
**André Miguel Augusto Gonçalves**

Licenciado em Engenharia Informática

# Estimating Data Divergence in Cloud Computing Storage Systems

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Nuno Manuel Ribeiro Preguiça,
Professor Auxiliar,
Universidade Nova de Lisboa

Co-orientador : Rodrigo Seromenho Miragaia Rodrigues,
Professor Associado,
Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor João Seco

Arguente: Prof. Doutor Luís Veiga

Vogal: Prof. Doutor Nuno Preguiça


FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**Setembro, 2013**

**Estimating Data Divergence in Cloud Computing Storage Systems**

*À Milu.*

# Acknowledgements

I would like to thank my advisor, professor Nuno Preguiça, not only for the opportunity to develop this thesis, but also for all the availability, help and support in seeing it to good end. Thanks also to my co-advisor, professor Rodrigo Rodrigues, for the chance for this work to become a part of the project "Towards the dependable cloud: Building the foundations for tomorrow's dependable cloud computing". This work was partially funded by project PTDC/EIA-EIA/108963/2008 and by an ERC Starting Grant, Agreement Number 307732, to which I would like to thank. Many thanks also to Valter Balegas for all the help and suggestions. Thanks to my college, Faculdade de Ciências e Tecnologia, especially Departamento de Informática, my department, which has become my second home in these last five years. Thanks to my colleague friends who traveled this path with me, particularly João Silva, Lara Luís, Helder Martins, Laura Oliveira, Joana Roque and Diogo Sousa, for all the support, ideas, discussions, and most of all, companionship, during this last year.

I would also like to thank my "brother from another mother" Pedro Cardoso, for always believing in me and sticking with me through thick and thin since always. Thanks also to my academic goddaughters, friends, and great listeners, Rita Pereira and Andreia Santos, who supported me since the very beginning of this thesis. Special thanks to my group of friends, Pedro Almeida, Tiago Chá, Ricardo Neto, Pedro Verdelho, Flávio Moreira, Sasha Mojoodi, Gonçalo Homem, Sandro Nunes, Daniela Freire, Carlos Realista and Diana Patacão, for the much needed breaks from work, support, patience, and friendship, providing me a great deal of encouragement to proceed with my ordeals.

Last but not least, major thanks to my parents, without whom I would not have been able to complete this journey and be the man I am today, I owe all of this to them both.

# Abstract

Many internet services are provided through cloud computing infrastructures that are composed of multiple data centers. To provide high availability and low latency, data is replicated in machines in different data centers, which introduces the complexity of guaranteeing that clients view data consistently. Data stores often opt for a relaxed approach to replication, guaranteeing only eventual consistency, since it improves latency of operations. However, this may lead to replicas having different values for the same data.

One solution to control the divergence of data in eventually consistent systems is the usage of metrics that measure how stale data is for a replica. In the past, several algorithms have been proposed to estimate the value of these metrics in a deterministic way. An alternative solution is to rely on probabilistic metrics that estimate divergence with a certain degree of certainty. This relaxes the need to contact all replicas while still providing a relatively accurate measurement.

In this work we designed and implemented a solution to estimate the divergence of data in eventually consistent data stores, that scale to many replicas by allowing client-side caching. Measuring the divergence when there is a large number of clients calls for the development of new algorithms that provide probabilistic guarantees. Additionally, unlike previous works, we intend to focus on measuring the divergence relative to a state that can lead to the violation of application invariants.

**Keywords:** cloud computing, geo-replication, eventual consistency, bounded divergence, probabilistic metrics

# Resumo

Muitos serviços na internet são fornecidos através de infraestruturas de *cloud computing*, que são compostas por múltiplos centros de dados. Os serviços na internet dependem de sistemas de armazenamento que são implantados nestas infraestruturas. Para providenciar alta disponibilidade e baixa latência, os dados são replicados em máquinas em diferentes centros de dados, o que introduz a complexidade de garantir que os clientes veem os dados de forma consistente. Os sistemas de armazenamento optam frequentemente por uma abordagem relaxada à replicação, garantindo consistência fraca, já que esta melhora a latência das operações. No entando, isto pode fazer com que réplicas fiquem com valores diferentes para os mesmos dados.

Uma solução para controlar a divergência dos dados em sistemas fracamente consistentes é o uso de métricas que medem o quão desatualizados estão os dados numa réplica. No passado, vários algoritmos foram propostos para estimar o valor destas métricas de forma determinista. Uma solução alternativa é usar métricas probabilísticas que estimam a divergência com um certo grau de incerteza. Isto relaxa a necessidade de contactar todas as réplicas, mas ainda providenciando uma medida de divergência relativamente precisa.

Neste trabalho desenhou-se e implementou-se uma solução para estimar a divergência de dados em sistemas de armazenamento de consistência fraca, que escalam para muitas réplicas por permitirem *caching* do lado do cliente. Medir a divergência quando existe um elevado número de clientes exige o desenvolvimento de novos algoritmos que forneçam garantias probabilísticas. Adicionalmente, e ao contrário de outros trabalhos, nós queremos focar-nos em medir a divergência relativa a um estado que possa dar aso a que invariantes da aplicação sejam quebradas.

**Palavras-chave:** cloud computing, geo-replicação, consistência eventual, divergência limitada, métricas probabilísticas

# Contents

# List of Figures

# List of Tables

# Listings

# 1

# Introduction

## 1.1 Context

The Internet has become a popular way to provide several services, from file storage and sharing, to collaborative editing and social networking. These services are usually supported by global computing infrastructures, often called *cloud computing* infrastructures. Many of these services have become a big part of people's lives, being used in a daily basis in all kinds of devices, from desktop computers, to mobile phones, tablets and laptops, and often at the same time.

An important component of cloud computing infrastructures is the storage system to store application data. These systems take care of the complexity of storing and managing data in a large number of distributed computing resources. Applications only worry about storing data and retrieving it.

As such, these cloud storage systems must guarantee that the data is accessible, that the system can scale to many users/large amounts of data, and can keep working despite software and hardware failures. Due to all of this, data in this kind of distributed systems is often *replicated* across several servers, which are often very far apart, in different data centers, in different countries and even continents. This geo-replication, however, is transparent to the user.

With data being replicated in several locations, when clients update data in one location, other locations have to be updated with the new value, or else they become stale. This propagation of updates may be done synchronously to all replicas when data is changed, blocking the clients' access to data, or it may be done lazily, with updates being propagated asynchronously after the result is returned to the client. The first approach guarantees that the client sees the same data independently of the contacted replica, but

needs more communication, and therefore, increases the latency of the system. The latter approach eases the problem of latency, but may result in the client seeing data that is not up to date, because replicas may have different values for the same data, or updates may have not reached a certain replica yet. This problem is even more complex if the storage system allows clients to concurrently modify different replicas of the same object. In this case, concurrent updates need to be merged to guarantee that replicas evolve to the same state.

As the well-known the CAP theorem states, a distributed system cannot provide consistency, availability and partition tolerance at the same time, at most it can provide two of those properties [Bre00; GL02]. As partition tolerance cannot be avoided in cloud computing infrastructures, the choice in these storage systems is between consistency and availability.

Many distributed storage systems, like Amazon's Dynamo [DHJKLPSVV07] or Bayou [TTPDSH95], opt for the approach of letting replicas converge *eventually* in order to favor high availability. This allows users to access any replica for reads and writes without coordination with other replicas, making the system highly available. High availability is important because downtime decreases the usage and profitability of services [LM06; GDCCSXDSLLL12].

The divergence among data replicas is a problem in many popular systems where replicas converge only eventually. Replicas being updated with different values leads to *conflicts*, and late propagation of updates may cause clients to have a stale view of the system.

## 1.2 Motivation

Data management in cloud infrastructures is approached using different solutions. While some systems provide a strictly synchronous approach, like Google's Spanner [Cor+12], others provide a completely asynchronous replication model where replicas can be updated without coordination, like Sporc [FZFF10]. A number of systems lie in-between, providing client views that exhibit some degree of guarantees, for example: guaranteeing that all causal dependencies are observed [LFKA11], showing a consistent view of data at some given moment in time [SPAL11], or guaranteeing different synchronization mechanisms for different operations [BBCFKLLLLY11; LPCGPR12].

All systems that provide some form of relaxed consistency experience the problem of replica divergence. Several mechanisms have been proposed to provide guarantees about the data clients access. Session guarantees [TDPSTW94] guarantee that the client's view of the system evolves in a systematic way, even if not always completely consistent. Other techniques limit data divergence, by allowing clients to be able to make operations that will not conflict with any others for sure [PMCD03; STT08], or by establishing certain bounds in terms of the updates or staleness of observed data, by using metrics to measure how different replicas are [YV02; SVF07].

In the latter approach, it is necessary to define divergence metrics that can be used by a node to estimate the divergence between its local replica and the state of other replicas. The algorithms to bound these metrics are usually deterministic [YV02; SVF07], so if there is a given threshold for a metric, it is guaranteed that the metric is kept under that value. This often implies communication between many replicas. As such, it is unrealistic to use such algorithms for systems that scale to many replicas. Recently, techniques have been proposed to bound staleness in a probabilistic way, guaranteeing that *probably* the value of the metric is less than a given threshold [BVFHS12]. This relaxes the previous constraint of having to contact a large number of replicas.

Furthermore, these metrics are usually defined on the value of data maintained by applications – either using absolute or relative values. Although this approach is important, applications also need to know how close they are to break application invariants. In this case, it would be interesting to define divergence metrics that could directly estimate this situation. For instance, in an on-line shopping application where an account can have several billing sources, it would be useful to guarantee that the total of purchases does not go over a certain threshold. One system provides probabilistic guarantees that these invariants are kept [KHAK09]. However, these guarantees are provided at the data centers, by changing the consistency level of operations at runtime. There are no solutions yet that measure and bound the divergence of replicas with respect to application invariants.

## 1.3 Proposed Solution

The goal of this work was to design and implement a solution for estimating and bounding the divergence of data in cloud computing storage systems. In a scenario where a large number of clients can cache data and perform operations locally on that data, known algorithms to measure and bound this divergence cannot be used, because of the amount of communication required. With this in mind, we designed an approach where clients can estimate how divergent they are from the actual state of the object. Clients can also obtain probabilistic guarantees that no application invariants are broken when executing an operation locally.

This approach is based on divergence metrics. We used metrics defined in previous systems: a value metric, which measures how divergent in numeric value a replica is, and an operations metric, which measures how many operations made by other replicas have not been seen. In order to estimate the value of these metrics, statistics are gathered about the evolution of objects' values and number of operations executed. These statistics are used to predict how divergent the client is for each metric. Statistics are generated on the data centers, and propagated to the clients. The estimate has an associated certainty degree. Bounds are placed on these metrics, and when the client estimates have low confidence that the bounds are kept, coordination is started with the data center to obtain the latest updates and reduce its divergence.

Our approach allows mapping application invariants to metrics' bounds. In this way, clients can represent application invariants as bounds for the metrics, and guarantee probabilistically that invariants are kept, as long as the bounds are respected.

The approach is defined in generic terms, and can be realized in different ways. We present a model that realizes this approach, which we call the rhythmic model. This model gathers statistics as a simple rate of how the object grows, and uses it in combination with a Poisson distribution to obtain the certainty degree of the estimates, and to estimate the probability that the bounds/invariants are kept. While simple to understand, this model can be implemented with low time complexity, and provides accurate estimates. Another possible way of realizing this approach is by using advanced forecasting models [MWH08] to predict the state of the objects.

We present a design of the system as a middleware composed of two components that interact with one another: a generator component, on the data center's side, and an estimator component, on the clients. These components are placed between the application and storage layers. On a new update in the server, the generator updates the statistics about the object. Periodically, the generator component sends statistics to the estimator. When the client performs operations, the estimator uses the statistics to check if bounds are kept. If no coordination is needed, the operation is executed locally, otherwise the estimator tells the client the operation cannot be done locally.

In order to access these components, we defined an API. The API allows several operations, such as:

- setting the metric bounds or invariants for a certain object;
- check if an operation can cause an invariant to be broken;
- adding new statistics about an object;

To validate our approach, we implemented it in a simulator of a distributed system. A simulator allows us to obtain the real divergence of a replica, which is not obtainable in a real system. This is important to test the precision of our estimates. Besides precision, we also wished to test how our approach avoids communication while allowing operations to be executed at clients. We evaluated our rhythmic model, as well as an implementation with the advanced forecasting techniques.

## 1.4 Contributions

The contributions of this work are:

- A metric-based approach for estimating and bounding the divergence of data in eventually consistent storage systems;
- A method for probabilistically guaranteeing application invariants are kept;

- A model that realizes this approach, representing the evolution of objects as growth rates;
- A proof-of-concept implementation of this approach, with the defined model and other forecasting techniques;
- An evaluation of the precision of the estimates using a standard benchmark, TPC-W.

## 1.5 Organization

This report is organized as follows: chapter 2 presents the current work done in the field, introducing replication techniques, consistency levels, eventually consistent systems' guarantees and bounding divergence techniques, as well as providing an overview of data store systems that enforce several levels of consistency or that bound divergence in some way; in chapter 3 the full approach and system design are described; in chapter 4 the estimates' system implementation is detailed, and in chapter 5 an analysis on its behavior is made; chapter 6 sums up the conclusions we took from this work, and discusses future work.

# 2

# Related Work

## 2.1 Data Stores and Replication Basics

Cloud storage systems are data stores that are distributed, replicated through many storage hosts (frequently called *nodes*). As such, many of the basic mechanisms and concepts they employ come from data stores. This section introduces such concepts, as well as some specifics from distributed stores and a brief overview of replication.

### Data Stores

**Types** There are two main types of data stores: *Relational Database Management Systems* (RDBMSs), and *NoSQL* ("not only SQL"). RDBMSs are database systems that are based on the popular relational model [Cod70]. In this model, information is represented in terms of tuples, which are related through relations. These systems commonly use the Structured Query Language (SQL) [CB74] to manage data, or a language with similar semantics. NoSQL data stores are stores that do not employ the relational model. The most common model used by these stores is the key-value model, which is a simple model where data has a key that identifies it, and is stored as a value [DHJKLPSVV07; LFKA11]. Data is managed with two simple operations that work with keys, one for accessing data, and one to update it. RDBMSs are more expressive, but representing data is also more complex, while NoSQL stores present a simple model, but less expressiveness. Some systems present a hybrid between both approaches, in what is called a semi-relational model [BBCFKLLLLY11; Cor+12].

**Operations** There are two kinds of operations, independently of the store's type: *write* operations, that add or change data in the system, and *read* operations, that access stored data. In key-value stores these are generally called *put* and *get*, respectively.

**Transactions** A *transaction* is a sequence of operations that is *atomic* (either all operations complete sequentially with success, or none of them have any effect on the data), *consistent* (changes the data in a correct way), *isolated* (it must appear that a transaction either happened before or after another) and *durable* (after a transaction completes, its changes to the data are permanent). These properties are referred to as ACID semantics. If a transaction completes, it is *committed*. If it failed, it is *aborted*.

**Timestamps** To order operations (or transactions) and detect concurrency, *timestamps* are used. However, in a distributed system, clocks cannot be synchronized, so operations cannot be stamped with the replica's clock. A common solution to detect concurrency between operations is the usage of a *version vector* for each object [PPRSWWCEKK83]. Each replica has a vector, with the last version of the object it saw from all other replicas. When a replica updates the object, it increments its own version in the vector. When replicas synchronize, the vectors are exchanged and compared. If more than one replica changed its version, the operations made are concurrent.

**Quorums** Some distributed stores use *quorum systems* to serve reads and writes. For each object, there is a quorum system, composed of $N$ replicas [Vog09]. $R$ replicas are the *read quorum*, and $W$ replicas are the *write quorum*. When a read is made, all replicas in the read quorum are contacted, and the most recent value among replicas is returned. When a write is made, the operation is sent to the replicas in the write quorum, and only when all $W$ acknowledge the write, is the write considered successful. In *strict quorum* systems, $W + R > N$, which means the read and write quorums overlap, guaranteeing reads see the most recent value [BVFHS12]. Systems where $W + R \leq N$ are called *partial quorum* systems.

When operations are performed in a quorum system, the replicas in the quorum must reach an agreement in what the operation returns to the client. Paxos is the most used algorithm that solves this consensus problem [Lam98]. Paxos is frequently used to implement state-machines [BBCFKLLLLY11; Cor+12].

### Replication

In a distributed data store, data is replicated among many nodes to guarantee fault-tolerance. There are different replication models for each dimension of replication systems [SS05].

**Coordination** Replication can be synchronous or asynchronous. Synchronous (or pessimistic) replication makes replicas coordinate to get the same actual data, and implies

some communication overhead. In asynchronous (or optimistic) replication, updates are propagated in the background, not needing *a priori* synchronization, relaxing the constraints of conflict handling.

**Update types** Data can be propagated to other replicas in two ways: either by sending the state at which the object is, or by sending the last operations made to the object. If the object is larger than the operations' log, sending the state requires larger messages, or more messages to send the whole object. However, propagating operations may be more complex for the receiving replicas, that have to apply the operations to their local copy.

## 2.2 Consistency Levels

An important aspect in a replicated system is the level of consistency it provides to clients. For supporting fault-tolerance and low latency, it is common that the replicas of the system diverge temporarily. It is up to the system's algorithms to determine how the state of the replicas are updated and what is the state that can be observed by the clients. Keeping this in mind, systems may provide different *consistency levels*, that is, a system may provide stronger consistency guarantees, providing clients with a consistent view of the object at all times, or may relax those guarantees, causing the view to be stale or inconsistent at times. Stronger models of consistency require coordination for the execution of operations, which can lead to delays or aborted operations. Weaker models allow replicas to diverge, but require the system to include a conflict-resolution mechanism to guarantee the eventual convergence of all replicas.

This section presents the main consistency models usually provided by distributed systems.

### 2.2.1 Strong

Strong consistency, also known as single-copy consistency, is the consistency level that provides a semantic similar to a system with a single replica. A system is strongly consistent if clients always see the current state of an object, as if there is only one server, serializing the operations. Frequently, the access to an object is blocked until all replicas are at the same state for that object.

There are two common strong consistency semantics: *linearizability* [Lam86] and *serializability* (also referred to as *sequential consistency*) [Lam79]. Originally these concepts were in regard to shared memory registers, but were generalized to arbitrary shared objects by Herlihy and Wing [HW90] and are explained next, as seen in [CDKB11].

#### Linearizability

A replicated system is linearizable if for any execution there is some interleaving of all operations issued by clients that satisfies the following criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

The second part of the definition makes linearizability the ideal model in an ideal world. The real-time guarantee makes operations made by clients consistent with the single correct copy of the object as they occur. The problem with this is that clock synchronization is often not as precise as desired, making this consistency level hard to provide.

#### Serializability

A replicated system is serializable (or sequentially consistent) if for any execution there is some interleaving of all operations issued by clients that satisfies the following criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

Sequential consistency grants a relaxation in terms of the interleaving. Lightening the real-time property makes it a weaker model, but as the order of clients' executions is kept, the view the client obtains is always consistent with the idea of a single correct copy of the objects.

### 2.2.2 Per-key Sequential

Per-key sequential consistency is a further relaxation of sequential consistency. This model guarantees serializability at an object-level, that is, instead of a global ordering of all operations, there is an order for the operations of each object. This means each replica applies the same updates in the same order for that object [CRSSBJPWY08].

This level guarantees that a client always views an object at a consistent state, but the global ordering of operations across all objects may not be consistent with the order in which each individual client executed them, and the state observed across a set of objects might not be consistent.

### 2.2.3 Snapshot Isolation

Snapshot isolation is a database isolation level. This level guarantees that a transaction will read a consistent state, that corresponds to the values that were in the database up to the start of the transaction [BBGMOO95]. Two transactions conflict if both write the same data element. Figure 2.1 represents this in a simple way.

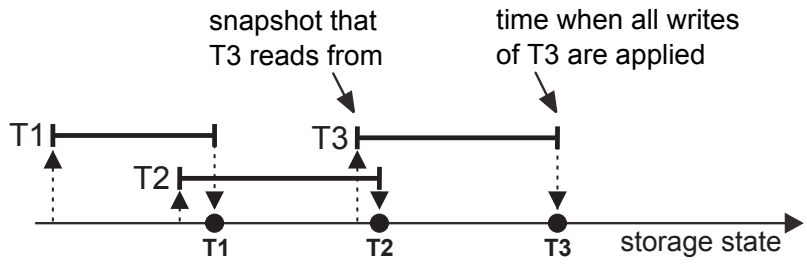



Figure 2.1: Snapshot isolation example. The writes of T1 are seen by T3, but not by T2, since T2 reads from a snapshot prior to T1's commit (Taken from [SPAL11])

Many geo-replicated data stores provide the guarantee that a client may access a consistent view of the system as it was in a specific time in the past, or for a certain version of the object (sometimes called *snapshot reads*) [BBCFKLLLLY11; SPAL11; Cor+12]. Therefore, snapshot isolation (or consistency), is frequently regarded as a consistency level [CRSSBJPWY08; LPCGPR12; BVFHS12] for this kind of systems. This consistency level is used, because it is simple to implement in a system that allows access to multiple versions of an object. Coupled with this advantage is the fact that read transactions never block or abort.

### 2.2.4 Causal

Causal consistency is a consistency level that guarantees only that operations are executed always after all operations on which they causally depend. This is achieved by ensuring a partial ordering between dependent operations [ANBKH95]. The view of the system clients have, when performing a read, is consistent with all the writes that were made on that object prior to the execution of the read. This model is weaker than sequential consistency because different clients may disagree on the relative ordering of concurrent writes.

To properly represent dependency relations between operations, Lamport defined the *happens-before* relation that captures the potential causality relation that can be established in a distributed system [Lam78]. The happens-before relation adapted to a storage system can be defined using the following three rules:

1. Execution Thread. If $a$ and $b$ are two operations in a single thread of execution, then $a \rightsquigarrow b$ if operation $a$ happens before operation $b$.

2. Gets From. If $a$ is a put operation and $b$ is a get operation that returns the value written by $a$, then $a \rightsquigarrow b$.

3. Transitivity. For operations $a$, $b$, and $c$, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

At a replica, the order by which the operations are executed is called the execution history. An execution history is causally consistent if any operation is executed always after the operations on which it causally depends.

Figure 2.2 exemplifies how causal consistency works. For the client to have a causally consistent view of the system, it must appear that `put`(y,2) happened before `put`(x,4), that happened before `put`(z,5). If client 2 saw `get`(z)=5, and then `get`(x)=1, causal consistency would be violated.

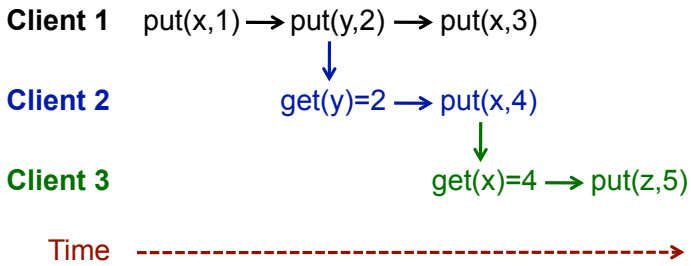



Figure 2.2: Graph showing the causal relationship between operations at a replica. An edge from $a$ to $b$ indicates that $a \rightsquigarrow b$, or $b$ depends on $a$ (Taken from [LFKA11])

### 2.2.5 Eventual

Eventual consistency is the weakest consistency level. There is no formal definition of it in terms of operations, like the previously introduced levels. A system is eventually consistent if it guarantees that the state of replicas will eventually converge. As such, a client may have an inconsistent or stale view of data at times.

A big difference from stronger consistency levels, is that in those levels some replica may be outdated in relation to other, because it has not got the most recent updates. In eventual consistency, two replicas may have two different values written concurrently, like in causal consistency. Because of this, causal consistency is sometimes considered a sub-level of eventual consistency.

Eventually consistent systems are relaxed in terms of concurrency control. The main mechanism associated with this kind of systems is optimistic replication, that does not block clients' access to data when it is inconsistent [SS05]. This allows for better availability and latency, since synchronization between replicas is not enforced *a priori*. This results in updates being propagated when they can, and conflict detection and resolution being handled afterwards.

## 2.3 Guarantees in Eventually Consistent Systems

It is possible to provide additional guarantees, other that *the state of a replica will eventually converge*, in eventual consistent systems. A system may guarantee certain properties about the result of operations (session guarantees), or may impose different consistency requirements for different objects, with the goal of limiting "how much" replicas can diverge (bounded divergence). This section presents such techniques.

### 2.3.1 Session Guarantees

*Session guarantees* were created to alleviate the lack of guarantees in ordering the read and write operations in a replicated database providing only eventual consistency, while maintaining the main advantages of read-any/write-any replication [TDPSTW94].

A *session* is an abstraction for the sequence of read and write operations performed during the execution of an application. The idea is to provide to each application a view of the database that is consistent with its own actions, even when executing reads and writes in possibly inconsistent servers.

With this in mind, *guarantees* are applied independently to the operations in a session. Four types of guarantees are proposed: *Read Your Writes, Monotonic Reads, Writes Follow Reads, Monotonic Writes*.

#### Read Your Writes

*Read Your Writes* guarantees that the effects of any writes made within a session are visible to reads that follow those writes in the same session. To implement this guarantee, reads are restricted to copies of the database that include all writes in that session. More formally, *If read R follows write W in a session and R is performed at server S at time t, then W is included in with DB(S,t), the state of the database in server S at moment t.*.

To provide this guarantee, each write's assigned Write ID (WID) is added to the session's write-set. Before each read to server S, at time t, the session manager must check that the write-set is a subset of DB(S,t). A more practical way to explain this, is that each write has a timestamp, which the client knows. When accessing a server, if the server's version vector has that version or a more recent one, then this guarantee is provided.

#### Monotonic Reads

Monotonic Reads guarantees that read operations are made only to database copies containing all writes whose effects were seen by previous reads within the session. More formally, if: *If read R1 occurs before R2 in a session and R1 accesses server S1 at time t1 and R2 accesses server S2 at time t2, then RelevantWrites(S1,t1,R1) is a subset of DB(S2,t2)*. (RelevantWrites is the smallest set that is enough to completely determine the result of R).

To provide this guarantee, before each read the server must ensure that the read-set is a subset of DB(S,t), and after each read R to server S, the WIDs for each write in RelevantWrites(S,t,R) should be added to the session's read-set.

#### Writes Follow Reads

Writes Follow Reads guarantees that traditional write/read dependencies are preserved in the ordering of writes at all servers. More formally, *If read R1 precedes write W2 in a session and R1 is performed at server S1 at time t1, then, for any server S2, if W2 is in DB(S2)*

*then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S2), and WriteOrder(W1,W2).* (WriteOrder(W1,W2) means W1 is ordered before W2 in DB(S) for any server S that has received both W1 and W2).

Providing this guarantee requires servers to constrain their behaviours so that a write is accepted only if is ordered after all other writes in the server, and that anti-entropy is performed in a way that the write order is kept the same in both servers. To provide this guarantee, each read R to server S results in RelevantWrites(S,t,R) being added to the read-set, and before each write to server S at time t, the session manager checks that this read-set is a subset of DB(S,t).

#### Monotonic Writes

Monotonic Writes guarantees that writes must follow previous writes within the session. More formally, *If write W1 precedes write W2 is in a session, then for any server S2, if W2 in DB(S2) then W1 is also in DB(S2) and WriteOrder(W1,W2).*

To provide this guarantee, DB(S,t) must include the session's write-set, for the server to accept a write at time t. Additionally, whenever a write is accepted by a server, its assigned WID is added to the write-set.

### 2.3.2 Bounded Divergence

In systems that use eventual consistency, it may be interesting that some objects are more up to date than others. Also, clients may perform operations on some objects, while being disconnected from a server, which results in higher divergence between replicas. In such systems, an interesting approach is to bound the divergence between replicas, with regard to some property of the system.

There are two basic approaches in limiting divergence between replicas:

- Computing the divergence between replicas, through the use of metrics. The divergence can then be bounded, by guaranteeing that the value for each metric does not vary more than a certain threshold.

- Clients can obtain reservations that allow them to guarantee that certain conditions will hold when all updates are merged. When connected, clients ask for a reservation for a given object. They can then perform operations on the object when disconnected with the guarantee that some constraints will hold.

#### Metrics

Systems that use this approach, rely typically on three metrics to measure divergence: one for the order of operations, one for the value of data, and one for staleness [YV02; SVF07]. For each metric, the application can specify the limits of divergence.

- *Order of updates* refers to the number of updates that were not applied to a replica. The higher the value of this metric, the more updates from other replicas may be seen out of order.
- *Value* refers to the difference in the contents of an object between replicas, or when compared to a constant. The higher the value, the largest can be the difference from the observed value and the correct value computed after merging all unreceived updates.
- *Staleness* refers to the maximum time a replica may be without being refreshed with the latest value. In other words, it limits the staleness of data.

Bounding each metric must be performed with a specific algorithm. Bounding these metrics to 0 makes the system guarantee strong consistency, since no updates are lost, objects are always at their freshest value at all replicas, and data is never stale, so clients always get a consistent view of the system. Reversely, bounding them to infinity, grants full relaxed consistency.

#### Reservations

Reservations are used in systems where clients have a need to perform operations while disconnected from the server. Reservations are leased locks for some operations on a given object that clients can obtain while connected to the server [PMCD03]. These reservations guarantee *a priori* that the result of the operations made will be successful for the client who obtained them, avoiding the need for conflict resolution. Reservations are typically used in systems with relational databases [PMCD03; STT08], hence some of them being directly related to relational database elements (like rows and columns).

There are several types of reservations, each providing different guarantees:

- *Value-change* reservations provide the exclusive right to modify the state of an existing data item (i.e., a subset of columns in some row).
- *Slot* reservations provide the exclusive right to insert/remove/modify rows with some given values.
- *Value-use* reservations provide the right to use a given value for some data item (despite its current value).
- *Escrow* reservations provide the exclusive right to use a share of a partitionable resource represented by a numerical data item (or *fragmentable object* [WC95]).
- *Shared value-change* reservations guarantee that it is possible to modify the state of an existing data item (i.e., a subset of columns in some row).
- *Shared slot* reservation provide the guarantee that it is possible to insert/remove/-modify rows with some given values.

When a client obtains a reservation, the server blocks accesses from other clients that would conflict with the granted reservation. The client may then perform the operation (while disconnected) on its cached copy of the database. When it reconnects, the operations are executed against the server database, with the guarantee that no conflicts will arise with other clients' operations.

## 2.4 Systems that Enforce Consistency Levels

This section presents some storage systems that provide one or more consistency levels to the applications they support.

### 2.4.1 PNUTS

PNUTS is a massive-scale hosted database system to support Yahoo!'s web applications [CRSSBJPWY08]. Its focus is on serving data to web applications, which require usually only simple reads, rather than complex queries. It has a simple relational model, with flexible schemas, provides a facility for bulk loading of updates, and supports a publish-subscribe system to propagate updates asynchronously. Record-level mastering is used since not all reads need the data at its most current version, which provides per-key sequential consistency. The API supports three levels of read operations (read any version; read latest version; read a version that is equal or more recent than a given version) and two levels of writes (*blind write*, where a write may overwrite a concurrent write; and *test and write*, where a write fails if a concurrent write has been performed).

#### Architecture/Implementation

PNUTS is divided into *regions*, which contain a full set of system components and a complete copy of each table. Regions are usually geographically distributed. Data tables are partitioned into groups of records (tablets), that are scattered across servers. In the event of a failure, information is recovered by copying from another tablet. The scatter-gather engine receives a multi-record request, splits it into individual requests (by tablet or record) and initiates them in parallel.

To implement record-level mastering, for each record there is a master replica, usually the replica getting the most requests for that record. All requests for that record are forwarded to the master. Publish-subscribing is done through Yahoo! Message Broker (YMB). All data updates are published to YMB, then propagated asynchronously to replicas.

#### Strong Aspects

PNUTS record level mastering allows to place the master replica of each object close to the users that modify them, providing low latency for reads and writes. Additionally, the

support for different levels of consistency in read and write operations allows to trade consistency by performance, depending on the specific requirements of each application.

### 2.4.2 Megastore

Megastore is a semi-relational database designed by Google [BBCFKLLLLY11]. It takes the middle-ground between NoSQL and RDBMS to provide scalability while keeping high availability and consistency. This is done by partitioning the data store and replicating each partition, by providing full ACID semantics within partitions, but limited consistency guarantees across them.

In terms of availability, a synchronous, fault-tolerant system to replicate logs was modeled, that uses Paxos to replicate a write-ahead log over a group of symmetric peers. To allow for better availability, several replicated logs are used, each governing its own partition of the data set. For partition and locality, *entity groups* are used. An entity group is a partition of the data, that is synchronously replicated over a wide area. The underlying data is stored in a NoSQL data store. Within the entity group, entities are mutated with single-phase ACID transactions(stored in the replicated log). Across groups, communication is done asynchronously through a queue. This architecture is shown in figure 2.3.

#### Architecture/Implementation

Megastore's data model is relational, based on entities, and is backed by a NoSQL data store. Reads are completed locally, there's a coordinator server that checks if a replica has committed all Paxos writes needed for that read in that entity group(that can be current, snapshot or inconsistent). Writes begin by reading, to obtain the last timestamp and log position. Then writes are gathered into a log entry, and Paxos is used to append that entry to the replicated log.

There are 3 kinds of replicas: full, read-only (store entities and logs, but do not vote in Paxos), and witnesses (only back up logs).

#### Strong Aspects

Megastore's usage of replicas and partitioning of data and logs makes for a very reliable system in terms of availability and fault-tolerance. It is also scalable due to providing weak consistency across entity groups, but remains synchronous in its log-replication. Fast reads and writes are also a performance plus.

### 2.4.3 COPS

COPS is a distributed key-value store that provides causal+ consistency [LFKA11]. Causal+ consistency is causal consistency, with convergent conflict handling. This means that conflicts are handled in the same way at all replicas.

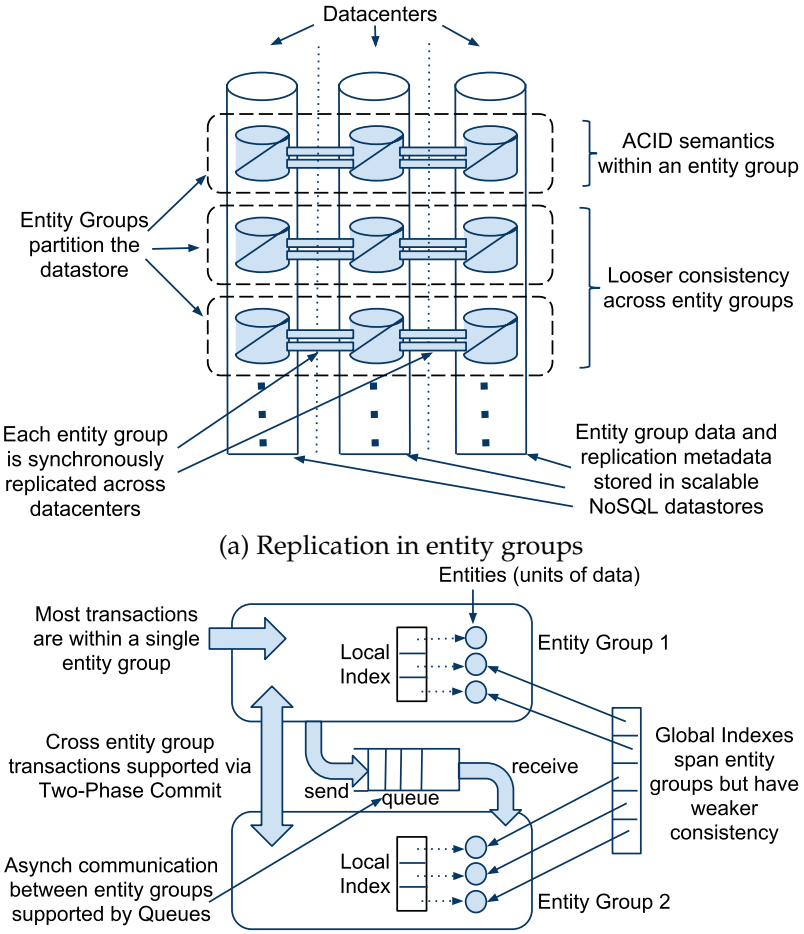



Figure 2.3: Megastore's architecture (Taken from [BBCFKLLLLY11])

In a key-value store such as COPS, two operations exist, a get and a put. Two concurrent put operations are in conflict if they write to the same key. COPS comes in two versions, one that only supports regular get and put operations, and one with support for get-transactions, which allow for several values to be retrieved at once, while guaranteeing a consistent view. This requires for dependencies being stored as metadata, so that the returned snapshot of values is also causal+ consistent.

### Architecture/Implementation

In COPS, each data center completely replicates all data in a set of nodes.. Each node in a cluster is a multi-version key-value store. Each node is responsible for different partitions of the keyspace. When keys are replicated remotely, dependency checks are made at the destination before committing the incoming version. Every key stored in COPS has a primary node in a cluster, and primary nodes on other clusters are equivalent nodes. When a write completes locally, it is placed in a replication queue to be sent asynchronously to remote equivalent nodes.

Gets are made to the node responsible for the key. Puts require dependency checking,

and when they are propagated, dependencies are checked at the remote replica before committing. Each version of an object maintains its dependency set – the dependency set contains all version of objects read before the given version was written. On a write, when a conflict is detected, an appropriate conflict resolution function is called – by default: last-writer-wins. Old versions and no longer used dependencies can be garbage-collected.

#### Strong Aspects

This system provides causal consistency. This guarantee is strong enough that data can be asynchronously replicated, while being read fast. This provides latency scalability.

### 2.4.4 Walter

Walter is a geo-replicated key-value store that uses transactions to guarantee Parallel Snapshot Isolation (PSI) [SPAL11]. PSI extends snapshot isolation to distributed environments, by allowing different sites to have different commit orderings, but preserving causal ordering at all sites.

There are two mechanisms in Walter to enforce PSI. First, each object in the system is assigned to a preferred site. In this site, the object can be committed without checking other sites for write-write conflicts. Second, there is a special kind of object, *conflict-free counting set* (cset), that keeps elements, and a count for each element. An add increments that element's count, and a remove decrements it. Operations on these elements are commutative, so there are no write-write conflicts.

#### Architecture/Implementation

Walter is logically organized in containers. All objects in a container have the same preferred site. This allows for organization in terms of what objects may benefit from being seen consistently by other objects faster.

Walter supports transactions that include a sequence of read and write operations. Transactions are propagated asynchronously.

Each transaction is assigned a version number, to order transactions in a site. A snapshot is implemented by a vector timestamp, that indicates for each site how many transactions are reflected in the snapshot. Each time a transaction is executed, it is saved in a history variable, per object, with the version of the transaction. Reading an object returns the most recent version of that object at the site at which it is being read.

A commit can be fast, or slow. Fast commits are for objects in a local preferred site or for transactions accessing only csets, that are not in a slow commit protocol. A fast commit can complete without contacting other replicas. A slow commit is one to a non-preferred site, and is performed using a two-phase commit protocol.

#### Strong Aspects

With Walter, applications do not have to deal with conflict-resolution logic, since there are no write-write conflicts. PSI PSI is stronger than eventual consistency, providing the guarantees of snapshot isolation for a single client, with no dirty reads, lost updates and non-repeatable reads.

### 2.4.5 Spanner

Spanner is Google's state of the art distributed database [Cor+12]. Spanner is scalable, multi-version, globally-distributed and synchronously-replicated. Replication configurations for data can be controlled at a fine grain by applications, and data can be dynamically and transparently moved between data centers to balance data center usage. Data is replicated using Paxos. It provides semi-relational tables, with an SQL-like language, and data is versioned automatically with a commit timestamp. These timestamps reflect serialization order and are globally-meaningful, which allows for externally-consistent reads and writes, and globally-consistent reads.

To allow for these consistency properties, an API was developed called TrueTime. This API exposes clock uncertainty, so timestamps are assigned accordingly to provide the mentioned guarantees. The implementation of the API uses GPS and atomic clocks as references, in order to keep uncertainty small.

#### Architecture/Implementation

Spanner is composed by *spanservers*, that replicate *tablets* (bags of mappings). For each tablet, a Paxos state machine is implemented, with long-lived leaders. Writes initiate the Paxos protocol, while reads access state directly from the replica if it is up-to-date. Replicas keep a value that is the maximum timestamp at which a replica is up-to-date, so any read with a timestamp older than that value can be satisfied.

The TrueTime API returns the current time. To achieve this, each data center has a set of time masters, which have either GPS antennas or atomic clocks. Masters' time references are regularly compared to each other to synchronize, and each machine polls several masters so that errors from one master are reduced.

#### Strong Aspects

The strong aspects of Spanner are that it is highly-scalable, has good performance despite being synchronously-replicated, and that supports externally-consistent transactions. The usage of globally-unique timestamps is the main point that permits all this, both for usage in the Paxos protocol and to allow local reads.

### 2.4.6 Gemini

Gemini is a storage system that provides RedBlue consistency [LPCGPR12]. RedBlue (RB) consistency defines two types of operations: red and blue. Red operations are operations that must follow a serialization order, and require immediate cross-site coordination. Blue operations are operations that can be propagated asynchronously, so their position in the serialization order has no impact on the final result. Gemini can be applied on top of any data store. Operations in application logic are divided in generator operations, and shadow operations. Generator operations calculate the changes the original operation should make, but have no side effects, and shadow operations perform the changes at the replicas. Shadow operations are colored red or blue, depending on their semantics.

#### Architecture/Implementation

The Gemini storage system was implemented to provide RB consistency. Each site is composed by four components: a storage engine, a proxy server, a concurrency coordinator, and a data writer. The proxy server is where requests are issued. A generator operation is is executed in a temporary scratchpad, and operations are executed against a temporary table. The proxy server then sends the generated shadow operation to the concurrency coordinator, who accepts or rejects the operation, based on RB consistency. The data writer then executes the operation against the storage.

Gemini uses timestamps (logical clocks) to determine if operations can complete successfully. These are checked when a generator completes, to determine if the operation reads a coherent system snapshot and obeys the ordering constraints.

As a performance optimization, some blue operations may be marked read-only. These operations are not incorporated into the local serialization or global order, but the proxy is notified of the acceptance.

#### Strong Aspects

The strongest aspect of Gemini and RedBlue consistency is that it allows for different consistency levels. Red operations require strong consistency guarantees, while blue operations can manage with eventual consistency. Gemini is flexible in the sense that it can use any data store, and can support applications with different level of consistency requirements.

### 2.4.7 Comparison

These systems provide different operations and replication mechanisms to guarantee some consistency level. Table 2.1 presents an overview of the systems' characteristics.

Many of these systems use optimistic replication, while providing consistency levels that are stronger than eventual consistency. However, this does not guarantee that

replicas are always consistent. Most of them also provide some kind of read that accesses older versions of data. This allows clients to use the kind of read most adequate to application's latency requirements. This shows that it would be interesting to have a mechanism to measure just how divergent replicas are.

| System | System Type | Consistency Level | Replication Technique | Operation Types |
|---|---|---|---|---|
| PNUTS | Relational Database | Per-key Serializability | Lazy | Multi-record updates, Different level of consistency reads |
| Megastore | Semi-relational Database | Serializability/ Snapshot/ Eventual | Paxos/Lazy | Current, Snapshot or Inconsistent reads |
| COPS | Key-value Store | Causal+ | Lazy | Puts, Gets, Multi-value gets |
| Walter | Key-value Store | Parallel Snapshot/ Eventual | Lazy | Non-conflicting writes, (Parallel) Snapshot reads |
| Spanner | Semi-relational Database | Linearizability | Paxos | Globally-consistent reads, Writes |
| Gemini | Relational Database | RedBlue | Cross-site/Lazy | Reads, Writes |

Table 2.1: Comparison of features between systems

## 2.5 Bounded Divergence Systems

As presented in the previous section, most cloud storage system do not provide strong consistency to provide low latency of operations. In this case, clients may access data values that do not reflect updates that have been performed in other replicas. Several systems have been proposed to bound the divergence among replicas or to provide guarantees over the data that is accessed. Most of these systems have been proposed in the context of mobile systems, where a replica could stay offline for long periods of time. This section reviews the most important of these proposals.

### 2.5.1 TACT

TACT is a middleware system that bounds the rate of inconsistent accesses to an underlying data store [YV02]. TACT mediates read/write accesses to the data store, based on consistency requirements, executing them locally if no constraints are violated, or waiting to contact other remote replicas for synchronization.

TACT relies on a basic data abstraction, a *conit*, and on a set of metrics. Conits are used to specify consistency requirements, and the metrics are bounded in order to achieve those requirements.

Each application defines the granularity of its conits and what metrics to bound according to the applications' semantics and their consistency needs.

#### Architecture/Implementation

In TACT, a conit is a set or partition of the data, a unit whose consistency level is enforced at each replica, instead of existing a system-wide policy. Each conit at a replica has a logical time clock associated with the last updates seen from each replica.

The defined metrics are *Numerical Error* (NE), *Order Error* (OE), and *Staleness*. Numerical error limits the total weight of writes that can be applied across all replicas before being propagated to a given replica. Order error limits the number of tentative writes (subject to reordering) that can be outstanding at any one replica, and staleness places a real-time bound on the delay of write propagation among replicas. Bounding all metrics to 0 guarantees strong consistency, while bounding to infinity provides optimistic replication only.

Bounding NE is achieved by pushing updates to other replicas and OE by pulling updates from other replicas. The decision on what to push or pull is based only on the state of the replica they are running on. Staleness is bound using real-time vectors.

#### Strong Aspects

TACT provides a "continuous" consistency model, by allowing applications to choose the kind of consistency they need for certain sets of data. This is versatile in the sense that different kinds of data (modeled by different conits) can be replicated in different ways by bounding the metrics differently.

### 2.5.2 Mobisnap

Mobisnap is a middleware that extends a client/server SQL database system [PMCD03]. A single server maintains an official database state, and mobile clients cache and work with snapshots of the database. They then submit small SQL programs, which are propagated against the official state.

Besides having a conflict resolution strategy, Mobisnap prevents conflicts with *reservations*. Clients obtain reservations for a certain period of time, and when any local transaction is executed, these are checked. If the client has enough reservations to guarantee no conflict when re-executing in the server, the result of the transaction can be considered definite. When a client obtains a reservation, the server must guarantee that no transaction from another client violates the promise made to the first one.

#### Architecture/Implementation

Each client maintains two copies of the state, a tentative and a committed one. The tentative version contains the state after executing all disconnected transactions. The committed one contains the state after executing transactions guaranteed by reservations.

There are several types of reservations: value-change, slot, value-use, escrow, shared value-change and shared slot. When a reservation is requested, the server checks if it is possible to grant the reservation. After granting it, triggers are added to the database to prevent values being changed, or to prevent writes that would conflict with the reservation.

When a transaction is run, the client checks if it has reservations that guarantee some (or all) of the operations. When it is ran against the server's database, if it was not guaranteed on the client, traditional conflict detection/resolution is applied.

#### Strong Aspects

Mobisnap's usage of reservations provides a strong way of limiting data divergence, since clients that are expecting to update some partition of the data can guarantee *a priori* that no other client's transactions will interfere. As such, clients keep an updated view of the partition/values of the store they have reservations for.

### 2.5.3 Exo-Leasing

Exo-leasing is a system that combines escrow with disconnected reservation transfer between clients [STT08]. This allows all the benefits of using fragmentable (escrow) objects and reservations while reducing the need to communicate with a server to get the reservations.

Escrow objects are stored in the server database instead of regular objects, and represent fragmentable objects. Clients have these objects cached, and operations are applied on top of them. These objects have two commutative operations, *split* and *merge*. Split makes a reservation for a certain part of the object, while merge returns a certain amount. They also have a *reconciler log*, which records operations made. Reservations are requested and a lease is associated to them. If the client does not reconnect in time, the reservation is not valid anymore.

Reservations can be split and transferred between clients. This means a disconnected client may obtain reservations from another client, without server interaction.

#### Architecture/Implementation

Exo-leasing needs two kinds of transactions: *top-level* and *base (open-nested)*. Top-level transactions contain the latter, and execute operations on cached objects. These transaction effects become permanent when committed to the server. Base transactions are

revertible within the top-level ones, in the sense that they are tentative when there is intention to commit, and are run on local copies of the objects at a server, after which they become durable.

To obtain a reservation for a certain object, a client may ask the server for it, or another client that has that reservation. To do this, both clients just change their reconciler logs, noting a new merge operation for each of them, and the amount of the object each one has. The server object's reconciler log is updated when one of them connects, and both reservations have the old one's lease time.

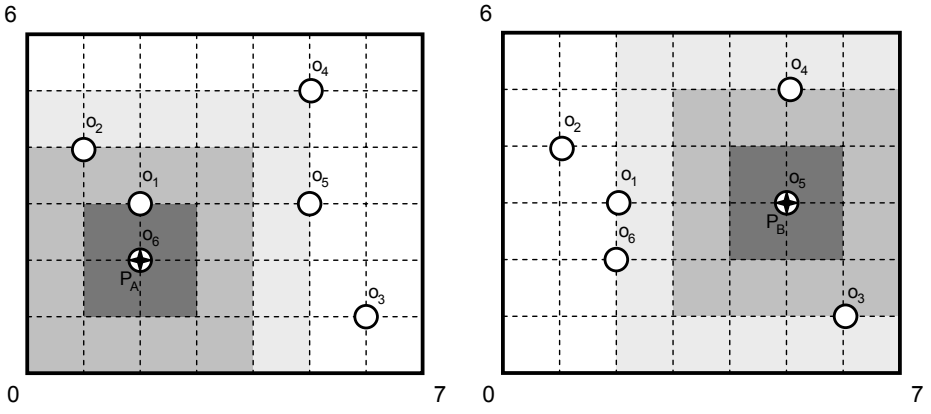
#### Strong Aspects

The great advantage of this system is that it allows bounded divergence by using reservations (like Mobisnap), with the property of reservations being transferable between clients. Clients keep a consistent view of the part of the objects they have the reservations to update, while being able to get new reservations without server interaction.

### 2.5.4 Mobihoc

Mobihoc is a middleware implemented to support the design of multiplayer distributed games for ad-hoc networks, and provides Vector-Field Consistency (VFC) [SVF07]. It has a client-server architecture, when the network is established, one of the nodes becomes the server. The server coordinates write-locks, propagation of updates and VFC enforcement, and may also act as a client.

VFC is an optimistic consistency model that allows bounded divergence of the object replicas. VFC selectively and dynamically strengthens or weakens replica consistency based on the ongoing game state. Each object has a view of the system, and for that view, it is a pivot object. The consistency degree is then stronger for objects in a certain range, and grows weaker the "further" an object is from the pivot.




(a) View for replica A (b) View for replica B

Figure 2.4: Different views of the system for different replicas (Taken from [SVF07])

#### Architecture/Implementation

In VFC, consistency degrees are defined as a vector. These vectors have three dimensions (as in TACT): *time*, that specifies the maximum time a replica can be without being refreshed with the latest value, *sequence*, that specifies the maximum number of lost replica updates, and *value*, which specifies the maximum relative difference between replica contents, or against a constant. This vector is the maximum divergence allowed for objects in that view.

In Mobihoc, each node keeps local replicas of all objects. The server has the primary copy of objects. Reads are done locally without locking, writes need to acquire locks to prevent loss of updates. Periodically, the server starts rounds. Updates are piggybacked in round messages, and merged at the clients.

The Consistency Management Block (CMB) at the server enforces VFC. First, clients register the objects to be shared, and send their consistency parameters, in a setup phase. In an active phase, clients may access the objects, the server processes write requests that were sent asynchronously, and round events. To support VFC, consistency views per each client are maintained, meaning that updates from replicas that are close are propagated to the client, while updates from replicas far away are not, saving communication.

#### Strong Aspects

Mobihoc presents an interesting consistency model to bound divergence. The idea of dynamically changing consistency degrees for certain objects based on distance is useful because it allows different objects to have different consistency degrees, while keeping an up-to-date view (according to its requirements) for each client. The advantage of this consistency model is that it lowers traffic in the network.

### 2.5.5 Probabilistically Bounded Staleness for Practical Partial Quorums

Probabilistically Bounded Staleness (PBS) describes the consistency provided by eventually consistent data stores, in partial quorum systems [BVFHS12]. PBS presents three metrics: $k$-staleness, that bounds the staleness of versions returned by read quorums; $t$-visibility, that bounds the time before a committed version appears to readers; and $\langle k, t \rangle$-staleness, a combination of both $k$ and $t$-staleness. These metrics are *probabilistic*, which means that they do not guarantee that staleness is limited at some bounds, but instead provide staleness bounds with varying degrees of certainty.

These metrics are defined mathematically in terms of the number of replicas in the quorum system. They are defined non-mathematically as follows:

- An object is bounded with $k$-staleness if, with probability $1 - p_{sk}$, at least one value in any read quorum has been committed within $k$ versions of the latest committed version when the read began.

- An object is bounded with $t$-visibility if, with probability $1 - p_{st}$, any read quorum started at least $t$ units of time after a write commits, returns at least one value that is at least as recent as that write.

#### Architecture/Implementation

Only the $t$-visibility metric was tested, by simulation with a latency model. The predicted $t$-visibility was compared with the simulated latency. Then $t$-visibility was predicted for production systems and compared with the observed latency. $k$-staleness was only analyzed in closed-form.

#### Strong Aspects

The strongest aspect of this approach is that these metrics relax the need to guarantee the bounds. Providing a degree of certainty to which staleness is within certain bounds allows for a "close-enough" estimate of staleness in a system.

### 2.5.6 Consistency Rationing

Consistency rationing is a technique for adapting the consistency requirements of applications at runtime. The goal of the system is to reduce the total monetary cost of storage requests. The price of a particular consistency level can be measured in terms of the number of service calls needed to enforce it.

This system defines three data categories, with different consistency requirements. Category A, serializable, is the strongest and the most expensive, requiring additional services to assure data consistency. Category C provides session guarantees – read your writes – within a session, and is used if the savings compensate the expected cost of inconsistency. The B category comprises data that can be processed as C or A depending on the context.

Five policies are presented to adapt the consistency of data. The *general* policy is based on conflict probability, which is determined by the transactions' arrival rate. The cost of inconsistency is obtained with the probability of having conflicting updates. The *time* policy changes the consistency when a timestamp is reached. The *fixed threshold* policy, allows setting a threshold which forces the system to handle the record with strong consistency when the update exceeds that limit. This allows the invariant to be broken if there is more than one update in different servers exceeding the threshold. The *demarcation* policy prevents this by assigning a portion of the value to each replica and allowing the replica to update it up to that value without synchronization, like escrow techniques. If an update requires more than that portion, then the operation must be executed with strong consistency or request the portion from other replica. Finally, the *dynamic* policy for numeric objects adjusts the threshold according to the probability of updates exceeding it.

#### Architecture/Implementation

The system architecture is composed by clients that communicate with the application servers that run inside the cloud on top of Amazon's Elastic Computing Cloud (EC2). Application servers cache data and buffer updates before sending them to the storage system. A statistical component gathers statistics about the objects. Evaluation has shown a cost reduction and performance boost, with the dynamic policy being the most effective in terms of cost and response time.

#### Strong Aspects

This system's strongest aspect is that it can provide probabilistic guarantees about invariants being kept and conflicts being avoided, by switching to different consistency levels at runtime. This permits less communication between servers.

### 2.5.7 Comparison

These systems bound divergence in different ways. The reservation-based systems bound divergence by providing a way for operations to complete without conflicts in disconnected operation. The metrics-based systems guarantee that divergence is never more than a certain threshold, and PBS provides a way to estimate how divergent replicas are. Consistency rationing enforces stronger consistency levels when invariants would be broken by concurrent updates. Table 2.2 presents a comparison between these systems.

Reservations guarantee deterministically that invariants are not broken. However, as the system requires clients to obtain reservation before being able to execute operations, in a system with a large number of clients it might be impossible to assign reservations to all clients, rendering the system unusable. The metrics presented in TACT and Mobihoc are useful, however, as the algorithms used to estimate their value are deterministic, they are not usable in a system with a large amount of replicas. The idea of PBS provides a good solution, but the metrics must be adapted to different dimensions of the objects besides staleness. Consistency rationing provides a good approach for changing the consistency levels at runtime in order to probabilistically keep the invariants, but is focused only on the server side of the applications.

| System | Technique | Conflict Avoidance | Client-side Caching | Scalable |
|---|---|---|---|---|
| Mobisnap | Reservations | Yes | Yes | Yes |
| Exo-leasing | Reservations/ Escrow | Yes | Yes | Yes |
| TACT | Deterministic Metrics | No | No | No |
| Mobihoc | Deterministic Metrics | No | Yes | No |
| PBS | Probabilistic Metrics | No | N/A | Yes |
| Consistency Rationing | Probabilistic Policies | Yes | No | Yes |

Table 2.2: Comparison of features between bounded divergence systems

# 3

# Estimating and Bounding the Divergence

In this chapter, we explain in detail our method for estimating and bounding the divergence in a cloud computing storage system, as well as our approach to these estimates, the rhythmic model. We start off by detailing further the model of systems we took into consideration for designing this approach, as well as how the estimates can be used in such systems. We then describe the metrics used, and how their value is estimated and used for bounding the divergence. Finally, the architecture of such a system is discussed.

## 3.1 System Model

In this work we consider a cloud storage system with two replication levels. At the first level, there is a small number of data centers (less than ten), with a database fully replicated in each data center. Data centers communicate periodically or asynchronously between them to exchange updates. We assume that client and server's clocks are loosely synchronized.

At a second level, a large number of clients (tens or hundreds) replicate only small subsets of the data. Clients execute operations on their local copies of the objects, and submit them to a data center. This submission may be done in batches of updates, when the clients perform a series of operations offline and then synchronize with the server, or asynchronously, when each operation completes on the client. This architecture is shown in figure 3.1.

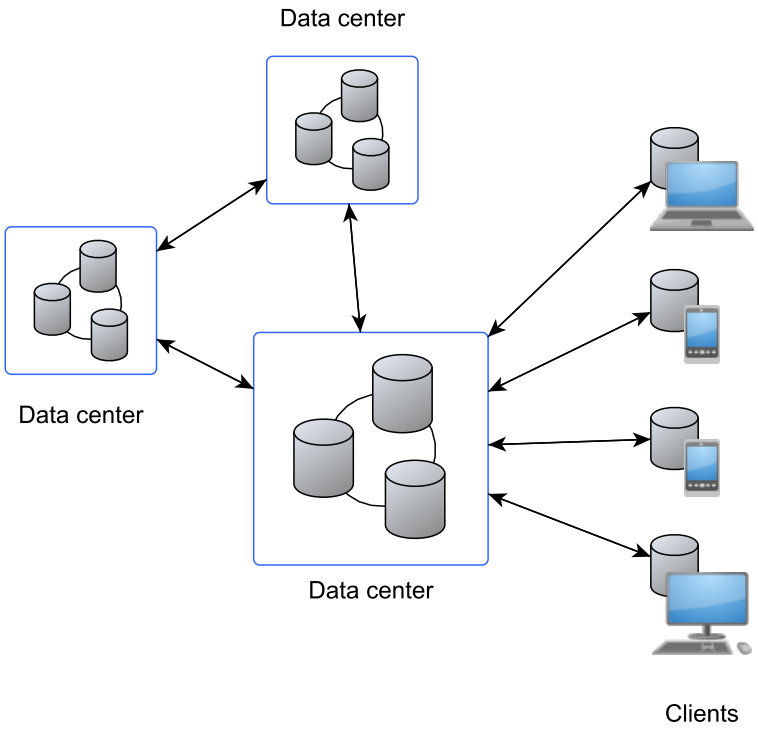



Figure 3.1: Distributed system model with client-side caching of data

The above mentioned database is assumed to be a key-value store, where stored objects are Conflict-free Replicated Data Types (CRDTs) [SPBZ11]. These data types guarantee that replicas converge to the same state, independently of the order in which operations are applied at each replica. This mechanism relaxes the need for conflict-resolution techniques when propagating to the server the operations made locally on a client replica. Note that the approach for estimating divergence is the same for any kind of distributed store and data types. We keep this assumption in mind because of the system implementation described in chapter 4.

### 3.1.1 Adding the Estimates

The developed approach estimates the divergence of client replicas, and the probability of an operation violating application invariants. The base idea is to gather statistics from the updates replicas have seen over the course of the application's execution. These statistics about the objects, combined with probabilistic methods, are used to predict an object's abstract state, the *single-copy* of the object. The single-copy (from "single-copy consistency", as seen in section 2.2.1) is the object's state that reflects all updates from all replicas applied in serialization order. In an eventually consistent system, this state is not deterministically obtainable by a replica, that is why our system relies on predicting this state.

Applications may have certain integrity constraints (also called application invariants

– in the text we use both terms as synonyms) that have to be kept.

Based on the single-copy estimate, the client-side of the application checks if an application invariant is possibly violated when an operation is made. The client may then choose to not execute the operation, to contact the server, or to execute it locally anyway.

By estimating how divergent from the single-copy a client is, that replica is able to perform operations locally, with a certain degree of certainty that integrity constraints are respected. This relaxes the need for synchronizing with the server on every operation. To represent the estimates our approach relies on metrics, and to represent application invariants, bounds are placed on those metrics' values. These metrics, and how they are used by the system, are explained in the following section.

To support the approach, we developed a model based on a simple rate of updates. We call this the rhythmic model. Throughout this chapter, when we explain how the estimates and bounds are used, we will also detail how the calculations are made with this rhythmic model, and what formulas are needed. Note that besides this model, clients can also use more advanced forecasting techniques [MWH08], with the trade-off of extra computing time and storage space. These forecasting models were used in our implementation and will be discussed later.

## 3.2 The Metrics

We are interested in estimating how divergent a replica is, and using that information to limit communication, while keeping the divergence under certain limits to avoid breaking application invariants. To this end, like TACT and Mobihoc [YV02; SVF07], our system uses metrics to measure how divergent a replica is from the single-copy. Different metrics measure and bound divergence on different dimensions of the object. As such, we are interested in the same metrics as these approaches: a *value* metric and an *operations* metric.[1]

### The Value Metric

The value metric measures how divergent the replica is in terms of the object's value, this is, the numerical difference between the value the replica has, and the real value of the object, in the single-copy. This metric is semantically equal to TACT's numeric error metric.

To better understand the metric's behaviour, consider the following example. An integer, X, is replicated across several nodes. In these replicas, operations (increments and decrements) are made on the local copies of X, and propagated asynchronously to

[1] In these systems, that have a deterministic approach to the calculation of metrics' values, there is usually a third metric. The staleness metric measures how long a replica has been without being updated with the most actual state of the object. This metric is usually used to place a real-time bound on write propagation, to guarantee that a replica does not go over a certain time limit without being updated. As such, from an estimate point of view, there is no real contribution to make, so the system depicted in this dissertation does not use this metric.

the other replicas. These operations are identified by the replica who made the operation, and a timestamp of when the operation was made. Figure 3.2 represents the state of two replicas, A and B, as well as the single-copy state for that object, with all operations serialized and the real value of the object.

In this example, replica A has X=10, because it has not seen update (B,15), that decrements X by 3. As such, its value is 3 units off the real value, 7. So the value metric measures 3 for replica A. The same rationale is used for replica B, which has X=9, so the metric measures 2.

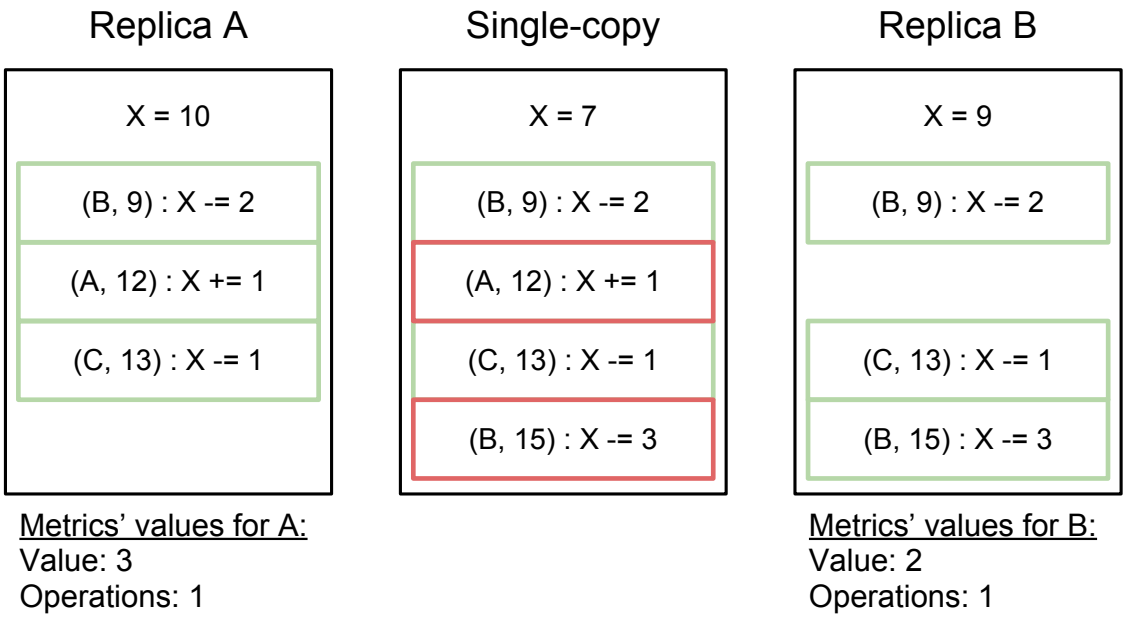



Figure 3.2: Example of the metrics behavior. Represented are two replicas of an integer object, as well as the single-copy for the object

### The Operations Metric

The operations metric measures how many operations made by other replicas are not applied to the current one. Consider the previous example. Both replicas measure 1 on the operations metric, because both of them are missing one operation. Update (B,15) has not been applied on replica A, and update (A,13) has not been applied on replica B. Since both replicas saw all other updates, they only miss one each.

Note that this metric is semantically different from TACT's order error metric in the sense that we do not consider a total ordering of updates in our system model. As such, our metric does not measure the amount of pending, out of order operations, but the ones that have not been seen by the replica.

## 3.3 Estimating the Divergence

Having defined the metrics, we must detail how their values are calculated. Other systems use deterministic algorithms that involve contacting all replicas in some way, to measure accurately the value of a metric. In a cloud computing system, with up to tens or hundreds of clients that cache data, these approaches are not feasible. The idea is to collect statistics about the object's evolution, from the updates made. For a large number of clients, the aggregation of operations leads to a steady rhythm of updates, which allows predictions to be made about the state of the object.

To estimate how divergent the replicated object is from the single-copy, we need the statistic of how the object changes as time passes. If a model representing the object's evolution is built, the replica can calculate how much the object would have grown in the time interval where the client did not communicate with the server. With the last known state of the object, and the estimate of how much it grew, a replica can then predict the single-copy state of the object, and how divergent its own state is. As such, this model is generated on the server-side, where all the operations are propagated to, and then disseminated to the clients, along with the data center's current object state.

As stated earlier, the divergence between a client's copy and the ideal single-copy is impossible to obtain deterministically on a large scale geo-replicated system, so our goal is to attain an estimate of the divergence that is as close as possible to the real divergence. Estimating the single-copy's state is needed in order to know how divergent the client replica is.

Estimating the single-copy's state consists in calculating how much the object changed in the time passed since the client last obtained a copy from the server. This change, $\eth$ is the difference between the single-copy's state and that last seen state, so it represents how much the object diverged in that period, and is estimated with the evolution model of the object. Adding this estimated divergence to the last seen state results in the estimated single-copy state.

It must be noted, however, that the data center may not have the most updated state of an object. In other words, the data center may have some divergence of its own, and with this procedure that divergence is being overlooked, so the client assumes that the state it obtains from the server is the single-copy's state at that point in time. As such, the data center must include a statistic about how divergent its copy itself may be in the object's state it sends to the client.

For an estimate to be useful, one must analyse how strong it is. In other words, we must know just how much confidence we can have on the estimated state being close to the actual single-copy state. For that, we calculate the certainty degree associated with the estimate. In this case, the certainty degree is the probability that the state varied in $\eth$.

### 3.3.1 Estimating With the Rhythmic Model

In our rhythmic model, the evolution of an object is seen simply as a rhythm of updates, represented mathematically as a rate. This *growth rate*, represented in the following formulas by the Greek symbol $\lambda_{obj}$, is calculated differently for each metric.

#### 3.3.1.1 Value Metric

For the value metric, the growth rate expresses how much the value of the object varies by unit of time. Having $\Delta_{Val}$ as the numeric variation of the object, and $\Delta_{Time}$ the interval of time where that numeric variation occurred, the growth rate for an object's value is calculated as follows:

$$\lambda_{objVal} = \frac{\Delta_{Val}}{\Delta_{Time}} = \frac{V_f - V_i}{T_f - Ti} \tag{3.1}$$

where $T_i$ and $T_f$ are the timestamps of the beginning and end (respectively) of $\Delta_{Time}$, and $V_x$ the object's value at timestamp $T_x$.

Using these variations allows us to obtain an average variation of the value by time unit.

#### 3.3.1.2 Operations Metric

For the operations metric, the rate expresses how many operations are made by unit of time, in the whole system. As such, its calculation is a simple mean:

$$\lambda_{objOps} = \frac{n}{\Delta_{Time}} \tag{3.2}$$

where $n$ is the total number of operations made during the interval of time $\Delta_{Time}$.

This provides an average of how many operations/updates were made to the object, by unit of time, on all replicas.

#### 3.3.1.3 The Growth Rate's Role on the Estimate

As $\lambda_{obj}$ is a time-based rate, $\eth$ is obtained by multiplying the rate for each unit of time that passed since the client last obtained a fresh copy from the server: $\eth = \lambda_{obj} \times \Delta_{Time}$. Adding this growth to the last state seen from a data center, yields the estimated state of the single-copy. Notice that this estimated state has different meanings for each dimension of the object. From a value's perspective, this state is the object's value, and $\eth$ is how much it changed over time. For the operations metric, this state is the set of all operations made on the object, and the $\eth$ is how many operations were made in that time period.

#### 3.3.1.4 The Data Center's Divergence

The rationale for obtaining the data center's divergence is similar to the one to obtain $\eth$: a rate is obtained, and multiplied by a time value. The idea is to obtain a rate of how

divergent the data center replica is by time unit, and multiply it by the average time an update takes to reach the data center.

For the value metric, this rate is obtained as follows: being $T^u_{diff}$ the time an update $u$ takes to reach the data center after being made at a client, and $\Delta^u_{Val}$ the change $u$ causes on the object, the rate is obtained by summing all of those changes, and dividing it by the sum of all $T^u_{diff}$:

$$\lambda_{DC} = \frac{\sum_{u=1}^{n} \Delta^u_{Val}}{\sum_{u=1}^{n} T^u_{diff}} \tag{3.3}$$

For the operations metric, the equation is the same, but replacing $\Delta^u_{Val}$ by 1, since it is the change an update makes in the total number of operations. So, for both metrics, the data center's divergence is obtained by multiplying the rate by the average time the data center takes to see updates, $\overline{T_{diff}}$:

$$\eth_{DC} = \lambda_{DC} \times \overline{T_{diff}} \tag{3.4}$$

#### 3.3.1.5 The Certainty Degree

As explained earlier, the certainty degree of the estimate is the probability of the single-copy's state varying $\eth$ in a certain period of time. To obtain this probability, we resort to the Poisson distribution [Hai67].

The Poisson distribution is exactly what is needed to obtain this certainty degree with this approach, because it expresses the probability of a given number of events occurring in a fixed interval of time, if these events occur with a known average rate and independently of the time passed between each other. Operations on the objects happen independently of one another, and we have an average rate of how the object grows, as well as the time passed since the last time a fresh state was observed, which makes this distribution useful.

Being the number of events that happen on a given time period represented by the random variable $X$, that follows a Poisson distribution with an average of $\lambda$ events per time period, the probability of having $k$ events trigger in that time period is:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad k \in \mathbb{N}_+,\ \lambda > 0 \tag{3.5}$$

In our case, the time period is the time since the replica last saw a fresh state, $\Delta_{Time}$. To get an average for that time period, we multiply the average growth rate by the time, so: $\lambda = \lambda_{obj} \times \Delta_{Time} \Leftrightarrow \lambda = \eth$. Since we want to know the probability of the object having a growth of $\eth$, we want to calculate $P(X = \eth)$. So both parameters' value is $\eth$: $\lambda = k = \eth$.

There are instantly two limitations with this formula. Its rate, $\lambda$, must be positive, because a negative number of events cannot happen (and zero means no events ever happen). Also, the Poisson distribution is a discrete distribution, so $k$ must be a positive integer.

There can be negative growth rates, which lead to negative growths (for example, more decrements than increments to a value), which would mean $\lambda$ being negative. However, having a negative growth or a positive one, in absolute terms, is the same for divergence. For example, if a numeric item remotely suffers a decrement by 2, or an increment by 2, either way it is divergent by 2 until that operations applies at the replica. Therefore, the probability that the object has grown 2 or -2 is the same. So to tackle this, we simply use the absolute value of the estimated growth as parameter: $\lambda = k = |\eth|$.

The second limitation is trickier, because there may be non-integer numeric values being stored in the system (for example, while the stock of an item of an online store is an integer, its price may be a decimal number), but the factorial in the formula cannot be used with non-integers. The literature usually solves this by making the Poisson distribution continuous, by replacing the factorial in the formula by the Gamma function of $k - 1$ [Ili13]:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{\Gamma(k+1)}, \quad k \in \mathbb{R}_+,\ \lambda > 0 \tag{3.6}$$

## 3.4 Bounding the Divergence

To limit how divergent a replica gets, bounds are placed on the values the metrics can reach. When the metric's value reaches that bound, coordination between that replica and the server is started. While the data center's copy's state may not equal the single-copy, all updates are eventually propagated to the data center, and contacting all other clients to obtain all pending updates is not feasible. Also, the data center's divergence estimate allows the client to obtain a state that is closer to the single-copy.

As the estimate has an associated certainty degree, the bound itself may not be deterministically guaranteed, since the estimate is used to enforce it. Because of this, when the bound is set, the degree of confidence with which the bound is assured must also be set. The higher this degree, the less probable it is that the bound has been broken, but there will be forced coordination more often, especially for low divergence bounds, when coordination already happens more frequently. In contrast, for higher bounds, coordination will be scarcer, so a higher confidence degree will increase coordination mostly when the bound is close to being broken.

Too bound the client's divergence under a certain limit, $lim$, we resort to the estimated divergence of the single-copy, $\eth$. Put simply, if the single-copy's state diverged more than $lim$, then the bound is broken. As such, we want to guarantee that, with confidence $\zeta_{lim}$, the single-copy's growth was less than or equal to $lim$.

The confidence level $\zeta$, that $\eth$ is below or equal to $lim$, is the sum of the certainty degrees of all values between 0 and $lim$. This is because the probability of $\eth \leq lim$ is the probability of the single-copy having not diverged at all (that is, having diverged 0), or

having diverged in any value up to, and including $lim$. Therefore:

$$\zeta = P(X \leq k) = \sum_{k=0}^{lim} P(X = k), \quad k \in \mathbb{N}_+ \tag{3.7}$$

If $\zeta \geq \zeta_{lim}$, then the bound is estimated to be kept.

### 3.4.1 Bounding With the Rhythmic Model

With our model, to estimate if the bound is kept with confidence level $\zeta$, one must only calculate $P(X \leq k)$ using the Poisson distribution. This is made by simply using expression 3.5 to obtain $P(X = k)$ inside the sum in equation 3.7.

This is valid for the value metric for integer objects, and for the operations metric, since both of these metrics' values are integers (there are no "half" operations). In the case of the value metric for a non-integer number, there are infinite values of divergence between 0 and $lim$. As such, instead of a summation, $\zeta$ is calculated using an integral. This alters expression 3.7 to:

$$\zeta = P(X \leq k) = \int_0^{lim} P(X = k)\,\mathrm{d}k, \quad k \in \mathbb{R}_+ \tag{3.8}$$

where P must be calculated using the Gamma-function-based expression, as seen in equation 3.6.

### 3.4.2 Preserving Integrity Constraints

To guarantee that an integrity constraint is kept, we must assure that the single-copy has not diverged to a state where the integrity constraint is not kept. To this end, it is necessary to define the maximum possible growth the single-copy can achieve without breaking the invariant, $\eth_{inv}$. Keeping this in mind, this problem is reduced to checking if a divergence bound of $\eth_{inv}$ is kept with a certain degree of confidence, $\zeta_{inv}$, as seen in the previous section. If that bound has not been broken, then the single-copy is preserving that integrity constraint.

Assuming the rhythmic model is used, let's consider the example of the online store, adding the constraint that an item's stock may not reach negative values. This constraint is of utmost importance for the store, since a client cannot purchase an item that the store cannot deliver, so the degree of confidence of this estimate must be high, let's assume 95% (so $\zeta_{inv} = 0.95$). A client saw 3 seconds ago that a certain item's stock was 10, with a growth rate of -1 per second (so $\lambda = k = |\eth| = |-1 \times 3| = 3$). In this case, $\eth_{inv} = 10$, since the maximum the stock could be decremented without breaking the invariant is 10, reaching a stock of 0, but not a negative one. To certify this, we must calculate the degree of confidence, $\zeta$, that assures $\eth \leq \eth_{inv}$. Using expression 3.7 with $lim = 10$ and $\lambda = k = 3$, we obtain $\zeta = 0.9997$. Since $\zeta \geq \zeta_{inv}$, the integrity constraint is kept with the desired degree of confidence.

However the procedure just described only estimates if the constraint has been kept since the last coordination. In a real system setting, the interesting estimate to make is if a certain operation executed locally at a client will break the invariant. To estimate this, the same method is used, but $\eth_{inv}$ must take into consideration the effect the pending operation will have on the single-copy's state. For this, $\eth_{inv}$ becomes the maximum possible growth the single-copy can achieve, minus the effect of the operation, because the client wants the guarantee that the operation's effect, coupled with what the single-copy has already diverged, still keeps the constraint in check.

Assume the previous example, but this time, the client wants to make this estimate when buying an amount of 2 of the item. $\eth_{inv}$ is now 8, because that is the maximum decrement that could have been made to the stock, so that a decrement of 2 still keeps it non-negative. This changes $lim$ to 8. Recalculating with expression 3.7, we get $\zeta = 0.9962$. Seeing that $\zeta \geq \zeta_{inv}$ is still true, the client may execute the operation locally, with the desired degree of confidence that the integrity constraint is kept. If the decrement amount were 5 instead (so $lim = 5$), the client would obtain $\zeta = 0.9161$, which makes $\zeta < \zeta_{inv}$. With this in mind, the client would not be able to execute the operation locally without contacting the server first to check for re-stocks.

Some applications may have integrity constraints of the form $upper \geq X \geq lower$, as opposed to the example we just saw, in the form $X \geq lower$. These are treated in the same way, however, the invariant must be checked to see what is the bound to keep. Take the above example, before we introduced the operation that decremented the value. Given the invariant, the bound, in terms of value, would be $lim = 10$, because that was the difference between the last seen state, and the state that broke the invariant. Consider that the invariant changed to "the stock must be positive, and less than 16", so $15 \geq stock \geq 0$. Since the objects evolve in one direction, either the $15 \geq stock$ or the $stock \geq 0$ may be at risk of being broken. Since the object had been evolving towards the value 0, the latter would be the one at risk. If the object had been evolving in the opposite direction, the bound to be kept would be $lim = 5$ (since the stock value was 10, we had to estimate the probability of it increasing in 5 units). With the rhythmic model in particular, this check is easily done, just by looking at the rate's sign. If it is negative, then the stock has been evolving towards 0, and the bound $lim = 10$ would have to be kept. If positive, then it was evolving towards 15, so the bound $lim = 5$ would have to be kept.

## 3.5 Architecture

The proposed system if provided as a middleware, to allow an easy integration on existing cloud storage systems. However, the system provides an API for the client-side of applications that may further customize how the system behaves.

The middleware is composed by two components. A generator component, on the data center's side, and an estimator component, on the client's side. These components are situated between the application and storage levels, and communicate with each

other. The components provide the underlying storage's interface to the application. This allows that existing applications are not altered just to use the middleware. The architecture described in this section is shown in figure 3.3, as well as the flow of execution of an operation: The application makes a request to the component, that implements the underlying database's interface (1); the component communicates with the database (2) and obtains a response (3); after executing its operations of generation/estimation/communication, the component replies to the application (4).

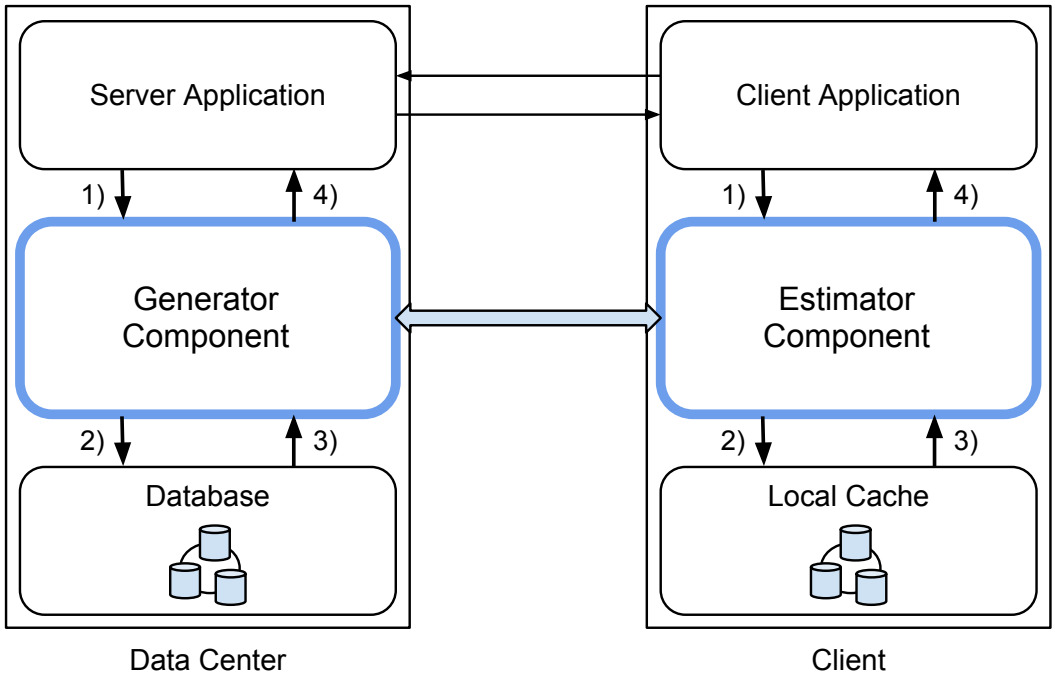



Figure 3.3: System architecture, composed by two components, a generator and an estimator

### 3.5.1 Generator Component

The generator component is responsible for generating the statistics about the objects, and storing them. When the server application calls a store operation (intended for the storage system, but the request is made on the generator component), the component stores the relevant information to generate the evolution model. This information is composed by the timestamps of the operations and their value (if the object is a number). The generator keeps a total of operations made for that object, as well as the total time passed since the object was introduced in the system. This information is stored as metadata in the underlying storage system. This component estimates how divergent the state of the object in the data center may be from the single-copy. In order to send statistics to the clients, the generator keeps a list of all the clients that keep a local copy of a certain object. If the generator component is gathering statistics about an object, we say the object is being *tracked*.

### 3.5.2 Estimator Component

The estimator component is responsible for using the data generated by the generator (the statistics, the last state received from the data center, and that state's divergence) to estimate the divergence of objects, and to ensure that bounds are kept. This component manages the different bounds for each object for that client, storing them in the underlying cache or database. The estimator component also keeps a list of operations made locally that have not been propagated to the data center yet.

When the client application invokes a read or write operation, the component checks if the bounds defined for that object's metrics are all kept, and if the operation can be performed locally.

### 3.5.3 Communication Between Components

Both components may at times communicate with each other, to update the statistics on the estimator component, or to obtain a fresher state of the object. This happens in two distinct occasions:

- Periodically, the generator component will broadcast the statistics, object state, and data center divergence of an object, to all the clients that have that object in their local cache. This allows the estimator component to update these statistics.

- On a possible bound violation. When this occurs, the estimator gets the statistics, state and divergence for that object, and if the bound break was estimated while performing an operation, that operation is executed (or aborted, if a bound is broken) after getting the most recent state, and its result sent to the estimator, which replies to the client.

### 3.5.4 The API

Despite being designed to easily integrate existing storage systems, the components offer an API so that applications (both clients and servers) may be written with this system's communication model in mind. This API allows choosing what objects should have their divergence estimated, setting the bounds and invariants for each metric for a certain object, which estimate model to use (rhythmic, forecasting, or a developer-defined one), and getting the estimates for the current time.

#### 3.5.4.1 Generator Component API:

**startTracking**(*objectID*) Starts gathering statistics for the object with key *objectID*.

**setCoordinationPeriod**(*timeInSeconds*) Changes the time between coordinations between data centers and clients.

**putUpdate**(*objectID*, *timestamp*, *state*) Adds a new update to the statistics about an object in the generator, that was executed at the client at time *timestamp*, and merges the state from that client with the data center's. The pseudo-code for this function is presented in algorithm 1.

**getStatistics**(*objectID*) Returns the statistics about the object with key *objectID* to be used by the estimator, as well as the current object's state from the data center's store. The pseudo-code for this function is presented in algorithm 2.

**Algorithm 1 putUpdate** pseudo-code

**Require:** $evolutionModels$: map with the evolution models of objects
**Require:** $store$: the data center's underlying key-value storage
1: **function** PUTUPDATE(*objectID*,*timestamp*, *state*)
2: $mergedState \leftarrow store.put(state)$ ▷ Merges the states
3: $objectModel \leftarrow evolutionModels.get(objectID)$
4: $objectModel.add(objectID, timestamp, mergedState)$
5: $evolutionModels.put(objectID, objectModel)$
6: $updateDataCenterDivergence(timestamp, mergedState)$
7: **return** $mergedState$
8: **end function**

**Algorithm 2 getStatistics** pseudo-code

**Require:** $evolutionModels$: map with the evolution models of objects
**Require:** $store$: the data center's underlying key-value storage
1: **function** GETSTATISTICS(*objectID*)
2: $objectModel \leftarrow evolutionModels.get(objectID)$
3: $data \leftarrow objectModel.get(objectID)$
4: $statistics \leftarrow generateStatistics(data)$ ▷ Generates statistics according to some estimate model (rhythmic, forecasting, etc...)
5: $state \leftarrow store.get(objectID)$
6: **return** $\langle statistics, state \rangle$
7: **end function**

#### 3.5.4.2 Estimator Component API:

**startLocalUsage**(*objectID*) Starts using statistics to decide when to coordinate for object *objectID*. The client-side equivalent of `startTracking`.

**setValueBound**(*objectID*, *newBound*, *confidenceDegree*) Sets the bound for the value metric for a certain object, along with desired the confidence degree. Overrides a previously set bound or invariant.

**setOperationsBound**(*objectID*, *newBound*, *confidenceDegree*) Same as the previous, for the operations metric.

**setValueInvariant**(*objectID*, *lowerBound*, *upperBound*, *confidenceDegree*) Sets or edits the invariant for an object. Overrides a previously set bound or invariant. In this case, the bounds are not in terms of divergence, but rather the values between which the single-copy's value must be.

**setOperationsInvariant**(*objectID*, *lowerBound*, *upperBound*, *confidenceDegree*) Same as the previous, but the bounds are divergence-wise, in terms of unseen operations.

**newStatistics**(*objectID*, *statistics*) Adds or replaces statistics about an object with key *objectID*, with more recent statistics obtained from the generator component. *statistics*. The pseudo-code for this function is presented in algorithm 3.

**getCoordination**(*objectID*, *update*) Verifies, using the estimates, if the object with key *objectID* requires communication with the data center. If called in the context of an operation, *update* indicates the change the operation is making on the object, to check if that change will break the invariant. Otherwise, *update* defaults to 0. The pseudo-code for this function is presented in algorithm 4.

A few notes about this pseudo-code: the auxiliary functions $getValueBound$ and $getOpsBound$ obtain the bounds to be kept, according to the defined invariants, as seen in section 3.4.2. The shown auxiliary functions $getValueConfidence$ and $getOpsConfidence$ use the state (with the update being made, to check if the update breaks the invariant) and the statistics to calculate the sum of certainty degrees of the divergence for a bound, and return the confidence level, as seen in section 3.4. These four functions' implementation depends on the estimate model being used. The return of this function is $false$ (no coordination needed) if both confidence levels are greater than or equal to the ones set by the application. Otherwise, $true$ is returned (coordination is needed).

**Algorithm 3 newStatistics** pseudo-code

**Require:** $objectsStatistics$: map with the objects' statistics and the last seen state

1: **function** NEWSTATISTICS(*objectID*, *statistics*, *state*)
2: $objectsStatistics.put(objectID, \langle statistics, state \rangle)$
3: **end function**

**Algorithm 4 getCoordination** pseudo-code

**Require:** $objectsStatistics$: map with the objects' statistics and the last seen state
**Require:** $vCLevel$: confidence level to keep the value bound
**Require:** $opCLevel$: confidence level to keep the operations bound

1: **function** GETCOORDINATION(*objectID, update*)
2: $\langle statistics, state \rangle \leftarrow objectsStatistics.get(objectID)$
3: $valueBound \leftarrow getValueBound(objectID)$
4: $opsBound \leftarrow getOpsBound(objectID)$
5: $stateUpdate \leftarrow state + update$
6: $estimatedVCLevel \leftarrow getValueConfidence(stateUpdate, statistics, valueBound)$
7: $estimatedOpCLevel \leftarrow getOpsConfidence(stateUpdate, statistics, opsBound)$
8: **if** $estimatedVCLevel \geq vCLevel \wedge estimatedOpClevel \geq opClevel$ **then**
9: **return** $false$
10: **else**
11: **return** $true$
12: **end if**
13: **end function**

# 4

# Implementation

This chapter presents an implementation of the system described in the previous chapter. This implementation had to be designed with two main goals in mind: to test the accuracy of the estimates, and to evaluate the approach's impact on the communication between clients and data centers. While the latter can be measured in a distributed system, the former cannot. This is because the actual single-copy state is needed to compare to the estimate, and this state is not obtainable at a given moment in an eventually-consistent system.

Taking this into consideration, our implementation is built on top of a simulator of a distributed system, instead of a real system. This allows us to keep a single-copy of the objects to compare with the estimates. As such, this chapter describes the simulator basics, the architecture of the system built on top of the simulator, and how the nodes communicate and make use of this work's approach. The entire project has been implemented in Java.

## 4.1 The Simulator

The simulator, named SimSim, is a Java project developed at FCT-UNL that simulates a distributed system. It allows the creation of nodes that send and receive messages between them. The nodes are created in the context of a *simulation*. A simulation uses its own virtual time (called *simulation time*) to schedule operations, instead of real time, and executes operations represented by *tasks*. A scheduler orders these tasks according to simulation time. Because of this, the simulator is single-threaded, to guarantee total ordering of tasks.

With this brief overview in mind, there are three main components to be used when

creating a system in the simulator: tasks, messages, and nodes. In coding terms, each of these components is a Java class.

**Tasks** Tasks are the main abstraction for running anything in the simulation. A task may be created at any point in the simulation, by a node, by another task, or by any other element (such as a static class). A task is created with a `due` parameter, which indicates in how many seconds (in simulation time) it must execute. If no `due` parameter is passed, the task is executed as soon as the current task ends. A second type of task, called periodic task, allows a second parameter, `period`. This parameter makes the task be executed repeatedly, every `period` seconds. Listings 4.1 and 4.2 exemplify the tasks' creation.

When a task is created, it is placed in a queue, and its place in the queue is defined according to when it is supposed to execute. The task's behavior is defined overriding a `run` method, that is executed automatically when due.

Listing 4.1: Task creation

```
1 new Task(0.5){
2     public void run(){
3         System.out.println("A task
              that executes after half
              a second has passed in
              simulation time");
4     }
5 };
```

Listing 4.2: Periodic task creation

```
1 new PeriodicTask(0.5, 2){
2     public void run(){
3         System.out.println("A task
              that executes after half
              a second and that is
              repeated every two
              seconds");
4     }
5 };
```

**Messages** Messages are the abstraction for communication between nodes within the simulation. Different message types may be introduced (by extending the abstract `Message` class), so that each different type is processed in a different way by the nodes. When defining a new message, the `deliverTo` method must be overridden for each kind of node that is expected to receive messages of that type. This method is called when the task that processes the message is executed, and is responsible for calling the node's methods necessary to process the message.

**Nodes** Nodes are the basic building block of the system. In essence, a node is a message handler. Each node may have its own clock (running on simulation time). A node also has an endpoint, with its own address, that other nodes use when sending messages to it. Similarly to messages, different nodes are created by extending an abstract class – `AbstractNode`.

Nodes have two methods for sending messages. They may send messages to other nodes in a one-way fashion, using the `udpSend` method, not expecting a response from the receiving node (simulating the UDP protocol). Nodes may also send a message and

wait for a reply (somewhat like TCP), with the `tcpSend` method. This method opens a channel that the sending node uses to wait for the reply. The endpoint of the destination node is necessary for sending a message with these methods. Listings 4.3 and 4.4 exemplify how the `udpSend` and `tcpSend` are used, respectively.

Listing 4.3: `udpSend` example

```
1 void udpExample(EndPoint destination, Message message){
2   // no extra delay
3   udpSend(destination, message);
4
5   // extra delay of 75ms
6   udpSend(destination, message, 0.075);
7 }
```

Listing 4.4: `tcpSend` example

```
1 void tcpExample(EndPoint destination, Message request){
2   // sends the request and creates a channel for the reply
3   TcpChannel channel = tcpSend(destination, request);
4
5   // blocks waiting for reply
6   Message reply = channel.tcpRead();
7 }
```

These methods create a task to process the message. The delay in message sending is introduced automatically by the simulator, but when sending a message, the node may introduce additional delay (to simulate lag between clients and data centers, for example). When the task to process the message is created, it takes this delay into account by delaying the task's due time. This extra delay is shown in listing 4.3, but is added in the same way with the `tcpSend` method.

For each type of message that a node is supposed to process, the `onReceive` method must be overridden at that node. This method has two parameters: the endpoint (or TCP channel) of the sending node, and the type of message being received. The method is called by the message being sent, when the respective task is processed.

After a node is created in the simulation, it must be initialized. This initialization is made in the `init` method. The node can only send and receive messages after this method is called. This method is also where the node defines the tasks that it will run. Usually, there is a fixed number of nodes per simulation, that are all initialized when the simulation starts.

## 4.2 The System

The system for simulation was built using the tools explained in the previous section. The concept is simple: it simulates the execution of a cloud computing system, that has

data center nodes and client nodes. Each node has a storage unit and a statistics component (estimator or generator), and sets up tasks to run operations. For communication, a few different kinds of messages are used. In order to avoid the complexity of conflict resolution, our implementation relies on CRDTs, so every object stored is a CRDT. In order to merge CRDTs, besides the CRDT object, a causality clock is also stored for each CRDT. This is a logical clock that is used for concurrency checking on the CRDT operations. In our implementation, we use a version vector [PPRSWWCEKK83] as the causality clock. In practice, an object is stored as a CRDT-Causality clock pair. The estimates in the simulation system may be done according to the approach detailed in chapter 3, or by using a library that uses advanced forecasting techniques [MWH08]. In the latter case, the simulation is referred to as a *forecasting simulation*.

In order to know the exact state of the single-copy, we introduced an important class in the simulation: the Big Brother, which maintains a copy of the database where all updates are immediately executed. A concrete replica will converge to this state when no unknown updates exist in the system.

This section details how the nodes and Big Brother are composed, how they interact, and how they keep and use estimates.

### 4.2.1 Big Brother

The Big Brother is a static class that has a list of references to all nodes in the system, and can check the state of these replicas. Whereas nodes must communicate with each other through messaging to obtain states, this class can directly call methods on any node's store. This allows it to work as an oracle that does not need to make use of communication to know the actual divergence between a node's state and the copy of the Big Brother. The copy of the Big Brother, or single-copy, would not be available in a real distributed system.

To maintain the single-copy state, this class has a store of its own with all the objects in the system, and offers a public static method, `putSingleCopy`, that is called when an operation is executed at any node. This method applies the operation at the Big Brother's store and is called whenever an operation is executed at any node.

### 4.2.2 Messages

Before we detail the nodes' structures, it is important to notice, as stated earlier, that messages are the base of communication between nodes. In order to properly understand the interactions in the following section, we must understand the composition of the messages being exchanged.

All messages have two basic parameters: the key of the object the message is related to, and a timestamp indicating when it was sent. Besides these parameters, a message may have optional ones:

- the state of the object (a CRDT-Causality clock pair);

- information about an object at a certain timestamp, represented by the `ObjectInfoLog` class;
- a list of `ObjectInfoLog`s.

An `ObjectInfoLog` is a simple structure that stores information about an object at a certain timestamp. It contains a timestamp, a boolean that represents if the object is a number or not, and may contain optional parameters: the value of the object at that timestamp, if it is a number; the change an operation made to the object at that timestamp; the growth rate of the object for each metric; and the object state's forecasts for the next time period, if the forecasting library is being used. Basically, this structure works as a container for storing additional information about an object to be sent in a message.

Using these elements, the following messages are used in the system.

**`RequestMessage`** This message is used to propagate the state of an object from the client nodes to the data center. This message contains the state of the object, and a list of `ObjectInfoLog`s. Each log in that list has information about an update made in the client that has not been propagated to the data center yet, namely the numeric change applied to the object, or the kind of operation that was made on the object.

**`ReplyMessage`** This message represents the data center's reply to the `RequestMessage`. Besides the state of the object, this message also contains a single `ObjectInfoLog`, that carries the statistics information about the object, namely: the growth rate for each metric, as well as the estimated data center divergence, if the simulation is using the rhythmic approach; the object state's forecasts if the simulation is a forecasting simulation.

**`CoordinationMessage`** This message is sent from the data center to notify the clients that it is sending updated statistics about the object in a near future, so that clients can propagate local changes to the data center before that happens. It has no extra information besides the key of the object it is referring to.

**`RhythmMessage`** This message contains the same information as the `ReplyMessage`, but is sent periodically. This message is sent a short time after clients are notified.

Figure 4.1 shows the overall communication in the system, using these messages. It also shows the static accesses between the Big Brother and the nodes (being that the Big Brother accesses the state of the nodes, and nodes call the `putSingleCopy` method to update the single-copy state). The dashed lines represent messages that are sent periodically. Notice that "messages" is used generally in the picture, because a data center propagates to other data centers any messages it receives from clients.

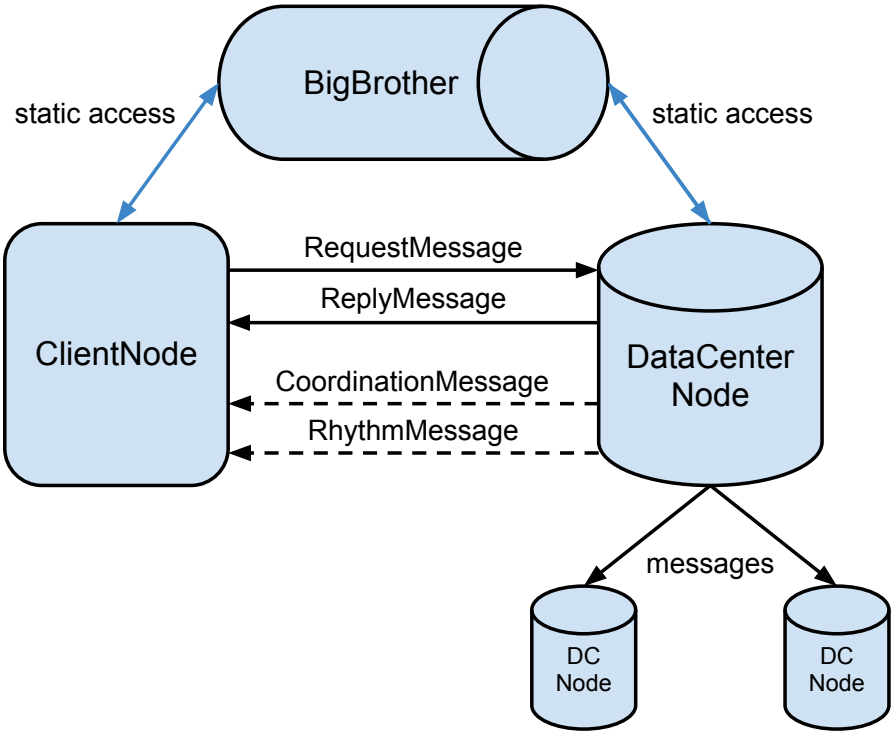



Figure 4.1: Diagram representing communication in the system

### 4.2.3 Components

At the heart of our system, two components are used in combination to allow nodes to estimate data evolution: the generator and estimator components. Each of the components has a few data structures to hold the necessary information for the statistics, and provides methods to use them.

#### 4.2.3.1 Generator Component

Data center nodes make use, primarily, of two methods from the generator component: `newUpdate`, which is called to add an operation on an object to the list of updates; and either `getObjectInfo` or `getForecasts`, depending on the simulation using the rhythmic or forecasting approach, respectively. This component starts keeping statistics about an object after the client calls the `startTracking` method for that object.

`newUpdate` gets two arguments besides the key of the updated object: the time at which the update was made and the value of that update if it is a numeric object. How it stores that information depends on the approach being used in the simulation. In practice, two different classes of generators exist, one for each approach. Both have a map, but they store different objects as values. These classes are detailed next.

**Rhythmic generator** This generator uses a class, `KVObjectLog`, to organize the statistics of an object. One instance of this class is stored for each object being tracked. This class maintains a mapping of timestamps-value, that represents the updates, ordered by

time, and the value of the object at that timestamp. Besides that, a count of updates made and the total sum of the updates' values are kept. These are used with the timestamps to calculate the growth rates. When `newUpdate` is called, a new entry is added to the `KVObjectLog`'s map, with the timestamp and value passed as arguments. The component also gets the time at which the update was seen by the data center. The difference between this time, and the timestamp is calculated, and a sum of these differences is kept, to calculate the average divergence per time unit of the data center's replica.

The `KVObjectLog` class provides public methods to calculate the growth rates and the local data center divergence with the formulas detailed in chapter 3. When `getObjectInfo` is called at the generator, it makes use of these methods to obtain the rates and data center divergence, and returns a `ObjectInfoLog` with that information.

When `startTracking` is called, the generator creates a new `KVObjectLog` for that object, with the initial value of the object if it is numeric, and with a count of one update.

**Forecasting generator** This generator uses the OpenForecast[1] library to predict the state of the object. The library's workings are simple, and follow the concepts presented on chapter 3: a `DataSet` is a collection of `Observations`. A static method `getBestForecast` is applied on a `DataSet` to decide on the best statistical model. The obtained `ForecastModel` then calls the `forecast` method with the `DataSet` as parameter, and returns an `Observation` with the prediction. In this case, the dependent variable in an observation is either the value of the object, or the number of operations, and the only independent variable is time.

This class keeps a `DataSet` for each object being tracked. When `newUpdate` is called, a new `Observation` is inserted in the dataset. When the `getForecast` method is called, a forecast is obtained for each 100ms remaining until the next coordination with the clients. These forecasts are put into an array and returned. This array is for the estimator component to use as a way of estimating the real state of the object. The state is forecasted for every 100ms so that when the estimator is verifying if an invariant is kept when performing an operation, it estimates the state os the single-copy at the closest tenth of a second.

When `startTracking` is called, the generator creates a new `DataSet` for that object, with one `Observation`.

#### 4.2.3.2 Estimator Component

The estimator component has two main functions: to log the updates that have been made locally at the client since the last coordination with the data center, and to check if coordination is needed.

For the first function, this component keeps a linked list of `ObjectInfoLogs` for each object the client has a copy of. Whenever a change is made to an object, that change is

[1] `http://www.stevengould.org/software/openforecast/index.html`

"logged" by inserting a new `ObjectInfoLog` in the list, with the timestamp at which the change occurred and the value of the object (if it is a numeric object). This is done by calling the `addToSend` method. When a `RequestMessage` is created by the client, this list of changes is obtained with the `getUpdatesList` and put into the message, to be used by the generator component at the data center.

For the second function, the component keeps for each object a `ObjectInfoLog` with the statistics received from the data center. This log is updated when the client receives a `ReplyMessage`, by calling the estimator method `addToStats` passing the obtained log as argument. In this implementation, the component keeps metrics' bounds as simple variables indicating the upper and lower bounds for that object, with another variable representing the desired confidence level.

When the client calls the boolean-returning `getCoordinate` method for a certain object, the component's action depends on the approach to the estimates being used in the simulation, as explained next.

**Rhythmic approach** If the simulation is using this approach, the component obtains the growth rate and last seen state from the log, and calculates the estimates and confidence degrees with the formulas presented in the previous chapter. A small helper `PoissonDist` class is used for calculating the Poisson probabilities. According to the desired confidence degree, the method returns if coordination is or is not needed.

**Forecast approach** For this approach, the solution is simpler. Since the OpenForecast library provides no way of getting certainty degrees, the estimator component makes a simple check if the forecasted value for that time slot violates an invariant. If it does, true is returned.

Figure 4.2 summarizes the relations between these classes and the nodes. Note that this class diagram is simplified, only showing the most important methods and attributes. Its purpose is to help understanding the interaction between the pieces of the system.

### 4.2.4 Nodes

The basic node structure is the same for both clients and data centers. Both nodes have a storage unit, and a statistics component.

The storage, to simulate a key-value store, is a hashtable. Since objects are stored as CRDT-Causality clock pairs, the node's hashtable is viewed as a key-CRDT store. Applications are expected to retrieve and store CRDT objects. Since the store follows the key-value paradigm, nodes provide two operations to access the store, a `storeGet` and a `storePut`. An abstract class is used to represent these basic functions of a node, the `KVReplicaNode` class, as seen in figure 4.2.

It is in these methods that the nodes call the statistics components's operations before and/or after retrieving or storing the CRDT. At a basic level, the `storePut` operation

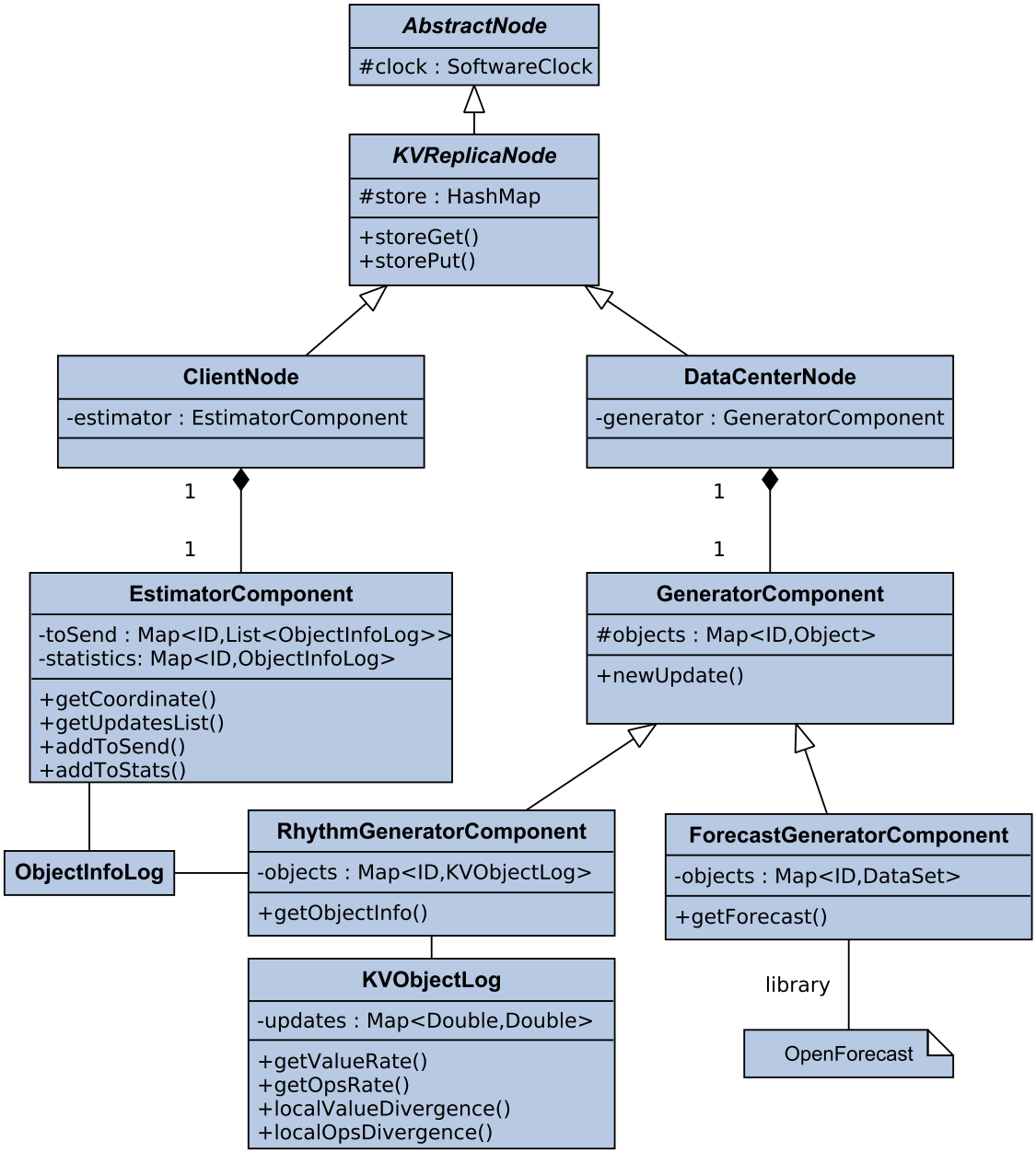



Figure 4.2: Simplified class diagram with the relations between nodes and components

merges the CRDT to be stored with the one in storage. Listing 4.5 shows the part of the code for putting a CRDT into a store. As can be seen, the stored CRDT's `merge` method is used to merge the store's state with the new one. Note that the clocks are also merged to accommodate all operations from both CRDTs. It is in this method that the Big Brother's `putSingleCopy` method is called, to update the single-copy state of that object.

Listing 4.5: `storePut` method with the merging of CRDTs

```
1  public Pair<CvRDT, CausalityClock> storePut(CRDTIdentifier key,
2      Pair<CvRDT, CausalityClock> value)
3      // ... //
4
5      // If there is an object in storage, merge the CRDTs and clocks
6      CvRDT newCRDT = value.getFirst();
7      CausalityClock newClock = value.getSecond();
8      Pair<CvRDT, CausalityClock> ccPair;
9
10     ccPair = (Pair<CvRDT, CausalityClock>) store.get(key);
11
12     CvRDT storeCRDT = ccPair.getFirst();
13     CausalityClock storeClock = ccPair.getSecond();
14
15     storeCRDT.merge(newCRDT, storeClock, newClock);
16     storeClock.merge(newClock);
17
18     store.put(key, new Pair<CvRDT, CausalityClock>(storeCRDT, storeClock));
19
20     BigBrother.putSingleCopy(key, value.getFirst(), value.getSecond());
21
22     // ... //
23
24     return oldPair;
25 }
```

In order to simulate real-world latency, delay is added whenever the `udpSend` or `tcpSend` methods are called. Between clients and data centers, the added delay is somewhere between 20 and 50 milliseconds, calculated randomly in each method call. Between data centers the added delay is between 150 and 175 milliseconds. This delay was chosen based on latency measurements done between Amazon's EC2 data centers [NAEA13; Bal12]

#### 4.2.4.1 Client Nodes

Client nodes keep a partition of the data in the storage unit as a local copy. Initially, the hashtable is empty, and when an operation is made on a key that has no value, the client coordinates with the data center to obtain the object, if it is a `storeGet` operation, or to create it at server-side, if it is a `storePut` operation.

The client keeps the information about the endpoint of the data center it must synchronize with, which is defined when the client is created. A client node always coordinates with the same data center for the remainder of the simulation. This data center

acts as a primary copy of the data for the client, and all operations made by the client are propagated to that data center, which in turn propagates them to the other ones.

The core of the client nodes' usage is the `storeGet` and `storePut` methods, since these are the methods that applications use to store and retrieve data. In the client nodes, besides getting and putting CRDT objects, these methods make use of methods from the estimator component, to check if coordination is needed. `storePut` is used as an example of code, as can be seen in listing 4.6 (the `storeGet` method is ommitted because it is very similar, except nothing is added to the estimator). The first thing to note when looking at this code, is that the operation is immediately logged calling the estimator's `addToSend` operation. Then, the need for communication is checked, and the operation is made locally. Afterwards, if coordination is needed, a request to the data center is made. If not, and if the system is propagating operations asynchronously with no delivery guarantees, such a request message is built and sent. The private method for making a request to the data center, and treating the response is shown in listing 4.7. The most relevant thing about that piece of code is the usage of the estimator methods to get the updates when the request is built, and to add the new obtained statistics. The auxiliary method for asynchronous propagating of operations is very similar to this one, except no reply is obtained and treated.

Listing 4.6: Client-side `storePut`

```
@Override
public Pair<CvRDT, CausalityClock> storePut(CRDTIdentifier key,
      Pair<CvRDT, CausalityClock> value) throws NullPointerException {
    double currentTime = clock.currentTime();

    estimator.addToSend(key, value);

    boolean coordinate = estimator.getCoordinate(key, currentTime, true);

    Pair<CvRDT, CausalityClock> tmpRes = super.storePut(key, value);

    if (coordinate) {
        ReplyMessage reply = requestToDC(key, super.storeGet(key));
        return reply.value();
    } else {
        if(async)
            asyncRequestToDC(key, super.storeGet(key));
    }

    return tmpRes;
}
```

Listing 4.7: Method to make a synchronized request to the data center.

```
1  private ReplyMessage requestToDC(CRDTIdentifier key,
2        Pair<CvRDT, CausalityClock> value) {
3      RequestMessage request = new RequestPutMessage(key, value,
4        clock.currentTime(), estimator.getUpdatesList(key));
5
6      ReplyMessage reply;
7
8      // Adds 20ms to 50ms delay on contacting DC
9      TcpChannel tc = endpoint.tcpSend(connectedDC.endpoint, request,
10       Simulation.rg.nextDouble() * 0.03 + 0.02);
11     reply = tc.tcpRead();
12
13     estimator.addToStats(key, reply.getInfo());
14     super.storePut(key, reply.value());
15
16     return reply;
17 }
```

#### 4.2.4.2 Data Center Nodes

Data center nodes keep a hashtable with all objects in the system and send and receive messages from the clients.

When the data center is initialized, two periodic tasks are started, with one second difference of each other. The period between two of these tasks' executions is configurable when the node is created. These tasks serve the purpose of propagating the growth rate and the data center replica's state to the clients. The second message, a `RhythmMessage`, is the one that sends these objects. The first message, a `CoordinateMessage`, serves only as a warning that the data center is going to coordinate with the clients, and is important because it creates a periodic "checkpoint" where the data center receives all updates that were made locally at the clients since the last coordination. Clients are supposed to send their updates at this time, and the generator component re-calculates the growth rate, so that when the second message is sent, the statistics are updated to reflect all operations made locally at the clients.

To establish this coordination with the clients, the data center keeps a list of all the clients that access that data center as a primary copy. This list is updated every time a new client accesses the data center.

Whenever a `RequestMessage` is received from a client, the data center calls the generator component's `newUpdate` method, for each `ObjectInfoLog` in the list contained in the message, with the timestamp and value in that object. If the message was sent using a TCP channel, then the data center builds a `ReplyMessage` to reply to the client, with the state of the object in the store, and the statistics obtained from the generator node. The data center then merges the CRDT in the message with the one in storage, and the `RequestMessage` is propagated to the other data centers. The code that does this for a TCP channel is contained in a `onReceive` method presented in listing 4.8.

Listing 4.8: `onReceive` method that processes a `RequestMessage` sent by a client.

```
1  public void onReceive(TcpChannel ch, RequestMessage m) {
2      // ... //
3
4      // Log the updates
5      for (ObjectInfoLog log : m.getLogUpdates())
6          generator.newUpdate(m.key(), log);
7
8      // Merge the state
9      Pair<CvRDT, CausalityClock> pair = super.storePut(m.key(),
10         m.value());
11
12     double ts = clock.currentTime();
13
14     // ReplyMessage with the statistics obtained from the generator
15     ReplyMessage reply = new ReplyMessage(m.key(), pair,
16         generator.getObjectLog(m.key()));
17
18     // Propagate message to other data centers
19     propagateMessageToDCs(ch.src, m);
20
21     // Reply to the client
22     ch.tcpReply(reply);
23 }
```

# 5

# Evaluation

This chapter describes the evaluation done using the implementation of the system described earlier. First, the adaptation of the standard TPC-W benchmark used for these tests is detailed. Then, some tests on the estimates' precision when several clients are using them are presented, as well as demonstrating how the clients keep the invariants. Afterwards, the impact of different properties (such as the rate of operations or the estimate model used) is studied.

## 5.1 The Benchmark

For our evaluation, we used the TPC-W benchmark [Cou02; GG03]. This benchmark simulates an e-commerce platform that sells books, and provides the users with options such as registration, browsing and purchasing items. TPC-W is frequently used to test the performance of relational databases [CPW07; PA04], and has also been used previously with CRDTs in a key-value store context [BP12]. Despite this, our usage of this benchmark is not to get a performance measurement of our implementation, but rather to get a view of how the estimates fare in a realistic scenario.

We adapted the benchmark implementation that uses CRDTs, to work with the simulated system. This implementation is based off a publicly-available implementation for the Cassandra database[1].

This section presents an overview of the benchmark, its operations and organization, as well as the adaptations we made for our tests.

[1] `https://github.com/PedroGomes/TPCw-benchmark`

### 5.1.1 TPC-W

TPC-W simulates an e-commerce platform. The original TPC-W specification includes a set of operations that simulate the users' interactions through a web application with a graphical interface. Table 5.1 shows the operations defined by TPC-W that are implemented in the version of the benchmark we adapted.

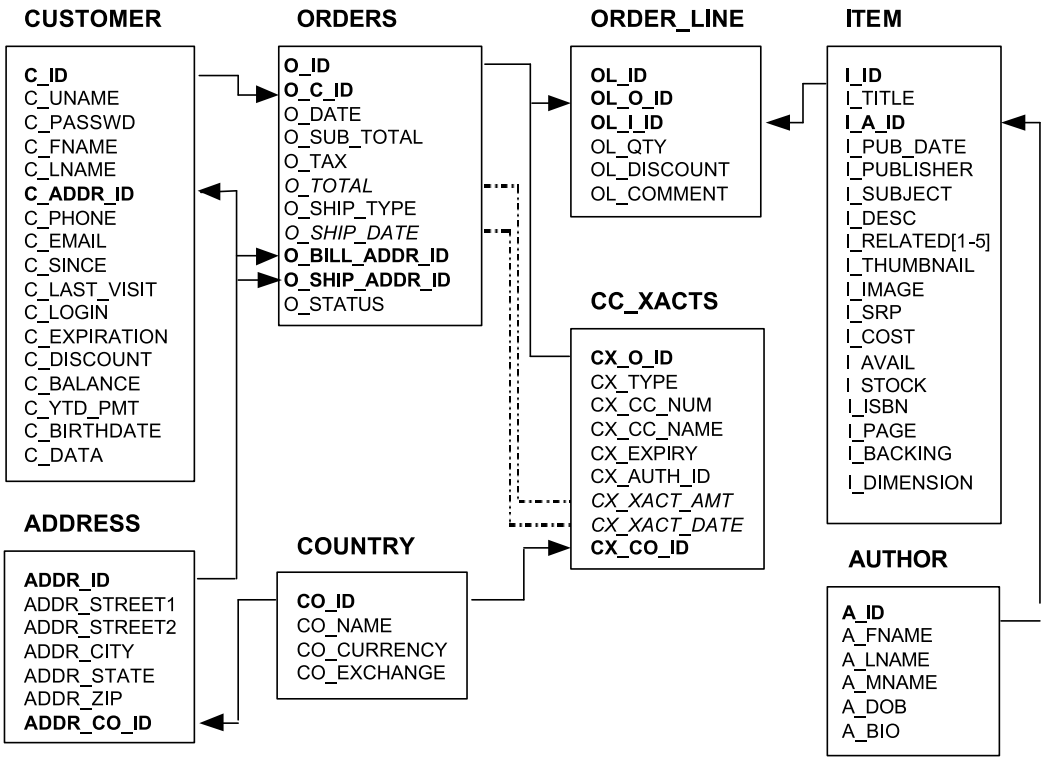
| Operation | Parameters | Description |
|---|---|---|
| PRODUCT DETAIL | item_id | retrieves information about an item with item_id |
| HOME | item_id, customer_id | retrieves information about an item with item_id and customer with customer_id |
| SHOPPING CART | item_id, cart_id, CREATE | adds a new item, with item_id, to an existing shopping cart with cart_id, or a new one if CREATE is set to true. |
| SHOPPING CART | item_id, qty | adds quantity qty of item_id items to the shopping cart |
| BUY REQUEST | cart_id | computes the total cost of a shopping cart and the billing information |
| BUY CONFIRM | customer_id, cart_id | Creates a new order and a new payment for a shopping cart that was previously processed in BUY REQUEST |
| ORDER INQUIRY | order_id | checks the status of an order |
| BEST SELLER | | Computes the Best Seller information for each category of items |
| ADMIN CHANGE | item_id | Adds the item with item_id to its subject index, adds the five most sold items to the related |
| CUSTOMER REGISTRATION | customer_id | Registers a new customer |

Table 5.1: Description of TPC-W operations (Adapted from [Bal12])

This set of operations, as well as the benchmark's data-model, was designed taking into account an underlying relational database. The schema for such a database is shown in figure 5.1. The table *ORDERS* registers the clients' orders, *ORDER_LINE* the items from a particular order, and *CC_XACTS* represents the payment of an order. The other tables store information for the customers, addresses, countries, items and authors.

#### 5.1.1.1 Workloads

TPC-W proposes three workloads: *shopping*, *browsing*, and *ordering*. Each workload simulates a different usage pattern of the book store. Workloads use different amounts of each operation, resulting in some workloads being more read-heavy than others. The shopping workload has 95% of read-only operations, the browsing workload has 80%, and the ordering workload has only 50%. The latter is the workload we use for all our tests, to maximize write operations on an object.




**Legend:**

- Dotted lines represent one-to-one relationships between non-key fields related through a business rule. These fields are shown in italic.
- The arrows points in the direction of one-to-many relationships between tables.
- Bold types identify primary and foreign keys.

Figure 5.1: TPC-W database schema (Taken from [Cou02])

### 5.1.2 Benchmark Configuration

In our tests, we focused on evaluating if the proposed approach meets the objective of guaranteeing that application invariants are not broken. To this end, in the context of the TPC-W benchmark, we have used only the value metric, to guarantee that the stock of products remains positive despite concurrent orders. In the context of TPC-W, this was the only constraint that seemed relevant to maintain (as also identified in [LPCGPR12]). Interesting tests with other metrics could be possible with other applications or datasets, as discussed in chapter 6.

According to the TPC-W specification, on a BUY CONFIRM operation, after the stock is checked, if the sale would bring it under 10, then the stock is increased in 20 units. This would make it fairly easy to guarantee the restriction. To test for the metric's precision, our implementation changed this condition to only increase the stock when the sale would bring it under 0 units. This causes repositions to be less common, and as clients do not instantly see other operations, when the single-copy is close to 0, a client may see a higher value and perform the sale without restocking. In this case, it would lead to the stock becoming negative. In order to avoid this, we check if the metric tells the client it should obtain more recent updates before doing that.

One thing to be noted is that the performance tests that use the TPC-W benchmark, usually use populations of the database with thousands of different items and several authors. For simplifying the evaluation, in our tests we use a population with only one item and one author. With only one item, we guarantee that all BUY REQUEST and BUY CONFIRM operations by all clients are always made on the same item.

A test consists in a simulation, with a given number of clients and data centers, that runs for a certain amount of virtual time. Different parameters must be configured:

- whether the clients batch updates made locally, or propagate them asynchronously to the data center after applying them;
- the time between components' coordinations;
- whether the rhythmic model or forecasting library are be used for the estimates.

We study how changing these parameters influences the estimates. The default configuration is: 100 clients, 3 data centers, asynchronous propagation of updates, inter-operation time for each client of 5 seconds, a period of 15 seconds between consecutive coordinations of the estimator and generator components, and estimates using our rhythmic model. The restriction must be kept with 99% confidence.

Each client node runs a client instance of the benchmark. A periodic task that executes every `period` seconds is started at each client node when it is initialized. That task executes the next operation in the workload, and its start time is a random time between 0 and `period` seconds after the clients are initialized, to guarantee that operations from clients do not occur only every `period` seconds.

All tests were run in a single machine, running Debian Linux 4.4.5-8, with 64 *AMD Opteron Processor 6272 1.4Ghz* CPUs and 64 gigabytes of RAM. While all those CPUs do not necessarily make the simulation run faster, since the simulator is single threaded, they allow several simulations to be run simultaneously.

### 5.1.3 Tests

There are five points we wish to evaluate:

- if the estimates are precise – i.e. are invariants kept when the clients use the estimates to decide when to contact the data center?
- if the estimates are scalable to a large number of clients – i.e. are invariants kept successfully even with many clients performing operations locally on an object?;
- how the estimates fare with different communication models – i.e. if clients can rely on the estimates regardless of how the updates are propagated to the data center;
- how different estimate models influence the results – in this case, our rhythmic model vs. the forecast models;
- how other properties of the system or applications, such as time between coordinations, or amount of repositioned stock, influence the estimates.

Before presenting our evaluation, we show the evolution of the system in the specific situation we have been using as example, when the client wants to make a decrement operation when the invariant is close to be broken. This allows a better understanding of how the system works. Consider figure 5.2, where the evolution of the stock value is represented, in its real value, and as seen by the client replica and the data center. Represented by the purple line is the confidence degree, as computed by the client, that the invariant $stock \geq 0$ is kept, which uses the right axis as reference. A bit before the 116 second mark, the single-copy value of the stock decreased, and a bit later the data center accommodated that change. The client maintained the value it had after the last operation. Near the 116.5 mark, the client wanted to perform an operation. As can be seen, the confidence degree of the estimate is way below the 99% limit. As such, the client coordinated with the data center, and performed a restock.

Another interesting thing about this figure is how the confidence degree changes. One may find it strange that right after the reposition, the confidence degree drops quickly until the next contact with the data center. This happens because a BUY CONFIRM operation retrieves the value of the stock, using the `storeGet` operation, and then stores the new stock value using `storePut`. On the value retrieval, communication is made with the data center, and that state is stored by the estimator as the last seen state. Since the next operation, is an increment (because of the reposition), it executes locally without a problem, because the restriction is preserved. This causes the confidence degree to be

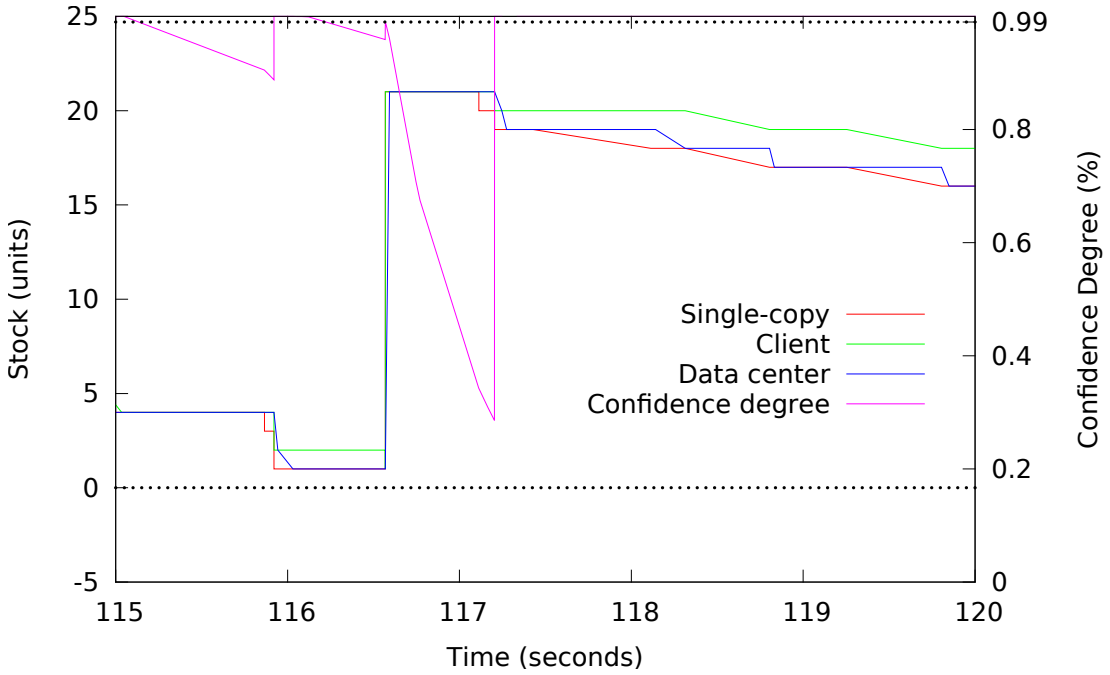



Figure 5.2: Extract of a simulation run, at a time when the client is making an operation that would break the invariant

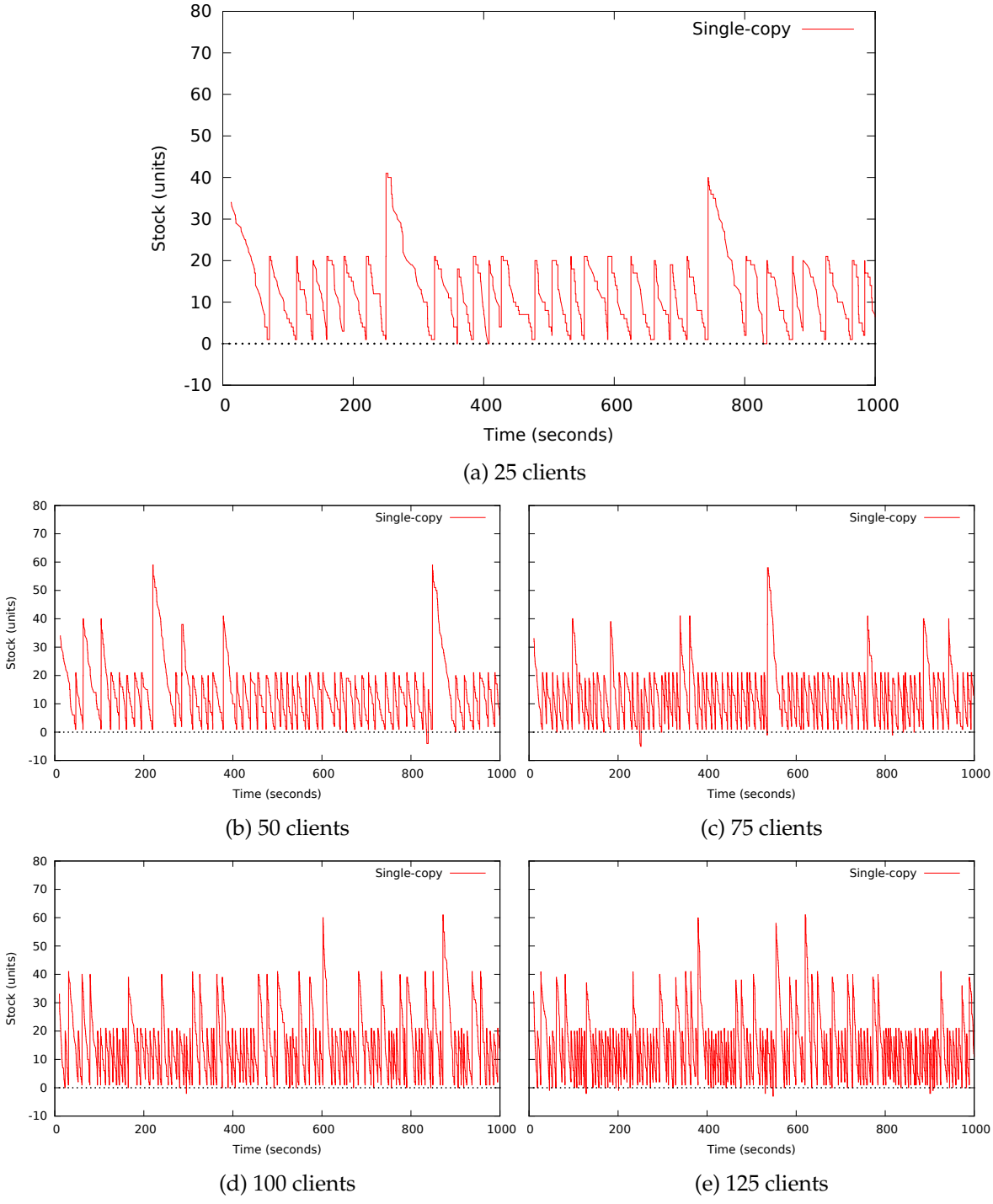
pessimistic until the next time the data center is contacted, because the stored state in the estimator is a stock value that is still close to 0. However, after the next communication, the client performs several operations without synchronizing with the data center, because the confidence degree remains over 99% for a while, since the restriction is far from being broken. Opposed to this, before the reposition, the client communicates with the data center more often, since the confidence degree drops quickly after every communication, as the invariant is closer to being broken.

#### 5.1.3.1 Scalability

In this section, we evaluate how the increase in the number of clients impacts the system. Figure 5.3 shows the evolution of the stock's real value. Particularly, as can be seen in subfigure 5.3a, for few clients, the stock's value is always incremented as it should, when the invariant is close to being broken. Sometimes this value peaks to the double of the usual restock value (20). This happens when more than one client simultaneously sees a state where a restock is in order. This is not a problem, since no upper bound on the value is set.

As can be observed, in general, for larger number of clients, the stock evolves in the same way. As expected, as more clients are in the system, the more times the value approaches 0 and is restocked, since more clients means more operations on the object. For the charts of 50 clients onward, very rarely, the invariant is broken. The estimates are not perfect, but we can also observe that this situation does not scale as the number of

clients in the system increases.




(a) 25 clients

(b) 50 clients

(c) 75 clients

(d) 100 clients

(e) 125 clients

Figure 5.3: Evolution of the real value of the stock, in simulations with different numbers of clients

These results were all taken from simulations using the rhythmic model. Figure 5.4 shows that similar results are observed when using the OpenForecast library for many clients.

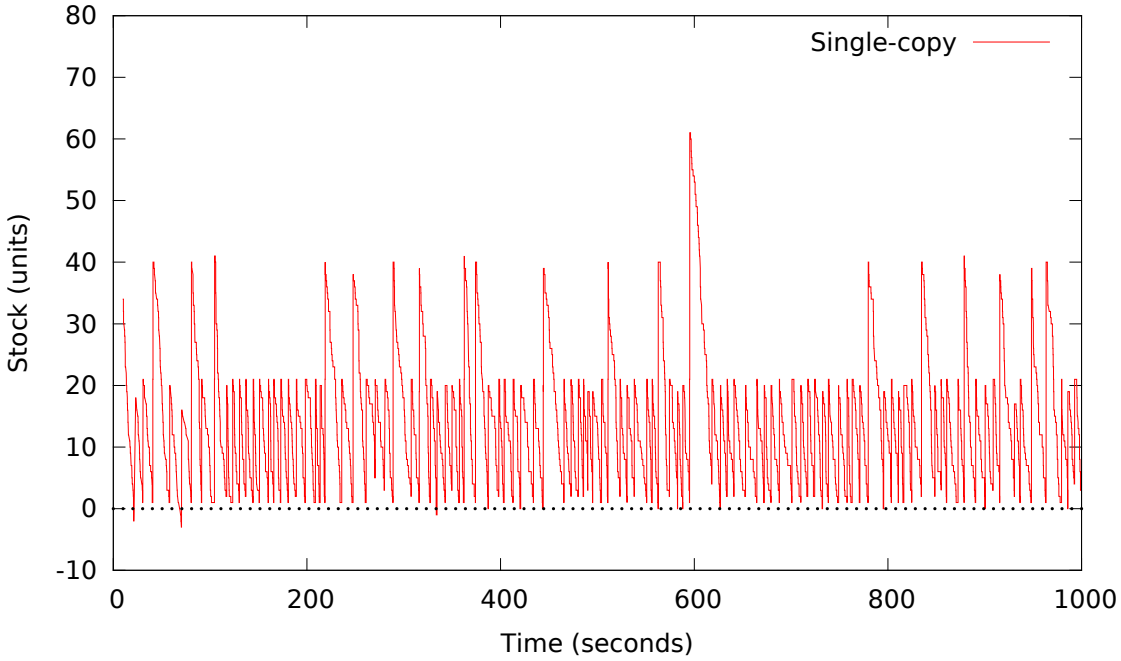



Figure 5.4: Evolution of the real value of the stock, when using the forecasting library, for 100 clients

For evaluating how our approach reduces the need for contacting the servers, we have measured the number of operations that are performed locally at the clients without requiring immediate coordination with the data center. Whereas the previous test results evaluate how the accuracy of the estimates is kept, the results in figure 5.5 show how the usage of the estimates prevents coordination between clients and data centers. These results are from the same tests as figure 5.3. Note that the operations measured in this chart are `storeGet`s and `storePut`s, not benchmark operations.

We can observe that the percentage of operations executed without the need for coordination is quite high, being over half of all the operations done in the system. As can be seen, when the number of clients increases, the amount of operations done locally decreases slightly, however, even for 125 clients, this amount is never less than 65.5%, which we consider positive. Besides, for a higher number of clients, from 75 to 125, the difference in percentage is not all that significant, even though the number of clients increased noticeably. This occurs because since the the rate of updates increases and the stock diminishes faster, it is more probable to estimate that an operation could lead to the invariant being broken, since more concurrent updates may have happened.

#### 5.1.3.2 Update Propagation

The previous tests were all made considering an asynchronous approach to the propagation of updates to data centers. This means clients propagate operations in the background, with no delivery guarantees, each time an operation is made. However, some

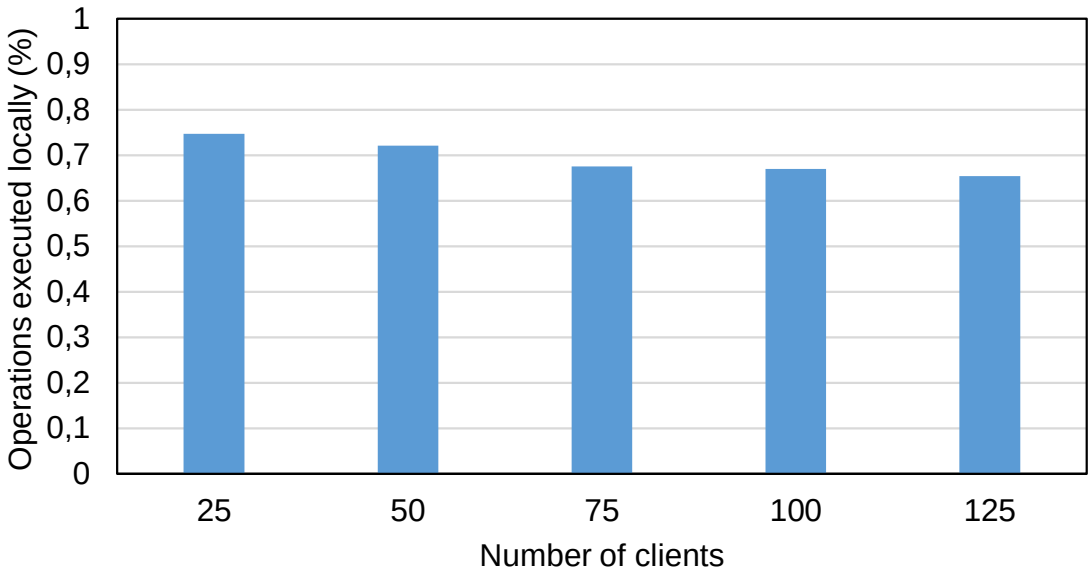


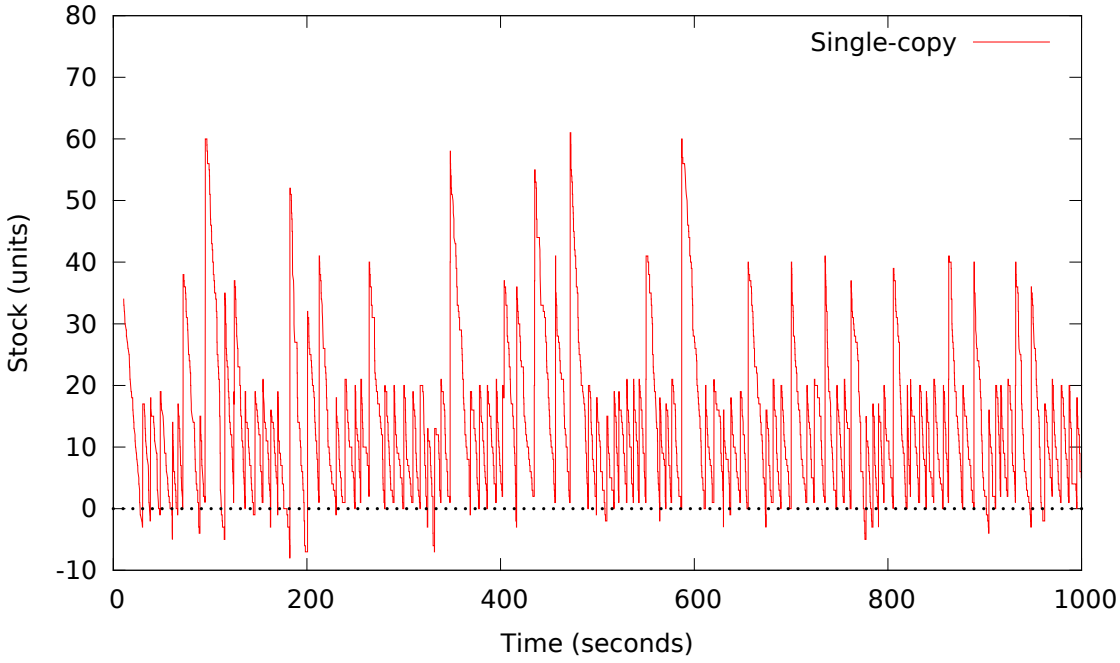

Figure 5.5: Percentage of operations executed locally, for different numbers of clients

systems [CRSSBJPWY08; BBCFKLLLLY11] have batching mechanisms for propagating updates. We tested to see how using this mechanism changes the metrics' precision.

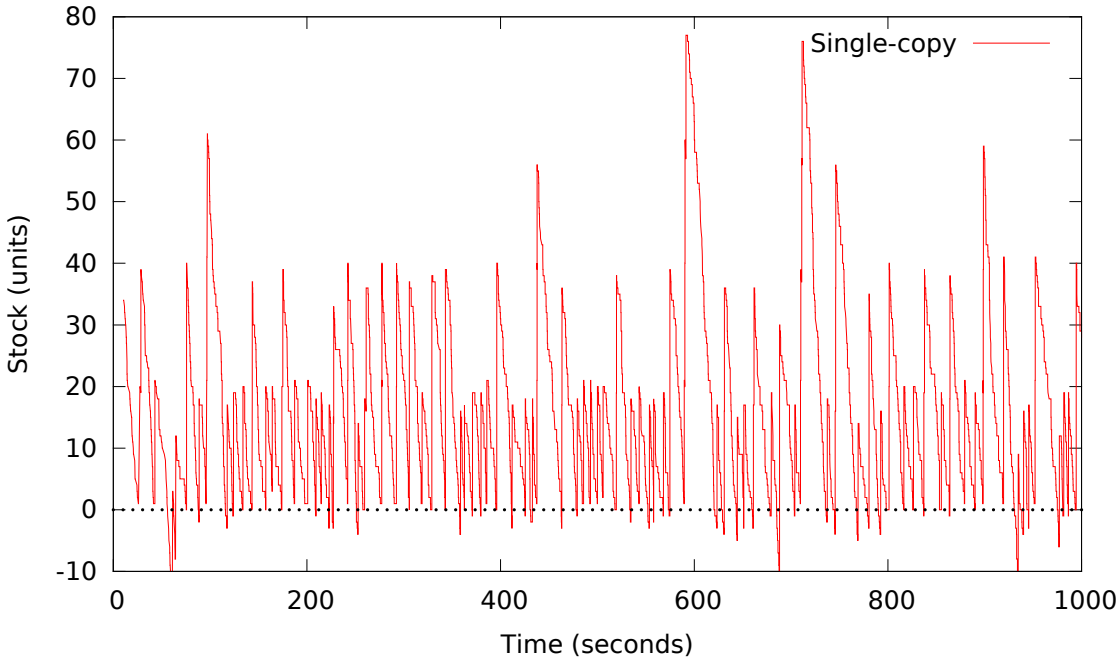
Figure 5.6 shows the evolution when using batching. As can be observed, the invariant is broken more often than when propagating the updates asynchronously, especially when using forecasting. The main reason for this, is that since updates come in batches, the value of the data center itself is more divergent from the real value for longer periods, and the generator does not update the statistics at a regular rate. Thus, besides the generated statistics probably being less accurate, the the data center's divergence will play a bigger role in the estimation. For the rhythmic model, this statistic is simply an average. If the amount of pending operations varies greatly from batch to batch, then this statistic may not provide an accurate estimate of how divergent the data center is at some points. The estimator would be then using an optimistic statistic.

As to why the solution with forecasting behaves somewhat worse, this is because with this library, we have no real way to estimate data center divergence. Observations are used to generate forecasts of the actual single-copy state, which are sent to the estimators. If several pending operations are being batched, and batches sizes vary, the forecasting models may be lacking actual observations to generate a precise forecast of recent history.

The execution of operations locally when batching updates is also affected as shown in figure 5.7. Using the rhythmic model with batching, as would be expected, yields less operations executed locally, because higher divergence on the data center side causes the estimate to be less certain, and still causes the restriction to be broken sometimes. On the contrary, when using forecasts, the amount of operations done locally is similar to the asynchronous propagation setting, however, as seen earlier, many of these operations




(a) Using the rhythmic model




(b) Using forecasting

Figure 5.6: Evolution of the real value of the stock, using batching to propagate updates

cause the invariant to be broken, because the forecasts are not accurate.

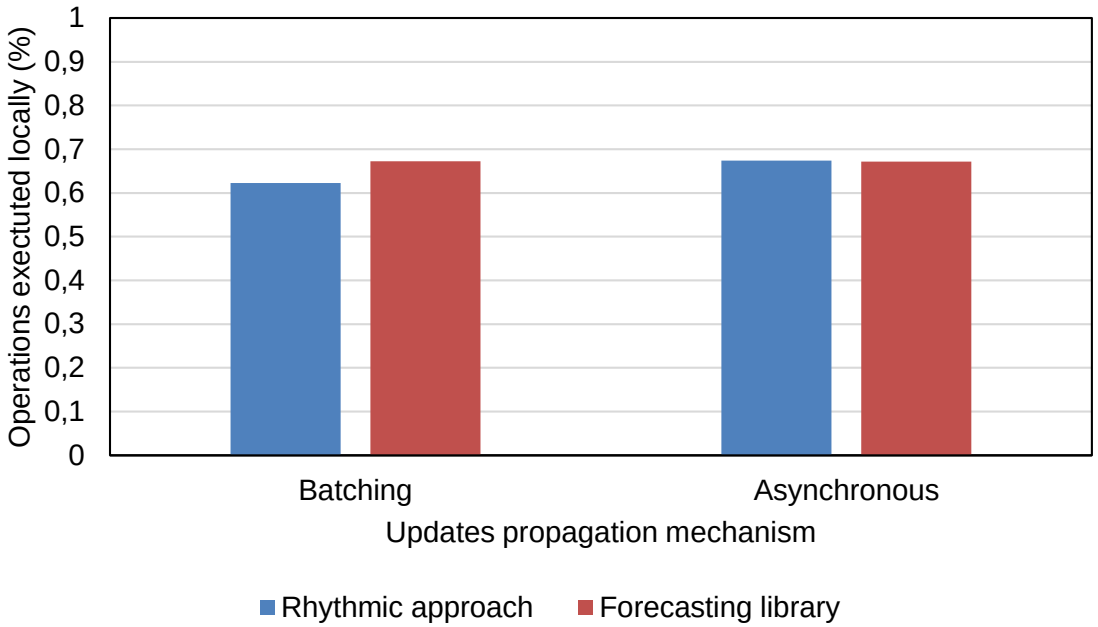


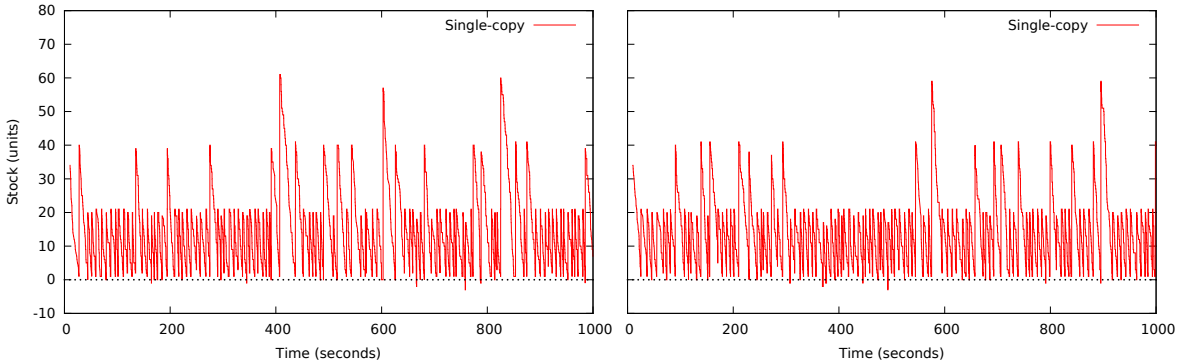

Figure 5.7: Percentage of operations executed locally, for different propagation techniques

#### 5.1.3.3 Coordination Between Components and Application Configurations

An important detail of our approach is the coordination between components. Previously shown tests have all been run with the components coordinating every 15 seconds, so the generator gets the newest state and updates made during that time, and the estimator gets updated statistics. Figure 5.8 exemplifies how the object's evolution is affected if we double (5.8a) or triple (5.8b) this time. The evolution of the object is not affected, the estimates estimate correctly when coordination is needed with the data center in order for the invariant to be preserved.

Another time-related parameter that may affect the estimates is the rate of updates. Previous experiences assumed a regular rate of one operation every 5 seconds, at each client. Testing with a rate twice as fast proved that the statistics accommodated this rate naturally, and the object's evolution behaved in the same way, though the value decreases twice as fast (as expected). This behaviour can be observed in figure 5.9.

Finally, we combined both changes – twice the rate of operations, and more time between coordinations – and observed how they affect the amount of operations executed without contacting the data center. This is presented in figure 5.10. As the time between coordinations increases, slightly less operations are executed locally. This makes sense, because as the components interact less, it is more likely that when estimating, the statistics are stale and the confidence degree lowers. However, even for 4 times the original period, the difference in percentage is less than 5%. It can also be seen that the rate of operations has barely any effect on this result: for the three different periods, the amount




(a) 30 seconds (b) 60 seconds

Figure 5.8: Evolution of the real value of the stock with different times between coordination of the components

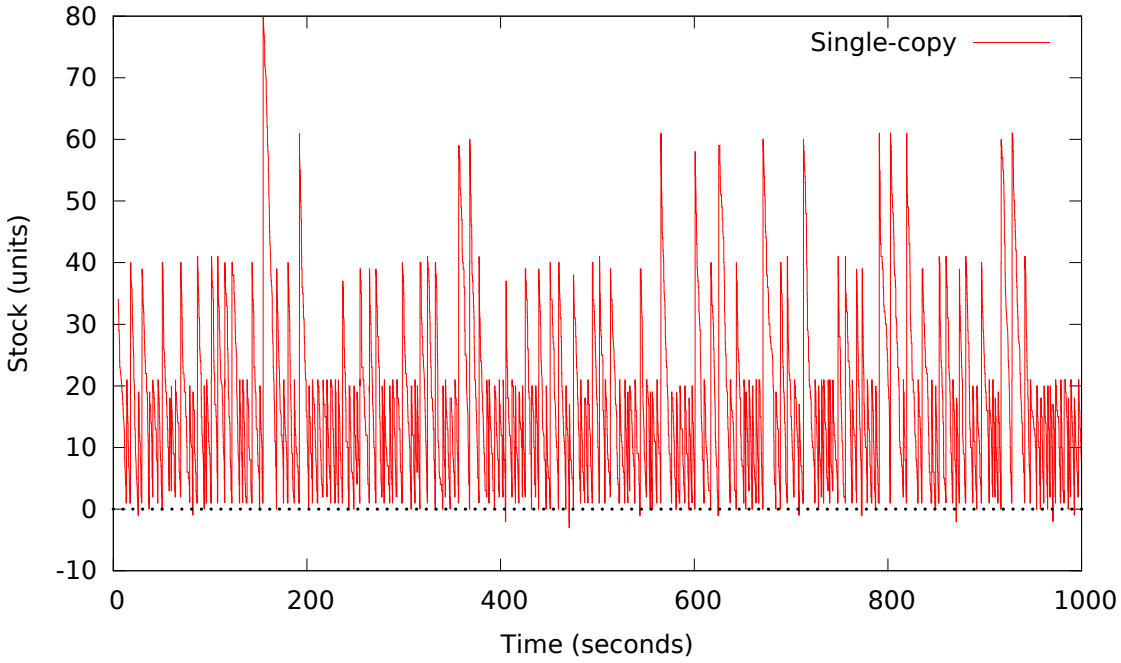



Figure 5.9: Evolution of the real value of the stock, where each client performs an operation every 2.5 seconds

of operations remained close to or the same as the simulations that used the original rate.

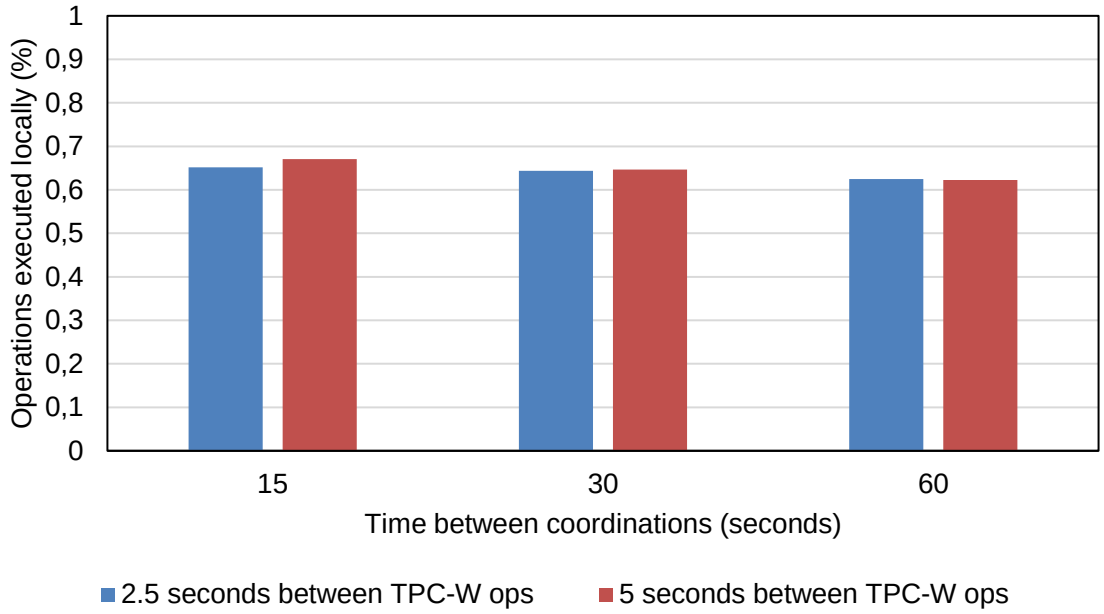



Figure 5.10: Percentage of operations executed locally, for different times between benchmark operations

Another parameter that influences the amount of communication required is the number of units restocked. By default, the TPC-W specification defines 20 as the amount of units to add when restocking an item. Is this amount is increased, with the same rate of updates, it will take longer to reach a state where the invariant is close to be broken, as demonstrated in figure 5.11. This means that the amount of operations that can be executed locally increases, because the probability of the invariant being broken is lower for a larger amount of time. This is confirmed by our tests, as seen in figure 5.12.

#### 5.1.3.4 Complexity and Message Size Comparison Between Models

We end this section with a simple analysis of how the estimate models fare against each other in terms of complexity and system overhead. For the rest of this section, we consider the following: $P$ – the number of updates at a client that have not been propagated to the data center yet; $N$ – the number of updates seen by the generator; $L$ – the difference between the upper and lower bound on a metric at a client, i.e., the size of the interval in which the metric's value must be; $double$ – the size of a double; $period$ – the time between the coordinations of the components. Notice that this comparison is made for each object that is being tracked, to simplify the expressions.

In terms of space complexity, for both models on the estimator side, the volume of information that is kept is a constant number of parameters (bounds for the metrics, last timestamp from when communication was made with the generator), and a list with the $P$ updates that have not been propagated to the data center, so they can be sent the next time communication is established. Furthermore, for the forecasting models, a forecast

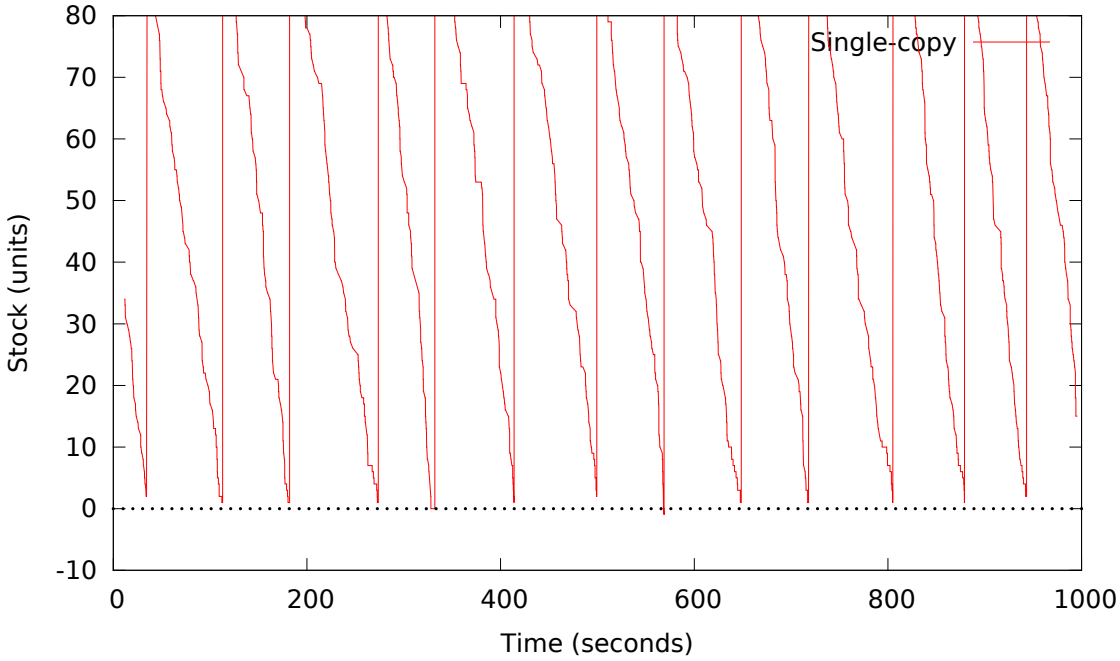



Figure 5.11: Evolution of the real value of the stock, with a restock amount of 100

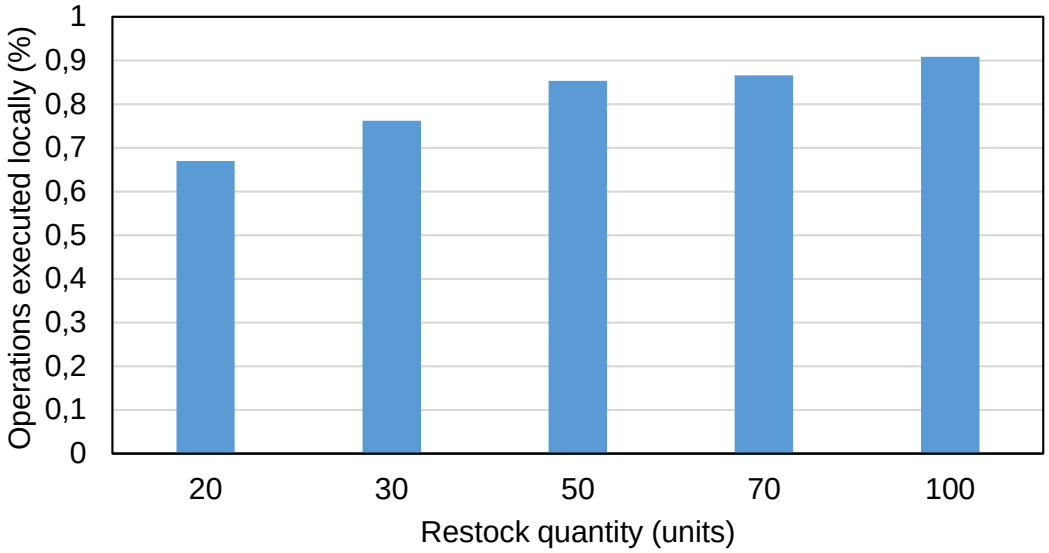



Figure 5.12: Percentage of operations executed locally, for different restock amounts

is stored for each 100 milliseconds in $period$. On the generator side, both models store a list of updates, of size $N$. The rhythmic model also stores a constant number of values as explained in the previous chapter.

For the time complexity, in the estimator component with the rhythmic model, the process that causes the complexity is the computation of the confidence degree, which is a summation or an integral as seen in chapter 3. This depends on the size of the interval, $L$. For the forecasting models, the operation to check if coordination is needed is done is

constant time, it is only a check against a value. On the generator's side, for the rhythmic model, the calculations of the rates are done in constant time, using the constant values in storage that are updated when a new operation is seen. With the forecasting models, the documentation is not very clear, however, at least one passage on the whole collection of observations is made to decide which model to use, and then again to forecast the value.

Finally, we analyze the messages' sizes. For simplicity purposes, we will only describe the extra information each model needs, and not the information the messages must already contain (the object's key and value, etc). The messages sent from the generator to the estimator, for the rhythmic model, contain two rates and the average divergence the data center experiences. The forecasting models need much more: one forecast value for each slice (in our implementation, 100 milliseconds) of $period$.

Table 5.2 summarizes these complexities and message sizes.

| | **Rhythmic** | | **Forecast** | |
|---|---|---|---|---|
| **Comparison** | **Estimator** | **Generator** | **Estimator** | **Generator** |
| Space Complexity | $O(P)$ | $O(N)$ | $O(P + \frac{double \times period}{0.1})$ | $O(N)$ |
| Time Complexity | $O(L)$ | $O(1)$ | $O(1)$ | $O(N+)$ |
| Message Size | $P \times 2\ double$ | $3\ double$ | $P \times 2\ double$ | $double \times \frac{period}{0.1}$ |

Table 5.2: Complexities and message sizes for both models

The rhythmic model, performance-wise, is the lightest one. This is because the forecasting library, every time the statistics need to be computed, the best model must be found (in linear time), and many values must be forecasted (in at least linear time) to be sent to the estimator components. When the collection of observations becomes large, or the time between coordinations is larger, this may require lots of computation time. As opposed to this, the rhythmic model obtains the rates and average divergence in constant time. As a rough comparison, the longer simulations would finish in minutes when using the rhythmic model, while the forecasting library made them take several hours.

# 6

# Conclusion

Internet applications rely on cloud computing storage systems to store application data. Many of these systems rely on optimistic replication mechanisms to propagate updates between replicas, guaranteeing weaker consistency models, but allowing better latency for operations [CRSSBJPWY08; LFKA11; SPAL11]. These mechanisms cause replicas to diverge, since not all replicas reflect the same updates.

If the divergence is known, bounds can be established on a per-replica basis to guarantee that a replica does not diverge more than a certain bound allows. Systems exist where metrics are used to measure the divergence the replica presents relative to an abstract state that reflects all the updates in the system, the single-copy [YV02; SVF07]. Each replica sets bounds for each metric, and communication is forced between replicas to guarantee that bounds are kept. These systems do not scale to a geo-replicated scenario, because communication between all replicas is needed. Furthermore, applications may have integrity constraints that have to be kept (such as a stock being always positive in an online store, for instance) even if the replica has an inconsistent view of the system. One system aims to keep these invariants in a probabilistic manner, taking into account the consistency level the storage provides versus the monetary cost of providing better consistency for an operation [KHAK09]. This guarantee is provided for server accesses though, not considering client-side updates.

This dissertation presents a metric-based approach to estimating and bounding the divergence, offering probabilistic guarantees that application invariants are kept. While the approach does not provide an exact measurement of the divergence, it allows an estimate with a degree of confidence, which can be calculated locally without contacting other replicas, as well as determining if an operation would break an invariant if executed locally at the client.

To estimate the divergence, a model about the evolution of the object is built. This evolution is generated on the data center side, by gathering statistics about the object when updates are received from the clients. Some information about the data center's own divergence is also kept. These statistics are sent to the clients, and used to estimate the divergence and the single-copy state of the object, and calculate the certainty degree of the estimate.

To bound the divergence, bounds are placed on the metrics. A confidence level is defined with which the bounds must be kept. To guarantee that these bounds are kept, the estimates' certainty degree is used to calculate the probability of the object having diverged to a point that violates the bound. If this calculated probability is greater than or equal to the desired level, then the bound is kept with that confidence level. If not, the client may decide to coordinate with the data center to obtain the most recent updates and reduce divergence. Keeping application invariants is achieved by representing the invariants as metrics' bounds, and using this procedure to check if the bounds are kept with a certain confidence. If they are, the invariant is estimated to be kept.

To fulfill this approach, we developed a model, the rhythmic model. This model treats the evolution of the objects as a simple rate. This growth rate is used with the last seen state of the object to estimate the single-copy. To obtain the certainty degrees, the Poisson distribution is used.

Architecture-wise, the approach is implemented as a middleware with two components: an estimator component, on the clients, and a generator component, on the data centers. The components are placed between the application and storage layers. The generator component gathers statistics about the objects, and sends them to the estimator component periodically. The estimator component uses these statistics to estimate the divergence and to decide if coordination is needed to keep the divergence bounded.

To validate our idea, we implemented our approach in a simulated distributed system, in order to compare the precision of the estimates with the actual divergence, which is not obtainable in a real system. We implemented the approach with the rhythmic model, and with the use of advanced forecasting models [MWH08]. Clients can propagate updates asynchronously, with no delivery guarantees, or in batches of updates.

We evaluated our system with a standard benchmark, TPC-W, which we adapted to the simulator, and to use the estimates. Evaluation shows that our approach can successfully keep the invariant that an item's stock must be greater than 0, for a large number of clients (tens or hundreds), and that most operations can be executed locally at the clients, without the need to coordinate with the data center. The estimates are not as precise when batching of updates is used, mostly due to the data center's divergence used in the estimate. A comparison shows that the rhythmic model fares as well as the advanced forecasting techniques, and with better time complexity.

**Contributions:** In summary, this work contributed with:

- A metric-based approach to probabilistically estimating and bounding the divergence of data in eventually consistent storage systems;
- A method for representing application invariants as metric bounds, and to probabilistically guarantee these invariants are kept while minimizing communication, even when the system scales to many replicas;
- The rhythmic model, a model that fulfills this approach, relying on the growth rate of objects to estimate the divergence of replicas, and if an operation breaks an invariant. The rate is also used to obtain a degree of certainty about the estimates;
- A proof-of-concept implementation of the approach, using a simulated cloud storage system to compare the estimated divergence with the real divergence. This implementation features the usage of the rhythmic model, as well as the usage of advanced forecasting models.
- A comparison between the estimates obtained with the rhythmic model and the ones obtained with the forecasting techniques, using a standard benchmark application, TPC-W. Besides evaluating the accuracy of the estimates, this study also showed how this approach permits offline operations by the clients while keeping invariants.

## 6.1 Future Work

This work presents and tests an approach to estimating the divergence between replicas in a distributed system and providing probabilistic guarantees about keeping the integrity constraints of applications. The evaluation supports the concept, showing that two models that use this approach succeed in doing this, making it useful for cloud storage systems. With this in mind, the next step is implementing this approach on an existing system – SwiftCloud [BP12] is a perfect candidate. It uses Conflict-free Replicated Data Types (CRDTs) to guarantee that states converge and is a geo-replicated system with many clients that can easily cache data on the client-side and make use of the estimates. This would allows us to verify that the estimates behave in a real system with the same precision as in a simulation.

Although not possible to test in the simulator, one thing that may be a problem in a real system with this approach, is that for many objects and many clients, a huge amount of communication may exist in the network when coordination is necessary. An interesting study to do if implemented in a real system, would be how this possible excess of communication may influence latency between clients and data centers, and if so, how to tweak the approach to deal with it.

In this work we only used the Poisson distribution as a way to obtain certainty degrees. This distribution is useful assuming an average rate of updates, however, an interesting study could be made if some other distributions allow for a better representation of the rate of updates, and if a model could be built with those distributions in mind. Datasets exist taken from real-world applications that rely on large-scale distributed systems, such as twitter[1]. If such datasets represented operations that followed a certain distribution, an interesting evaluation could be made taking the operations metric into account, such as estimating how many tweets were not seen at a certain time, and comparing estimates made with different distributions against the real distribution of operations, for instance.

Another possible use of this approach would be in an epsilon serializability context [RP95]. Epsilon serializability (ESR) differs from classic serializability in the sense that it allows some execution orders that would typically violate serializability, assuring that inconsistency is provided only within some *bounds*, taking into consideration *preconditions* that operations must guarantee. ESR could be provided in a distributed system by making use of our approach, mapping the operations' preconditions to application invariants, while relying on our metrics to estimate and bound inconsistency of accesses.

Useful in other contexts as it may be, our approach is not without its flaws. As noted in the evaluation, the precision of the estimates when used with batching mechanisms is not as good as with asynchronous propagation. The main failure point for this is the data center's divergence being considered as an average. In a future iteration we would like to tackle this in a way that allows the data center's estimate to be done with low time complexity, but that can be more precise when updates come in batches of variable size. Though not a flaw, it would also be interesting that the estimate took into account what data centers got more activity on a certain object, allowing propagation of updates to be relaxed for data centers with less activity.

[1] `http://trec.nist.gov/data/tweets/`

# Bibliography


[ANBKH95] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. "Causal memory: Definitions, implementation, and programming". In: *Distributed Computing* 9.1 (1995), pp. 37–49.

[BVFHS12] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. "Probabilistically bounded staleness for practical partial quorums". In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 776–787. ISSN: 2150-8097.

[BBCFKLLLLY11] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. "Megastore: Providing scalable, highly available storage for interactive services". In: *Proc. of CIDR*. 2011, pp. 223–234.

[Bal12] V. Balegas. "Key-CRDT Stores". MA thesis. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.

[BP12] V. Balegas and N. Preguiça. "SwiftCloud: replicação sem coordenação". In: *INForum* (2012).

[BBGMOO95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A critique of ANSI SQL isolation levels". In: *SIGMOD Rec.* 24.2 (May 1995), pp. 1–10. ISSN: 0163-5808.

[Bre00] E. A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 7–. ISBN: 1-58113-183-6.

[CPW07] L. Camargos, F. Pedone, and M. Wieloch. "Sprint: a middleware for high-performance transaction processing". In: *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), pp. 385–398. ISSN: 0163-5980.

[CB74] D. D. Chamberlin and R. F. Boyce. "SEQUEL: A structured English query language". In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. SIGFIDET '74. Ann Arbor, Michigan: ACM, 1974, pp. 249–264.

[Cod70] E. F. Codd. "A relational model of data for large shared data banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782.

[CRSSBJPWY08] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. ISSN: 2150-8097.

[Cor+12] J. C. Corbett et al. "Spanner: Google's globally-distributed database". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264. ISBN: 978-1-931971-96-6.

[CDKB11] G. F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Fifth. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2011. ISBN: 0132143011.

[Cou02] T. P. P. Council. *TPC Benchmark W (Web Commerce) Specification*. 2002.

[DHJKLPSVV07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980.

[FZFF10] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. "SPORC: group collaboration using untrusted cloud resources". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–.

[GDCCSXDSLLL12] M. Gagnaire, F. Diaz, C. Coti, C. C'erin, K. Shiozaki, Y. Xu, P. Delort, J.-P. Smets, J. Le Lous, S. Lubiarz, and P. Leclerc. *Downtime statistics of current cloud solutions*. Tech. rep. The International Working Group on Cloud Computing Resiliency, 2012, p. 2.

[GG03] D. F. García and J. García. "TPC-W e-commerce benchmark evaluation". In: *Computer* 36.2 (2003), pp. 42–48.

[GL02] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700.

[GPO11] P. Gomes, J. Pereira, and R. C. M. d. Oliveira. "An object mapping for the Cassandra distributed database". In: (2011).

[Hai67] F. A. Haight. *Handbook of the Poisson distribution*. Wiley New York, 1967.

[HW90] M. P. Herlihy and J. M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925.

[Ili13] A. Ilienko. "Continuous counterparts of Poisson and binomial distributions and their properties". In: *arXiv preprint arXiv:1303.5990* (2013).

[JGH07] C. Jay, M. Glencross, and R. Hubbold. "Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment". In: *ACM Trans. Comput.-Hum. Interact.* 14.2 (Aug. 2007). DOI: 10.1145/1275511.1275514. URL: http://doi.acm.org/10.1145/1275511.1275514.

[Klo10] R. Klophaus. "Riak Core: building distributed applications without shared state". In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 14:1–14:1. ISBN: 978-1-4503-0516-7.

[KHAK09] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. "Consistency rationing in the cloud: pay only when it matters". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 253–264. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1687627.1687657.

[Lam79] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Trans. Comput.* 28.9 (Sept. 1979), pp. 690–691. ISSN: 0018-9340.

[Lam78] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782.

[Lam86] L. Lamport. "On interprocess communication". English. In: *Distributed Computing* 1 (2 1986), pp. 86–101. ISSN: 0178-2770.

[Lam98] L. Lamport. "The part-time parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071.

[LPCGPR12] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making geo-replicated systems fast as possible, consistent when necessary". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278. ISBN: 978-1-931971-96-6.

[LM06] G. Linden and M. Mayer. *Marissa Mayer at Web 2.0*. Nov. 2006. URL: http://glinden.blogspot.pt/2006/11/marissa-mayer-at-web-20.html.

[LFKA11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 401–416. ISBN: 978-1-4503-0977-6.

[LFKA13] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Stronger Semantics for Low-Latency Geo-Replicated Storage". In: *Proc. 10th USENIX NSDI*. Lombard, IL, Apr. 2013.

[MWH08] S. Makridakis, S. C. Wheelwright, and R. J. Hyndman. *Forecasting methods and applications*. John Wiley & Sons, 2008.

[NAEA13] F. Nawab, D. Agrawal, and A. El Abbadi. "Message Futures: Fast Commitment of Transactions in Multi-datacenter Environments." In: *CIDR*. 2013.

[O'N86] P. E. O'Neil. "The Escrow transactional method". In: *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pp. 405–430. ISSN: 0362-5915. DOI: 10.1145/7239.7265. URL: http://doi.acm.org/10.1145/7239.7265.

[PPRSWWCEKK83] J. Parker D.S., G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of Mutual Inconsistency in Distributed Systems". In: *Software Engineering, IEEE Transactions on* SE-9.3 (1983), pp. 240 –247. ISSN: 0098-5589.

[PA04] C. Plattner and G. Alonso. "Ganymed: scalable replication for transactional web applications". In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Middleware '04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pp. 155–174. ISBN: 3-540-23428-4.

[PMCD03] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. "Reservations for Conflict Avoidance in a Mobile Database System". In: *Proceedings of the 1st international conference on Mobile systems, applications and services*. MobiSys '03. San Francisco, California: ACM, 2003, pp. 43–56.

[RP95] K. Ramamritham and C. Pu. "A Formal Characterization of Epsilon Serializability". In: *IEEE Trans. on Knowl. and Data Eng.* 7.6 (Dec. 1995), pp. 997–1007. ISSN: 1041-4347.

[SS05] Y. Saito and M. Shapiro. "Optimistic replication". In: *ACM Computing Surveys* 37.1 (Mar. 2005), pp. 42–81. ISSN: 03600300.

[SVF07] N. Santos, L. Veiga, and P. Ferreira. "Vector-field consistency for ad-hoc gaming". In: *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*. MIDDLEWARE2007. Newport Beach, CA, USA: Springer-Verlag, 2007, pp. 80–100. ISBN: 3-540-76777-0, 978-3-540-76777-0.

[SB] E. Schurman and J. Brutlag. *Performance Related Changes and their User Impact*. Presented at Velocity Web Performance and Operations Conference, June 2009.

[SPBZ11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types". In: *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 978-3-642-24549-7.

[STT08] L. Shrira, H. Tian, and D. Terry. "Exo-leasing: escrow synchronization for mobile clients of commodity storage servers". In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '08. Leuven, Belgium: Springer-Verlag New York, Inc., 2008, pp. 42–61. ISBN: 3-540-89855-7.

[SPAL11] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. "Transactional storage for geo-replicated systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 385–400. ISBN: 978-1-4503-0977-6.

[TTPDSH95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing update conflicts in Bayou, a weakly connected replicated storage system". In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 172–182. ISSN: 0163-5980.

[TDPSTW94] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. "Session guarantees for weakly consistent replicated data". In: *Proceedings of the third international conference on on Parallel and distributed information systems*. PDIS '94. Autin, Texas, USA: IEEE Computer Society Press, 1994, pp. 140–150. ISBN: 0-8186-6401-0.

[Vog09] W. Vogels. "Eventually consistent". In: *Commun. ACM* 52.1 (Jan. 2009), pp. 40–44. ISSN: 0001-0782.

[WC95] G. Walborn and P. Chrysanthis. "Supporting semantics-based transaction processing in mobile database applications". In: *Reliable Distributed Systems, 1995. Proceedings., 14th Symposium on*. 1995, pp. 31 –40.

[YV02] H. Yu and A. Vahdat. "Design and evaluation of a conit-based continuous consistency model for replicated services". In: *ACM Trans. Comput. Syst.* 20.3 (Aug. 2002), pp. 239–282. ISSN: 0734-2071.