TÉCNICO LISBOA

Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication

Chathuri Lanchana Rubasinghe Gunawardhana

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Doutor Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Doutor José Carlos Alves Pereira Monteiro

Supervisor: Prof. Doutor Luís Eduardo Teixeira Rodrigues

Member of the Committee: Prof. Doutor Nuno Manuel Ribeiro Preguiça

July 2016

Acknowledgements

I am obliged to my advisor, Professor Luís Rodrigues for giving me the opportunity to work on this thesis under his supervision. His advices and fantastic ideas are the key to produce this thesis.

Moreover I would like to thank Manuel Bravo for his ideas and instructions. His experience on Riak and Erlang helped me to quickly overcome difficult problems I had during the implementation. Furthermore, I should thank Kuganesan Srijayanthan for helping with the c++ version of Eunomia by providing his master thesis work on efficient queue implementation. Finally I should thank the members of the Cluster Management team for their great support for the maintenance of the cluster during the experiments.

A special thank also to my family and friends for their support during this thesis.

Lisboa, July 2016

Chathuri Gunawardhana

For my family,

Resumo

A geo-replicação é um requisito para a maioria das aplicações que se executam na nuvem. Um problema fundamental que os sistemas geo-replicados necessitam resolver é o de como garantir que as alterações remotas são aplicadas e tornadas visíveis aos clientes numa ordem coerente. Para atíngir este objectivo, tanto os clientes como os servidores necessitam manter algum tipo de meta-dados. Infelizmente, existe um equílibrio entre a quantidade de meta-dados que um sistema necessita de manter e o nível de concorrência disponibilizado aos clientes. Dados os elevados custos de gerir grandes quantidades de meta-dados, muitos dos sistemas na prática optam por serializar algumas das alterações, utilizando um mecanismo sequenciador das actualizações, o qual, tal como iremos demonstrar, pode reduzir significativamente o rendimento do sistema. Nesta dissertação defendemos uma abordagem alternativa que consiste em permitir concorrência sem restrições no processamento de alterações locais e na utilização de um procedimento de serialização *local* deferido, que é aplicado antes das alterações serem enviadas para os centros de dados remotos. Esta estratégia permite recorrer a mecanismos de baixo custo para garantir os requisitos de coerêcia do sistema e, ao mesmo tempo, evitar efeitos intrusivos nas operações, os quais acareetam limitações no desempenho do sistema. Concretizámos e avaliámos experimentalmente a nossa abordagem. Os dados obtidos mostram que conseguimos um melhor desempenho que as abordagens baseadas em sequenciadores, com ganhos no débito de quase uma ordem de grandeza. Para além disso, ao contrário das soluções sem sequenciador propostas anteriormente, a nossa abordagem oferece uma latência na visibilidade das alterações remotas quase ótima.

Abstract

Geo-replication is a requirement of most cloud applications. A fundamental problem that geo-replicated systems need to address is how to ensure that remote updates are applied and made visible to clients in a consistent order. For that purpose, both clients and servers are required to maintain some form of metadata. Unfortunately, there is a tradeoff between the amount of metadata a system needs to maintain and the amount of concurrency offered to local clients. Given the high costs of managing large amounts of metadata, many practical systems opt to serialise some updates, using some form of sequencer, which, as we will show, may significantly reduce the throughput of the system. In this paper we advocate an alternative approach that consists in allowing full concurrency when processing local updates and using a deferred *local* serialisation procedure, before shipping updates to remote datacenters. This strategy allows to implement inexpensive mechanisms to ensure system consistency requirements while avoiding intrusive effects on update operations, a major performance limitation. We have implemented and extensively evaluated our approach. Experimental data shows that we outperform sequencer-based approaches by almost an order of magnitude in the maximum achievable throughput. Furthermore, unlike proposed sequencer-free solutions, our approach reaches nearly optimal remote update visibility latencies without limiting throughput.

Palavras Chave
Keywords

Palavras Chave

Geo-Replicação

Tolerância a Faltas

Coerência Causal

Replicação Assíncrona

Relógios Híbridos

Keywords

Geo Replication

Fault Tolerance

Causal Consistency

Asynchronous Replication

Hybrid Clocks

Contents

1 Introduction **3**

1.1 Motivation 3

1.2 Contributions 5

1.3 Results 5

1.4 Research History 6

1.5 Structure of the Document 6

2 Related Work **7**

2.1 Concepts 7

2.1.1 Key Value Stores 7

2.1.2 Geo Replication 8

2.1.3 Low Latency vs Consistency 8

2.1.4 Consistency Models 9

2.1.4.1 Linearizability 9

2.1.4.2 Sequential Consistency 9

2.1.4.3 Eventual Consistency 10

2.1.4.4 Causal Consistency 11

2.2 Dependency Tracking Mechanisms in Causally Consistent Systems 11

2.2.1 Explicitly Maintaining Metadata per Item 11

2.2.2 Dependency Matrix to Track Metadata 13

2.2.3 One Entry per Datacenter to Track Metadata 14
2.2.4 Use of Physical Clocks to Ensure Causality 14
2.3 Existing Systems which provide Causal Consistency 15
2.3.1 COPS 15
2.3.1.1 GET Operation in COPS 15
2.3.1.2 PUT Operation in COPS 16
2.3.1.3 Update Propagation in COPS 16
2.3.1.4 Benefits and Limitations 16
2.3.2 ChainReaction 17
2.3.2.1 Put Operation in ChainReaction 18
2.3.2.2 Get Operation in ChainReaction 18
2.3.2.3 Benifits and Limitations 19
2.3.3 SwiftCloud 19
2.3.4 Benefits and Limitations 19
2.3.5 Orbe 19
2.3.5.1 State Maintained by Server and Client 20
2.3.5.2 Get Operations in Orbe 20
2.3.5.3 Put Operations in Orbe 20
2.3.5.4 Update Replication 20
2.3.5.5 Read Only Transactions in Orbe 21
2.3.5.6 Benefits and Limitations 21
2.3.6 Charcoal: Causal Consistency for geo-distributed partial replication 21
2.3.6.1 Dependency Checking Approach of the Protocol 23
2.3.6.2 Benefits and Limitations 23

2.3.7 GentleRain . . . 23
2.3.7.1 GentleRain Architecture . . . 23
2.3.7.2 Get Operation in GentleRain . . . 24
2.3.7.3 Put Operation in GentleRain . . . 25
2.3.7.4 Benefits and Limitations . . . 25
2.3.8 Cure . . . 26
2.3.8.1 Benefits and Limitations . . . 26
2.3.9 Saturn: Distributed Metadata Service for Causal Consistency . . . 27
2.3.9.1 Write Operation in Saturn . . . 27
2.3.9.2 Read Operation in Saturn . . . 28
2.3.9.3 Remote Updates in Saturn . . . 28
2.3.9.4 Benefits and Limitations . . . 28
2.3.10 Comparison . . . 28
2.4 Other Related Systems . . . 29
2.4.1 Clock-RSM . . . 29
2.4.2 Logical Physical Clocks (Hybrid Clocks) . . . 30
2.4.3 Riak Key Value Store (Riak KV) . . . 30
2.4.3.1 Buckets and Bucket Types . . . 31
2.4.3.2 Riak KV Data Types . . . 31
2.4.3.3 Tunable Consistency . . . 31
2.5 Summary . . . 31
3 Eunomia **33**
3.1 *Eunomia* Into Play . . . 34
3.2 Fault-Tolerance . . . 37

3.3 Discussion . . . 38
3.3.1 Correctness . . . 38
3.3.2 Optimizing the Communication Patterns . . . 39
3.4 Geo-replication . . . 40
3.4.1 Protocol Extensions . . . 40
3.4.2 Discussion . . . 42
3.4.2.1 Vector Clocks . . . 42
3.4.2.2 Separation of Data and Metadata . . . 42
3.4.2.3 Datacenter Client Failover . . . 43
3.5 Implementation . . . 43
3.5.1 *Eunomia* implementation in c++ . . . 44
3.5.2 Geo Replication . . . 44
3.5.3 Benefits of using Logical Physical Clocks in Erlang . . . 45
4 Evaluation **47**
4.1 *Eunomia* Throughput . . . 48
4.1.1 Maximum throughput achieved in Erlang Implementation . . . 48
4.1.2 Throughput Upper-Bound with Eunomia c++ Implementation . . . 49
4.1.3 Fault-Tolerance Overhead . . . 51
4.2 Experiments with Geo-Replication . . . 52
4.2.1 Throughput . . . 53
4.2.2 Remote Update Visibility . . . 54
4.2.3 Throughput vs Remote Visibility Tradeoff . . . 56
5 Conclusions **59**
Bibliography **64**

List of Figures

2.1 Sequence of operations in a linearizable system 9

2.2 Sequence of operations in a Sequentially Consistent system 10

2.3 Overhead of Eiger's Approach in Partial Replication 17

2.4 Dependency Checking overhead in Partial Replication (from Crain and Shapiro (2015)) . 22

2.5 Get Operation at Server in GentleRain 24

2.6 Put Operation at Server in GentleRain 25

3.1 High Level Design of Geo Replicated Version built using Eunomia 40

4.1 Throughput Limit of Eunomia Implemented in Erlang. 49

4.2 Maximum throughput achieved by *Eunomia* and an implementation of a sequencer. We vary the number of partitions that propagate operations to *Eunomia*. 50

4.3 Maximum throughput achieved by a fault-tolerant version of *Eunomia* and sequencers. Non-FT denotes non fault-tolerant versions while 1-, 2-, and 3-FT denote fault-tolerant version with 3 replicas and throughput at 1st, 2nd, and 3rd replica 51

4.4 Impact of failures in *Eunomia*. 52

4.5 Throughput comparison between *EunomiaKV* and state-of-the-art sequencer-free solutions. 54

4.6 Visibility latency of remotes updates originating at DC1 measured at DC2 (40ms trip-time). 55

4.7 Visibility latency of remotes updates originating at DC2 measured at DC3 (80ms trip-time). 56

4.8 Tradeoff between throughput and remote update visibility latency posed by global stabilization checking procedures. 57

List of Tables

3.1 Notation used in the protocol description. 34

3.2 Notation used in the protocol description. 41

4.1 Round Trip Time Between Datacenters. 53

Acronyms

CRDT Conflict-Free Replicated Data Type

NTP Network Time Protocol

HTTP Hyper-Text Transfer Protocol

DHT Distributed Hash Table

FIFO First In First Out

LWW Last Writer Wins

GST Global Stable Timestamp

JSON JavaScript Object Notation

LST Local Stable Timestamp

1 Introduction

Geo Replicated Systems play a major role in distributed storage due to several reasons: to satisfy user requirements independent of his/her location; to move data closer to users; and to survive datacenter failures. For example, Facebook has its datacenters spread across different regions such as Europe, United States and Asia. Furthermore, users expect such systems to be always available and respond quickly, while preserving the consistency of their data. Unfortunately, due to the long network delays among geographically remote datacenters, synchronous replication (where an update is applied to all replicas before the operation returns to the client) is prohibitively slow for most practical purposes. Therefore, many approaches require some form of asynchronous replication strategy. A fundamental problem associated with asynchronous replication is how to ensure that updates performed in remote datacenters are applied and made visible to clients in a consistent manner. Many different consistency guarantees that allow for asynchronous replication have been defined (Bailis, Davidson, Fekete, Ghodsi, Hellerstein, and Stoica 2013). Among them, causal consistency (Ahamad, John, Kohli, and Neiger 1994) has been identified as the strongest consistency model an always-available system can implement (Attiya, Ellen, and Morrison 2015), becoming of practical relevance in geo-replicated settings.

1.1 Motivation

Today's geo-replicated datacenters need to provide low latency and always on experience to users even in the case of network partitions. Still, these systems at least need to make sure that the data they store make sense. Causal consistency is identified as the best fit to satisfy all these requirements. In fact, most weak consistency criteria require that updates are causally ordered. The need for causality can be illustrated using a simple example. Consider the following post and the replies to them:

Post by A	> *Oh god! My cat is going to die.*
After 1 second, reply by A	> *Miracles happen! He is getting better.*
After 2 seconds, reply by B	> *I love when that happens!*

However, if we do not guarantee causality, user C may observe the replies in an order that violates the original intent, namely:

$>$ *Oh god! My cat is going to die.*

$>$ *I love when that happens!*

$>$ *Miracles happen! He is getting better.*

A common solution to reduce the cost of implementing causal consistency consists in serialising all updates that are executed at a given datacenter (Belaramani, Dahlin, Gao, Nayate, Venkataramani, Yalagandula, and Zheng 2006; Almeida, Leitão, and Rodrigues 2013; Preguiça, Zawirski, Bieniusa, Duarte, Balegas, Baquero, and Shapiro 2014). In this case the state observed by a client when accessing a datacenter can be deterministically identified by the sequence number of the last update that has been applied locally. Unfortunately, serialising all updates that are applied to a given datacenter may severely limit the concurrency among independent updates. Even if different data items are stored in different physical machines, all updates are still synchronously ordered by some sequencer (operating in the critical path of local clients), typically executed by a small set of machines. Such serialisation may become a bottleneck in the system. Therefore, the implementation of efficient sequencers is, by itself, a relevant research topic. CORFU (Balakrishnan, Malkhi, Prabhakaran, Wobber, Wei, and Davis 2012) is a prominent example of an efficient sequencer implementation for datacenters.

Given the limitations above, several approaches have attempted to increase the amount of concurrency allowed at a single datacenter, at the cost of increasing the amount of metadata managed by both clients and servers. For instance, one can split the datacenter into several logical partitions and use a sequencer for each partition (Du, Elnikety, Roy, and Zwaenepoel 2013). This allows updates on different partitions to proceed uncoordinated. One can push this idea to the limit as in COPS (Lloyd, Freedman, Kaminsky, and Andersen 2011), that allows full concurrency among updates by finely tracking dependencies for each individual data item in the system. However, maintaining and processing large quantities of metadata is not without costs. Not surprisingly, the metadata problem has been extensively studied in the literature (Lloyd, Freedman, Kaminsky, and Andersen 2011; Du, Elnikety, Roy, and Zwaenepoel 2013; Almeida, Leitão, and Rodrigues 2013; Belaramani, Dahlin, Gao, Nayate, Venkataramani, Yalagandula, and Zheng 2006; Preguiça, Zawirski, Bieniusa, Duarte, Balegas, Baquero, and Shapiro 2014; Du, Iorgulescu, Roy, and Zwaenepoel 2014; Akkoorath, Tomsic, Bravo, Li, Crain, Bieniusa, Preguiça, and Shapiro 2016) (in fact, most solutions have roots in the seminal works of (Ladin, Liskov, Shrira, and Ghemawat 1992; Birman, Schiper, and Stephenson 1991)).

Recent systems such as GentleRain (Du, Iorgulescu, Roy, and Zwaenepoel 2014), and Cure (Akkoorath, Tomsic, Bravo, Li, Crain, Bieniusa, Preguiça, and Shapiro 2016) have proposed an alternative technique to circumvent the tradeoff between the metadata size and the concurrency allowed in the system. In these systems, updates are tagged with a small-sized timestamp value, that can either be a single scalar (if the metadata overhead is the major consideration) (Du, Iorgulescu, Roy, and Zwaenepoel 2014) or a vector clock (if remote update visibility is the major consideration) (Akkoorath, Tomsic, Bravo, Li, Crain, Bieniusa, Preguiça, and Shapiro 2016). When a remote update is received, it is hidden until one is sure that all updates with a smaller timestamp have been locally applied. For this, servers have to periodically run among them a *global stability checking* procedure. Interestingly, each of this stabilisation rounds generates N^2 of intra-datacenter messages and $N * M^2$ of inter-datacenter messages, where N is the number of partitions per datacenter and M is the total number of datacenters. Thus, in order to reduce the amount of messages that are exchanged among servers to avoid overloading them, one is forced to either limit the number of partitions per datacenter, throttling the concurrency inside a datacenter; or to reduce the frequency at which the stabilisation rounds occur, deteriorating the quality-of-service provided to clients.

1.2 Contributions

In this dissertation, we advocate for, implement, and evaluate a novel approach to address the metadata versus concurrency tradeoff in weakly causal consistent geo-replicated systems. The contributions of this dissertation are the following:

- The introduction of *Eunomia*, a new service for unobtrusively ordering updates, and the techniques behind it
- A fault tolerant version of *Eunomia*

1.3 Results

We have implemented our approach as a variant of the open source version of Riak KV (Riak 2011). We have augmented the system with a service that totally orders all the updates, before shipping them, that we have called *Eunomia*. The results of this dissertation can be listed as follows:

- An integration of *Eunomia* into Riak KV; which is an always-available data store
- An extention of Riak KV, to provide causally consistent geo-replication
- A sound experimental comparison of the throughput and latency of previous work against the newly introduced *Eunomia*

Our experimental results show that *Eunomia* outperforms sequencer-based systems by almost an order of magnitude while serving significantly better quality-of-service to clients compared with systems based on *global stabilisation checking* procedures.

1.4 Research History

In the beginning, the main focus of this work was to study the different existing causally consistent systems and potential bottlenecks of them. Then I implemented the *Eunomia* service on top of Riak KV, which is responsible for ordering updates before shipping to remote datacenters. I have used logical physical clocks (Kulkarni, Demirbas, Madappa, Avva, and Leone 2014) to order updates within a datacenter. *Eunomia* was implemented on top of Riak KV, which is a distributed key value storage written in Erlang. As Erlang limits the throughput that can be achieved by *Eunomia* due to the lack of memory efficient datastructures, I implemented *Eunomia* in c++. In order to compare *Eunomia* with sequencer based approaches, I also implemented a sequencer that reside on client's critical path. Then I implemented the fault tolerant version of *Eunomia*. To assess the overhead of fault tolerant sequencer compared to fault tolerant version of *Eunomia*, I have also implemented a fault tolerant sequencer which relies on chain replication. Finally, we built a geo-replicated causally consistent datastore on top of Riak KV, using *Eunomia* to order updates in the local datacenter. This project was influenced by Saturn (Bravo, Rodrigues, and Van Roy 2015), developed by Manuel Bravo, a member of the GSD team. During my work, I benefited from the fruitful collaboration with my advisor Prof. Luis Rodrigues and Manuel Bravo.

1.5 Structure of the Document

The remaining of the dissertation is organized as follows. In Chapter 2 we present all the background related to our work. Chapter 3 describes the proposed architecture and its implementation. Chapter 4 presents results from a experimental evaluation. Finally, Chapter 5 concludes the dissertation.

2 Related Work

This chapter summarizes the main concepts and systems relevant to our work. We start by describing the main concepts required to understand the rest of the document. Then we describe different causal consistency models. After that, we describe various systems which implement causal consistency using different techniques.

2.1 Concepts

First we describe the concept of a Key Value Store. Then we introduce the concept of geo-replication and the we follow to introduce the trade-off between consistency and low latency. After that, we discuss a number of relevant consistency models. Finally, we focus on causal consistency.

2.1.1 Key Value Stores

Key Value Stores (Seeger. 2009) are storage system that rely on NoSQL to avoid the bottlenecks of SQL based approaches and to provide high scalability. Even though SQL provides the support for highly dynamic queries on structured data, this support comes at a high cost, preventing SQL system to process very large amounts of information with high performance. To circumvent this limitation, Key Value Stores store the data simply as $\langle key, value \rangle$ tuples, where *value* represents an object and *key* is the identifier for that object. Two basic operations in a Key Value Store are *read* and *update*. The update operation modifies the value identified by the key, if key exists, or creates a new entry for the key otherwise. The read operation retrieves the value identified by the key, if it exists. The paradigm can be extended to support more complex operations, such as read and write transactions.

Key Value Storage scale very well compared to SQL systems (Katsov. 2009). Due to the simple data model adopted, it is easy to provide fault tolerance by adding replication. Most of the well known key value storage systems such as Cassandra (Lakshman and Malik 2010) and Riak Key Value Store

(Riak 2011) can be easily tuned to achieve either eventual consistency or strong consistency when data is replicated. We will return to this topic later in the chapter.

2.1.2 Geo Replication

Companies such as Facebook, Amazon and Twitter have their datacenters spread all over the world due to two primary goals. On one hand, their clients are spread over different geographical locations. Consequently the user communities are normally formed considering the geographical location. For example, in Facebook interactions among clients in a certain geographical location is higher than that of different geographical locations. Therefore, deploying datacenters in different regions let the clients connect to their closest datacenters, reducing the latency. On the other hand, when datacenters are deployed across different regions, even if a whole datacenter is destroyed by catastrophic failures such as flooding and earthquakes, client requests can still be served by connecting to a remote datacenter.

Geo-replicated systems can be categorised under two models: fully replicated datacenters, which store copies of the same data at each datacenter, and partially replicated datacenters which store only a fraction of data at each datacenter. Although fully replicated systems are simpler to implement, as data grows it may become infeasible. Even though new datacenters are normally added to deal with the limited capacity of existing datacenters, in a fully replicated model adding a new datacenter may require upgrading the capacity of existing datacenters as well (Bailis, Fekete, Ghodsi, Hellerstein, and Stoica 2012).

2.1.3 Low Latency vs Consistency

The main challenge of using replication is to provide consistency while satisfying the low latency requirements of users. If there is a network partition among datacenters, replicas may diverge if updates are allowed. On the other hand if strong consistency mechanisms are used, operations may need to be coordinated among remote datacenters violating the requirement of low latency. Furthermore, in a strong consistent system, we cannot ensure availability in the presence of network partitions. The CAP theorem has identified this tradeoff (Gilbert and Lynch 2002) and states that it is not possible to achieve consistency, availability and partition tolerance at the same time. Due to the strict low latency and availability requirements of many deployments, today several systems sacrifice strong consistency.

2.1.4 Consistency Models

2.1.4.1 Linearizability

Linearizability (Herlihy and Wing 1990) provide the abstraction that an operation takes effect in a single instant of time, between the invocation and completion. When a write has completed, the result of the write operation should be visible to all other read operations issued after the completion time of the write. If there are concurrent reads with the write, they can return either the state before write or the state after write. However, if a read sees the value written by a concurrent write, all the other subsequent reads should see the same or higher version of previous read. This is illustrated in Figure 2.1.

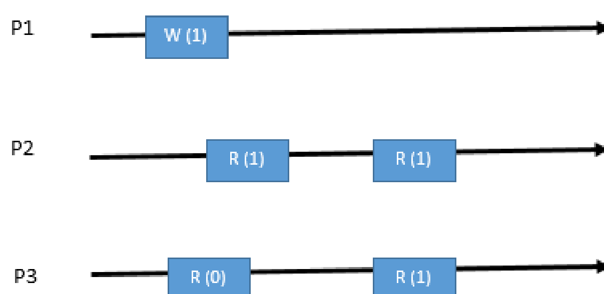


Figure 2.1: Sequence of operations in a linearizable system

According to CAP theorem it is not possible to achieve linearizability while providing low latency and partition tolerance (Gilbert and Lynch 2002). Furthermore, low latency is not compatible with linearizability, as linearizability requires applying the writes in remote datacenters before returning to the client. Moreover, it requires blocking any concurrent reads until the write is stable. Even though non-blocking algorithms have also been derived, they require readers to write back the value they read; making the reads more expensive.

2.1.4.2 Sequential Consistency

Sequential Consistency(Attiya and Welch 1994) is another form of strong consistency. It states that the execution of multiple threads is similar to execution of single thread which executes the operations of multiple threads in some sequential order, while respecting the partial order defined by each individual thread. An example is illustrated in Figure 2.2. Sequential consistency does not require writes to be immediately visible to other read operations executed by different threads, but requires subsequent reads

to see the values written by the same thread. Sequential Consistency requires operations to be totally ordered, which requires coordination among datacenter and, thus, is not compatible with low latency.

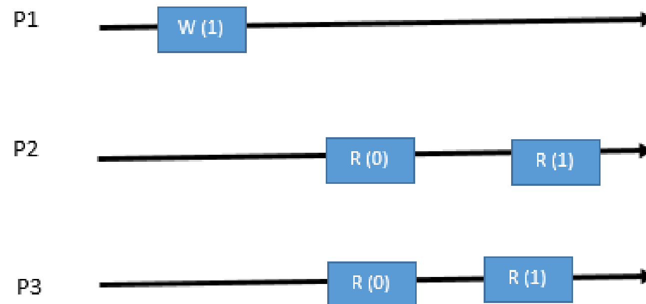


Figure 2.2: Sequence of operations in a Sequentially Consistent system

2.1.4.3 Eventual Consistency

Eventual Consistency (also called Weak Consistency) lets the operations complete with low latency by only blocking the client until the operation is applied in local datacenter. As a result, it allows different replicas to observe the update operations in different orders during a certain period. The time interval between a write to the time where every replica observes the effect of the write is called inconsistency window for that write. During this window, the system can return inconsistent results to the client. Consider again the example already provided in Chapter 1:

Post by A	> *Oh god! My cat is going to die.*
After 1 second, reply by A	> *Miracles happen! He is getting better.*
After 2 seconds, reply by B	> *I love when that happens!*

However, in an eventually consistent system, a remote datacenter may receive the reply by B, before it receives the reply by A. In that case, the order of posts and replies could appear as follows:

> *Oh god! My cat is going to die.*

> *I love when that happens!*

> *Miracles happen! He is getting better.*

This would obviously look weird to whoever sees the post and subsequent replies. As a solution to the problem above, the concept of Causal Consistency was introduced.

2.1.4.4 Causal Consistency

Causal Consistency (Ahamad, John, Kohli, and Neiger 1994) makes sure that the operations are applied while respecting the happened before partial order as defined by Lamport (1978). It provides well known semantics and avoids certain anomalies that can happen in an eventually consistent system, as mentioned before.

We denote causal order by →. For example, for two events A and B, if A→B, we say that B is causally after A or B depends on A. We say A→B if and only if one of the following three rules hold (Lamport 1978; Ahamad, John, Kohli, and Neiger 1994) :

- *Thread of Execution.* A and B are in a single thread of execution. A happens before B
- *Reads from.* A is a write operation and B is a read operation. B reads the value written by A.
- *Transitivity.* There is some other operation C where A→C and C→B

The above rules establish a causality between the operations in the same thread of execution and operations of different threads if they interact with each other.

2.2 Dependency Tracking Mechanisms in Causally Consistent Systems

In order to guarantee causal consistency, when a datacenter receives a replicated update it needs to make sure that all its causal dependencies are applied before applying the update. Therefore, each replicated update should include some kind of dependency tracking mechanism. In this section we will look at different dependency tracking mechanisms used in previous solutions.

2.2.1 Explicitly Maintaining Metadata per Item

This mechanism assumes that the datastore is fully replicated. According to this approach (Lloyd, Freedman, Kaminsky, and Andersen 2013; Lloyd, Freedman, Kaminsky, and Andersen 2011), clients maintain explicitly a dependency list, that captures their causal past. A *read* operation on an item adds the ⟨*key*, *timestamp*⟩ pair to the client's dependency list. Furthermore, the client provides its dependency list when performing *update* operations on the datastore. Remote updates, shipped from one replica to

another, also carry the dependency list associated with the update. Before applying a remote update, a replica needs to make sure that all the dependencies in the dependency list have been previously applied. Therefore, in a fully replicated model, dependency pruning can be done after each update operation. As an update operation is going to check all the dependencies in the dependency list, once it returns we can clear the dependency list and add the tuple $\langle$*key*, *timestamp*$\rangle$ returned by the new update to the dependency list. However, even with dependency cleaning, these systems can introduce storage and communication overhead and affect the throughput (Du, Elnikety, Roy, and Zwaenepoel 2013).

This approach relies on dependency cleaning to reduce the metadata overhead at the client. But in a partially replicated system, it is not obvious to apply dependency cleaning. For example, consider the two examples below:

1. The DC1 is responsible for key A, B and C. But DC2 is responsible only for key A and C.
2. The client co-located with DC1 updates value A in DC1. After the operation client updates his metadata as $(A, ts1)$.
3. The same client update B in DC1. This update carries metadata $(A, ts1)$. After this, as we clear previous dependencies with an update, client update his metadata as $(B, ts2)$
4. The same client writes C in DC1. This update carries metadata $(B, ts2)$.
5. Update to C is propagated into remote datacenter DC2, before A's update. This remote update carries metadata $(B, ts2)$

C's update carries metadata for B. But DC2 is not responsible for key B. So If C immediately applies the operation it will violate the causality as DC2 has still not seen the update to A. On the other hand, if we design the system not to apply the update immediately, but to wait for an acknowledgement from DC1 who is responsible for B and verify B's dependencies, this may require communication between datacenters (DC2 verifies dependency B and DC1 verifies dependency A). Furthermore, dependency verification may not even work in the presence of network failures (for example if DC1 fails, then there is no way to verify B and its dependencies). Because of this, in a partially replicated system it is not so easy to apply dependency verification.

1. The DC1 is responsible for key A and B. But DC2 is responsible only for key A.

2. The client co-located with DC1 updates value A in DC1. Client has metadata $(A0, ts1)$.

3. The same client update A again in DC1. This update carries metadata $(A0, ts1)$. After this, as we clear previous dependencies with an update, client has metadata $(A1, ts2)$

4. The same client writes B in DC1. After the operation, client will update his metadata as $(B, ts3)$

5. Update A0 is propagated into remote datacenter DC2, but not the A1. Then client does a Read on A at DC2 (as DC1 has failed).

Now client only has dependency B in his dependency list. But DC2 does not replicate B. Therefore DC2 does not know whether it is safe to serve the read or it should wait. But if client did not apply dependency pruning and kept all dependencies namely A1 and B, then it can avoid reading from A0 and wait until it receives A1.

With this model, updates on different objects can be performed concurrently without requiring a single serialization point. This mechanism does not introduce false dependencies as the dependencies are tracked per item. However, the overhead due to metadata can limit the scalability and throughput of the system.

2.2.2 Dependency Matrix to Track Metadata

This mechanism uses a two-dimensional dependency matrix to compactly track dependencies at client side. Each storage server maintains a vector, which identifies the updates propagated from other replicas. Replicated updates carry the client's dependency matrix. When a storage node receives a replicated update, it verifies the dependencies in the dependency matrix to ensure causality (Du, Elnikety, Roy, and Zwaenepoel 2013).

Rather than tracking dependencies explicitly per each item, this model encodes the dependencies of a storage server in a single element of the matrix. Therefore, the size of dependency matrix is bounded by the number of storage nodes in the system. Furthermore, sparse matrix encoding techniques can be used to ignore the empty elements in the dependency matrix to further reduce the size of metadata. This mechanism does not require serializing the updates issued to different partitions at a central component. It requires however serializing all the updates going to the same replica group in the same datacenter. Furthermore, when the amount of storage nodes is high, the metadata overhead can still be significant.

Moreover, applying this technique in a partially replicated system requires cordination between datacenters. Thus, in a partially replicated system this can limit the throughput and introduce remote update visibility delays.

2.2.3 One Entry per Datacenter to Track Metadata

This mechanism uses a single clock per datacenter. This was initially designed for systems where each datastore stores whole copy of data (Petersen, Spreitzer, Terry, Theimer, and Demers 1997a; Petersen, Spreitzer, Terry, Theimer, and Demers 1997b). The clients keep track of a version vector, in which each element in the vector indicates the dependencies from a given datacenter. Updates, sent to remote replicas, carry the corresponding dependency vector. When a storage node receives a remote update, it verifies its dependency vector to ensure causality.

Unfortunately, version vectors do not scale very well when partial replication needs to be supported because the dataset is too large to fit into a single server. This mechanism, also treats the whole datacenter as a single logical replica and requires a single serialization point to order updates in a given datacenter. Furthermore, this mechanism can introduce false dependencies which means that a client may be forced to wait for updates that he does not really depend upon, and that are just artifacts of the serial ordering imposed to concurrent events.

2.2.4 Use of Physical Clocks to Ensure Causality

As explained before, by explicitly tracking metadata associated with each object, updates on different objects can be performed concurrently without requiring a single serialization point; but this can introduce a large metadata overhead for both storage and transmission. On the other hand, the other two mechanisms (using one entry per datacenter or one entry per replica group in a datacenter) require some kind of serializer to order updates. As a solution to these limitations, recent work focus on using loosely synchronized physical clocks (Du, Iorgulescu, Roy, and Zwaenepoel 2014; Du, Sciascia, Elnikety, Zwaenepoel, and Pedone 2014) to causally order updates issued to the same datacenter with the notion of time.

Physical clocks can be used either with vector clocks (Akkoorath, Tomsic, Bravo, Li, Crain, Bieniusa, Preguiça, and Shapiro 2016) or a single scalar clock (Du, Iorgulescu, Roy, and Zwaenepoel 2014) to implement a geo-replicated protocol. The protocol relies on stabilization mechanisms to make remote

updates visible. These stabilization mechanisms require coordination among nodes from different datacenters; this can limit the scalability of the system. Furthermore, clock skews among storage nodes may degrade the quality of service provided to the client.

2.3 Existing Systems which provide Causal Consistency

In this section we describe the most relevant existing solutions to enforce causal consistency and their drawbacks.

2.3.1 COPS

COPS (Lloyd, Freedman, Kaminsky, and Andersen 2011) is a causally consistent key value storage which aims at running on a small number of datacenters. It assumes a fully-replicated datastore, where each individual item replica is reliable and linearizable. It supports two basic operations: *Get* and *Put*. Furthermore, it also supports *Read-Only Transactions* at the expense of more metadata and storing multiple versions. COPS can be divided into two main components: the Key Value Store and the Client Library. The COPS key value store stores tuples ⟨*key*, *value, version*⟩ in its storage. It uses consistent hashing to route messages and distribute the keys across a cluster inside each datacenter. Operations are returned to the clients as soon as they are applied on local datacenter; therefore COPS completes operations with low latency. The client library is used to track dependencies.

Each replica in COPS returns a non decreasing version of a key. Authors define this property as *Causal+* consistency (Lloyd, Freedman, Kaminsky, and Andersen 2011). Causal+ makes sure that the concurrent operations are eventually ordered in a consistent way across all datacenters (this is not provided by causal consistency). Last writer wins rule is used to resolve the conflicting write operations.

2.3.1.1 GET Operation in COPS

When a client issues a get operation, client library forwards it to the corresponding storage server. The storage server then returns the tuple ⟨*key*, *value, version*⟩. Upon receiving this tuple, the client library adds the ⟨*key*, *version*⟩ pair to it's dependency list.

2.3.1.2 PUT Operation in COPS

When a client issues a put operation, client library associates all the previous dependencies of the client with the put request and sends it to the storage. The storage node in the local cluster immediately applies the operation and returns a $\langle status, version \rangle$ tuple to the client. Upon receiving the reply to a put operation, client library clear all its previous dependencies and add the new dependency to the client's context. Then in the background, storage node propagate the update into other datacenters.

2.3.1.3 Update Propagation in COPS

The update propagation carries all the dependencies associated with a put request. When a remote storage node receives this request, it needs to make sure that all the dependencies are satisfied before committing the operation. This requires communication with the partitions who owns the dependent keys associated with the request and waiting for their acknowledgement.

Other than the above two operations, COPS supports for read only transactions, which completes in at most two rounds. However, support for this operations requires the storage nodes to maintain multiple versions of each key.

2.3.1.4 Benefits and Limitations

COPS does not require a single serialization point in each datacenter as it tracks the metadata per item. But when the read fraction is high, amount of metadata need to maintain at client library can be high and the rare writes need to carry huge amount of metadata. As in real world almost all the workloads are read dominated, carrying and verifying dependencies with writes can have a negative impact on throughput and increase the communication cost. For example, in Facebook, a user may read hundreds of posts and then he may write something on his wall before closing the page.

As we have described before, COPS approach only works in a fully replicated system. In a partially replicated system, we may need additional communication between datacenters to verify dependencies. Furthermore, client can't apply dependency pruning techniques in a partially replicated system. In order to experiment the overhead of tracking all dependencies in a partially-replicated setting, we slightly modify Eiger (Lloyd, Freedman, Kaminsky, and Andersen 2013); which uses a similar approach as in COPS, to not to clear dependencies with each write operation. Observed results are shown in Figure 2.3.

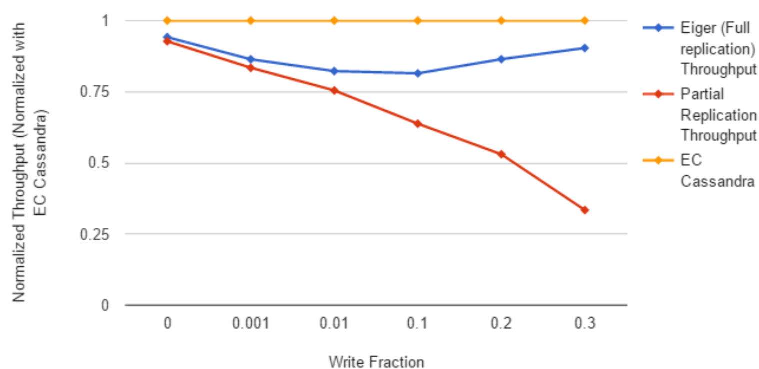


Figure 2.3: Overhead of Eiger's Approach in Partial Replication

According to the figure there is nearly a 50% loss in throughput even with a smaller ratio of updates (such as with 20% writes).

2.3.2 ChainReaction

ChainReaction (Almeida, Leitão, and Rodrigues 2013) is a variant of Chain Replication (van Renesse and Schneider 2004). In chain replication, writes are forwarded to the head of the replication chain and then propagated over the chain until they reach the tail. Reads are forwarded to the end of the tail. Therefore, chain replication provides a linearized execution by serializing reads and writes at the tail. However this approach can limit the throughput and does not leverage the existing other replicas to distribute the read workload.

In ChainReaction, if a client has observed the value returned by node x for a given object as a result of a read or a write operation, any node between head and x can return causally consistent values for the subsequent reads. Furthermore, this constraint can be removed once the ongoing write operation of a client is propagated to the tail, as a consistent state can be observed by reading from any server in the chain. When a write operation is applied into the whole chain, they call it DC-Write-Stable.

ChainReaction has three modules: the Client Library, the Client Proxy and the Storage Servers. The Client Library keeps track of metadata for each accessed object. It maintains the metadata in a table in the form of $\langle$ *key*, *version*, *chainIndex*$\rangle$. Each read adds a dependency to this metadata. Before applying a write client proxy makes sure that all its previous reads has become DC-Write-stable. The client proxy receives the request from client library and redirect the request to an appropriate Storage Server whose index is between head and the chain index. The storage nodes keep the objects distributed over the cluster

using consistent hashing.

2.3.2.1 Put Operation in ChainReaction

When a client issues a PUT operation, the client library tags this with the metadata related to the last put operation and metadata for get operations that happened after the last put. The metadata is only maintained for the objects whose version of writes is not yet DC-Write-Stable. When the storage proxy receives the PUT request, it has to make sure that all the dependencies associated with the operations have become DC-Write-Stable before performing the operation. Once this property is satisfied, it will forward the request to the head of the node. The head will assign a new version, apply the operation and propagate the update to the next k elements in the chain. Then the most recent version of the object and the chain index representing the kth node will be forwarded to the client. Finally, the client will update its corresponding entry in the table. Updates sent to the rest of the storage replicas in the chain will happen in the background. Once the write become stable, the metadata entry for the object with that version will be removed from the table maintained at client library.

2.3.2.2 Get Operation in ChainReaction

When the client library receives a get request, it attaches the version and chain index parameter with request. Then it is forwarded to the proxy. The proxy then uses the chain index to decide a storage server to forward the request. It can forward the request to any node where its node id is between head and chain index. This approach allow ChainReaction to distribute the load for reads. The selected server t then process the request and return value and version to the client library. The client library then updates its metadata as follows: if the newer version is same as previous version, chain index is set to *max(chain index,t)* or else if the newer version is greater, then the chain index is set to t.

ChainReacion supports Read only Transactions as well. Furthermore the system has been extended to support Geo-Replication. In this extension, the version of a data item is identified by a version vector. Instead of single chain index, a chain index vector is used to estimate how far the current version has been propagated across chains in each datacenter.

2.3.2.3 Benifits and Limitations

ChainReaction can distribute the load for concurrent read operations. The Client Library needs to maintain metadata for each key and PUT operations need to carry dependencies associated with the read operations that happened after the previous PUT. Furthermore, as ChainReaction was developed to improve the performance of read operations, write operations are delayed until the version of any object of which the current write depends on has become DC-Write-Stable. Moreover, writes in a given hash range are serialized as they are handled by the head of the node for that hash range.

2.3.3 SwiftCloud

Swiftcloud (Preguiça, Zawirski, Bieniusa, Duarte, Balegas, Baquero, and Shapiro 2014) is a causally consistent data storage implemented using a log based update propagation approach and it supports partial replication while maintaining bounded metadata. The metadata includes a sequence number which uniquely idenitfies the update and set of datacenter assigned timestamps. Therefore, the metadata grows with the number of datacenters. The storage nodes maintain a vector which represent the causal history of updates it has seen from remote datacenters so far. Each update record also stores the version vector which summarise the client's dependencies. SwiftCloud relies on CRDTs which includes Last Writer Wins(LWW) semantics and Multi Value Registers(MVR) for conflict resolution.

2.3.4 Benefits and Limitations

Eventhough SwiftCloud was able to maintain bounded metadata, it requires a sequencer to order updates issued in the same datacenter. This limits the throughput of the whole system.

2.3.5 Orbe

Orbe (Du, Elnikety, Roy, and Zwaenepoel 2013) is a partitioned, geo-replicated, and causally consistent data store. It uses a 2-dimensional dependency matrix to track dependencies at the client side. Dependency matrix is an extension of version vectors to support both replication and partitioning. Each storage server maintains a vector clock to keep track of updates applied in different replicas. It uses a single entry per partition; therefore, any two dependencies that belongs to the same partition will be represented as a single scalar. Orbe supports three operations: Get, Put, and Read-Only Transactions.

2.3.5.1 State Maintained by Server and Client

Clients in Orbe maintain a dependency matrix, DM. $DM[n][m]$ indicates that the client session depends on $DM[n][m]$ updates at the mth replica of nth partition. Each storage node maintains a version vector, VV. $VV[i]$ indicates that the server has applied first $VV[i]$ updates from i^{th} replica of the same partition. Consistent hashing can be used to distribute objects across partitions, and to locate corresponding server for a key. DHT based routing can be used to route the message to a given server.

2.3.5.2 Get Operations in Orbe

The client sends a request for a given key to the corresponding partition n. Storage server then retrieve the $\langle$*value*,*timestamp*$\rangle$ pair for the key from the storage and return the $\langle$ *value*, *timestamp*, *replica_id* $\rangle$ to the client. The client then updates his dependency matrix DM to $max(DM[n][replica_id], timestamp)$. In this way, dependency matrix track all the dependencies associated with read operations.

2.3.5.3 Put Operations in Orbe

The client sends a put request with $\langle$*key*, *value*, $DM\rangle$ to the nth partition at local datacenter with replica id *m*. Once the partition receives the request, it increments its own clock position, $VV[m]$. Then it creates a new version v$\leftarrow$ $(key, value, VV[m], DM)$ for item d identified by the key. It stores v in the stable storage. It assigns $ut \leftarrow VV[m]$, *rid*$\leftarrow$ $m, dm \leftarrow DM$ and returns (ut, rid) to the client. In the background it propagates the updates with $\langle$*key, value, ut, dm*, *rid*$\rangle$ to the other replicas. When the client receives this tuple, it updates the dependency matrix as follows: $DM \leftarrow 0, DM[n][rid] \leftarrow ut$. As the put operation is going to verify all the dependencies in remote replicas, by updating DM to keep the dependency for the put, we can remove all the previous dependencies without violating causality.

2.3.5.4 Update Replication

When a remote replica *m* of partition *n* receives $\langle$*key*, *value,* v,*ut*, *dm*, *rid*$\rangle$, first it checks whether it has seen all the dependencies specified by the client, by checking $VV \geq dm[n]$ for all the elements in the vector. Once it is satisfied, it checks whether causality is satisfied in other local partitions. For any partition $dm[i]$ with non zero elements, it sends a dependency check message to i^{th} partition, which will

then block until VV at $i \geq dm[i]$. Once all these conditions are satisfied new version will be created in the storage. Finally, it will update it's VV as $VV[rid] \leftarrow ut$ to reflect the newly received update.

2.3.5.5 Read Only Transactions in Orbe

Orbe has another protocol to support Read-Only Transactions. It assumes the existence of monotonically increasing, loosely synchronized physical clocks. When compared to previous systems, it has few more metadata. Each version of an item stores the timestamp of the source replica at the item creation time. The client stores the highest physical update timestamp it has seen from all the items. Each partition maintains a physical version vector with an element for each replica, which indicates latest physical timestamp it has seen. Orbe is the first causal consistency system which uses the physical clocks.

2.3.5.6 Benefits and Limitations

Orbe encodes dependencies that belong to the same partition in a single scalar, which can create false dependencies between operations and needs to serialize updates issued to the same partition. This can limit the the throughput and level of concurrency in the system. The use of one clock entry per partition avoids the requirement for a single serialization point to order the updates that go to different partitions. Furthermore, it uses a sparse matrix encoding to keep the size of the matrix small; as a result, it uses less metadata compared to COPS but still there is a considerable overhead at storage and transmission as each write needs to carry the dependency matrix and remote replicas need to verify dependencies before applying propagated updates. Orbe proposes a dependency cleaning mechanism where the client does not need to track the dependencies of a read, if it is propagated to all the replicas. But this requires communication between replicas and dependency cleaning is not obvious due to failures and network partitions.

2.3.6 Charcoal: Causal Consistency for geo-distributed partial replication

All the geo-replicated systems presented before assume a fully-replicated datastore. These systems are difficult to adapt to operate in a geo-distributed system supporting partial replication. Furthermore, ensuring causality in a partially-replicated system is not trivial (Bailis, Fekete, Ghodsi, Hellerstein, and Stoica 2012). But partial replication is essential to avoid unnecessary usage of networking and hardware.

In Charcoal (Crain and Shapiro 2015), the authors describe a protocol that enforce causality in a partially replicated system by relying on loosely synchronised physical clocks, heartbeats and FIFO links.

For dependency tracking, Charcoal uses one clock per datacenter. Each update is tracked with a unique timestamp given by the local datacenter and the vector clock describing client's dependencies. Clients uses a vector clock to track dependencies it has seen from each datacenter. Charcoal ensures that the client reads a value containing all the dependencies in his vector. In partial replication, a datacenter may need to forward the request if it does not replicate the value requested by the client.

Most of the existing systems ensure causality by making update visible only when all the dependencies are satisfied. This is straightforward in a fully replicated system as the receiving node can verify dependencies by checking them in corresponding partitions. But in a partially replicated system, some objects may not be replicated in the receiving datacenter. Therefore a remote datacenter needs to communicate with the originating datacenter which owns that data item to verify the dependencies. This can introduce additional overhead and dependency check messages. A scenario, extracted from (Crain and Shapiro 2015), with 2 datacenters, where A1, A2 belongs to DC1, and B1,B2 belongs to DC2 is shown in Figure 2.4. Note that the dependency checking from B1 in DC2 to A1 in DC1 is an extra message overhead introduced due to partial replication.

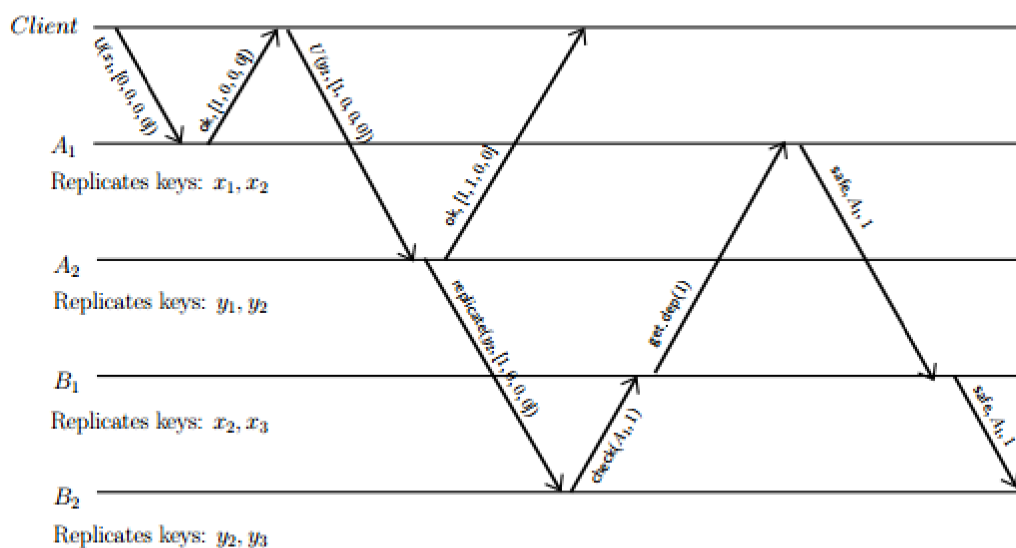


Figure 2.4: Dependency Checking overhead in Partial Replication (from Crain and Shapiro (2015))

2.3.6.1 Dependency Checking Approach of the Protocol

Rather than performing dependency checking at remote replicas, in Charcoal the origin datacenter notifies the remote datacenter when it is safe to make the remote update visible. For this, a coordinator server at origin datacenter performs heartbeats with its servers. Servers reply with the timestamp of the latest remote update it has sent. When the coordinator is sure that all the servers have sent remote updates before time t, it will notify other datacenters to make any updates visible with timestamp before t.

2.3.6.2 Benefits and Limitations

Charcoal avoids the dependency checking overhead and metadata traversal between datacenters, but this can delay the visibility of remote updates as now the remote datacenters need to wait until local datacenter informs the safe point to apply updates. This delay can be further increased by the clock skew as the updates should be delayed by the clock skew. Furthermore, the protocol heavily relies on correctness of the clocks. The protocol may not even provide the correctness if the clock skew is significant and due to the non monotonic behavior of the clocks. Moreover, their protocol still require serializing all the updates issued to a single datacenter as they use a single scalar per datacenter to order operation issued to a single datacenter.

2.3.7 GentleRain

GentleRain (Du, Iorgulescu, Roy, and Zwaenepoel 2014) tracks causality by attaching a timestamp to updates derived from loosely synchronized physical clocks. This work was motivated by the protocol used for Read-Only Transactions in Orbe (Du, Elnikety, Roy, and Zwaenepoel 2013).

2.3.7.1 GentleRain Architecture

The system is partitioned into N partitions and each partition is replicated in M replicas in M different datacenters. They assume the fully replicated model and a multi-version datastore. Each version stores a small amount of metadata (physical clock at creation time and source node). Old versions are garbage collected periodically. As Gentlerain uses physical clock to track dependencies within a datacenter, storage nodes should be equipped with monotonic clocks. Furthermore, clock skew should be very small to avoid memory overhead on storage nodes.

Each server maintains a version vector with M elements, which keeps track of the updates seen by the other replicas in the same replication group. They define the *Local Stable Timestamp* (LST) as $min(versionvector)$. This indicates that the server has received all the updates below LST from all the replicas. The authors also define the *Global Stable Timestamp* (GST) for each server. It keeps track of the minimum LST across all partitions in the datacenter. This value is calculated periodically. Rather than broadcasting LST values to compute GST, GentleRain uses a tree based protocol to calculate GST, which has a less message complexity. If a partition does not receive any update, GST does not change. Therefore, to ensure the progress, heartbeats are sent periodically with the latest clock. Once a heartbeat is received from a node, corresponding clock value for the server in the version vector will be updated.

The clients in GentleRain maintain the maximum update timestamp that they have seen so far and the latest GST they are aware about.

GentreRain supports 4 operations: Get, Put, GetSnapshot and Read-Only transactions. We will describe GentleRains's approach and algorithms for GET and PUT operations below.

2.3.7.2 Get Operation in GentleRain

The client sends a request to the partition with the key and GST it knows about. The Server then updates its GST if the one it knows is smaller than the one which is received from the client. The server then get the latest version; which is either created by itself, or the one with an update timestamp no greater than server's GST. This property is used to make sure the causality between reads. The server then returns the value, timestamp of the value and its GST back to the client. The client then updates the value of maximum update timestamp and GST to reflect the latest values. The algorithm runs at the mth replica of nth partition is shown in Figure 2.5.

upon receive $\langle \text{GETREQ}\ k, gst \rangle$
    $GST_n^m \leftarrow max(GST_n^m, gst)$
    obtain latest version d from version chain of key k
s.t. $d.sr = m$ or $d.ut \leq GST_n^m$
    send $\langle \text{GETREPLY}\ d.v, d.ut, GST_n^m \rangle$ to client

Figure 2.5: Get Operation at Server in GentleRain

2.3.7.3 Put Operation in GentleRain

The client sends a put request with the maximum timestamp it has seen. If the clock of the partition which receives the request is smaller than client's timestamp, it will be blocked until the server's timestamp is greater than received timestamp from client. This blocking can happen if the clock skew is significant or due to the non monotonic clocks. This is necessary to make sure that the put operation is after the client's previous read or put operations. Once the condition is satisfied server will update its position at version vector with its physical timestamp and will create a new version. Server then will reply to the client with the update timestamp of the newly created version. Client will update his maximum update timestamp to the one received from the server. Once a server executes a local update, update will be replicated asynchronously to other replicas. When a server receives a remote update, it will be inserted to the version chain. But the update will not be visible till the update timestamp is smaller than GST. Algorithm for local put and remote propagation is shown in Figure 2.6.

upon receive $\langle \text{PUTREQ}\ k, v, dt \rangle$
  wait until $dt < Clock_n^m$
  update version vector: $VV_n^m[m] \leftarrow Clock_n^m$
  create new item d
  set key: $d.k \leftarrow k$
  set value: $d.v \leftarrow v$
  set update time: $d.ut \leftarrow VV_n^m[m]$
  set source replica: $d.sr \leftarrow m$
  insert d to version chain of key k
  send $\langle \text{PUTREPLY}\ d.ut \rangle$ to client

upon new version d created
  for each server p_n^k, $k \in \{0..M-1\}, k \neq m$ **do**
    send $\langle \text{REPLICATE}\ d \rangle$ to p_n^k

upon receive $\langle \text{REPLICATE}\ d \rangle$ from p_n^k
  insert d to version chain of key $d.k$
  $VV_n^m[k] \leftarrow d.ut$

Figure 2.6: Put Operation at Server in GentleRain

2.3.7.4 Benefits and Limitations

The use of loosely synchronised physical clocks helps GentleRain to track dependencies while avoiding the overhead of storage and transmission of metadata which is a bottleneck in most of the existing solutions (Lloyd, Freedman, Kaminsky, and Andersen 2011; Lloyd, Freedman, Kaminsky, and

Andersen 2013; Du, Elnikety, Roy, and Zwaenepoel 2013; Almeida, Leitão, and Rodrigues 2013). GentleRain is the first causally consistent system to avoid dependency checking, which authors claims as the major overhead in existing solutions. It is capable of tracking the causality with less metadata and avoids the requirement for serializing the writes of a single datacenter. Therefore, it achieves throughput very similar to eventual consistency. The existence of skews among clocks does not violate causality, but can delay the visibility of remote updates and block the put operations at the source replica for a small time. GentleRain relies on monotonically increasing clocks and clock synchronization protocols such as NTP to keep the clocks loosely synchronized. The approach used by GentleRain achieves high throughput by delaying the remote update visibility. As it uses a single scalar to track causal dependencies, the remote update delay can be in the order of the trip time to the furthest datacenter.

2.3.8 Cure

Concepts behind Cure (Akkoorath, Tomsic, Bravo, Li, Crain, Bieniusa, Preguiça, and Shapiro 2016) is similar to Gentlerain. However, instead of single scalar to track dependencies, it uses a dependency vector with one entry per datacenter. Each entry in the vector represent a physical timestamp and it encodes all the causal dependencies up to that time. Therefore, Cure requires monotonically increasing timestamps in order for the algorithm to work correctly. It also uses NTP to loosely synchronise the clocks of the storage servers. Cure provides causal+ consistency: ensure causality and make sure replicas converge in the same order in the case of concurrent conflicting updates. It relies on CRDTs (Conflict-free Replicated Data Types) (Shapiro, Preguiça, Baquero, and Zawirski 2011) to provide this guarantee. It manages multiple versions in order to server requests from causally consistent snapshot. Old versions are garbage collected periodically.

Similar to Gentlerain, updates originating at the local datacenter are immediately visible to the client. But remote updates only become visible when their GST vector $\geqslant$ commit timestamp vector of the received update. Instead of single scalar(GST) in Gentlerain, now the whole vector is compared.

2.3.8.1 Benefits and Limitations

Maintaining one entry per datacenter reduces the overhead of fault dependencies but increases the amount of metadata to manage, which increases the communication and storage overhead compared to Gentlerain. However both Gentlerain and Cure can limit the amount of metadata without relying on

a sequencer; but as both approaches relies on physical clocks to order updates, it requires additional communications to make the update visible in remote datacenters.

2.3.9 Saturn: Distributed Metadata Service for Causal Consistency

The focus of Saturn (Bravo, Rodrigues, and Van Roy 2015) is to provide a higher level of concurrency while maintaining the amount of metadata smaller, even for challenging environments such as partially-replicated geo-distributed systems. Saturn assumes a geo-replicated system that supports partical replication and that consists of several data-nodes. A single data-node stores a fraction of data. A client contacts his preferred data-node and if it requests an item which is not available in the local data-node, the request will be forwarded to another data-node. Saturn has four main features:

- It uses a small amount of metadata to ensure causality
- It uses two separate channels for metadata and data, and introduce delays for metadata traversal to make sure that the metadata and data will be available in a storage node at the same time
- It uses a tree based topology to propagate metadata to storage replicas. Links between nodes in the tree are implemented as FIFO. Actual content will be transferred directly to the replicas
- It avoids the negative impact of false dependencies by delaying visibility of updates on less frequently accessed objects

Updates in Saturn are identified by a label. When an update request is received from client, the datanode adds a tuple $\langle object_{id}, timestamp \rangle$ to the Saturn service. Saturn then output these tuples at every other data nodes in an order that satisfies the causality. In Saturn system, only metadata is carred in the labels. Saturn provides 2 basic operations: Writes and Reads.

2.3.9.1 Write Operation in Saturn

When a datanode receives an update, it applies update locally and then reply to the client. Then it adds the label to the Saturn Metadata Service. The actual update is tagged with the label will be sent to the other datanodes which replicate the object. Saturn delays labels of less-frequently accessed objects to make sure that the updates of frequently accessed objects are visible quickly; therefore, only a few clients are stalled. The tree topology used for metadata traversal provides several positions to introduce

the delay for label propagation. Furthermore, it tracks access patterns of the objects and introduces delays accordingly.

2.3.9.2 Read Operation in Saturn

When a datanode receives a request from client, if it replicates the requested object, the most recent value will be provided. Otherwise it will issue a remote read request. Remote node will only reply to the client if causality is not violated. Saturn will indicate to remote node the safe time to return the value to the client.

2.3.9.3 Remote Updates in Saturn

A datanode applies a remote update when it has received the label and all the payloads associated with the previously delivered labels have been applied.

2.3.9.4 Benefits and Limitations

Saturn reduces the metadata overhead and dependency checking even with the challenging setting like partial replication while still achieving high concurrency. But still, the Saturn service for metadata management may be a bottleneck as it needs to heartbeat with all the servers in the datacenter to make sure that it does not delay the visibility of updates blindly in the case of failures or due to servers who do not receive any updates from clients.

2.3.10 Comparison

Other than GentleRain, Cure and Saturn, all the causally consistent systems discussed above have a tradeoff between the metadata size and concurrency. SwiftCloud maintains bounded metadata but limit the concurrency by serializing all the updates issued in a single datacenter. On the other hand, COPS and Eiger allow concurrent updates without serializing them, but require maintaining metadata per each object. Orbe only partially solves this issue by aggregating updates issued to the same datacenter into a single scalar, but still it has to maintain a dependency matrix to track dependencies. ChainReaction need to maintain metadata per each key, but it provides mechanisms to clear metadata for items with versions that has been replicated in all the replicas. Futhermore, as it uses a bloom filter to causally order remote

updates issued from a given datacenter, it may introduce false dependencies and increase the remote visibility delay.

GentleRain achieves higher throughput by not serializing write operations and only uses the loosely synchronized physical clock value to track dependencies. But as it relies on receiving heartbeats from all the remote replicas to update the GST, it can delay the visibility of remote updates in the order of trip time to the furthest datacenter. Cure avoid this problem by maintaining a single entry per datacenter to track dependencies, but still the remote update receiver has to wait until it receives the heartbeat from all the partitions from the originating datacenter (to update the GST). Furthermore, there are two bottlenecks which can limit the scalability and throughput of both Gentlerain and Cure: exchanging heartbeats between replicas, and periodically exchanging the min replica timestamp (in Gentlerain) or version vector representing the replica timestamp (as in Cure) with all the nodes in the same datacenter. However compared to the previous systems which can either limit concurrency by adding a sequencer or increase the metadata overhead, both Gentlerain and Cure plays a nice tradeoff between throughput and metadata overhead. Similarly, Saturn achieves high throughput and only uses a label per update as metadata but it can provide stale data for less frequent objects. Furthermore, the metadata service in Saturn may be a bottleneck as it has to heartbeat with all the other servers in the datacenter.

2.4 Other Related Systems

2.4.1 Clock-RSM

Clock-RSM (Du, Sciascia, Elnikety, Zwaenepoel, and Pedone 2014) provides total order over executing set of commands among a replica group. It reduces the communication and latency overhead to achieve total order by using loosely synchronised physical clocks. The stabilization procedure in our work Eunomia is inspired by this work.

Replicas exchange heartbeats periodically and maintain a vector indicating the latest timestamp it has seen from other replicas. Updates initiated by a replica are marked with incrementing order. Therefore Clock-RSM requires monotonically increasing clocks. Before committing a command, replicas wait for a majority. Then each replica checks whether the minimum value in the heartbeat vector is greater than its own to make sure that it will not mark any future updates with a timestamp smaller than the committed updates. Then, the command is committed only if it is the command with the smallest times-

tamp. This property make sure the total ordering as all replicas will commit the commands in physical timestamp order marked in the commands.

2.4.2 Logical Physical Clocks (Hybrid Clocks)

The concept of Logical Physical Clock or Hybrid clock has been introduced recently (Kulkarni, Demirbas, Madappa, Avva, and Leone 2014). Logical clocks help to capture the causality of events but it does not capture the actual physical time of the events. On the other hand, physical clocks capture the true notion of time about when an event happens, but they cannot be used safely to order events due to the effect of the clock skew and non-monotonic behaviors. Combination of these two clocks capture the best of both worlds: they allow to tracks causality of events while still providing the true notion of time. This combination has been named Hybrid Logical Clocks (HLC). The goal of HLC is to provide causality detection while maintaining the clock value to be always close to physical clock. The protocol can tolerate common NTP problems such as non monotonicity and clock skews. The logical part of the HLC can make progress masking the errors in the physical clock. However, there are still very few examples of systems using hybrid clocks and, to the best of our knowledge, the only distributed datastore which uses this approach is CockroachDB (CockroachDB 2016).

2.4.3 Riak Key Value Store (Riak KV)

In this section we will describe the architecture and main design goals of the Riak (Riak 2011) key value store as we are going to use it for our prototype implementation.

Riak KV is a Dynamo-inspired key value store with additional features such as map-reduce, search, pluggable backends, HTTP and binary interfaces. As it has been inspired by Dynamo, it was build considering availability and low latency as important aspects (rather than consistency). Therefore, it relies on eventual consistency by default. However it provides us tunable consistency with the support of quorums. But using strong consistency can contradict with initial goals of Riak KV as strong consistency requires coordination and may not provide availability in the presence of network partitions. Riak KV is written in Erlang and C/C++.

Riak KV uses consistent hashing to partition the data across the cluster and a DHT based routing to locate a node which stores a specific key. It has two different storage backends: unordered and ordered storage. Unordered storage layer uses Bitcask and it keeps an index to retrieve the keys. Ordered storage

layer allows for efficient range queries and it uses the LevelDB by Google. Multiple keys are grouped together into a logical namespace called a Bucket. Configurations such as replication, quorum settings and conflict resolution policies can be configured bucket wise. It uses the gossip protocol to maintain the membership information.

2.4.3.1 Buckets and Bucket Types

Buckets in Riak KV provide logical namespaces so that identical keys in different buckets will not conflict. Therefore, a unique key in Riak KV is identified by Bucket+Key. They have other benefits such as defining properties per bucket basis. This can be compared to tables in relational databases.

2.4.3.2 Riak KV Data Types

Other than typical key value pairs stored in a Riak KV Cluster, it also provides five different convergent replicated data types (CRDTs). Riak KV handles conflict resolution on behalf of developers for these data types. These five data types are: Flags, Maps, Registers, Counters and Sets.

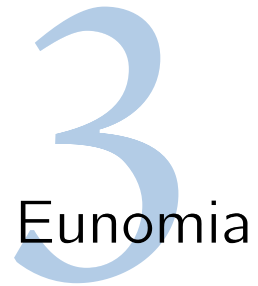
2.4.3.3 Tunable Consistency

The default Riak KV configurations assumes an eventually consistent data store. With this, write operations are considered successful as soon as they are applied on a single node. Then, the updates to the other nodes will be propagated using anti-entropy mechanisms and read-repairs (Basho 2011a). However, Riak KV can be tuned either to achieve consistency or availability. It defines n_value (number of replicas per item), r (number of replicas to contact per read) and w (number of replicas to contact per write) in its configuration to achieve this.

2.5 Summary

In this chapter we explained the related work which we consider important for our work. We started by introducing the basic concepts that is required to understand the rest of the report: the concept of key value store, concept of Geo Replication and the tradeoff between low latency and consistency. Then we discussed different consistency models used in distributed storage systems. We have started with

the strongest and most expensive consistency levels: linearizability and sequential consistency. Then we introduced the concept of eventual consistency, which is the weakest consistency. Then we explained the causal consistency model, which lies between strong and eventual consistency. After that, we have surveyed the most relevant dependency tracking mechanisms available in causally consistent systems: using one clock per datacenter, using dependency matrix based tracking, using one clock per partition, and using physical clocks. Finally we presented and compared existing causally consistent systems. We also described the concept of hybrid clock. In the next chapter will introduce the algorithms and implementation details of our system.



Eunomia

Sequencer is a powerful building block for distributed applications. They can significantly simplify the reasoning about the passage of time and the cause-effects relations among events. Particularly, sequencers become a key piece in the design of weakly consistent geo-replicated data stores, a fundamental piece for modern internet-based services. Examples of solutions that subsume the use of some form of a sequencer are (Petersen, Spreitzer, Terry, Theimer, and Demers 1997b; Belaramani, Dahlin, Gao, Nayate, Venkataramani, Yalagandula, and Zheng 2006; Preguiça, Zawirski, Bieniusa, Duarte, Balegas, Baquero, and Shapiro 2014; Almeida, Leitão, and Rodrigues 2013). In this particular context, sequencers are used to abstract the internal complexity of datacenters (composed by a large number of servers), aggregating metadata and simplifying the application of remote updates (updates coming from remote datacenters). Unfortunately, the interaction with sequencers usually falls into the critical path of client operations. This limits the system concurrency to the load of a single machine (the sequencer) can handle and increases operation latencies.

In this chapter, we propose *Eunomia*, a new service conceived to replace sequencers in these settings. *Eunomia* aims at simplifying the design of weakly consistent geo-replicated data stores without compromising intra-datacenter concurrency. Unlike traditional sequencers, *Eunomia* lets local client operations to execute without synchronous coordination, an essential characteristic to avoid limiting concurrency and increasing operation latencies. Then, in the background, *Eunomia* establishes a serialization of all updates occurring in the local datacenter in an order consistent with causality, based on timestamps generated locally by the individual servers that compose the datacenter. We refer to this process as *site stabilization procedure*. Thus, *Eunomia* is capable of abstracting the internal complexity of a multi-server datacenter without limiting the concurrency. Therefore, *Eunomia* can be used to improve any existing sequencer-based solution to enforce causal consistency across geo-locations (Petersen, Spreitzer, Terry, Theimer, and Demers 1997b; Preguiça, Zawirski, Bieniusa, Duarte, Balegas, Baquero, and Shapiro 2014; Almeida, Leitão, and Rodrigues 2013), as shown later in this chapter.

N	Number of partitions
Clock$_c$	Client c clock
p_n	Partition n
Clock$_n$	Current physical time at p_n
Ops	Set of unstable operations at *Eunomia*
PartitionTime	Vector with an entry per partition at *Eunomia*
$u_j.ts$	Timestamp assigned to update u_j

Table 3.1: Notation used in the protocol description.

3.1 *Eunomia* Into Play

In order to better illustrate how *Eunomia* works, we now present a detailed protocol of the interaction between *Eunomia* and nodes that constitute a datacenter. In the following exposition, we assume that updates to individual items are serialized by the native update protocol. In fact, we generalize this assumption for the case where multiple items are managed by a given set of servers that serialize all updates of those items. We call each of these set of servers a *partition*. We assume FIFO links among partitions and *Eunomia*. Table 3.1 provides a summary of the notation used in our protocols.

Eunomia assumes that each individual partition can assign a timestamp to each update without engaging in synchronous coordination with other partitions, or with *Eunomia* servers. We will explain below how this can be easily achieved. These timestamps have to satisfy two properties.

Property 1 *If an update u_j causally depends on a second update u_i, then the timestamp assigned to u_j ($u_j.ts$) is strictly greater than $u_i.ts$.*

Property 2 *For two updates u_i and u_j received by Eunomia from partition p_n, if u_i is received before u_j then $u_j.ts$ is strictly greater than $u_i.ts$.*

These two properties imply that updates are causally ordered across all partitions and that once *Eunomia* receives an update coming from a partition p_n, no update with smaller timestamp will be ever received from p_n.

In order to provide the above properties, clients play a fundamental role. A client c maintains during its session the largest clock seen (*Clock*$_c$). This clock aggregates its causal dependencies and it is included in the client's update requests. As described below, partitions compute update timestamps considering client clocks, which is key for the protocol's correctness.

Algorithm 1 Operations at client c

```
1: function READ(Key)
2:     send READ(Key) to server
3:     receive ⟨Value, Ts⟩ from server
4:     Clock_c ← MAX(Clock_c, Ts)
5:     return Value

6: function UPDATE(Key, Value)
7:     send UPDATE(Key, Value, Clock_c) to server
8:     receive Ts from server
9:     Clock_c ← Ts
10:    return ok
```

The protocol assumes that each partition p_n is equipped with a physical clock. Clocks are loosely synchronized by a time synchronization protocol such as NTP (NTP 2008). The correctness of the protocol does not depend on the clock synchronization precision—namely on the clock drift. However, as discussed later, large clock drifts may have some negative impact on the protocol's performance (in particular, on how fast the datacenter can ship updates to other remote datacenters). Furthermore, if we only use physical clock to track dependencies, we need to make sure that the clock behaves monotonically; which is expensive to achieve. To circumvent this limitation, our protocol uses hybrid clocks (Kulkarni, Demirbas, Madappa, Avva, and Leone 2014), which have been shown to overcome the limitations of simply using physical time. The use of physical time is fundamental for the efficiency of the *site stabilization procedure* performed by *Eunomia*.

We now proceed to describe how events are handled by clients, servers and *Eunomia* (Algorithms 1, 2, and 3 respectively).

Read A client c sends a read request operation on a data item (identified by *Key*) to the server(s) that host the partition (p_n) responsible for *Key* (Alg. 1, line 2). When p_n receives the request, it fetches the *Value* and the timestamp *Ts* that is locally stored for *Key* and returns both to the client. *Ts* is the clock used when the update operation was issued. After receiving the pair $\langle Value, Ts \rangle$, the client computes the maximum between $Clock_c$ and *Ts* (Alg. 1, line 4) to ensure that the read operation is included in its causal history.

Update A client c sends an update request operation to the server(s) hosting the responsible partition p_n for the object being updated. Apart from the *Key* and *Value*, the request includes client's clock $Clock_c$ (Alg. 1, line 7). When p_n receives the request, it first computes the timestamp of the new update (Alg. 2, line 5). This is computed by taking the maximum between $Clock_n$ (physical time), the maximum

Algorithm 2 Operations at partition p_n

```
1: function READ(Key)
2:     ⟨Value, Ts⟩ ← KV_GET(Key)
3:     return ⟨Value, Ts⟩

4: function UPDATE(Key, Value, Clock_c)
5:     MaxTs_n ← MAX(Clock_n, Clock_c + 1, MaxTs_n + 1)
6:     KV_PUT(Key, ⟨Value, MaxTs_n⟩)
7:     Op ← ⟨ Key, Value, MaxTs_n, p_n⟩
8:     send ADD_OP(Op) to Eunomia
9:     return MaxTs_n

10: function HEARTBEAT                                  ▷ Every Δ time
11:     if Clock_n ≥ MaxTs_n + Δ then
12:         MaxTs_n ← MAX(Clock_n, MaxTs_n + 1)
13:         send HEARTBEAT(p_n, MaxTs_n) to Eunomia
```

timestamp ever used by p_n (*MaxTs*$_n$) plus one and *Clock*$_c$ (client's clock) plus one. This ensures that the timestamp is greater than both *Clock*$_c$ and any other update timestamped by p_n. Then, p_n stores the *Value* and the recently computed timestamp in the local key-value store and asynchronously sends the operation to the *Eunomia* service. *Eunomia* adds the operation to the set of non-stable operations Ops and updates the p_n entry in the *PartitionTime* vector with operation's timestamp (Alg. 3, lines 2–4). Finally, p_n returns the update's timestamp to the client who updates *Clock*$_c$ with it, since it is guaranteed to be greater than its previous one.

Timestamp Stability We say that a timestamp *Ts* is *stable* at the *Eunomia* servers when one is sure that no update with lower timestamp will be received from any partition (i.e., when *Eunomia* is aware of all updates with timestamp *Ts* or lower). Periodically, *Eunomia* computes the value of the maximum stable timestamp (*StableTime*), which is computed as the minimum of the *PartitionTime* vector (Alg. 3, line 8). Property 2 implies that no partition will ever timestamp an update with an equal or smaller timestamp than *StableTime*. Thus, *Eunomia* can confidently serialize all operations tagged with a timestamp smaller than or equal to *StableTime* (Alg. 3, line 9). *Eunomia* can serialize them in timestamp order, which is consistent to causality (Property 1), and then send them to other geo-locations (Alg. 3, line 10). Note that non-causally related updates coming from different partitions may have been timestamped with the same value. In this case, operations are known to be concurrent and *Eunomia* can process them in any order. For instance, it could use the identifier of the partition that sent the update to break ordering ties.

Heartbeats If a partition p_n does not receive an update for a fixed period of time, it will send a heartbeat including its current time to *Eunomia* (Alg. 2, lines 10–13). This is fundamental to ensure progress of

Algorithm 3 Operations at *Eunomia*

```
 1: function ADD_OP(Op)
 2:     Ops ← Ops ∪ Op
 3:     ⟨Key, Value, Ts, p_n⟩ ← Op
 4:     PartitionTime[p_n] ← Ts

 5: function HEARTBEAT(p_n, Ts)
 6:     PartitionTime[p_n] ← Ts

 7: function PROCESS_STABLE                                  ▷ Every θ time
 8:     StableTime ← MIN(PartitionTime)
 9:     StableOps ← FIND_STABLE(Ops, StableTime)
10:     PROCESS(StableOps)
11:     Ops ← Ops \ StableOps
```

the *site stabilization procedure*. Thus, if a partition p_n receives updates at a slower pace than others, it will not slow down the processing of other partitions updates at *Eunomia*. When *Eunomia* receives a heartbeat from p_n, it simply updates its entry in the *PartitionTime* vector. As an optimization to save the network bandwidth, partitions only send the heartbeat if it has not sent an update to Eunomia withing the heartbeat interval. If it has sent an update, the next timer will be triggered after MaxTs$_n$ $+ \Delta$ - Clock$_n$.

3.2 Fault-Tolerance

In the description above, for simplicity, we have described the *Eunomia* service as if implemented by a single non-replicated server. Naturally, as any other service in a datacenter, *Eunomia* must be made fault-tolerant. In fact, if *Eunomia* fails, the *site stabilization procedure* stops, and thus, local updates can no longer be propagated to other geo-locations. In order to avoid such limitation, we now propose a fault-tolerant version of *Eunomia*.

In this new version, *Eunomia* is composed by a set of *Replicas*. Algorithm 4 shows the behaviour of a replica e_f of the fault-tolerant *Eunomia* service. We assume the initial set of *Eunomia* replicas is common knowledge: every replica knows every other replica and all servers (that implement data partitions) know the full set of replicas. Partition servers send operations and heartbeats (Alg. 2, lines 8 and 13 respectively) to the whole set of *Eunomia* replicas. Each replica processes operations and heartbeats exactly as in Algorithm 3. Note that the algorithm is deterministic, and its output does not depend on the order of inputs. Thus, if the system would become quiescent, all replicas of *Eunomia* would eventually reach the same state and output the same stream of (serialized) operations.

Algorithm 4 Operations at *Eunomia* replica e_f

```
1: function PROCESS_STABLE                                        ▷ Every θ time
2:     if Leader_f == e_f then
3:         StableTime ← MIN(PartitionTime_f)
4:         StableOps ← FIND_STABLE(Ops_f, StableTime)
5:         PROCESS(StableOps)
6:         Ops_f ← Ops_f \ StableOps
7:         send STABLE(StableTime) to Replicas_f \ {e_f}

8: function STABLE(StableTime)
9:     StableOps ← FIND_STABLE(Ops_f, StableTime)
10:    Ops_f ← Ops_f \ StableOps

11: function NEW_LEADER(e_g)
12:    Leader_f ← e_g
```

To avoid unnecessary redundancy when exchanging metadata among datacenters, a leader-based strategy is used to select the replica in charge of propagating this information. The existence of a unique leader is not required for the correctness of the algorithm; it is simply a mechanism to save network resources. Thus, any leader election protocol designed for asynchronous system can be plugged into our implementation (such as Ω (Chandra, Hadzilacos, and Toueg 1996)). A change in the leadership is notified to a replica e_f through the NEW_LEADER function (Alg. 4, line 12).

The notion of a leader is used to optimize the service's operation as follows. When the PROCESS_-STABLE event is triggered, only the leader replica computes the new stable time and processes stable operations (Alg. 4, lines 2–5). Then, after operations have been processed, the leader sends the recently computed *StableTime* to the remaining replicas (Alg. 4, line 7). When replica e_f receives the new stable time, it removes the operations already known to be stable from its pending set of operations, since it is certain that those operations have been already processed (Alg. 4, lines 9–10).

3.3 Discussion

3.3.1 Correctness

We provide an informal proof that our protocol satisfies the two properties required by *Eunomia* (Properties 1 and 2).

Property 2 is trivial to prove. We need to ensure that updates handled by a partition p_n are tagged with strictly increasing timestamps and that heartbeats do not break the monotonicity. By Algorithm 2

line 5, p_n ensures that consecutive updates are tagged with increasing timestamps. On the other hand, heartbeats are only sent when the physical clock at p_n is greater or equal to the lastest timestamp used to tag an update plus a fixed time Δ (Alg. 2, line 10). This ensures that a heartbeat message is always tagged with a larger timestamp that all previously processed updates. Finally, an update happening right after a heartbeat is always tagged with a larger timestamp than the heartbeat's timestamp as we update the MaxTs_n after sending a heartbeat (Alg. 2 line 5).

In order to prove Property 1 we need to prove that the partial order derived from update timestamps is consistent with causality. The three properties of causality ($\rightsquigarrow$) are:

(i) *Execution thread:* for two operations a and b issued during the same client session, if a happens before b then $a \rightsquigarrow b$;

(ii) *Read from:* for an update operation a and a read operation b, $a \rightsquigarrow b$ if b reads the state written by a;

(iii) *Transitivity:* for operations a, b and c, if $a \rightsquigarrow c$ and $c \rightsquigarrow b$, then $a \rightsquigarrow b$.

These properties, applied to our protocol, imply that an update u_j issued by client c has to be tagged with a timestamp strictly greater than all its previous updates and than any version previously read. *Clock*$_c$, which is the clock maintained by the client, aggregates the client's causal history in a single scalar. By Algorithm 2 line 5, we know that the timestamp assigned to a client update is strictly greater than *Clock*$_c$. Thus, we only need to prove that *Clock*$_c$ is always equal or greater than all previously read versions, ensured by Algorithm 1 line 4, and that it is always greater or equal to the timestamp assigned to its last update, ensured by Algorithm 1 line 9.

3.3.2 Optimizing the Communication Patterns

Eunomia constantly receives operations and heartbeats from partitions. This is an all-to-one communication schema and, if the number of partitions is large, it may not scale in practice. In order to overcome this problem and efficiently manage a large number of partitions, two simple techniques have been used: build a propagation tree among partition servers; and batch operations at partitions, and propagate them to *Eunomia* only periodically. Both techniques (that can be combined) aim at reducing the amount of messages received by *Eunomia* per unit of time at the cost of a slight increase in the stabilization time.

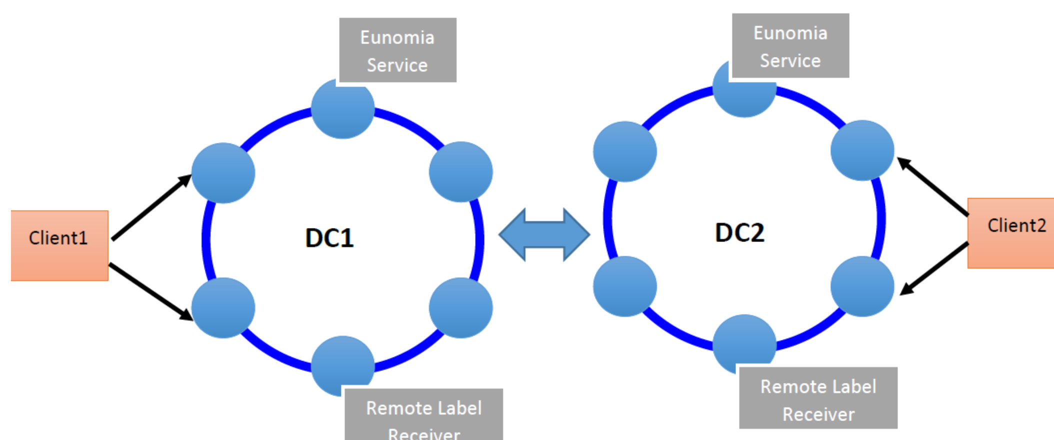


Figure 3.1: High Level Design of Geo Replicated Version built using Eunomia

3.4 Geo-replication

In our previous protocol we have shown how to unobtrusively timestamp local updates in a partial order consistent with causality. Still, we have not shown how to enforce causal consistency across geo-locations. In this section, we complete our previous protocol with the necessary mechanisms to ensure that remote updates—coming from other geo-locations—are made visible locally without violating causality. We assume a total of M geo-locations, each of them replicating the full set of objects. Each of these geo-locations uses the *Eunomia* service and thus propagates local updates in a total order consistent to causal consistency. We assume FIFO links between datacenters. For the simplicity of experiments we assumed that the clients always connect to their local datacenters. However, our algorithm does not restrict clients connecting to the remote datacenters. The overview of geo-replicated Eunomia is shown in Figure 3.1. For each datacenter we have an Eunomia instance running in order to receive labels from partitions in the same datacenter and a remote label receiver to receive the ordered labels from Eunomia instances from other datacenters. Delivering remote labels to the partitions at Remote Receiver is serialised: second label is only delivered when a confirmation is received from the partition which received the first label, indicating that the update corresponds to first label has committed at the responsible partition.

3.4.1 Protocol Extensions

We proceed to explain how the metadata is enriched and the changes we need to apply to our previous algorithms. Table 3.2 provides a summary of the notation used in this section.

M	Number of datacenters
VClock$_c$	Client c vector (M entries)
r_m	Receiver at datacenter m
SiteTime$_m$	Applied updates vector at r_m
Queue$_m$	Queues of pending updates at r_m
$u_j.vts$	Update u_j timestamp vector (M entries)

Table 3.2: Notation used in the protocol description.

Updates are now tagged with a vector with an entry per datacenter, capturing inter-datacenter dependencies. The client clock is consequently also extended to a vector (*VClock*$_c$).

Update When a client c issues an update operation, it piggybacks its *VClock*$_c$ summarizing both local and remote dependencies. A partition p_n computes u_j vector timestamp ($u_j.vts$) as follows. First, the local entry of the vector $u_j.vts[m]$ is computed as the maximum between *Clock*$_n$, *MaxTs*$_n + 1$ and *VClock*$_c[m] + 1$, similarly to Algorithm 2, line 5. This permits *Eunomia* to still be able to causally order local updates based on $u_j.vts[m]$. Second, the rest of the entries (remote datacenter entries) are assigned to its sibling entries in *VClock*$_c$. When the operation is completed, p_n returns $u_j.vts$ to the client who can directly substitutes its *VClock*$_c$ since $u_j.vts$ is known to be strictly greater than *VClock*$_c$.

Read Read operations execute as in Algorithms 1 and 2. The only difference is that the returned timestamp is a vector instead of a scalar. Thus, in order to update *VClock*$_c$, a client c applies the MAX operation per entry.

Update Propagation The *site stabilization procedure* proceeds as before, totally ordering local updates based on the local entry of their vector timestamp ($u.vts[m]$). *Eunomia* propagates local updates to remote datacenters in $u.vts[m]$ order. Each update piggybacks its $u.vts$.

Remote Update Visibility Algorithm 5 shows how a receiver handles a remote update arrival. When datacenter m receives a remote update u_j coming from datacenter k, two conditions have to be satisfied before applying it locally: all previously received updates coming from k have already been applied locally; and u_j dependencies, which are subsumed in $u_j.vts$, are visible locally. Both conditions can be trivially checked. The first condition can be enforced by simply keeping a queue (*Queue*$_m[k]$) of pending updates per remote datacenter (Alg. 5, line 3). The second condition can be enforced by maintaining a vector with an entry per remote datacenter (*SiteTime*$_m[k]$) indicating the latest update operation applied from each of the remote datacenters (Alg. 5, line 4). If any of the two conditions is not satisfied, u_j is added to *Queue*$_m[k]$ and will be eventually applied when both conditions hold.

Algorithm 5 Operations at r_m

```
1: function NEW_UPDATE(u_j, k)
2:     Queue_m[k] ← [Queue_m[k]|u_j]                    ▷ add to tail
3:     if HEAD(Queue_m[k]) == u_j then
4:         if ∀d ∈ M \ {m, k}, SiteTime_m[d] ≥ u_j[d] then
5:             send APPLY(u_j) to server
6:             receive ok from server
7:             SiteTime_m[k] ← MAX(SiteTime_m[k], u_j[k])
8:             POP(Queue_m[k])
9:             FLUSH_PENDING
```

FLUSH_PENDING is a recursive function (Algorithm 5 line 9) that makes sure no pending operation satisfying the above two conditions is left without being applied. When a pending operation u_j originating at k is applied, both $Queue_m[k]$ and $SiteTime_m[k]$ are updated consequently.

3.4.2 Discussion

3.4.2.1 Vector Clocks

Our protocol relies on vector clocks to ensure causal consistency across different geo-locations. We could easily adapt our protocols to use a single scalar, as in (Du, Iorgulescu, Roy, and Zwaenepoel 2014). Nevertheless, vector clocks make a more efficient tracking of causal dependencies introducing no false dependencies across datacenters, which reduces the update visibility latency, at the cost of slightly increasing the storage and computation overhead. Note that the lower-bound update visibility latency for a system relying on vector clocks is the latency between the originator of the update and the remote datacenter, while with a single scalar it is the time distance to the farthest datacenter regardless of the originator of the update.

3.4.2.2 Separation of Data and Metadata

In the protocols described before, partitions send updates (including the update value) to the *Eunomia* service, which is responsible for eventually propagating them to remote datacenters. This can limit the maximum load that *Eunomia* can handle and become a bottleneck due to the potentially large amount of data that has to be handled. In order to overcome this limitation, we propose decoupling data and metadata.

Thus, in our prototype, for each update operation, partitions generate a unique update identifier

($u.id$), composed of the local entry of the update vector timestamp ($u.vts[m]$) and the object identifier (*Key*). We avoid sending the value of the update to *Eunomia*. Instead, partitions only send the unique identifier $u.id$ together with the partition id (p_n). *Eunomia* is only responsible for handling and propagating these lightweight identifiers, while the partitions itself are responsible for propagating the update values together with $u.id$ to its sibling partitions in other datacenters. A receiver r_m proceed as before, but a partition p_n can only execute the remote operation once it has received both the data and the metadata. This technique slightly increases the computation overhead at partitions, but it allows *Eunomia* to handle a significantly heavier load independently of update values.

3.4.2.3 Datacenter Client Failover

In case a client c cannot connect to its preferred datacenter due to a network partition, client c may fail-over to another datacenter k if required. Nevertheless, before c can start issuing operations in k, the system has to ensure that all updates in the client's causal history have already been executed in k. By construction, *VClock*$_c$ encapsulates enough causal information to ensure a safe fail-over. Thus, a client only needs to connect to a remote receiver r_k (the authority keeping the information regarding remote updates) and send its *VClock*$_c$. The receiver r_k proceeds as with remote updates by checking causal dependencies. Once *SiteTime*$_m$ is greater or equal than *VClock*$_c$, c can safely issue operations in the remote datacenter k without violating causality.

3.5 Implementation

The *Eunomia* service has been implemented in both `C++` (to find maximum upper bounds) and `Erlang` (to integrate with Riak KV) programming languages. Riak KV is a weakly consistent datastore used by many companies offering cloud-based services including bet365 and Rovio. Since Riak KV is implemented in Erlang, we first attempted to build *Eunomia* using the `Erlang/OTP` framework, but unfortunately we rapidly reached a bottleneck in our early experiments due to the inefficiency of `Erlang` data structures. Note that for *Eunomia* to work, we need to store a potentially very large number of updates, coming from all logical partitions composing a datacenter, and periodically traversed them in timestamp order when a new stable time is computed. Inserting and traversing this (ordered) set of updates was limiting the maximum load that *Eunomia* could handle. The `C++` implementation does not suffer from these performance limitations.

3.5.1 *Eunomia* implementation in c++

Our calculations showed that the number of nodes required to achieve maximum throughput in the cluster is around 100 nodes. Therefore, We modified basho bench to simulate the partition behavior as in a real Riak KV cluster the throughput is limited by the capacity of the cluster itself. For the message serialization between partitions and Eunomia Service we used JSON. The updates from partitions to the *Eunomia* is batched for two milliseconds to reduce the communication overhead in case of frequent updates. The *Eunomia* Service runs two threads: one to receive the batch labels, deserialize the received data and add them to the queue; and the other thread to process messages in the queue. Eunomia uses epolling for the asynchronous io multiplexing in order to handle large number of client connections. The queue used by the receiver and processing thread is implemented in a way to minimize the locking overhead. The Queue Holder is composed of three different types of queue sections such as owner queue, common and foreign queue. These 3 queues are logical representation of the linked-list. These 3 queues inside the singe queue holder decreases the frequent thread locking by bulk copying the messages from producer queue to the common queue and common queue to the consumer queue periodically. To guarantee FIFO links we used Erlang's tcp send, receive and gen-server primitives.

At its core, *Eunomia* is implemented using a *red-black tree* (Guibas and Sedgewick 1978), a self-balancing binary search tree optimized for insertions and deletions, which guarantees logarithmic search, insert and delete cost, and linear in-order traversal cost, a critical operation for *Eunomia*. For our particular case, the *red-black tree* turned out to be more efficient than other self-balancing binary search trees such as AVL trees (Pfenning 2011).

3.5.2 Geo Replication

Furthermore, in order to fully explore the capacities of *Eunomia* and experimentally demonstrate our hypothesis, we have integrated *Eunomia* with a causally consistent geo-replicated datastore implementing the protocol presented in §3.1 and §3.4. Our prototype, namely *EunomiaKV*, is built as a variant of Riak KV (Riak 2011), and includes the optimizations discussed in §3.3 and §3.4.2. Since the open source version does not support replication across Riak KV clusters, we have also augmented the open source version of Riak KV with geo-replication support. For this implementation, the receiving data node forward the update to a random node in each remote datacenter. Then these messages will be routed to the intended receiver based on DHT routing.

The remote label receiver maintains a queue per each datacenter to track labels based on datacenter identifier. This reduces the overhead of checking the deliverable labels. Every time when a remote label is delivered to a partition, it should scan whether there are any other labels in queues of other remote datacenters than from the one we received can be delivered. As we maintain a queue per datacenter, now we only need to scan the head of the queue per each datacenter to make sure all the deliverable labels are delivered to partitions. If the head is not deliverable, as labels from a remote datacenter are ordered, other labels in the queue are also not deliverable. In order to deliver a label, all the dependencies encoded in the label's vector should have been applied. This can be verified by comparing label's vector against the Remote Receiver's vector. Other than the vector position corresponds to label's own datacenter, all the other positions in the label's vector should be $\leqslant$ Remote Receiver's vector.

In partitions we maintain a hashmap with a unique identifier for each remote update/label. Every time when a label is received, it checks the hashmap to see whether data has already been received. If data is there, update is committed and then a reply is sent to the remote label receiver. Otherwise it adds the label to the hashmap. Similarly when the partition receives the data, it checks whether label is in the hashmap. If it is, it commit the update and sends a reply to the remote receiver. In our case, due to the delay caused by the stabilization procedure at Eunomia, data is received first in most of the scenarios.

3.5.3 Benefits of using Logical Physical Clocks in Erlang

Erlang uses *erlang:now()* primitive to get monotonic timestamps. Since it is incremental, the Erlang virtual machine must keep the previous value and check the next value against it. This is a synchronization point. On a multicore though, synchronisation is expensive and kills the scalability. The other function call, *os:timestamp()* is not strictly monotonic, but it avoids the bottleneck of synchronization. As we use logical physical clocks, properties of logical physical clocks make sure that we always have incremental timestamps for causally related updates even if the clock is not monotonic.

Summary

In this chapter we have been through the design and implementation of *Eunomia*. Initially, we elaborated the concept behind *Eunomia* and introduced the algorithm to order updates within the same datacenter. Furthermore, we introduced the fault tolerant protocol for *Eunomia*. Then we explained in

detail the correctness of the proposed algorithm theoretically. Then we introduced the algorithm to provide causally consistent geo-replication and proved the correctness of the algorithm. Finally, explained the efficient implementation of *Eunomia* and integrating the algorithms to a well-known distributed key value storage Riak KV.

In the next chapter we present the experimental evaluation made using this prototype.

4 Evaluation

Our main goal with the evaluation is to show that *Eunomia* does not suffer from the limitations of the competing approaches. Therefore, we compare *Eunomia* with both *sequencer based* approaches and with approaches based on *global stabilization checking* procedures. We recall that the main disadvantage of sequencers is to throttle throughput, by operating in the critical path of local clients. Therefore, we aim at showing that *Eunomia* does not compromise the intra-datacenter concurrency and can reach much better throughput than sequencer based approaches. Conversely, the disadvantage of the global stabilization approach is to introduce long delays in update visibility at remote sites. Thus, we also aim at showing that *Eunomia* does not significantly delay remote update visibility.

Experimental Setup The experimental test-bed used is a private cloud composed by a set of virtual machines deployed over 20 physical machines (8 cores and 40 GB of RAM) connected via a Gigabit switch. Each VM, which runs Ubuntu 14.04, and is equipped with 2 (virtual) cores, 10GB disk and 9GB of RAM memory; is allocated in a different physical machine. Before running each experiment, physical clocks are synchronized using the NTP protocol (NTP 2008) through a near NTP server.

Workload Generator Each client VM runs its own instance of a custom version of Basho Bench (Basho 2011b), a load-generator and benchmarking tool to conduct accurate and repeatable performance tests. For each experiment, we deploy as many client instances as possible without overloading the system. Latencies across datacenters are emulated using `netem` (netem 2011), a Linux network emulator tool. The round trip time between two nodes within the same datacenter was around 1ms.

In our experiments, unless specified, we use the following parameters. Values used in operations are a fixed binary of 100 bytes. We use a uniform key distribution across a total of 100k keys (objects). The ratio of reads and updates is varied depending on the experiment. Before running the experiments, we populate the database. Each experiment runs for 10 minutes. For the *Eunomia* max throughput experiments, we batch labels for 2ms. For the Geo Replication experiments, we batch labels within a heartbeat interval and send them along with the heartbeats. In our results, the first and the last minute of each experiment is ignored to avoid experimental artifacts.

4.1 *Eunomia* Throughput

We first report on a number of experiments that aim at following considerations:

(i) Experimentally measuring the maximum load that can be handled by *Eunomia* implemented in Erlang

(ii) Experimentally measuring the maximum load that our efficient implementation of *Eunomia* can handle, varying the number of partitions connected to it

(iii) Comparing *Eunomia* to traditional sequencers

(iv) Assessing how failures affect the performance of the *Eunomia* service

For comparison, these experiments also show the maximum load that a traditional sequencer can handle. Our implementation of a sequencer is the simplest possible and mimics traditional implementations. In every update operation, data servers synchronously request a monotonically increasing number to the sequencer before returning to the client. We have also implemented a fault-tolerant version of the sequencer based on chain replication (van Renesse and Schneider 2004): Replicas of the sequencer are organized in a chain. Partitions send requests to the head of the chain. Requests traverse the chain up to the tail. When the tail receives a request, it replies back to the data server, which in turn returns to the client. In our experiments the chain is composed of three replicas.

In order to stretch as much as possible the implementation, circumventing other potential sources of bottlenecks in the system, we directly connect clients to *Eunomia*, bypassing the data store. Thus, each client simulates a different server (in charge of a data partition) in a multi-server datacenter. This allowed us to emulate the use of very large datacenters, with much more data servers than the ones that were at our disposal for this experiments, and overload *Eunomia* in a way that would be otherwise impossible with our testbed.

4.1.1 Maximum throughput achieved in Erlang Implementation

This experiment shows the result of the implementation of non fault-tolerant version of Eunomia in Erlang. Eunomia Erlang implementation suffered from the inefficient datastructure implementation and garbage collection issues. Since Erlang is immutable, the data structures used for large number of record handling such as gbtree (Ericsson 1997b) and ets (Ericsson 1997a) table are implemented in C and

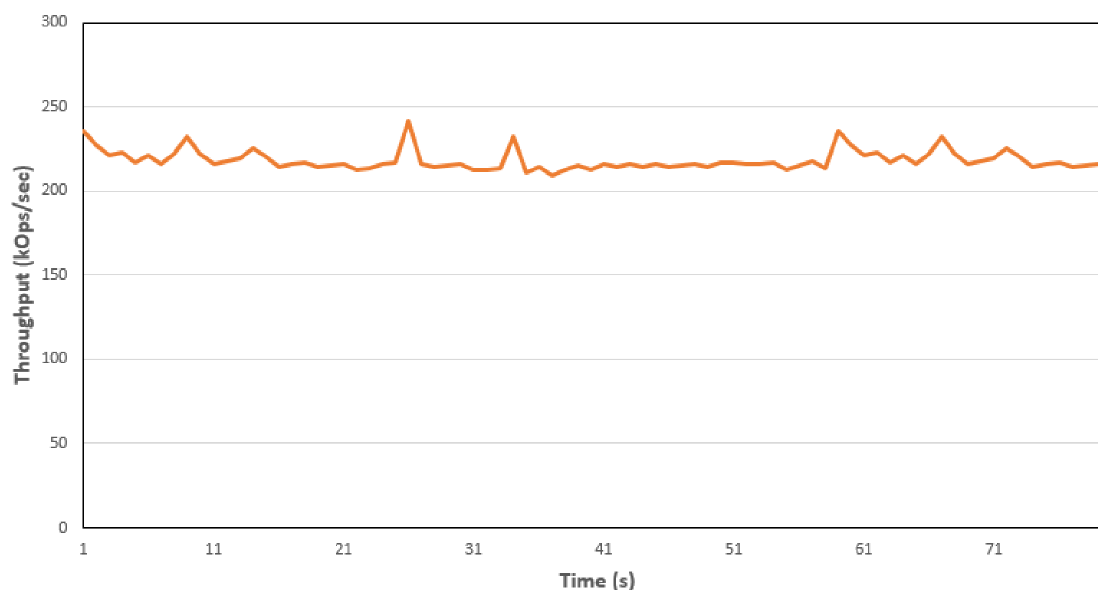


Figure 4.1: Throughput Limit of Eunomia Implemented in Erlang.

communication between them happens via the native interface calls. But during the communication with the c native interface, it does not allow context switching which causes other scheduled Erlang threads to starve. As in our case, we need to call the native interface for each batch insert and delete, these calls are very frequent. This causes garbage collector to not run regularly; which in turn causes the memory to overflow soon. Therefore throughput of this version was limited around 215k operations per second as shown in Figure 4.1.

4.1.2 Throughput Upper-Bound with Eunomia c++ Implementation

To overcome the overhead of memory management and see the maximum bound that can handled by *Eunomia* we implemented Eunomia in c++ and measured the throughput upper bound.

In this experiment we compare the non fault-tolerant version of the *Eunomia* against a non fault-tolerant implementation of a sequencer implemented in c++. The *Eunomia* implementation used for the experiment is configured to batch updates and only send them to *Eunomia* after 2ms. Figure 4.2 plots the maximum throughput achieved by both services.

As results show, *Eunomia* maximum throughput is reached when having 60 servers (data partitions) issuing operations eagerly (with zero waiting time between operations). We observe that *Eunomia* is able to handle almost an order of magnitude more operations per second than a sequencer (more precisely, 7.7 times more operations, exceeding 370kops while the sequencer is saturated at 48kops). Considering

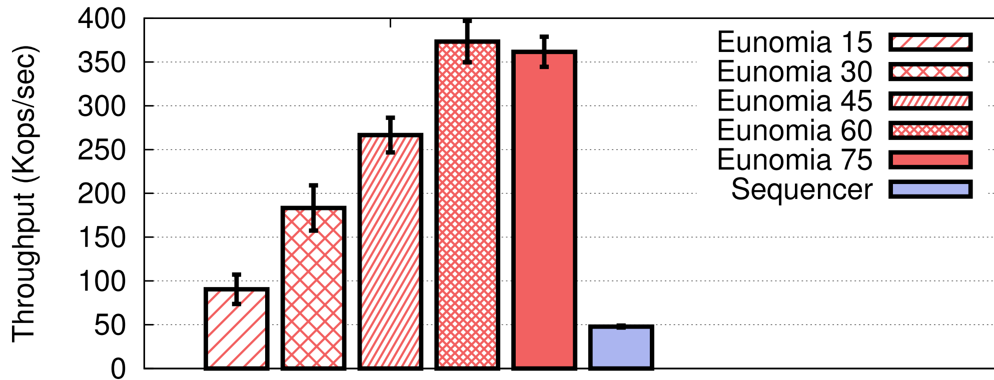


Figure 4.2: Maximum throughput achieved by *Eunomia* and an implementation of a sequencer. We vary the number of partitions that propagate operations to *Eunomia*.

that according to our experiments, a single machine in a Riak KV cluster is able to handle approximately 3kops per second, These results confirm that sequencers limit intra-datacenter concurrency and can easily become a bottleneck for medium size clusters (i.e, for clusters above 150 machines, the sequencer would be the limiting factor of system performance), even assuming a read dominant (9:1) workload, a common workload for internet-based services. On the other hand, under the same workload assumptions, more than a thousand data servers could be used before saturating *Eunomia*.

Another advantage of *Eunomia* in comparison to sequencers is that, as discussed in §3.3, batching is not in the client critical path. Thus, *Eunomia*'s throughput can be further stretched by increasing the batching time (at the cost of slightly increasing the remote update visibility latency). Such stretching cannot be easily achieved with sequencers, as any attempt to batch requests at the sequencer blocks clients.

A final conclusion can be drawn from this experiment: *Eunomia* maximum capacity does not significantly varies with the number of data servers. Although we hit the maximum load with 60 partitions, we run an extra experiment increasing the number to 75 to see if this negatively impacts *Eunomia*'s performance and we observed a very similar throughput. The reason is that the bottleneck of our *Eunomia* implementation is the propagation to other geo-locations rather than the handling of operations. This confirms that the use of a red-black self-balancing search tree was an appropriate design choice.

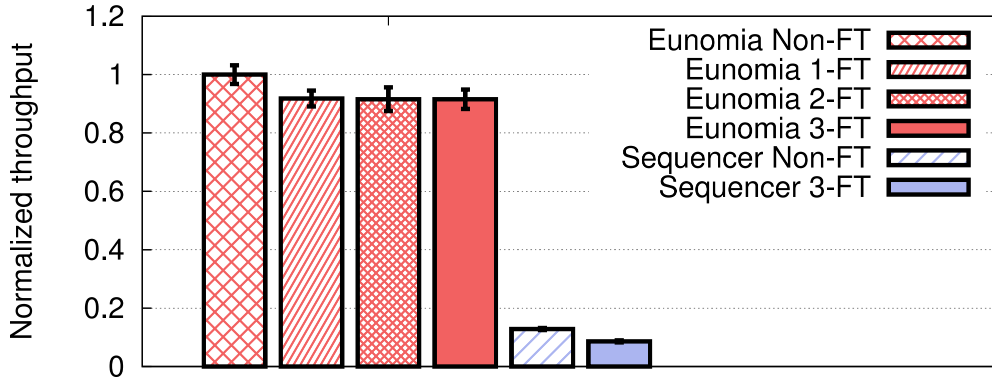


Figure 4.3: Maximum throughput achieved by a fault-tolerant version of *Eunomia* and sequencers. Non-FT denotes non fault-tolerant versions while 1-, 2-, and 3-FT denote fault-tolerant version with 3 replicas and throughput at 1st, 2nd, and 3rd replica

4.1.3 Fault-Tolerance Overhead

In the following experiments we measure the overhead introduced by the fault-tolerant version of *Eunomia* and the impact of faults in the service. Figure 4.3 compares the maximum throughput achievable by *Eunomia* with 3 replicas. For completeness, the plot also includes the throughput for a non fault-tolerant sequencer and its fault-tolerant version with three replicas. We normalized the throughput against the non fault-tolerant version of *Eunomia*. As results show, the fault-tolerant version of *Eunomia* only adds a small overhead (roughly 9% penalty) when we are having 3 replicas. We expect this overhead to increase as the number of replicas increases, but we consider three replicas to be a realistic number. On the other hand, adding fault-tolerance to the sequencer version adds a penalty of almost 33%, thus being more expensive proportionally. The reason for this difference is that, as explained before, *Eunomia* replicas do not need to coordinate as their results are independent of relative order of inputs, while sequencer replicas need to coordinate to avoid providing inconsistent sequence numbers.

Finally, we experiment injecting failures into *Eunomia*. Figure 4.4 plots the results normalized against the non fault-tolerant *Eunomia* (Non-FT line). We compare *Eunomia* with one, two, and three replicas. As the figure shows, at the begging of the experiment, all three versions produce similar throughput (confirming Figure 4.3 results). After 160 seconds, we crash one replica. As expected, the throughput of 1-FT drops to zero since no more replicas are available. The rest of the versions (2-FT and 3-FT), after a short period of fluctuation, slightly increase their throughput up to 95% of the Non-FT version throughput. Finally, after 210 more seconds (at 470), we crash a second replica. Again, the

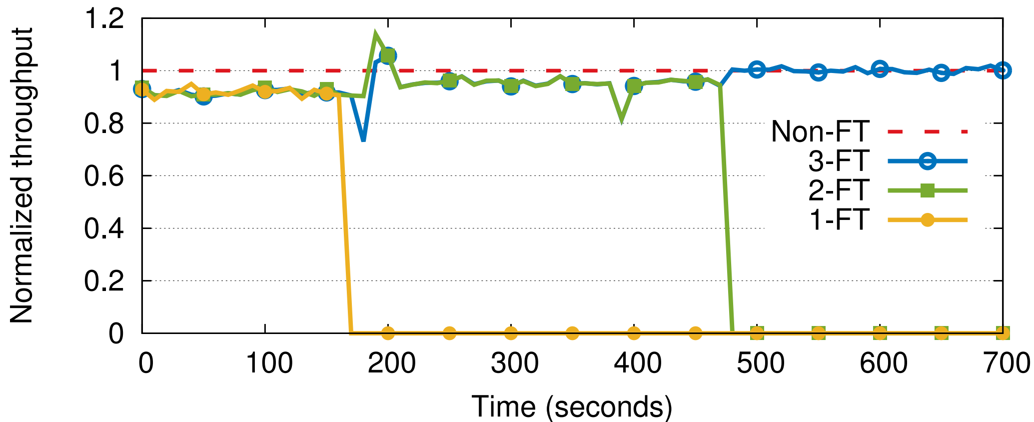


Figure 4.4: Impact of failures in *Eunomia*.

2-FT as expected drops its throughput to zero. The 3-FT version, this time almost without fluctuations, is capable of achieving the maximum throughput in few seconds. These results demonstrate that failures have practically no impact in *Eunomia*. Note that sometimes the multi-replica version go beyond the Non-FT line. This is just because the Non-FT line is drawn by computing the average.

4.2 Experiments with Geo-Replication

We now report on a set of experiments offering evidence that a causally consistent geo-replicated datastore built using *Eunomia* is capable of providing higher throughput and better quality-of-service than previous solutions that avoid the use of local sequencers.

For this purpose, we have implemented GentleRain (Du, Iorgulescu, Roy, and Zwaenepoel 2014) and a variation of it that uses vector clocks instead of a single scalar to enforce causal consistency across geo-locations. The latter resembles the causally consistency protocol implemented by Cure (Akkoorath, Tomsic, Bravo, Li, Crain, Bieniusa, Preguiça, and Shapiro 2016). Both approaches are sequencer-free implementations that rely on a global stabilization checking procedure in order to apply operations in remote locations consistently with causality. For this, sibling partitions across datacenters have to periodically send heartbeats, and each partition within a datacenter has to periodically compute its local-datacenter stable time. In our experiments, we set the frequency of this events to 10 ms and 5 ms respectively unless otherwise specified. These values are in consonance to the ones used by the authors of these works. Both approaches are implemented using the codebase of *EunomiaKV* and thus integrated with Riak KV. To simplify the implementation, we avoided building the propagation tree and used all to

all communication to calculate the GST. This is an acceptable design choice as tree is only required if the number of partitions in the cluster is large.

In most of our experiments, we deploy 3 datacenters, each of them composed of 8 logical partitions balanced across 3 servers. The emulated round-trip-times across datacenters (DCs) are shown in Table 4.1 . These latencies are approximately the round-trip-times between Virginia, Oregon and Ireland regions of Amazon EC2.

DC1 to DC2	80ms
DC1 to DC3	80ms
DC2 to DC3	160ms

Table 4.1: Round Trip Time Between Datacenters.

4.2.1 Throughput

In the following experiments, we measure the throughput provided by *EunomiaKV*, GentleRain, Cure, and an eventually consistent multi-cluster version of Riak KV. Note that the latter does not enforce causality, and thus partitions execute remote updates as soon as they are received. Therefore, the comparison of Eunomia with Riak KV allows to assess the overhead induced by Eunomia for providing causal consistency. As discussed below, this overhead is very small.

We experiment with both uniform and power-law key distributions, denoted with U and P respectively in Figure 4.5. For each of them, we vary the read:write ratio (99:1, 90:10, 75:25 and 50:50). These ratios are representative of real large internet-based services workloads. As shown by Figure 4.5, the throughput of all solutions decreases as we increase the percentage of updates. Nevertheless, *EunomiaKV* always provides a comparable throughput to eventual consistency. Precisely, on average, *EunomiaKV* only drops 4.7% of throughput, being extremely close in read intensive workloads (1% drop). On the other hand, GentleRain and Cure are always significantly below both eventual consistency and *EunomiaKV*. This is due to the cost of the global stabilization checking procedure. Note that the throughput difference between GentleRain and Cure is caused by the overhead introduced by the metadata enrichment procedure of the latter (as discussed in §3.4.2). Based on our experiments, it is possible to conclude that the absolute number of updates per unit of time is the factor that has the largest impact in *EunomiaKV* (rather than key contention).

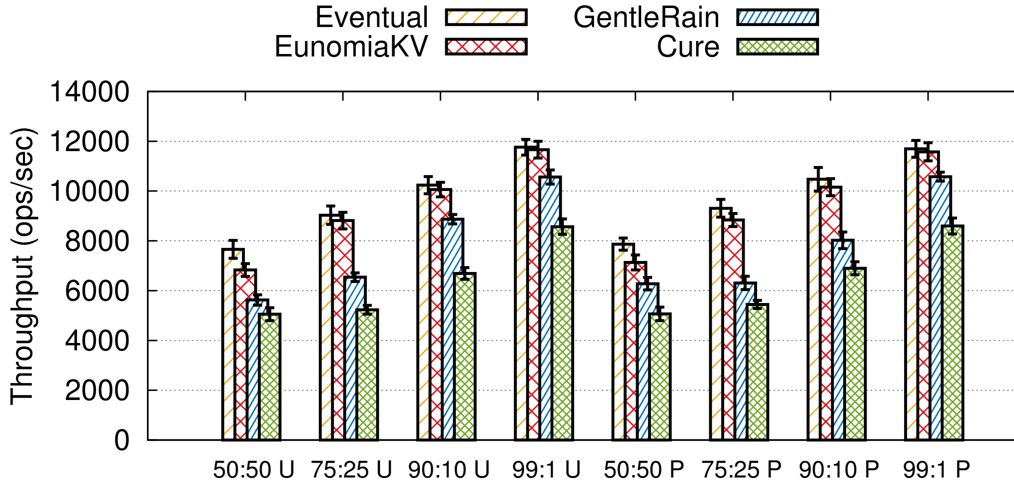


Figure 4.5: Throughput comparison between *EunomiaKV* and state-of-the-art sequencer-free solutions.

4.2.2 Remote Update Visibility

In order to compare the quality-of-service that can be provided by *EunomiaKV*, GentleRain, and Cure, we measure remote update visibility latency. In *EunomiaKV*, we measure the time interval between the data arrival and the instant in which the update is executed at the responsible partition. Note that, for an update to be applied, a datacenter needs to have access to the metadata (in our case, provided by *Eunomia*) and check that all of its causal dependencies have also been previously applied locally. In our implementation, partitions ship updates immediately to remote datacenters. Therefore, we have observed that updates are always locally available to be applied by the time metadata indicates that its causal dependencies are already satisfied locally. Although other strategies could be used to ship the payload of the updates, this has a crucial advantage for the evaluation of Eunomia: under this deployment the update visibility latency is exclusively influenced by the performance of the metadata management strategy, including the stabilization delay incurred at the originating datacenter.

On the other hand, for GentleRain and Cure, we measure the time interval between the arrival of the remote operation to the partition and when the global stabilization checking procedure allows its visibility. Note that all values presented in the figures already factor-out the network latencies among datacenters (which are the same for all protocols); thus numbers capture only the artifical artifacts inherent to the different approaches.

Figure 4.6 shows the cumulative distribution of the latency before updates originating at DC1 become visible at DC2. We observe that *EunomiaKV* offers, by far, the best remote update visibility latency.

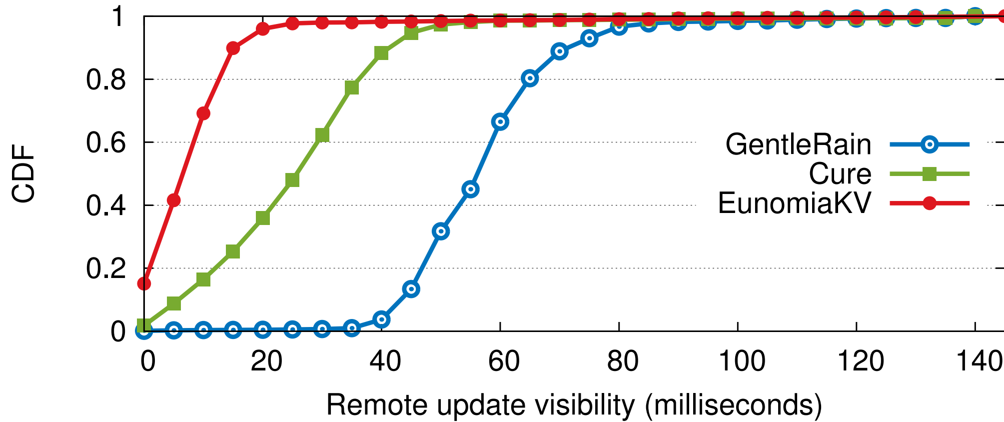


Figure 4.6: Visibility latency of remotes updates originating at DC1 measured at DC2 (40ms trip-time).

In fact, for almost 95% of remote updates, *EunomiaKV* only adds 15ms extra delay. On the other hand, with GentleRain and Cure the extra delay goes up to 80ms and 45ms respectively for the same amount of updates. Unsurprisingly, GentleRain extra delay is larger than Cure's because of the amount of false dependencies added when aggregating causal dependencies into a single scalar. In fact, GentleRain is not capable of making updates visible without adding 40ms of extra delay. Again, the scalar is the cause of this phenomenon since the minimum delay will not depend on the originator of the update but on the travel time to the furthest datacenter. This confirms the rationale presented in the discussion of §3.4.2.

Although both Cure and *EunomiaKV* rely on vector clocks for tracking causal dependencies, *EunomiaKV* is able to offer better remote update latencies because partitions are less overloaded since checking dependencies in *EunomiaKV* is trivial due to *Eunomia*. Note that in *EunomiaKV*, even 20% of remote updates are made visible without any extra delay, and thus reaching the optimal remote update visibility latency.

Finally, in order to isolate the impact of GentleRain's global stabilization checking procedure independently of the metadata size, we measure the remote update visibility latency at DC3 for updates originating at DC2. As one can observe in Figure 4.7, GentleRain exhibits better remote update latencies than Cure but still worse than *EunomiaKV*. In this setting, vector clocks does not help reducing latencies. Thus, the gap between Cure and GentleRain is exclusively due to the storage and computational overhead caused by vector clocks. Furthermore, the fact that *EunomiaKV* still provides better latencies is, once again, an empirical evidence that global stabilization checking procedures are expensive in practice.

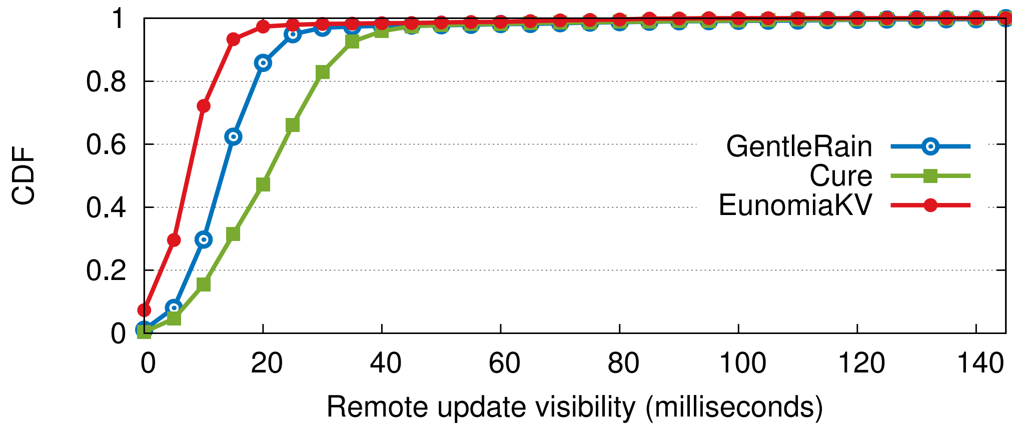


Figure 4.7: Visibility latency of remotes updates originating at DC2 measured at DC3 (80ms trip-time).

4.2.3 Throughput vs Remote Visibility Tradeoff

In this last subsection, we discuss the tradeoff between throughput and remote update visibility latency posed by solutions relying on a global stabilization checking procedures. In our previous experiments, we set the local-datacenter stable time computation frequency to 5ms, which, according to our experiments, does the best tradeoff. In this section we explore this tradeoff in more detail, to provide a better insight on the cost of global stabilization checking procedures.

For this purpose, we vary the local-datacenter stable time computation frequency between 1ms and 100ms. We measure both the remote update visibility latency and the throughput. Again, latencies refer to the artificial delay added by the system at DC2 for updates originating at DC1. Figure 4.8 shows the results of this experiment. The graph at the bottom plots throughput numbers. The graph on the top plots the 90th percentile of remote update visibility latencies. As one can observe, as we increase the frequency of the computation, lower artificial delays are added but also lower throughput is provided. We can observe that the throughput decreases more rapidly when the experiment moves towards higher frequencies. The figure also shows the throughput and update visibility latencies of *EunomiaKV*. Even when the computation is very frequent (1ms), GentleRain and Cure do not reach remote update latencies comparable to *EunomiaKV* while its drop in throughput is quite significant (more precisely 18.2% for GentleRain and 44.4% for Cure). We noticed that our version of Cure was slightly overloaded for 1ms frequency. Thus, artifical delays where drastically increased (90th percentile of 75ms), even surpassing GentleRain's 90th percentile (66ms) for the same frequency. We have not plotted this to avoid obstructing plot readability.

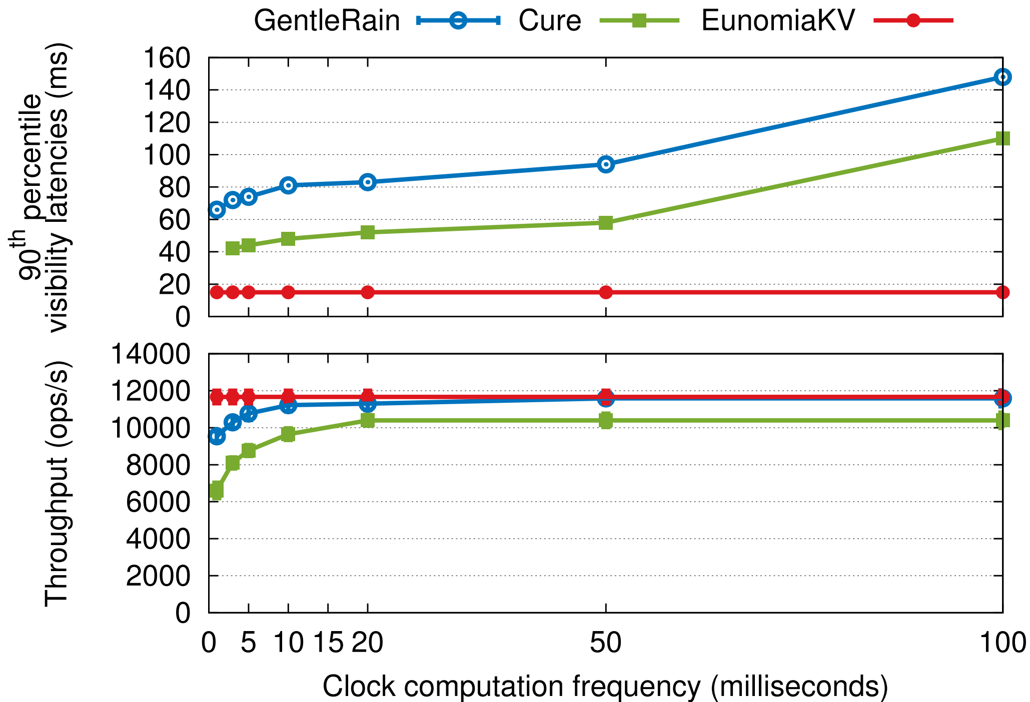


Figure 4.8: Tradeoff between throughput and remote update visibility latency posed by global stabilization checking procedures.

These results confirm that global stabilization checking procedures are quite expensive in practice and that solutions based on them are far from achieving remote update latencies close to optimal latencies.

Summary

In this chapter we introduced the experimental evaluation made with the implementation. First we implemented Eunomia service, which orders the updates initiating from the local datacenter. As the Erlang implementation of Eunomia suffers from the issue of inefficient memory handling in Erlang, we used the c++ implementation of Eunomia to measure the max bounds. Experimental results proved that Eunomia can perform 7.7 times more operations than sequencer. Then we evaluated the overhead of fault tolerant implementations compared to non fault tolerant versions for both Eunomia and Sequencer. Experiment results confirmed that making Eunomia fault tolerant causes only around 9% throughput loss whereas making Sequencer fault tolerant causes around 33% throughput loss.

Then we evaluated our geo-replicated system build on top of Riak KV. For the comparison we used

two state of the art solutions: Gentlerain and Cure. We could observed that for all the common workload patterns observed in the real world Eunomia was able to achieve higher throughput while maintaining the remote update visibility delay less than that of Gentlerain and Cure. We also observed that Eunomia only causes around 4.7% throughput loss compared to Eventual consistency.

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

5 Conclusions

We have presented a novel approach for building weakly consistent geo-replicated data stores that require updates to be causally ordered. Our solution relies on a new service, namely *Eunomia*, that abstracts the internal complexity of datacenters, a key characteristic to inexpensively implement causal consistency across geo-locations. Furthermore, unlike sequencers, *Eunomia* does not limit the intra-datacenter concurrency by performing a cheap and unobtrusive ordering of updates.

We have presented the techniques behind *Eunomia* and a high performant causally consistent geo-replicated protocol that integrates it. Our experimental results demonstrate that *Eunomia*, unlike sequencers, is able to handle very heavy loads without becoming a performance bottleneck (up to 7.7 times more operations per second than a sequencer). Furthermore, we have built a data store, namely *EunomiaKV*, that implements the proposed geo-replicated protocol. Our prototype was built as a variant of the Riak KV data store. According to our experiments, *EunomiaKV* is currently the most performant causal consistency implementation, providing appreciably higher throughput than GentleRain and Cure, the two most performant solutions of the state-of-the-art. In fact, *EunomiaKV* only adds a slight overhead (4.7% on average) when compared to an eventually consistent data store that makes no attempt to enforce causality. In addition, experiments have demonstrated that, unlike GentleRain and Cure, *EunomiaKV* does not deteriorate the quality-of-service provided to clients, introducing exceptionally small artificial delays on remote update visibility.

The Geo-Replicated system built on top of *Eunomia* still need to serialize the remote updates. Even though we did not observe any overhead with our workloads, in an update-heavy environment, this serialization can increase the remote visibility delay and introduce some limitations to the throughput of the system. To avoid such serialization while keeping the matadata small is a problem not trivial to solve but that would be interesting to address in the future.

References

Ahamad, M., R. John, P. Kohli, and G. Neiger (1994). Causal memory meets the consistency and performance needs of distributed applications. In *Proceedings of the 6th Workshop on ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, EW 6, Wadern, Germany, pp. 45–50.

Akkoorath, D. D., A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro (2016). Cure: Strong semantics meets high availability and low latency. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems*, ICDCS'16, Osaka, Japan.

Almeida, S., J. Leitão, and L. Rodrigues (2013). Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, Prague, Czech Republic, pp. 85–98.

Attiya, H., F. Ellen, and A. Morrison (2015). Limitations of highly-available eventually-consistent data stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, Donostia-San Sebastiao, Spain, pp. 385–394.

Attiya, H. and J. L. Welch (1994). Sequential consistency versus linearizability. *ACM Transactions on Computer Systems 12*(2), 91–122.

Bailis, P., A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica (2013). Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment 7*(3), 181–192.

Bailis, P., A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica (2012). The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, San Jose, California, pp. 22:1–22:7.

Balakrishnan, M., D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis (2012). Corfu: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, pp. 1–1.

Basho (2011a). Active anti-entropy. Accessed on 2015-02-01. `http://docs.basho.com/riak/latest/theory/concepts/aae/`.

Basho (2011b). Benchmarking. Accessed on 2016-02-16.
`http://docs.basho.com/riak/kv/2.1.4/using/performance/benchmarking/`.

Belaramani, N., M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and 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.

Birman, K., A. Schiper, and P. Stephenson (1991). Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems 9*(3), 272–314.

Bravo, M., L. Rodrigues, and P. Van Roy (2015). Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, Middleware Doct Symposium '15, Vancouver, BC, Canada, pp. 5:1–5:4.

Chandra, T. D., V. Hadzilacos, and S. Toueg (1996). The weakest failure detector for solving consensus. *Journal of the ACM 43*(4), 685–722.

CockroachDB (2016). The scalable, survivable, strongly consistent, sql database. Accessed on 2016-01-16. `http://www.cockroachlabs.com/`.

Crain, T. and M. Shapiro (2015). Designing a causally consistent protocol for geo-distributed partial replication. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, Bordeaux, France, pp. 6:1–6:4.

Du, J., S. Elnikety, A. Roy, and W. Zwaenepoel (2013). Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, California, pp. 11:1–11:14.

Du, J., C. Iorgulescu, A. Roy, and W. Zwaenepoel (2014). Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, Seattle, WA, USA, pp. 4:1–4:13.

Du, J., D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone (2014). Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, Washington, DC, USA, pp. 343–354.

Ericsson (1997a). Built-in term storage. Accessed on 2016-04-19.

http://erlang.org/doc/man/ets.html.

Ericsson (1997b). General balanced trees. Accessed on 2016-01-16.
http://erlang.org/doc/man/gb_trees.html.

Gilbert, S. and N. Lynch (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*(2), 51–59.

Guibas, L. J. and R. Sedgewick (1978). A dichromatic framework for balanced trees. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science 0*, 8–21.

Herlihy, M. P. and J. M. Wing (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*(3), 463–492.

Katsov., I. (2009). Nosql data modeling techniques. Accessed on 2015-01-25. https://goo.gl/1P8mzX.

Kulkarni, S. S., M. Demirbas, D. Madappa, B. Avva, and M. Leone (2014). *Logical Physical Clocks*, pp. 17–32. Cham: Springer.

Ladin, R., B. Liskov, L. Shrira, and S. Ghemawat (1992). Providing high availability using lazy replication. *ACM Transactions on Computer Systems 10*(4), 360–391.

Lakshman, A. and P. Malik (2010). Cassandra: A decentralized structured storage system. *Special Interest Group on Operating Systems Review 44*(2), 35–40.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*(7), 558–565.

Lloyd, W., M. J. Freedman, M. Kaminsky, and 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.

Lloyd, W., M. J. Freedman, M. Kaminsky, and D. G. Andersen (2013). Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, Lombard, IL, pp. 313–328.

netem (2011). Network emulator tool for linux. Accessed on 2016-05-01.
http://goo.gl/Ib3HDh.

NTP (2008). The network time protocol. Accessed on 2016-04-16.
http://www.ntp.org.

Petersen, K., M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers (1997a). Flexible update propagation for weakly consistent replication. *ACM SIGOPS Operating Systems Review 31*(5), 288–301.

Petersen, K., M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers (1997b). 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.

Pfenning, F. (2011). Lecture notes on avl trees. Accessed on 2016-03-19. `http://www.cs.cmu.edu/~fp/courses/15122-s11/lectures/18-avl.pdf/`.

Preguiça, N., M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro (2014). Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, SRDSW '14, Washington, DC, USA, pp. 30–33.

Riak (2011). Riak distributed key value storage. Accessed on 2015-01-03. `http://docs.basho.com/riak/latest/theory/why-riak/`.

Seeger., M. (2009). Key-value stores: a practical overview. Accessed on 2015-01-03. `http://goo.gl/O8v5E9`.

Shapiro, M., N. Preguiça, C. Baquero, and M. Zawirski (2011). *Conflict-Free Replicated Data Types*, pp. 386–400. Berlin, Heidelberg: Springer.

van Renesse, R. and F. B. Schneider (2004). Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, San Francisco, CA, pp. 7–7.