TÉCNICO LISBOA

A Distributed and Hierarchical Architecture for Deferred Validation of Transactions in Key-Value Stores

João Bernardo Sena Amaro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. João Carlos Antunes Leitão

October 2018

Acknowledgements

I would like to acknowledge my dissertation supervisor Prof. Luís Rodrigues for giving me the opportunity to work on this thesis under his supervision. His insights, support and sharing of knowledge were fundamental to produce this thesis.

Moreover I would like to thank Manuel Bravo, Miguel Matos, and Paolo Romano for the fruitful discussions and comments during the preparation of this work.

I would also like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life.

To each and every one of you, thank you.

Abstract

Key-value stores are today a fundamental component for building large-scale distributed systems. Early key-value stores did not offer any support for running transactions but, with time, it became obvious that such support could simplify the application design and development. The key challenge in this context is to support transactional semantics while preserving the scalability properties of key-value stores. In this thesis we propose an architecture to perform transaction validation in a distributed and scalable manner. The architecture exploits the fact that in the current key-value stores, data is partitioned across multiple servers and that correlated data can be stored in the same server if the appropriate partitioning function is used. In this context, transactions that access correlated data in the same partition (local transactions) can be validated and committed concurrently, by different servers. Transactions that access multiple partitions (distributed transactions) can take longer to be validated and committed, but do not interfere with local transactions. The architecture is based on a tree of transaction *validators*, where the leaf nodes are responsible for single partitions, offering higher throughput and lower latency, while non-leaf nodes are responsible for several partitions. We have performed an extensive experimental validation of the proposed architecture that highlights its advantages and limitations. The evaluation shows that, in some scenarios, the proposed architecture can offer improvements up to 23% when compared to other validation and commit strategies, such as distributed two-phase commit.

Keywords

Distributed Systems; Transactions; Key-value Stores;

Resumo

Os sistemas de armazenamento chave-valor são hoje um componente central dos sistemas distribuídos de grande escala. Os primeiros sistemas deste tipo não ofereciam suporte para transações, mas com a sua evolução tornou-se relevante oferecer este tipo de garantias. O desafio que se coloca é como oferecer suporte transacional sem comprometer o elevado débito que caracteriza os sistemas chave-valor. Nesta tese propomos uma nova arquitetura para realizar a validação das transações, de forma distribuída. A arquitetura tira partido to facto dos dados guardados nos sistemas de armazenamento chave-valor atuais estarem particionados em vários servidores e de que dados correlacionados podem ser guardados no mesmo servidor desde que seja usada uma função de particionamento adequada. Neste contexto, transações que acedam a dados correlacionados da mesma partição (transações locais) podem ser validadas e confirmadas de forma concorrente, por diferentes servidores. Transações que acedam a múltiplas partições (transações distribuídas) podem demorar mais a ser validadas e confirmadas, mas sem atrasarem o processamento das primeiras. A arquitetura recorre a uma hierarquia em árvore de *validadores*, em que os nós folha são responsáveis por uma só partição, oferecendo maior débito e menor latência, enquanto que os nós que não são folha são responsáveis por múltiplas partições. Apresenta-se uma extensa avaliação experimental da arquitetura proposta que permite aferir as suas vantagens e limitações. Esta avaliação mostra que, nalguns cenários, a arquitetura proposta consegue melhorias até 23% em relação a outras estratégias alternativas para fazer a validação das transações, tal com a confirmação distribuída em duas fases.

Palavras Chave

Sistemas Distribuídos; Transacções; Armazenamento Chave-Valor;

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

RDBMS Relational Database Management Systems

ACID Atomicity Consistency Isolation Durability

2PC Two-Phase Commit

PCC Pessimistic Concurrency Control

OCC Optimistic Concurrency Control

I/O Input/Output

1

Introduction

Contents

Key-value storage systems have today a crucial role in large scale distributed systems due to their scalability and ability of supporting high access rates. An example of a system of this kind is Cassandra [1], which is able to store terabytes of information while supporting hundreds of thousands of accesses per second. Unfortunately, in order to achieve good performance, these systems do not support strong consistency models and do not implement concurrency control, which makes the development of applications that use key-values stores harder and prone to errors [2], when compared with application that use traditional databases. As such, there has been a growing interest in developing mechanisms that allow to strengthen the guarantees offered by these systems without strongly penalising their performance.

This thesis addresses the problem of supporting transactions in key-value storage systems. In particular, the focus of this work is on developing mechanisms that allow the process of validating transactions to scale.

1.1 Motivation

A transaction is an abstraction that allows to execute a sequence of operations as if it was an indivisible single operation that is executed atomically. This intuitive notion can be described more precisely by a set o properties known as the Atomicity Consistency Isolation Durability (ACID) properties [3].

Key to the implementation of transactions is a concurrency control mechanism, whose purpose is to ensure that the interleaving of the operations executed by concurrent transactions produces a result that is *consistent*. One of the strongest isolation levels is serialisability. Serialisability states that the execution of concurrent transactions should be equivalent to some sequential execution of the same transactions. Although there are many different techniques to implement concurrency control, any concurrency control algorithm must be able to totally order transactions that access the same objects. The need for totally ordering transactions represents a fundamental impairment for the scalability of a transactional system.

Therefore, in this thesis we look for techniques that can mitigate the bottlenecks that may result from the need for totally ordering transactions. In particular, we are inspired by the work of [4], that uses an hierarchical network of transaction validators, to validate transactions that access disjoint sets of objects in parallel. Concretely, the goal of this thesis is to combine the hierarchical communication pattern used in [4] with message batching techniques to build a novel transactional system that is able to support a larger number of clients and higher throughput.

1.2 Contributions

This thesis proposes, implements, and evaluates a novel approach to execute the certification phase of optimistic concurrency control when implementing transaction in distributed key-value stores. The contributions of this dissertation are the following:

- An hierarchical architecture for deferred validation of transactions.
- An identification of the benefits and limitations of the proposed architecture.

1.3 Results

This work has produced the following results:

- A detailed description of the proposed architecture and its implementation.
- An implementation of the proposed architecture over an industry used key-value store named Riak KV [5].
- An experimental evaluation of the prototype and a comparison of its performance against other approaches such as distributed two-phase commit.

A paper describing parts of this work was accepted as a communication and presented at INForum 2018 [6].

1.4 Organisation of the Document

The remaining of this document is organised as follows. Chapter 2 briefly describes the concepts and systems related with this work. Chapter 3 describes the proposed architecture, how transactions are executed over it and how it was implemented over Riak KV [5]. Chapter 4 presents the results of the experimental evaluation. Finally, Chapter 5 concludes the dissertation.

2

Related Work

Contents

This chapter surveys the main concepts and systems that are relevant for our work. We start by briefly describing the operation of key-value stores. Then we introduce the concept of a transaction and how it can be implemented. Then we present the main classes of concurrency control mechanisms that exist and the importance of total order in this context. Finally we present systems that use total order protocols and that are relevant for the work we have developed.

2.1 Key-Value Stores

Key-value stores are a category of storage systems that were born out of the necessity of increasing the scalability and performance of traditional Relational Database Management Systems (RDBMS). Traditional RDBMS were unable to provide the performance required by large internet applications, that need to store large volumes of data that was concurrently accessed by millions of users.

RDBMS use a structured data model that allows very complex data manipulation and query operations, but the support for such functionality is computationally expensive. Also, many RDBMS were not build with distribution in mind, which makes them very difficult to scale. Key-value stores, on the other hand, use a much less structured data model where data is stored as key-value pairs. The value captures the state of an object of some kind and the key uniquely identifies that object. Accessing data in this model is a lot simpler when compared to the relational model, clients can do two basic operations: (1) *get*, which is a simple way to fetch an object by key; (2) *put*, that allows the client to write an object associated to a given key.

Due to the simplified data model used by key-value stores and given the limited set of operations they support, key-value stores are much easier to scale and can provide better performance, when compared to previous RDBMS. A fundamental strategy to support scale consists in storing different objects on different servers, using a simple hashing function to assign keys to nodes; this avoids the need for a directory service to locate objects in a cluster of many nodes. In essence these systems traded functionality and consistency guarantees for performance. For instance, most key-value stores lack support for transactions offering the ACID properties.

2.2 Transactions

A transaction is a sequence of read and write operations that are executed as if they were just a single operation executed atomically. The transaction either succeeds (commits) or not (aborts, rolls back). If it aborts, the transaction can be safely retried, since no intermediate results from the aborted run took effect. This abstraction simplifies error handling, because the application does not have to worry about partial failures. Also, transactions shield the programmers from dealing explicitly with concurrency

control, given that the final outcome is guaranteed to be the same as if the transaction was executed in serial order with regard to other transactions.

2.2.1 ACID Properties

The acronym ACID was first introduced in [3] and it describes the safety guarantees provided by transactions. It stands for Atomicity, Consistency, Isolation, and Durability, which can be defined as follows:

Atomicity: states that a transaction is indivisible, and therefore, it is not possible to observe partial results. In other words, either all operation that execute a transaction take effect (if the transaction commits) or none of the operations take effect (if the transaction aborts).

Consistency: refers to the application specific notion of consistency. Every application has its own notion of consistency, often translated into what is called the application invariants (statements about the application data that must always be true). A transaction is said to be consistent if it keeps the application invariants true after it has been executed. Each application has its own invariants, thus the responsibility of ensuring that transactions are consistent, belongs to the application developers and not to the database. Although, some databases provide developers with mechanisms to make this easier, and in this case databases have a partial role in keeping data consistency.

Isolation: means that two concurrent transactions are isolated from each other. The classic notion of isolation was formalised as *serialisability* [7], which states that the outcome of running multiple transactions concurrently is guaranteed to be the same as if these transactions had run serially. Therefore, the programmer does not need to worry about concurrency control and can program "as if" the transaction was going to run alone in the system.

Durability: refers to the promise that once a transaction is committed successfully, any data that was written by it will be stored durably (will not be forgotten), even if there is an error or fault of some kind. In not replicated databases it usually means that once a transaction is committed all its writes were stored in non volatile storage. In replicated databases it means that the data was replicated to a certain number of nodes.

In this dissertation we are mainly interested on how atomicity and isolation can be guaranteed.

2.3 Guaranteeing Atomicity

The challenge of guaranteeing atomicity is to ensure that in the presence of partial failures, all outcomes of a transaction either persist or none do. Partial failures can have many forms: in a transaction that has several reads and writes a partial failure can be one that happens midway through the execution of the transaction, and only half of the operations have executed; another example is a single write

transaction, and a power failure happens while the transaction was writing to disk. Guaranteeing an all or nothing result in these cases requires a recovery algorithm, like the ARIES [8] algorithm, that is capable of recovering a partially executed transaction to a clean state.

The ARIES algorithm uses write-ahead logging: before any update is performed in the database, a redo/undo entry is appended to a persistent log. This way the log has enough information to redo or undo the effect of any update. If a failure occurs while an update is being executed, there is no problem, because the corresponding log entry is already persisted and can be used to redo the update. Recovery works in two phases. The first phase replays all the operations in the log, to bring the system to the state it was before the crash (this includes the effects of transactions that have committed but also of transaction that were still executing when the failure occurs). Then, in the second phase all the operations of transactions that did not committed are undone.

Guaranteeing atomicity in distributed scenarios is even harder, because transactions can fail in some nodes but succeed on others. In order to guarantee atomicity all nodes must agree on the outcome of the transaction, i.e, the transaction either commits or aborts at all nodes. This problem is related to the consensus problem and can be solved with the help of a consensus algorithm, like Paxos [9]. However, many commercial database systems use a simpler Two-Phase Commit (2PC) Protocol [10], that can be implemented more efficiently than a fault-tolerant consensus protocol, but may block in some faulty scenarios. The next subsection describes two-phase commit in detail.

Regardless of the consensus protocol used, a recovery algorithm is still necessary at each node. In fact both protocols should work with each other to recover form possible faults.

2.3.1 Two-Phase Commit

2PC is a protocol used to atomically commit transactions across several nodes, i.e. to make sure that either all nodes commit or all nodes abort. The protocol works in two phases (hence the name): prepare and commit/abort.

Prepare Phase: is the first phase of the protocol. Once a transaction has finished executing and is ready to commit, the coordinator (which is the process responsible for coordinating the distributed procedure) starts the prepare phase by sending a *prepare request* to all the data nodes involved in the transaction asking them if they are able to commit. Then each data node verifies if it can or not commit the transaction and communicates that information to the coordinator, in the form of a *prepare response*. The coordinator keeps track of the responses it receives from the nodes and compiles them into a final prepare result. The outcome of the prepare phase is a decision to commit the transaction if all the nodes replied with a ”yes” (i.e., if they can commit the transaction), or a decision to abort the transaction otherwise.

Commit/Abort Phase: starts right after the coordinator computes the outcome of the transaction. If

the outcome is a decision to commit, it means that all data nodes agreed to commit the transaction, so the coordinator starts the second phase by sending a *commit message* to all the data nodes. If the outcome is a decision to abort, at least one of the data nodes could not commit the transaction, so all nodes must abort it. To do so, the coordinator sends an *abort message* to all the data nodes. On receiving the message with the outcome from the coordinator, data nodes commit or abort the transaction accordingly.

Note that if a transaction involves a single node, the two phases above can be collapsed. In fact, the (single) participant has all the information required to decide the outcome of the transaction at the end of the prepare phase, and can immediately proceed to commit or abort the transaction accordingly.

There are two crucial points in this protocol that make sure it guarantees atomicity. The first is when data nodes respond to a prepare message, if they reply with a ”yes”, they are making a promise that no matter what they will commit the transaction. The second crucial point is also around promises, and is when the coordinator calculates the final prepare result, if it decides to commit or abort the transaction, it can not go back on that decision. In order for these nodes to keep their promises they have to durably store them in some way, such that, if an error, crash, or fault occurs, they can recover their past decisions back and not break the promise.

Keeping these promises can have some implications, specially in terms of performance. An example of this is the following, considering a scenario where two coordinators try to commit one transaction each, and transactions from both coordinators have object A in common. If the data node responsible for object A responds first to coordinator one, and promises to commit the value it has, coordinator two has to wait to commit its value of object A. If in this process coordinator one crashes, coordinator two remains blocked until coordinator one recovers, which can be a long time.

The network costs of this protocol can be described in number of communication steps (latency) and total number of messages exchanged (bandwidth) as follows:

$$N = number\ of\ data\ nodes\ involved\ in\ the\ transaction \tag{2.1}$$

$$communication\ steps = \begin{cases} 2 & N = 1 \\ 4 & N > 1 \end{cases} \tag{2.2}$$

$$total\ messages = \begin{cases} 2 & N = 1 \\ 4 \times N & N > 1 \end{cases} \tag{2.3}$$

If one data node is involved in the transaction the protocol requires 1 prepare message from the client to the data node and 1 prepare result message from the data node to the client, to commit a transaction, thus it takes 2 communication steps and uses 2 network messages in total. On the other hand, if the transaction involves more than one data node, the protocol require 4 communication steps in total that

correspond to the 4 messages exchange between the client and the data nodes involved (1 prepare, 1 prepare result, 1 commit/abort, 1 commit/abort result), and requires a total o 4 messages per data node.

2.4 Guaranteeing Isolation

When it comes to the problem of guaranteeing isolation among concurrent transactions, the first question to ask is which isolation level do we want to guarantee. In fact, several different isolation levels have been proposed and implemented by database researchers and vendors. The next subsection will explore the main isolation levels available in today's databases.

2.4.1 Isolation Levels

As described in Section 2.2.1 the classic notion of isolation was formalised as *serialisability* [7], which means that the execution of transactions can occur concurrently but the outcome is equivalent to a serial execution.

Unfortunately, in practice, using serialisability as an isolation level may be expensive and may lead to poor performance. As an alternative, weaker isolation levels were proposed. These weaker isolation levels sacrifice on accuracy for performance. In practice this translated into what are called concurrency anomalies or race conditions. Below we present the several isolation levels that have been proposed and the next subsection will discuss some of the anomalies that may be observed when serialisability is not used:

Read Uncommitted: is the weakest isolation level that can be used, and the one that provides the best performance. With this isolation level dirty reads, non-repeatable reads and phantom reads are possible.

Read Committed: this isolation level guarantees that no dirty reads are possible. Non-repeatable reads and the phantom reads are still possible.

Repeatable Read: is a step up the read committed isolation level, guaranteeing everything Read Committed guarantees plus the lack of non-repeatable reads, i.e. repeatable reads of the same object will always return the same value.

Snapshot Isolation: as the name implies guarantees that any reads done by a transaction will be done as if the transaction is accessing a snapshot of the entire database. This isolation level guarantees that dirty reads, non-repeatable reads and phantom reads are not possible. With this isolation level the write skew anomaly is possible.

2.4.2 Race Conditions and Anomalies

The concurrent transactions of transactions may create race conditions that may result in the following anomalies:

Dirty read: this anomaly happens when one transaction writes a value to the database, but does not commit or abort, and other concurrent transaction reads the value that the first transaction wrote but did not yet commit.

Non-repeatable reads: happen when during the execution of a transaction the same object is read more than once and the value of it is different between reads.

Dirty write: this anomaly happens when two concurrent transactions write on the same object, and one of them overwrites the changes of the other before they are committed.

Phantom read: happens when a write in one transaction changes the result of a search query in another transaction.

Write skew: happens when two concurrent transactions read the same objects, and then update some of those objects (different transactions may update different objects).

2.4.3 Concurrency Control Mechanisms

Regardless of the isolation level used, a concurrency control mechanism is always needed to enforce it. There are two main classes of concurrency control mechanisms: pessimistic and optimistic [11]. The next two subsections will going into each one separately.

2.4.3.A Pessimistic Concurrency Control

Pessimistic Concurrency Control (PCC) mechanisms work, as the name implies, based on a pessimistic assumption that conflicts among concurrent transaction will happen frequently. Therefore, for every operation executed by a transaction, the PCC mechanism checks for conflicts and, if needed, the operation is blocked until it can run without violating the desired consistency level. Note that a conflict occurs if two concurrent transactions access the same data item and at least one of the operations is an update. Conflict detection and resolution is usually performed using locks.

One disadvantage of this approach is the extra cost added to every operation. This, in a scenario where conflicts are rare, may reduce the overall system performance. On the up side, if conflicts are common, resolving conflicts as soon as possible can improve the system performance, since it prevents transactions from making progress that needs later to be aborted.

2.4.3.B Optimistic Concurrency Control

In contrast with PCC mechanisms, the Optimistic Concurrency Control (OCC) mechanisms assume that conflicts among concurrent transactions are rare. Therefore OCC mechanisms do not check for conflicts until a transaction is finished. When a transaction finishes, the concurrency control mechanism will check if any conflicts occurred during the execution of the transaction and commit/abort the transaction accordingly. This process is known as *transaction validation*.

An advantage of OCC mechanisms when compared to PCC mechanisms is the fact that they do not add any overhead to the execution of transactions; instead, they add an extra validation step at commit time. A downside of this approach is if conflicts are common, a transaction may waste resources executing operations that are doomed to abort. Also, because no early conflict resolution is performed, OCC may exhibit higher abort rates when compared to a PCC mechanism.

In essence, the choice of the concurrency control mechanism to use is highly dependent on the expected workload. In general, PCC mechanisms are better in scenarios where conflicts among concurrent transactions are common, while OCC mechanisms are better in scenarios where conflicts are rare.

In this work our focus is on OCC mechanisms due to their potential of achieving an higher throughput in scenarios where conflicts among concurrent transactions are rare.

2.5 Transaction Validation

Data storage systems that use OCC mechanisms execute transactions in four steps:

1. Begin: the client starts the transaction and records a timestamp marking a (logical) point in time when the transaction started.
2. Modify: the transaction read and write operations are executed. Writes are executed as tentative.
3. Validation: check whether the transaction has conflicts with other already committed transactions.
4. Commit/Rollback: according to the result of the validation, the tentative writes of the transaction are committed or rolled back.

The way the validation of transactions is done, depends on the isolation level used. When enforcing serialisability, the validation of transactions is done by checking if previously committed transactions have modified data that was read and/or written by the transaction being validated. In such case, the transaction must be aborted. In practice, this means that tentative writes are discarded and not applied to the database.

The validation procedure described above assumes that transactions can be validated in serial order. Most of the systems we have studied rely on total ordering algorithms to achieve this.

2.6 Event Ordering Algorithms

In the literature we have identified four categories of ordering protocols that are used in the context of transactional systems, they are: fixed sequencers, rotative sequencers, distributed sequencers with coordination in line, and distributed sequencers with deferred stabilisation.

2.6.1 Fixed Sequencer Algorithms

The fixed sequencer algorithms work, as the name implies, based on the existence of fixed sequencer node, that has the role of assigning sequence numbers to events. The remaining nodes interact with the sequencer node whenever they want to order an event.

A downside of this class of algorithms is that the fixed sequencer can become a bottleneck pretty fast as all clients must contact it to order events. Examples of systems that use this class of algorithms are CORFU [12], Tango [13], vCorfu [14] and Megastore [15].

2.6.2 Rotative Sequencer Algorithms

Algorithms based on a rotative sequencer algorithms work by shifting the role of being a sequencer from node to node. This is done by passing a token from node to node in a logical ring. The node that has the token plays the sequencer role, and can order events without coordination with other nodes. A node that has events to order need to wait until it receives the token; only then it has exclusive access to assign sequence numbers to the events. This class of algorithms has been used in the past in message ordering protocols [16], but it can easily become a bottleneck in large scale systems due to the latency associated with passing the token from node to node.

2.6.3 Distributed Sequencers With Coordination In Line

This class of algorithms is a variant of an algorithm by Dale Skeen originally proposed in [17]. Each node involved in the ordering process proposes a sequence number, then a coordinating process uses all the proposes and assigns a final sequence number (usually the maximum of all the proposed sequencer numbers). An example of a system that uses this principle is Spanner [18]. A downside of this class of algorithms is the fact that the ordering of events is blocked by the distributed coordination process, that requires at least two communication steps.

2.6.4 Distributed Sequencers With Deferred Stabilisation

Distributed sequencers with deferred stabilisation also use a coordination process that gathers proposals, but instead of running in the critical path of nodes it happens in a deferred manner. Each participant gathers several events to order and sends them as a batch to the coordinating processes. These processes wait from proposals from all the participants, and based on them, order all the events received in a total order, consistent with the order seen by each participant. By allowing batching when ordering events, the system throughput can be increased at the cost of an higher latency. Examples of systems that use this class of algorithms are Chariots [19], Eunomia [20] and Calvin [21].

2.7 Systems That Rely On Event Ordering

2.7.1 CORFU

CORFU [12] is a totally ordered log abstraction built over a cluster of flash storage units. CORFU's architecture is composed by a sequencer and an array of flash units. Flash units are treated as passive storage devices that are accessed over the network and the sequencer is used as a *pre-assignment* technique, that is responsible for ordering log records. In CORFU the ordering of records is decoupled from their persistence, allowing parallelism when appending to the flash units. The functionality of CORFU is all made available to clients in the form of a client library. It works by implementing three fundamental functions:

- Mapping function: responsible for mapping logical log positions to flash pages in the cluster of flash units.
- Tail finding mechanism: for finding the next available logical position in the log.
- Replication protocol: to persist log records in several flash pages in the flash unit cluster.

The mapping function works by having each client maintain a local read-only copy of a structure called a *projection*. The projection structure splits the log into disjoint ranges, and each range is mapped to a list of extents within the address space of individual flash units. In essence, a projection is a mapping of logical addresses space to flash pages. When a flash unit fails or the log grows past the maximum available position in the projection a reconfiguration mechanism takes place to install a new projection in all the clients. To treat the address space as an appendable log, clients must be able to find the tail of the log and write to it. CORFU implements two tail finding mechanisms:

- Contention mechanism: relies on the flash unit *write-once* semantics that CORFU requires. Clients that wish to append records to the log try to concurrently write to the first position. One client wins,

while the rest fails. The client that wins successfully appends to the log while the others have to try again in in the next position.

- Sequencer: is used in CORFU as an optimization over the contention mechanism. The sequencer is responsible for assigning empty log positions to clients. A client that wants to append a record to the log, first contacts the sequencer to get a log position, once he receives the position from the sequencer, he uses his local projection to find the flash unit responsible for storing that log position and persists the record there without contention from other clients.

When writing to a log position, CORFU clients use a chain replication protocol. It works in two steps: (1) clients use their local projections to map a single log position to a set of flash pages, and (2) write to this set of flash pages in a deterministic order waiting for each flash unit to respond before moving to the next one. The write is successfully completed when the last flash unit in the chain is updated. As a result, if two clients attempt to concurrently update the same replica set of flash pages, one of them will arrive second at the first unit of the chain and receive an overwritten error.

CORFU's design ensures that the log throughput is not a function of the I/O bandwidth of any single flash unit, instead clients can append records as fast as the sequencer can assign positions.

2.7.2 Tango + CORFU

Tango + CORFU [13] is a log based object store intended to be used as a building block of highly available metadata services. Objects stored in Tango are replicated, in-memory data structures that have their state durably stored in a log. Tango uses CORFU as a log, taking advantage of all the properties that it provides.

The state of a Tango object exists in two forms, a *history*, which is an ordered sequence of updates stored durably in the CORFU log, and several *views*, that are full or partial copies of an object stored in the memory of the clients. Because the state of Tango objects is stored as an ordered sequence of updates in the log, application developers can roll back to any point in the history of an object simply by creating a new instance and syncing it with the appropriate prefix of the log.

The implementation of a Tango object requires three main components. The first is a view of the object, which is an in memory representation of the object's state. The second is an *apply()* function, responsible for updating the object's view when there are new updates in the log. And the third is the object's public interface with mutator and accessor methods. Mutator methods do not change the object's view directly, instead they append the mutation to the log. When accessor methods are called, they first get the latest mutations from the log, update the object's view using the *apply()* function and only then execute the accessing logic.

The log based design of Tango allows strongly consistent operations across objects to be achieved trivially, by just using reads and appends on the log. But the authors of Tango went a step further and implemented an optimistic concurrency control system over the log, that allows applications to run transactions across objects. It works by appending a speculative commit record to the log that ensures atomicity of the transaction. It marks a point in the total ordering of updates at which the changes of the transaction can be made visible. Each commit record includes a *read set*, a list of object read during the transaction, that is used to ensure transaction isolation. A transaction succeeds if the objects in the read set have not been changed since they were read.

Having all the object's state in the same log is important to offer strongly consistent operations and transactions, however it might introduce the *playback bottleneck* problem. If clients just want to host views of certain objects, consuming updates from all the objects can be wasteful. To overcome this problem Tango implements objects streams over a single log, allowing clients to selectively consume the log updates corresponding to certain objects. This means that clients will host a layered partition of the log with all the same strongly consistent and transactional semantics of the full log.

To allow clients to consume the log via a streaming interface the authors proposed a modification to the underlying CORFU design, a stream header that is present in every log record. This stream header includes the stream ID as well as back pointers to the last k records of that stream, allowing clients to construct a linked list of records from the same stream. Reading and appending to streams work as before, the only difference is in the appends that require the client to build the stream header before appending a record.

2.7.3 vCorfu

vCorfu [14] is a strongly consistent cloud-scale object store built over a log just like Tango + CORFU, that uses a *stream materialisation* technique to overcome the playback bottleneck problem that logs have.

Stream materialisation is a technique that enables the design of systems that store large quantities of state in the form of a log without introducing the playback bottleneck problem to clients. Stream materialisation is a step ahead of Tango's streams, which are implemented as simple tags on the log, materialised streams are as the name says a materialisation of the streams, which means that streams are stored as independent logs that support random and bulk reads just like a normal log. vCorfu uses a materialised stream for each object it stores.

Log appends to a materialised stream are stored in two replicas, the *log replicas* and the *stream replicas*. These two replicas store the same data, but unlike Tango + CORFU the data is indexed in different ways. Log replicas index the log records according to their position on a global log, while stream replicas store log records indexed by their position in the stream log. This replication scheme,

inspired in the idea of Replex [22] of using different replicas to store the same data indexed in different ways, allows clients to directly read the latest version of an object by simply contacting the corresponding stream replica.

Materialised streams can be seen as independent logs. Appending to them is very similar to appends in Tango, the difference introduced is in the client-sequencer communication. Clients that contact the sequencer receive two log positions instead of one. One corresponding to the global log and one corresponding to the stream log. Using the two log positions clients persist the record in both replicas using a chain replication protocol. Because appends to these *stream logs* have a corresponding append in the global log, there is a total order over all appends from all the streams, enabling vCorfu to support atomic appends across streams (objects).

As Tango, vCorfu went a step further the atomic updates and implemented a transactional system that enables application to run transactions across objects. Transactions are implemented by combining the atomic update capabilities with the optimistic concurrency control techniques used in Tango. The sequencer is exploited as a lightweight transaction manager, but unlike Tango it only issues log positions to commit a transaction if the transaction has no conflicts, an important optimisation that frees clients from the transaction validation logic every time they playback the log.

The vCorfu system can be seen as a new version of Tango, a version with all the same functionality but with a better implementation and performance. On the other hand this also means that vCorfu suffers from the same problem that Tango does, the sequencer I/O bottleneck. Chariots and Kafka are examples of systems that avoid this problem by not using a sequencer, an overview of how they do it lies ahead in the report.

2.7.4 Megastore

Megastore [15] is a log based storage system that blends the scalability of NoSQL data stores with the convenience of a traditional RDBMS, providing both high availability and strong consistency guarantees.

The interface and data model provided by Megastore is very similar to the one provided by a relational database, applications can create schemas, tables and indexes, that can later be queried using a SQL like language. An addition that Megastore introduced over the conventional relational databases is the ability to partitioned the data in what the authors call *entity groups*. Entity groups are an a priori grouping of related data for fast operations that can be seen as small databases with ACID semantics.

Each entity group has its own write-ahead log, where all the updates that happen inside it are recorded. The log is independently and synchronously replicated over a wide area, and is stored in Bigtable [23], leveraging its scalability and fault-tolerance capabilities. Updates inside entity groups are done as single phase ACID transactions, while updates across entity groups require expensive two

phase-commit protocols or Megastore's asynchronous messaging system.

When it comes to transactions inside an entity group, their life cycle is the following. First the client obtains the timestamp of the latest committed transaction from the log, then reads from a consistent snapshot using the timestamp gathered in the first step and writes into a log record. Finally, in order to commit the transaction the client appends the log record with all the writes to the end of the log using the Paxos algorithm. After the record is appended to the log the updates are applied to the data in Bigtable.

To allow clients to read from a consistent snapshot, Megastore relies on Bigtable's capability of storing multiple values for the same row/column pair. This makes the implementation of multi version concurrency control trivial: updates within a transaction are written with the timestamp of the transaction while concurrent reads that use lower timestamps never see partial updates. In other words reads are isolated from the writes.

Appending to the log is done by running an independent instance of the Paxos algorithm for each log position, so when a client wants to append a record to the log, he has to propose the record he wants to append as the one to be placed at the tail of the log, and has to block until he gets an answer from the majority of the replicas. If another client takes the position first, he has to abort and start again in the next log position. Because Paxos is responsible for assigning log positions to clients it can be seen as the log sequencer.

Transactions across entity groups are also possible, using Megastore's asynchronous messaging system. Entity groups can communicate between them in an RPC style, each entity group has an inbox to where other entity groups can send messages. A client that wants to update several entity groups, runs a transaction to atomically send updates to several entity group's inboxes. Each entity group will then process the updates they receive as an isolated transaction.

2.7.5 Chariots

Chariots [19] is a highly available, geo-replicated, causally-ordered log that overcomes I/O bottlenecks of existing sequencer based log designs.

The Chariots paper includes two main contributions:

- FLStore: sequencer free log store.
- Chariots: multi stage log replication pipeline.

FLStore or Fractal Log Store is a log store that uses a post-assignment technique to assign log positions to clients. It consists of two groups of servers, the *log maintainers* and the *indexers*. Log maintainers are responsible for disjoint ranges of the log, for each range they store log records, they

serve read requests and they assign log positions. Indexers are responsible for indexing log record's tags.

The post assignment technique works based on the fact that log maintainers are responsible for distinct ranges of the log, which means that a log position can only be assigned, written and stored by a single log maintainer. Appending to the log works by sending a record to a log maintainer at random, the log maintainer will assign it the next available position in the range that he is responsible for and the record is persisted in that position. This capability of assigning log positions without coordinating with others is a big improvement over CORFU, because the append throughput is now a function of the aggregate I/O bandwidth of *all* log maintainers and not a function of the I/O bandwidth of a *single* machine.

On the other hand, allowing clients to contact any log maintainer at random introduces a couple of new problems. The first is the existence of holes at the tail of the log, which may delay the visibility of certain records. This happens because reading a certain log position is only allowed when all the previous log positions are filled up. Consider the case of an append to position 15 of the range 10-19, if the range 0-9 is empty or is not yet full, the record appended to position 15 will not be available for reading. The second problem introduced by this design is the lack of explicit order between records appended by the same client. For example, when a client appends two records sequentially, he can not know the order between them in the log, even though they were added sequentially. When it comes to solving these problems, no solutions exist for the first problem, and for the second one two solutions are described in the paper but they all have some drawbacks that might impact the client or the system performance.

Chariots, the second main contribution of the paper, is a log replication pipeline designed to replicate a log to new geographic locations, but its ideas can also be applied to local replication. It runs over FLStore and is responsible for processing all append requests. FLStore is used as the persistence layer in the pipeline. Chariots' design favours availability of the log by relaxing the log consistency, providing applications with a causally ordered log.

The Chariots pipeline works as follows. When a client wants to append a record to the log, it sends a request with the record to be appended to Chariots. The record enters the pipeline, is tagged with its causal dependencies and is moved into a queue where it is assigned a position in the log. After having a position assigned, the record is sent to FLStore, more precisely to the log maintainer responsible for the position that it was assigned. Finally after being persisted in FLStore, the senders from Chariots catch it and send it to other Chariots instances for them to incorporate it.

Appends received from remote data centres, go through the pipeline as a local append would, the main difference is in the queue stage, where their causal dependencies are verified. If they are met, the record can be inserted in the local log, if they are not met the record is kept in the queue until they are.

2.7.6 Kafka

Kafka [24] is a log based messaging system that was developed for collecting and delivering high volumes of data with low latency. It combines the benefits of log aggregators and messaging systems to build a system that allows applications to consume data in real time.

As every messaging system, Kafka's interface and data model is around messages. The smallest data storage unit in Kafka is called a *message*, messages are stored in *topics*, which are aggregations of messages of a certain type. Topics are used later to retrieve messages according to their type. The Application Programming Interface (API) provided to applications is very similar to a publish subscribe system, clients can publish messages to topics and can later subscribe to them to read the messages.

When it comes to the storage of messages, they are stored in *brokers*. Inside each broker they are stored per topic, as a log on disk. In order to sustain high volumes of messages, topics can be partitioned. At the storage level this corresponds to the creation of independent logs for the same topic. Having independent logs means that no order exists between them, a limiting factor for some applications. All this behaviour makes brokers very similar to log maintainers in Chariots, with the small difference that brokers are responsible for completely independent logs and not ranges of a single one.

Publishing a message in Kafka is very simple, a client randomly chooses one of the partitions of the topic where he wishes to publish a message, finds which broker is holding the partition and sends the message to that broker. The broker receives the message and simply appends it to the corresponding log. Just like in Chariots brokers can assign log positions from the logs they are responsible for without entering in any coordination with other brokers, which makes each broker a sequencer of the logs they hold.

In order to read messages from a given topic, clients have to join what is called in Kafka a *consumer group*. A consumer group is a group of one or more consumers that jointly consume a set of subscribed topics. To avoid locking mechanisms and state management Kafka makes the partition of a topic the smallest unit of parallelism, i.e. at any given time all the messages of a partition are only consumed by a single consumer within a group. Messages from a partition are always consumed sequentially.

As described before, Kafka does not require any coordination for publishing messages, however, consuming messages requires some level of coordination between consumers in a consumer group. For this, Kafka does not rely in a central node, consumers coordinate among themselves in a decentralised fashion. To facilitate the coordination a consensus service called Zookeeper [25] is used for detecting the addition and the removal of brokers and consumers, triggering a re-balance process in each consumer when necessary, maintaining the consumption relationship between partitions and consumer, and keeping track of the consumed offset of each partition.

Even though Kafka has some similarities with Chariots, its design is optimised for delivering messages, which results in a log based storage system that provides applications with a partitioned log with a total order per partition.

2.7.7 Eunomia

Eunomia [20] is an event ordering service that produces an causally ordered log of events.

In the paper, Eunomia is presented as a building block of a key-value store called EunomiaKV, where it is used as an update serialiser. It works in the background serialising all the updates occurring in the store. The result of the serialisation is a causally ordered log of updates that is used to replicate the updates to new geographic locations. In our description of the system we will ignore the geographic replication capabilities of EunomiaKV, instead we will focus on the log building techniques used.

The EunomiaKV architecture is composed by *partitions* and the *Eunomia* service. Partitions are responsible for storing ranges of key-value pairs and serving read and update requests on the them. The Eunomia service receives all the updates that each partition handles and orders them before sending them to new geographic locations.

Read requests handled by the partitions, correspond to simple key-value fetches on the underlying storage layer, returning to the client the value of the key requested and the timestamp it is tagged with. The client uses the timestamp returned by the partition to update its local clock, which is responsible for keeping track of his causal dependencies. Update requests are tagged with timestamps, representing the causal dependencies of both the client and the partition, are applied locally and are sent to Eunomia to be serialised.

At Eunomia, updates received from the partitions are placed in a *non-stable* updates queue. Regularly Eunomia runs a *process stable* routine that finds a set of stable updates in the non-stable updates queue, removes them and orders them in timestamp order, producing a causally ordered log of updates. We should note that the log produced by Eunomia has a significant difference when compared to the log produced by all the systems presented. It is a log composed only by ids, this means that the log does not include the content of the updates. This happens because the only thing that partitions send to Eunomia are the update id and its timestamp. A significant difference that avoids overloading the network.

Allowing partitions to serve read and update requests without entering in synchronous coordination with any other component in the system, is a fundamental characteristic of the Eunomia design. It does not limit the system throughput like sequencer based designs do, because the load is distributed across all the partitions, and the latency of operations is decreased, because partitions can reply to client without having to wait from a response from other component. On the other hand, this design introduces one main trade-off when compared to sequencer based designs, an increase of the visibility

latencies of the log, i.e. the log takes longer to be available for reading.

2.7.8 FLACS

FLACS [4], or Flexible Location-Aware Consistency, is a method proposed to manage transactions on replicated data while providing low latency and scalability.

In the paper, FLACS is proposed based on the premise that in cloud systems, transactions accessing the same data (same partition) usually originate in the same area, and as such FLACS organises partitions in a tree structure that allows transactions to be validated and committed as close to their origin as possible. The tree structure allows transactions to be ordered incrementally, which in turn allows them to be validated without having a full view of concurrent transactions.

The FLACS system is presented as a system composed by several processes organised in a tree structure. Each process is responsible for: replicating a partition of data, ordering transactions and validating transactions. These processes interact with each other to validate and commit transactions. According to the FLACS protocol transactions are executed by a set of these processes in the four steps that follow.

1. Execution: a transaction starts by executing all its operations at a processes known by transaction's initiator. This process executes operations but does not commit them. Operations are only committed once the validation result is known.

2. Ordering: the transaction is ordered against other conflicting transactions by a set of processes known by transaction observers. The transaction observers are determined according to the write set of the transaction, and the idea is to choose observers that are near the client. Each observer keeps a strict local order of the update transactions seen so far, in which any two transactions that might be in conflict are ordered against each other. To order a new transaction, each observer inserts the new transaction in his local order and propagates that order to its parent process. The idea then is to combine the local order of every observer of a given transaction and have an order in which the transaction is ordered against all other conflicting transactions. Eventually, all transactions will be totally ordered at the root of the hierarchy, but the validation of a transaction may take place as soon as a transaction is ordered against all other conflicting transactions.

3. Validation: once the transaction is ordered against all the conflicting transactions, it is validated. The validating process is the lowest process in the tree which its decedents include at least one observer of each item read by the transaction and include all the observers of all the item written by the transaction. Then it informs all the processes replicating the data written by the transaction if the validation was successful or broadcasts an abort message to all the participating processes.

4. Commit: finally, if the transaction was validated with success the updates of the transaction are committed at the initiator process and applied at all the processes replicating the data written by the transaction (in the same order they were seen by the validator).

The key idea behind this system is that the higher the locality of a transaction, this is, the lower the number of observers involved in the process of validating a transaction is, the lower the latency of committing a transaction will be, because the validation process will be in a lower level of the tree. On the other hand, transactions that have lower locality will take longer to be committed but are guaranteed to be consistent with all the other. The tree structure also allows an high level of parallelism, because transactions that involve completely independent sets of observers can be validated and committed in parallel without interfering with each other.

Summary

In this chapter we have introduced the main challenges that emerge when designing and implementing distributed transactional stores and we have surveyed several systems that implement different techniques to overcome those challenges.

3

Hierarchical Architecture for Deferred Validation of Transactions

Contents

This chapter describes the main contribution of this work. It starts in Section 3.1 by describing the proposed architecture. Section 3.2 describes how the proposed architecture integrates with existing storage systems and how transactions are executed, validated and committed. Section 3.3 explains how the timestamps used to order transactions are generated. Section 3.4 explains how message batching works and how it is applied to the validation nodes. Section 3.5 does a theoretical analysis of the network usage of the proposed architecture and compares it with the network usage of 2PC. Finally, Section 3.6 ends with a detailed description of how this architecture was implemented over Riak KV.

3.1 Architecture

Figure 3.1 illustrates the architecture we are proposing.

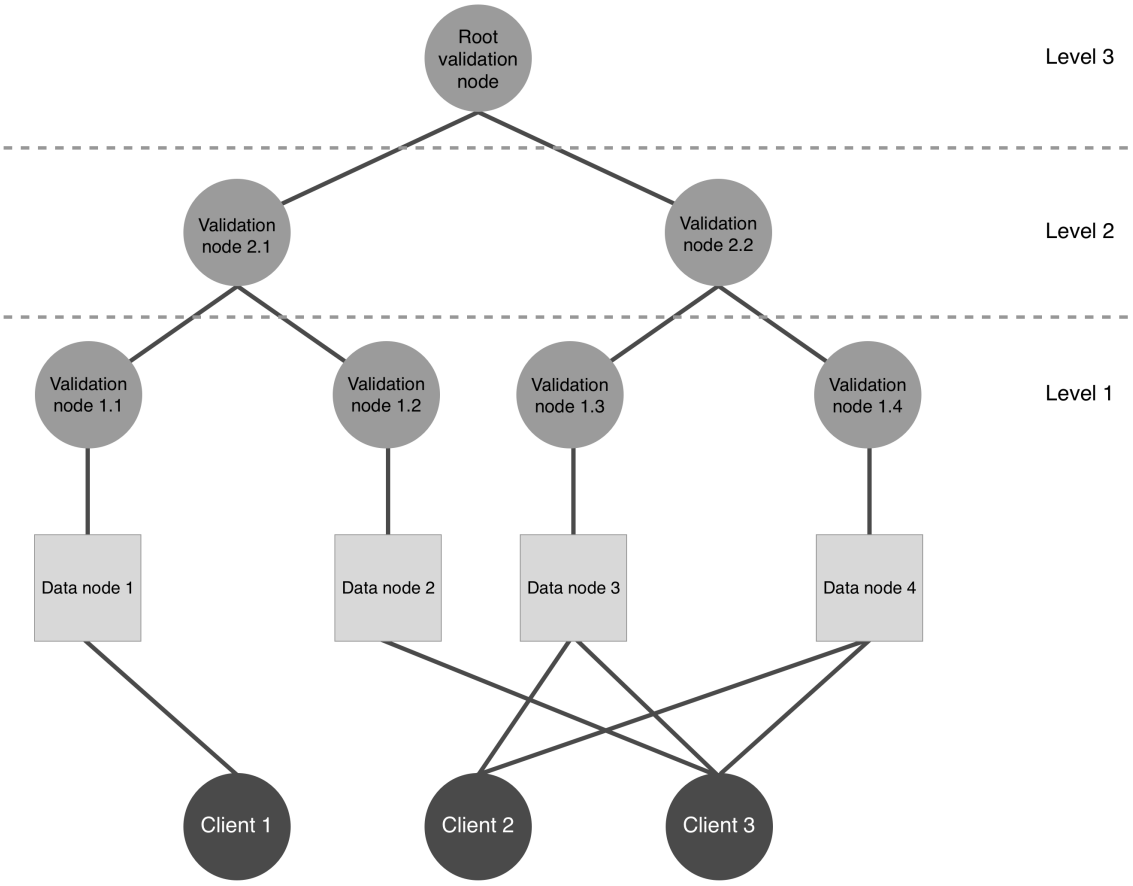

Figure 3.1: Proposed architecture. Key-value pairs are stored in the data nodes. Each leaf validation node is co-located with the corresponding data node.

The proposed architecture is intended to work over a distributed key-value storage system. It as-

sumes a key-value storage system that is partitioned over a set of data nodes. These data nodes are complemented by a set of validation nodes. The validation nodes are organised in a tree hierarchy in which there is a root node with two or more children. Each children node also has two or more children. This organisation is repeated until the leaf nodes. Each leaf node is co-located with one of the data nodes from the key-value store.

This architecture is based on the premised that network communication in distributed systems is often the bottleneck to be able to scale and achieve better performance. With this in mind our objective with this work has always been to try to minimise the amount of network Input/Output (I/O) used in the process of validating and committing transactions.

Today's distributed databases that use optimistic concurrency control mechanisms spend a considerable part of their network I/O in validating and committing transactions. The standard protocol used for this purpose is 2PC, and as described in Section 2.3.1, 2PC is an expensive protocol in terms of network I/O.

One of the common ways to optimise network usage is to use message batching. However, in order for batching to be possible, network communication has to have some sort of structure. 2PC is a protocol that uses a non-structured communication model, where each client communicates directly with the partitions they want, independently of other clients. This communication model makes it very difficult, or impossible, to use batching. The architecture proposed in this section on the other hand, has a very structured communication model, that in turn allows the use of batching.

The architecture we are proposing does not enforce a specific concurrency control mechanism. We have studied two variants of it, one that uses a lock based concurrency control mechanism and one that uses a timestamp based concurrency control mechanism. Section 3.2 describes how this architecture integrates with existing key-value storage systems to execute transactions, in this description we will highlight the differences between the two concurrency control mechanisms when appropriate.

The description of the architecture that follows in the next sections, will assume a serialisable isolation level. We should note that no replication and no fault tolerance mechanisms were considered, as we believe these are subjects mostly orthogonal to this work.

3.2 Execution of Transactions

This section will give a detailed description of how the proposed architecture integrates with existing key value stores. It describes in detail each step involved in the execution of transactions.

3.2.1 Starting a Transaction

A client initiates a transaction by issuing a *begin_transaction* call. This call is local and does not involve communication with data nodes. As a result of this call, a transactional context is created on the client proxy.

3.2.2 Reading Objects

Once in the context of a transaction clients interact directly with data nodes to read the objects they want, as they would normally do with a key-value store. But because they are in the context of a transactions there are some differences.

Our architecture assumes a key-value storage system that has support to store multiple versions per object. Each version is identified by a logical timestamp that corresponds to the logical time when the transaction that wrote that version was committed. In order to read an object in this setting the correct version must be chosen.

Reads done in the context of a transaction are done over a global snapshot of the whole system. This global snapshot is identified by a logical timestamp that is set on the first read the client does. So, in the context of a transaction when a client reads an object for the first time and no snapshot is set, he reads the latest version of that object and sets the transaction snapshot equal to that object's version. If the transaction does not do any reads the transaction snapshot is equal to the maximum snapshot that the client has ever seen.

Reads that follow return the object version that is consistent with the transaction snapshot, this is, the biggest version of an object which is smaller or equal than the transaction snapshot. If this version is marked as tentative (more on tentative versions in Section 3.2.4), the client blocks until that version becomes final or is discarded. On the other hand if the latest version of the object that the client wants to read is bigger than the transaction snapshot, the transaction can abort immediately, because this means that the transaction is not serialisable. There is a special case that occurs when a transaction is marked as read-only: in this case the transaction does not need to be aborted and can proceed, being serialised in the past.

Data nodes are responsible to return a version that is compatible with the transaction snapshot to the clients and to notify the client if the transaction needs to be aborted (if a conflict is detected). Clients contact data nodes directly to read an object and send the transaction snapshot along, the nodes return the corresponding version according to the description above. Reads of the same object that happen more than once in the same transaction take advantage of a local client cache, avoiding contacting the data nodes. Also, if a transaction attempts to read an item that has been written by that transaction, the value is returned from the local cache.

3.2.3 Writing Objects

All writes in the context of a transaction are cached locally until the transaction is ready to commit. This means that any write done while in a transaction, is not communicated to the data nodes until the transaction is ready to commit. Only then the clients send the writes to the data nodes.

3.2.4 Committing a Transaction

Committing a transaction is done by sending a commit request to every data node that was involved in the transaction. This is the set of data nodes that result from the intersection of data nodes that host all the objects read and written during the transaction. The commit request sent to each one of these data nodes is specific to each one, it includes the keys of the objects read on that node and the new objects that were written on that node during the transaction.

Upon receiving a commit request, data nodes integrate with the local validation node to validate the transaction. We have experimented with different strategies to implement transaction validation, that use slightly different concurrency control mechanisms and that use different communication patterns.

Regardless of the concurrency control mechanism used, each data node on receiving a commit request saves the new objects written by that transaction as tentative, with a tentative version equal to the transaction snapshot plus one and sends the commit request information to the local validation node in order for the transaction to be validated and committed.

The tentative versions of the objects are kept until the transaction is fully validated. If the validation results shows that the transaction can be committed, the tentative versions are committed with a version equal to the transaction commit timestamp, otherwise the tentative versions are discarded.

The client remains blocked until he receives the validation result from one of the validation nodes.

3.2.5 Validating a Transaction

The way a transaction is validated by the validation nodes depends on whether it accesses a single data node (local transaction) or several data nodes (distributed transaction).

3.2.5.A Local Transactions

A transaction that only accesses a single data node is considered a local transaction. Local transactions, as the name implies, are local to a single node, which allows them to be validated locally with no coordination with other nodes. The validation is done by the local validation node that is co-located with the data node.

Regardless of the concurrency control mechanism used (lock based or timestamp based) the validation process starts with the validation node assigning a *commit timestamp* to the transaction. This

commit timestamp is generated the way it is described in Section 3.3. Then, for every object read and written by the transaction, the validation node verifies if there was an update to that object between the transaction snapshot and the commit timestamp. This is done by comparing the latest version of an object with the transaction snapshot, if the latest version of the object is bigger than the transaction snapshot there was a new update, otherwise there was not.

If for every object no new updates have happened since the transaction snapshot, the serilisability rules are respected and the transaction can be committed, otherwise it must be aborted. When the validation is finished the validation node informs both the client and the local data node of the result.

If the system is using the lock based concurrency control mechanism, locks for all objects read and written by the transaction must be acquired before starting the validation process. If a lock conflict is detected it means that a concurrent transaction is already trying to commit a new version of that object, so the validation node automatically releases all locks acquired (if any) and sends an unsuccessful validation result to the data node. If locks are acquired successfully the validation process is started as described before.

3.2.5.B Distributed Transactions

A transaction that accesses multiple data nodes is said to be a distributed transaction. As these transactions run across several data nodes they can not be validated locally. Instead each leaf validation node partially validates the transaction and sends the partial validation to the validation node above him in the hierarchy. This process repeats itself until a validation node is able to do a full validation of the transaction.

Depending on the concurrency control mechanism used (lock based or timestamp based) the way distributed transactions are validated is significantly different, so we will present them separately.

Even though they are different, the validation process starts in an identical manner for both. Like a local transaction, the validation process of distributed transactions starts with the data nodes sending the transaction to the local validation nodes, that do exactly what is described in the previous section, with one exception, instead of sending the validation result directly to the client and the data nodes, the validation nodes send the transaction information, its commit timestamp and its partial validation result to the validation node above in the hierarchy.

A – Using The Lock Based Concurrency Control Mechanism

In the current version, when using the lock based concurrency control mechanism, the hierarchy of validation nodes is composed only by two levels, the first with several leaf nodes and the second by a single root node. As such, when leaf nodes propagate information up in the hierarchy they are sending

it all to the root node, that waits for the validation results from its children.

When it has received all the validation results corresponding to one transaction, it calculates a new commit timestamp and a final validation result. The final commit timestamp corresponds to the maximum commit timestamp received from its children, and the final validation result is successful if all its children successfully validated the transaction, and unsuccessful if at least one children did not validate the transaction with success. The final commit timestamp and validation result is then communicated to the data nodes and the client.

Data nodes on receiving the final validation result of a transaction, commit or discard the tentative versions, depending on whether the result was successful or not, and release the locks acquired when the commit was requested. The local validation node clock, used to generate timestamps, is moved forward if it is behind the transaction commit timestamp.

B – Using The Timestamp Based Concurrency Control Mechanism

When using the timestamp based concurrency control mechanism the hierarchy of validation nodes may have more than two levels. In this case, a validation node that is not a leaf node waits for the validation results from its children.

When it has received all the validation results corresponding to one transaction it calculates a new commit timestamp, which corresponds to the maximum commit timestamp received from its children, and validates the transaction again with this new commit timestamp. If the decedents of this validation node cover all the data nodes involved in the transaction, it can commit the transaction by sending the validation result to both the client and the data nodes directly (does not need to go through all the nodes in the hierarchy). Otherwise it propagates the new commit timestamp and new validation result to the validation node above him, like the leaf nodes did. If in this process, a violation of the serialisability rules is found the client and the data nodes can be informed immediately.

As the number of data nodes touched by one transaction increases, the higher in the tree will be the validation node capable of doing a final transaction validation. Ultimately, the transaction will be validated by the root validation node. Crucial to this architecture is the hypothesis that the majority of transactions will have an high locality degree, so that the number of transactions that has to be validated in higher levels of the tree is small.

We should note that when validation nodes validate distributed transactions they do not know the final validation result, because this is only computed by the validation above him in the hierarchy. Nonetheless, in order not to block transaction validation, the validation nodes assume that transactions that were successfully validated locally will be successfully validated in the levels above. In practice this means that if a transaction aborts in an higher level validation node, other concurrent transactions might be

aborted as well, when they could have been committed. This behaviour is conservative and, as such, it will never violate correctness; it only impacts the system performance by increasing the abort rate of transaction. Nevertheless, since we assume that distributed transactions are a small fraction of the overall transactions executed in the system, we believe that the impact of this decision should be minimal.

3.3 Timestamp Generation

This section explains how the logical timestamps used for ordering transactions relatively to one another are generated. The way they are generated depends on which concurrency control mechanism is used, so we present them separately in the next two sections.

3.3.1 Lock Based Concurrency Control

When using the lock based concurrency control mechanism, validation nodes generate commit timestamps by finding the smallest timestamp that is bigger than their local clock and bigger than the transaction snapshot. The local clock of a validation node tracks the biggest timestamp ever generated by that node.

Crucial to the correctness of this generation is moving the local clock forward every time a node receives the final commit timestamp of a transaction from a node above him in the tree.

3.3.2 Timestamp Based Concurrency Control

When using the timestamp based concurrency control mechanism each leaf validation node has a set of pre-assigned timestamps that it can use. These pre-assigned timestamps are distributed among the validation nodes in a round robin manner. Assuming a scenario where there are 8 leaf validation nodes, the first node will be responsible for timestamps 1, 9, 17, etc, the second node will be responsible for 2, 10, 18, etc, and so on.

To generate a tentative commit timestamp, each leaf validation node needs to pick the smallest unused timestamp (from the subset of timestamps assigned to it) that is larger than its local clock and that is larger than the transaction snapshot. The local clock of a validation node tracks the largest timestamp ever generated by that node. Assuming the same scenario of 8 validation nodes, a transaction with snapshot 3 that is locally validated in the second node, and this node has not yet committed any transaction (its local clock is 0), will have a tentative commit timestamp of 10.

3.4 Batching

Crucial to the design of the architecture we propose in this dissertation is the use of message batching, when exchanging messages between nodes in the validation hierarchy. As discussed in the section that follows, message batching can have a large impact on the network cost of validating and committing transactions. This section will explain how message batching works and where it is used in the proposed architecture.

Message batching is a technique used to reduce network and CPU usage. It works by packing multiple application-level messages in one single network-level message, resulting in a reduction of the number of messages that have to be sent through the network and processed by both the sender and the receiver. The key idea is that batching amortises fixed costs such has network protocol headers, interrupt processing, etc, over multiple application-level messages.

Batching is done by caching messages at the sender until a configurable batch size is reached, once it is, the batch is sent to its destination. If the batch does not fill up in a certain time, called the batch timeout, the batch is sent anyway.

In the proposed architecture message batching is used in two distinct message paths: (1) validation node to validation node and (2) validation node to data node. The first corresponds to the messages sent from a lower level validation node to an higher level validation node, in order to validate and commit transactions. The second corresponds to the messages sent by the validation nodes to the data nodes to commit or abort transactions.

3.5 Network Analysis

In order to understand the impact of the structured communication pattern and the usage of batching on the network costs of the proposed architecture, we did a theoretical analysis of its network cost based on the number of communication steps (latency) and total number of messages exchanged (bandwidth) used to commit a transaction.

The network costs of validating and committing a transaction in the proposed architecture can be calculated as follows (note that these values are highly dependent on the structure of the hierarchy and the nodes involved in the transaction, the following formulas capture the worst case cost):

$$H = level\ of\ the\ validation\ node\ in\ the\ hierarchy\ that\ validates\ and\ commits\ the\ transaction \quad (3.1)$$

$$B = size\ of\ the\ batch\ used \quad (3.2)$$

$$communication\ steps = \begin{cases} 2 & N = 1 \\ 2 + \frac{H}{B} & N > 1 \end{cases} \tag{3.3}$$

$$total\ messages = \begin{cases} 2 & N = 1 \\ N + \frac{(H \times N) + N}{B} + 1 & N > 1 \end{cases} \tag{3.4}$$

If one data node is involved in a transaction, the protocol requires 1 message from the client to the data node and 1 message from the data node to the client, to commit it. As such, it takes 2 communication steps and uses 2 network messages in total.

On the other hand, if a transaction involves more than one data node, the protocol require 2 communication steps (1 message from the client to the data node and 1 message from the validation node to the client); plus H messages from a lower level validation node to an higher level validation node until it reaches the validation node that validates and commits the transaction, divided by B because messages are batched. And requires a total of N messages from the clients to the data nodes; plus H messages from a lower level validation node to an higher level validation node until it reaches the validation node that validates and commits the transaction, times N because these messages are sent from all the data nodes involved, all divided by B because messages are batched; plus N messages from the validation node that validates and commits the transaction to each data node involved, divided by B because messages are batched; plus 1 message from the validation node that validates and commits the transaction to the client.

3.5.1 Example

Considering a concrete example with 8 data nodes, a hierarchy of validation nodes with 3 levels and 100 transactions that access 4 data nodes and are validated by a validation node in the first level.

The network cost of 2PC is:

$$communication\ steps\ per\ transaction = 4 \tag{3.5}$$

$$total\ messages\ per\ transaction = 4 \times 4 = 16\ messages \tag{3.6}$$

$$total\ communication\ steps = 4 \times 100 = 400\ messages \tag{3.7}$$

$$total\ messages = 16 \times 100 = 1600\ messages \tag{3.8}$$

The network cost of the proposed architecture without any batching is:

$$communication\ steps\ per\ transaction = 2 + \frac{1}{1} = 3 \tag{3.9}$$

$$total\ messages\ per\ transaction = 4 + \frac{(1 \times 4) + 4}{1} + 1 = 13\ messages \tag{3.10}$$

$$total\ communication\ steps = 3 \times 100 = 300\ messages \tag{3.11}$$

$$total\ messages = 13 \times 100 = 1300\ messages \tag{3.12}$$

The network cost of the proposed architecture with a batching size of 50 is:

$$communications\ steps\ per\ transaction = 2 + \frac{1}{50} \approx 2.02\ messages \tag{3.13}$$

$$total\ messages\ per\ transaction = 4 + \frac{(1 \times 4) + 4}{50} + 1 \approx 5.16\ messages \tag{3.14}$$

$$communications\ steps = 2.02 \times 100 = 202\ messages \tag{3.15}$$

$$total\ messages = 5.16 \times 100 = 516\ messages \tag{3.16}$$

The results above show that the proposed architectures without any batching reduces the communication steps by 25% and reduces the total number of messages by 18%. With batching the reduction goes even further, reducing the total communication steps by 49% and reducing the total number of messages by 67%.

3.6 Implementation

In order to evaluate the proposed architecture in a realistic way we implemented it over a key-value storage system used in the industry [26], Riak KV [5]. We have implemented two versions of this architecture, one that uses lock based concurrency control and another that uses timestamp based concurrency control, as described in Section 3.2. As a base of comparison, we implemented another transactional system over Riak KV, using a lock based concurrency control mechanism and 2PC. Finally, we implemented a custom version of Basho Bench [27] that allowed us to measure the performance of these prototypes.

3.6.1 Riak KV

Riak KV is a distributed key-value storage system built with the objective of providing high availability to its clients. Riak's architecture, similar to the one proposed by Amazon Dynamo [28], is based on the usage of virtual nodes (*vnodes*) that are organised in a logic ring and are deployed in a set (usually smaller) of physical machines.

Each virtual node is responsible for storing independent fragments of data according to an hash function. Clients interact directly with the virtual nodes, using a client library, to execute reads and writes

on the data.

In order to provide high availability, Riak by default replicates data across several virtual nodes, so that if a virtual node goes down, the others can still access the data. By default Riak provides eventual consistency across replicas of the same data, but it also allows quorum reads and writes to provide strong consistency.

3.6.1.A Changes and Extensions

In order to implement the proposed architecture over Riak KV, we had to introduce several changes and add several new modules to its code base. Riak KV's code base is very extensive, and is distributed across several code repositories. The one which we had to interact with to introduce changes was *riak_kv* [29], which has close to 50k lines of code.

The first set of changes we have introduced in the code base were to Riak's base configuration. With the objective of simplifying the system as much as possible we disable data replication and set the number of virtual nodes equal to the number of physical machines. As the focus of our work is on concurrency control and commit mechanisms, having data replication and several data partitions per physical node would introduce a lot of complexity and noise, so we disable them.

The second set of changes we have introduced were in the virtual node module. First we added an interface to read objects in the context of a transaction. This interface is responsible for receiving read requests accompanied by a transaction snapshot and returning the correct version of the object, that the client is requesting, according to the transaction snapshot. Then an interface to validate and commit a transaction was also added. This is the interface that receives commit requests from clients and redirects them to the local validation node. Finally, an interface to commit or discard the tentative writes of transactions was added. This interface is used by the validation nodes to commit or abort a transaction once the validation result is known.

When it comes to extensions to the system, the first one was the introduction of what we call the *transactional client*. The transactional client as the name implies is a client capable of executing transactions. This client was developed as a separate client from the original Riak client so that backward compatibility with older versions of Riak were kept, this is, the original Riak client keeps working with original virtual node interfaces to read and write objects and the new one works with the newly introduced interfaces to execute transactions. The transactional client is also responsible for keeping state about the transactions that he executed or is executing.

The second extension was the addition of the transactions validation module. This is the module that has all the validation logic described in Section 3.2 and is deployed in independent validation nodes. For the two distinct versions that we developed this is the module that has the majority of the differences.

Finally the last addition to the code base was the *message batching* module. This is the module

that is responsible for batching messages and sending them when a batch is full or the batch timeout was reached. Both the batch size and the batch timeout parameters were added to the Riak KV's configuration.

Pseudocode of both versions of the proposed architecture are available in Appendix A and Appendix B.

3.6.1.B Base of Comparison

In order to evaluate the performance of the proposed architecture we implemented another transactional system over Riak KV that we used as a base of comparison. This system uses a lock based concurrency control mechanism and 2PC to validate and commit transactions. The way it works is similar to the way the proposed architecture works when it uses the lock based concurrency control mechanism, the main difference is that instead of using the hierarchy to validate and commit the transactions it uses 2PC.

Committing a distributed transaction starts with the client sending prepare messages to all the data nodes involved in the transaction. On receiving a prepare message each data node assumes the responsibility of a leaf validation node and locally validates the transaction using the same method as the proposed architecture with the lock based concurrency control mechanism. The main difference to the proposed architecture is at this point, instead of sending the validation result, to the validation node above him in the hierarchy it sends it to the client, that acts as the transaction coordinator.

The client receives validation results from all the data nodes and compiles them into a final validation result, like the root validation node would do. Then, according to the final validation result, a commit or an abort message is sent back to the data nodes. Data nodes, on receiving the final validation result, do the exact same thing as they would do in the proposed architecture. Local transactions are validated and committed in the exact same way as in the proposed architecture.

When it comes to the implementation of this version, because it is very similar to the proposed architecture with the lock based concurrency control mechanism a lot of code was reused from that version. Although, two major changes were made to it. First the transactional client had to be changed to incorporate the logic of the root validation node, of receiving several validation results and producing a final validation result. And secondly the virtual nodes module had to be changed to incorporate the leaf validation logic changed to send the validation result to the client and not to the validation node above him in the hierarchy.

Pseudocode of this version is available in Appendix C.

3.6.2 Basho Bench

Basho Bench is a load generator and benchmarking tool created by the authors of Riak KV to conduct performance tests against Riak KV itself. Even though it was created to benchmark Riak KV, its modular

design allows it to benchmark any other system. All the user needs to do is to develop a custom driver for that system.

In order to be able to use Basho Bench to benchmark the prototypes of the architecture we propose, we had to introduce several changes to the Basho Bench code base [30], which has close to 8k lines of code.

The first change was the addition of the *transactional client driver*. This is a Basho Bench driver that is able to execute transactions using the newly added transactional client. This driver allowed us to configure the type of transactions it executed (local or distributed) and which operations (get, put, update) each transaction executed.

The second change was the creation of a custom key generation module. Basho Bench, like it does with drivers, allows the user to create custom key generation modules. These modules are the ones responsible for generating keys during a benchmark. The one we have developed provides a very fine control over the keys it generates. In particular, using our generator, we can control the key itself, the node in which the key is stored, and also control if other clients are going to also access that key (and, in this way, control the degree of contention in the workload).

Summary

In this chapter we have introduced a novel architecture to implement distributed key-value stores. The architecture facilitates the use of message batching, to save network and processing resources. We have also discussed the technologies that have been used to build a prototype of the proposed architecture.

4

Evaluation

Contents

This chapter presents the experimental results obtained while evaluating the prototype of the proposed architecture. The main goal of the evaluation is to understand if, and in which scenarios, the proposed architecture is able to achieve better throughput and support larger numbers of clients than existing solutions. Therefore, our evaluation compares the proposed architecture with the standard protocol used to commit distributed transaction, 2PC. We recall that one of the main disadvantages of this protocol is its high network I/O overhead. Therefore, we aim to show that the proposed architecture has lower network I/O overhead and as a result is able to achieve better throughput and support larger numbers of clients. Conversely, a disadvantage of the proposed architecture is increasing the latency of executing a transaction. Thus, we also aim to show that the impact on the latency of executing transactions is not significant.

4.1 Experimental Setup

4.1.1 Hardware Configuration

All the experiments presented in this chapter were executed in Google Cloud [31]. All the virtual machines used were configured to run Ubuntu 14.04. Different hardware configurations were used for nodes executing different roles in the experiment. Each data node, responsible for hosting the Riak KV virtual nodes and the leaf validation nodes, ran in a virtual machine with 2 cores, 7.5GB of memory and 10GB of disk. Non-leaf validation nodes ran in virtual machines with 4 cores, 15GB of memory and 10GB of disk.

In order to benchmark the proposed architecture we used a modified version of Basho Bench [27]. The modifications we have introduced were the minimum necessary to make it work with the new transactional client. A more detailed list of changes is described in Section 3.6.2. Basho Bench was deployed in virtual machines, each with 8 cores, 30GB of memory and 10GB of disk.

4.1.2 Benchmark Configuration

Unless specified, the experiments presented in this chapter have the following characteristics. A total of 800k objects are used. These objects are populated before running the experiment. Keys are generated using an uniform distribution. Each object value is a fixed binary of 1000 bytes. The experiments run for 10 minutes, from which the first and the last minutes are discarded.

4.1.3 Systems Under Test

The evaluation presented in this chapter compares the performance of the proposed architecture in two variations against a system that uses a lock-based concurrency control mechanism and 2PC to commit

the transactions. This system is used as a baseline of comparison. The two variations of the proposed architecture are the one that uses a lock-based concurrency control mechanism and the one that uses a timestamp-based concurrency control mechanism.

Abbreviation	System Description
2PC+Locking	Baseline of comparison
Hierarchy+Locking	The proposed architecture using a lock based concurrency control mechanism
Hierarchy+Timestamps	The proposed architecture using a timestamp based concurrency control mechanism

Table 4.1: Abbreviations used to describe each system and its variations

For the sake of brevity and in order to make it easy to identify each system, we will refer to these systems and their variations as follows: *2PC+Locking* refers to the baseline of comparison, *Hierarchy+Locking* refers to the proposed architecture with a lock based concurrency control mechanism and *Hierarchy+Timestamps* refers to the proposed architecture with a timestamp based concurrency control mechanism. These abbreviations are summarised in Table 4.1.

To make the comparison between these systems as fair as possible all of them were tested in a scenario with 8 data nodes. For the Hierarchy+Locking and the Hierarchy+Timestamps versions, the same hierarchy of validation nodes, composed by 8 leaf nodes and 1 non-leaf validation node, was used.

4.2 Network Load Experiments

As shown in Section 3.5, the proposed architecture is able to reduce the network traffic required to validate and commit a transaction when compared to 2PC. In order to better understand the impact of this reduction on the system performance, we ran multiple experiments with the transaction validation logic disabled. This allowed network communication to be evaluated in isolation. With the transaction validation logic disabled, all transactions were committed successfully.

4.2.1 Throughput and Latency

In order to understand the performance of the proposed architecture we ran multiple experiments with an increasing number of clients. We ran experiments against seven different systems/variants: (1) the 2PC+Locking version as a baseline of comparison, (2) the Hierarchy+Locking version with no batching, (3) the Hierarchy+Locking version with a batch size of 10, (4) the Hierarchy+Locking version with a batch size of 50, (5) the Hierarchy+Timestamps version with no batching, (6) the Hierarchy+Timestamps version with a batch size of 10 and (7) the Hierarchy+Timestamps version with a batch size of 50.

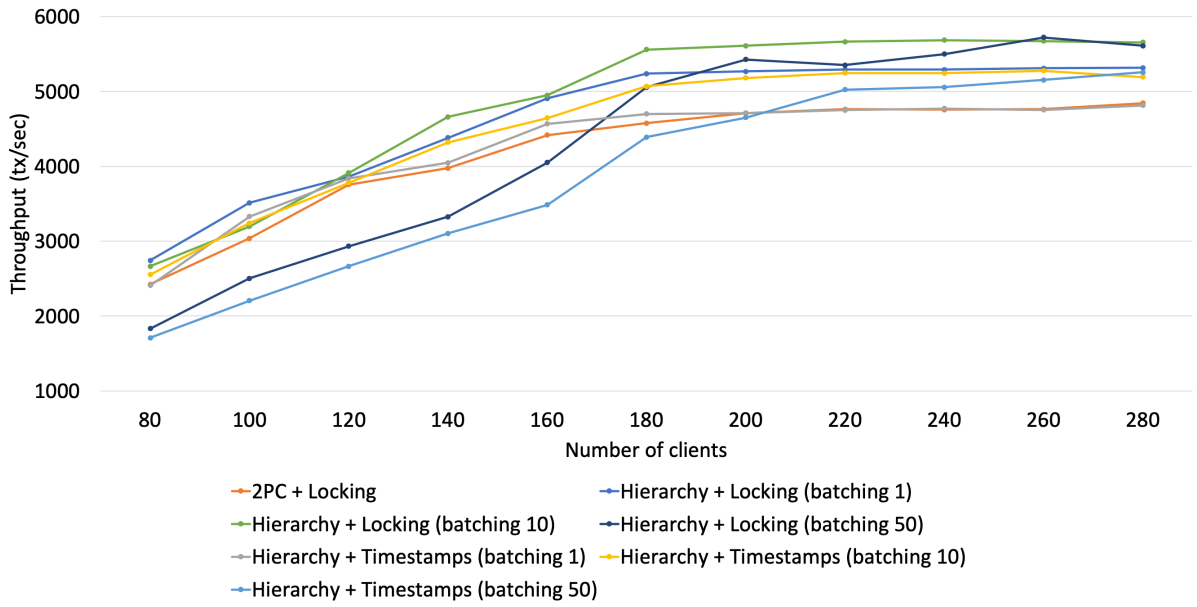

Figure 4.1: Throughput variation as the number of clients increases, with concurrency control logic disabled

The graphs presented in Figure 4.1 and Figure 4.4 show the results of these experiences. The graph in Figure 4.1 shows how the throughput of each system varies as the number of clients increases, and the graph in Figure 4.4 shows how the average latency to execute a transaction varies as the number of clients increases. The experiments presented use a workload with 100% of distributed transactions. Each transaction executes 8 updates operations (read followed by a write on the same object), each in a different data node. Because all the logic except the network logic is disabled all transactions are committed successfully. Due to the density of the graphs in Figure 4.1 and Figure 4.4, Figure 4.2, Figure 4.3, Figure 4.5 and Figure 4.6 contain sub graphs of the first two, highlighting the results obtained for a subset of the systems.

As the results in Figure 4.2 and Figure 4.3 show, the systems that do not use batching or use a batching of 10 reach their saturation point (i.e. the point when the system is processing the maximum number of messages it can per unit of time) when 180 clients are used. While systems that use a batch size of 50 reach their saturation point when 240 clients are used. This is confirmed by the results in Figure 4.5 and Figure 4.6, which show that the latency of executing transactions in systems that do not use batching or use a batching of 10 starts to spike when 180 clients, meaning that the system is not able to process all the messages that are arriving and thus the system takes on average more time per message. Systems that use a batching of 50, have an higher latency from the beginning due to the time it takes to fill the bigger batch, but are able to keep that latency while supporting approximately 30% more clients than the others.

The fact that the systems that use a batching of size 50 can support higher numbers of clients without

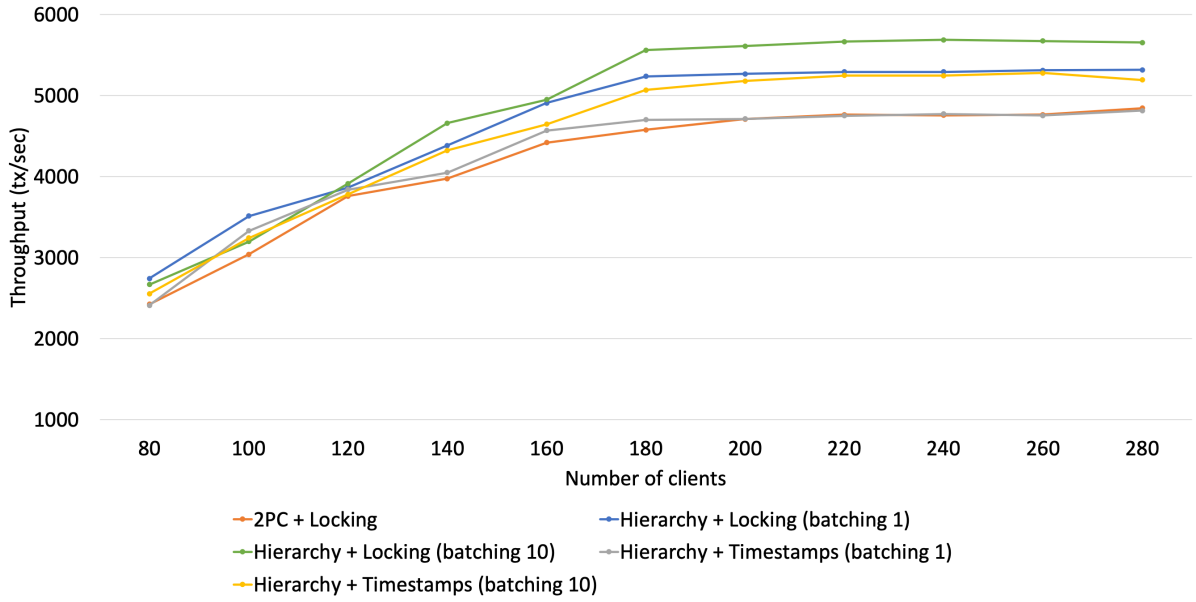

Figure 4.2: Sub graph of Figure 4.1, highlighting the throughput variation as the number of clients increases for systems that use no batching or a batching of 10

saturating, results in an increase in throughput. The results show that the Hierarchy+Timestamps version is able to achieve approximately 15% more throughput than the baseline, while the Hierarchy+Locking version is able to achieve approximately 23% more throughput than the baseline. The difference in throughput between these two systems is due to the concurrency control mechanisms used by each one. The Hierarchy+Timestamps versions uses more network bandwidth, which in other words means that it uses bigger network messages. Bigger messages take more time to process and as such the throughput is lower.

4.2.2 Batching vs Latency Trade-off

As we have stated before, and as the graphs in Figure 4.1 and Figure 4.4 show, batching can have significant impact on the system performance. Message batching reduces the amount of network I/O used by nodes, that in turn results in a network bandwidth and CPU reduction. We could imagine that, if we kept increasing the batch size used, the bigger these reductions would be. However, as the batch size increases, the amount of work required to process a bigger batch and the time it takes to fill up a batch also increases. We can say that there is a trade-off here, between the batch size used and the latency increase we get as a result.

In order to understand the impact the batch size has on the latency observed by the clients, due to the longer validation phase, we ran multiple experiments where we gradually increased the batch size. The results of these experiments are depicted in Figure 4.7. The graph in this figure shows the

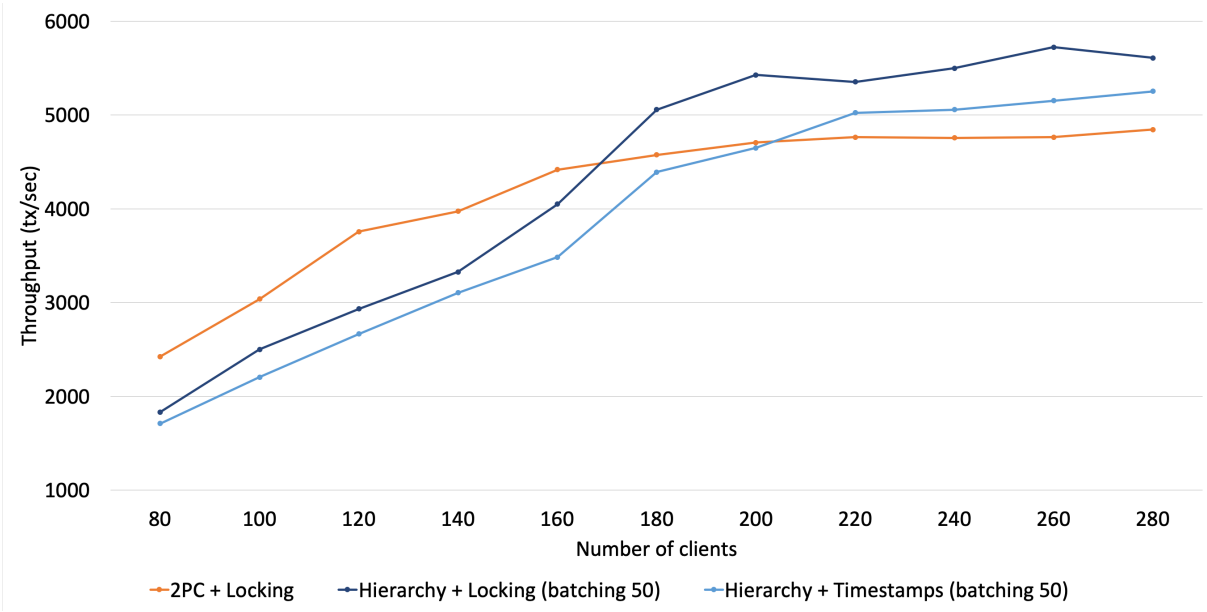

Figure 4.3: Sub graph of Figure 4.1, highlighting the throughput variation as the number of clients increases for systems that use a batching of 50

average latency of executing a transaction in three different systems with different batching sizes. The same workload of 100% of distributed transactions, each executing 8 updates operations in different data nodes was used. For all these experiments we used 100 clients, a number of clients that did not saturate the system.

As results show, the average latency of executing a transaction in the 2PC+Locking version is 33ms, while in the Hierarchy+Locking version and the Hierarchy+Timestamps version without any batching is around 29ms. Increasing the batch size to 10 seems to have a negligible impact on the latency, with the systems presenting a average latency of 31ms. Increasing the batch size further to 25, increases the average latency to a value closer to the latency presented by 2PC+Locking version, 35ms.

When using batching with sizes larger 25, it is possible to start observing a more significant impact on the latency. For a batch size of 50, the average latency of the Hierarchy+Locking version and the Hierarchy+Timestamps version increases to 40ms and 45ms respectively, values that represents an increase of approximately 50% over the base latency of the system without any batching. Using a batch size of 75 the increases the latency approximately 75%, and using a batch size of 100 increases the latency approximately 150%.

4.3 Concurrency Control Experiments

In order to understand if the performance gains shown by the results of Section 4.2.1 were kept if the transaction validation logic was enabled, we ran a very similar set of experiences as the ones presented

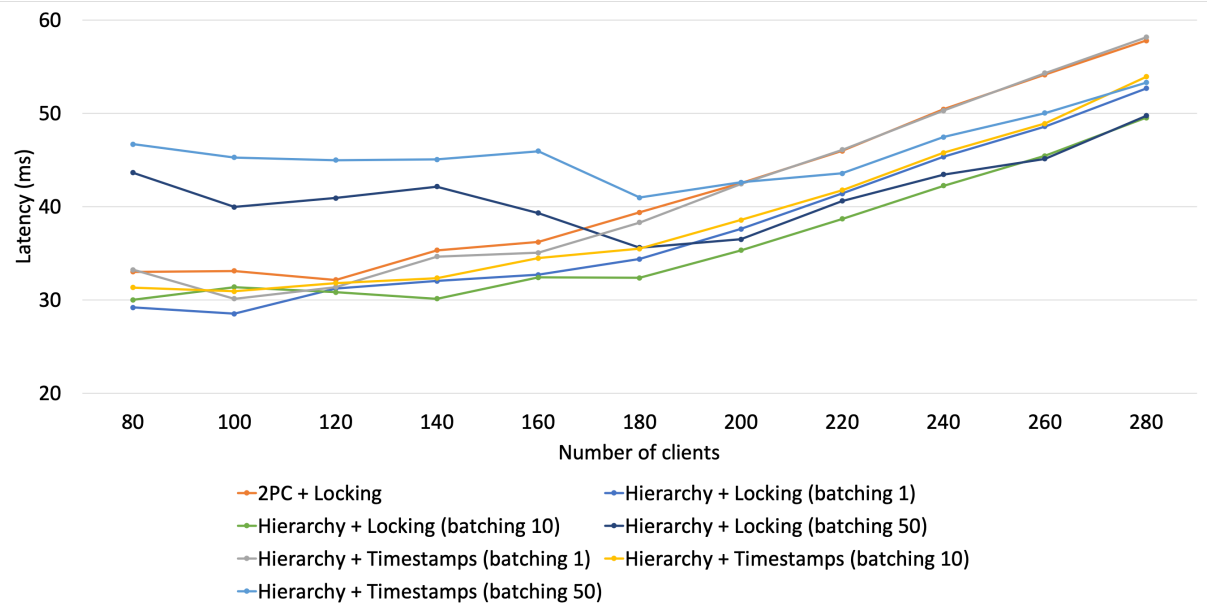

Figure 4.4: Latency variation as the number of clients increases, with concurrency control logic disabled

in that section, but with all the logic enabled.

The graphs presented in Figure 4.8 and Figure 4.9 show the results of these experiences. The graph in Figure 4.8 shows how the throughput of each system varies as the number of clients increases, and the graph in Figure 4.9 shows how the average latency to execute a transaction varies and the number of clients increases. The experiments presented use the same workload as the one used in Section 4.2.1. In this workload the contention between transactions is non existent, so all transactions are committed successfully.

As results show, the performance gains obtained in Section 4.2.1 were not kept when the validation logic is enabled. Although, the results still show some improvement, namely, the solution that uses batching is able to support 25% more clients before saturating. This is visible in Figure 4.8, the 2PC+Locking version saturates when 80 clients are used while the solution with batching saturates when 100 clients are used.

Even though it saturates with a larger number of clients its throughput is not larger than the system without batching. We believe this is justified by the results in Figure 4.9, that shows that the average latency of executing one transaction keeps increasing as the number of clients increases. This happens because transactions are validated serially (one after the other) at each node, so as more clients are added, more transactions need to be validated, and more transactions will be waiting to be validated, so, the average latency per transactions increases. This increase still happened when the validation logic was disabled, even though the results in Section 4.2.1 do not show it clearly. In these experiences it became more clear because the validation logic is enabled and the time it takes to validate a transaction

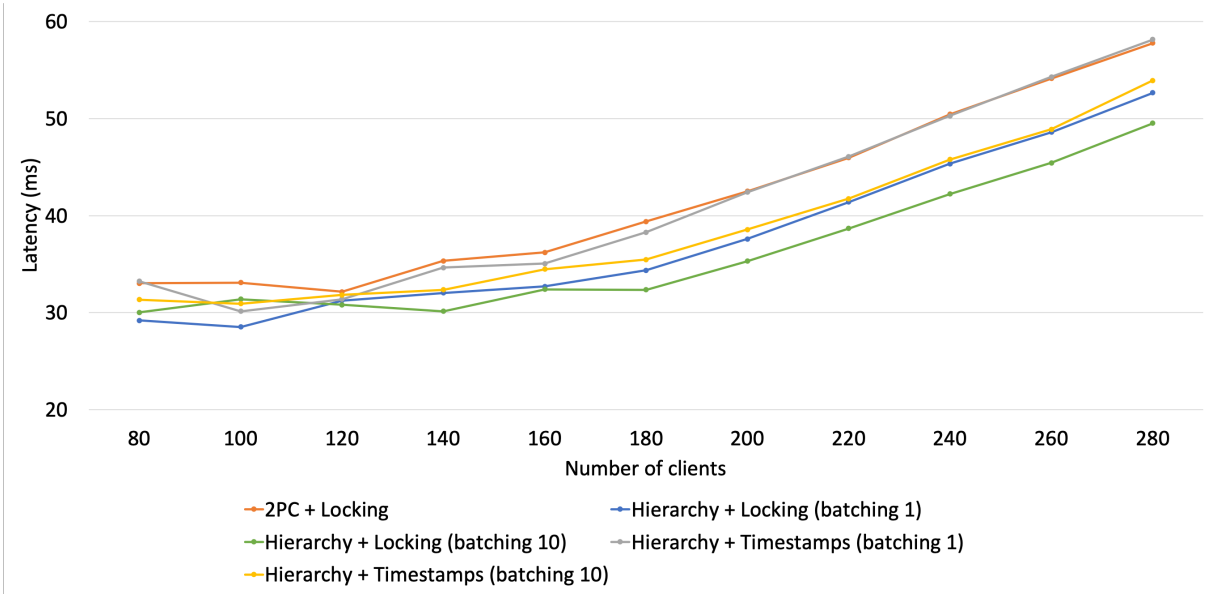

Figure 4.5: Sub graph of Figure 4.4, highlighting the latency variation as the number of clients increases for systems that use no batching or a batching of 10

is much higher.

As these experiments run in a closed loop model the latency of operations directly affects the throughput of the system, if the latency goes down the throughput increases and vice versa. When batching is used, it is expected that the latency of operations will increase due to the time it takes o fill up the batch, but at the same time less network I/O and CPU are used because less network messages are sent. To overcome the latency increase introduced by batching the solution is to add more clients. If more clients are added it is expected that the throughput goes up to match the throughput of a system where no batching is used. Eventually the network I/O and CPU savings obtained by using batching are big enough that allow the system to support a big enough number of clients, that the throughput surpasses the base throughput of the system without any batching. This is what the results in Section 4.2.1 show. However, in this case this does not happen, because as more clients are added the average latency of executing a transaction goes up, and as a result the throughput goes down, which is the exact opposite of what is expected by adding more clients. The reason for this is the fact that transactions are processed serially, which results in transactions waiting for other transactions to be validated.

4.3.1 Experiments With Profiling

To validate the conclusions obtained from the results presented in Figure 4.8 and Figure 4.9 we ran an extra set of experiments where we used a profiling tool while varying the number of clients and the batch size. These experiments include three different scenarios: scenario one uses 100 clients and a batch size of 1, scenario two uses 100 clients and a batch size of 50, and scenario 3 uses 200 clients

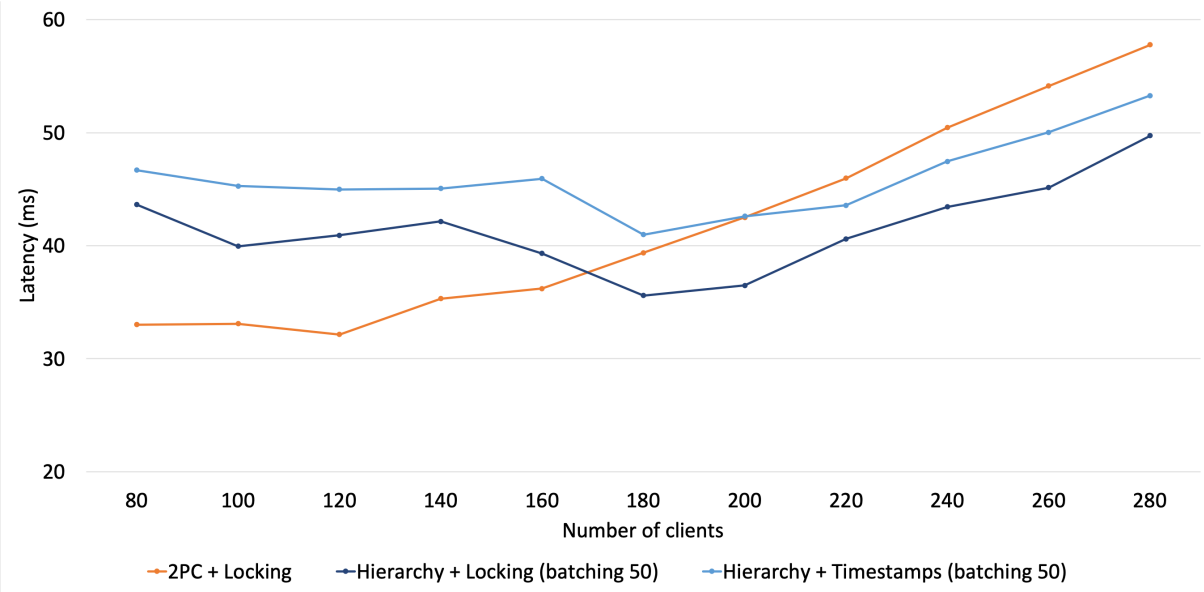

Figure 4.6: Sub graph of Figure 4.4, highlighting the latency variation as the number of clients increases for systems that use a batching of 50

and a batch size of 50. All the experiments use the same workload as the experiments presented in Section 4.3.

The results from these experiments are presented in Table 4.2 and in Table 4.3. Table 4.2 shows the average request processing time for each scenario. Table 4.3 shows the average waiting time between requests for each scenario.

Request description	Scenario 1	Scenario 2	Scenario 3
Get request	0.20	0.24	0.21
Commit request	0.06	0.07	0.07
Validation request	0.56	0.52	0.50

Table 4.2: Average request processing times across experiments. Results are presented in milliseconds.

Scenario	Average time
Scenario 1	0.04
Scenario 2	0.26
Scenario 3	0.10

Table 4.3: Average waiting time between requests. Results are presented in milliseconds.

The results in Table 4.2 show that across experiments the time it takes to process any of the requests is constant. These results are consistent with what was expected. As the number of clients and the batch

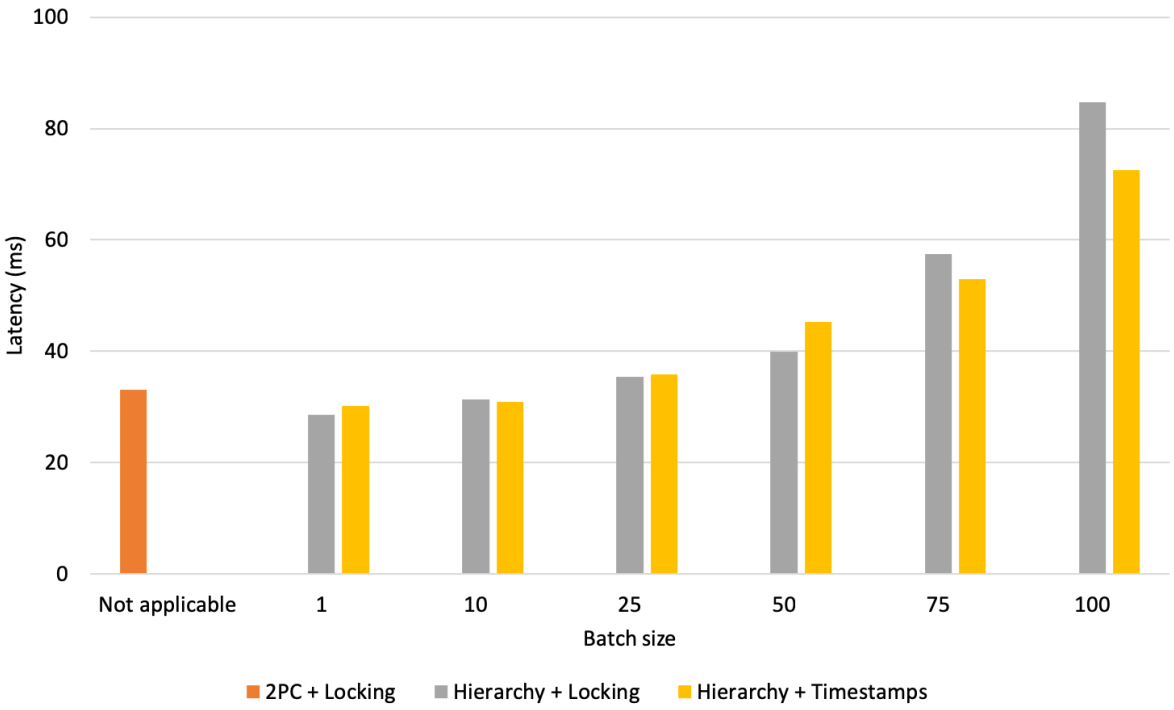

Figure 4.7: Latency variation as the batch size increases

size varies the time it takes for each node to process any of these requests should be the same, and these results show exactly that.

When it comes to the results in Table 4.2, they show the time each node spends waiting for and processing network messages (network I/O). As results shows, the time between each request varies from scenario to scenario. The difference between the first and second scenarios can be justified by the added latency that batching introduces, as clients wait more time for a response, due to batching, they will send less requests per unit of time, and as a consequence the time between requests is longer. From the second to the third scenario, as the number of clients increases, the number of requests sent by them is also going to be bigger, and as such the time between requests decreases because nodes spend less time waiting for messages. Finally the difference in time between the first and the second scenario can be justified by the message sizes, messages in the third scenario are much larger, due to batching, and as a consequence they take more time to be processed.

During these experiments we also measured CPU usage and compared it with the CPU usage of the experiments presented in Section 4.2.1. In these experiments CPU usage did not go above 75%, while in the experiments with the validation logic disabled, CPU usage went up to 100%. The major difference between the two experiments that could have caused this difference is disk I/O, the version where all the validation logic is disabled never accesses the disk, either to read or to write data, while the version evaluated in this section does. This combined with the fact that the server is single threaded results in

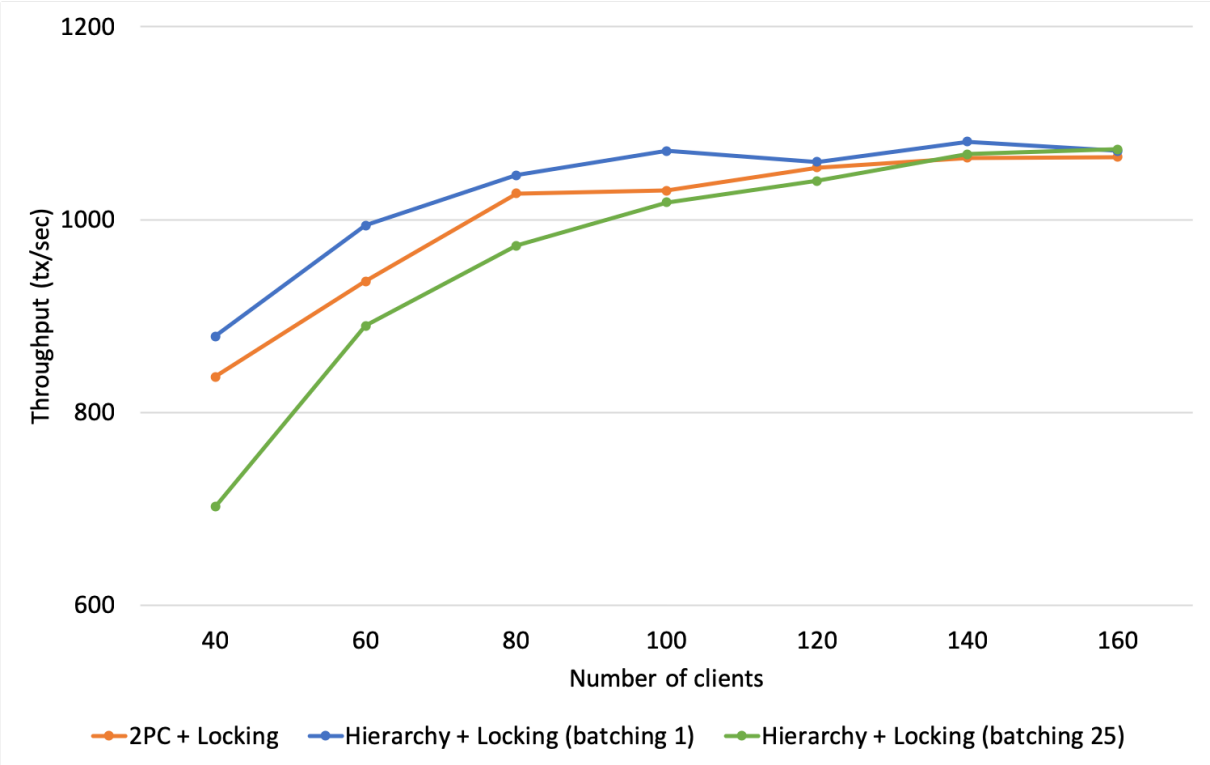

Figure 4.8: Throughput variation as the number of clients increases, with concurrency control logic enabled

this poor utilisation of CPU resources, where the CPU is not used for anything else while it is waiting for disk I/O.

As we are not using the CPU to its maximum potential, the addition of batching will not produce tangible benefits. Instead if we were using the CPU to its maximum potential, the usage of batching could indeed improve the system performance, as it is shown by the results in Section 4.2.1.

4.4 Discussion

The experimental results presented in Section 4.2.1 show that, in a scenario where the main bottleneck lies in the network and in the overheads induced by the exchange of individual messages, the proposed architecture has the potential to support up to 45% more clients and achieve 23% more throughput when compared to a system that uses the 2PC protocol. However, the results in Section 4.3 show that the current implementation of the proposed architecture are not able to achieve the performance improvement that the results in Section 4.2.1 suggest.

Our understanding of these results, as explained in Section 4.3, is that the prototypes of the proposed architecture have a limitation in their implementation that limits the usage of the CPU to its maximum potential. This limitation is in the way requests are processed, which is serially. More precisely, data

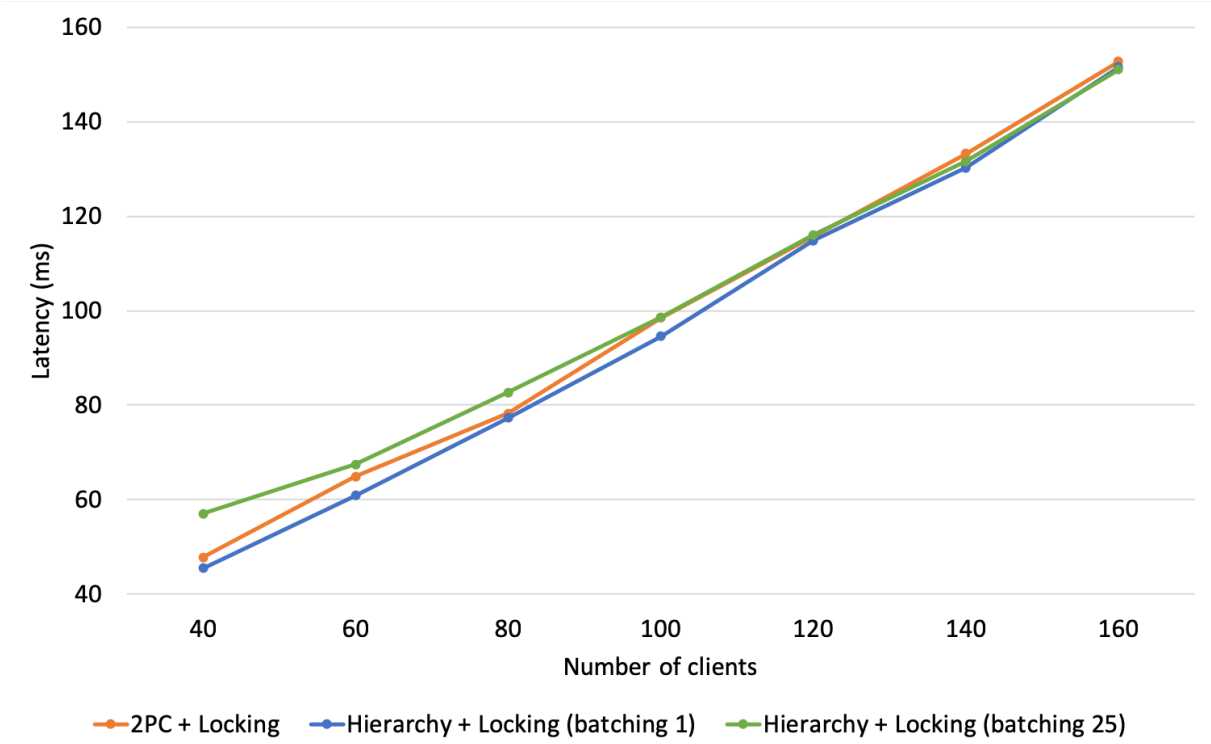

Figure 4.9: Latency variation as the number of clients increases, with concurrency control logic enabled

nodes are single threaded and process every request one at the time. By processing requests serially, as more requests need to be processed the bigger their processing latency will be. The system is implemented this way, because transactions need to be certified in total order, and using a single thread ensures that no re-orderings may be caused by scheduler during the validation procedure. However, not all code that is executed by the data nodes, when committing or aborting a transactions needs to be serialised. We believe that by increasing the degree of concurrency in the data nodes, some of the observed limitations may be eliminated.

When it comes to the batching vs latency trade-off presented in Section 4.2.2, our conclusions are the following. If latency is a priority, using a batch size up to 25 is ok, as the increase in latency is almost negligible. On the other hand, if latency is not a critical requirement, bigger batch sizes can be considered if the increase in latency is on par with the latency expectations.

Summary

In this chapter we have presented the experimental evaluation of our architecture. The results show that the architecture has the potential to offer performance benefits. However, we have not been able to achieve the impressive results we were expecting. The section has also discussed some limitations of

the current implementation that may justify the observed results.

5

Conclusions and Future Work

In this work we presented an hierarchical architecture for deferred validation of transactions. This architecture allows transactions with an high locality degree to be validated and committed concurrently and with low latency, while transactions that have a low locality degree have a slightly higher latency. The architecture also takes advantage of message batching techniques to be able to support higher numbers of clients and achieve higher throughput.

We have performed an extensive experimental evaluation of the proposed architecture. Experimental results show that the communication pattern, used by the proposed architecture to validate and commit transaction, allied with message batching, may allow the system to support up to 45% more clients and achieve 23% more throughput when compared to a system that uses the 2PC communication pattern. However, the results have also unveiled limitations in the implementation of the proposed architecture. These limitations prevent the current prototypes from achieving the performance improvements they are expected to.

As future work, we believe that the direction should be into further testing the proposed architecture, to better understand its benefits and limitations. We believe that the first priority should be to overcome the limitations of the current prototypes. Then, we believe that more testing should be done around different hierarchies, with different levels, and different branching degrees, to understand how these can impact the performance of the system. Finally we believe that the scalability of the system should be evaluated, more specifically, understanding if and when the root validation node, that receives information about all the transactions that execute in the system, can become a bottleneck or not.

Bibliography

[1] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[2] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Feral concurrency control: An empirical investigation of modern application integrity," ser. SIGMOD, 2015.

[3] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983. [Online]. Available: http://doi.acm.org/10.1145/289.291

[4] J. Grov and P. Ölveczky, "Scalable and fully consistent transactions in the cloud through hierarchical validation," in *Data Management in Cloud, Grid and P2P Systems*, A. Hameurlain, W. Rahayu, and D. Taniar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 26–38.

[5] "Riak KV," http://basho.com/products/riak-kv/, accessed 28th September 2018.

[6] "INForum 2018," http://inforum.org.pt/INForum2018, accessed 5th September 2018.

[7] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[8] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, no. 1, pp. 94–162, 1992.

[9] L. Lamport, "The part-time parliament," *ACM TOCS*, vol. 16, no. 2, pp. 133–169, 1998.

[10] B. Lampson and H. E. Sturgis, "Crash recovery in a distributed data storage system," January 1979. [Online]. Available: https://www.microsoft.com/en-us/research/publication/crash-recovery-in-a-distributed-data-storage-system/

[11] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[12] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, “Corfu: A shared log design for flash clusters.” in *NSDI*, 2012, pp. 1–14.

[13] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, “Tango: Distributed data structures over a shared log,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 325–340.

[14] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson *et al.*, “vcorfu: A cloud-scale object store on a shared log.” in *NSDI*, 2017, pp. 35–49.

[15] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, “Megastore: Providing scalable, highly available storage for interactive services.” in *CIDR*, vol. 11, 2011, pp. 223–234.

[16] J. Chang and N. Maxemchuck, “Reliable broadcast protocols,” *ACM, Transactions on Computer Systems*, vol. 2, no. 3, Aug. 1984.

[17] K. Birman and T. Joseph, “Reliable Communication in the Presence of Failures,” *ACM, Transactions on Computer Systems*, vol. 5, no. 1, Feb. 1987.

[18] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang, “Spanner: Google’s globally-distributed database,” in *OSDI*, 2012.

[19] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, “Chariots: A scalable shared log for data management in multi-datacenter cloud environments.” in *EDBT*, 2015, pp. 13–24.

[20] C. Gunawardhana, M. Bravo, and L. Rodrigues, “Unobtrusive deferred update stabilization for efficient geo-replication,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 83–95. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/gunawardhana

[21] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, “Calvin: Fast distributed transactions for partitioned database systems,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’12, Scottsdale (AZ), USA, 2012.

[22] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi, “Replex: A scalable, highly available multi-index data store.” in *USENIX Annual Technical Conference*, 2016, pp. 337–350.

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[24] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8. Boston, MA, USA, 2010, p. 9.

[26] "Riak KV Customers," http://basho.com/about/customers/, accessed 28th September 2018.

[27] "Basho Bench," https://github.com/basho/basho_bench, accessed 3rd October 2018.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[29] "Riak KV Code Repository," https://github.com/basho/riak_kv, accessed 30th September 2018.

[30] "Basho Bench Code Repository," https://github.com/basho/basho_bench/, accessed 30th September 2018.

[31] "Google Cloud," https://cloud.google.com/, accessed 3rd October 2018.

A

Pseudocode of the Proposed Architecture With a Lock Based Concurrency Control Mechanism

This chapter presents a pseudocode version of the proposed architecture using the lock based concurrency control mechanism, like it is described in Chapter 3.

A.1 Client Pseudocode

```
1 # TX is an object that keeps the transaction state
2 #
3 # CLOCK variable that keeps track of the highest timestamp ever seen
4 # by the client
5
6 def begin()
7   TX = Transaction.new()
```

```
 8 end
 9
10 def get(key)
11   partition = partition_for_key(key)
12   object = partition.transactional_get(key, TX.snapshot || CLOCK)
13
14   if object == nil
15     return :not_found
16   else
17     if TX.snapshot == nil
18       TX.snapshot = object.version
19     end
20
21     return object
22   end
23 end
24
25 def abort()
26   TX = nil
27   return :aborted
28 end
29
30 def put(key, value)
31   tentative_version = TX.snapshot || CLOCK
32   object = Object.new(key, value, tentative_version + 1)
33   TX.puts.insert(object)
34 end
35
36 def commit()
37   partition_gets = group_gets_per_partition(TX.gets)
38   partition_puts = group_puts_per_partition(TX.puts)
39   partitions = partition_gets.keys + partition_puts.keys
40
41   for partition in partitions
42     gets = partition_gets.get(partition)
43     puts = partition_puts.get(partition)
44     partition.validate_and_commit_transaction(TX.id,
45                                               TX.snapshot,
46                                               gets,
47                                               puts)
48   end
49 end
50
51 def on_transaction_validation_result(conflicts, commit_timestamp)
52   CLOCK = commit_timestamp
53   TX = nil
54   return conflicts == false
55 end
```

A.2 Partition Pseudocode

```
1  # KV is variable that holds a reference to the key value storage backend
2  #
3  # VALIDATOR is a variable that holds a reference to the local validation
4  # node
5
6  def on_client_transactional_get_request(key, snapshot)
7    tentative_object_versions = KV.get_tentative_versions(key)
8    committed_object_versions = KV.get_committed_versions(key)
9
10   snapshot_consistent_version =
11     select_snapshot_consistent_version(snapshot,
12                                        tentative_object_versions,
13                                        committed_object_versions)
14
15   return snapshot_consistent_version
16 end
17
18 def on_client_validate_and_commit_request(tx_id,
19                                            snapshot,
20                                            gets,
21                                            puts,
22                                            client)
23   KV.store_as_tentative(tx_id, puts)
24   VALIDATOR.leaf_validate(tx_id, snapshot, gets, puts, client)
25 end
26
27 def on_transaction_validation_result(tx_id,
28                                      gets,
29                                      puts,
30                                      conflicts,
31                                      commit_timestamp)
32   if conflicts
33     KV.delete_tentative_versions(tx_id, puts)
34   else
35     KV.commit_tentative_versions(tx_id, puts, commit_timestamp)
36   end
37
38   VALIDATOR.leaf_commit(gets ++ puts, commit_timestamp)
39 end
40
41 # This function assumes that both tentative_object_versions and
42 # committed_object_versions are arrays of object versions sorted
43 # in desceding order
44 def select_snapshot_consistent_version(snapshot,
45                                        tentative_object_versions,
46                                        committed_object_versions)
47   for object_version in tentative_object_versions
48     if object_version.version <= snapshot
49       wait_until_verion_is_committed()
50       return object_version
```

```
51      end
52    end
53
54    for object_version in object_versions
55      if object_version.version <= snapshot
56        return object_version
57      end
58    end
59
60    return nil
61  end
```

A.3 Validation Pseudocode

```
1   # PARENT_VALIDATION_NODE holds a reference to the validation node above
2   # in the hierarchy
3   #
4   # CLOCK is a varaible that holds the biggest timestamp ever used by the
5   # validation node
6   #
7   # RUNNING_TRANSACTIONS is a map of transaction ids to transaction objects
8   # that keep state about the transactions that have not yet been committed
9   #
10  # LATEST_OBJECT_VERSIONS is map used to store the latest object versions
11  # seen
12
13  def leaf_validate(tx_id, snapshot, gets, puts, client)
14    commit_timestamp = generate_commit_timestamp(snapshot)
15
16    locks_acquired = acquire_locks(gets ++ puts)
17
18    if locks_acquired
19      conflicts = check_conficts(gets ++ puts, snapshot)
20    else
21      conflicts = true
22    end
23
24    if enough_information_to_commit
25      commit_transaction(tx.tx_id,
26                         tx.gets,
27                         tx.puts,
28                         tx.client,
29                         tx.conflicts,
30                         tx.commit_timestamp)
31    else
32      PARENT_VALIDATION_NODE.non_leaf_validate_and_commit(tx_id,
33                                                          gets,
34                                                          puts,
35                                                          client,
36                                                          commit_timestamp,
```

```
37                                                          conflicts)
38    end
39
40    CLOCK += 1
41  end
42
43  def leaf_commit(puts, commit_timestamp)
44    unless conflicts
45      update_latest_object_versions(puts, commit_timestamp)
46    end
47
48    release_locks(puts)
49  end
50
51  def non_leaf_validate_and_commit(tx_id,
52                                   gets,
53                                   puts,
54                                   client,
55                                   commit_timestamp,
56                                   conflicts)
57    tx = RUNNING_TRANSACTIONS.get(tx_id)
58    tx.gets += gets
59    tx.puts += puts
60    tx.client = client
61    tx.commit_timestamp = max(tx.commit_timestamp, commit_timestamp)
62    tx.conflicts = tx.conflicts or conflicts
63
64    if enough_information_to_commit
65      commit_transaction(tx.tx_id,
66                         tx.gets,
67                         tx.puts,
68                         tx.client,
69                         tx.conflicts,
70                         tx.commit_timestamp)
71
72      RUNNING_TRANSACTIONS.delete(tx_id)
73    else
74      RUNNING_TRANSACTIONS.put(tx_id, tx)
75    end
76  end
77
78  def generate_commit_timestamp(snapshot)
79    if CLOCK <= snapshot
80      CLOCK = snapshot + 1
81    end
82    return CLOCK
83  end
84
85  def check_conficts(objects, snapshot)
86    for object in objects
87      if LATEST_OBJECT_VERSIONS.get(object.key) > snapshot
88        return true
89      end
```

```
90    end
91    return false
92  end
93
94  def update_latest_object_versions(objects, version)
95    for object in objects
96      LATEST_OBJECT_VERSIONS.put(object.key, version)
97    end
98  end
99
100 def commit_transaction(tx_id,
101                        gets,
102                        puts,
103                        client,
104                        conflicts,
105                        commit_timestamp)
106   client.send_validation_result(conflicts, commit_timestamp)
107
108   partition_gets = group_gets_per_partition(gets)
109   partition_puts = group_puts_per_partition(puts)
110   partitions = partition_gets.keys + partition_puts.keys
111
112   for partition in partitions
113     getss = partition_gets.get(partition)
114     putss = partition_puts.get(partition)
115     partition.send_validation_result(tx_id,
116                                      getss,
117                                      putss,
118                                      conflicts,
119                                      commit_timestamp)
120   end
121 end
```

B

Pseudocode of the Proposed Architecture With a Timestamp Based Concurrency Control Mechanism

This chapter presents a pseudocode version of the proposed architecture using the timestamp based concurrency control mechanism, like it is described in Chapter 3.

B.1 Client Pseudocode

```
1 # TX is an object that keeps the transaction state
2 #
3 # CLOCK variable that keeps track of the highest timestamp ever seen
4 # by the client
5
6 def begin()
7   TX = Transaction.new()
```

```
 8 end
 9
10 def get(key)
11   partition = partition_for_key(key)
12   object = partition.transactional_get(key, TX.snapshot || CLOCK)
13
14   if object == nil
15     return :not_found
16   else
17     if TX.snapshot == nil
18       TX.snapshot = object.version
19     end
20
21     return object
22   end
23 end
24
25 def abort()
26   TX = nil
27   return :aborted
28 end
29
30 def put(key, value)
31   tentative_version = TX.snapshot || CLOCK
32   object = Object.new(key, value, tentative_version + 1)
33   TX.puts.insert(object)
34 end
35
36 def commit()
37   partition_gets = group_gets_per_partition(TX.gets)
38   partition_puts = group_puts_per_partition(TX.puts)
39   partitions = partition_gets.keys + partition_puts.keys
40
41   for partition in partitions
42     gets = partition_gets.get(partition)
43     puts = partition_puts.get(partition)
44     partition.validate_and_commit_transaction(TX.id,
45                                               TX.snapshot,
46                                               gets,
47                                               puts)
48   end
49 end
50
51 def on_transaction_validation_result(conflicts, commit_timestamp)
52   CLOCK = commit_timestamp
53   TX = nil
54   return conflicts == false
55 end
```

B.2 Partition Pseudocode

```
1  # KV is variable that holds a reference to the key value storage backend
2  #
3  # VALIDATOR is a variable that holds a reference to the local validation
4  # node
5
6  def on_client_transactional_get_request(key, snapshot)
7    tentative_object_versions = KV.get_tentative_versions(key)
8    committed_object_versions = KV.get_committed_versions(key)
9
10   snapshot_consistent_version =
11     select_snapshot_consistent_version(snapshot,
12                                        tentative_object_versions,
13                                        committed_object_versions)
14
15   return snapshot_consistent_version
16 end
17
18 def on_client_validate_and_commit_request(tx_id,
19                                            snapshot,
20                                            gets,
21                                            puts,
22                                            client)
23   KV.store_as_tentative(tx_id, puts)
24   VALIDATOR.leaf_validate_and_commit(tx_id, snapshot, gets, puts, client)
25 end
26
27 def on_transaction_validation_result(tx_id,
28                                      puts,
29                                      conflicts,
30                                      commit_timestamp)
31   if conflicts
32     KV.delete_tentative_versions(tx_id, puts)
33   else
34     KV.commit_tentative_versions(tx_id, puts, commit_timestamp)
35   end
36 end
37
38 # This function assumes that both tentative_object_versions and
39 # committed_object_versions are arrays of object versions sorted
40 # in desceding order
41 def select_snapshot_consistent_version(snapshot,
42                                        tentative_object_versions,
43                                        committed_object_versions)
44   for object_version in tentative_object_versions
45     if object_version.version <= snapshot
46       wait_until_verion_is_committed()
47       return object_version
48     end
49   end
50
```

```
51    for object_version in object_versions
52      if object_version.version <= snapshot
53        return object_version
54      end
55    end
56
57    return nil
58  end
```

B.3 Validation Pseudocode

```
1   # PARENT_VALIDATION_NODE holds a reference to the validation node above
2   # in the hierarchy
3   #
4   # NODE_ID hold the node identifier
5   #
6   # CLOCK is a varaible that holds the biggest timestamp ever used by the
7   # validation node
8   #
9   # N_LEAF_COMMITTERS is a constant that holds the number of leaf validation
10  # nodes
11  #
12  # RUNNING_TRANSACTIONS is a map of transaction ids to transaction objects
13  # that keep state about the transactions that have not yet been committed
14  #
15  # N_CHILDS is a constant that holds the number of child nodes
16  #
17  # PENDING_COMMIT is a priority queue of transactions that are waiting to be
18  # committed, transactions are ordered in ascending order by their commit
19  # timestamps
20  #
21  # LATEST_OBJECT_VERSIONS is map used to store the latest object versions
22  # seen
23  #
24  # STABLE_TIMESTAMPS is map of child ids to the bigges timestamp ever
25  # received by that child
26
27  def leaf_validate_and_commit(tx_id, snapshot, gets, puts, client)
28    commit_timestamp = generate_commit_timestamp(snapshot)
29
30    conflicts = check_conficts(gets ++ puts, snapshot)
31
32    unless conflicts
33      update_latest_object_versions(puts, commit_timestamp)
34    end
35
36    if enough_information_to_commit
37      commit_transaction(tx_id, puts, client, conflicts, commit_timestamp)
38    else
39      if conflicts
```

```
40          client.send_validation_result(conflicts, commit_timestamp)
41        end
42
43        PARENT_VALIDATION_NODE.non_leaf_validate_and_commit(tx_id,
44                                                            snapshot,
45                                                            gets,
46                                                            puts,
47                                                            client,
48                                                            commit_timestamp,
49                                                            conflicts,
50                                                            NODE_ID)
51      end
52
53      CLOCK += N_LEAF_COMMITTERS
54    end
55
56    def non_leaf_validate_and_commit(tx_id,
57                                     snapshot,
58                                     gets,
59                                     puts,
60                                     client,
61                                     commit_timestamp,
62                                     conflicts,
63                                     child_id)
64      update_stable_timestamp(child_id, commit_timestamp)
65
66      tx = RUNNING_TRANSACTIONS.get(tx_id)
67      tx.gets += gets
68      tx.puts += puts
69      tx.client = client
70      tx.commit_timestamp = max(tx.commit_timestamp, commit_timestamp)
71      tx.conflicts = tx.conflicts or conflicts
72      tx.n_validations_received += 1
73
74      if tx.n_validations_received == N_CHILDS
75        unless tx.conflicts
76          tx.conflicts = check_conficts(tx.gets ++ tx.puts, tx.snapshot)
77        end
78
79        unless tx.conflicts
80          update_latest_object_versions(tx.puts, tx.commit_timestamp)
81        end
82
83        if enough_information_to_commit
84          if tx.commit_timestamp < calculate_stable_timestamp()
85            commit_transaction(tx.tx_id,
86                               tx.puts,
87                               tx.client,
88                               tx.conflicts,
89                               tx.commit_timestamp)
90          else
91            PENDING_COMMIT.add(tx)
92          end
```

```
93      else
94        if conflicts
95          client.send_validation_result(tx.conflicts, tx.commit_timestamp)
96        end
97
98        PARENT_VALIDATION_NODE.validate_and_commit(tx.tx_id,
99                                                   tx.snapshot,
100                                                  tx.gets,
101                                                  tx.puts,
102                                                  tx.client,
103                                                  tx.commit_timestamp,
104                                                  tx.conflicts,
105                                                  NODE_ID)
106     end
107
108     RUNNING_TRANSACTIONS.delete(tx_id)
109   else
110     RUNNING_TRANSACTIONS.put(tx_id, tx)
111   end
112 end
113
114 def generate_commit_timestamp(snapshot)
115   while CLOCK <= snapshot
116     CLOCK += N_LEAF_COMMITTERS
117   end
118   return CLOCK
119 end
120
121 def check_conficts(objects, snapshot)
122   for object in objects
123     if LATEST_OBJECT_VERSIONS.get(object.key) > snapshot
124       return true
125     end
126   end
127   return false
128 end
129
130 def update_latest_object_versions(objects, version)
131   for object in objects
132     LATEST_OBJECT_VERSIONS.put(object.key, version)
133   end
134 end
135
136 def commit_transaction(tx_id, puts, client, conflicts, commit_timestamp)
137   client.send_validation_result(conflicts, commit_timestamp)
138
139   partition_puts = group_puts_per_partition(puts)
140   for (partition, putss) in partition_puts
141     partition.send_validation_result(tx_id,
142                                      putss,
143                                      conflicts,
144                                      commit_timestamp)
145   end
```

```
146 end
147
148 def update_stable_timestamp(child_id, timestamp)
149   STABLE_TIMESTAMPS.put(child_id, timestamp)
150
151   stable_timestamp = calculate_stable_timestamp()
152
153   while PENDING_COMMIT.peak().commit_timestamp < stable_timestamp
154     tx = PENDING_COMMIT.remove()
155     commit_transaction(tx.tx_id,
156                        tx.puts,
157                        tx.client,
158                        tx.conflicts,
159                        tx.commit_timestamp)
160   end
161 end
162
163 def calculate_stable_timestamp()
164   stable_timestamp = nil
165   for (key, value) in STABLE_TIMESTAMPS
166     if stable_timestamp == nil
167       stable_timestamp = value
168     else
169       stable_timestamp = min(stable_timestamp, value)
170     end
171   end
172   return stable_timestamp
173 end
```

C

Pseudocode of the Transactional System Used as a Base for Comparison

This chapter presents a pseudocode version of the system used as a base of comparison in Chapter 4.

C.1 Client Pseudocode

```
1 # TX is an object that keeps the transaction state
2 #
3 # CLOCK variable that keeps track of the highest timestamp ever seen
4 # by the client
5
6 def begin()
7   TX = Transaction.new()
8 end
```

```
 9
10 def get(key)
11   partition = partition_for_key(key)
12   object = partition.transactional_get(key, TX.snapshot || CLOCK)
13
14   if object == nil
15     return :not_found
16   else
17     if TX.snapshot == nil
18       TX.snapshot = object.version
19     end
20
21     return object
22   end
23 end
24
25 def abort()
26   TX = nil
27   return :aborted
28 end
29
30 def put(key, value)
31   tentative_version = TX.snapshot || CLOCK
32   object = Object.new(key, value, tentative_version + 1)
33   TX.puts.insert(object)
34 end
35
36 def commit()
37   partition_gets = group_gets_per_partition(TX.gets)
38   partition_puts = group_puts_per_partition(TX.puts)
39   partitions = partition_gets.keys + partition_puts.keys
40
41   for partition in partitions
42     gets = partition_gets.get(partition)
43     puts = partition_puts.get(partition)
44
45     partition.prepare(TX.id, TX.snapshot, gets, puts)
46   end
47 end
48
49 def on_prepare_result(conflicts, commit_timestamp)
50   TX.commit_timestamp = max(TX.commit_timestamp, commit_timestamp)
51   TX.conflicts = TX.conflicts or conflicts
52
53   if enough_information_to_commit
54     partition_gets = group_gets_per_partition(TX.gets)
55     partition_puts = group_puts_per_partition(TX.puts)
56     partitions = partition_gets.keys + partition_puts.keys
57
58     for (partition, puts) in partition_puts
59       gets = partition_gets.get(partition)
60       puts = partition_puts.get(partition)
61
```

```
62          partition.commit(TX.id, gets, puts, TX.conflicts, TX.commit_timestamp)
63        end
64      end
65    end
66
67    def on_commit_result()
68      CLOCK = TX.commit_timestamp
69      result = TX.conflicts == false
70      TX = nil
71      return result
72    end
```

C.2 Partition Pseudocode

```
1   # KV is variable that holds a reference to the key value storage backend
2   #
3   # CLOCK is a varaible that holds the biggest timestamp ever used by the
4   # validation node
5   #
6   # LATEST_OBJECT_VERSIONS is map used to store the latest object versions
7   # seen
8
9   def on_client_transactional_get_request(key, snapshot)
10    tentative_object_versions = KV.get_tentative_versions(key)
11    committed_object_versions = KV.get_committed_versions(key)
12
13    snapshot_consistent_version =
14      select_snapshot_consistent_version(snapshot,
15                                         tentative_object_versions,
16                                         committed_object_versions)
17
18    return snapshot_consistent_version
19  end
20
21  def on_prepare_request(tx_id, snapshot, gets, puts, client)
22    commit_timestamp = generate_commit_timestamp(snapshot)
23
24    locks_acquired = acquire_locks(gets ++ puts)
25
26    if locks_acquired
27      conflicts = check_conficts(gets ++ puts, snapshot)
28    else
29      conflicts = true
30    end
31
32    unless conflicts
33      KV.store_as_tentative(tx_id, puts)
34    end
35
36    client.send_prepare_result(conflicts, commit_timestamp)
```

```
37
38    CLOCK += 1
39  end
40
41  def on_commit_request(tx_id,
42                        gets,
43                        puts,
44                        conflicts,
45                        commit_timestamp,
46                        client)
47    if CLOCK < commit_timestamp
48      CLOCK = commit_timestamp + 1
49    end
50
51    if conflicts
52      KV.delete_tentative_versions(tx_id, puts)
53    else
54      KV.commit_tentative_versions(tx_id, puts, commit_timestamp)
55
56      update_latest_object_versions(puts, commit_timestamp)
57    end
58
59    release_locks(puts)
60
61    client.send_commit_result()
62  end
63
64  # This function assumes that both tentative_object_versions and
65  # committed_object_versions are arrays of object versions sorted
66  # in desceding order
67  def select_snapshot_consistent_version(snapshot,
68                                         tentative_object_versions,
69                                         committed_object_versions)
70    for object_version in tentative_object_versions
71      if object_version.version <= snapshot
72        wait_until_verion_is_committed()
73        return object_version
74      end
75    end
76
77    for object_version in object_versions
78      if object_version.version <= snapshot
79        return object_version
80      end
81    end
82
83    return nil
84  end
85
86  def generate_commit_timestamp(snapshot)
87    if CLOCK <= snapshot
88      CLOCK = snapshot + 1
89    end
```

```
90    return CLOCK
91  end
92
93  def check_conficts(objects, snapshot)
94    for object in objects
95      if LATEST_OBJECT_VERSIONS.get(object.key) > snapshot
96        return true
97      end
98    end
99    return false
100 end
101
102 def update_latest_object_versions(objects, version)
103   for object in objects
104     LATEST_OBJECT_VERSIONS.put(object.key, version)
105   end
106 end
```