict handling.* This model has been suggested in a number of different papers (Belarami, Dahlin, Gao, Nayate, Venkataramani, Yalagandula, & Zheng 2006; Petersen, Spreitzer, Terry, Theimer, & Demers 1997) and later detailed in (Lloyd, Freedman, Kaminsky, & Andersen 2011).

In this consistency model, the enforcing of the convergence property can be made by using the last-writer-wins rule, as described before. Additionally, it allows semantic aware re-conciliation of inconsistent replicas, in which operations would be recognized as commutative.

2.2.2 Transactional Data Consistency Models

The consistency models listed previously consider operations in isolation. Transactional data consistency models consider *sequences* of operations that must be treated as a unit. These sequences of operations are named *transactions*, and are typically delimited by special operations, namely, BEGINTRANSACTION and ENDTRANSACTION. Transactions often serve a dual purpose: they are a unit of concurrency control and they are a unit of fault-tolerance.

For concurrency control purposes, a transactional system makes an attempt to isolate transactions from each other. As noted earlier in the text, the stronger consistency models ensure that the concurrent execution of transactions is the same as if they were executed one after the other (*i.e.*, in a serial execution).

For fault-tolerance, transactions are a unit of *atomicity*, *i.e.*, either all operations that constitute the transaction take effect or none does. In the former case, one says that the transaction has *committed.* In the latter case, one say that the transaction has *aborted.* A consistency model for a transactional system defines which are the valid outcomes of committed transactions.

2.2.2.1 Serializability

Serializability (Bernstein & Goodman 1981) states that the outcome of a concurrent execution of a set of transactions must be the same as the outcome of some serial execution of the same transactions. Moreover, the transactions issued by a process, which are included in this ordering, must respect the order defined in the program. A concurrent execution of transactions in a replicated system is serializable if the outcome is equivalent to that of some serial execution of the same transactions in a single replica.

Similarly to sequential consistency, serializability also requires concurrent transactions to be totally ordered, and therefore, consensus in a replicated system. Given the arguments captured by the CAP theorem (Brewer 2000), it is impossible to implement a geo-replicated system that combines consistency, availability, and partition tolerance under this model.

2.2.2.2 Snapshot Isolation

Snapshot Isolation (SI) (Berenson, Bernstein, Gray, Melton, O'Neil, & O'Neil 1995) is a weaker model of consistency that has been widely adopted by most database vendors (e.g. Oracle and SQLServer). According to this model, the state of the system at a given point in time includes all the updates of all transactions that have committed, as if these updates have been applied in some serial order. Furthermore, the outcome of each individual transaction is computed based on a valid system state, at some previous point in time (*i.e.*, the outcome of a transaction never depends on partial results from on-going transactions or from updates from transactions that later abort). Furthermore, if two transactions update a common variable, one transaction needs to be executed based on a system state that already includes the outcome of the other. However, unlike with serializability, the state observed by a transaction does not necessarily reflect the updates of all transactions that have been totally ordered in the past.

The difference between snapshot isolation and serializability may be subtle and is better illustrated with an example. Consider the following execution: two transactions concurrently read an overlapping set of values from the same system state and make *disjoint* updates to different subsets of these values. Accordingly to snapshot isolation, these transactions are allowed to commit (and their updates are applied using some serial order). However, they will commit without seeing each other's updates. This "anomaly" (with regard to the idealized system) is characteristic of snapshot isolation and it is called *write skew*. This particular example shows that SI is weaker than Serializability, as this write skew could never happen in a serializable execution.

2.2.2.3 Parallel Snapshot Isolation

Although weaker than serializability, snapshot isolation still requires that write-write conflicts are detected. Therefore, in a replicated system, a transaction cannot be committed without previous coordination among replicas. Recent work introduced a variation of this model, named Parallel Snapshot Isolation (Sovran, Power, Aguilera, & Li 2011). This model consists of a relaxation of snapshot isolation aimed at Geo-replicated systems that allows transactions to be applied in different orders at different nodes.

In PSI, each transaction is executed and committed at one site but different transactions may be executed in different sites. Therefore, under this model we can have concurrent transactions executing at the same site and concurrent transactions executing at different sites. Concurrency control operates differently in each of these cases.

In the case where concurrent transactions execute at the same site and there is a write-write conflict between them, then one of the transactions is required to abort. The decision on the transaction to be aborted depends on when the conflict is detected.

- If the conflict is detected when one of the transactions has already committed, the other must abort.

- If the conflict is detected and none of the transactions has committed yet, then one or both transactions may be required to abort, given that the decision policy is not deterministic.

On the other hand, if the conflicting transactions execute at different sites there are scenarios where both can commit. More precisely:

- If the conflict is detected when one of the transactions has already committed and the other has not, then the latter must be aborted.

- If the conflict is detected and both transactions have committed (in different sites), then the transactions must be ordered according to some criteria.

- If no conflict is detected, then both are committed and their updates applied in the same order at all sites.

Unfortunately, when conflicting transactions execute at different nodes their outcome is unpredictable as it depends on runtime conditions and implementation decisions related on how and when the system decides to propagate information about on-going and committed transactions among the different replicas.

2.3 Replication Techniques

The benefits of replication, as described previously, led to the appearance of a large number of replication approaches that offer different combinations of consistency guarantees and performance guarantees. The following sections introduce the replication mechanisms that we consider relevant for the work developed: *active replication*, *quorum replication*, *passive replication* (and its variants), *chain replication*, and *lazy replication*.

2.3.1 Active Replication

In active replication, requests made to the system are processed by all replicas. This requires that processes execute operations in a deterministic and synchronized way, receiving the same sequence of operations. In order to respect the previous constraint, the system must make use of an *atomic broadcast* protocol that guarantees that all replicas receive the messages in the same order and if one process receives the message then all the others receive it. The use of atomic broadcast hinders the scalability of the system since it must use a distributed consensus mechanism (Lamport 1998), which requires heavy communication.

2.3.2 Quorum Replication

Quorum replication is based on Quorum consensus (Agrawal & El Abbadi 1991; Malkhi & Reiter 1997) where requests are processed by a quorum of replicas before returning to the client. A simple quorum (Q) is defined as any majority of nodes such that $Q > N/2$, where N is the total number of replicas. In the ideal case all operations would be processed in a majority before returning to the client in order to guarantee that each client always access the most recent version in the system. However, some systems allow the configuration of the size of read (R) and write (W) quorums in order to achieve better performance. To ensure the same properties as when a majority is used, the quorums must intercept in some node such that $W + R > N$. Otherwise, the quorums may not overlap and stale versions can be observed.

2.3.3 Passive replication

Unlike the latter, the passive replication mechanism assumes that operations are made over a single replica. The *primary-backup replication*, and *multi-master replication* are two examples of passive replication mechanisms that we consider relevant for our work. These two solutions are based on the read-one-write-all-available (ROWAA) approach (Charron-Bost, Pedone, & Schiper 2010), in which operations are assigned to a local site where they are executed and later propagated to remote sites. This way, these two mechanisms can be classified using the following parameters: *transaction location* and *synchronization strategy.*

Primary-backup replication: According to the parameters defined above, in the primary-backup approach updates are made on a single replica (master) and then propagated to the remaining replicas, while read operations can be directed to any node. This propagation can be either made in an *eager* or *lazy* way. In the eager approach, the client issues an update and the master only replies after the update has been propagated to all the replicas, often providing strong consistency.

On the other hand, the lazy approach assumes that the master replies to the client after the update has been applied locally. This way, if the client issues a read operation to one of the other replicas, before the propagation of the update, she can see stale data. Additionally, in the primary-backup approach it is simple to control the concurrency of updates (serialized at the master), although a single master can be seen as a bottleneck in the system.

Multi-master replication: In multi-master replication the write and read operations can be made at any replica. Like in primary-backup, the propagation of updates is made in a similar way, either in the lazy or eager approach. However, in the eager approach the replicas must synchronize in order to decide on a single ordering of concurrent updates, which can lead to heavy communication. In the lazy approach, there are some situations where conflicts can occur, which can be hard to solve as was described before. Considering the previous description, we can state that the multi-master approach is more flexible than primary-backup and allows a better distribution of write operations among the replicas. Unfortunately, the existence of multiple masters imposes a higher level of synchronization.

2.3.4 Chain Replication

Chain Replication (van Renesse & Schneider 2004) is a specialization of the primary-backup approach that allows to build a datastore that provides linearizability, high throughput, and availability. This approach assumes that replicas are organized in a *chain topology.* A chain consists of a set of nodes where each node has a successor and a predecessor except for the first (head) and last (tail) nodes of the chain.

The write operations are directed to the head of the chain and are propagated until they reach the tail. At this point the tail sends a reply to the client and the write finishes. The way updates are propagated along the chain ensures the following property, named *Update Propagation Invariant*: the set of received updates by given node is contained on the set of updates of its predecessor in the chain. Contrary to write operations, read operations are always routed to the tail which returns the value of the object. Since all the values stored in the tail are guaranteed to have been propagated to all replicas, reads are always consistent.

The failure of a node can break the chain and render the system unavailable. To avoid this problem, chain replication provides a fault-tolerance mechanism that recovers from node failures in three different situations. This mechanism assumes the fail-stop model (Schneider 1984) and operates as follows:

- If the head (H) of the chain fails, then the head is replaced by its successor (H^+). All the pending updates, that were in H, and were not propagated to H^+ are assumed to have failed.

- The failure of the tail is handled by replacing it by its predecessor. By the property of update propagation invariant is guaranteed that the new tail has all the needed information to continue on normal operation.

- Unlike the failures described above, the failure of a node in the middle of the chain (S) is not simple to handle. To remove S from the chain, its successor (S^+) and its predecessor (S^-) must be informed. Also, it must be guaranteed that the updates propagated to S are propagated to S^+, so that the property of update propagation invariant is not violated.

Failed nodes are removed from the chain, which shortens its size. Obviously, a chain with a small number of nodes tolerates fewer failures. To overcome this problem, chain replication also offers a mechanism to add nodes to a chain. The simplest way of doing this procedure is to add a node after the tail (T). In order to add a node T^+, the current tail needs to propagate all the seen updates to T^+, and is later notified that is no longer the tail.

Chain replication is considered as a Read One, Write All Available (ROWAA) approach, which implements linearizability in a very simple way. However, chain replication is not applicable to partitioned operation and has some disadvantages, as follows; The overloading of the tail with write and read operations, which hinders throughput and constitutes a bottleneck in the system; The failure of just one server makes the throughput of read and write operations, drop to zero until the chain is repaired.

2.3.5 Lazy Replication

Lazy replication, introduced in (Ladin, Liskov, Shrira, & Ghemawat 1992), is considered as an example of a multi-master approach that allows to build causal consistent distributed systems. This technique allows for write operations to be processed at a single replca before returning to the client. The rest of the replicas are updated in the background using *lazy* gossip messages. Therefore, it aims at providing better performance and scalability at the cost of providing weaker consistency guarantees.

The causal consistency model is implemented by using *multipart timestamps*, which are assigned to each object. These timestamps consist on a vector that contains the version number of the object at each replica. This vector must be included (in the form of dependencies) in all update and query requests so that operations respect the causal ordering: an operation can only be processed at a given node if all the operations on which it depends have already been applied at that node. Furthermore, the nodes must be aware of the versions present in other replicas. This information is maintained in a table that is updated by exchanging gossip messages between replicas. When a node is aware that a version as already reached all replicas then it issues an acknowledgment message so that the gossip dissemination protocol can be stopped.

The causal consistency guarantees provided by this approach allow its adaptation to a Geo-replicated scenario while maintaining scalability. Moreover, this technique is able to handle network partitions since the replicas can synchronize the updates when network recovers from the partition, without affecting the operation of the system. However, this approach has a major drawback: large vector timestamps. These timestamps must be stored in the client and can be very large in a system that comprises hundreds of nodes.

2.4 Existing Systems

This section will introduce some systems that are relevant for the work that is going to be developed. We pretend to illustrate the implementation of some consistency models and replication techniques described before. First we present some key value store systems that are configured to operate on a single datacenter (*i.e.*, the systems are built assuming that the nodes are connected by low latency links) but that do not support the operation on multiple datacenters. Next, we present the systems that can be deployed in both a single and a multi-datacenter scenario.

2.4.1 Single Datacenter Systems

2.4.1.1 Fast Array of Wimpy Nodes (FAWN-KV)

FAWN-KV (Andersen, Franklin, Kaminsky, Phanishayee, Tan, & Vasudevan 2011) is a distributed key-value storage based on the FAWN architecture which allows to build systems for low-power data-intensive computing. This datastore also offers strong consistency, high availability, and high performance.

The FAWN architecture is composed by two types of server nodes: *front-ends* and *back-ends*. The front-ends are the entry point of the system, forwarding all client requests to the back-end FAWN-KV node responsible for a certain key. The back-end FAWN-KV storage nodes are organized in a one-hop DHT ring using consistent hashing (Karger, Lehman, Leighton, Panigrahy, Levine, & Lewin 1997), where the keys are mapped to the successor of the key in the ring. The storage of data is managed by a FAWN-DS (FAWN datastore) module that is optimized for flash memory.

To offer high availability guarantees, FAWN-KV uses Chain Replication in order to maintain multiple replicas of the data consistent. In this system, a chain is defined by the FAWN-KV storage node responsible for the key (head) and its R - 1 successors, where R is the configured replication factor. Since the chain are defined in the DHT ring, a node can be part of different chains.

The use of flash memory and of the Chain Replication technique allow FAWN-KV to achieve high throughput in operations while maintaining a low power consumption. However, this solution is not adaptable for a Geo-replication scenario with hundreds of nodes spread among different datacenters. The latter is due to the fact that chains can have internal links that cross the WAN usually with high latencies.

2.4.1.2 Chain Replication with Apportioned Queries (CRAQ)

Chain Replication with Apportioned Queries (CRAQ) (Terrace & Freedman 2009) is a distributed key value storage system that offers strong consistency, high availability, and high performance.

This system employs a replication technique that is an improvement of Chain Replication, allowing for a better throughput of read operations by distributing the load among the existing replicas. Like in chain replication, nodes are organized in a chain and all write operations are routed to the head of the chain and then propagated until the tail. Moreover, when the version written reaches the tail, the version is said to be *committed* and an acknowledgment message is sent upwards in the chain. However, unlike chain replication, all nodes can serve read operations in the following way: If the last version of the object stored in the node is committed, then the node can return that version; Otherwise, if the last version of the object is not committed, then the node contacts the tail asking the latest version that has been committed. Upon receiving the response, the node returns the version specified in the tail's response. This way, all operations are serialized with respect to the tail offering linearizable guarantees. Furthermore, CRAQ can be configured to offer eventual consistency guarantees where the nodes can return the latest version that is stored there, even if it is not committed (without contacting the tail).

This optimization of the Chain Replication protocol allows CRAQ to achieve better throughput, in read-heavy workloads, than the original protocol by distributing the read load among the multiple replicas. However, the major drawback of this solution is that a small percentage of write operations forces all nodes to contact the tail in read operations, reducing the throughput to values that are very close to the original protocol.

2.4.2 Multiple Datacenter Systems

2.4.2.1 Cassandra

Cassandra (Lakshman & Malik 2010) is a distributed data storage system developed by Facebook to overcome the storage needs of the Inbox Search problem. Due to the large (and growing) number of users in Facebook, this key value store has to operate at a high scale. This way, Cassandra was meant to be run on hundreds of nodes providing high availability and high scalability.

Like most key value stores, Cassandra does not support a relational data model. However, its data model is more complex than simple key-value pairs. In Cassandra the data is stored in tables that consists on distributed multi dimensional maps that are indexed by a key. The value is a highly structured object which consists on a row of the table. This row has multiple *Columns* that can be grouped together in *Column Families*. The latter can also be divided in two kinds: *Simple* and *Super* Column Families. Super Column Families can be seen as a column family inside another column family. Cassandra also allows applications to specify the order of columns inside the column family.

To be able to scale incrementally, the data is partitioned among the nodes using consistent hashing (Karger, Lehman, Leighton, Panigrahy, Levine, & Lewin 1997). In consistent hashing the output values of the hash function form a ring space. Nodes are positioned in this ring by assigning a random hash value in this space. Data items are assigned to nodes by hashing its key (which results in a position in the ring) and finding the first node with a position larger than the item position.

In order to achieve the high availability, Cassandra replicates its data items among N replicas, where N is the configured replication factor. In the common case a data item is replicated in the N-1 successors of the node responsible for the key. Cassandra uses a quorum technique to maintain replicas up to date, however Cassandra only provides weak consistency guarantees, namely eventual consistency. Moreover, it offers replication mechanisms that are datacenter aware allowing for a better adaptation to a Geo-replication scenario. Examples of these mechanisms are datacenter-aware replication, strategies, configurable quorums at each datacenter, among other features.

2.4.2.2 Clusters of Order-Preserving Servers (COPS)

COPS (Lloyd, Freedman, Kaminsky, & Andersen 2011) is a datastore designed to provide causal+ consistency guarantees while providing high scalability over the wide-area. It provides the aforementioned guarantees through the use of *operation dependencies*. The dependencies of an operation are the previous operations that must take place before the former, in order to maintain the causal order of operations. These dependencies are included in the read/write requests issued by a client. Operations can only take place if the version of an object, present in a datacenter, fulfills the dependencies. Otherwise, operations are delayed until the needed version is written in the datacenter.

The authors argue that COPS is the first system to achieve causal+ consistency in a scalable way. Previous work on this topic (Petersen, Spreitzer, Terry, Theimer, & Demers 1997; Belarami, Dahlin, Gao, Nayate, Venkataramani, Yalagandula, & Zheng 2006) described the implementation of systems that provide causal+ consistency. However, these systems do not scale and are not suitable for a Geo-replication scenario. COPS also introduces a new type of operations named as *get-transactions*. These operations allow a client to read a set of keys ensuring that the dependencies of all keys have been met before the values are returned. The usefulness of these operations can be better assessed with resort to an example: imagine that we issue two writes to objects A and B in a sequential order, however it could happen that the write on B is propagated faster among the replicas. Considering this, if we issue two consecutive reads to A and B it could happen that we see the old value of A and the new value of B, which is correct according to the causal ordering of operations but is not desirable in some applications. To overcome this situation we could use the get-transaction construct, which imposes that if we read the new version of B we must also read the new version of A (because the write on A happens-before the write on B). Additionally, in the case that we read the old version of B, then either the old or the new version of A can be observed.

2.4.2.3 Walter

Recent work on Geo-replication introduced a new key-value store that supports transactions, known as Walter (Sovran, Power, Aguilera, & Li 2011). This datastore implements parallel snapshot isolation (described previously) allowing the transactions to be asynchronously propagated after being committed at one site, by a central server. Transactions are ordered with resort to vector timestamps, assigned at their beginning, which contains an entry for each site. Each vector timestamp represents a snapshot of the data in the system.

Walter also introduces two novel mechanisms that are also used to implement PSI: preferred sites and counting sets. Each object has a preferred site that corresponds to the datacenter closer to the owner of that object. This enables transactions that are local to the preferred site to be committed more efficiently using a *Fast Commit* protocol. This protocol allows for a transaction to commit at the preferred site without contacting other sites. To commit a transaction in this way, Walter must perform two checks: check if all objects in the write-set have not been modified since the transaction started; check if object in the write-set are unlocked. If the two conditions are verified then the transaction can be committed. However, if the transaction is not local it must execute a *Slow Commit* protocol. This protocol consists in a two-phase commit protocol between the preferred sites of the objects being written.

The counting sets are a new data type, similar to commutative replicated data types (Shapiro & Preguiça 2007), that allows to avoid write-write conflicts. Operations where counting sets are acceptable can be quickly committed using the Fast Commit protocol, improving the throughput in those cases.

Walter was designed to provide PSI in a scalable way considering a Geo-replicated scenario. However, there are some factors that can hinder its scalability. The fact that transactions are executed and committed by a central server at each site corresponds to a bottleneck in performance. Moreover, if transactions are not local, then they must be committed by the Slow Commit protocol, which can limit the throughput and scalability.

2.4.2.4 Amazon's Dynamo

Amazon has many services that must be highly available and scalable in order to provide a good user experience. One of the components that helps satisfying these requirements is the Dynamo key-value store (Hastorun, Jampani, Kakulapati, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007). This store is based on a Distributed Hash Table that implements eventual consistency.

In Dynamo read and write operations are made over a number of replicas that represent a majority (quorum technique), which only offers eventual consistency. The quorum size can be configured by applications, however the number of read and write replicas must be higher than the existing replicas (*i.e.*, read and write quorums must overlap). Each write creates a new version that corresponds to a vector clock, which is used in read operations to obtain the most recent version in a read quorum.

Unlike most datastores, Dynamo was designed to provide high write availability. Therefore, it allows multiple writes at multiple replicas resulting in write-write conflicts. These conflicts can be handled either automatically or by notifying the application, that then solves the conflict by other means.

2.4.2.5 Google's Spanner

Google has recently introduced Spanner (Corbett, Deana, Epstein, Fikes, Frost, Furman, Ghemawat, Gubarev, Heiser, Hochschild, Hsieh, Kanthak, Kogan, Li, Lloyd, Melnik, Mwaura, Nagle, Quinlan, Rao, Rolig, Saito, Szymaniak, Taylor, Wang, & Woodfordh 2012), a scalable, multi-version and globally distributed datastore. This system has evolved from Bigtable (Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber 2008) and has the purpose of covering the flaws of Google Megastore (Baker, Bond, Corbett, Furman, Khorlin, Larson, Leon, Li, Lloyd, & Yushprakh 2011), which provides low write performance over the wide-area.

Spanner shards data across many replicas all over the world in order to provide global availability and geographic locality to clients. The data shards are maintained in many Paxos (Lamport 1998) state machines that are responsible for guaranteeing the consistency of the data. This system also provides linearizable transactions by using globally-meaningful commit timestamps that guarantee the total order of operations. These timestamps are based on uncertainty bounds and are assigned by a service called TrueTime, which makes use of GPS and atomic clocks as reference.

The system offers three types of operations to clients: read-write transactions, read-only transactions, and non-blocking reads. Read-write transactions are serialized at a leader replica (Paxos leader) and require pessimistic concurrency control. Read-only transactions have similar performance to transactions in systems that offer snapshot isolation and can be executed using lock-free mechanisms. This type of transactions are similar to the ones provided by COPS (although the consistency guarantees offered are different). Finally, the non-blocking reads allow for a client to obtain a snapshot of the database in the past. The acceptable staleness of the read is defined by the client through a timestamp or by an upper bound of staleness.

2.4.3 Discussion and Comparison

The previous section introduced a range of existing solutions that offer distributed storage services. These systems were listed as Single Datacenter systems and Multiple Datacenter systems according to their capability of being used in a Geo-replicated scenario or not. Moreover, systems that are considered Multiple Datacenter systems can also be deployed in a single datacenter. Additionally, these systems implement different consistency models, concurrency control mechanisms, and replication techniques, as can be seen in Table 2.1.

As shown in the table, most of these solutions tend to use optimistic concurrency control in order to exploit the maximum usage of the existing resources. The studied systems that use pessimistic concurrency control (FAWN and CRAQ) require that all writes are serialized at a single replica (avoiding the existence of concurrent updates). According to the latter, the original Chain Replication technique is not suitable for a Geo-replicated scenario due to the low scalability of the approach.

Apache Cassandra and Amazon's Dynamo use the Quorum replication technique which can be configured to offer different guarantees in read and write operations, however they only offer eventual consistency guarantees to exploit maximum performance and scalability. The CRAQ system, which employs an optimization to Chain Replication, can be configured to offer eventual or linearizable consistency guarantees. On the other hand, FAWN can only offer Linearizability guarantees resulting in reduced scalability and inadaptability to a Geo-replication scenario. Google's Spanner is the first system to offer linearizable guarantees on transactions over the wide-area with an acceptable performance.

According to the latter one can observe that the majority of the presented systems offer linearizability or eventual consistency guarantees (some offer both configurations). Most existing systems are also categorized in one of these extremes: systems that offer linearizability (with reduced scalability) and systems that offer weaker guarantees (usually eventual consistency) but exhibit high performance and scalability. Therefore, there is a lack of solutions that try to fit somewhere between these two extremes. Only recently COPS and Walter, which try to overcome this problem, were introduced. Both systems provide useful guarantees to the programmer (causal consistency and PSI) while maintaining the scalability required by large scale applications.

Systems	Suitable for Geo-replication	Consistency Model	Concurrency Control		Replication Technique
			Optimistic	Pessimistic	
Amazon's Dynamo	✓	Eventual	✓		Quorum
Apache Cassandra	✓	Eventual	✓		Quorum
COPS	✓	Causal+	✓		Chain/Lazy
CRAQ		Eventual/ Linearizability		✓	Chain
FAWN		Linearizability		✓	Chain
Walter	✓	PSI	✓		Lazy
Google's Spanner	✓	Linearizability	Operation Dependent		Active

Table 2.1: Comparison between the presented systems.

Summary

In this chapter we introduced the related work that we consider important for our work. We started by introducing some basic concepts that are required to understand the rest of the document: the concept of a key-value storage (or datastore), the concepts of concurrency control and replication management, and more important, the challenges behind building applications that are Geo-replicated. Then we introduced some consistency models that were divided in non-transactional and transactional consistency models. The main focus was on the non-transactional models since they are intimately related to our work. We also described some existing replication techniques with a special focus on Chain Replication since our solutions is based on this technique. Finally, we presented and compared some key-value storage systems and divided them in systems that are able to support Geo-replication and systems that are not. We are particularly interested in systems that can support multiple datacenters as our contribution is in this area.

The next chapter will introduce the architecture and operation details of our solution in both single and multiple datacenter scenarios, named ChainReaction.

3 Chain Reaction

In this chapter we present the work developed in the scope of this thesis. As a result of our work, we designed a new key-value storage system called ChainReaction. Our system employs a new replication technique which is based in a variation of chain replication, offering causal+ consistency and an improved performance. We start by describing the system model focusing on the chosen consistency model. Next we will present an overview of our system's architecture, which is based on the FAWN architecture, and explain in detail each component. Then we will provide a detailed explanation of our system operation on a single and multi datacenter scenario. Finally, we present the extension of our solution to provide a transactional primitive that allows for a client to obtain a consistent snapshot of a set of objects.

3.1 System Model

ChainReaction is a distributed key-value store built to provide causal+ consistency, high availability and high scalability. Our solution can be deployed in a single datacenter scenario, only with local data replicas, and in a multiple datacenter scenario, where data is replicated both locally and among different geographic locations (Geo-replication). We intend to provide a data storage service to clients spread among different locations. In our system, a client can be an end-user application (for instance a browser) or an application server that uses our system as a data storage layer of a bigger application. Next we describe the consistency model being offered by our system and also the operations offered to the clients.

3.1.1 Consistency Model

As described on the previous chapter, linearizability, serializability, sequential consistency, and snapshot isolation provide a set of guarantees that are most programmer-friendly, since they allow the programmer to build applications without having to worry about concurrency and replication issues. However, these models offer low scalability when implemented over the wide area, because they require the use of a expensive construct (Distributed Consensus) to maintain replicas consistent. Other consistency models like causal, causal+, and PSI provide weaker guarantees than the models above. Although, they provide some guarantees to the programmer while maintaining scalability and performance, making them attractive for Geo-replicated scenarios.

Hereupon, we have opted to offer the *causal+* consistency model (Petersen, Spreitzer, Terry, Theimer, & Demers 1997; Belarami, Dahlin, Gao, Nayate, Venkataramani, Yalagandula, & Zheng 2006; Lloyd, Freedman, Kaminsky, & Andersen 2011). This model is supported both at the level of a single datacenter and across multiple datacenters. We have selected *causal+* because it provides a good tradeoff among consistency and performance. Contrary to linearizability, *causal+* allows for a reasonable amount of parallelism in the processing of concurrent requests. Still, *causal+* ensures that concurrent write operations are totally ordered and that the most recent value is eventually applied to all replicas, thus avoiding the existence of divergent replicas (a common problem of causal consistency). On the other hand, and in opposition to eventual consistency, it provides precise guarantees about the state observed by applications.

The *causal+* consistency guarantees and scalability offered by COPS are a result of the maintenance of metadada in the clients, a strategy that we also follow. However, contrary to COPS, we do not require each individual datacenter to offer linearizability (in (Lloyd, Freedman, Kaminsky, & Andersen 2011) the authors rely on a classical chain-replication solution to provide this) as this requirement imposes a significant overhead.

3.1.2 Client API - Client Application Programming Interface

The basic API offered by ChainReaction is similar to that of most existing distributed key-value storage systems. The operations available for clients are described as follows:

- PUT (key, val): A PUT operation allows to assign (write) the value *val* to an object identified by *key*. According to the specification described in the following section, write operations are always made over the newest version of the object.
- val ← GET (key): The GET operation returns (reads) the value of the object identified by the *key*, reflecting the outcome of previous PUT operations.
- {val1, ..., valN} ← GET-TRANSACTION (key1, ..., keyN): This type of operation is similar to GET-TRANSACTIONS provided by COPS (Lloyd, Freedman, Kaminsky, & Andersen 2011), which allows an application to obtain a consistent snapshot of a set of keys.

3.2 System Architecture

Our architecture is similar to that of the FAWN-KV system (Andersen, Franklin, Kaminsky, Phanishayee, Tan, & Vasudevan 2011). We consider that each datacenter is composed of multiple *data servers* (back-ends) and multiple *client proxies* (front-ends). Data servers are responsible for serving read and write requests for one or more data items. Client proxies receive the requests from end-users (for instance a browser) and redirect the requests to the appropriate data server. In order to allow end-users to access our system (using the API described above), the architecture includes a Client Library that is deployed on the client side. This library will be explained in greater detail in the following sections. An overview of ChainReaction can be observed in Figure 3.1.

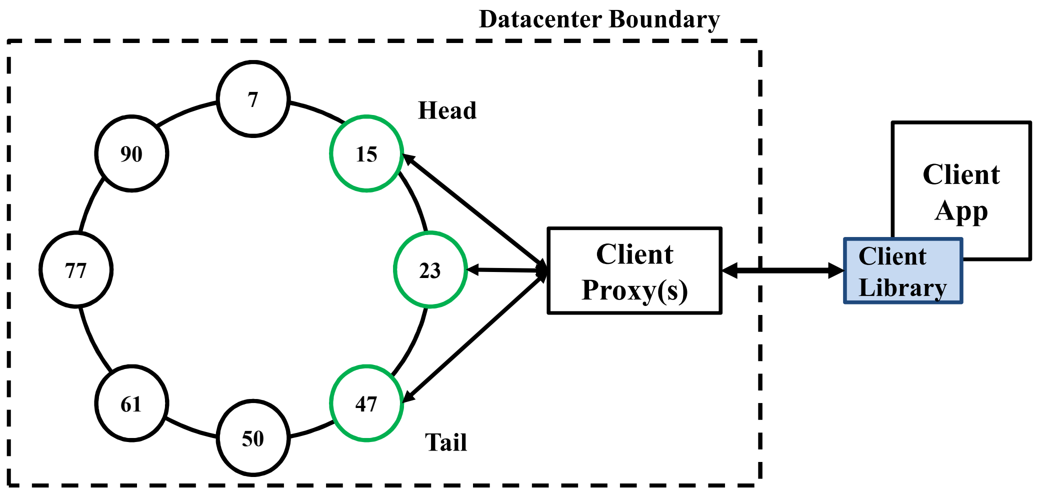


Figure 3.1: Overview of ChainReaction architecture.

Data servers self-organize in a DHT ring such that consistent hashing can be used to assign data items to data servers. Each data item is placed in R consecutive data servers in the DHT ring. Servers that store a given data item execute the chain-replication protocol to keep the copies of the data consistent: the first node in the ring serving the data item acts as head of the chain and the last node acts as tail of the chain. Note that, since consistent hashing is used, a data server may serve multiple data items and, thus, being a member of multiple chains. Also, its role in the chain may be different for each item: it can serve as the head of the chain for an item, and a middle node for some other item, and a tail for yet another item.

We further assume that, in each datacenter, the number of servers, although large, can be maintained in a one-hop DHT (Lesniewski-Laas 2008). Therefore, each node in the system, including the client proxies, can always locally map keys to servers without resorting to an external directory service. Contrary to data servers, the client proxies are not maintained in a structure since a proxy does not need to contact or know about the existence of other proxies (*i.e.*, the number of client proxies depends on the desired level of load distribution).

Considering the architecture above, we now describe the lifecycle of a typical request in the FAWN-KV system which employs a classical chain-replication solution. ChainReaction uses a variant of this workflow that will be explained in the next subsections. The client request is received by a client proxy. The proxy uses consistent hashing to select the first server to process the request: if it is a write request it is forwarded to the head data server of the corresponding chain; if it is a read request, it is forwarded directly to the tail data server. In the write case, the request is processed by the head and then forwarded to the next node in the chain, and then forwarded "down" in the chain until it reaches the tail. For both read and write operations the tail sends the reply to the proxy which, in turn, forwards an answer back to the source of the request.

3.2.1 A Chain Replication Variant

The operation of the original chain replication protocol, briefly sketched in the previous chapter, is able to offer linearizable executions. In fact, all operations (*i.e.*, both read and write operations) need to be processed by (and are serialized by) a single node, the tail of the chain. The drawback of this approach is that the availability of multiple replicas is not leveraged to promote load balancing among concurrent read operations.

In ChainReaction we decided to provide *causal+* consistency as this allows us to make a much better utilization of the resources required to ensure fault-tolerance and also to add additional replicas, that can be used for load balancing, at little additional cost.

Our approach departs from the following observation: if a node x, in the chain, is causally consistent with respect to some client operations, then all nodes that are predecessors of x in the chain are also causally consistent. This property trivially derives from the update invariant of the original chain replication protocol. Therefore, assume that a node observes a value returned by node x for a given object O, as a result of a read or a write operation *op*. Future read operations over O that causally depend on *op*, in order to obtain a consistent state (according to the *causal+* criteria) are constrained to read from any replica between the head of the chain and server x. However, as soon as the operation *op* becomes stable (*i.e.*, when it reaches the tail), new operations over O that causally depend on *op* are no longer constrained, and can get a consistent state by reading *any* server in the chain. This behavior can be observed in Figure 3.2, in this case version 2 of object O has reached the first three nodes while version 1 already reached all nodes. So, if the client has seen version 2 he can read from any node between the TAIL and node 2, otherwise version 1 can be read in any node of the chain.

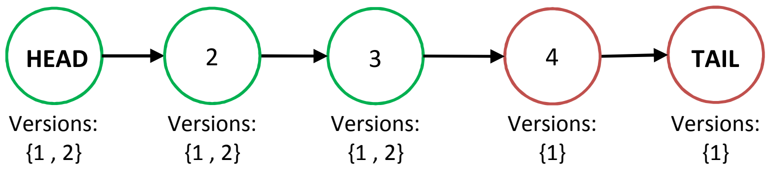


Figure 3.2: Example of version propagation in a chain for object O. The first three nodes contain version 1 and 2 of object O while the two last nodes only know version 1 (not yet propagated to these nodes).

ChainReaction uses this insight to distribute the load of concurrent read requests among all replicas.

Furthermore, it permits to extend the chain (in order to have additional replicas for load balancing) without increasing the latency of write operations, by allowing the later operations to return as soon as they are processed by the first k replicas (where k defines the fault-tolerance of the chain, k is usually lower than the total number of replicas). The propagation of operations from node k until the tail of the chain can be performed lazily, and its only purpose is to augment the number of legal targets for read operations.

To ensure the correctness of read operations according to the *causal+* consistency model across multiple objects, clients are required to know the position in the chain of the node that processed its last read request for each object they have read. This information is stored in the form of *metadata* entries which in turn are stored by a *client library*, which is responsible for managing all metadata from the client side. Additionally, we ensure that the results of write operations only become visible when all their causal dependencies have become stable in a datacenter. This allows the version staleness to be only one level deep, which avoids violation of the causal order when accessing multiple objects. Figure 3.2 shows that version 2 can only be propagated after version 1 has reached the tail and future writes must wait for version 2 to reach the tail, also there are only nodes with one level of staleness with respect to object O versions. In the following subsections we discuss the ChainReaction client library and provide a detailed description of the ChainReaction Key-Value Store interface.

3.2.2 Client Library and Interface

Each client of our system must make use of a client library that is responsible for managing client metadata, which is then automatically added to requests and extracted from replies. When considering a system deployed over a single datacenter, the metadata stored by the client library consists on a *Index* table and an *Accessed Objects* list. The table includes one entry for each accessed object. Each entry comprises a tuple on the form (*key, version*, *chainIndex*). The *chainIndex* consists of an identifier that captures the position in the chain of the node that processed and replied to the last request issued by the client for the object to which the metadata refers. The list includes a reference to all versions that were accessed between two write operations (including the last write). Notice that the list only contains a single version per object, meaning that if the client accesses a newer version of a certain object it will replace the previous version in the list (*i.e.*, if a newer version was observed it means that the previous version is already stable). Also, the list does not contain any version of an object that is known to be already stable. Figures 3.3(a) and 3.3(b) show an overview of the metadata structures maintained in the client library.

When a client makes a read operation on a data item identified by *key*, it must present the *chainIndex* that corresponds to the key. However, if there is no entry on the table for the object being read, then no metadata is sent (*i.e.*, the client has not seen any version of the object so it is allowed to read on any data

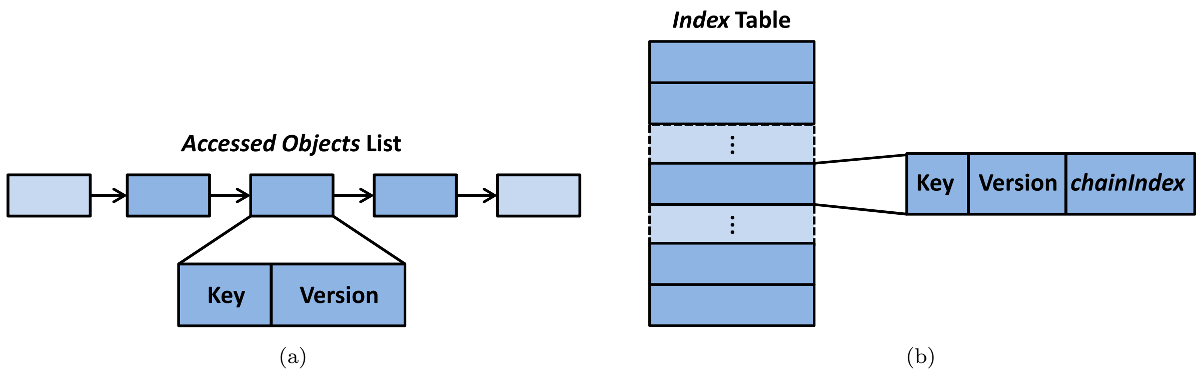


Figure 3.3: Ilustration of the structures used in the Client Library to maintain metadata: a) *Accessed Objects* List with a zoomed list member; b) *Index* Table with a zoomed table entry.

server). On write operations, the client library includes the access list in the request and upon receiving the response resets the list. The library then includes the version written in the access list. Furthermore, ChainReaction can update the table and the access list as a result of executing read or write operations.

In order to forward the requests to the ChainReaction Key-Value Store, the client library must use the following interface provided by the client proxies (front-ends):

- {version, metadata} ← PUT (key, val, metadata)
- {val, version, metadata} ← GET (key, metadata)
- {val1, ..., valN, ver1, ..., verN, meta1, ..., metaN} ← GET-TRANSACTION(key1, ..., keyN)

This interface is a specialization of the Client API which includes the versions and metadata related to a certain object. The information enclosed in the metadata field depends of the deployment of the solution being used (single or multiple datacenter) as it will be explained in the following sections.

3.3 Operation in a Single Site

In this section we describe how ChainReaction operates in a single datacenter scenario. In this scenario we assume that all data servers (back-ends) and client proxies (front-ends) are enclosed in the same datacenter. This way, the latency between servers is very low and the network bandwidth is very large. The nodes inside the datacenter are organized as described before, *i.e.*, data servers self-organize in a local DHT and chains are defined using consistent hashing for maintaining local copies of the data items. In this scenario the object versions are identified by a single integer and the metadata returned only reflects the state of the local (and only) datacenter.

3.3.1 Put Operation Processing

We now provide a detailed description on how PUT operations are executed in ChainReaction. When a client issues a PUT operation using the Client API, the client library makes a request to a client proxy including the *key* and the value *val*. The client library also tags this request with the metadata relative to the last PUT performed by that client as well as the metadata that relate to the GET operations performed over objects since that PUT. Metadata is only maintained for objects whose version is not stable yet; stable versions do not put constraints on the execution of PUT or GET operations (we discuss GET operation further ahead). Because we aim at boosting the performance of read operations while ensuring *causal+* consistency guarantees, we have opted to delay (slightly) the execution of PUT operations on chains, as to ensure that the version of any object from which the current PUT casually depends has become stable in its respective chain (*i.e.*, the version has been applied to the respective tail). This ensures that no client is able to read versions of two distinct objects that may violate causal dependencies.

Before forwarding the PUT operation to the appropriate dataserver, the proxy performs a *dependency stabilization procedure*. This procedure aims at ensuring that all objects from which the PUT operation causally depends (*i.e.*, all previous object versions accessed by the client since the last PUT operation) are already stable before the new operation is applied. As noted before, the purpose of this phase is to limit the amount of metadata that needs to be maintained and exchanged in the system, as also to make reads more efficient at the cost of slighter slower writes. Dependency stabilization is implemented by reading all the versions in the causal past from the tails of the corresponding chains (this may involve to wait until such versions are propagated in the corresponding chains).

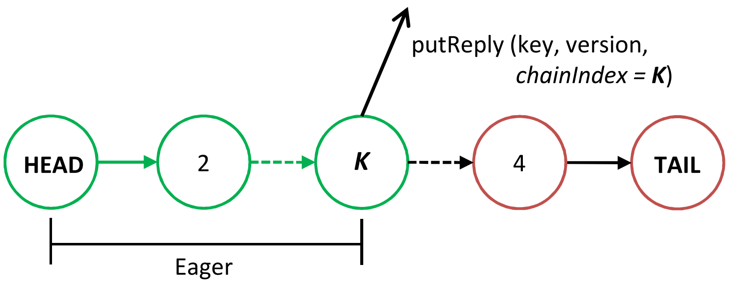


Figure 3.4: Example of the *eager phase* in the propagation of a PUT operation.

As soon as the dependencies have stabilized, the proxy uses consistent hashing to discover which data server is the head node of the corresponding chain, and forwards the request to that node. The head then processes the PUT operation, assigning a new version to the object, and forwarding the request down the chain, as in the original chain replication protocol, until the k element of the chain is reached (we call this phase of the propagation, the *eager phase*). An example of this propagation phase can be observed in Figure 3.4. At this point, a result is returned to the proxy, which includes the most recent version of the object and a *chainIndex* representing the k node. The proxy, in turn, forwards the reply to the client library. Finally, the library extracts the metadata and updates the corresponding entry in the table (replacing the version of the object by the new version, and *chainIndex* by the value k).

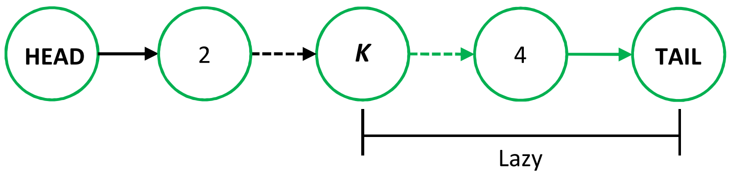


Figure 3.5: Example of the *lazy phase* in the propagation of a PUT operation.

In parallel with the processing of the reply, the update continues to be propagated in a lazy manner until it reaches the tail of the chain (shown in Figure 3.5). As we have noted, a data server may belong to multiple chains and, therefore, may be required to process and forward write requests for different data items. Updates being propagated in lazy mode have lower priority than operations that are being propagated in eager mode. Although the processing of each lazy/eager request is otherwise indistinguishable, the priority given to eager updates ensures that the latency of write operations of a given data item is not negatively affected by the additional replication degree of another item. When the PUT reaches the tail, the version written is said to be stable and an acknowledgment message is sent upwards in the chain to inform the remaining nodes. This message includes the key and version of the object so that a node can set a certain version of the object to the stable state. This acknowledgment message is propagated until it reaches the head of the chain.

3.3.2 Get Operation Processing

We now provide a detailed description on how GET operations are executed in ChainReaction. Upon receiving the GET request, the client library consults the metadata entry for the requested *key* and forwards the request along with the *version* and the *chainIndex* to the client proxy. The client proxy uses the *chainIndex* included in the metadata to decide to which data server the GET operation is forwarded to. If *chainIndex* is equal to R, the size of the chain (*i.e.*, the version is stable), the request can be sent to any node in the chain at random. Otherwise, it selects a target data server t at random with an index from 0 (the head of the chain) to *chainIndex* (shown in Figure 3.6). This strategy allows distributing the load of the read requests among the multiple available servers. The target data server t processes the request and returns to the proxy the value of the data item, and the version read. Then the client proxy returns the value and the metadata to the client library which, in turn, as with the PUT operation, extracts the metadata and updates its local metadata. Let *tindex* be the index in the chain of the node t that has processed the GET operation. Let also *pversion* be the previous version of the object as stored in the metadata, and *newversion*, the version read by the GET operation. The metadata is updated as follows: i) If the *newversion* is already stable (*i.e.*, known to have reached the tail of the chain), *chainIndex* is set to R (*i.e.*, the length of the chain); ii) If *newversion* is the same as *pversion*, *chainIndex* is set to *max*(*chainIndex*,*tindex*); iii) If *newversion* is greater than *pversion*, *chainIndex* is set to *tindex*.

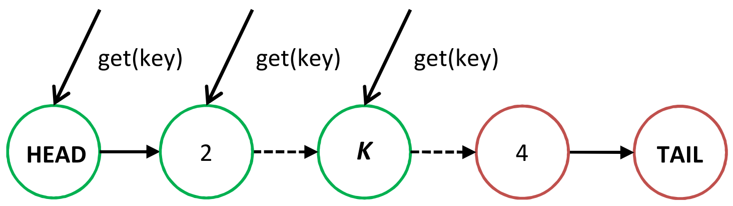


Figure 3.6: Valid targets for processing a GET operation with an *chainIndex* equal to k.

3.3.3 Fault-Tolerance

The mechanisms employed by ChainReaction to recover from a failure of a node in the head, middle, or the tail of the chain are the same as in chain replication. However, unlike the original chain replication, ChainReaction can continue to serve clients even if the tail fails. In our system a chain with R nodes can sustain $R - k$ node failures, as it cannot process any PUT operation with less than k nodes. If a node fails, two particular actions are taken: i) Minimal chain repair for resuming normal operations (with a reduced number of nodes). ii) Chain recovery by adding to the tail of the chain a node that already is in the system (*i.e.*, recover the original chain size); Moreover, a node can later join the system (and the DHT) for load balance and distribution purposes.

Considering that all chains are devised dynamically by the node responsible for a certain key (head) and its $R-1$ successors we can have the following 3 types of failures and corresponding minimal repairs:

Head Failure and Repair: When the head node fails (H), its successor (H^+,) takes over as the new head, as H^+ contains most of the previous state of H. All updates that were in H but were not propagated to H^+ are retransmitted by the client proxy when the failure is detected.

Tail Failure and Repair: The failure of a tail node (T), its easily recovered by replacing the tail with T predecessor, say T^-. Due to the properties of the chain, T^- is guaranteed to have newer or equal state to the failing tail T.

Failure and Repair of a middle node: When a middle node fails (x) between nodes A and B, the failure is recovered by connecting A to B without any state transfer, however node A may have to retransmit some pending puts that were sent to x but did not arrive to B. Since our solution only allows for version divergence to be one level deep, node A has to retransmit at most a single version per object. The failure of k node or its predecessor is treated in the same way and is completely transparent to the client, it will only notice a small delay in receiving the response to the write operation (due to the fault-detection period and recovery).

After performing the minimum repairs, the chain must be extended in order to maintain the initially configured replication factor (if there are nodes available). The size of the chain is repaired by adding a node in the tail (see node join details in the following paragraphs) as it is the easiest way to add a node in a chain.

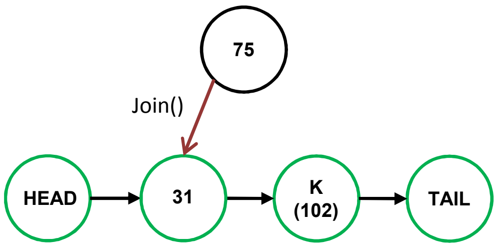
The failure and repair of nodes are one important part of the membership dynamics of the system, although they are not the only events that can change the membership. Nodes can also join and leave the system in a orderly manner. When a node leaves the system one can assume that it failed and the procedures used to process a node leaving can be the same as the ones described above for fault-tolerance. However, node joins must be handled differently as the new nodes must be aware of the chain's current update history.

When a node joins the system it is attributed an ID that will correspond to a certain position in the DHT. After figuring out its position in the DHT it can find the place of the chain when it will be added (Figure 3.7(a)). If a node x joins the system between nodes A and B then we must guarantee that x has a state that is newer or equal to B state. The state of node A must be transfered to node x (Figure 3.7(b)). During the state transfer x is in a zombie state and all updates are propagated from A to B and also from A to x, guaranteeing the normal execution of the system while the state is being transfered . The new updates received by x are saved locally for future execution. When the state transfer ends, node x is finally added to the chain and applies any pending updates sent by A (Figure 3.7(c)). When the join procedure finishes the chain resumes normal operation like shown in Figure 3.7(d).

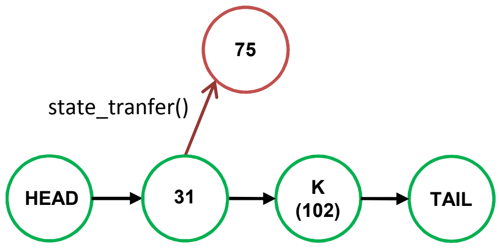
If the node joins at the tail or at the head of the chain the process is similar, however in the head the state is transfered from the previous head H to the new head $Hnew$. Moreover, when the state transfer ends $Hnew$ coordinates with H and with the proxies to be inserted in the chain and take over as the new head.

Since our algorithm only allows for the existence of one level of version divergence, the state transfer between nodes contains only one version per object for that chain. This results in a state transfer mechanism that allows add nodes in a very efficient way. Moreover, the failure of nodes is repaired by adding a node at the tail of the chain which can be done with this fast state transfer mechanism reducing the time for failure recovery.

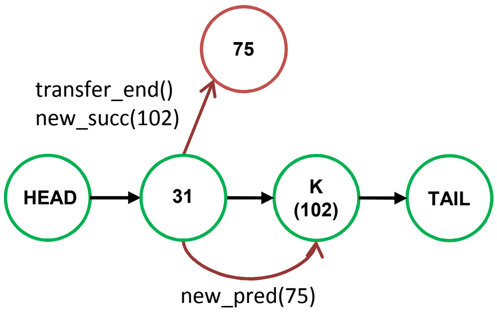
Additionally, the reconfiguration of a chain, after a node leaving/crash or when a node joins, may invalidate part of the metadata stored by the client library, namely the semantics of the *chainIndex*. However, since the last version read is also stored in the metadata, this scenario can be safely detected. If the node serving GET request does not have a version equal or newer than the last seen by the client, the request will be routed upwards in the chain until it finds a node that contains the required version.



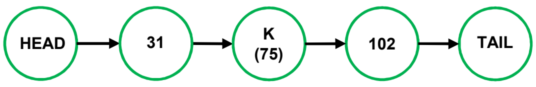
(a) Send join request to predecessor.



(b) State transfer from predecessor to the new node (new node is in zombie state).



(c) End of state transfer and chain reconfiguration procedure.



(d) Final chain state (the new node takes over as the new *k* node).

Figure 3.7: Example of a node join in the middle of the chain between nodes 31 and 102.

3.4 Supporting Geo-replication

We now describe how ChainReaction addresses a scenario where data is replicated in multiple datacenters. We support Geo-replication by introducing a minimal set of modifications with regard to the operation in a single site. Therefore, each datacenter continues to be organized as described in the previous section. However, metadata needs to be enriched to account for that fact that multiple replicas are maintained at different datacenters and that concurrent write operations may now be executed across multiple datacenter concurrently. We start by describing the changes to the metadata and then we describe the modifications to the operation of the algorithms.

First, the version of a data item is no longer identified by a single version number but by a *version vector* (similarly to what happens in classical systems such as Lazy Replication (Ladin, Liskov, Shrira, & Ghemawat 1992) or CODA (Braam 1998)). Similarly, instead of keeping a single *chainIndex*, a *chainIndexVector* is maintained, that keeps an estimate of how far the current version has been propagated across chains in each datacenter. Also, since the *causal+* consistency model enforces convergence of conflicting versions (*i.e.*, versions produced by concurrent writes to the same object in different datacenters), some deterministic rule must be used to order concurrent updates. For this purpose, each update is also timestamped with the physical clock value of the proxy that receives the request. Note that timestamps are only used as a tie-break if operations are concurrent. Operations that causally depend on each order are always ordered according to the order specified in their vector clocks. Therefore, physical clocks do not need to be synchronized although, for fairness, it is desirable that clocks are loosely synchronized (for instance, using NTP). Finally, if two concurrent operations happen to have the same timestamp, the identifier of the datacenter is used as the last tie-breaker.

With these changes in mind, we can now describe how the protocol for PUT and GET operations needs to be changed to address Geo-replication. For simplicity of exposition, we assume that datacenters are numbered from 0 to $D-1$, where D is the number of datacenters, and that each datacenter number corresponds to the position of its entry in the version vector and *chainIndexVector* (*i.e.*, the first entry of the vector is reserved for datacenter 0 and so on).

3.4.1 Put Operation Processing

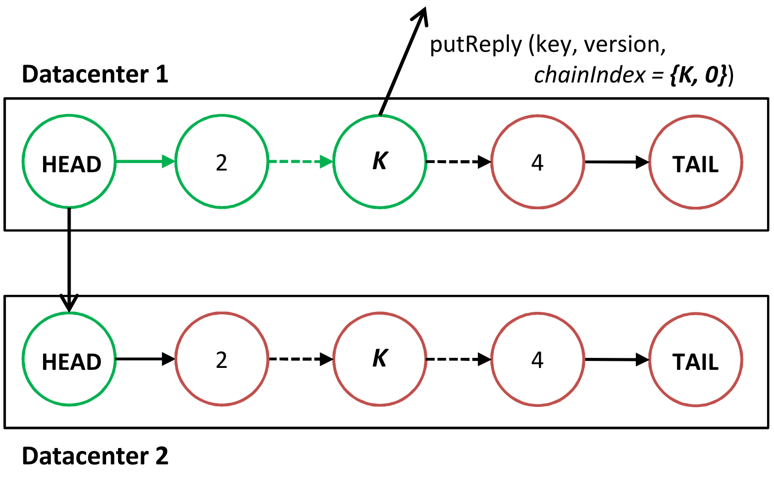


Figure 3.8: Example of the initial phase of propagation in a PUT operation with multiple datacenters. Notice that the PUT is propagated down the chain and to the other datacenter in parallel, however the propagation inside each datacenter is not synchronized. Also the *chainIndex* included in the response now includes two values, k for the local datacenter and 0 for the remote datacenter (*i.e.*, the state of propagation in the other datacenter is unknown)

The initial steps of the PUT operation are similar to the steps used in the single datacenter scenario. Let's assume that the operation takes place in datacenter number i. The operation is received by a client proxy, the dependency stabilization procedure executed, and then the request is forwarded to the head of the corresponding chain. The operation is processed and the object is assigned with a new version, by incrementing the ith entry of the version vector. The updated is pushed down in the chain until it reaches node k of the local datacenter (shown in Figure 3.8). At this point a reply is returned to the proxy, that initializes the corresponding *chainIndexVector* as follows: all entries of the vector are set to 0 (*i.e.*, the conservative assumption that only the heads of the sibling chains in remote datacenters will become aware of the update) except for the ith entry that is set to k. This metadata is then returned to the client library. In parallel, the update continues to be propagated lazily down in the chain. When the update finally reaches the tail, an acknowledgment is sent upward (to stabilize the update) *and* as well as to the tails of the sibling chains in remote datacenters as shown in Figure 3.9 (since all siblings execute this procedure, the global stability of the update is eventually detected in all datacenters).

Also, as soon as the update is processed by the head of the local chain, the update is scheduled to be transferred in background to the remote datacenters as shown in Figure 3.8. When a remote update arrives at a data center, it is sent to the head of the corresponding chain. If the update is more recent than the update locally known, it is propagated down the chain. Otherwise, it is silently discarded as it has already been superseded by a more recent update.

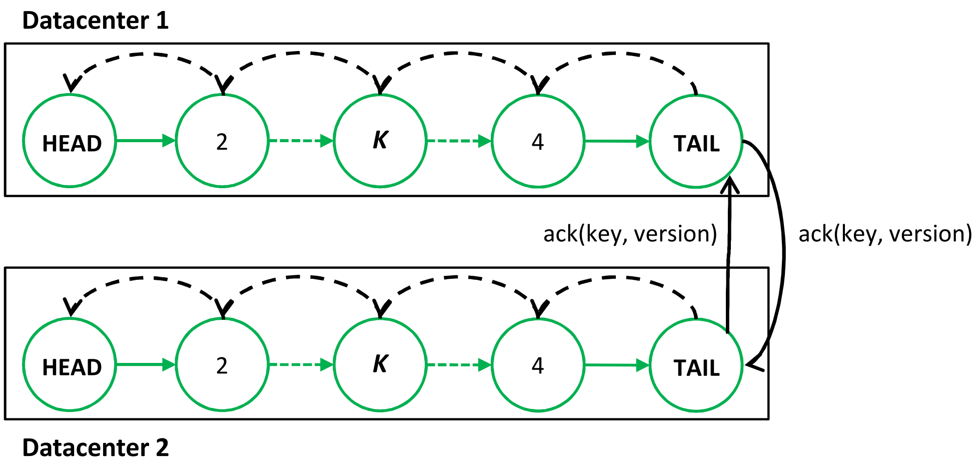


Figure 3.9: Propagation of acknowledgment messages between datacenters and inside each local datacenter. When a tail receives an acknowledgment from each datacenter the version is guaranteed to be stable in all datacenters.

It is worth noting that, using the scheme above, each datacenter may configure a different value of k for the local chain, as the configuration of this parameter may depend on the characteristics of the hardware and software being used in each datacenter.

3.4.2 Get Operation Processing

The processing of a GET operation in a Geo-replicated scenario is mostly identical to the processing in a single datacenter scenario. The only difference is that, when datacenter i receives a query, the set of potential targets to serve the query is defined using the ith position of the *chainIndexVector*. Finally, it may happen that the head of the local chain does not have yet the required version (because updates are propagated among different datacenters asynchronously). In this case, the GET operation can be redirected to another datacenter or block until a fresh enough update is applied locally. Additionally, an hybrid solution can be employed, the local datacenter would receive the request and send a request to a remote datacenter (or several in parallel). Meanwhile if a fresh update reaches the local datacenter, the request can be processed locally and a response sent to the client, otherwise one must wait for the reponse of the remote datacenter.

3.4.3 Conflict Resolution

Since the metadata carries dependency information, operations that are causally related with each other are always processed in the right order. In particular, a read that depends (even if transitively) from a given write, will be blocked until it can observe that, or a subsequent write, even if it is submitted to a different datacenter.

On the other hand, concurrent updates can be processed in parallel in different datacenters. However, similarly to many other systems that ensure convergence of conflicting object versions, ChainReaction's conflict resolution method is based on the last writer wins rule (Thomas 1979) (however, any other conflict resolution mechanism could be used). This conflict resolution mechanism is needed to ensure the causal+ consistency of concurrent write operations. Therefore, if PUT operation arrives "too late" to a datacenter, it is silently discarded. On the other hand, if the update is the most recent operation, it overwrites other concurrent updates that have been serialized in the past.

3.4.4 Wide-Area Fault-Tolerance

In ChainReaction, we have opted to return from a PUT operation as soon as it has been propagated to k nodes in a single datacenter. Propagation of the values to other datacenters is processed asynchronously. Therefore, in the rare case a datacenter becomes unavailable before the updates are propagated, causally dependent requests may be blocked until the datacenter recovers. If the datacenter is unable to recover, those updates may then been lost.

There is nothing fundamental in our approach that prevents the enforcement of stronger guarantees. For instance, the reply could be postponed until an acknowledgment is received from d datacenters, instead of waiting just for the acknowledgment of the local k^{th} replica (the algorithm would need to be slightly modified, to trigger the propagation of an acknowledgment when the update reaches the k^{th} node in the chain, both for local and remote updates). This would ensure survivability of the update in case of disaster. Note that the client can always re-submit the request to another datacenter if no reply is received after some pre-configured period of time.

Although such extensions would be trivial to implement, they would impose an excessive latency on PUT operation, so we have not implement them. In a production environment, it could make sense to have this as an optional feature, for critical PUT operations.

3.5 Providing Get-transactions

The work presented in (Lloyd, Freedman, Kaminsky, & Andersen 2011) introduced a transactional construct which enables a client to read multiple objects in a single operation in a *causal+* consistent manner, however these transactions do not offer ACID guarantees. This construct is named GET-TRANSACTION, an operation which can significantly simplify the work of application developers, as it offers a stronger form of consistency on read operations over multiple objects.

Consider a scenario where client c_1 updates two objects. More precisely, c_1 updates twice objects X and Y, exactly in this order, creating the following sequence of causally related updates $x_1 \rightarrow y_1 \rightarrow x_2 \rightarrow y_2$. Lets now assume that another client c_2 concurrently reads objects X and Y, also in this sequence. If c_2 reads values x_1 and y_2 this does not violate causality. However, each of these values belongs to a different snapshot of the database, and this may violate the purposes of client c1. To illustrate the problem, we borrow the example provided in (Lloyd, Freedman, Kaminsky, & Andersen 2011): consider a social network that allows users to share a photo album (object Y). The album is protected by an access control list (ACL, object X). The initial value of the ACL is x_1 and the initial value of the album is y_1. Alice is a user c_1 of this application and she removes Bob (client c_2) from the list of users that can see her photos (update x_2). Then Alice adds a new photo that Bob should not be able to see (update y_2). In this case, if two read operations were used to fetch both the ACL and the album, it could happen that Bob reads an older version of the ACL (x_1) and the new version of the album (y_2). This way, Bob would see the new photo and would violate the semantics of the ACL. The usage of GET-TRANSACTIONS aims at avoiding this scenario as the semantic of this operation enforces the most recent version of the ACL to be returned if the newer version of the album was returned. Otherwise, if the returned version of the album was an old one, then the version of the ACL returned can be either the newer or the older.

To support GET-TRANSACTION operations the system must ensure that the values returned where not written by a PUT operation that was issued after the GET-TRANSACTION. For this purpose, our implementation supports an *atomic multi-enqueue operation*. Conceptually, each head of a chain has a queue, where all PUT operations are enqueued. To support snapshot consistent reads, we also enqueue reads that are part of the same GET-TRANSACTION in the processing queue of the heads of the corresponding chains. Furthermore, the atomic multi-enqueue operation allows to enqueue all the reads from the same GET-TRANSACTION, in the different queues of the corresponding chain heads in an atomic manner. An interesting feature of this scheme is that, in opposition to (Lloyd, Freedman, Kaminsky, & Andersen 2011), we do not require the existence of 2 rounds to process a GET-TRANSACTION.

With this in mind, a GET-TRANSACTION is processed as follows. When the operation is received by a proxy, it invokes the atomic multi-enqueue operation to enqueue all the reads in a single atomic step. Then, when the individual reads reach the head of the corresponding chains the value is returned to the proxy. The proxy waits for all values, along with the corresponding metadata, assembles a reply, and sends it back to the client library. Similar to what happens with PUT and GET operations, upon receiving the reply from the client proxy, the client library extracts the metadata enclosed in the reply, and updates its local metadata. Notice that the metadata enclosed in the reply is equivalent to the metadata of multiple GET operations.

GET-TRANSACTIONS in a Geo-replicated scenario have the following additional complexity. Assume that a GET-TRANSACTION is being processed by datacenter i but it includes dependencies from values read in a different datacenter. Such updates may have not yet been propagated to datacenter i when the GET-TRANSACTION is processed. In this case, the read is aborted and retried in a (slower) two-phase procedure. First, the proxy reads all dependencies that have failed from the corresponding heads, using a specialized blocking read operation (the same as used in the *dependency stabilization procedure*. Then, the GET-TRANSACTION is reissued as described above (and it is guaranteed to succeed).

Summary

This chapter described in detail the mechanisms behind the working of ChainReaction. In the first place, we discussed the reasons behind implementing the *causal+* consistency model which was followed by the description of the Client API that allows client applications to access ChainReaction. Next we described the architecture of ChainReaction focusing on each of its components (data nodes, client proxies and client library) and introduced the modifications made to the Chain Replication protocol. Then the solution for a single and multiple datacenters was introduced. We focused in describing the inner workings of each type of operation (PUT and GET) and also on fault-tolerance mechanisms. Finally, we described our extension for implementing GET-TRANSACTIONS allowing for a client to obtain a consistent view for a set of keys.

The next chapter will describe the implementation details of our solution, we focus on the inner workings of the original FAWN-KV, on the optimizations made to FAWN-KV and on the implementation of the algorithms described in this chapter.

4 Implementation Details

This chapter describes the implementation of the ChainReaction prototype. Our prototype was built on top of the FAWN-KV system so we start by describing some details of its implementation. We used FAWN-KV as a starting point because it implements Chain Replication so it saves some of the work of implementing the protocol from scratch. On the other hand, the code of FAWN-KV is simple and easy to understand. The description includes presenting the three main components of the FAWN-KV system: Back-end, Front-end, and Manager. Next we describe the optimizations and improvements made to the FAWN-KV system before starting our implementation, the resulting optimized version was used in the experimental evaluation. Finally, we describe some implementation details of ChainReaction.

4.1 Implementation of FAWN-KV

According to the FAWN architecture (Andersen, Franklin, Kaminsky, Phanishayee, Tan, & Vasudevan 2011), the FAWN-KV is composed by two main components: front-end (client proxy) and back-end (data server). In the original implementation, FAWN-KV only supports one front-end and multiple back-ends. The clients connect to the front-end through a client library that exports the FAWN-KV API. All the communication between client and server and also between server applications is made using the Apache Thrift 0.5.0 framework, which exports an RPC interface that allows to build scalable cross-language services supporting both synchronous and asynchronous communication. To coordinate and maintain the DHT structure of the back-ends, FAWK-KV also employs a manager application (which can be deployed in an independent node) that is responsible for adding and removing nodes from the DHT ring. This manager also informs the front-ends of changes to the the ring structure, namely when a node joins or leaves. In the following subsections we will describe the three main components of FAWN-KV: Back-end, Front-end and Manager.

4.1.1 Back-end Implementation

The Back-end application is the central component of the FAWN-KV key-value storage, implementing the main semantics of the datastore. It is composed of two distinct modules that are described below:

- FAWN-KV Backend Handler: This component is responsible for implementing the operation semantics (*i.e.*, PUT and GET operations) ensuring their correction;
- FAWN Data Store: Module responsible for storing the client data.

The first component (backend handler) offers a distributed service that is used by front-end (service client) to redirect client requests. Each backend component can be responsible for various key-ranges due to the existence of replicas. Therefore, each component maintains a different structure and a different FAWN Data Store, associated to that structure, for each key range. Such structure can be accessed concurrently by different operations: to implement concurrency control, each structure is protected by a write-read lock that is obtained before executing any operation. This lock allows for multiple GET operations to occur concurrently in the same key range, however PUT operations are serialized with respect to other PUT and GET operations. Operations that are related to different key-ranges can be processed concurrently without any concurrency control mechanisms. Now we describe how each operation is processed by the backend handler:

Put Operation: All PUT operations are received from the front-end and each request spawns a new execution thread that processes it. First, the execution thread finds the key-range structure responsible for the key and then tries to obtain a write lock on that structure. Next, the backend handler forwards the request to the FAWN Data Store associated with the key-range structure so that the value gets written. After returning from the FAWN Data Store module, if the node is the last node in the chain (*i.e.*, the tail) then a response is sent to the front-end,the object is acknowledged locally, and an acknowledge message is sent upwards in the chain. Otherwise, the backend handler propagates the PUT operation to its successor, which in turn repeats this process.

Get Operation: As opposed to PUT operations, GET do not spawn multiple threads to be executed. When a GET request is received it is placed in a queue to be processed by a number of consumer threads (which can be configured). A consumer thread removes a request from the queue, finds the key-range structure associated to the key, and waits to obtain a read lock for that structure. After obtaining the lock, a read request is sent to the FAWN Data Store which returns the value of the object. Finally, the value is included in a reply message that is sent to front-end and then forwarded to the client.

The FAWN Data Store (FAWN-DS) is a log-structured datastore that contains the values associated with a certain key-range (*i.e.*, each instance of the FAWN Data Store is responsible only for a single range of keys). FAWN-DS is optimized for flash storage and to operate with low RAM resources. This way, all writes to the datastore are sequential, and reads only require a single random access. For this purpose it maintains an in-memory hash table (Hash Index) that map keys to an offset in the append-only Data

Log on flash. This datastore offers persistent storage to clients by supporting the following operations: Store, Lookup and Delete.

Store: This operation appends a new entry to the log and updates the associated entry on the hash table to point to this offset. If the key already exists then the old value is set to *orphaned* for later garbage collection;

Lookup: Allows to retrieve the hash entry (containing the offset) for a certain key, and returns the corresponding data;

Delete: Invalidates the hash entry associated with the key by writing a delete entry to be garbage collected.

As nodes can join and leave the DHT ring, it is expected that key-ranges can be split in two (when a node joins) and merged in one (when a node leaves). Also the datastore must be periodically compacted to remove orphaned or deleted entries created by other operations. For this purpose, the datastore also supports *Split*, *Merge*, and *Compact* operations.

4.1.2 Front-end Implementation

As described before, the Front-end (or client proxy) is responsible for forwarding client requests to the correct nodes. This application deploys a distributed service using the Apache Thrift framework, therefore FAWN-KV presents a 3-tiered architecture where the FAWN Client acts as the Front-end client and the Front-end acts as the Back-end client.

To forward the client requests the Front-end must know about the existence of the Back-end nodes and their structure. That information is provided by the Manager that informs the Front-end whenever a change in the state of the ring happens (nodes leave or join). Using this information the Front-end can forward PUT and GET requests in the following way:

Put Request: The Front-end uses the key [1] sent by the client to find the *owner* (head of the chain) of the node. If the node is up, the request is forwarded to it. Otherwise, if the node has failed, the front-end will check the successors until it finds an active node.

Get Request: In GET operations the Front-end also uses the key to find the owner, but contrary to the latter it checks if the R-1 successor (tail) of the owner is valid. If it is valid, then the request is redirected to it, otherwise the Front-end will check the R-2 successor and so on until it reaches the owner.

[1] The original implementation of FAWN-KV assumes that the keys sent by the client are already hashed.

The Front-end is also responsible for informing the Back-ends about the replication factor being used and how many replicas are available. So, in each PUT operation the Front-end sends to the back-end a value indicating how many *hops* the put request has to make (*i.e.*, if all replicas are available the number of hops is equal to R, otherwise the number of hops is R-number of failed nodes).

Finally, as all communication between FAWN-KV components is made asynchronously, the Front-end must make sure that the responses are sent to the right client. To this end it assigns a number to the requests (sent to the Back-ends) and maintains them in a structure that associates a request number with the client addresses. Upon receiving a PUT or GET response the Front-end searches for the entry associated with the request. Then it retrieves the correct address of the client and sends it the reply.

4.1.3 Manager Implementation

The Manager is an auxiliary component responsible for managing the DHT ring and also to detect and repair Back-end nodes failures. This component also exports a Thrift service that is used by both Front-ends and Back-ends. It maintains a ring structure that contains all available Back-end nodes.

A Back-end node joins the ring by sending a join message to the manager. The join operation can be a static join (at system initialization) or a dynamic join (during system operation). Static joins do not trigger any Split or Merge operations since no operations have been executed. However, if a node joins dynamically it can happen that some nodes must Split their FAWN Data Store to share the key space with the newly joined node [2].

To detect the failure of a Back-end node the Manager uses an *heartbeat* mechanism. Back-ends periodically send an heartbeat message to the manager: if the manager fails to receive two heartbeats in a row it assumes that the Back-end has failed and triggers a chain repair procedure. The repair procedure is based on the Chain Replication technique which has 3 different mechanisms to repair the chain depending on the node that has failed (head, tail or mid node).

4.2 Optimizations and Changes

Before describing the implementation of our solution we will present some optimizations and improvements made to FAWN-KV as it was used as the base of our system implementation. By inspecting the code of FAWN-KV we found that there were many places where the implementation could be improved. This way, we made a number of changes that we did find crucial for implementing our solution, which are described in the following paragraphs.

[2]In our implementation dynamic join was never used since it is not supported in the original code

First we decided to optimize the Front-end component of the FAWN-KV. In early experimental runs of FAWN-KV we found that the Front-end was the bottleneck of the whole system, which is not desirable. To circumvent this problem, FAWN-KV was extended to support multiple Front-ends moving the bottleneck to the Back-end nodes (data nodes). This way, when the Front-end is overloaded one only needs to add more Front-end nodes. This change was easy to achieve as the Front-ends have no distributed state.

We also found that the data structure that maintains client information (described in the previous section) was not optimized. Every client request was generating a new entry with repeated information (for each client) leading to information duplication (furthermore, this information was not garbage collected) which led to an memory consumption problem. We changed this implementation to re-utilize entries that belong to the same client and also we add a periodic garbage collection to remove unused entries. We also removed some logic that was not needed for our system and improved some locking mechanisms. Locks were being used in a coarse-grained fashion and we have implemented a fine-grained version improving the Front-end performance.

A problem with FAWN-KV is that it assumes that keys are sent to the Front-end already hashed; The only thing that the Front-end had to do is to transform the key received in a 160 byte hexadecimal (if the key was smaller it would pad it with zeros). This was a problem in our setting because the benchmark tool did not send the keys hashed, which led to a bad distribution of the keys in the ring. We changed the Front-end to hash keys upon receiving any request, by using an implementation of the SHA-1 algorithm. Moreover, we changed the internal key structure to support any size of the key (in this case a size of 20 bytes was used).

Finally, we decided to make some changes to the Back-end node, since it is the core of the key-value store. We optimized the processing of PUT and GET operations by removing some logic that we would not need. Next we changed the FAWN-KV system by making it support multi-versioned data. FAWN-KV only supports a single version for each object but that architecture is not compatible with our solution. We changed the Back-end to support multi-version by adding a map inside each key-range structure (described previously), which we call *version map*. This map is responsible for maintaining the information about each version of each object (*i.e.*, it maintains an entry for each object which in turn contains an entry for each version). Anytime an operation is processed it must access the version map in the following way:

Put Operation: When a PUT is received, the Back-end must check if there is an entry in the map for the requested key. If there is one, then a new version entry is added and the new version number is obtained by incrementing the previous version number (or vector entry). Otherwise, an entry is created for the object and the version with number 0 is added.

Get Operation: On a GET operation, the Back-end checks if there is an entry for the requested object. If there is then it searches for the latest version for that object and returns it. Otherwise, the object does not exist and an error is returned.

Since the FANW-DS module also does not support multi-version we decide not to change it. Therefore, we created a new key object that represents the key concatenated with the version number. When writing this key on the FAWN-DS each version appears as a different object and it will be preserved. This avoids changing the FAWN-DS module which could be complex and troublesome.

4.3 Chain Reaction Implementation

We have implemented ChainReaction using the optimized and improved version of FAWN-KV, described in the previous section, as a base. The final prototype has approximately 26.000 lines of C, C++, and Java code comparing with the 18.000 lines of code in the original FAWN-KV system. Our implementation follows the descriptions provided in Chapter 3. Note however, that we did not implement the mechanisms for wide area fault tolerance and recovery, as this was not the main focus of our contribution. In our implementation we kept the FAWN architecture composed by the three main components: Back-end (data node), Front-end (client proxy), and Manager. In the following subsections we will describe in detail the implementation of the three main components and also some details about the management of the metadata.

4.3.1 Back-end Implementation

The Back-end in ChainReaction is also the central component of the system implementing all operation semantics and is composed by the FAWN-DS and the Backend Handler. The FAWN-DS module used in our solution is the same as the one used in the FAWN-KV system, however we changed the limit of data storage from 4GB to 16GB (the FAWN-DS was optimized for working on 4GB flash memory modules). On the other hand, the Backend Handler was completely modified to cope with our system specifications.

The main structure of the Backend Handler is similar to the one included in the FAWN-KV system as it maintains the key-range structures to support multiple key-ranges which are also protected by locking mechanisms. However, the Thrift service interface of the FAWN Backend Handler was extended to support GET-TRANSACTION operations and also to include an operation that we named *stable check*. The stable check operation has the purpose of checking if a certain version of an object as already reached the tail, which is needed to implement our solution described in Chapter 3. In the following paragraphs we will describe how PUT, GET and GET-TRANSACTION operations are implemented in our system:

Put Operation: Contrary to the original FAWN-KV system, each PUT request does not trigger a new thread, instead it is placed in a queue by sequence number order (check Section 4.3.3 and 4.3.2 for details on sequence number attribution). This order in queue insertion is needed to ensure the correctness of GET-TRANSACTIONS, as it was described in the previous chapter. The PUT requests are then processed by a consumer thread that removes the requests in order from the queue. In our system a PUT request can have two sources, the local client or a remote datacenter. Each require a different processing: If the put is local then the request is processed in a way similar to the one described for the FAWN-KV system, with the exception that the head node is responsible for sending the PUT request (including a timestamp and dependencies) to the local manager to be propagated to the remote datacenters; Otherwise, in a remote PUT request, the consumer thread first checks if the remote version conflicts with the local version, if it conflicts then it needs to run the conflict resolution mechanism based on the received and local timestamps. If there are no conflicts then if the version is strictly higher then the local version, the local version is overwritten, otherwise the version is lower or equal and it is ignored. The rest of the request is processed as described before with exception to the reply sent to the client. In this case the reply to the client is sent after reaching the K node in the chain as described in Chapter 3. For both local and remote PUT operations, an acknowledgment message is sent to the remote datacenters when the request reaches the tail. A version is only said to be stable when all acknowledgment messages have been received from the other datacenters. At this moment the local acknowledgment message is sent upwards in the chain, stabilizing the version.

Get Operation: As the main focus of the work is to optimize the performance of GET operations we do not added much logic to this operation. The GET requests are processed in a similar way to the ones in FAWN-KV system. The only change is that the consumer thread has to check if the local version is higher or equal to the version included in the request. If it is not, then the request is forwarded to the predecessor node which repeats this process until an equal or higher version is available.

Get-transaction Operation: The GET-TRANSACTION operation was implemented from scratch as its semantics are completely new to the system. According to the algorithm described in the previous chapter, a GET-TRANSACTION request also contains a sequence number and can only be processed after all requests with lower sequence numbers have been processed. This way, there is also a queue where GET-TRANSACTION requests are placed before being processed. These requests are processed by a specific consumer thread that checks if all sequence numbers prior to the one in front of the queue have already been processed. If it is the case then the GET-TRANSACTION request can be removed from the queue and processed as follows: As all keys in a GET-TRANSACTION request belong to a single key-range (*i.e.*, the node processing the request is the head of that objects chain), the consumer thread searches for the structure related to that key-range and waits to acquire a single read-lock; Then it reads a version of each key on the corresponding head; On each read a response is sent to the Front-end including the key, version, value, and metadata related with that key; Finally, after all values have been read the read-lock is released and the operation ends.

To ensure the correctness of GET-TRANSACTION operations the *version map* entries were extended to include the sequence number of the PUT operation that wrote that version. This way, the map can be used to check if there is a version prior to the GET-TRANSACTION sequence number.

4.3.2 Front-end Implementation

The main structure of our system Front-end (client proxy) is similar to the one described for FAWN-KV system. We included all the optimizations made to the FAWN-KV and we also changed the Thrift service interface to include GET-TRANSACTIONS and also all the service calls related with the propagation of PUT operations to remote datacenters (PUT propagation and acknowledgment propagation). The main changes in the Front-end are related with the forwarding of PUT, GET and GET-TRANSACTION requests. These requests are processed and forward by the Front-end as follows:

Put Operation: When the Front-end receives a PUT request it first checks the object versions included in the metadata field. Then it issues a *stable check* request for each object version (in parallel) to the corresponding tail node. When the Front-end collects all replies from all tails then it is guaranteed that all dependencies are stable and the operation can proceed. Next, the Front-end issues a request to a sequencer process (in this case the sequencer was implemented in the manager as it will be described further ahead) which replies with a sequence number to be added in the request. This sequence number is used to implement the *atomic multi-enqueue operation* described in the previous chapter. The Front-end then finds the head node for the requested key and forwards the request including the value of K (so that Back-ends know when to reply to the client).

Get Operation: In the original FAWN-KV implementation all get operations are forward to the tail to the chain as it implements the original Chain Replication protocol. In our solution the routing of GET varies according to the *chainIndex* included by the client in the request. If the *chainIndex* is empty it means that the last seen version was stable so the request can be forwarded to any node between the head and the tail. Otherwise, the request can only be forwarded to a node between the head and the node identified by *chainIndex*. The choice of the node is made by using the random function with an max of the tail or *chainIndex* respectively. The request is then sent to the chosen node including the corresponding *chainIndex*, which is returned by the Back-end without being modified and included in the client response if the returned version is not stable (*i.e.*, this prevents the Back-end from using resources in finding its *chainIndex* when the Front-end already found it).

Get-transaction Operation: A GET-TRANSACTION can be seen as a set of GET operations, however GET-TRANSACTIONS can only be directed to the heads of the chains. But before sending the request to the corresponding heads, the Front-end groups the keys that belong to the same key-range (*i.e.*, they belong to the same chain). For each group of keys the Front-end will request a sequence number to the sequencer process, as each group belongs to a different chain. Next a request containing each key group is sent in parallel to the corresponding chain heads. Finally, the Front-end waits for all responses from the heads, collects the values and sends an response to the client.

4.3.3 Manager Implementation

In our system the Manager was assigned with three different roles: it is the coordinator of the DHT ring, acts as sequencer for implementing the atomic multi-enqueue operation, and acts as a proxy for the remote datacenters. As the DHT ring coordinator its functionality is similar to the one described for the FAWN-KV system as we do not improved any fault recovery or fault detection mechanisms. However the other two functionalities are completely new and are described in the following paragraphs.

The implementation of the atomic multi-enqueue operation is a combination of the functionality of the Front-end, Back-end and Manager. The first two components have already been described and we now focus on the Manager functionality. To avoid the intensive use of low level concurrency control mechanisms, all PUT operations and GET-TRANSACTIONS are first routed through a specialized sequencer process, similar to the one used in (Malkhi, Balakrishnan, Davis, Prabhakaran, & Wobber 2012), that is implemented by the manager. The sequencer process keeps a counter for each chain, which represents a position in the queue that is maintained by the head of the chain. All PUT operations and reads that are part of a certain GET-TRANSACTION are attributed a sequence number (associated with each chain) and are enqueued in each chain head queue taking into consideration their sequence number order. This sequencer could be implemented in a separate component but we have chosen to implement it in the manager from a practical point of view.

This sequencer could bring some problems if the system was used in a large-scale setting (hundreds of servers), however the implementation of a efficient distributed sequencer is not the focus of this work as we postpone this issue to the future work. Moreover, our experimental evaluation has shown that the overhead incurred by the use of the sequencer process in the Manager is negligible in the performance of the entire system.

As our solution supports multiple datacenters each one has a corresponding Manager that is responsible for coordinating the local datacenter and also to send and receive requests to/from remote datacenters. To this end each Manager exports a Thrift service with an interface that allows to propagate PUT operations and to propagate acknowledgment messages. The messages are directly sent by the Back-end nodes to the local Manager that is responsible for propagating the messages to all remote managers. On the other hand, all received messages are forwarded to a Front-end node (client proxy), chosen at random, which in turn redirects the message to the correct Back-end node.

Finally, the Manager also implements a failure detector, based on heartbeats, that allows to detect the failure of a remote datacenter, although we have not implemented any recovery mechanisms related with this kind of failure.

4.4 Metadata Management

In our system all existing metadata is stored in the client as the correctness of operations depends on the sequence operations issued by the clients. In this way, we implemented a new Client Library that stores metadata in a map that contains an entry for each object. Each entry contains the key, version, and *chainIndex* for a certain object version. This map is updated upon receiving the reply for each PUT, GET or GET-TRANSACTION operation.

To efficiently encode and transmit dependency information between datacenters, we resort to an implementation of Adaptable Bloom Filters (Couceiro, Romano, Carvalho, & Rodrigues 2009). To this end, whenever a client issues a *get* or *put* operation, ChainReaction returns to that client, as part of the metadata, a bloom filter which encodes the identifier of the accessed object version. This bloom filter is stored by the Client Library in a list named *AccessedObjects*. When the client issues a *put* operation, it tags its request with a bloom filter, named *dependency filter*, which is locally computed by the client library by performing a *binary* OR over all bloom filters locally stored in the *AccessedObjects* set. Upon receiving the reply, the Client Library removes all bloom filters from the local *AccessedObjsects* set, and stores the bloom filter encoded in the returned metadata.

The *dependency filter* tagged by the Client Library on the *put* request, and the bloom filter that is returned to the issuer of the PUT (we will refer to this bloom filter as *reply filter* in the following text), are used by the datacenter that receives the PUT operation as follows:

When a PUT request is propagated across datacenters it is tagged with both the *dependency filter* and the *reply filter* that are associated with the local corresponding PUT request. On the remote datacenter the client proxy receives the PUT request and places it in a waiting queue for being processed in the near future.

The two bloom filters associated with PUT requests encode causal dependencies among these requests. If a *wide-area-put* request op_1 has a *dependency filter* that contains all bits of a *reply filter* associated with another *wide-area-put* request op_2, we say that op_2 is potentially causally dependent of op_1. We say *potentially* because bloom filters can provide false positives, as the relevant bits of the *dependency filter* of op_1 can be set to one due to the inclusion of other identifiers in the bloom filter. The use of adaptable bloom filters allows us to trade the expected false positive rate with the size of the bloom filters. In our experiments, which we present further ahead in the text, we have configured the false positive rate of bloom filters to 10%, which resulted in bloom filters with 163 bits.

Summary

This chapter introduced the inner details of the implementation of both FAWN-KV system and also ChainReaction prototype. We started by describing the three main components of the FAWN-KV system: back-end (data server), front-end (client proxy) and manager. Next we introduced the main drawbacks of FAWN-KV when used in our scenario and how we changed and improved the system to overcome those flaws. Finally, we introduced the details and challenges behind implementing the algorithms described in Chapter 3.

The resulting prototype implements all functionality that was devised in previous chapters and in the next chapter we present the experimental evaluation made using this prototype.

5 Evaluation

In this chapter we present experimental results obtained through the execution of our prototype of in our own cluster. We extracted comparative performance measures for three other systems: FAWN-KV, Cassandra and COPS (emulation). Moreover, we conducted experiments in five distinct scenarios, as follows: i) we have first assessed the throughput and latency of operations on ChainReaction in a single datacenter, and compare its results with those of FAWN-KV and Cassandra; ii) then we have assessed the performance of the system in a Geo-replicated scenario (using 2 virtual datacenters), again comparing the performance with FAWN-KV and Cassandra; iii) next we measured the performance of ChainReaction using a custom workload able to exercise GET-TRANSACTIONS; iv) after we tested the fault-tolerance of our system in a single datacenter scenario by measuring its performance during an occurrence of a node failure; v) finally, we measured the overhead in terms of size that the metadata imposes in our system, and compared the results to the size of metadata stored by COPS. All results for the throughput presented were obtained from five independent runs of each test. The results for the latency tests reflects the values provided by YCSB in a single random run. Finally, the results from the metadata overhead experiments were obtained from ten different clients. Confidence intervals are plotted in all figures.

5.1 Single Datacenter Experiments

We first compare the performance of ChainReaction against FAWN-KV (Andersen, Franklin, Kaminsky, Phanishayee, Tan, & Vasudevan 2011) and Apache Cassandra 0.8.10 in a single datacenter scenario. It should be noted that ChainReaction was built on top of an optimized version of FAWN-KV, therefore, for sake of fairness, in the comparisons we have used the version of FAWN-KV with the same optimizations. Our experimental setup uses 9 data nodes plus one additional independent node to generate the workload. Each node runs Ubuntu 10.04.3 LTS and has 2x4 core Intel Xeon E5506 CPUs, 16GB RAM, and 1TB Hard Drive. All nodes are connected by a 1Gbit Ethernet network. In our tests we used 5 different system configurations, as described below:

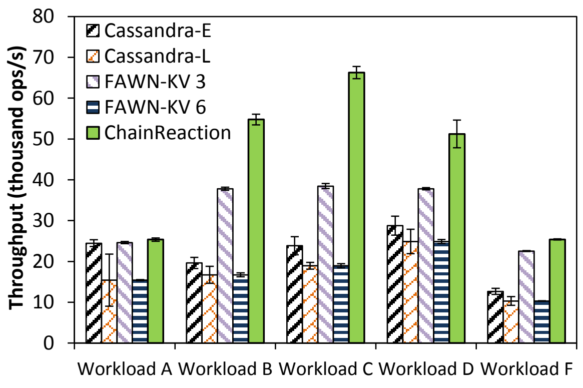
Cassandra-E: Deployment of Apache Cassandra configured to provide eventual consistency with a replication factor of 6 nodes. Write operations are applied on 3 nodes before returning while read operations need only to be processed at one node.

Cassandra-L: Deployment of Cassandra that provides eventual consistency, however operations are applied to a majority of nodes (4 nodes) before returning. A replication factor of 6 is also used.

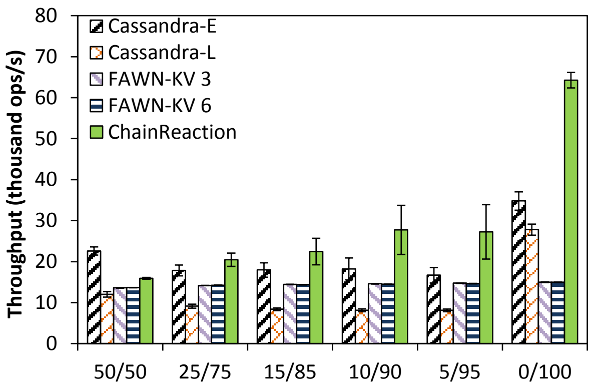
FAWN-KV 3: Deployment of the optimized version of FAWN-KV configured with a replication factor of 3 nodes which provides linearizability (chain replication).

FAWN-KV 6: Deployment of the optimized version of FAWN-KV configured with a replication factor of 6. Also provides linearizability.

ChainReaction: Single Site deployment of ChainReaction, configured with $R = 6$ and $k = 3$. Provides *causal+* consistency.



(a) Standard YCSB Workloads (multiple objects).

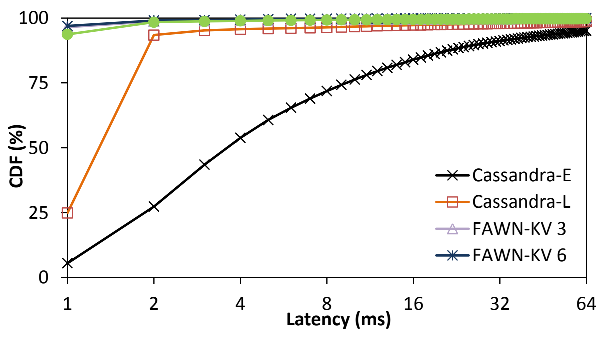


(b) Custom Workloads (single object).

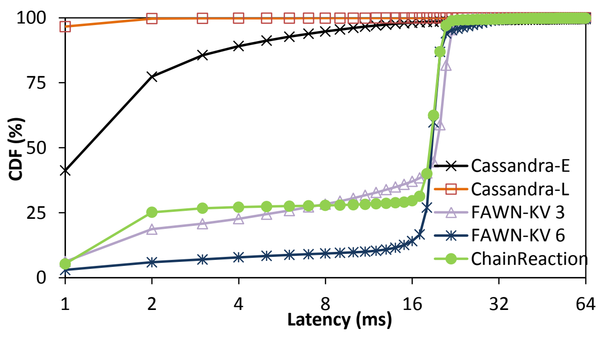
Figure 5.1: Throughput in thousand operations per second for each type of workloads considering a single datacenter deployment.

All configurations have been subject to the Yahoo! Cloud Serving Benchmark (YCSB) version 0.1.3 (Cooper, Silberstein, Tam, Ramakrishnan, & Sears 2010). We choose to run standard YCSB workloads with a total of 1,000,000 objects. In all our experiments each object had a size of 1 Kbyte. We have also created a micro benchmark by using custom workloads with a single object varying the write/read ratio from 50/50 to 0/100. The latter allows assessing the behavior of our solution when a single chain is active. All the workloads were executed by a single node simulating 200 clients that, together, submit a total of 2,000,000 operations. The next paragraphs will describe in detail each one of the used workloads.

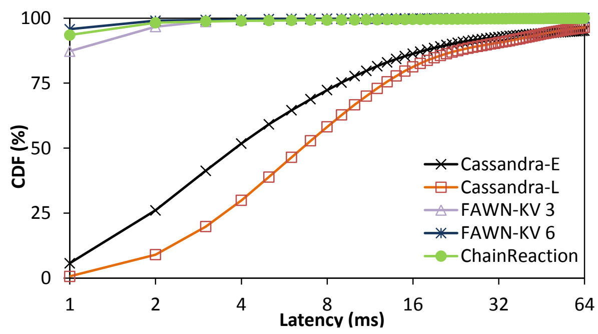
To run our experiments we used the following standard YCSB workloads: A, B, C, D and F. Workload E was not used because FAWN-KV and our system (by consequence of using FAWN-KV as a building block) do not support *scan* operations. In the first place, we want to make the following remark: all workloads executed by the YCSB are constructed in way that guarantees that all objects were accessed at least once during the workload execution. Workloads A, B, and C are similar as they all use a *Zipfian distribution* of the keys included in requests during the execution of the workload. They only differ in



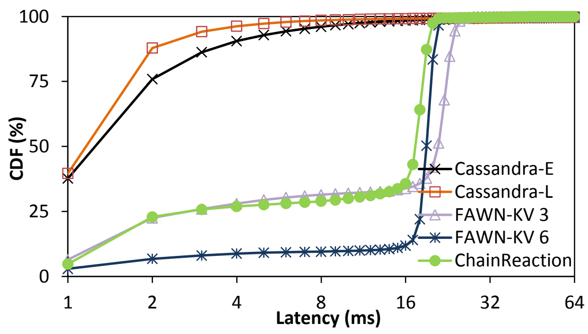
(a) Read (Workload A).



(b) Write (Workload A).



(c) Read (Workload F).



(d) Write (Workload F).

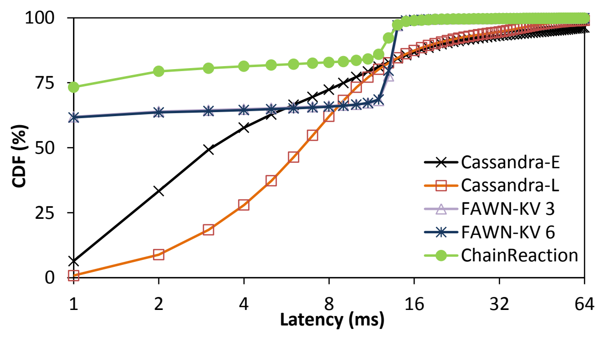
Figure 5.2: Latency CDF for Write-Heavy Standard Workloads A and F on a single datacenter deployment.

the ratio of write/read operations which is, respectively, 50/50, 5/95 and 0/100. The Zipfian distribution allows to simulate a scenario where there are objects there are more popular then the others, leading to a scenario where a set of keys are much more accessed than the rest. This scenario can be observed in a social distributed application, for example Facebook or Twitter, where there are profiles or pages that are more popular (celebrities, events, brands, among others), and thus more accessed than others.

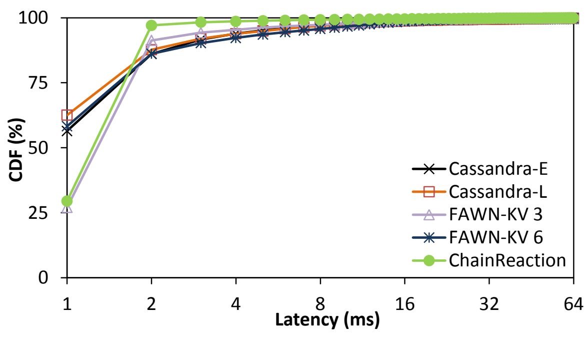
The workload D operates in a different manner than the previous workloads. It uses a *Latest Distribution* to distribute the access to certain keys during the workload execution and write/read operations are distributed in a 5/95 ratio. This distribution is similar to the Zipfian Distribution but the popularity of the keys is related with the time at which the object was inserted. This way, recently introduced objects are more accessed than objects inserted in the past. This mimics the behavior of some applications, for example Twitter, where users update their status and people want to follow their latest status.

Finally, the last standard workload, workload F, is similar to workloads A, B, and C as it also uses a zipfian distribution on key access. However, this workloads introduces a new type of operations: *read-modify-write*. This operation type considers that a read operation, a modification of the value in the client and a write of that need value is a single operation. This workload has a distribution of operations that is composed by 50% reads and 50% read-modify-write operations.

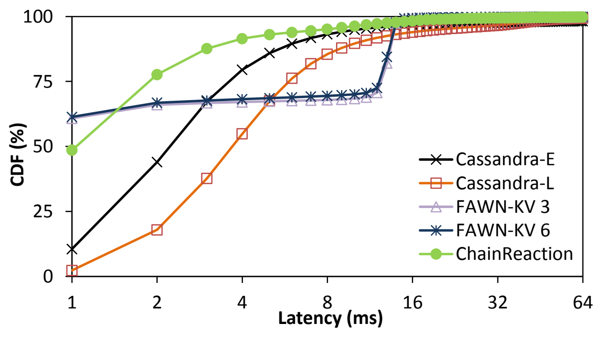
To conduct the micro benchmark we used custom made workloads to exercise our system and the



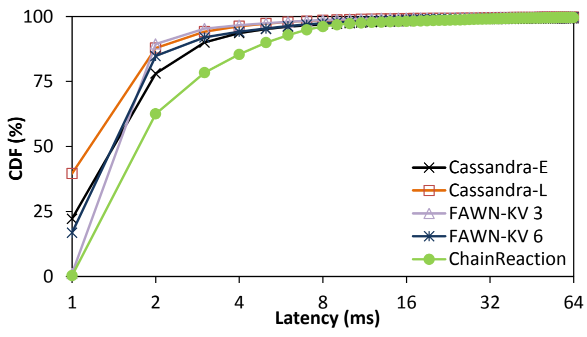
(a) Read (Workload B).



(b) Write (Workload B).



(c) Read (Workload D).



(d) Insert (Workload D).

Figure 5.3: Latency CDF for Multiple Objects Workloads B and D (single site).

remaining with a single object. These workloads use a special distribution called *Hotspot* that allows to focus operations in a certain range of keys with a certain intensity. For example, we could say that 80% of the operations are made over 10% of the keys. We configured this distribution to make 100% of the operation over a single key. To create new set of workloads we changed the write/read ratio of operations distribution and created the following workloads: 50/50, 25/75, 15/85, 10/90, 5/95 and 0/100

The throughput results are presented in Figure 5.1. Latency Cumulative Distribution Function (CDF) results [1] are presented in Figures 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7. Figure 5.1(a) shows that ChainReaction in a single datacenter outperforms both FAWN-KV and Cassandra in all standard YCSB workloads. In workloads A and F (which are write-intensive) the performance of ChainReaction approaches that of Cassandra-E and FAWN-KV 3. This is expected, since our chain-replication variant does not optimize write operations. In fact, for write-intensive workloads, it is expected that our solution under-performs when compared to FAWN-KV, given that ChainReaction needs to write on 6 nodes instead of 3 and also has to make sure, at each write operation, that all dependencies are stable before executing the next write operation. Fortunately, this effect is compensated by the gains in the read operations. This can be observed in the latency results for workload A and F in Figures 5.2(a), 5.2(b), 5.2(c), and 5.2(d). These figures also show that Cassandra exhibits a better write latency. Notice however, that Cassandra has

[1]Note that in all Latency CDF charts the CDF is in percentage and the latency is in a logarithmic scale.

much slower read operations than ChainReaction and FAWN-KV since it is optimized for write-heavy environments.

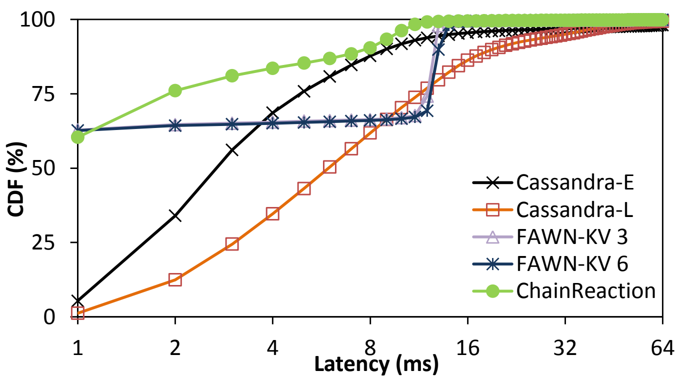
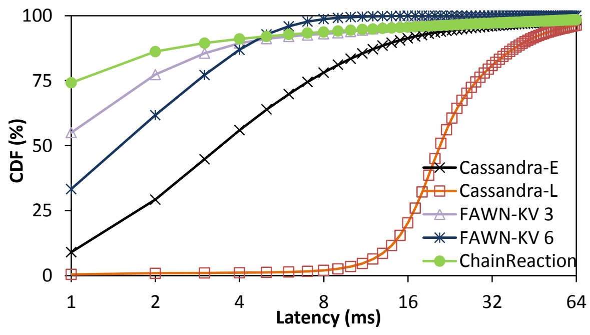


Figure 5.4: Latency CDF for Read-Only Standard Workload C on a single datacenter deployment.

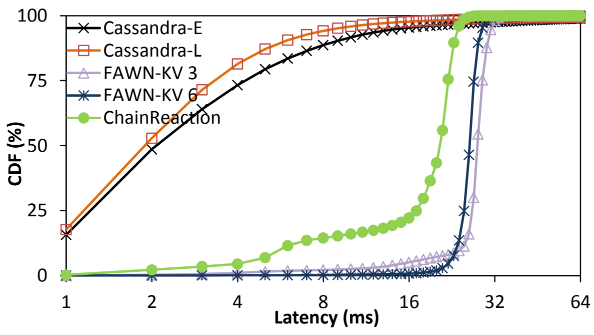
On the other hand, for workloads B and D, which are read-heavy, one expects ChainReaction to outperform all other solutions. Indeed, the throughput of ChainReaction in workload B is 178% better than that of Cassandra-E and 45% better than that of FAWN-KV 3. Performance results for workload D (Figure 5.1(a)) are similar to those of workload B. Notice that the latency of read operations for our solution is much lower when compared with the remaining solutions (Figures 5.3(a),5.3(b), 5.3(c), 5.3(d)). Additionally, in workload C (read-only) ChainReaction exhibits a boost in performance of 177% in relation to Cassandra-E and of 72% in relation to FAWN-KV 3.

The micro benchmark that relies on the custom single object workloads has the purpose of showing that our solution makes a better use of the available resources in a chain to improve the performance of read operations, when compared with the remaining tested solutions. In the write-heavy workload (50/50) one can observe that Cassandra-E outperforms our solution by 70%. This can be explained by the fact that Cassandra is highly optimized for write operations specially on a single object. However, when we rise the number of read operations our solution starts to outperform Cassandra by 13%, 20%, 34%, and 39% in workloads 25/75, 15/85, 10/90, and 5/95, respectively. In terms of latency one can see that ChainReaction always exhibits a better read latency than Cassandra-E having more operations to complete at lower latencies. We can also observe that as the number of reads increases the write latency of ChainReaction is also better than Cassandra-E write latency.

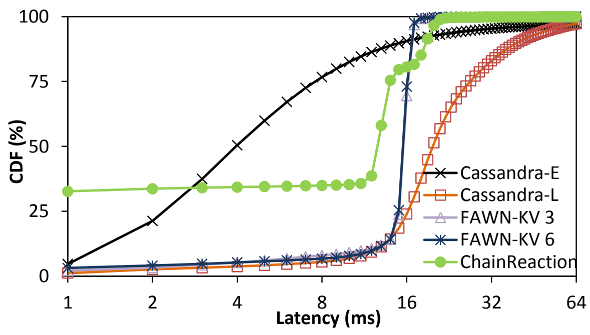
Additionally, ChainReaction outperforms FAWN-KV 3 and FAWN-KV 6 in all single object custom workloads. The performance increases as the percentage of read operations grows. Moreover, the throughput of the latter systems is always the same which can be explained by the fact that the performance is bounded by a bottleneck on the tail node. If a linear speedup was achievable, our solution operating with 6 replicas would exhibit a throughput 6 times higher than FAWN-KV on a read-only workload (0/100 workload) with a single object. Although the speedup is sub-linear.



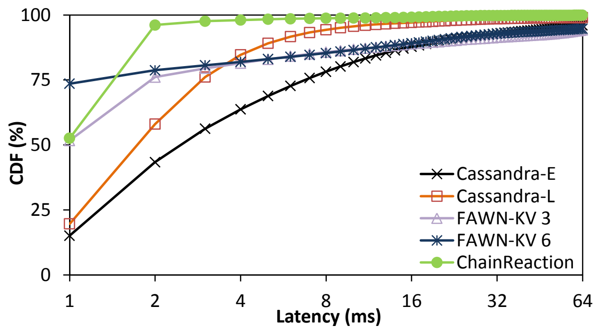
(a) Read (Workload 50/50).



(b) Write (Workload 50/50).

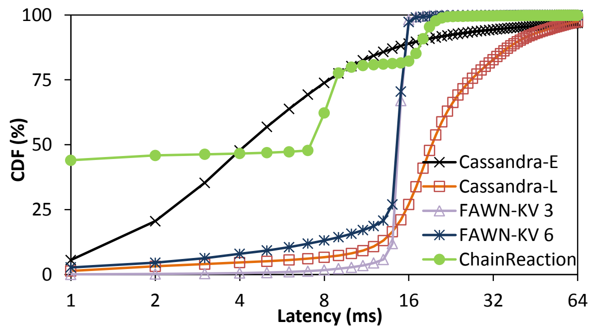


(c) Read (Workload 15/85).

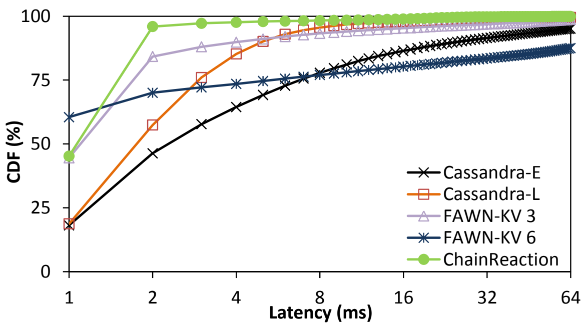


(d) Write (Workload 15/85).

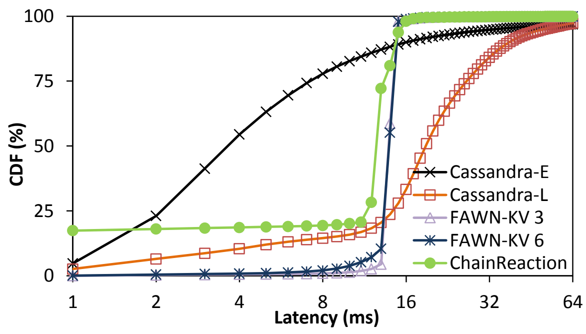
Figure 5.5: Latency CDF for Custom Workloads 50/50 and 15/85 on a single datacenter deployment.



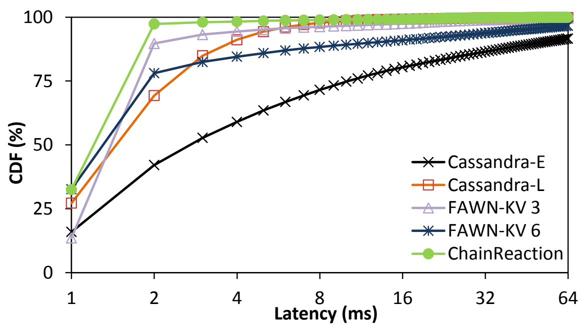
(a) Read (Workload 10/90).



(b) Write (Workload 10/90).



(c) Read (Workload 5/95).



(d) Write (Workload 5/95).

Figure 5.6: Latency CDF for Custom Workloads 10/90 and 5/95 on a single datacenter deployment.

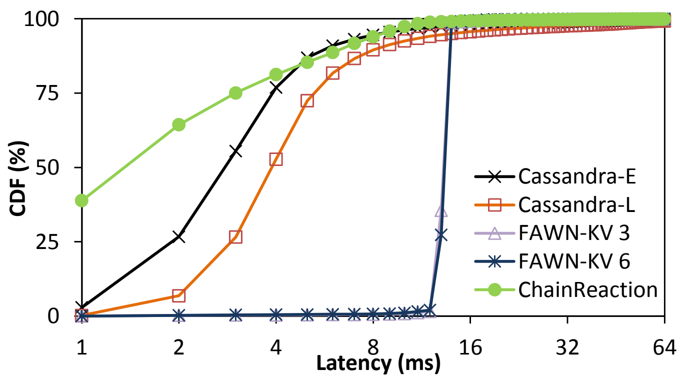


Figure 5.7: Latency CDF for Custom Workload 0/100 on a single datacenter deployment.

As depicted in Figure 5.1(b), still achieves a throughput that is 4.3 times higher than the throughput of FAWN-KV 3. The sub-linear growth is due to processing delays in proxies and network latency variations. Figures 5.5, 5.6, and 5.7 depicts the cumulative latency distribution for some of these workloads which shows that the latency results for ChainReaction always surpass the latency results of FAWN-KV 3 and FAWN-KV 6.

5.2 Multi Datacenter Experiments

To evaluate the performance of our solution in a Geo-replicated scenario, we ran the same systems, by configuring nodes in our test setup to be divided in two groups with high latency between them to emulate 2 distant datacenters. In this test setup, each datacenter was attributed 4 machines, and we used two machines to run the Yahoo! Cloud Serving Benchmark (each YCSB client issues requests to one datacenter). The additional latency between nodes associated to different datacenters, was achieved by introducing a delay of 120 ms (in RTT) with a jitter of 10 ms. We selected these values as we measured them with the PING command to www.facebook.com (Oregon) from INESC-ID (Lisbon). Each system was executed on 8 nodes (4 at each datacenter) and considered the following configurations:

Cassandra-E: Eventual-consistency with 4 replicas at each datacenter. Write operations are applied on 2 nodes. Read operations are processed at a single node.

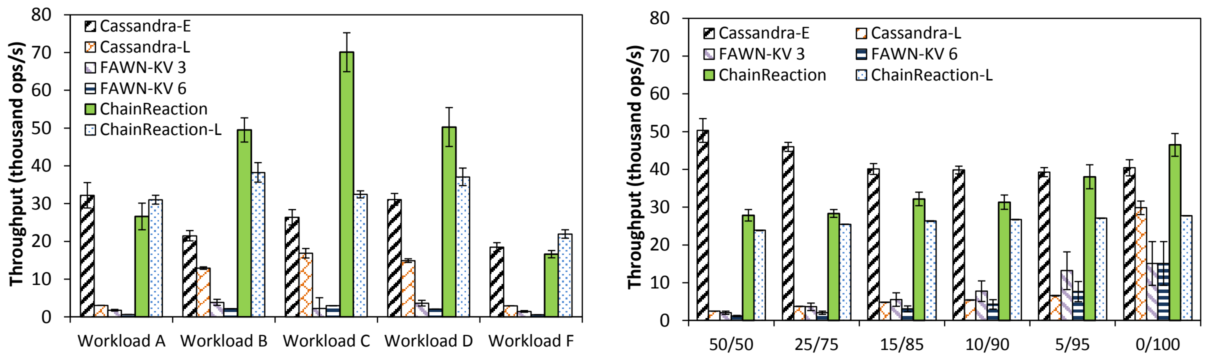
Cassandra-L: Eventual-consistency with 4 replicas at each datacenter. Writes and reads are made to a majority of nodes in each datacenter (6 nodes across both datacenters).

FAWN-KV 3: Deployment of FAWN-KV configured with a replication factor of 4. In this case the chain has a size of 4 nodes and it crosses the two datacenters (*i.e.*, some nodes of the chain can are in one datacenter and the rest on the other datacenter).

FAWN-KV 6: Deployment of FAWN-KV configured with a replication factor of 8. In this specific case all chains include a similar number of nodes in each datacenter.

ChainReaction: Deployment of our solution with a replication factor of 4 for each datacenter and a k equal to 2.

ChainReaction-L: We introduced a new system deployment that consists in our system although it offers stronger guarantees (linearizability) on the local datacenter with a replication factor of 4 nodes. This deployment allows to compare the performance with systems that offer stronger guarantees locally and weaker guarantees over the wide-area (in particular, COPS).

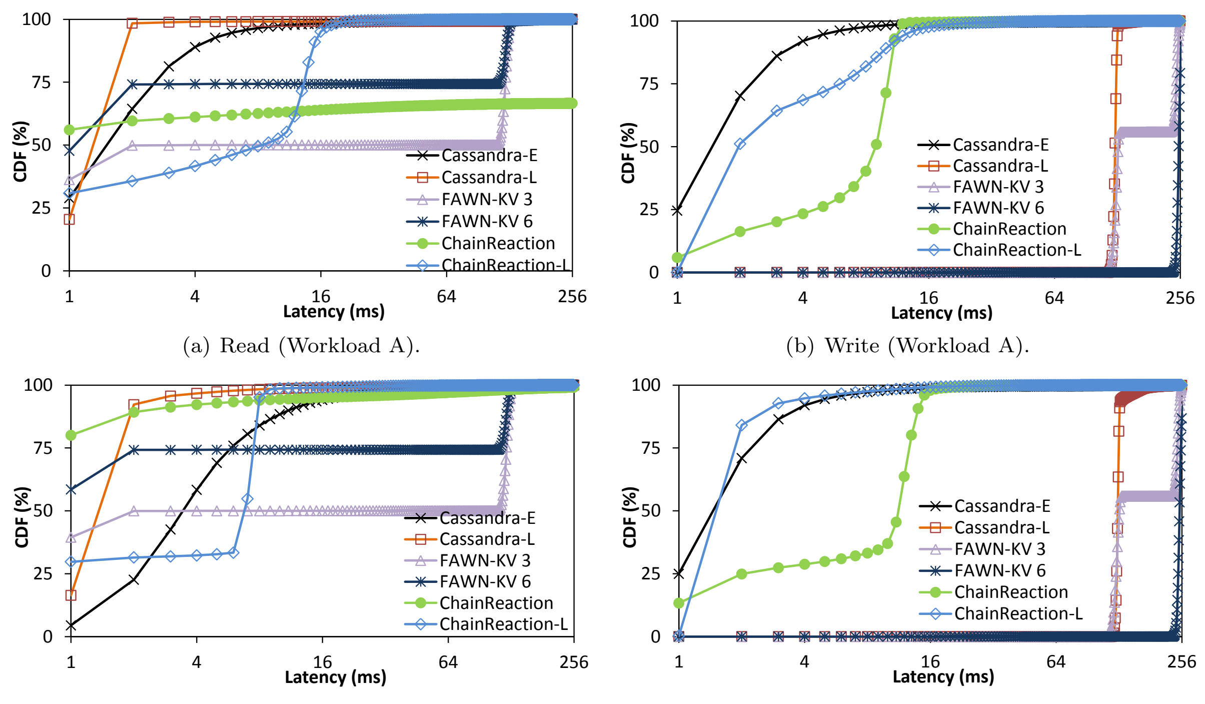


(a) Standard YCSB Workloads. (b) Custom Workloads (single object).

Figure 5.8: Throughput in thousand operations per second for each type of workloads considering a multiple datacenter deployment.

We employed the same workloads as in the previous experiment. However, in this case we run two YCSB clients (one for each datacenter) with 100 threads. We also divided the workload among the two sites, meaning that each workload generator performs 1,000,000 operations on top of 1,000,000 objects. We aggregated the results of the two clients and present them in the following plots.

The throughput results are presented in Figure 5.8. Latency Cumulative Distribution Function (CDF) results are presented in Figures 5.9, 5.10, and 5.11. Considering the standard YCSB workloads, we can see that ChainReaction outperforms the remaining solutions in all workloads except the write-heavy workloads (A and F) where Cassandra-E and ChainReaction-L are better. These results indicate that ChainReaction, Cassandra-E, and ChainReaction-L are the most adequate solutions for a Geo-replicated deployment. The difference in performance between our solution and Cassandra-E is due to the fact that Cassandra offers weaker guarantees than our system and is also optimized for write operations resulting in an increase in performance. When comparing with ChainReaction-L our system needs to guarantee that a version is committed before proceeding with a write operation while ChainReaction-L does not, leading to some delay in write operations. In terms of latency in write-heavy workloads our system is



(a) Read (Workload A). (b) Write (Workload A).

(c) Read (Workload F). (d) Write (Workload F).

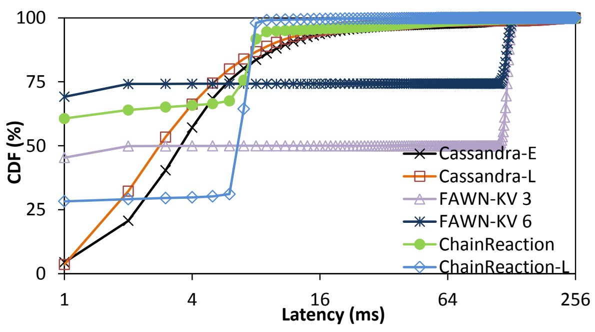
Figure 5.9: Latency CDF for Write-Heavy Standard Workloads A and F on a multiple datacenter deployment.

slightly outperformed by to Cassandra-E and ChainReaction-L. However, our system exhibits a good behavior in workload F read operation latency (Figure 5.9(c)).

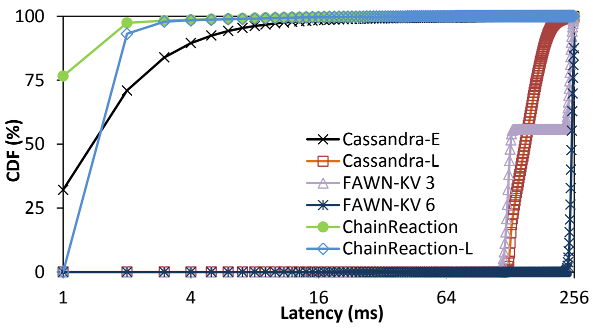
On read-heavy workloads (B and D), our solution surpasses both Cassandra-E and ChainReaction-L achieving 56%/22% better throughput in workload B and 38%/26% better performance in workload D. The latency results depicted in Figure 5.10 show that our solution provides better write and read latency than the other solutions. Finally, on workload C our solution exhibits an increase in performance of 62% and 53% in comparison with Cassandra-E and ChainReaction-L, respectively.

The low throughput of Cassandra-L and both FAWN-KV deployments is due to the fact that write operations always have to cross the wide-area network, inducing a great latency in operations. Moreover, in FAWN-KV (original chain replication) when the objects' chain tail is on a remote datacenter, read operations on that objects must cross the wide-area. Additionally, ChainReaction has a significantly higher throughput than FAWN-KV 3 ranging from 1,028% (workload F) to 3,012% (workload C) better. The comparison of the results for the remaining systems is similar.

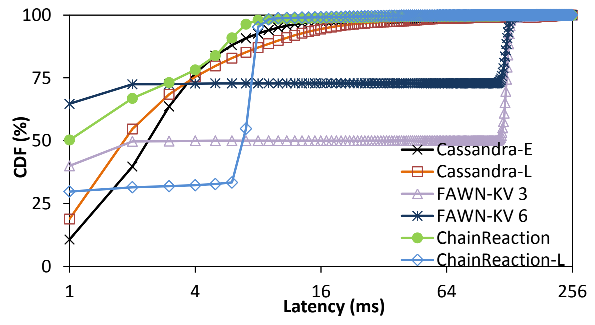
The results for the micro benchmark (Figures 5.8(b), 5.12, 5.13, and 5.14) in the Geo-replicated scenario are interesting because they show that the original Chain Replication protocol is not adaptable to a Geo-replicated scenario. The large error bars for both FAWN-KV deployments are a result of the difference in throughput in each datacenter. The client that has the tail of the object in the local



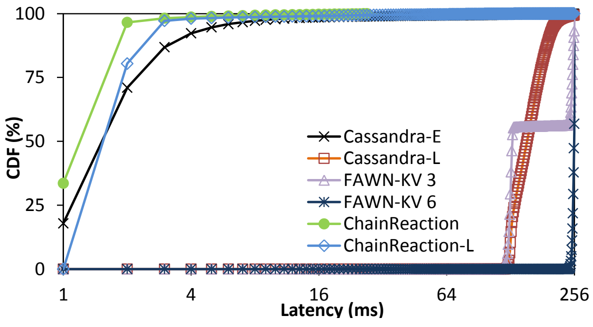
(a) Read (Workload B).



(b) Write (Workload B).



(c) Read (Workload D).



(d) Write (Workload D).

Figure 5.10: Latency CDF for Read-Heavy Standard Workloads B and D on a multiple datacenter deployment.

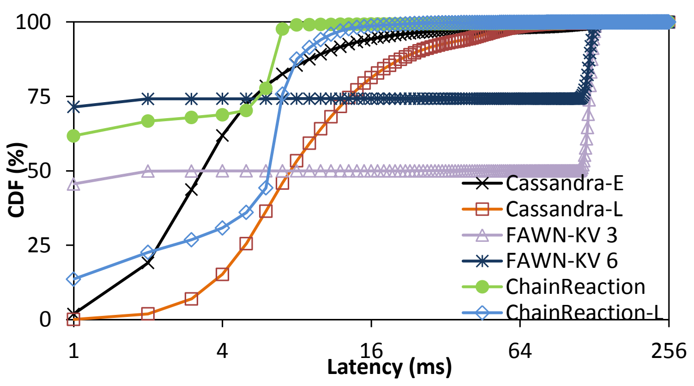
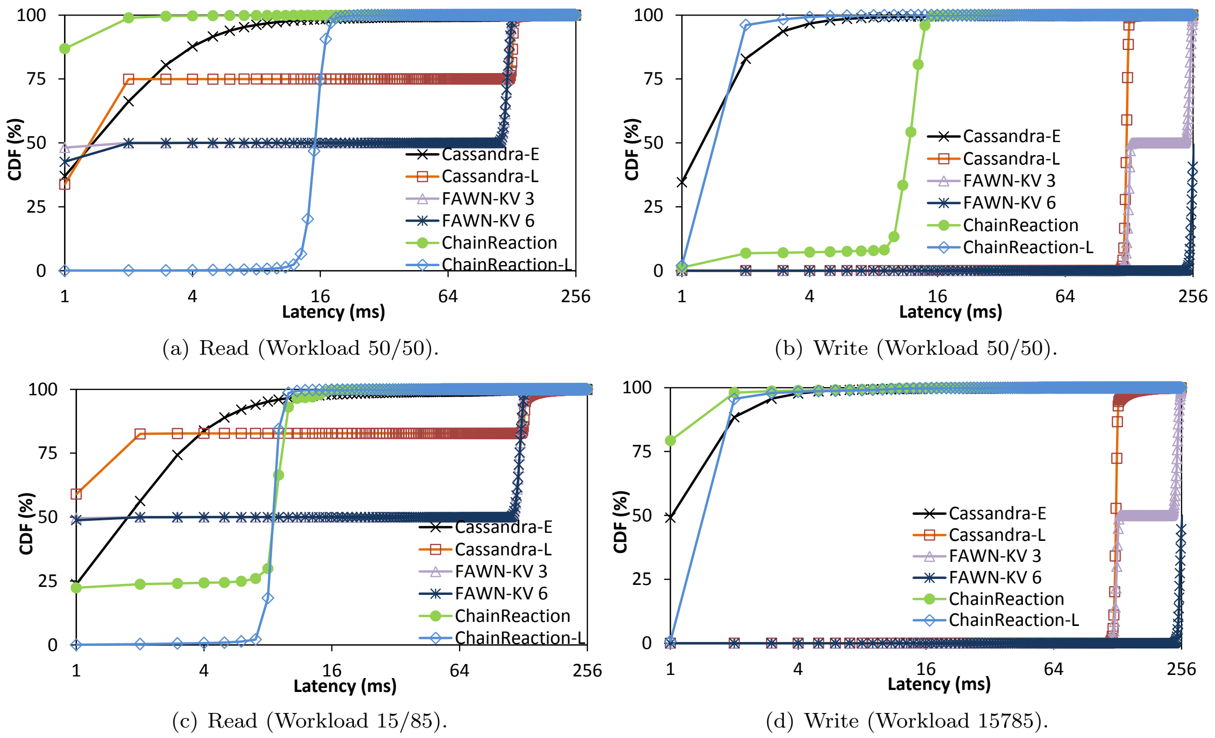


Figure 5.11: Latency CDF for Read-Only Standard Workload C on a multiple datacenter deployment.

datacenter has a better read throughput than the client on the remote datacenter, resulting in a great difference in each datacenter performance. Our solution outperforms FAWN-KV 3 in all workloads with a difference that ranges from 188% (Workload 5/95) to 1,249% (Workload 50/50). In terms of latency its possible to observe that 50% of write operations in FAWN-KV 3 take more than 100 ms to complete corresponding to the latency introduced between datacenters. The results for Cassandra-L and FAWN-KV 6 are similar.

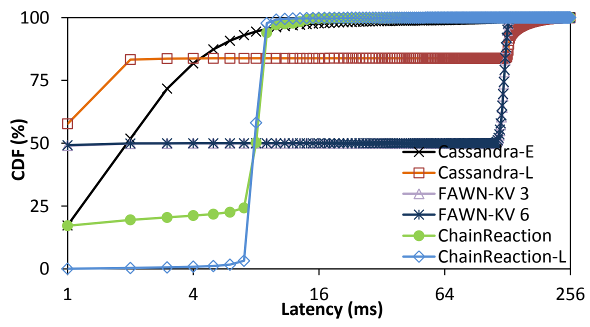


(a) Read (Workload 50/50). (b) Write (Workload 50/50).

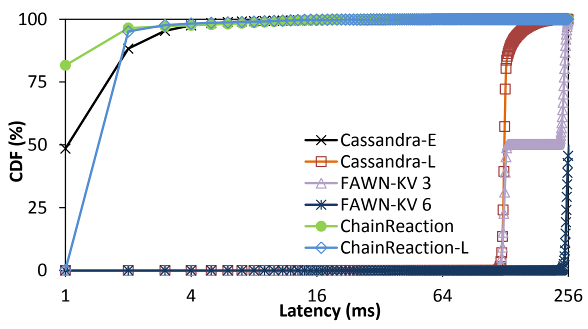
(c) Read (Workload 15/85). (d) Write (Workload 15785).

Figure 5.12: Latency CDF for Custom Workloads 50/50 and 15/85 on a multiple datacenter deployment.

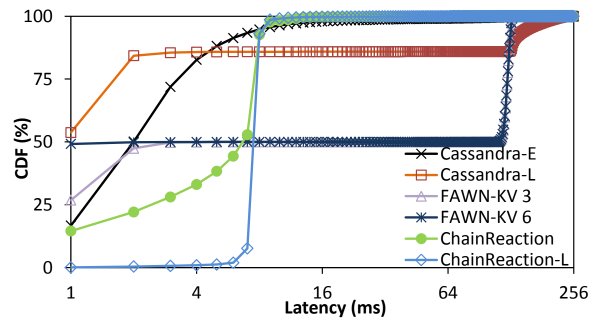
Also, the results shows that Cassandra-E outperforms our solution in all single object workloads with exception to the read-only workload (where our solution is 15% better). This happens because Cassandra behaves better with a single object and is optimized for write operations (achieving better performance in write-heavy workloads like in the single datacenter scenario). Moreover, performs better than ChainReaction-L in all workloads achieving an improvement of 17%, 11%, 22%, 17%, 40%, and 68% in workloads 50/50, 25/75, 15/85, 10/90, 5/95 and 0/100, respectively. In terms of latency our solution exhibits lower latency than Cassandra-E and ChainReaction-L in write operations in all workloads with exception to workload 50/50 where Cassandra-E performs better. Additionally, our solution always exhibits lower latency than ChainReaction-L in all workloads, although Cassandra-E has better latency on read operations.



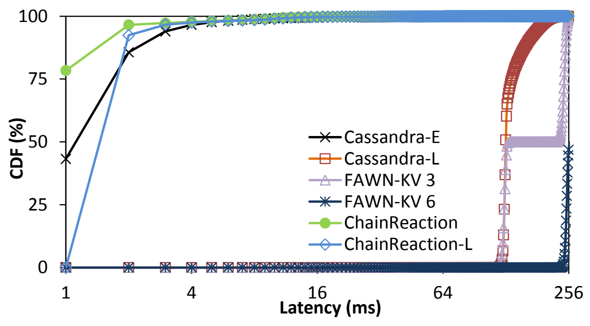
(a) Read (Workload 10/90).



(b) Write (Workload 10/90).



(c) Read (Workload 5/95).



(d) Write (Workload 5/95).

Figure 5.13: Latency CDF for Custom Workloads 10/90 and 5/95 on a multiple datacenter deployment.

5.3 Get-Transactions Experiments

In this experiment we evaluate the performance of GET-TRANSACTION operations in the Geo-replicated scenario. In this case we only executed ChainReaction (the other solutions do not support this operation) deployed in the 8 machines (4 in each simulated datacenter) like in the previous scenario. We have attempted to perform similar tests with COPS, unfortunately, we were unable to successfully deploy this system across multiple nodes. We have created three custom workloads and changed the YCSB source in order to issue GET-TRANSACTION operations. The created workloads comprise the following distribution of write, read and GET-TRANSACTION operations: 10% writes, 85% reads, 5% GET-TRANSACTIONS on workload 10/85/5, 5% writes, 90% reads, 5% GET-TRANSACTIONS on workload 5/90/5, and 95% reads, 5% GET-TRANSACTIONS on workload 0/95/5. A total of 500,000 operations were executed over 10,000 objects, where a GET-TRANSACTION includes 2 to 5 keys (chosen randomly). This workload was executed by 2 YCSB clients (one at each datacenter) with 100 threads each.

Results depicted on Figure 5.15 show that we achieve an aggregate throughput that approximates of 12,000 operations per second in all workloads showing that the percentage of write and read operations do not affect the performance of GET-TRANSACTIONS and vice-versa.

In terms of operation latency we can see on Figure 5.16 that the introduction of GET-TRANSACTIONS does not affects the latency of write and read operations in those 3 workloads. On the other hand, since

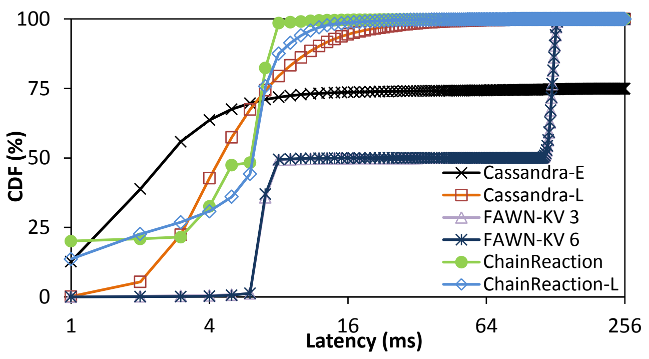


Figure 5.14: Read latency CDF for Custom Workload 0/100 on a multiple datacenter deployment.

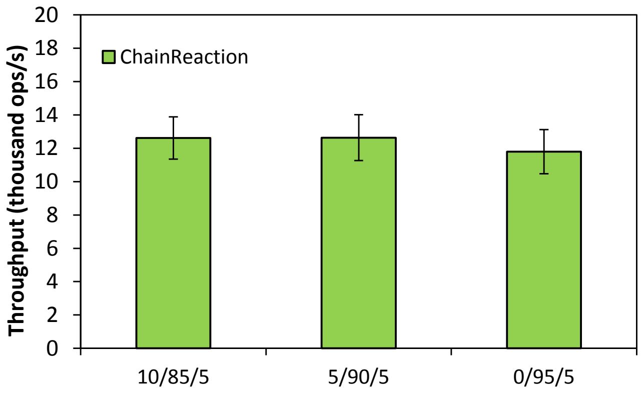


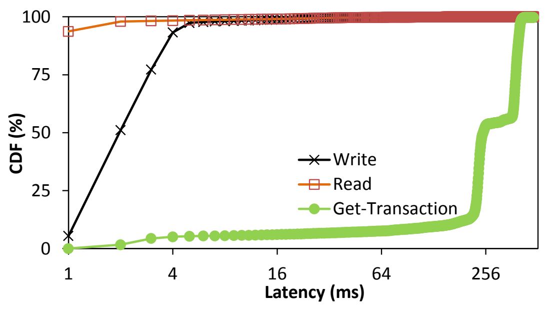
Figure 5.15: Throughput with Custom Multi-Read Workloads (multiple sites).

we give priority to the other two operations, the average latency for GET-TRANSACTIONS is in the order of approximately 400 ms (which we consider acceptable from a practical point of view).

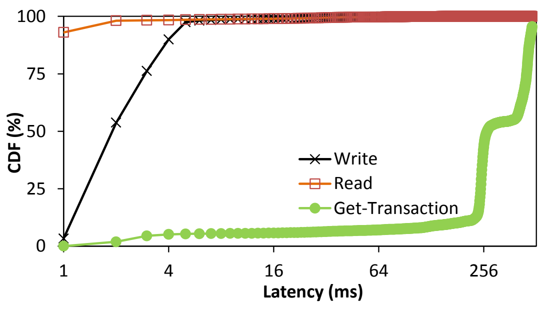
5.4 Fault-Tolerance Experiments

To assess the behavior of our solution when failures occur we deployed ChainReaction in a single data center with 9 data nodes and a single chain, with a replication factor of 6 and a k equal to 3. A single chain was used so that the failures could be targeted to the different zones of the chain. We used the custom-made workloads 50/50 and 5/95 to measure the average throughput of our solution during a period of 140 seconds of execution time. During the workload we failed a single node at 60 seconds of execution. We tested two scenarios of failure: a) a random node between the head and node k (including k); b) a random node between k and the tail (excluding k). The workloads were executed with 100 client threads that issue 3,000,000 operations over a single object.

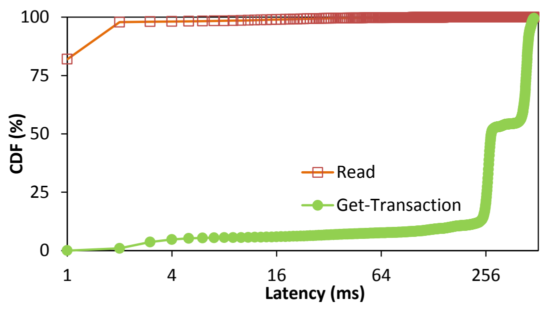
The results for the average throughput during execution time can be observed in Figure 5.17. In the first scenario, depicted by Figure 5.17(a), one can observe that the failure of a node between the head of



(a) Workload 10/85/5.



(b) Workload 5/90/5.

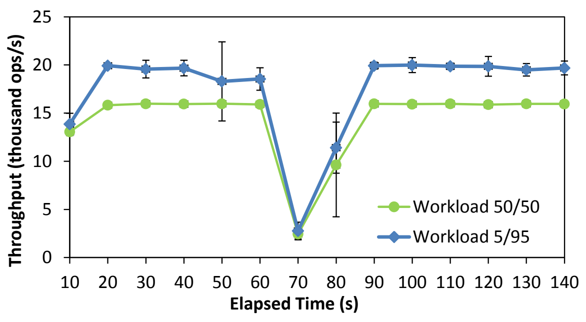


(c) Workload 0/95/5.

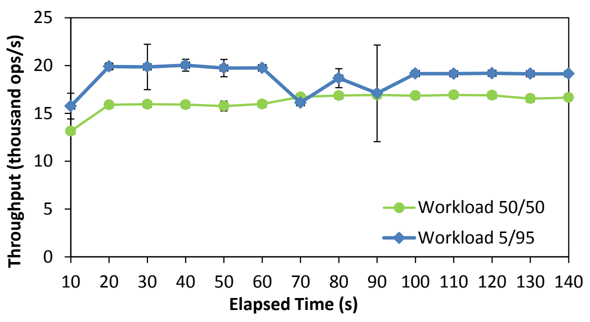
Figure 5.16: Latency CDF for Custom Multi-Read Workloads 10/85/5, 5/90/5 and 0/95/5 on a multiple datacenter deployment.

the chain and node k results in a drop in throughput. This drop reaches approximately 2000 operations per second in both workloads and is due to the fact that write operations are stalled until the failure is detected and the chain is repaired. This drop in throughput is somewhat pronounced, however all operations are being executed by a single chain. In a real scenario, with multiple chains, operations in chains that were not affected by the failure could continue normal operation reducing the impact on the global throughput. Also, 20 seconds after the failure of the node the throughput starts increasing reaching its initial peak 30 seconds after the failure. The results for the second scenario, depicted in Figure 5.17(b), show that the failure of a node after node k has a reduced impact in the performance of the system, as the write operations can terminate with no problems. Also, the variations of the throughput during the repair of the chain are due to the fact that read operations are processed by only 5 nodes while the chain is repaired. This variation is most noted in Workload 5/95 which is read-heavy.

These results allow to show that our replication technique is more resilient to failures than the original chain replication protocol (in the original chain replication protocol write/read throughput reaches 0 for a certain time interval (van Renesse & Schneider 2004)). All active nodes can continue to process read operations when a failure of a node occurs, although write operations may have to be delayed (until the chain connectivity is repaired) if the failing node belongs to the first k nodes of the chain.



(a) Node between the head of the chain and node k.

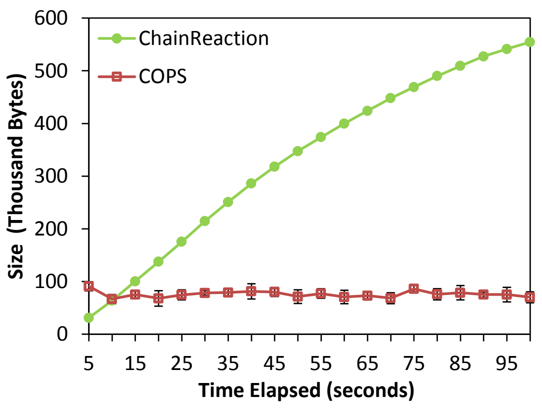


(b) Node between $k$and the tail of the chain.

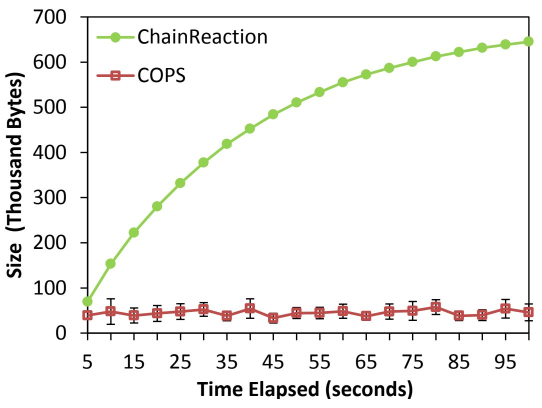
Figure 5.17: Average throughput during the first 140 seconds of execution in workloads 50/50 and 5/95. In both scenarios a single node has failed at 60 seconds of execution time. The node that has failed is: a) a random node between the head and node k (including k); b) a random node between k (excluding k) and the tail.

5.5 Metadata Experiments

In the last experiment we focus on measure the overhead, in terms of space, imposed by the metadata stored and traded in our system. To evaluate this overhead we compared our solution and COPS by deploying both systems in a single machine. This machine uses Ubuntu 11.04 and has a 2 core Intel Core2Duo P8600 CPU, 4GB RAM, and 360 GB Hard Drive. We simulated two datacenters in a single machine and deployed a data node for each datacenter. The two systems were also tested using the Yahoo! Cloud Serving Benchmark standard workloads. Each workload was executed by 10 YCSB client threads that submitted a total of 1,000,000 operations over 1KByte objects. In each execution we measured, during the first 100 seconds of execution, the size of metadata stored in the client, the size of metadata sent across datacenters and the metadata included in requests/replies of write and read operations. The presented measurements are result from an average of the 10 clients, the error bars are also displayed.

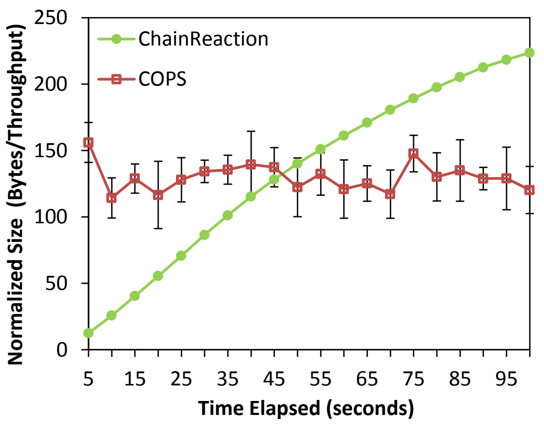


(a) Metadata Size Workload A.

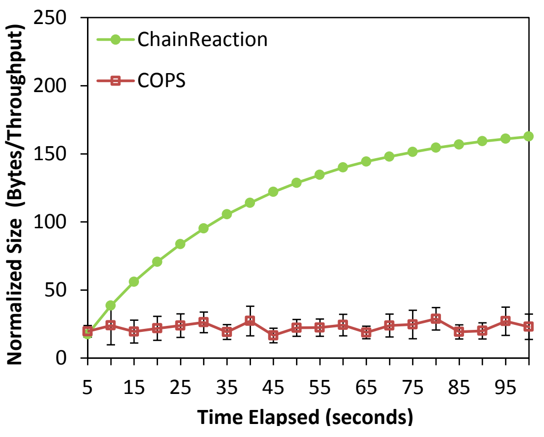


(b) Metadata Size Workload B.

Figure 5.18: Size of the metadata stored in client in standard workloads A and B.



(a) Normalized Metadata Size Workload A.



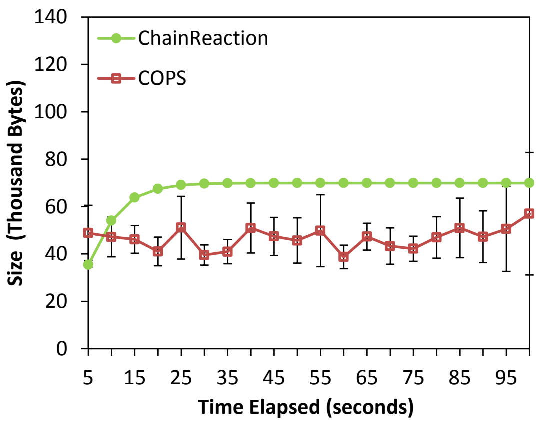
(b) Normalized Metadata Size Workload B).

Figure 5.19: Normalization of the size of the metadata stored in client in standard workloads A and B.

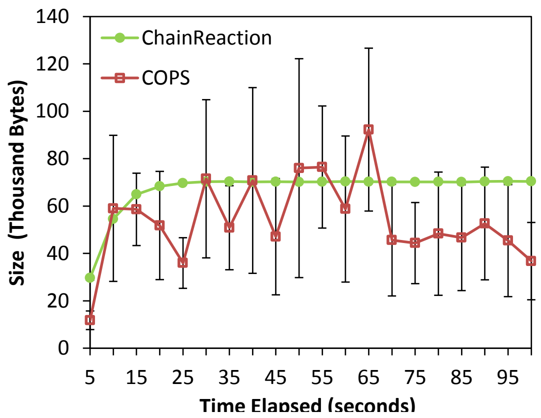
To test the amount of metadata stored in the client we ran the standard workloads A (write-heavy) and B (read-heavy) with 10,000 objects. The results can be observed in Figure 5.18, showing that in both workloads our solution stores more metadata than COPS in both workloads, peaking at 554,000 Bytes (541 KBytes) in workload A and at 645,000 Bytes (630 KBytes) in workload B. This results are due to the fact that our solution stores a metadata entry for each object accessed. As the YCSB accesses all objects until the end of the workload we can observe the size of metadata increasing during the 100 seconds of the execution. However, the size of metadata stored in COPS clients does not depend on the objects that are accessed. In COPS, the metadata entries are garbage collected if the object version has already been committed to all datacenters. So the size of metadata stored depends on the time that operations take to complete (and propagate to remote datacenters) and the number of concurrent operations being executed.

Additionally, the throughput on both systems is different. Our system exhibits a throughput of 2480 operations per second on workload A and 3967 ops/s on workload B. On the other hand, COPS only exhibits 583 ops/s on workload A and 1992 ops/s on workload B. To incorporate these differences in the results, in Figure 5.19 we show the size of metadata stored normalized by the throughput achieved in the workloads. In this case one can observe that the metadata size stored in our solution clients approximates the size of metadata stored in COPS.

To assess our hypothesis about the size of our metadata increasing with the number of accessed objects, we choose to run the same workloads but with only 1000 objects. The results in Figures 5.20 and 5.21 show that the size of the metadata stored in our clients peaks at approximately 70,000 Bytes (68 KBytes) in both workloads. One can also observe that our solution approximates the results of COPS and is also more stable than COPS. In the normalized results our solution surpasses COPS is both workloads storing less metadata by client. Although, we could improve our metadata garbage collection

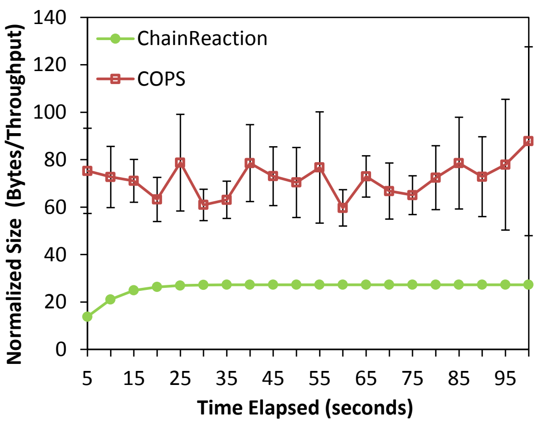


(a) Metadata Size Workload A.

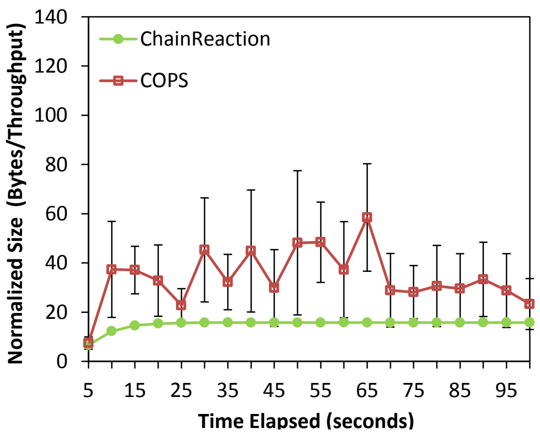


(b) Metadata Size Workload B.

Figure 5.20: Size of the metadata stored in client in standard workloads A and B (with 1000 objects).



(a) Normalized Metadata Size Workload A.

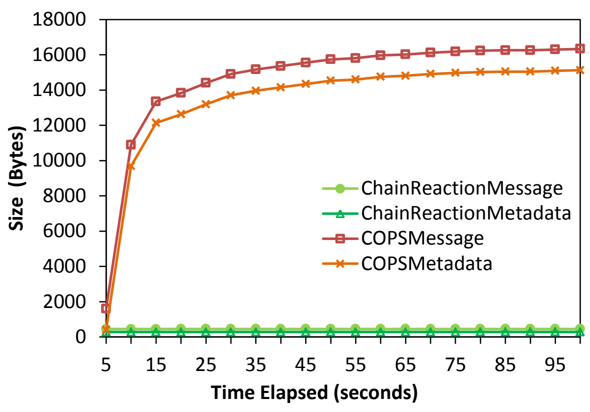


(b) Normalized Metadata Size Workload B).

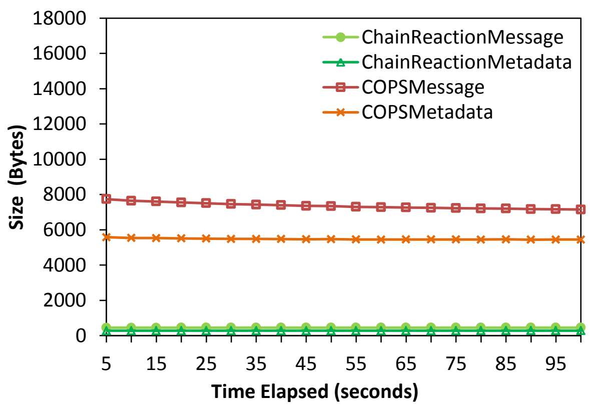
Figure 5.21: Normalization of the size of the metadata stored in client in standard workloads A and B (with 1000 objects).

since we are currently focused in garbage collecting the *Accessed Objects* list. To this end, we could use a *best effort* mechanism in which client proxies would also store some metadata for each object. Every time the proxy receives a response message from the data nodes would update the metadata and would inform the relevant clients if the version of the object is already stable. To gather this information, each client would piggyback in each message a fixed number of object versions that she wants to know about (decided randomly). The proxy would then piggyback information about these versions in the response sent to the client, which would remove the metadata entry if the version is stable. This would allow for clients to garbage collect metadata entries without accessing the relevant objects. However, we do not implement this mechanism and is treated as future work.

In Figure 5.22 are depicted the results for the measurements of the size of the propagation messages (and corresponding metadata) sent across datacenters. We can observe that COPS sends much more data



(a) Average size of one message in workload A.



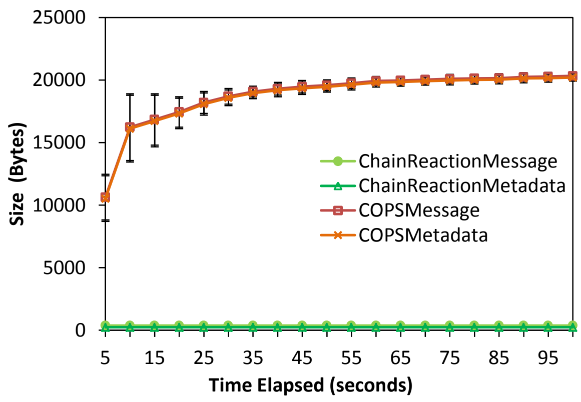
(b) Average size of one message in workload B.

Figure 5.22: Average message and metadata size in propagation messages between datacenters on workloads A and B.

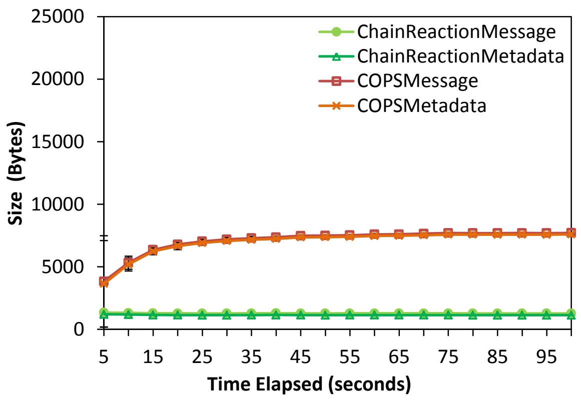
across datacenters, increasing during the execution of the workload, peaking at an average of approximately 16,300 Bytes (16 KBytes) in workload A and 7,700 Bytes (7 KBytes) in workload B. However, in our the solution the average size of one message remains stable during the workload execution and is the same for both workloads peaking at 450 Bytes. One can also observe that on COPS most of the message payload is metadata in the form of dependencies reaching 92% of the message payload in workload A and 76% in workload B. In our solution the metadata size is approximately 280 Bytes corresponding to the key, value and the two bloom filters sent in the message.

The measurements for the size of write requests are presented in Figure 5.23. Similar to the results for the propagation messages we can observe that COPS also sends larger write requests than our solution. Also the size of messages, in COPS, increases during the execution of both workloads peaking at 20,000 Bytes (19 KBytes) in workload A and 7,500 Bytes (7 KBytes) in workload B. The behavior exposed by our system is similar to the latter scenario as the size of messages remains stable at 380 Bytes in workload A and 1200 Bytes in workload B. The difference in the size between workloads is due to the fact that workload B is read-heavy and since our solution sends that last accessed keys between write operations, the number of read operations between write operations is higher in workload B. Moreover, in both systems the majority of the message payload is metadata. In terms of write replies both systems present the same message size during both workload execution. Our system exhibits a write reply message size of 180 Bytes while COPS reply message has a size of 24 Bytes. In our solution the reply message includes the bloom filter identifying the written object version which explains the larger message size.

In terms of read messages, in both systems the read request messages have a constant size that is stable during the workload execution. Our read request message has a payload of 38 Bytes (including 4 Bytes for the *chainIndex*) while COPS has a message size of 16 Bytes. The results for the size of received read replies are shown in Figure 5.24. The results for read reply message resemble the results of the two

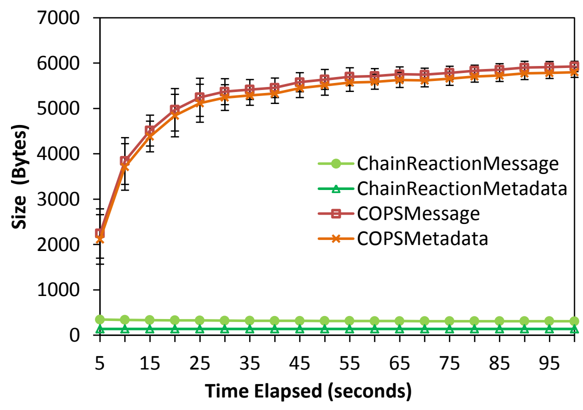


(a) Average size of one message in workload A.

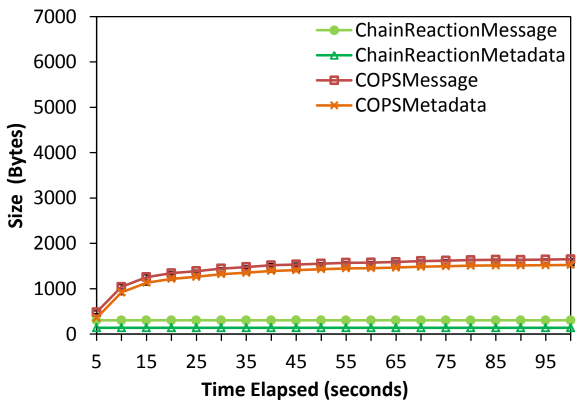


(b) Average size of one message in workload B.

Figure 5.23: Average message and metadata size sent by the client in write operations on workloads A and B.



(a) Average size of one message in workload A.



(b) Average size of one message in workload B.

Figure 5.24: Average message and metadata size received by the client in read operations on workloads A and B.

previous scenarios where COPS exhibits much larger read reply messages than our system. Moreover, COPS read replies peak at a size of 5,700 Bytes in workload A and 1,600 Bytes at workload B, while in our system the messages maintain a constant size of 140 Bytes (includes the bloom filter representing the read version and the *chainIndex*) in both workloads. Also most of the payload of the messages in both systems is metadata.

The results presented in this section allow to conclude that our system although storing more metadata in the clients provides lower message size increasing the efficiency of communication. The size of COPS messages can be problematic has larger messages can lead to more packet loss and network delays specially when crossing the wide area network. This way, our system provides better efficiency in metadata transactions at the cost of storing more metadata in the clients, although the increase in storage peaks at around 600 KBytes in a scenario where a client accesses all objects in the system. This amount of memory is negligible in most existing hardware even in mobile devices.

Summary

In this chapter we introduced the experimental evaluation made to ChainReaction and its results. In each section a different experiment was described starting by the used experimental setting, followed by the introduction of the system deployments that were subject of the experiment and finally we present the results for each experiment. In the first experiment we tested the performance of ChainReaction in a single datacenter against other solutions, showing that it achieves better performance than the competitors. Next we introduced the results for the experiments made with multiple datacenters showing that the benefits obtained in a single datacenter are reflected in this scenario. To assess the behavior of ChainReaction when failures occur, we introduced results for fault detection and recovery showing that our system continues operation and exhibits fast recovery when a failure occurs. Finally, we tested our system against COPS in order to compare the overhead in terms of metadata of the two solutions. The experimental results show that our solution improves network utilization due to the reduced size of messages when comparing to COPS.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduced some directions in terms of future work.

6 Conclusions

This thesis focused on studying mechanisms to build systems that provide storage for distributed applications which is a very active area of research. These systems are usually needed in applications that support a large user base operating at a large scale. Therefore, storage systems need to offer both consistency, high availability and partition tolerance. However, the CAP Theorem shows that is not possible to achieve the three properties in a distributed system. Therefore, these systems must choose to offer two of those properties in scalable way to overcome the large number of users of cloud computing applications.

Chapter 3 introduced ChainReaction, a new distributed key-value store that offers high-performance, scalability, and high-availability. This datastore can be deployed in a single datacenter scenario or across multiple datacenters, in a Geo-replication scenario, which is compatible with applications built using the Cloud Computing paradigm. ChainReaction achieves better performance than competing solutions in both scenarios by employing a novel replication protocol that is based on chain replication. Our solution offers the recently introduced *causal+* consistency guarantees, which are useful for programmers, in a Geo-replicated scenario. We also employ a metadata compression method, based on Adaptable Bloom Filters, with the purpose of avoiding the exchange of large messages over the wide area network. Similarly to COPS, we also provide a transactional construct called GET-TRANSACTION, that allows to get a consistent view over a set of objects. We also have implemented a prototype of ChainReaction using a optimized version of the FAWN-KV Key-value store as base.

To assess the performance of our prototype, we used the Yahoo! Cloud Serving Benchmark to test our solution against existing datastores. Experimental results using YCSB standard and custom workloads show that ChainReaction outperforms Cassandra, FAWN-KV, and a simulation of COPS in most workloads in single and multiple datacenter scenarios. We also evaluated our solution in a scenario with failures showing that our system continues to operate during the failure of a node, contrary to the original Chain Replication protocol. Also the recovery of a failure is made in a very efficient way by adding a node in the tail of the chain, which do not affect the normal operation of the system. Moreover, we measured our system metadata overhead in terms of space and compared the results with COPS, showing that our system provides better communication efficiency.

The work developed and described in this thesis is far from finished and can be further developed in many ways. First of all, for the system to be deployed in a real production setting, the fault-tolerance mechanisms for the local datacenter (specially regarding GET-TRANSACTION operations) should be refined so that they could be more robust and resilient to real failures. Also, we need to consider the possibility of failures in wide-area operation (replication over datacenters), namely entire datacenter unavailability. This way, a mechanism for full datacenter recover and synchronization is need in a real production scenario. These new mechanisms should be tested and experimentally evaluated to assess their robustness and efficiency.

As mentioned previously, even if our solution sends less metadata than COPS over the network we could improve the amount of metadata stored in the client library. To this end, the client library combined with the proxies should implement a best-effort mechanism so that clients could remove metadata entries that refer to objects that are already stable. This mechanism would allow for clients to garbage collect some metadata without accessing all the objects (*i.e.*, garbage collection information should be piggybacked on all messages).

To provide GET-TRANSACTIONS we make use of a centralized sequencer to implement the *atomic multi-enqueue* operation. In our prototype this solution is reasonable as the overhead of the sequencer is completely negligible. However, in a real scenario with hundreds of servers this sequencer can become a bottleneck of the system. To this end, in future work the solution to provide sequence numbers should get improved by implementing a distributed sequencer to balance the load. Moreover, this sequencer should also be resilient to failures and sequence number loss (due to the failure of client proxies).

Finally, we propose the development of a new type of operation similar to GET-TRANSACTIONS that would allow a client to write multiple objects that depend on each other. For example, considering an social application where users can have a friend relationship, if Bob adds Alice to friends then both profiles would need to be changed to cope with this new relation. If one of the profiles is changed and the other is not then it makes no sense that Alice is a friend of Bob but Bob is not a friend of Alice, as friendship is a binary relationship.

References

Agrawal, D. & A. El Abbadi (1991, February). An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems 9*(1), 1–20.

Ahamad, M., G. Neiger, J. Burns, P. Kohli, & P. Hutto (1995). Causal memory: definitions, implementation, and programming. *Distributed Computing 9*, 37–49. 10.1007/BF01784241.

Almeida, S., J. Leitão, & L. Rodrigues (2012). ChainReaction: uma variante de replicação em cadeia com coerência causal+. INForum 2012, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias.

Andersen, D. G., J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, & V. Vasudevan (2011, July). FAWN: a fast array of wimpy nodes. *Communications of the ACM 54*(7), 101–109.

Baker, J., C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, & V. Yushprakh (2011). Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Asilomar, CA, USA, pp. 223–234.

Belarami, N., M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, & J. Zheng (2006). PRACTI replication. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI '06, San Jose, CA, pp. 5–5. USENIX Association.

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil, & P. O'Neil (1995). A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, San Jose, California, United States, pp. 1–10. ACM.

Bernstein, P. A. & N. Goodman (1981, June). Concurrency control in distributed database systems. *ACM Computing Surveys 13*, 185–221.

Bernstein, P. A., V. Hadzilacos, & N. Goodman (1987). *Concurrency control and recovery in database systems*. Addison-Wesley.

Braam, P. J. (1998, June). The coda distributed file system. *Linux J. 1998*(50es).

Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, Portland, Oregon, United States, pp. 7. ACM.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, & R. E. Gruber (2008, June). Bigtable: A distributed storage system for structured data. *ACM*

Trans. Comput. Syst. 26(2), 4:1–4:26.

Charron-Bost, B., F. Pedone, & A. Schiper (2010). *Replication theory and practice*, Volume 5959. Springer Berlin.

Cooper, B. F., R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, & R. Yerneni (2008, August). PNUTS: Yahoo!'s hosted data serving platform. Volume 1, pp. 1277–1288. VLDB Endowment.

Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, & R. Sears (2010). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, Indianapolis, Indiana, USA, pp. 143–154. ACM.

Corbett, J. C., J. Deana, 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, & D. Woodfordh (2012). Spanner: Google's globally-distributed database. In *Proceedings of the 10th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI ' 12, Hollywood, CA, USA.

Couceiro, M., P. Romano, N. Carvalho, & L. Rodrigues (2009). D2STM: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, Washington, DC, USA, pp. 307–313. IEEE Computer Society.

Gustavsson, S. & S. F. Andler (2002). Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, Charleston, South Carolina, pp. 105–107. ACM.

Hastorun, D., M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007). Dynamo: amazon's highly available key-value store. In *Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles*, SOSP '07, Stevenson, Washington, USA, pp. 205–220.

Herlihy, M. P. & J. M. Wing (1990, July). Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 463–492.

Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, & D. Lewin (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, El Paso, Texas, United States, pp. 654–663. ACM.

Ladin, R., B. Liskov, L. Shrira, & S. Ghemawat (1992, November). Providing high availability using lazy replication. *ACM Transactions on Computer Systems 10*, 360–391.

Lakshman, A. & P. Malik (2010, April). Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review 44*, 35–40.

Lamport, L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 558–565.

Lamport, L. (1979, September). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*(9), 690 –691.

Lamport, L. (1998, May). The part-time parliament. *ACM Transactions on Computer Systems 16*, 133–169.

Lesniewski-Laas, C. (2008). A sybil-proof one-hop DHT. In *Proceedings of the 1st Workshop on Social Network Systems*, SocialNets '08, Glasgow, Scotland, pp. 19–24. ACM.

Lloyd, W., M. J. Freedman, M. Kaminsky, & D. G. Andersen (2011). Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, Cascais, Portugal, pp. 401–416. ACM.

Malkhi, D., M. Balakrishnan, J. D. Davis, V. Prabhakaran, & T. Wobber (2012, February). From paxos to corfu: a flash-speed shared log. *SIGOPS Operating Systems Review 46*(1), 47–51.

Malkhi, D. & M. Reiter (1997). Byzantine quorum systems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, El Paso, Texas, United States, pp. 569–578.

Petersen, K., M. J. Spreitzer, D. B. Terry, M. M. Theimer, & A. J. Demers (1997). Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, Saint Malo, France, pp. 288–301. ACM.

Schneider, F. B. (1984, May). Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems 2*, 145–154.

Schneider, F. B. (1990, December). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 299–319.

Shapiro, M. & N. M. Preguiça (2007). Designing a commutative replicated data type. *CoRR abs/0710.1784*.

Sovran, Y., R. Power, M. K. Aguilera, & J. Li (2011). Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, Cascais, Portugal, pp. 385–400. ACM.

Terrace, J. & M. J. Freedman (2009). Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, San Diego, California, pp. 11–11. USENIX Association.

Thomas, R. H. (1979, June). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems 4*, 180–209.

van Renesse, R. & F. B. Schneider (2004). Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI ' 04, San Francisco, CA, pp. 91–104. USENIX Association.

Vogels, W. (2009, January). Eventually consistent. *Communications of the ACM 52*, 40–44.