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

(extended abstract of the MSc dissertation)

João Bernardo Sena Amaro
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Prof. Luís Eduardo Teixeira Rodrigues

***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 work we propose an architecture to perform transaction validation in a distributed and scalable manner. 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 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.**

I. Introduction

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 have support for running transactions, which makes the development of applications that use key-values stores harder and prone to errors [2]. As such, there has been a growing interest in developing mechanisms that allow the execution of transaction in these systems without strongly penalising their performance.

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 ACID properties [3], namely Atomicity, Consistency, Isolation and Durability.

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*. 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.

Therefore, in this work 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 work 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.

II. Related Work

A. 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.

Formally the concept of a transaction was first described by a set o properties known as the ACID properties [3]. It stands for Atomicity, Consistency, Isolation and Durability. In this work our focus is in the atomicity and isolation properties, and these can 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).
- Isolation: means that two concurrent transactions are isolated from each other. The classic notion of isolation

was formalised as *serialisability* [5], which states that the outcome of running multiple transactions concurrently is guaranteed to be the same as if these transactions had run serially.

B. 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 are failures that happen midway through the execution of a transaction and leave the state incomplete. Guaranteeing an all or nothing result in these cases requires a recovery algorithm, like the AIRES [6] algorithm, that is capable of recovering a partially executed transaction to a clean state.

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, it either commits or aborts. This problem is related to the consensus problem and can be solved with the help of a consensus algorithm, like Paxos [7]. However, many commercial database systems use a simpler Two-Phase Commit Protocol [8], that can be implemented more efficiently than a fault-tolerant consensus protocol, but may block in some faulty scenarios.

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.

1) 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 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.
2) 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.

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

$$N = number\ of\ data\ nodes\ involved\ in\ the\ transaction \tag{1}$$

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

$$total\ messages = \begin{cases} 2 & N = 1 \\ 4 \times N & N > 1 \end{cases} \tag{3}$$

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.

C. 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. As described in Section II-A the classic notion of isolation was formalised as *serialisability* [5], 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 result several other isolation levels have been proposed and implemented by database researchers and vendors.

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 (PCC) and optimistic (OCC) [9]. The choice of which one to use comes down 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.

Systems that use OCC mechanisms execute transactions optimistically and rely on a validation procedure at the end of a transaction to verify if any conflicts occurred during its execution. 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. This procedure assumes that transactions can be validated in serial order. Most of the systems we have studied rely on total ordering algorithms to achieve this.

D. 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.

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 Tango [10], vCorfu [11].

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 [12], but it can easily become a bottleneck in large scale systems due to the latency associated with passing the token from node to node.

3) Distributed Sequencers With Coordination In Line: This class of algorithms is a variant of an algorithm by Dale Skeen originally proposed in [13]. 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 [14]. 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.

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. several events to be order in batches. Examples of systems that use this class of algorithms are Eunomia [15] and FLACS [4].

III. Hierarchical Architecture for Deferred Validation of Transactions

A. Architecture

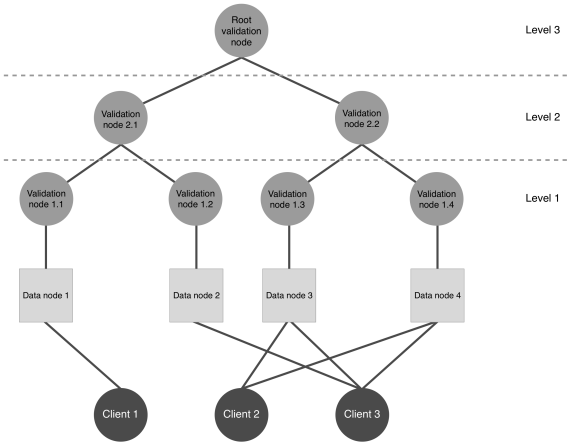

Figure 1. Proposed architecture. Key-value pairs are stored in the data nodes. Each leaf validatio node is co-located with the corresponding data node.

Figure 1 illustrates the architecture we are proposing.

The proposed architecture is intend to work over a distributed key-value storage system. It assumes that the key-value storage system 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 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 II-B, 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 III-B 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 orthogonal to this work.

B. Execution of Transactions

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.

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 the 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 III-B4), 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 serialized 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 form the local cache.

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.

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.

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).

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 in the way that it described in Section III-C. 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.

Distributed Transactions A transaction that accesses multiple data nodes is said to 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.

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, release the locks acquired when the commit was requested and depending on weather the result was successful or not the tentative versions are committed or discarded. The local validation node clock, used to generate timestamps, is moved forward if it is behind the transaction commit timestamp.

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. 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 in 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 is minimal.

C. Timestamp Generation

This section explains the 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.

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 nodes 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.

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.

D. Batching

Crucial to the design of the architecture we propose in this work is the use of message batching, when exchanging messages between nodes in the validation hierarchy. 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.

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 the messages sent by the validation nodes to the data nodes to commit or abort transactions.

E. Network Analysis

In order to understand the impact of that the structured communication pattern and the usage of batching has 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):

$$H = level\ of\ the\ validation\ node\ in\ the\ hierarchy\ that\ validates\ and\ commits\ the\ transaction \tag{4}$$

$$B = size\ of\ the\ batch\ used \tag{5}$$

$$communication\ steps = \begin{cases} 2 & N = 1 \\ 2 + \frac{H}{B} & N > 1 \end{cases} \tag{6}$$

$$total\ messages = \begin{cases} 2 & N = 1 \\ N + \frac{(H \times N) + N}{B} + 1 & N > 1 \end{cases} \tag{7}$$

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.

Applying the formulas presented in this section and the formulas presented in Section II-B, to a scenario 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, we can verify that the proposed architectures without any batching reduces the total turn around messages by 25% and reduces the total number of messages by 18%. With batching the reduction goes even further, reducing the total turn around messages by 49% and reducing the total number of messages by 67%.

F. 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, Riak KV [16]. 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 III-B. 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 [17] that allowed us measure the performance of these prototypes.

IV. Evaluation

A. Experimental Setup

1) Hardware Configuration: All the experiments presented in this chapter were executed in Google Cloud [18]. All the virtual machines used were configured to run Ubuntu 14.04. 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.

As a benchmarking tool we used a custom version of Basho Bench [17], that was modified to support the execution of transactions. Basho Bench was deployed in virtual machines, each with 8 cores, 30GB of memory and 10GB of disk.

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.

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.

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.

B. Network Load Experiments

As shown in Section III-E, the proposed architecture is able to reduce the network traffic required to validate and commit transaction when compared to 2PC. In order to better understand the impact of this reduction on the system performance, we ran multiple experiments with all logic but the network logic disabled, allowing network communication to be evaluated in isolation.

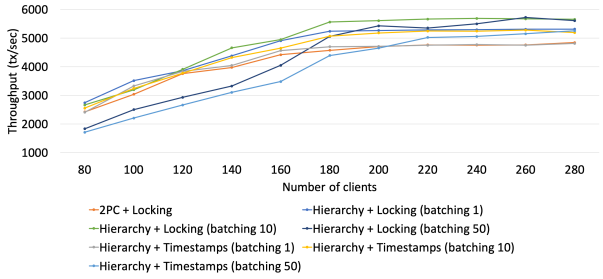

Figure 2. Throughput variation as the number of clients increases, with concurrency control logic disabled

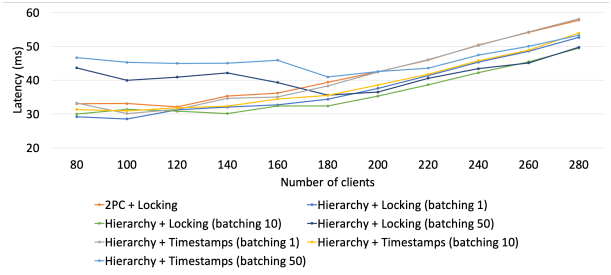

Figure 3. Latency variation as the number of clients increases, with concurrency control logic disabled

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.

The graphs presented in Figure 2 and Figure 3 show the results of these experiences. The graph in Figure 2 shows how the throughput of each system varies as the number of clients increases, and the graph in Figure 3 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.

As the results 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 260 clients are used. This can be observed by looking at the graph in Figure 3, 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 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.

2) Batching vs Latency Trade-off: As we have state before, and as the graphs in Figure 2 and Figure 3 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.

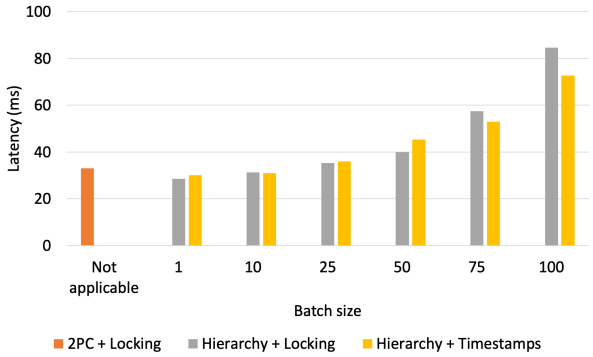

Figure 4. Latency variation as the batch size increases

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. The graph in this figure shows the 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%.

C. Concurrency Control Experiments

In order to understand if the performance gains shown by the results of Section IV-B1 were kept if the transaction validation logic was enabled, we ran a very similar set of experiences as the ones presented in that section, but with all the logic enabled.

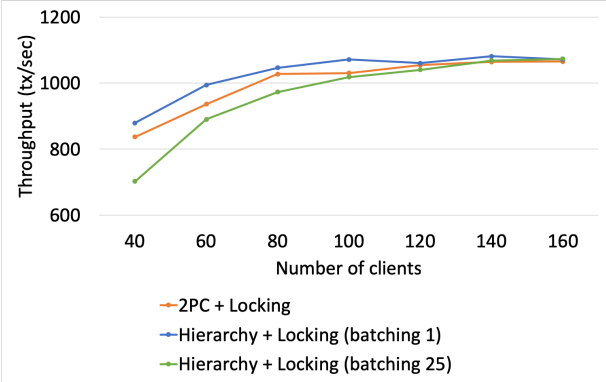

Figure 5. Throughput variation as the number of clients increases, with concurrency control logic enabled

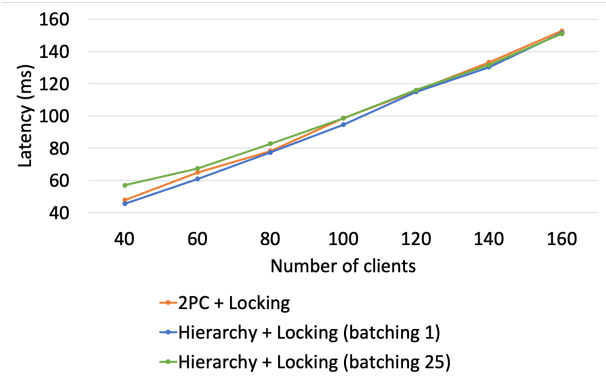

Figure 6. Latency variation as the number of clients increases, with concurrency control logic enabled

The graphs presented in Figure 5 and Figure 6 show the results of these experiences. The graph in Figure 5 shows how the throughput of each system varies as the number of clients increases, and the graph in Figure 6 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 IV-B1. 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 IV-B1 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 5, 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 6, 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 IV-B1 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 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 IV-B1 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.

D. Discussion

The experimental results presented in Section IV-B1 show that the proposed architecture has the potential to support up to 30% more clients and achieve 23% more throughput when compared to a system that uses the 2PC protocol. However, the results in Section IV-C show that the current implementations of the proposed architecture are not able to achieve the performance improvement that the results in Section IV-B1 show.

Our understanding of these results, as explained in Section IV-C, is that the prototypes of the proposed architecture have a limitation in their implementation. This limitation is in the way transactions are processed, which is serially. More precisely, data nodes are single threaded and process every request one at the time. By processing transactions serially, as more transactions 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 IV-B2, 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.

V. Conclusion 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 30% 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.

Acknowledgments

Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Manuel Bravo, Miguel Matos and Paolo Romano.

References

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

[6] 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.

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

[8] 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/

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] M. B. Chathuri Gunawardhana 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

[16] "Riak KV," http://basho.com/products/riak-kv/.

[17] "Basho Bench," https://github.com/basho/basho_bench.

[18] "Google Cloud," https://cloud.google.com/.