DEPARTMENT OF
COMPUTER SCIENCE

TIAGO ANDRÉ ENVIA MATOS

PARTIAL REPLICATION WITH STRONG CONSISTENCY

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2022

DEPARTMENT OF
COMPUTER SCIENCE

PARTIAL REPLICATION WITH STRONG CONSISTENCY

TIAGO ANDRÉ ENVIA MATOS

Adviser: Nuno Preguiça
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2022

To my family. Thank you.

Acknowledgements

I want to start by thanking my advisor Prof. Nuno Preguiça for guiding me since the beggining and always being available to help. An year ago I could not even imagine myself doing a project like this. Couldn't have done it without him. Also want to thank colleague Pedro Fouto for lending me his research and work on ChainPaxos which was an integral part of this dissertation. He always helped with any questions I had and he's just a cool guy overall.

I too want to thank all the teachers I had over these five long years for all their hard work. I learned a lot thanks to them. They also gave me some of the greatest challenges I've ever faced, but I'd be lying if I said they didn't feel good to conquer.

I also must thank my friends who have been beside me for so long. Could always count on them for a nice coffee after an hard day of work, or to procrastinate together as we play some good video-games.

And finally, but most important, I want to thank my family. I want to thank my father, my mother, my brother and my sister who have always supported me, during the good and the bad times. Without them I could never have got this far. Thank you.

Abstract

In response to the increasing expectations of their clients, cloud services exploit geo-replication to provide fault-tolerance, availability and low latency when executing requests. However, cloud platforms tend to adopt weak consistency semantics, in which replicas may diverge in state independently. These systems offer good response times but at the disadvantage of allowing potential data inconsistencies that may affect user experience.

Some systems propose to adopt solutions with strong consistency, which are not as efficient but simplify the development of correct applications by guaranteeing that all replicas in the system maintain the same database state. Therefore, it is interesting to explore a system that can offer strong consistency while minimizing its main disadvantage: the impact in performance that results from coordinating every replica in the system. A possible solution to reduce the cost of replica coordination is to support partial replication. Partially replicating a database allows for each server to only be responsible for a subset of the data - a partition - which means that when updating the database only some of replicas have to be synchronized, improving response times.

In this dissertation, we propose an algorithm that implements a distributed replicated database that offers strong consistency with support for partial replication. To achieve strong consistency in a partially replicated scenario, our algorithm is in part based on the Clock-SI[10] research, which presents an algorithm that implements a multi-versioned database for strong consistency (snapshot-isolation) and performs the Two-Phase Commit protocol when coordinating replicas during updates. The algorithm is supported by an architecture that simplifies distributing partitions among datacenters and efficiently propagating operations across nodes in the same partition, thanks to the ChainPaxos[27] algorithm.

Keywords: distributed systems, consistency, partial replication

Resumo

Como forma de responder às expectativas cada vez maiores dos seus clientes, as operadoras cloud tiram partido da geo-replicação para oferecer tolerância a falhas, disponibilidade e baixa latência dos seus sistemas na resposta aos pedidos. No entanto, as plataformas cloud tendem a adotar uma semântica de consistência fraca, na qual as réplicas podem variar em estado de forma independente. Estes sistemas oferecem bons tempos de resposta mas com a desvantagem de que têm de lidar com potenciais inconsistências nos dados que podem ter impacto na experiência dos utilizadores.

Alguns sistemas propõem adotar soluções com consistência forte, as quais não são tão eficientes mas simplificam o desenvolvimento de aplicações ao garantir que todas as réplicas do sistema mantêm o mesmo estado da base de dados. É então interessante explorar um sistema que garanta replicação forte mas que minimize a sua principal desvantagem: o impacto de performance no momento de coordenar o estado das réplicas nos sistema. Uma possível solução para reduzir o custo de coordenação das réplicas durante transações é o suporte à replicação parcial. Replicar parcialmente uma base de dados permite que cada servidor seja apenas responsável por uma parte dos dados - uma partição - o que significa que quando são realizadas escritas apenas algumas das réplicas têm de ser sincronizadas, melhorando os tempos de resposta.

Neste trabalho propomos um algoritmo que implementa um sistema de armazenamento distríbuido replicado que oferece consistência forte com suporte a replicação parcial. A fim de garantir consistência forte num cenário de replicação parcial, o nosso algoritmo é em parte baseado no algoritmo Clock-SI[10], que implementa uma base de dados parcial com multi-versões para garantir consistência forte (snapshot-isolation) e que realiza o protocolo Two-Phase Commit para coordenar as réplicas no momento de aplicar escritas. O algoritmo é suportado por uma arquitectura que torna simples distribuir partições por vários centros de dados e propagar de forma eficiente operações entre todos os nós numa mesma partição, através do algoritmo ChainPaxos[27].

Keywords: sistemas distribuidos, consistência, replicação parcial

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Modern online applications are evolving to large scale architectures that target an increasingly larger number of users across the globe. These applications are expected to have low response times and always be online. Distributing data across different nodes improves performance because different replicas may process user requests concurrently. It also provides fault-tolerance because, when a replica fails, another replica may just take over the work leading to a system that is always available. For these reasons, most services nowadays choose to geo-replicate their system. Geo-replication allows the system to be spread across multiple replicas across the globe, possibly closer to users of each geographic location decreasing client latency times.

Replicating the system provides availability and fault-tolerance, however, it comes with the cost of consistency. As the number of replicas increases, coordinating the replicas to keep integrity of the data becomes increasingly difficult. In fact, the CAP theorem [5] has demonstrated that, in a distributed system prone to network partitions, it is impossible to provide both availability and strong consistency. Because of this, many commercial systems nowadays decide to sacrifice consistency in favor of availability and better client response times. However, application correctness might become vulnerable since replicas may diverge when there are multiple concurrent updates. This divergence may lead to violations of critical system invariants, and makes it harder for programmers to reason about the system's state and develop correct implementations for an application.

Still, despite its performance issues, there are systems that provide strong data consistency even in a globally-replicated setting. This is usually the case for applications where maintaining data integrity is critical. A common example are banking systems, as they must assure that no transaction ever violates system invariants, like not allowing an account to spend more money than its current balance. Even applications that do not require always consistent data may implement strong consistency for system correctness and easier application development.

Nowadays, the modern demands and expectations from clients have led cloud providers to highly increase their number of datacenters. As time passes, the challenge of providing fault-tolerance while keeping replicas reasonably synchronized as the number of datacenters increases has only become more difficult.

1.2 Motivation

As the number of datacenters increases, there is a need for solutions that provide strong consistency while still showing good performance and availability in a global scale. One way to achieve this is through partial replication. In a partially replicated database, each replica in the system only stores a subset of the data. When some data object is updated, only the replicas responsible for that object have to be coordinated, which significantly decreases latency between replicas. Partial replication also pairs well with geo-replication, because it allows data to be stored only in places where it is likely to be accessed. A common example are social networks, where the data accessed by users is largely dependent on their regions.

Despite the performance potential of partial replication, most distributed systems use full replication because it offers higher availability and is easier to implement and coordinate compared to partial replication. However, full replication requires all replicas to participate in every transaction which hurts latency. To compensate, programmers feel forced to use weaker consistency models that commit transactions more quickly, at the cost of system correctness.

As such, there is interest in a model that can guarantee strong consistency in a globally replicated setting, possibly optimized by partially replicating data. The main challenge comes from trying to coordinate concurrent transactions in a setting where no replica is aware of the entire database, which may lead to inconsistencies. Some systems guarantee consistency between data in the same partition but not in separate partitions, and usually rely on the application to define the boundaries of each partition [2]. Spanner [7] offers good scalability in a globally-distributed setting and partitions the data in *shards* stored in different servers, still, each datacenter stores a full replica of the data. It also requires many cross-datacenter communication steps to coordinate datacenters [22], by requiring one of the datacenters to work as a Two-Phase Commit coordinator. Spanner also has the disadvantage of relying on external machines to assign global timestamps to transactions, which is bad for availability. Blotter [23] presents an approach similar to Spanner's that shows good throughput in a global scale, but also requires full replication and provides a consistency model weaker than Snapshot Isolation.

1.3 Proposed Solution

We propose a strongly consistent partial replication protocol that tries to tackle the relevant limitations of the solutions described above while still being responsive. Like previous work done in this regard (ex. Spanner), our algorithm is supported by an architecture that allows partitions to be distributed across different datacenters that communicate with each other through some Paxos-based protocol. Unlike previous solutions, however, partitions may be arbitrarily assigned to a datacenter, and no datacenter is required to store the entirety of the data.

Our main improvement to previous solutions is reducing the amount of inter-datacenter communication required to replicate updates across partitions. Previous work selects one of the datacenters to work as a transaction coordinator that has to communicate with the other datacenters to decide whether a transaction should commit or not. Instead, our solution puts the burden of transaction coordination on the client, using a specialized API on the client side. This helps reduce the workload on the side of the datacenter, improving throughput and possibly reducing response times.

Another improvement we introduce is the use of the ChainPaxos[27] protocol to replicate operations across nodes in the same partition. In comparison to other famous Paxos-based protocols, ChainPaxos shows better throughput by minimizing the amount of messages that have to be sent over the network by having a designated leader that does not need to communicate with every other server in the cluster - we present an explanation in section 2.2.

Finally, our algorithm adapts the work of the Clock-SI[10] paper, which presents an implementation of a partitioned multi-versioned database offering snapshot-isolation consistency. It also explains how to implement a database that derives snapshot and commit timestamps from loosely synchronized clocks, rather external machines like in Spanner.

1.4 Document Organization

This document is organized as follows:

- **Chapter** 2 covers basic distributed system concepts related to the work here presented and discusses proposed solutions in the literature.
- **Chapter 3** describes our algorithm and the architecture supporting it.
- **Chapter 4** presents the results of our experimental evaluation, comparing the performance of our algorithm across different test configurations.
- In **Chapter 5** we discuss and summarize what was accomplished and the conclusions derived from our solution.

2 Related Work

This section introduces related work, starting by an introduction to replication models, followed by replication algorithms used for enforcing strong and weak consistency in distributed systems. The section proceeds by addressing database replication and algorithms used for database replication.

2.1 Replication

A distributed system manages data that is often replicated for allowing the system to tolerate faults and provide high availability. In this section we address different aspects related with replication.

2.1.1 Consistency Models

A consistency model defines a set of safety properties that decide the order and coherency of updates that can be seen by a client when contacting a server. It specifies a contract between the client and the system, where the system guarantees that the data read will be consistent as long as the client follows the rules of the model. Each model comes with its own trade-offs and programmers must decide which is the most suited solution when designing a distributed system.

Multiple consistency models have been proposed in literature. We now present the most relevant ones.

2.1.1.1 Strong Consistency

A strong consistent model guarantees that all clients accessing the system read every update operation in the same order, ensuring they always observe a consistent state of the system. The system acts as if there was only a single copy of the data, making it easy to reason about its evolution and develop solutions based on it. Strong consistency is the adequate choice when having an always consistent and up-to-date state is essential to the overall system correctness. However, coordinating every replica can be slow and not

acceptable for services that focus on low response times.
Some of the models that fit strong consistency are:

Linearizability: Linearizability [15] is a guarantee about single operations on single objects. It provides **atomicity** and a total order guarantee over writes that happen in the system, based on the real-time these write requests where issued.

In distributed systems, atomicity states that, for every write sent to the system, there is a point in time (**serialization point**) between the client sending the write request and receiving the reply from the system, where we can consider the write operation to be completed. After this point, any read operation should see this write, and any new updates to the system should be ordered after this write.

Serializability: Serializability [1, pp.812] is a guarantee about one or more operations over one or more objects (usually called transactions, more on that on Section 2.3.1). It guarantees that the execution of a set of transactions over multiple data objects follows some total order that may or may not be based on the real-time order these operations were seen by the system. To achieve this, the system may reorder transactions, to possibly avoid inconsistencies or deadlocks.

2.1.1.2 Weak Consistency

In weak consistency model, replicas apply updates and execute reads without coordinating with other replicas, which allows clients to see inconsistent states of the database. Operations may not be seen by every replica, and clients might read out-of-date values or in different orders. These models trade consistency for availability and lower latency of communication between replicas, and are usually the choice when the system is dependent on fast response times. Some of the models that fit weak consistency are:

Eventual Consistency: The eventual consistency model [31] tries to achieve high availability by giving no guarantees about the state of the system, clients are allowed to see any writes made in a replica and in any order. This model only provides a liveness guarantee that states that, if no writes are received for a long enough period of time, eventually replicas will converge to the same state. The way this convergence is done is usually a design choice, as an example, the replicas may converge to the last write ever seen in a replica (last-writer-wins approach), or they may follow some merge-procedure that merges the state of every replica.

Causal Consistency: Causal consistency [18] is one of the strongest of the weak consistency models and guarantees that implicit dependencies between operations are respected across every replica. To achieve this, the system has to keep track of causal dependencies between operations, and ensure that clients always see updates in an order that respects their causal dependencies.

Because the system only guarantees consistency between causally dependent operations, concurrent operations that have no relation with each other may be seen out-of-order, allowing users to see the system in different and inconsistent states, weakening consistency.

Some consistency models, such as Red-Blue consistency [20], have combined weak and strong consistency in the same systems, by allowing applications to specify which operations should run under each consistency model.

2.1.1.3 CAP Theorem

The CAP theorem [5] states that it is impossible for a distributed system to simultaneously provide:

- **Strong Consistency**: Every client always reads the most recent write (consistent data).
- **Availability**: The system is always available to process client requests even in the presence of failures.
- **Partition-Tolerance**: The system stays functional even in the presence of network-partitions.

The theorem also states that, while providing all three properties is impossible, a system can guarantee any two of the three properties. However, this assertion has proven to be misleading [4]. Network partitions are unavoidable in large scale systems, and because in a partitioned distributed system it is impossible to guarantee both consistency and availability, distributed applications must choose a combination of partition-tolerance and one of the other two properties.

2.1.2 Replica Location

The distance between replicas of a distributed system is one of the main factors that influence latency times of communication between replicas.

Co-located: Co-located replicas are placed on the same physical location. Having replicas close to each other allows faster communication between them and reduces latency times. These systems are usefull when clients are geographically close, because latency to clients would increase when trying to serve users globally.

Geo-replication: Geo-replicated replicas are distributed at various geographic locations that might span long distances between replicas. The longer distance may lead to an increase in latency when coordinating replicas, but it allows replicas to be placed closer to clients across the globe, decreasing latency between a user and a replica. Since most distributed applications nowadays are globally distributed and focused on user experience, this trade-off is usually acceptable.

2.1.3 Data Redundancy

Systems must decide how many copies of the data do they need. They may choose to copy all data in every node, or to have fewer copies of each data object located in a few select nodes.

Full Replication: Fully replicated systems replicate the entire data set across all replicas. Fully replicated systems guarantee that no data is lost as long as one replica that executed every update survives. These systems provide reliable fault-tolerance, at the cost of sending large, possibly redundant, amounts of data to every replica in the system.

Partial Replication: In partially replicated systems, each replica is only responsible for a subset of the data, which means that the amount of copies of a data object is actually less than the amount of replicas. This model provides less reliable fault-tolerance, but having to distribute data across less replicas is proven to lead to better performance and scalability [17]. Partial replication is usually combined with Geo-replication to provide replication of specific parts of data only in places where clients are likely to acess it. For example, it would make sense to store partial data related to european users of a social media in a server in Europe, as they are much more likely to access it instead of, for instance, an american user.

2.1.4 Active and Passive Replication

Defines if every node in the system should execute an operation, or if only one node executes the operation and the other replicas just store the result.

Active Replication: In an active replicated system, all replicas execute the requested operations. The replica that received the operation broadcasts it to every replica in the system, or the client may send the request to all replicas directly. Because every replica has to execute the operation locally, only deterministic operations are allowed so the replicas do not diverge in state. **State Machine Replication** is a special case of active replication that must also respect the total order of operations across all replicas when operations are not commutative.

Passive Replication: In a passive replicated system, only the replica that received the operation processes it, and then delivers the result to other replicas. Because only of the replicas executes the operation, passive replication may be used for non-deterministic operations, but unlike active replication, only having one replica execute operations may lead to load imbalance.

2.1.5 Synchronous and Asynchronous Replication

Also one of the main factors that influences response times. Decides if the coordination between nodes in the system is done while executing an update, or is the update propagated asynchronously to other replicas after the system already replied to the client.

Synchronous Replication: In a synchronous system, operations are only considered complete after replicated in all (or a majority of) replicas. When a server receives an operation from a client, it immediately sends it to other replicas and only when sure a majority of replicas in the system have processed the operation, the server replies to the user. This form of replication is usually associated with strong consistency models, since it ensures at all times the state between replicas is consistent. However, executing each update synchronously is costly and operations may take a long time executing, hurting scalability.

Asynchronous Replication: In an asynchronous system, when a server executes an operation, it immediately replies to the client that requested it. The operation is then eventually propagated to every other replica asynchronously. Because a server does not have to wait for a response from a majority of replicas before replying to a user, asynchronous systems have much lower response times. However, because there is no coordination between writes in different replicas, these protocols often allow replicas to diverge in state. This form of replication is mostly associated with weak consistency models, since it focus on lower response times and availability at the cost of weakening consistency. It often scales better than stronger models and is usually the choice for large-scale applications that focus on user experience.

2.1.6 Single Master and Multi-Master

Defines if every replica in the system is available to answer client updates, or should clients only request updates from a single node working as a leader.

Single-Master: In single-master systems (also called **Primary-Backup**), one of the servers in the system is designated as a leader that receives every operation coming from the clients. This server has complete control over the system and the other replicas work as backups that replicate the data. To distribute some of the workload, read-only operations may be sent to the backups, at the risk of reading stale values. When the leader fails, some leader election procedure must take place to choose a new leader from the remaining backups. These systems are usually used for implementing strong consistency models, since it's easier to maintain consistency over replicas when only one server receives operations. However, this protocol lacks scalability because adding more replicas does not increase performance (it actually reduces it, because the leader will have to wait for more replicas), and because a failure of the leader means the whole application will be down while a new leader is being elected, it provides weak availability.

Multi-Master: In multi-master systems, any server in the system may process client requests, it's up to the user to choose which replica to connect to (usually the closest one geographically). After finishing the operation, the server will broadcast the operation to other replicas asynchronously. Having more servers able to answer requests leads to better availability, and allowing clients to communicate with the servers closer to them leads to better response times, helping scalability. However, it becomes very difficult to synchronize replicas and so divergence is common and these systems usually implement weak consistency models. Still, there are systems that implement strong consistency models in a multi-master system, with the help of coordination protocols like Paxos (Section 2.2).

2.2 Replication Protocols

For maintaining the state of the replicas, a number of replication protocols have been proposed in literature. In this section, we overview some replication protocols/systems.

For an example of a weakly consistent replication protocol, **Dynamo**[8] is an eventually consistent NoSQL database. Each object is identified by a key, and the system only exposes two operations: get() and put(). For providing fault-tolerance, each object is replicated in N replicas, through consistent hashing, these replicas are placed in a ring and each replica is responsible for a subset of the key-space. Each object is also stored with *context* data, which allows the system to find and re-conciliate divergent versions of the same object in different replicas. Divergence comes from allowing any node to receive updates from clients which means that different nodes may end up seeing data in different and concurrent states. When contacting the system, the *context* object acts as a vector clock that allows the system to detect and merge concurrent operations.
The following are all replication protocols that offer stronger consistency levels.

Chain Replication[30] is a primary-backup approach that offers good throughput and availability without compromising strong consistency guarantees. In this protocol, servers are linearly ordered as chain. All object updates are received by the head of the chain, and then propagated and processed along the chain up to the tail. Query operations are all processed by the tail. Write operations perform better because the head does not have to replicate the operation to a majority quorum, it just has to send the result to the next backup in the chain, and so on. Strong consistency is guaranteed because query requests and update requests are all processed serially in a single server, the tail.
One of the challenges of Chain Replication, however, is that it does not have a fair distribution of workload, since the head receives all the updates and the tail all the queries. Moreover, the analysis Fouto et. al [20] as shown that, in the presence of network partitions in an asynchronous model, Chain Replication may violate linearisability.

Consensus Algorithms solve a *relaxed* variant [12] of the consensus problem, and are used for implementing strongly consistent systems where any replica may try to execute update operations. Consensus Algorithms ensure replicas will always decide to execute the same operations during the same instances and in the same order, allowing for the implementation of **State Machine Replication** (Section 2.1.4).

Undeniably, **Paxos**[19] is the most widely known consensus algorithm. In Paxos, a set of of *proposer* servers try to propose operations to a set of *acceptor* servers, which will decide among all the proposed values which one should be applied to every server. When a decision is made, the acceptors send the decided value to a set of *learner* servers, to be replicated (the same servers may act as proposers, acceptors and learners simultaneously). It is of note though, that the classic Paxos algorithm is only able to learn a single value per instance, thus a complete Paxos instance has to completed for each value to be decided in the state machine.

Since its conception, multiple variations of Paxos have appeared that try to optimize the original Paxos protocol. One of the most famous variations is **Multi-Paxos** [9], which distinguishes one of the proposers as leader and only allows the leader to propose values. This allows Multi-Paxos to decide values in one less synchronization phase (as long as the leader stays the same) compared to the original Paxos protocol.

Raft[25] is a consensus algorithm that, besides offering strong consistency with good performance, promises to be an easier to understand protocol compared to Paxos. Paxos as been regarded as a somewhat difficult to understand algorithm with vague explanations, which may lead to wrong implementations in real systems. Raft promises a better explained, easier to understand algorithm, making it a better platform for building real systems. The Raft algorithm relies on a replicated log that keeps information about previously executed operations. It also depends on a leader that has responsibility of managing the log and the state of the other replicas.

EPaxos[24] aims to provide high availability and performance by not having a designated leader process. Clients can choose which replica to submit a command and, if there is no interference with another concurrent command, it will be committed. This allows the system to evenly distribute the load across all replicas, eliminating the bottleneck of only having one server receive all requests like in Multi-Paxos. In EPaxos, commands that do not interfere with any other will be committed after a single round of communication (*fasthpath*), if there is an interference, then an extra step of communication is done where the command proposer has to send an accept message to a majority of replicas (similar to the accept phase in Paxos).

Atlas[11] is another leaderless consensus algorithm, similar to EPaxos. It allows parameterization of the number f of allowed failures. A smaller f results in smaller quorums, thereby decreasing latency (at the cost of fault-tolerance). Also, depending on f, it allows concurrent transaction to follow the *fastpath* even when they conflict. In fact, if $f = 0$, every transaction will follow the *fastpath*.

ChainPaxos[27] offers high throughput replication by ordering replicas in a chain. A distinguished leader acts as head of the chain and receives client operations, which it then sends to the next node in the chain and so on. When the operation has passed trough a majority of nodes, a response is sent to the client. This chain topology allows ChainPaxos to integrate membership managing without the use of external coordination devices, and allows for linearizable reads in any replica without communication overhead.

Solutions like EPaxos and Atlas allow operations to execute in a single round, however, these protocols still require the nodes executing a write operation to send and receive $O(n)$ messages, unlike ChainPaxos that has $O(1)$ message complexity. These protocols try to distribute the load imposed on the leader to multiple replicas. In contrast, ChainPaxos strives to minimize the load imposed by the protocol while still balancing the work between replicas.

The main goals of ChainPaxos are: to minimize the number of messages each node processes in a fault-free run to maximize throughput; and to integrate an efficient fault-tolerance system, without relying on an external system, by taking advantage of the Paxos messages. Figure 2.1 illustrates the ChainPaxos message flow in a fault-free run.

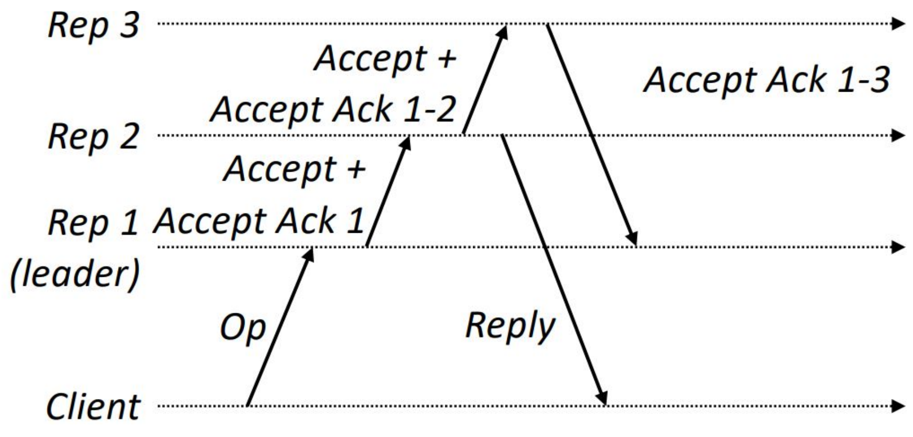


Figure 2.1: ChainPaxos message flow [13]

When it receives an operation, the leader sends an *accept* message, with its own *ack* included, to the next replica. Upon receiving the accept message, the receiving replica adds its own ack and forwards the accept message to the next replica, this cycle repeats until the end of the chain. When the tail receives the accept message, it directly sends to the head a message with the acks of all replicas. This step is necessary to make sure the head learns the decided value. Additionally, to inform the replicas in the chain that did not see enough acks at the time to decide a value, the next accept message sent by the head piggybacks this information.
Meanwhile, when the accept reaches the replica in the middle of the chain, it will include acks from a majority quorum. Thus, the replica in the middle knows a value has been decided, and will execute the request and return the result to the client.

Because ChainPaxos is just using a different communication pattern to convey the messages of Multi-Paxos, it can fall back to the regular two phases of Paxos to handle faults and manage membership. For detecting faults, each replica expects to receive periodic keep-alive messages from the following node. Also, if a replica does not receive accept messages for a long period of time, it will suspect the leader is faulty and attempts to take leadership. Reconfiguration of membership after a fault involves deciding the instance the new membership is active, by executing the ChainPaxos protocol in the functioning replicas and piggybacking additional information in the accept messages.

In our project, ChainPaxos will be the main mechanism to replicate operations among nodes in the same partition.

2.3 Database Replication

Database systems are central in modern applications and systems, by managing the data of the application. Unlike simple data stores, database system support transactions, which require specific algorithms for executing them. In this section we introduce database replication.

2.3.1 Transactions

Even in strongly consistent systems, some operations cannot be guaranteed to perform correctly by just replicating simple, single operations in single objects. For example, imagine a distributed banking system and a client that tries to transfer money from acount A to account B. This situation can be represented as the following sequence of (isolated) operations:

- Removing 100$ from account A
- Adding 100$ to account B

Now what happens if the first operation concludes but second one fails? Account B will still have the same amount of money it had before the transfer, while account A lost 100$ that just disappeared from the system. Even in this situation, strong consistency may not have been violated, because if an operation fails (in this case, the second one), it just means that it was not applied in any replica. The real problem is that the system allowed the first operation to complete and have a visible effect.

As such, there is need for an abstraction that allows to group two or more operations in a single meta-operation, and guarantees that either all operations are successfully completed or none of them are. This abstraction is called **transaction**, and it is the main logical mechanism that allows manipulating data in a replicated database system. Coming from the database community, transactions provide the **ACID** properties [1, pp. 799-804]:

- **Atomicity**: All operations in a transaction complete successfully or none of them do.
- **Consistency**: The state of the database respects all invariants of the data model before and after the execution of a transaction.
- **Isolation**: Transactions execute without being aware of other concurrent transactions.

- **Durability**: The effects of a transaction that terminates successfully are not lost, even if failures occur.

2.3.2 Isolation levels

Each transaction executes under an isolation (from ACID) level, which determines how it may be affected by other concurrent transactions. Transactions under a strong isolation level cannot see changes from other transactions, while relaxed isolation levels allow a transaction to have its execution affected by the updates of concurrent transactions, which may lead to wrong results.

Serilizability is the strongest of isolation levels, and guarantees that execution of concurrent transactions is equivalent to an ordered execution of these same transactions, one at a time. If no ordering between transactions is possible, then one of them should be aborted. However, ensuring a completely isolated environment for every transaction requires a lot of effort by the concurrency control protocol, and so serializability should only be used in databases where data integrity is crucial, even at the cost of performance.

To overcome this, more relaxed isolation levels have been proposed like **Snapshot Isolation**, **Cursor Stability**, **Repeatable Read** and **Read Commited** [3]. Applications should choose the most appropriate isolation level, leveraging throughput in exchange of isolation between transactions.

Most commercial databases implement isolation trough the use of locking mechanisms [3]. The Two-Phase Protocol (2PC) [16] is a common example of a locking mechanism used to coordinate transactions in a replicated database system.

2.3.2.1 Snapshot-Isolation

The problems with protocols providing serializability come from the fact that all types of read/write and write/write transaction conflicts have to be considered. In particular, since read/write conflicts tend to be very common in database systems, they limit the potential concurrency and scalability of the system.

Snapshot-isolation is a consistency model in which transactions see a snapshot of the database as it was when the transactions started. In SI readers and writers do not conflict, which has the advantage of allowing queries to be performed without interfering with updates, in fact, replication based on snapshot -isolation is only concerned with update operations. Also, compared to serializability, SI offers a similar level of consistency.

The main disadvantage of SI is that, because transactions perform all operations on a snapshot of the database, if the snapshot is too old, transactions may end up reading stale values that have already been overwritten in the database. Still, this is expected to be a rare occurrence and acceptable because of the potential performance gain.

Snapshot-isolation is the consistency model offered by our algorithm, as we think the pros outweigh its cons.

2.3.3 Database Replication Models

Similar to distributed systems, there are two ways updates may be propagated to replicas[14]:

- **Eager Replication**: Whenever a transaction updates an object, every replica that owns an instance of that object has to be updated for the transaction to commit successfully. Eager systems provide serializable execution and are usually implemented trough locking mechanisms. There are no inconsistencies and no need for reconciliation, as locking detects potential anomalies and converts them to waits or deadlocks. However, because transactions have to wait for updates to be done in every replica, eager replication reduces update performance and increases transaction response times.

- **Lazy Replication**: Only one replica is updated by the originating transaction. Updates to the other replicas are propagated asynchronously, as a separate transaction to each node. This model allows the original transaction to commit faster, because it does not have to wait for commits in other replicas. Of course, if the replica that received the operation crashes before sending it to all other replicas, lazy systems may diverge and have inconsistencies, requiring reconciliation. Reconciliation is usually done trough the use of timestamps or multi-version of objects (like Snapshot Isolation).

Also similar to distributed systems, there are two ways to regulate who receives updates[14]:

- **Master Replication**: Each object has a master node. Only the master can update the object, all other replicas are read-only. If a non-master replica wants to update the object, it must request the master for the update.

- **Group Replication**: Any server with a copy of an object may update. Also called **Update Everywhere**.

2.3.4 Strongly Consistent Database Replication

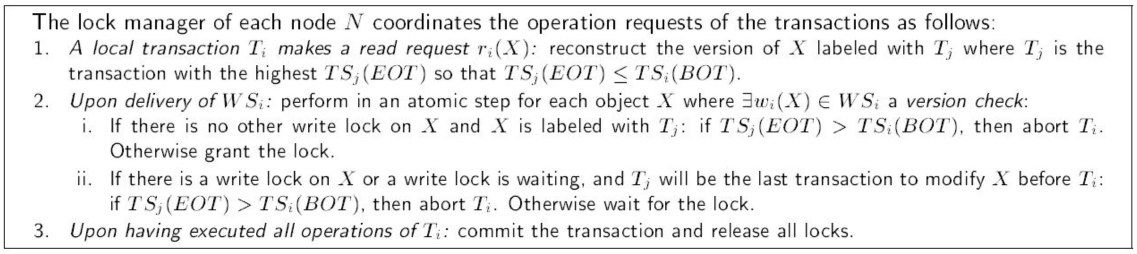
The analysis of [14] concluded that eager solutions generally do not scale and that both eager and lazy replication have high conflict rates. This lead researchers to find solutions that could eliminate the problems of eager and lazy replication while offering one of the stronger isolation levels with good performance. A common approach is to provide an hybrid replication that is both eager and lazy. In this hybrid approach, the system replies to the client after commiting in one replica (as if it was lazy), but there is a coordination step between every replica before the commit (as if it was eager). These solutions usually involve recording the set of write operations done at a replica during a transaction so it can be sent to other replicas for coordination and replication, also called *deferred update*.

These solutions are supported by an atomic broadcast primitive that ensures that all messages are delivered at the same order in all nodes, which guarantees they produce the same outcome for all transactions. Locally, transactions are synchronized trough the use of some concurrency control mechanism, like two-phase locking.

2.3.4.1 1-Copy-Serializability

[16, 17] present similar solutions providing 1-Copy-Serializabitizy (1CS), which is a guarantee that any client accessing the system will see the database as if there is only a single copy providing serializability. In [17], for a transaction T_i invoked at node N, all read operations from T_i are performed at N. The write operations are deferred until all read operations are executed and there is a description WS_i of the set of write operations of transaction T_i. This set is is bundled into a single message and broadcast to all nodes, including N. The atomic broadcast primitive will set a total order between these messages, deciding the order of conflicting transactions. After delivery of WS_i (totally ordered by the communication primitive) in a node, the transaction manager checks for read/write conflicts between local transactions and WS_i, with the use of locking mechanisms. If the write set intersects with the read set of a local transaction, the reading transaction is aborted. It is necessary to give write operations priority over read operations to avoid possible deadlocks over objects, because read operations are only known locally while write operations are known globally. The execution of a transaction T_i in this protocol requires a node N to broadcast two messages: one for the write set and another with the decision to abort or commit a transaction, because only N knows about a possible abort of T_i when in conflict with the write set of another transaction.

This process of delaying writes until the time of commit is also the basis of our algorithm when committing transactions.



The lock manager of each node N coordinates the operation requests of the transactions as follows:

1. *A local transaction T_i makes a read request $r_i(X)$:* reconstruct the version of X labeled with T_j where T_j is the transaction with the highest $TS_j(EOT)$ so that $TS_j(EOT) \leq TS_i(BOT)$.
2. *Upon delivery of WS_i:* perform in an atomic step for each object X where $\exists w_i(X) \in WS_i$ a *version check*:
   i. If there is no other write lock on X and X is labeled with T_j: if $TS_j(EOT) > TS_i(BOT)$, then abort T_i. Otherwise grant the lock.
   ii. If there is a write lock on X or a write lock is waiting, and T_j will be the last transaction to modify X before T_i: if $TS_j(EOT) > TS_i(BOT)$, then abort T_i. Otherwise wait for the lock.
3. *Upon having executed all operations of T_i:* commit the transaction and release all locks.

Figure 2.2: Replication protocol guaranteeing serializability [17]

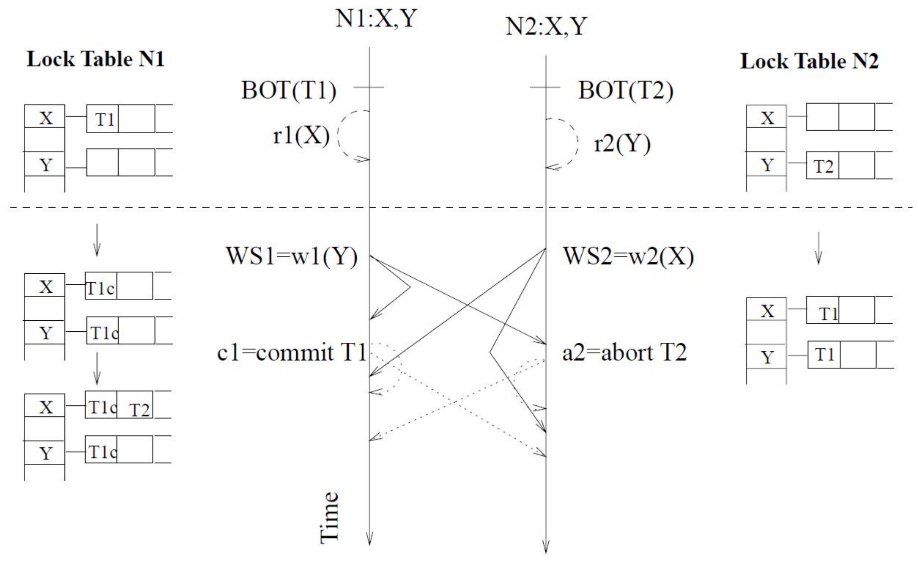


Figure 2.3: Example execution of the 1-Copy-Serializability protocol [16]

2.3.4.2 1-Copy-Snapshot-Isolation

Protocols offering serializability provide strong consistency guarantees but show poor scalability (as stated in Section 2.3.2). Solutions like [21] offer 1-Copy-Snapshot-Isolation (1CSI) to achieve better performance while still providing good consistency. With Snapshot Isolation, a transaction T reads a snapshot of the database which contains all updates that were commited at the time T started. Only conflicts between write operations are detected, and if two transactions try to update the same object, one of them will be aborted. The main advantage of SI compared to serializability is that reads never conflict with writes since they are read from a snapshot.

[21] models a system based on a middleware working on top of the database level. This middleware detects SI conflicts among transactions in different replicas. When a transaction is first executed in one replica, the write set is extracted and sent to the middleware, which then performs a validation to check write/write conflicts with transactions that executed at other replicas and that have already validated. If validation succeeds, the transaction commits at the local replica and the write set is sent to the other replicas in a lazy way. Otherwise, the transaction is aborted. To check if there was any transaction concurrent with a transaction T, the middleware needs to check if there was another transaction successfully validated after T started, but before T validated. If there is such a transaction, and there is common write between both, T is aborted.

The analysis by Serrano et. al [28] proves there is great scalability potential for 1-Copy-Snapshot-Isolation solutions by partially replicating the data. But it also brings to light some challenges of the 1CSI approach when used for partial replication. One may think that with partial replication validation of transactions should be only done by replicas that have copies of the modified data, however, that may lead to inconsistencies. For

example, assuming there are three sites $\{S_1, S_2, S_3\}$ and the database consists of objects $\{a, b, c\}$. Site S_1 stores $S_1 = \{b, c\}, S_2 = \{a, b\}$, and $S_3 = \{a, c\}$. There are two concurrent transactions t_1 and t_2, t_1 updates $\{b\}$ and t_2 updates $\{a, b\}$. Assuming the atomic broadcast totally orders t_1 before t_2, if S_1 only receives write sets related to b and c, S_1 would commit t_1 and abort t_2. The same would happen in S_2. S_3 however, having no knowledge of transaction t_1, would commit t_2, which would lead to inconsistencies between nodes. Therefore, all sites must receive and validate update operations.
There is also the issue that if a single transaction tries to update objects that are stored in different sites, some sort of coordination mechanism has to be applied. [28] presents a protocol that implements 1CSI with partial replication and tackles these issues. The protocol involves a coordinator (the site that originally receives the request from the client) that associates a timestamp with the transaction and redirects the transaction to other sites if they have data that is modified by the transaction. If the operation is redirected to another site, that site will execute the operation and send the results back to the coordinator. When a transaction executes in different sites, they must read from the same snapshot at all times. Sites that receive a redirected transaction must also guarantee that they apply all changes already done by that transaction before it was redirected.

Our solution provides a snapshot-isolation solution similar to the ones here presented, with some improvements to circumvent the problems of applying it to partially replicated data.

2.3.4.3 State Machine Approach

[26] improves on the deferred update technique used on previous solutions by improving on its main drawback: the lack of synchronization during a transaction which may lead to large transaction abort rates. It treats the distributed database as if it was a state machine where operations may be reordered. It reduces transaction abort rates by using a reordering certification test, which looks for possible serializable executions between the concurrent committing transactions before deciding on aborting them. An atomic broadcast protocol is still needed to guarantee total order in the delivery of messages so every replica reaches the same output. Unlike the previously presented protocols, however, both the writeset and the readset of transactions have to be delivered.

To guarantee all database sites eventually reach the same state, transaction execution is handled by the *Transaction Manager*, the *Lock Manager*, the *Data Manager* and the *Certifier*. The Certifier is responsible for the certification test of a transaction delivered by the atomic broadcast protocol. On certifying a transaction, the Certifier asks the Data Manager about previously committed transactions. If the transaction is successfully certified, its write lock requests are transmitted to the Lock Manager, and once granted, the updates are performed. The Certifier has to ensure that write-conflicting transactions get their locks following the order they are delivered, so that all databases apply conflicting transactions in the same order.

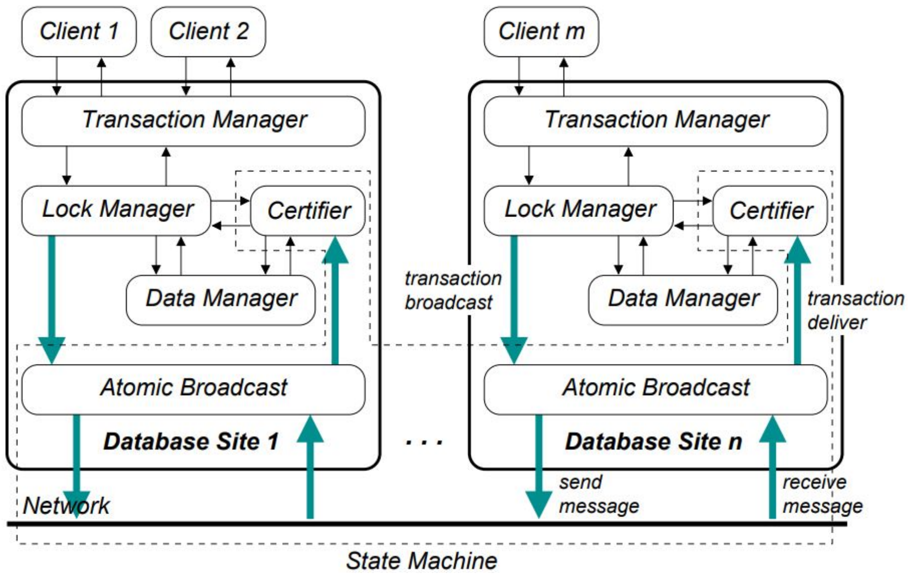


Figure 2.4: The Database State Machine [26]

The main mechanism of the certification test is the *Reorder List*. The Reorder List contains transactions that have been committed and have their locks granted, but have not yet been applied to the database. As such, they cannot yet be seen by other executing transactions. Transactions in the Reorder List may change their relative order if there is a serializable execution between concurrent transactions, if there is no possible serialization order, then some transaction has to be aborted. This reordering allows the system to apply more transactions without having to abort conflicting ones.

The size of the Reorder List is defined by the *Reorder Factor*. When the list is full (number of transactions the reached Reorder Factor), the leftmost transactions in the list is removed, its operations are applied to the database and its write locks are released. This technique reduces aborts but introduces some data contention because transactions in the Reorder List have to wait longer for their operations to be applied.

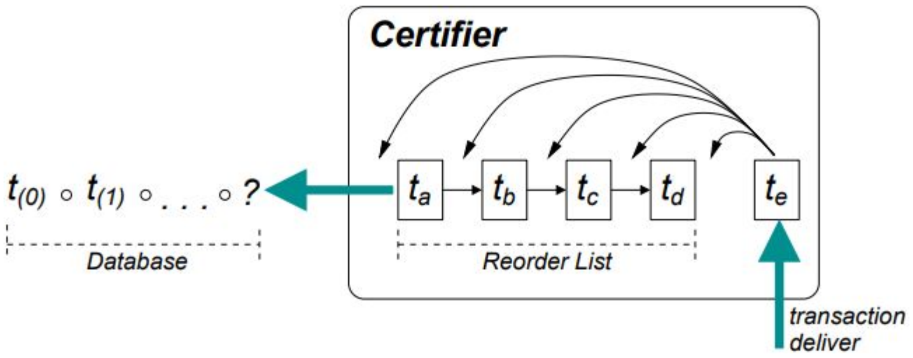


Figure 2.5: The Reorder List [26]

[29] proposes to extend the Database State Machine protocol to handle partial replication. The original DBSM protocol only assumes databases that offer full replication

to ensure that, after certifying a transaction, all database sites reach the same decision. When partially replicating objects this is impossible, as some replicas may decide to commit a transaction and others to abort it (example similar to the one in Section 2.3.4.2). Databases that hold only a partial copy of the data cannot decide to commit a transaction based only on the certification test, they must consider data objects stored in other database sites to reach a conclusion. This is possible by use of an atomic commit protocol [29], and each database should use the result of the certification test as its vote for the atomic commit protocol.

The certification of a transaction now has to involve:

- **Certification Test**: Similar test to the one used in the full replication protocol. But now, instead of committing or aborting a transaction, when a transaction is serializable the database site votes yes, otherwise votes no.

- **Atomic Commit**: Every site involved in a transaction's commit starts an atomic commit using as its vote the outcome of the certification test. If the result of the atomic commit is to commit, then the transaction is committed, its updates are performed and its locks are released.

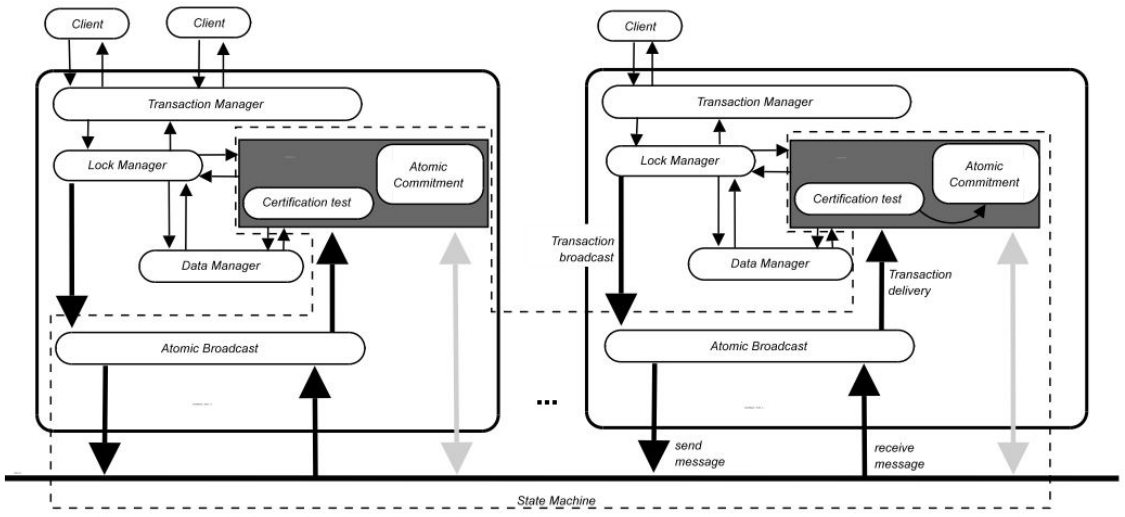


Figure 2.6: The architecture for partial replication [29]

2.3.4.4 Spanner

Spanner [7] was developed by Google and provides a scalable, globally-distributed database that provides ACID transactional guarantees. The database shards the data across many sets of Paxos state machines in datacenters spread across the world. Spanner allows replication configurations for data to be dynamically controlled by applications, and to dynamically and transparently move data between datacenters. It also provides globally-consistent reads across the database at a time-stamp by assigning globally-meaningful commit timestamps, even though transactions may be distributed. These timestamps are possible through the use of the TrueTime API, that accesses external machines to get a precise timestamp (more details in the original article).

In Spanner, transactions are replicated across datacenters through Two-Phase Commit (2PC) to guarantee serializable executions, on top of a Paxos replicated log. Basically, the decided operations during the execution of 2PC are broadcast between the datacenters as instances of the Paxos protocol. Figure 2.7 shows the messages exchanged during Two-Phase Commit on a system where logs are replicated across datacenters using Paxos.

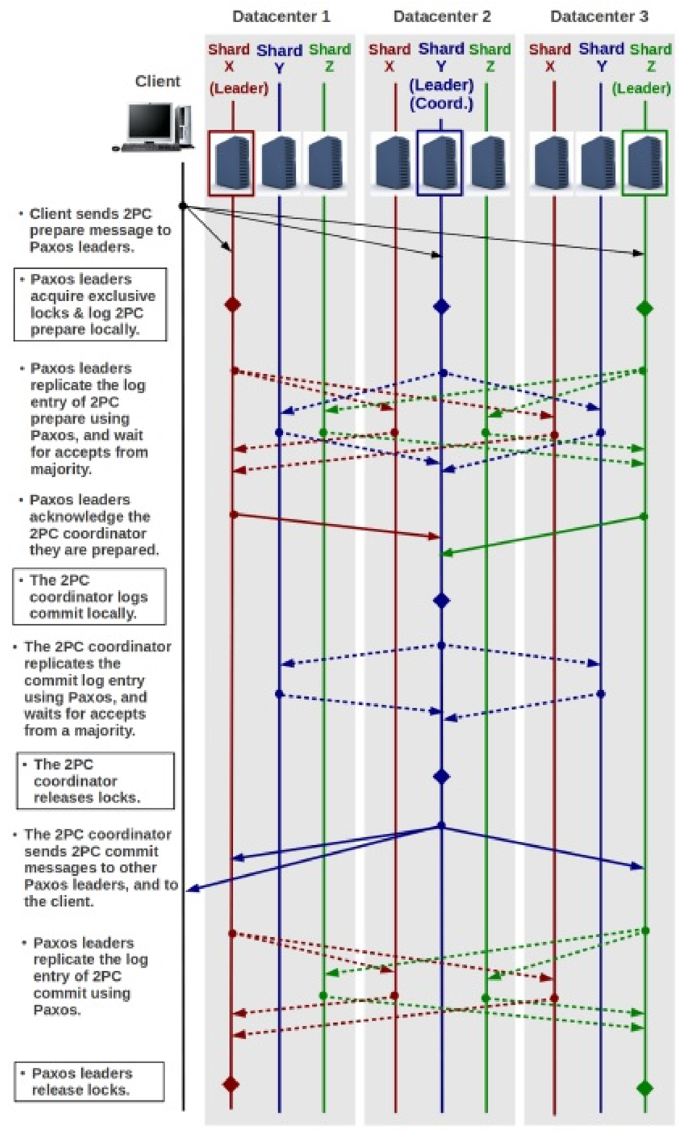

Datacenter 1
Datacenter 2
Datacenter 3
Shard X
Shard Y
Shard Z
(Leader)
(Leader) (Coord.)
Client
• Client sends 2PC prepare message to Paxos leaders.
• Paxos leaders acquire exclusive locks & log 2PC prepare locally.
• Paxos leaders replicate the log entry of 2PC prepare using Paxos, and wait for accepts from majority.
• Paxos leaders acknowledge the 2PC coordinator they are prepared.
• The 2PC coordinator logs commit locally.
• The 2PC coordinator replicates the commit log entry using Paxos, and waits for accepts from a majority.
• The 2PC coordinator releases locks.
• The 2PC coordinator sends 2PC commit messages to other Paxos leaders, and to the client.
• Paxos leaders replicate the log entry of 2PC commit using Paxos.
• Paxos leaders release locks.

Figure 2.7: Two-Phase Commit on top of a Paxos-replicated log [22]

2.3.4.5 Replicated Commit

The analysis by Mahmoud et. al [22] brings to light the high amount of inter-datacenter communication trips that have to be done when replicating the 2PC log, which hurts latency and scalability. As an optimization, [22] proposes to do the inverse, run Paxos on top of Two-Phase Commit to replicate the commit operation itself. That is, to execute the Two-Phase Commit operation multiple times, once per datacenter, with each datacenter only using Two-Phase Commit internally, and only using Paxos to reach consensus among datacenters about the fate of a transaction for the commit or abort decision. This approach is called Replicated Commit, as opposed to the Replicated Log approach. Figure 2.8 shows the messages exchanged during Replicated Commit execution.

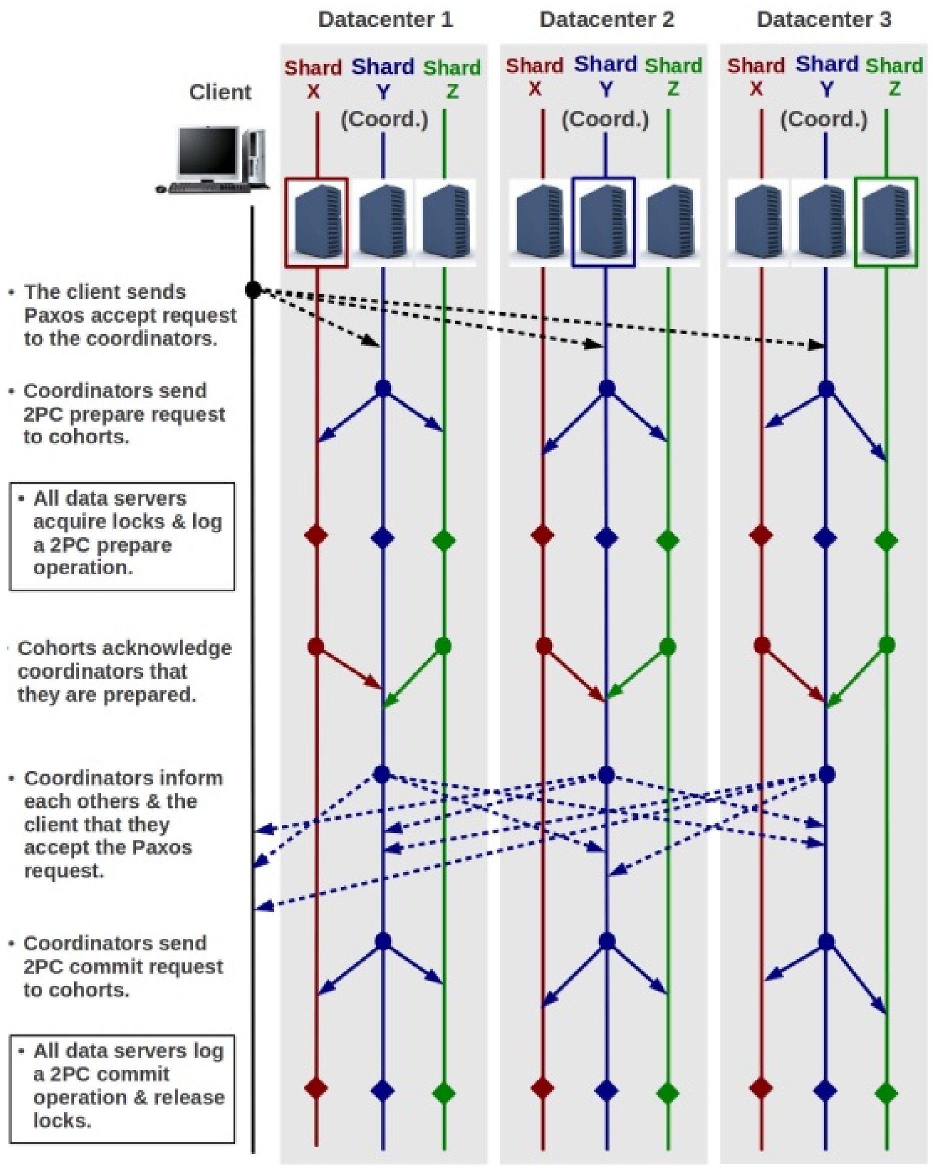

Datacenter 1
Datacenter 2
Datacenter 3
Client
Shard X
Shard Y (Coord.)
Shard Z
• The client sends Paxos accept request to the coordinators.
• Coordinators send 2PC prepare request to cohorts.
• All data servers acquire locks & log a 2PC prepare operation.
Cohorts acknowledge coordinators that they are prepared.
• Coordinators inform each others & the client that they accept the Paxos request.
• Coordinators send 2PC commit request to cohorts.
• All data servers log a 2PC commit operation & release locks.

Figure 2.8: Two-Phase Commit operations when using Replicated Commit [22]

The way Replicated Commit performs Two-Phase Commit is that each transaction executes a new Paxos instance to replicate Two-Phase Commit across datacenters. As can be seen by the Figure 2.8, the client acts as the proposer of the Paxos instance, and each datacenter as both an acceptor and a learner. The value to be agreed at the end of a Paxos instance is whether to commit a transaction or not. The default value is not to commit, so a majority of acceptors must accept the accept request from the client for the transaction to be committed.

Replicated Commit has the advantage of replacing several inter-datacenter communication trips with intra-datacenter communication, while still preserving ACID transactions on top of globally-replicated data. Also, by replicating the Two-Phase Commit operations rather than replicating log entries, communication trips are further reduced by eliminating the need for an election phase in Paxos.

2.3.4.6 Clock-SI

The Clock-SI protocol [10] is a fully distributed protocol that implements snapshot isolation for partitioned data stores. In Clock-SI transactions obtain their snapshot by reading the clock at the originating partition. This snapshot remains consistent across all partitions. By not depending on a centralized timestamp authority, Clock-SI provides better availability and performance. Commits are propagated through the Two-Phase Commit (2PC) protocol, with the originating partition acting as the coordinator. All partitions that were modified by a transaction must participate in 2PC to decide whether it should abort or commit, and what should be its final commit timestamp.

The main challenge of using of loosely synchronized clocks in different machines to coordinate transactions comes from the fact that machines may have their local clocks in different timestamps. To provide SI consistency, a snapshot with timestamp *t* must include, for each data item, the version written by the transaction with the greatest timestamp smaller than *t*. As such, the following situations must be addressed:

Situation one

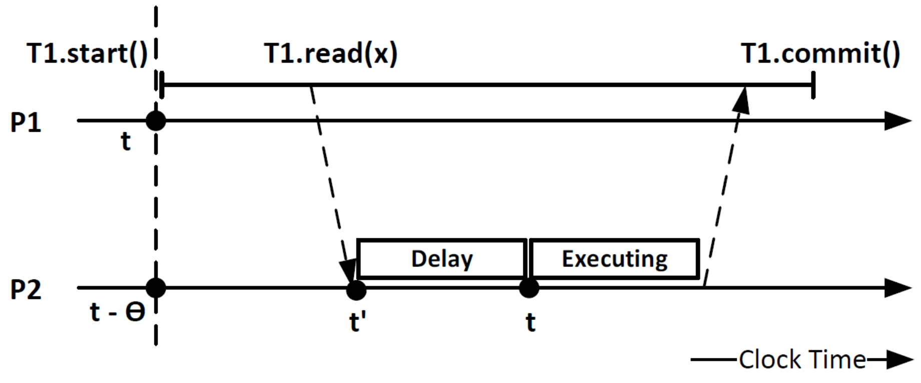


Figure 2.9: Snapshot unavailability due to clock skew [10]

Because of clock skew between different machines it is possible for a snapshot to be unavailable. In this example, transaction T_1 starts at partition P_1 and is assigned a snapshot with timestamp t. Because the clock at P_2 is behind by some amount θ, at time t on P_1, P_2's clock value is $t - \theta$. As such, when the read arrives P_2 at time t' ($t' < t$) the snapshot with time t is not yet available. This difference in clock times is important, because in P_2 between t' and t another transaction could have committed and its writes should be included in T_1's snapshot.

Situation two

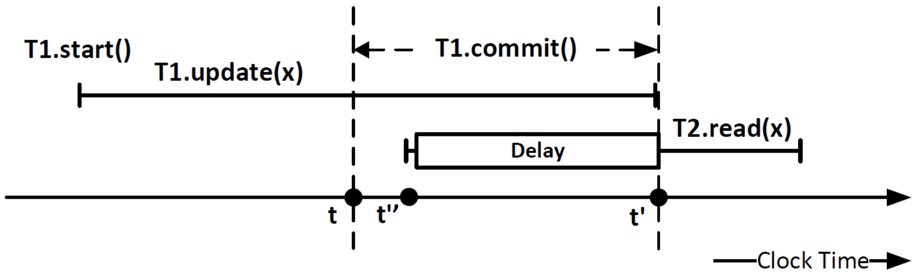


Figure 2.10: Snapshot unavailability due to pending commit of an update transaction [10]

In this case a pending commit of an update transaction causes a transaction to be unavailable. Figure 2.10 depicts two transactions running in the same partition. T_2's snapshot is unavailable because of the pending commit of transaction T_1, which assigned the value t as its commit timestamp. T_1 involves a write to stable storage and completes at time t'. If transaction T_2 starts at a timestamp t'' between t and t' and attempts to read an item x also modified by T_1, we cannot return the value written by T_1, because we do not yet know if the commit will succeed, and we cannot just return the previous value because if T_1 does commit, this older value will not be part of a consistent snapshot at t''

To solve situation one (fig. 2.9), when a transaction tries to read a data item on a remote partition and its snapshot timestamp is greater than the clock time at the remote partition, Clock-SI delays the transaction until the clock at the remote partition catches up with the snapshot timestamp, so it does not miss any committed changes.

In the second case (fig. 2.10), when a transaction T tries to access an item updated by a transaction T' that has not yet finished committing, Clock-SI delays transaction T until T' finishes its commit to ensure T's snapshot includes all committed updates.

Clock-SI also delays update requests when the snapshot timestamp is greater than the clock at some remote partition, the same way as reads, to ensure that the commit timestamp of a transaction is always greater than its snapshot timestamp.

To commit a multi-partition transaction, the 2PC protocol is performed with the coordinator running in the originating partition. Each partition performs locally a certification test to check if a transaction can commit, and if so, changes transaction state to *prepared* and sends its current clock value as prepare timestamp back to the coordinator. If each participant is able to commit the transaction, the coordinator informs all participants that they should commit the transaction (and store its changes) and with commit timestamp equal to the maximum prepare timestamp received from all participants.

2.4 Summary

In this chapter we have presented concepts that are important when discussing distributed systems. We have also presented previously done research that will be relevant for the work conducted in this thesis. Of special importance was ChainPaxos (Section 2.2), as we intend to make use of its chain topology for high throughput and to move data between nodes; Spanner (Section 2.3.4.4) for geo-replication and the possible advantages of integrating it with replicated commit; and the Database State Machine (Section 2.3.4.3) protocol as it may help us bypass the lack of partial replication support from the replicated commit approach. We also present Clock-SI, which is a protocol that coordinates multi-partition transactions without the need for a centralized timestamp authority.

3

Design and Implementation

In this chapter we will present the proposed system and algorithm for providing strong consistency in a partially replicated (or sharded) database. First, we will describe and illustrate the overall architecture of our solution, and present an overview on how the different components work with each other for processing client requests. After that, we will go more in-depth at the actual code of our algorithm, as we explain the code of our server and client API components with pseudo-code examples.

3.1 Architecture

Figure 3.1 presents the architecture of the system, illustrated with a system deployment composed by 3 server nodes.

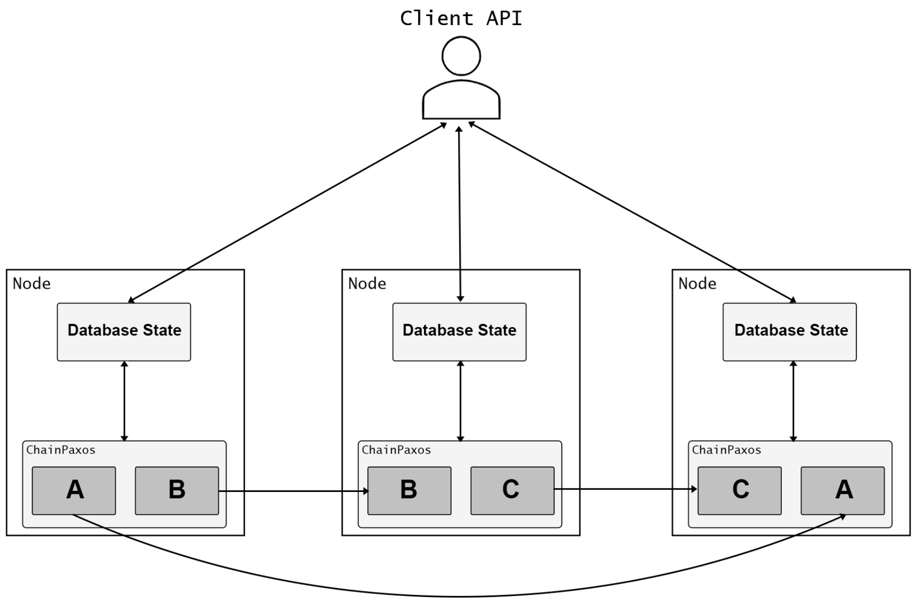


Figure 3.1: Architecture

The system organizes the database in a set of partitions, with each partition keeping a fraction of the full database state - the number of partitions in a given deployment is a parameter of the system. In the example of the figure there are three partitions, with each partition being replicated in two servers - the left node stores partitions A and B, the middle one stores B and C, and the right one stores C and A. In real deployments, each partition is typically replicated in at least 3 servers, but we are keeping them short here so the illustration doesn't get too large. Here the order of the nodes in each ChainPaxos thread is represented by arrows. This is just an example, as any order of head, tail and middle nodes in a chain could be possible. It all depends on how the replicas get configured during initialization.

Each server node maintains the database state of the partitions replicated in that node. As detailed later, for each key, the system maintains multiple data versions, thus adopting a multi-version database model. This is used to provide snapshot isolation model without resorting to locking.

For replicating each partition, each server node uses the ChainPaxos algorithm, with one instance of ChainPaxos running in each node for each partition replicated in that node. In the figure, each ChainPaxos chain is represented using arrows, with the head being the Paxos leader issuing proposals for replicating transactions.

Each node receives operations from the client - reads and commits - and processes them by accessing the local state and issuing operations in ChainPaxos instances. After processing an operation, the result is returned to the client.

This architecture is inspired by the one of Spanner (see 2.3.4.4), that has a datacenter with nodes for each partition that propagate data to the nodes in the other datacenters through Paxos.

Clients access and modify the database using the client API, which is responsible for transaction operations like reads and writes, and, when a user requests a commit, coordinates the nodes during the execution of the Two-Phase commit protocol. We now briefly describe the operations offered by the client API:

Begin

Sets the local state values required by the API to start a new transaction.

Read

The user requests to read a value with a given key from the database. The API will contact one of the servers that stores the partition with that key (should be the one closest to the user) and returns the value back to the client. Reads follow the same protocol as in Clock-SI (see 2.3.4.6), which means that if a user asks for a read with a snapshot timestamp higher than the current clock in that partition the read has to wait until the partition's clock catches up. If the read attempts to read a value that is also modified by a transaction that is prepared but has not yet finished committing, the read must wait for the transaction's conclusion.

Write

The user requests to write a key with a value. Writes are stored locally in the client API to be sent later when the user asks to commit the transaction (see *deferred update* in 2.3.4)

Commit

The user asks to commit the transaction. A commit is processed running the two-phase commit protocol among the partitions involved in the transaction. The client requests every partition that was modified by this transaction to prepare the transaction, by sending the transaction's write-set and snapshot timestamp. In this example, a transaction that modifies partitions A and B could send the prepare requests to the the left and middle node, or it could also send both requests to the left node since it contains both A and B partitions. Each modified partition will then do a certification check to see if it is possible to commit this transaction and replies back to the API with the result, it also sends back its current clock to be used as the commit timestamp of the transaction. The certification test checks if there is any committed transaction that happened concurrently to this one, and modified the same data (similar to protocol in 2.3.4.1, but only checks for write/write conflicts). If the certification test is successful in every modified partition, the API tells every modified partition to finalize the commit and write the updated values in storage with commit timestamp equal to the largest timestamp received by the API from all partitions. If the certification test fails in any partition, the API tells evey partition to discard the transaction.

As a final note, it is important mentioning that our algorithm is based on the Clock-SI protocol when assigning timestamps to transactions but some adjustments had to be made. The original Clock-SI paper [10] only studies the case where each partition is only stored in a single node, and so each partition can just check its own clock to get its own timestamp. In our case, because each partition can be located in multiple nodes it is not possible to just check the local clock of one of the nodes because the clocks in all nodes may be running at different speeds. As such, we had to find a way to make sure that clock reads of a partition stayed consistent across all nodes in that partition. Our solution was to use the messages propagated by the ChainPaxos protocol to piggyback information about the current timestamp across all nodes in a partition. For this, we introduced a virtual clock in each replica. When the Paxos leader (head of the chain) sends a new accept message to the other nodes, it also send its current clock value so that when the other nodes receive this message they all set their virtual clock values to the same one. This means that advancing the (virtual) clock in a partition is dependent on the number of messages sent by the ChainPaxos threads, because of this, the leader may occasionally send a NoOp message to the other nodes in the chain so their clocks don't fall behind too much. A more detailed explanation of this virtual clock in section 3.2.2;

3.1.1 Server

We now present the pseudo-code of the server side of our algorithm. The following code implements the protocol that modifies database state in our nodes (code for the ChainPaxos component can be found in the original article [27]). Algorithm 1 presents the state of each replica: *store* is a map that maps each partition in this node to their respective key-value store, each key-value store itself is a map that, for each key stored in a partition, maintains multiple data versions with their respective timestamps; *prepared* is a map of all prepared transactions that passed the certification test but are still waiting for the result of the Two-Phase Commit protocol to see if they should commit or not. Transactions are here represented as a triplet with an *id*, a writeset *ws*, and a a *prepareTimestamp*.

Algorithm 2 presents our server algorithm, with auxiliary functions detailed in Algorithm 3. Of note that, except for the *Read_Request* message, all received messages by the server have to first be accepted and ordered by the ChainPaxos layer, to ensure all nodes in a partition converge to the same state.

Algorithm 1 Server state

```
1: store : key-value store (for multiple partitions)
2: prepared : map of prepared transactions waiting for commit in each partition
```

Algorithm 2 Server algorithm

```
1: function INIT
2:     store ← {}
3:     prepared ← {}
4: upon receive <READ_REQUEST, key, snapshotTs, partition> from clientapi do:
5:     currentTs = GETCLOCKTIME(partition)
6:     if snapshotTs = 0 then
7:         snapshotTs ← currentTs
8:     if snapshotTs > currentTs then
9:         Wait for this partition to reach snapshot time
10:    for each (id,ws,prepareTs) in prepared[partition] do:
11:        if key ∈ ws.keys ∧ prepareTs <= snapshotTs then
12:            Wait for transaction with id to reach commit/abort
13:    end for
14:    val ← {val : (key,val,commitTs) ∈ store[partition] ∧ commitTs <= snapshotTs ∧ ∃!((key',commitTs') :
           (key',val',commitTs') ∈ store[partition] ∧ key ! = key' ∧ commitTs < commitTs') }
15:    SEND(clientapi,<READ_REPLY, val, snapshotTs>)
16:
```

```
17: upon receive <PREPARE_REQUEST, $id, ws, snapshotTs, partition$> from clientapi do:
18:     prepareTs = GetClockTime(partition)
19:     if snapshotTs = 0 then
20:         SnapshotTs ← prepareTs
21:     if snapshotTs > prepareTs then
22:         Wait for this partition to reach snapshot time
23:     if ∃ ((id',ws',prepareTs') ∈ prepared[partition] ∧ (ws.keys ∩ ws'.keys ! = ∅ ) ∧     prepareTs' >=
    snapshotTs ∧ prepareTs' <= prepareTime) then
24:         Send(clientapi,<PREPARE_REPLY, $false$, 0>)
25:     else
26:         prepared[partition] ← prepared[partition] ∪ {(id,ws,prepareTs)}
27:         Send(clientapi,<PREPARE_REPLY, $true, prepareTs$>)
28:
29: upon receive <COMMIT, $id, commitTs, partition$> from clientapi do:
30:     currentTs = GetClockTime(partition)
31:     if commitTs > currentTs then
32:         Wait for this partition to reach commit time
33:     transaction ← {(id',ws,prepareTs) ∈ prepared[partition] : id = id' }
34:     for each (key, val) in transaction.ws do:
35:         store ← store ∪ {(key, val, commitTs)}
36:     end for
37:     prepared[partition] ← prepared / {transaction}
38:
39: upon receive <ABORT, $id, partition$> from clientapi do:
40:     transaction ← {(id',ws,prepareTs) ∈ prepared[partition] : id = id' }
41:     prepared[partition] ← prepared / transaction
42:
```

Algorithm 3 Server Algorithm - private functions

```
1: function GetClockTime(String partition)
2:     return current timestamp of partition
```

On initialization, the maps and lists start empty (L1-3). When a read request is received (L4-16), the system first compares the snapshot timestamp of the read to the partition's current clock. If the snapshot timestamp is equal to 0, that means this is the first read done by this transaction, so the system sets the snapshot as the current timestamp of this partition (L6-7). If the snapshot timestamp has a value larger than the current clock, then the read should be postponed until the partition's clock has catched up to the snapshot timestamp (L8-9). Then, the system checks if there are any transactions currently pending commit that also modify the value of the key that was requested, if that is the case, the read waits until all these transactions have finished committing (L10-13). Finally, the system retrieves from the database the value for this key with the maximum commit timestamp lower than the snapshot timestamp and sends it back to the user (L14-16).

When a prepare request for a transaction T_i is received (L17-28), the system checks the current clock to be used as the prepare timestamp (L18). Like reads, if the transaction still has not set a snapshot then it will be set to this timestamp, and if the snapshot timestamp is larger than the current clock, then the partition must wait for the clock to catch up (L19-22). The system then checks if there is any pending transaction T_j with conflicting writes which entered its prepare state between the moment of T_i's snapshot and the current timestamp (L23). If that is the case, the system requests the client API coordinating Two-Phase Commit to abort T_i (L24). If there were no conflicts, T_i is now considered prepared and the system requests the coordinator to commit transaction (L25-27).

When a commit request for a transaction T_i is received (L29-37), the system starts by ensuring the partition's clock is up-to-date with the commit timestamp. It then stores the changes done by T_i in its local store (it does not overwrite old values, it just adds new ones with a larger commit timestamp). T_i is then removed from the list of pending transactions.

If an abort request is received for some transaction (L39-41), then the transaction is simply discarded from the prepared transaction list.

3.1.2 Client

The following is the pseudo-code of our client API. Algorithm 4 presents the state of the client: $servers$ is the list of the addresses of all nodes in the system; $partitions$ informs the client of the available partitions in the database; $transactionID$ is the unique id of the current transaction being processed by the API; $snapshotTs$ and $prepareTs$ are the snapshot and prepare timestamps of the current transaction; WS is the map storing the delayed writes done by this transaction, that will be sent to the servers when the user requests a commit; $partitionsWithWrites$ is a set informing of what partitions will be modified by this transaction, if it commits; $waitingResponse$ is the set of nodes for which the API is waiting for a response to the commit request, during Two-Phase Commit; $responses$ is a counter of the amount of nodes in $waitingResponse$ that have already answered the commit request; $commitTs$ is the final commit timestamp of this transaction, if 2PC is successful; $commit$ is just a bool that is true if the nodes in 2PC agree to commit the transaction.

Algorithm 5 is the algorithm behind our client API, with auxiliary functions detailed in Algorithm 6.

Algorithm 4 Client API state

```
 1: servers : array of server nodes
 2: partitions : array of existing partitions
 3: transactionId : id of current transaction
 4: snapshotTs : snapshot time of transaction
 5: prepareTs :prepare time of transaction
 6: WS : writeset of transaction
 7: partitionsWithWrites : set of partitions to which this transaction writes
 8: waitingResponse : set of nodes pending response to prepare
 9: commit : bool decision to commit or abort
10: responses : int number of responses received from nodes in waitingResponse
11: commitTs : commit time of transaction
```

Algorithm 5 Client API Algorithm

```
upon receive <BEGIN> from client do:
    transactionId = UniqueID()
    snapshotTs ← 0
    prepareTs ← 0
    WS ← {}
    partitionsWithWrites ← {}
    waitingResponses ← {}
    commit ← true
    responses ← 0
    commitTs ← 0

upon receive <READ, key> from client do:
    partition ← GetPartitionWithKey(key)
    node ← GetNodeInPartition(partition)
    Send(node,<READ_REQUEST, key, snapshotTs, partition>)

upon receive <READ_REPLY, val, readSnapshotTime> from node do:
    if snapshotTs = 0 then
        snapshotTs ← readSnapshotTime
    Send(client,<READ_RESULT, val>)

upon receive <WRITE, key, val> from client do:
    partition ← GetPartitionWithKey(key)
    partitionsWithWrites ← partitionsWithWrites ∪ {partition}
    WS ← WS ∪ {partition, key, val}

upon receive <COMMIT> from client do:
    for each partition in partitionsWithWrites do:
        node ← GetNodeInPartition(partition)
        waitingResponse ← waitingResponse ∪ {node}
        writes ← {(key, val) : (partition, key, val) ∈ WS}
        Send(node,<PREPARE_REQUEST, transactionId, writes, snapshotTs, partition>)
    end for

upon receive <PREPARE_REPLY, partitionCommit, partitionCommitTime> from node do:
    if node ∈ waitingResponse then
        responses ← responses +1
        if partitionCommitTime > CommitTime then
            CommitTime ← partitionCommitTime
        if partitionCommit = false then
            commit ← false
    if responses = #waitingResponse then
        if commit = true then
            for each node in waitingResponse do:
                Send(node,<COMMIT, transactionId, CommitTime, partition>)
            end for
        else
            for each node in waitingResponse do:
                Send(node,<ABORT, transactionId, partition>)
            end for
        Send(client,<COMMIT_RESULT, commit>)

```

Algorithm 6 Client API Algorithm - private functions

```
1: function GetPartitionWithKey(String key)
2:     return partition from partitions that contains key

3: function GetNodeInPartition(String partition)
4:     return node from servers that participates in partition        ▷ should be node closer to user
```

When a user starts a new transaction ($L1$-10), the API sets the default values to prepare for a new transaction.

When the user asks to read a value ($L12$-15), the system sends a read request to one of the nodes that stores the partition which contains the key. When the node responds ($L17$-20), if the transaction still had no snapshot timestamp set it will be set to the timestamp received from the node and then the value that was read is returned to the user.

When a user requests to write a value ($L22$-25), the system stores the information of what partition will be modified by this write and then stores the write locally to be sent later.

When the user requests to commit the transaction ($L27$-33), the system chooses one node of each partition and sends to each one the respective updates, this starts the 2PC protocol to find out whether the transaction can committed or not. The system then waits for the response of every partition that was modified ($L35$-51), whenever a response is received it sets the commit time of the transaction to the highest timestamp received ($L38$-39). After receiving a response from every partition, if they all accepted to commit the transaction, the system now tells every partition to finalize committing the transaction and sends the final commit timestamp ($L42$-46). If any of them called to abort the transaction, the system will tell every partition to discard the transaction ($L48$-50). After all of this, the API replies to the user with the final commit result.

3.1.3 Correctness

In this section, we present the correctness argument for our algorithm, by showing that its execution ensures the snapshot-isolation safety properties.

1. **Transactions commit in a total order**. Every operation is totally ordered in every replica by the ChainPaxos layer, and because every commit timestamp of transactions is assigned from reading values of physical clocks.

2. **Transactions read consistent snapshots**. By delaying reads until a partition's clock catches up to the snapshot timestamp of a transaction, we guarantee that a transaction reads all committed changes of transactions with commit timestamps smaller than its snapshot timestamp. By delaying reads when there are prepared (but no committed) transactions with overlapping writes, a transaction never reads a value from a transaction that was aborted.

3. **Committed concurrent transactions do not have write-write conflicts**. The algorithm identifies concurrent transactions by checking their snapshot and commit timestamps and aborts one of the two concurrent transactions if their write-sets overlap. Two nodes never reach different conclusions on whether to commit or abort a transactions because they execute the Two-Phase Commit protocol.

3.1.4 Faults

ChainPaxos already handles faults and membership reconfiguration. When a replica fails, the ChainPaxos chains the replica was a member of will just reconfigure and go back to normal routine. If the replica rejoins the chains later, ChainPaxos transfers the current state from one of the other replicas to the one that just joined, so it has the same state.

During the Two-Phase Commit protocol, replicas and the client log to stable storage any decisions that they make. If a replica crashes and then recovers, it checks its log to know what was its decision, and continues the 2PC protocol from where it stopped. If the client times out (possibly because it crashed), the partitions communicate with each other to reach a decision on whether to commit or abort the transaction (this would require each partition to know what partitions where involved in the transaction, this information could be sent in the prepare message). As of writing this paper, however, we have not yet implemented a working version of the recovery mechanism for two-phase commit.

3.2 Implementation

Our prototype was implemented in Java. Most of the pseudo-code here presented can be easily translated to Java code. Most of the logic is simple to translate and simple lists and maps are used to store most of the data. Still, there are a few details we feel we should elaborate.

3.2.1 Key-Value store

In section 3.1.1 we say that the each partition has its own key-value store that "is a map that, for each key stored in a partition, maintains multiple data versions with their respective timestamps". Basically, it is the shard of the database that is stored in a specific partition. In our implementation, the key-value store is a map with a string - the name of a value - as key and the "value" itself is a TreeMap (implemented by Java). In this TreeMap each key is a Long value - a timestamp - and the value is a byte array. This map is the part responsible for storing the multiple data versions of keys that allow snapshot reads. By using a TreeMap - which is a sorted map - we can store each value and its corresponding timestamp, and easily query the database for the value with the largest timestamp up to a given snapshot timestamp.

3.2.2 Global Partition Clock

In server pseudo-code in section 3.1.1, we state that read, prepare and commit operations wait for a partition's clock to advance if it is lower than the snapshot timestamp of a transaction. To achieve this, we implemented a class that manages the current clock of a partition. One object of this class gets instatiated for each partition stored in a server. It is responsible for storing and modifying the current clock of a partition, and stores operations from transactions that are currently waiting for an advance on the partition's clock. When the clock reaches the timestamp these operations where waiting for, this class will resume and finish them. The following is the pseudo-code for this class, which we called *GlobalPartitionClock*.

Algorithm 7 GlobalPartitionClock state

```
1: timestamp : long
2: waiting : map of operations waiting for a specific timestamp (timestamp is the key, and a list of pending
   operations is the value)
```

Algorithm 8 GlobalPartitionClock

```
1: function Init( )
2:     timestamp ← 0
3:     waiting ← {}

4: function GetClock( )
5:     return timestamp

6: function SetEventForTimestamp(ts, pending_op)
7:     waiting[ts] ← waiting[timestamp] ∪ {pending_op}

8: function SetClock(new_timestamp)
9:     timestamp ← new_timestamp
10:    pending_ops ← {op : op ∈ waiting[ts] ∧ ts <= timestamp}
11:    for each op in pending_ops do:
12:        op.continue()
13:    end for
```

When initialized ($L1$-3), the current *timestamp* is 0 and the map of *waiting* transactions is empty. In our implementation, *waiting* is implemented by a TreeMap, which is a sorted map. By using timestamps as the key, and a list of operations waiting for that timestamp as a value, it is easy to query the sorted map on all operations that are waiting up to a given timestamp.

The function *GetClock* ($L4$-5) is just a simple getter used by a partition to check the current clock.

Function *SetEventForTimestamp* ($L6$-7) is called when an operation needs to be delayed. The *waiting* map adds $pending_op$ to the list of operations currently waiting for timestamp *ts* (the list starts empty).

As mentioned at the end of section 3.1, the clock of a partition advances when the leader of the corresponding ChainPaxos sends accept messages to the rest of the chain.

If these messages get accepted, each instance of the corresponding partition will call $SetClock$ ($L8$-13) which sets the clock to the timestamp that came in the accept message. Then it will check for any pending operations that were waiting for the clock to reach this timestamp or lower. If there are any, they are resumed and finished sequentially.

3.2.3 Babel

The protocols running in the server nodes, the algorithm that modifies the database and the algorithm for the ChainPaxos threads, are implemented on top of Babel[1][13]. Babel is a Java framework to support quick prototyping of distributed algorithms. Babel provides a set of abstractions that allows the developer to focus on implementing the algorithm logic, following an API that is close to the typical presentations of pseudo-code of such algorithms. Babel simplifies developing and testing algorithms for distributed systems by providing useful abstractions that deal with the low level complexities usually associated with distributed system implementations. These complexities include handling communication among local and external protocols and concurrency-control. Notably, communication complexities are hidden behind abstractions called *channels*.

A Babel process can execute any number of protocols that communicate with each other and/or protocols in other processes. Each protocol is exclusively assigned a dedicated thread which handles received messages in a serial fashion. As stated before, communication between protocols is hidden behind the *channels* abstraction, which allows protocols to easily establish and accept TCP connections to/from other processes, including processes not running Babel. In our work, these *channels* are responsible for the communication between the client and servers, and among the servers themselves when propagating messages trough a ChainPaxos chain.

Figure 3.2 exemplifies the Babel Architecture, with two Babel processes in different machines, each running three distributed algorithm protocols that communicate with each other through *channels*.

[1]a repository is available at https://github.com/pfouto/babel-core

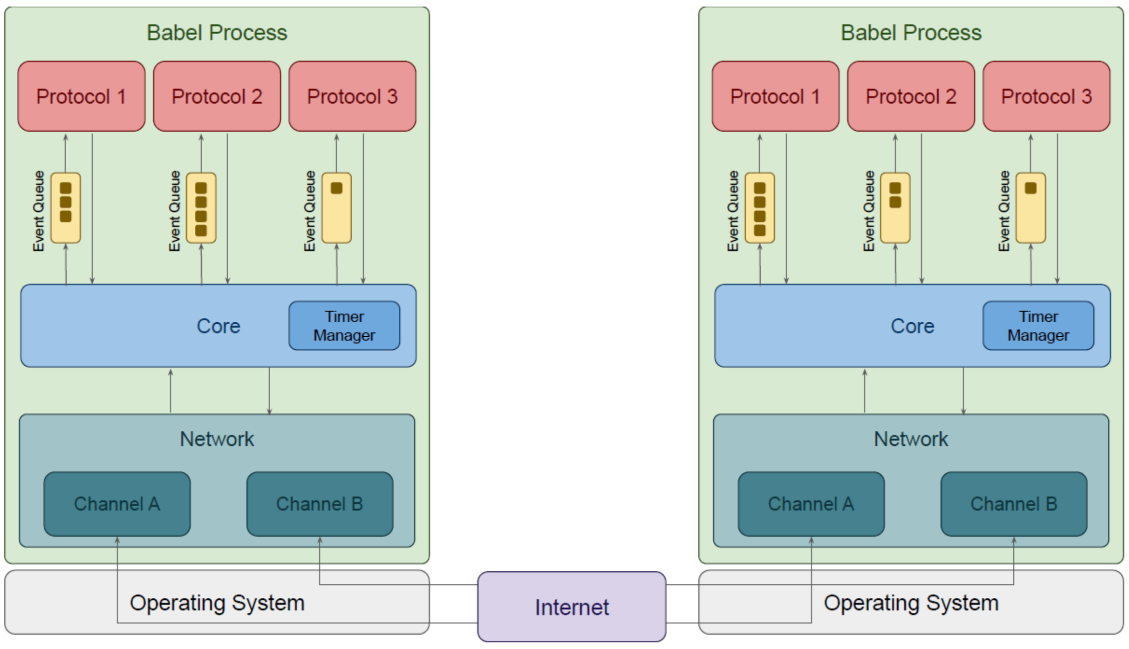


Figure 3.2: Babel Architecture [13]

4 Evaluation

In this chapter we report the experimental evaluation of our algorithm. The primary goal of this thesis is to research ways to achieve partial replication with strong consistency, and how that could improve on solutions based on full replication. As such, our tests focus on comparing our algorithm to a fully replicated one. We will evaluate both solutions by comparing execution times under different workloads, varying the number of clients and number of operations per transaction. We will also analyse how the percentage of aborted transactions changes with the number of clients.

4.1 Configuration

The experiments were conducted on the Grid5000 testbed, using a cluster of machines with one Intel Xeon Gold 5220 CPU with 18 cores and 96 GiB DDR4 RAM. The machines are connected through a 25 Gbps Ethernet switched network. Each replica executes in its own machine, and an extra 10 independent machines work as clients running the YCSB benchmark. Each client connects to a replica executing transaction operations in a closed loop. Each test was run 5 times.

The prototypes of both our server and client algorithms were implemented in Java. When testing our partial replication algorithm, each replica stores three different partitions, with partitions distributed across replicas in a round-robin way (configuration similar to the one in figure 3.1). This also means that each ChainPaxos chain will only have 3 nodes. The number of unique partitions is equal to the total number of servers, so with 5 servers there will be 5 partitions and so on. For our tests, the database stores 1000000 different keys. The following are the partition distributions used when testing partitioned databases, with 5 and 7 servers:

Total Partitions	
A,B,C,D,E	
Nodes	Partitions
1	A,B,C
2	C,D,E
3	E,A,B
4	B,C,D
5	D,E,A

Table 4.1: 5 servers

Total Partitions	
A,B,C,D,E,F,G	
Nodes	Partitions
1	A,B,C
2	C,D,E
3	E,F,G
4	G,A,B
5	B,C,D
6	D,E,F
7	F,G,A

Table 4.2: 7 servers

To use as a comparison, we also perform tests on a fully replicated database. A fully replicated protocol can be easily implemented by instantiating our algorithm with just a single partition. Therefore, during full replication tests, we setup the replicas such that there exists only a single ChainPaxos chain and every replica is part of it.

Our tests consist of running multiple clients with the YCSB benchmark[6]. These clients are set on a loop that keeps executing the required operations to create, modify, and commit a transaction (using our implemented client API). YCSB by itself does not actually have transaction support, it only offers simple operations, like reads and writes of single values, to work with single registers. As such, we actually had to extend YCSB to allow clients to perform more complex operations (like commits/aborts) to be able to test transaction support. The code looped by the clients works as follows:

1. Begin a new transaction. (performed locally)

2. Randomly select (with uniform distribution) n keys from the database to be read. (performed locally)

3. Request a read for each of the n keys, one at a time. (requires contacting the replicas)

4. Write a new value for a (variable) subset of the n keys. (performed locally, since updates are delayed)

5. Request to commit the transaction. (requires contacting the replicas and waiting for conclusion of 2PC)

In our tests, we consider that a user only writes values for keys that they have previously read. Therefore, modifying a key implies two operations, the read during step 3 of the loop and then the write during step 4. Each database value is 128 bytes.

Clients connect uniformly at random to a replica with the requested partition, and receive a reply from that same replica once the operation is over. This allows to maximize throughput by distributing the load of handling clients as much as possible.

Also of note that in every ChainPaxos chain each Paxos operation is executed in a different instance (no batching of operations).

4.2 Results

In this section we present and discuss the results of the the tests performed on our algorithm. The line graphs on the left present the performance results of our algorithm under different configurations, and the bar graphs on the right show the corresponding transaction abortion rates.

4.2.1 Partition Impact

In our first set of experiments, we study how the number of partitions in the system impact the overall performance of the system. In these tests, clients perform transactions that read 4 keys and write 50% of them (2 writes). We test how the number of partitions (and nodes) impact the overall latency of the system.

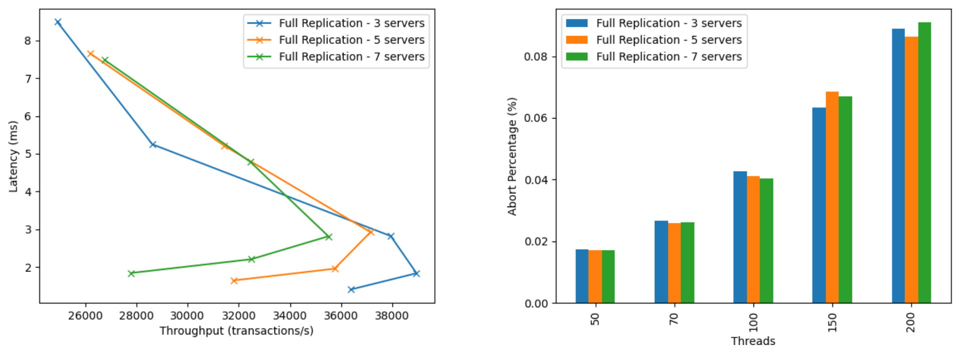


Figure 4.1: Partition impact - Full replication only

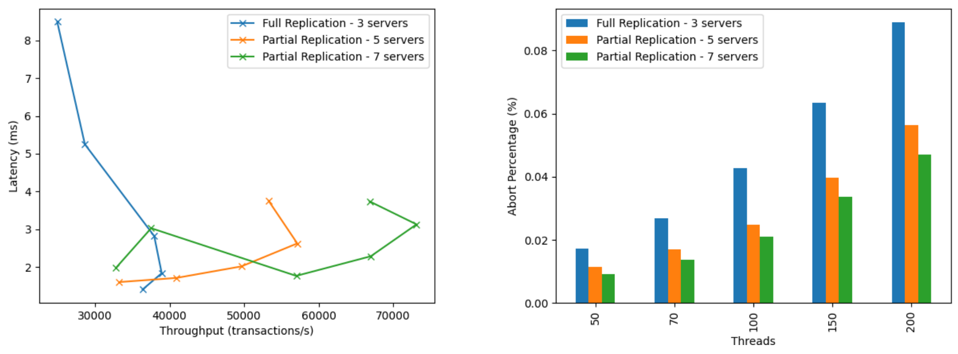


Figure 4.2: Partition impact - Partial replication only

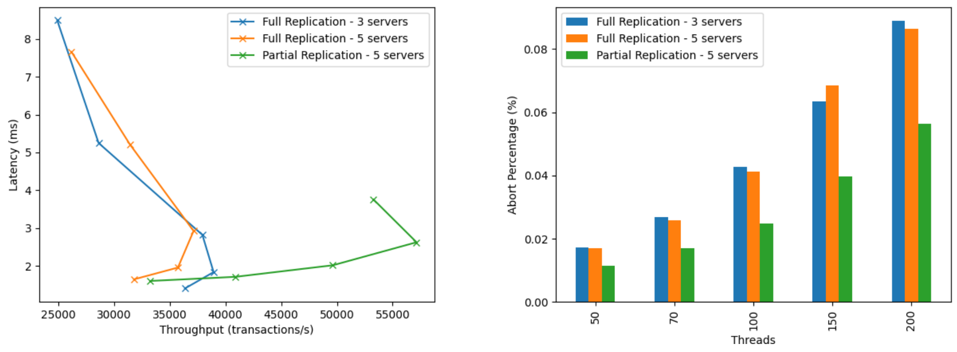


Figure 4.3: Partition impact - 5 servers, full and partial replication

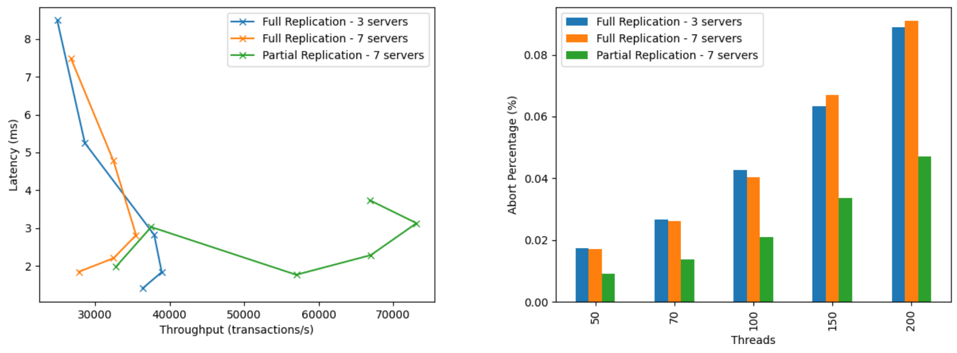


Figure 4.4: Partition impact - 7 servers, full and partial replication

We start by studying how the number of replicas impact the performance in a fully replicated deployment. Figure 4.1 shows that the performance of the system has only a very small drop in performance when increasing the number of replicas - as all replicas are involved in processing all transactions, this result is expected, with the small drop in performance being explained by the fact that ChainPaxos has some fix management overhead that depends on the number of replicas of the system.

Figure 4.2 compares the performance of a partial replicated deployment with 5 and 7 servers with the full replicated scenario with 3 servers. Figures 4.3 and 4.4 place the partial replication lines with their full replication counterparts in the same graph to make it easier for comparison.

The results show that partial replicated deployments have higher throughput than the full replicated deployments and that the performance of partial replication increases with the number of servers. This performance gain comes from the fact that, because in partial replication only a fraction of the total transactions have to be processed by a node, the CPU usage in each node is reduced. With low load (first points in the lines), however, the latency of the fully replicated deployments is lower than that of partial replicated deployments. The reason for this is that under partial replication, the commit needs to run a two-phase commit between the replica groups involved in the transaction, which is not necessary in the full replication scenario.

The bar graphs on the right show that abort rate is influenced by contention among transactions that execute concurrently. As the number of threads increase, the number of concurrent transactions also increase in each of the deployments, leading to an increasing ratio of aborts, as shown in the figure. The lower abort ratio of partial replication deployments, when compared with the full replicated deployment is, among other factors, influenced by the latency of transactions which starts growing more quickly with a low number of threads when doing full replication, which in turn leads to more transaction interference.

4.2.2 Write percentage impact

Here we tested how system responsiveness is affected by writes from a transaction. Clients perform transactions that read 4 keys and write a variable percentage of them (100% corresponds to 4 writes, 50% to 2, and 10% was rounded up to 1 write). In both tests, the system is composed of 5 servers.

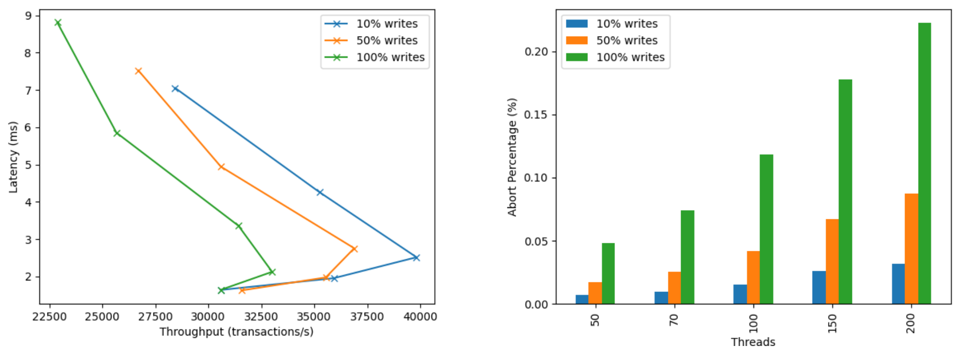


Figure 4.5: Write percentage impact - Full replication

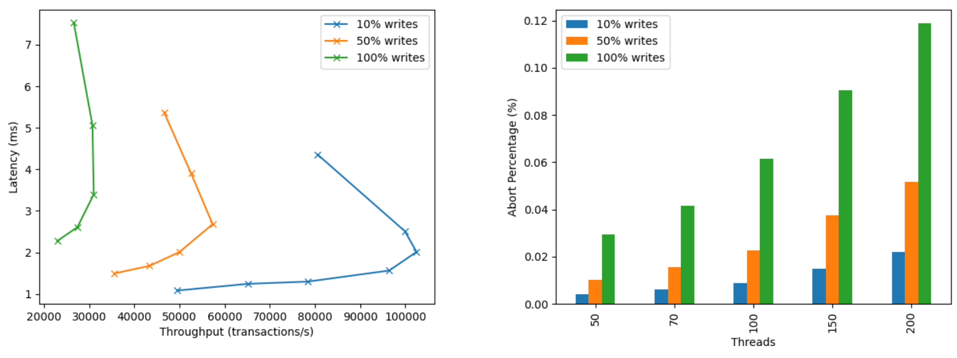


Figure 4.6: Write percentage impact - Partial replication

As expected, response times are heavily influenced by the amount of updates performed on the database. In both full and partial replication, system performance decreases as the number of writes increases. This is the result of many factors, like having to send more data over the network and the partitions having to spend more processing time to find conflicts with other transactions.

In the partitioned configuration, however, the major influence to latency times is the fact that by increasing client writes we also increase the likelihood of the transaction performing updates in multiple partitions , which means that, when a client requests a commit, much more time is spent coordinating multiple servers during the 2PC protocol. Because of this, partial replication performs worse than full replication when write percentage is very high, as can be seen when comparing figures 4.5 and 4.6. When write percentage is lower, the advantages of partial replication outweigh the resources spent during two-phase commit and allow partial replication to achieve much higher throughput thresholds when compared to full replication.

Abort percentage also proves to be inversely proportional to write percentage. Again, this is not surprising, because by increasing the amount of writes we also heavily increase the probability of a transaction conflicting with another by trying to update the same keys.

4.2.3 Key impact

In these tests we vary the number of keys that are read and set the write percentage to 50% (which means 2 keys will write 1, 4 will write 2, 12 will write 6). Again, the system is composed of 5 nodes.

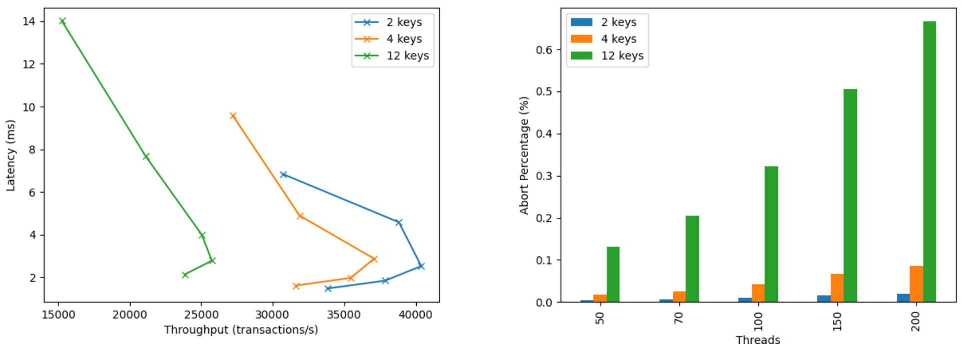


Figure 4.7: Key impact - Full replication

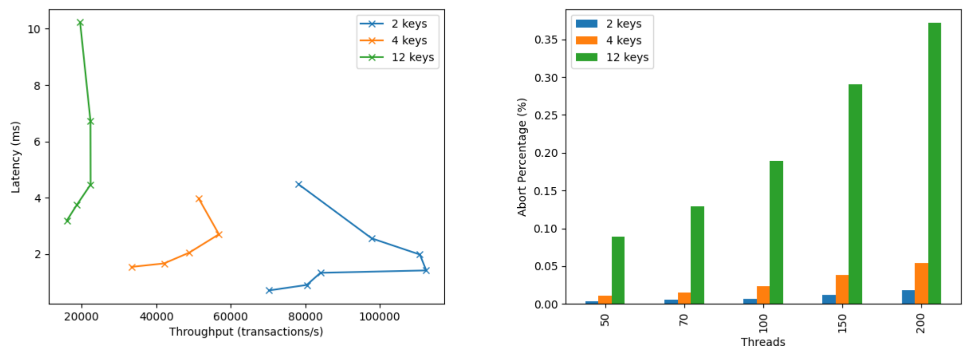


Figure 4.8: Key impact - Partial replication

These results lead to similar conclusions as the results in the variable write tests (section 4.2.2). By increasing the amount of keys (and writes), response times are expected to be worse, especially in a partitioned system where multiple partitions may have to be coordinated during 2PC. Additionally, because in these tests we also increase the amount of keys read by a transaction, extra time is also spent requesting key reads to the system.

Like in the write percentage tests, when comparing figures 4.7 and 4.8 we can see partial replication performing worse when transactions do a large amount of writes. Because in the 12-key tests each transaction performs 6 writes, that means committing transactions will often involve coordinating most, if not all, of the partitions in the system (in our tests a 5 server system stores 5 unique partitions). As the number of writes decrease, however, partial replication ends up outperforming full replication by a large margin.

4.2.4 Read-only impact

In these tests we introduce a new type of transactions, a read-only transaction that only reads keys but does not write any of them (always commits successfully). We vary the percentage of ”normal” transactions and read-only transactions. Both types of transactions read 4 keys, and normal transactions write 50% of them. The system is composed of 5 nodes.

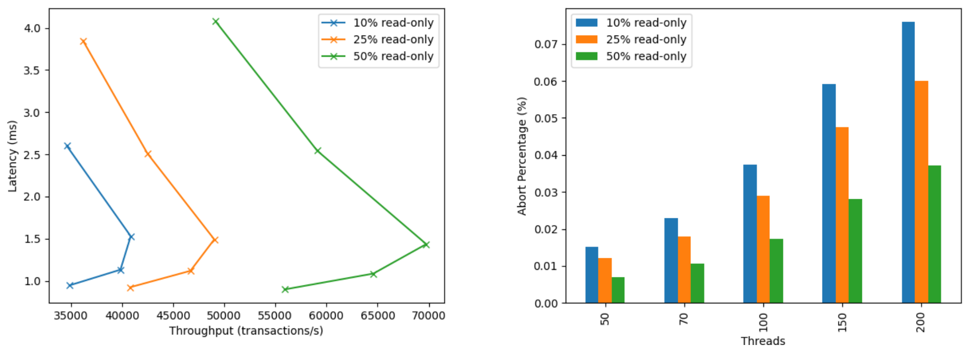


Figure 4.9: Read-only impact - Full replication

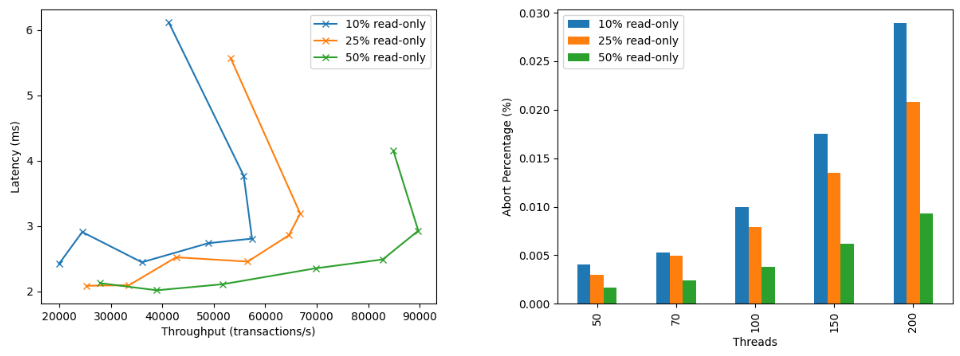


Figure 4.10: Read-only impact - Partial replication

These results show that latency times are better with the increase of read-only transactions. Because read-only transactions do not update the database in any way, no processing time has to be spent resolving conflicting transactions and coordinating partitions during commit. That means that read-only transactions are much shorter than normal transactions, and so it is expected that by increasing the percentage of read-only transactions (and decreasing normal ones) the system is able to achieve much higher throughput values

Same logic follows for the abort percentages, as lowering the percentage of transactions that perform updates (and so may conflict) should lead to much lower abort percentages.

When comparing figures 4.9 and 4.10, partial replication shows better results, always achieving an higher throughput threshold when compared to full replication, this is in part because in this test each transaction only performs 2 writes, and so only a short amount of time is lost by partial replication when coordinating partitions during two-phase commit.

4.2.5 Alternate Paxos variants

In these tests we study our algorithm when using different Paxos variants to broadcast updates to all nodes in a partition. We compare ChainPaxos to the use of the ”original” MultiPaxos protocol and EPaxos, mentioned in section 2.2. Clients perform transactions that read 4 keys and write 50% of them (2 writes), all tests used 5 servers.

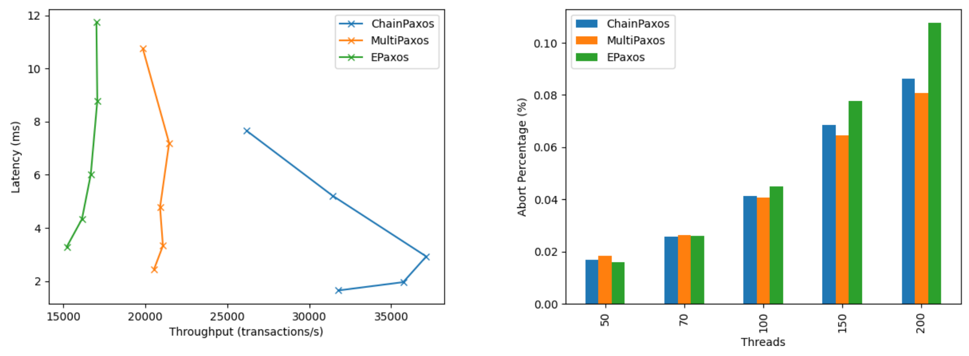


Figure 4.11: Alternate Paxos - Full replication

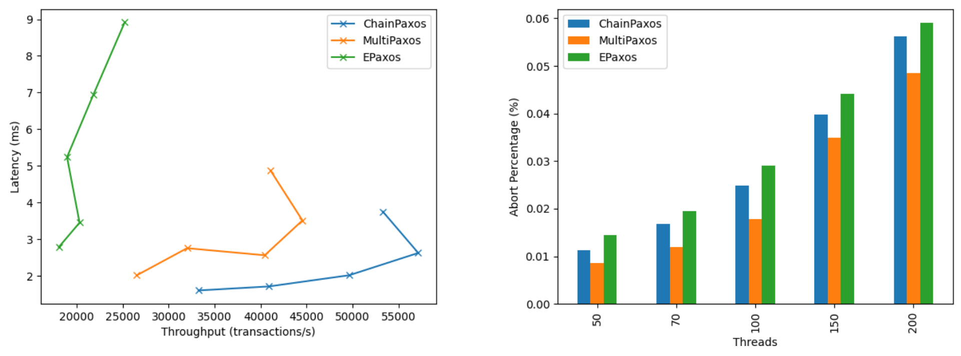


Figure 4.12: Alternate Paxos - Partial replication

In both tests the use of ChainPaxos shows better results, being able to reach a much higher throuput threshold when compared to the other two protocols. MultiPaxos is a leader-based Paxos protocol, like ChainPaxos, but suffers from requiring a larger number of messages to be sent over the network, because the leader has to send an accept message to every node in the Paxos group and then wait for a response from all of them. ChainPaxos only requires the leader to send an accept to the next node in the chain and then wait for a response from the tail, once the accept reaches it. It also allows a node in the middle of the chain to immediately know if the request was accepted by a majority, so it can execute the operation sooner.

EPaxos shows the worst performance, which might result from the fact that it does not use a designated leader, and requires two communication step in situations of contention, which are common in our tests.

4.3 Conclusion

Looking at the results, there is a clear improvement in system throughput when sharding the database into partitions. By decreasing the workload in each of the replicas, the system is able to respond to a much larger number of requests. Having less transaction contention in each of the nodes also seems to improve transaction abort rates, as seen in out tests. The major influence on latency in a partitioned database is the two-phase commit protocol, and its impact depends on the number of modified partitions during a transaction. If transactions make a lot of writes and modify a large number of partitions simultaneously then the two-phase commit protocol becomes a bottleneck and the partial solution performs worse than full replication. Still, in a realistic setting this may be a rare occurrence as transactions tend to only modify a small portion of the partitions, usually just one.

In the end, we also compared ChainPaxos to other famous Paxos variants, with ChainPaxos showing the best results. The original ChainPaxos paper only tested it to replicate simple operations over a single register, so its great to see that it also pairs well with database replication.

5 Conclusion

When designing large scale distributed applications programmers face the problem of selecting the appropriate consistency model. A weak consistency model provides good availability and system response times but allows replicas to diverge potentially leading to violations of data integrity. This might be acceptable for applications where having good client response times is the most important factor, but for applications where having consistent data is key, weak models are not enough. In those cases, a strongly consistent model is more adequate, ensuring system correctness across all replicas even if at the cost of performance. One possible way to minimize the performance problems of strong consistency models is to partially replicate the database, where each partition is replicated in only a subset of the nodes. This means that, during updates, only a fraction of the nodes in the system have to coordinate, requiring less communication trips among nodes and leading to better response times. Additionally, partial replication also has the advantage of allowing data to be stored only in locations where it is likely to be accessed.

In this dissertation, we propose a solution offering strong consistency (snapshot-isolation) for partially replicated databases, while still providing better performance when compared to other fully replicated systems.

The proposed algorithms build mostly on the Clock-SI[10] protocol, because it offers a good framework for building a partially replicated database supporting snapshot isolation. It solves well the problem of coordinating multiple nodes during a transaction, in a model where no node is aware of all the updates currently being done in the database. This is achieved by using timestamp-based mechanisms to delay reads and ensure consistent snapshots, and by doing the Two-Phase Commit protocol during commits.

Some changes had to be made, however, because the original Clock-SI paper only considers the case where each partition is stored in a single node. We adapted the original Clock-SI protocol to an architecture inspired by Spanner, where different data-centers, each with their own partitions, communicate with each other through a Paxos-based protocol to broadcast updates to all nodes in the same partition[22]. We also set the client as the coordinator for the 2PC protocol instead of relying on one of the datacenters to work as a leader, reducing the amount of communication trips that have to be done across datacenters and the amount of CPU resources that would have to be spent by the coordinating node.

Furthermore, we studied the use of ChainPaxos as a way to broadcast the updates in a partition with total order and in a efficient way. ChainPaxos is a novel Paxos variant that offers better performance when compared to other leader-based Paxos protocols, by greatly reducing the amount of messages that have to be sent and received by the leader when broadcasting updates to the other nodes in the Paxos group. The original ChainPaxos[27] paper only tests replicating simple read and write operations over a single register, so it was also interesting to test its behaviour in a transaction-based setting with more complex operations - like commits - and the results were very positive.

Lastly, we evaluated the performance of the proposed algorithm under different test configurations. In each test we compared full and partial replication and concluded that there is a high potential for performance gain by partitioning a database. Abortion rates also reduce when data is divided into partitions. We also studied response times when communication is supported by different Paxos variants, with ChainPaxos showing the best results by a large margin.

5.1 Contributions

In summary, this dissertation proposes an algorithm for database replication with strong replication in a distributed setting, that improves on transaction latency times by exploiting the advantages of partial replication. The algorithm is supported by an architecture that makes it simple to distribute partitions among datacenters and efficiently broadcast updates across nodes in the same partition through the use of ChainPaxos. Snapshot-Isolation consistency is guaranteed by implementing an algorithm for coordinating transactions based on the work of the Clock-SI[10] research.

5.2 Future Work

As mentioned in section 3.1.4, our current implementation of the algorithm does yet not implement the recovery mechanism for replicas and clients that fail during the Two-Phase Commit protocol. It would be very useful to get a working version running.

To improve on our work, it would be interesting to see how our algorithm performs under different isolation models other than snapshot-isolation, like serializability, for example. We could also test our algorithm when doing long distance geo-replication of data, since all our tests where performed in the Grid5000 cluster, with nodes located in France.

It would also be valuable to formally prove the definition of the algorithm, to confirm its correctness properties and prove that replicas converge to the same state and that snapshot-isolation consistency is working as intended.

Bibliography

[1] S. Abraham, K. Henry, and Sudarshan. *Database System Concepts* (cit. on pp. 5, 12).

[2] J. Baker et al. “Megastore: Providing scalable, highly available storage for interactive services”. In: (2011) (cit. on p. 2).

[3] H. Berenson et al. “A critique of ANSI SQL isolation levels”. In: *ACM SIGMOD Record* 24.2 (1995), pp. 1–10 (cit. on p. 13).

[4] E. Brewer. “CAP twelve years later: How the"rules"have changed”. In: *Computer* 45.2 (2012), pp. 23–29 (cit. on p. 6).

[5] E. A. Brewer. “Towards robust distributed systems”. In: *PODC*. Vol. 7. 10.1145. Portland, OR. 2000, pp. 343477–343502 (cit. on pp. 1, 6).

[6] B. F. Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154 (cit. on p. 38).

[7] J. C. Corbett et al. “Spanner: Google’s globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22 (cit. on pp. 2, 19).

[8] G. DeCandia et al. “Dynamo: Amazon’s highly available key-value store”. In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220 (cit. on p. 9).

[9] H. Du and D. J. S. Hilaire. “Multi-Paxos: An implementation and evaluation”. In: *Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02* (2009) (cit. on p. 10).

[10] J. Du, S. Elnikety, and W. Zwaenepoel. “Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks”. In: *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE. 2013, pp. 173–184 (cit. on pp. iv, v, 3, 22, 23, 27, 47, 48).

[11] V. Enes et al. “State-machine replication for planet-scale systems”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15 (cit. on p. 10).

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382 (cit. on p. 9).

[13] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: (2022) (cit. on pp. 11, 35, 36).

[14] J. Gray et al. "The dangers of replication and a solution". In: *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data*. 1996, pp. 173–182 (cit. on p. 14).

[15] M. P. Herlihy and J. M. Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492 (cit. on p. 5).

[16] B. Kemme and G. Alonso. "A new approach to developing and implementing eager database replication protocols". In: *ACM Transactions on Database Systems (TODS)* 25.3 (2000), pp. 333–379 (cit. on pp. 13, 15, 16).

[17] B. Kemme and G. Alonso. "A suite of database replication protocols based on group communication primitives". In: *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No. 98CB36183)*. IEEE. 1998, pp. 156–163 (cit. on pp. 7, 15).

[18] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563 (cit. on p. 5).

[19] L. Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25 (cit. on p. 10).

[20] C. Li et al. "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, 265–278. ISBN: 9781931971966 (cit. on p. 6).

[21] Y. Lin et al. "Middleware based data replication providing snapshot isolation". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 2005, pp. 419–430 (cit. on p. 16).

[22] H. Mahmoud et al. "Low-latency multi-datacenter databases using replicated commit". In: *Proceedings of the VLDB Endowment* 6.9 (2013), pp. 661–672 (cit. on pp. 2, 20, 21, 47).

[23] H. Moniz et al. "Blotter: Low latency transactions for geo-replicated storage". In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 263–272 (cit. on p. 2).

[24] I. Moraru, D. G. Andersen, and M. Kaminsky. "There is more consensus in egalitarian parliaments". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 358–372 (cit. on p. 10).

[25] D. Ongaro and J. Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319 (cit. on p. 10).

[26] F. Pedone, R. Guerraoui, and A. Schiper. "The database state machine approach". In: *Distributed and Parallel Databases* 14.1 (2003), pp. 71–98 (cit. on pp. 17, 18).

[27] F. Pedro, L. João, and P. Nuno. "High Troughput Replication with Integrated Membership Management". In: (2022, under submission) (cit. on pp. iv, v, 3, 10, 28, 48).

[28] D. Serrano et al. "Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation". In: *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. IEEE. 2007, pp. 290–297 (cit. on pp. 16, 17).

[29] A. Sousa et al. "Partial replication in the database state machine". In: *Proceedings IEEE International Symposium on Network Computing and Applications. NCA 2001*. IEEE. 2001, pp. 298–309 (cit. on pp. 18, 19).

[30] R. Van Renesse and F. B. Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *OSDI*. Vol. 4. 91–104. 2004 (cit. on p. 9).

[31] W. Vogels. "Eventually Consistent". In: *Commun. ACM* 52.1 (2009), 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432. URL: https://doi.org/10.1145/1435417.1435432 (cit. on p. 5).