TÉCNICO LISBOA

Efficient Implementation of Causal Consistent Transactions in the Cloud

Taras Lykhenko

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof.ª Carla Maria Gonçalves Ferreira

November 2019

Acknowledgments

I would first like to thank my thesis advisor 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.

I must express my very profound gratitude to my parents and to Marta for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

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 storage systems, offering weak-consistency, have emerged as a key strategy to increase the performance and the scalability of cloud applications. Unfortunately, experience has shown that weak consistency put a burden on programmers, making the development of applications harder and more prone to bugs. This has raised the interest in the search for alternative consistency models, that can simplify the application design without impairing scalability. Transactional Causal Consistency (TCC) is a consistency criteria that meets these requirements. In this thesis we present FastCCS, a new algorithm that can offer TCC with less communication round than previous work. The experimental results reported in this thesis, where the performance of FastCCS is compared to that of other competing systems that have been previously been proposed in the literature, show that FastCCS can sustain a throughput that is 20% higher than those works.

Keywords

Cloud Computing, Key-Value Stores, Transactional Causal Consistency, Data Partitioning

Resumo

Os sistemas distribuídos de armazenamento chave-valor, que oferecem modelos de coerência fraca, emergiram como uma estratégia para aumentar o desempenho e a capacidade de escala dos sistemas que operam na nuvem. Contudo, a coerência fraca dificulta o desenvolvimento de aplicações, existindo enorme interesse em oferecer outros modelos de coerência que sejam úteis para os programadores sem comprometerem a capacidade de escala. O modelo de Coerência Causal Transacional (CCT) é particularmente relevante neste contexto. Nesta dissertação apresentamos o FastCCS, um novo algoritmo para suportar CCT em menos rondas de comunicação que os trabalhos anteriores. Resultados experimentais, em que comparamos o desempenho do FastCCS com o desempenho de outros sistemas propostos na literatura, mostram que o FastCCS pode suportar um débito 20% superior ao oferecido pelos sistemas anteriores.

Palavras Chave

Computação na Nuvem, Armazenamento Chave-Valor, Coerência Causal Transacional, Particionamento de Dados

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

TCC	Transactional Causal Consistency
CI	Cut Isolation
MAV	Monotonic Atomic View
EVT	Earliest Valid Time
LVT	Logical Valid Time
ACK	Acknowledge
2PC	Two Phase Commit
SI	Snapshot isolation
PDT	Physical Dependency Time
ST	Snapshot Timestamp
GST	Global Stable Time
LST	Local Stable Time
DBT	Distributed Balanced Tree
RTS	Remote Stable Time
I-CI	Item Cut Isolation
P-CI	Predicate Cut Isolation
AWS	Amazon Web Services
FIFO	First In, First Out
CDF	Cumulative Distributed Function

1

Introduction

Contents

In this thesis, we consider systems in which applications are structured in read and write sequences of operations, denoted as *transactions*, which access data held in a key-value storage system. The concurrent execution of these transactions, without adequate concurrency control mechanisms, may yield results that are different from the ones intended by the programmer. These anomalies are due to the interleaving of multiple operations from distinct transactions [1]. Strong consistency models, such as serializability [2], avoid this problem by ensuring that the result of concurrent execution of a set of transactions is equivalent some serial execution of these transactions. Unfortunately, mechanisms that ensure serialization are either blocking or cause transactions to abort and rerun, severely limiting the performance of storage systems [3]. In this thesis we address transactional consistency criteria than can circumvent those limitations. In particular, we address the design of efficient algorithms to implement Transactional Causal Consistency (TCC), a consistency that can be implemented using non-blocking algorithms while limiting the number of times a transaction needs to be rerun when a conflict occurs.

1.1 Motivation

With the aim of overcoming the performance limitations inherent in traditional transactional database systems, the first key-value storage systems that have been designed to support cloud application are willing to drop strong consistency guarantees, offering instead weak consistency models, such as eventual consistency [4]. However, experience has shown that the use of weak consistency models make application development difficult [5], so there is a keen interest in finding new consistency models and techniques to support these models that can be useful to programmers without compromising scalability [6]. The TCC model is particularly relevant in this context. This consistency model extends the Causal Consistency model, initially defined for single operations, allowing applications to read multiple objects from a causal snapshot and to perform atomic writes of multiple objects. The relevance of this model stems from the fact that causal consistency is the strongest consistency model that can be supported without compromising system availability in the presence of network failures or partitions [5].

It is worth to underline that, even in a centralized system, the effects that may result from the interleaving of concurrent executions already makes it difficult to offer consistency guarantees. Distribution further amplifies this complexity [7]. In particular, if different keys are stored on different nodes, the risk of a client reading inconsistent versions becomes larger, as it is in practice impossible to ensure that the multiple effects of a single transaction are visible at the same time on all servers [7]. On the other hand, the ability to deploy, and the opportunity to partition data, and to store different partitions on different servers is crucial for improving the performance and achieving scalability in cloud storage systems as it enables different requests to be processed in parallel by different servers [8].

Systems that support TCC often use non-blocking algorithms, which use control information (meta-

data) to verify whether the transaction has read from a causal cut. This metadata is written, read, and stored together with the data. When a transaction perform a set of readings that do not return mutually consistent version of the data items, it may be required to read new versions of the data (and this may occur more than once). The size of the metadata held by the algorithm has a significant impact on system performance. On one hand, the larger the metadata volume, the less efficient the system is, as it can consume a significant fraction of the system resources. On the other hand, a larger amount of metadata allows for greater accuracy in identifying the causal cut and can avoid redundant rounds of read operations. Many systems choose to reduce the size of metadata by creating what we call *false dependencies*; that is, scenarios where the metadata suggests that two operations may be causally related when, in fact, they are not.

1.2 Contributions

This thesis compares, implements and evaluates strategies for enforcing transactional causal consistency. The resulting contributions are the following:

- The design of a low-latency algorithm to support TCC, Fast Causal Consistent Snapshot (FastCCS). FastCCS explores a new trade-off between the degree of concurrency the system offers, the size of the metadata, and the number of communication steps required to execute a transaction. In particular, while previous TCC algorithms require the storage system to be linearizable (which limits its parallelism) or to perform multiple communication rounds to read a consistent snapshot. In contrast, FastCCS offers TCC on partitioned storage systems using only two communication rounds in the worst case. Besides, when exposed to load profiles dominated by read transactions, FastCCS executes most transactions in just one round of communication.

- A novel non-blocking read-only transaction algorithm that is both performant and returns a causal consistent read in the worst case in only to two rounds of communication.

- A novel write-only transaction algorithm that atomically writes a set of keys, is lock-free (low latency), and does not block concurrent read transactions.

1.3 Results

This thesis produced the following results:

- An implementation of FastCCS, in a real key value store (Cassandra)

- An experimental evaluation of the system implementation, regarding its performance. To better position FastCCS in respect to other causally consistent strategies, some representative systems were included as a way of creating acceptable boundaries.

1.4 Research History

This work was developed in the context of the Cosmos research project, that aims at finding techniques to offer causal consistent storage for edge computing scenarios. Efficient techniques to offer Transactional Causal Consistency when accessing the data are expected to be a key component in the final COSMOS architecture.

In my work I have benefited from the useful feedback from the team members of COSMOS, both from INESC-ID Lisboa and from NOVA LINCS.

A paper that presents parts of this work has been published as:

> T. Lykhenko and L. Rodrigues. Concretização Eficiente de Coerência Causal Transaccional na Nuvem. In *Actas do décimo primeiro Simpósio de Informática (Inforum)*, Guimarães, Portugal, Setembro de 2019.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) as part of the projects with references UID/CEC/50021/2019 and COSMOS (financed by the OE with ref. PTDC/EEICOM/29271/2017 and by Programa Operacional Regional de Lisboa in its FEDER component with ref. Lisbon-01-0145-FEDER-029271).

1.5 Structure of the Document

This thesis is organized as follows: Chapter 2 presents some applications developed for the cloud and introduces transactional consistency with some techniques that allow data stores with this consistency guarantee. Chapter 3 describes the design of FastCCS and addresses the implementation of FastCCS' prototype, as well as the other systems considered for evaluation. Chapter 4 reveals the results of the evaluation and makes some remarks about the differences among the systems. Chapter 5 concludes this thesis by outlining the discoveries and unveils some directions for future work.

2

Related Work

Contents

In this section, we discuss the guarantees that can be provided to clients accessing data stored in a distributed key-value storage system. We distinguish two classes of consistency guarantees: those that apply individual read and write operations and those that apply a sequence of operations that are treated as a block (and that we generically denote as transactions).

2.1 Linearizability and Session Guarantees

We start by discussing *linearizability* [9], which is probably the most intuitive model of consistency. Then we describe relevant relaxations of linearizability.

In the following discussion we assume that write and read operations may take some arbitrary amount of time. In a distributed message passing system, this time is the time required for the nodes to coordinate. Each operation has a starting time (when it is invoked) and a termination time (when the operation returns). If an operation O_2 starts before another operation O_1 terminates, operations O_1 and O_2 are said to be concurrent. If an operation O_2 starts after another operation O_1 terminates, O_2 is said to be subsequent to O_1.

Linearizabilty Under linearizability [9] , read and write operations appear to execute instantaneously, at an arbitrary time instant between the moment the operation is invoked and the operation completes. Thus, after a write completes, all reads must observe that write (or a subsequent write). If a read is executed concurrently with a write, the read can observe the value before or after the write. However, if the read observe the new value, all subsequent reads must also observe the new value, even if they are also concurrent with the write operation.

A linearizable memory register, that supports read and writes operations is denoted to be an *atomic register* [10]. Fault-tolerant atomic registers can be implemented in message passing systems, by letting different nodes to maintain a copy of the register value and by using the appropriate coordination mechanisms to execute read and write operations. The implementation is based on quorums. A write operation only returns when receives an acknowledgment from a majority of the servers. A read operation is more complex. First, the read must obtain values from a majority of nodes and select the most recent version from the quorum. Then, before returning, the read operation must write back the value read. This will ensure that subsequent reads will also return that value. Only after the write back phase is concluded (i.e., a majority of acknowledgements is collected for the write phase) the read operation is terminated.

As we have just seen, implementing linearizability in a non-blocking way is expensive and requires clients to always contact a majority of replicas. It is interesting to discuss what guarantees can be provided while offering better availability (namely by allowing progress even if the client can contact only a single replica). In the next paragraphs, we discuss a number of properties that are known as “session

guarantees" because they require the client to keep some state regarding its past interactions with the system. A session is assumed to be started when the client first contacts the system and get the initial session state and terminates when the client discards the session state.

In the context of our work, we say that a system is highly available if a user that can contact at least one server is guaranteed to get a response, even if a network partition prevents that server from coordinating with other servers in the system. Thus, in order to ensure high availability we need to resort to consistency levels that require little or none coordination among servers or, if such coordination exists, that it can be performed asynchronously in background. This definition of high availability allows the use of protocols that offer low latency, something that is important in a wide-area setting, were the latency among different nodes is high and network partitioning is likely to occur, making coordination among replicas a bottleneck.

Writes Follow Reads (WFR) under WFR, if a session observes an effect of an operation O_1 and subsequently executes another operation O_2, then another session can only observe effects of O_2 if it can also observe O_1's effects (or later values that supersede O_1's). Thus, the sequence of a write after a read must satisfy to Lamport's "happens-before" relation [11].

Monotonic Reads (MR) under MR, within a session, subsequent reads to a given object "never return any previous values". Reads from each item progress according to a total order.

Monotonic Writes (MW) requires that each session's writes become visible in the order they were submitted.

It is worth noting that, using these properties, the availability of a system can be increased by delaying the visibility of update operation. For instance, assume that a client makes a write operation w_2 that depends on some previous write w_1. Assume that w_2 becomes visible to other clients before w_1 has been applied at all datacenters. Another client that reads w_2 can later be blocked when attempting to read w_1 from another replica (this can happen if clients are allowed to contact different replicas). If that client is served with an older snapshot of the database, with versions that have been applied at all datacenters, that client will miss the new update but it will also avoid being subsequently blocked when reading other objects.

Some systems are more restrictive and force clients to remain connected to a single node. The latter type of systems offers what is called *sticky availability* [4]. Sticky availability usually requires full replication, given that the server to which the client is attached must be able to serve all requests from that client. In systems that use partial replication, a server may not be able to serve requests for data that is not replicated locally without coordinating with other servers. In fact, in that case, it may be simpler to allow clients to migrate in order to access data that is only stored in other servers. With this availability model, we can ensure the following session guarantees that were previously intangible as proven by Bailis *et al.* [3].

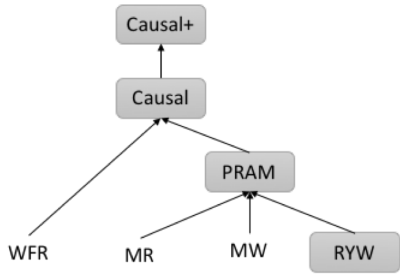


Figure 2.1: Relation among the session guarantees; the session guarantees in grey are only possible in sticky availability

Read your writes requires that whenever a client reads a given data item after updating it, the read returns the updated value (or a value that overwrote the previously written value).

PRAM (Pipelined Random Access Memory) lets clients observe a serialization of the operations (both reads and writes) within each session. Different clients can observe different serializations. It can be seen as a combination of monotonic reads, monotonic writes, and read your writes.

Causal consistency [12] is the combination of all of the session guarantees [13]. The causal dependencies of an operation are determined by happened-before relations ($\rightsquigarrow$), which are defined by three rules:

- **Thread of Execution**. If a and b are two operations executed by the same thread of execution (for instance, by the same client), then $a \rightsquigarrow b$ if a happens before b.

- **Reads From**. If a is an update operation and b is a read operation that reads the value set by a, then $a \rightsquigarrow b$.

- **Transitivity**. If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

Whether $w_a(k_a)$ and $w_b(k_b)$ are two write operations on the same or two separate keys, k_a and k_b. Let $r_a(k_a)$ and $r_b(k_b)$ be two read operations by the same client, where r_a is executed before r_b and where r_a returns the value written by w_a e r_b returns the value written by w_b. A storage system is said to be causally consistent if in the case that $w_a \rightsquigarrow w_b$ there is no write w'_b on the k_b key such that $w_b \rightsquigarrow w'_b \rightsquigarrow w_a$.

Note that two concurrent updates can be applied in different nodes in a different order and still respect causal consistency. In the lack of further updates, nodes would remain inconsistent indefinitely.

Causal+ consistency [14] is an extension to causal consistency that ensure that, in face of concurrent updates, one of the updates is applied last at all replicas.

Figure 2.1 summarizes the relation between the session guarantees mentioned above.

2.2 Transactional Properties

Transactions are sequences of operations that are grouped together and executed as one single indivisible operation. Transactions are widely used in database systems. The concurrent execution of transactions is coordinated by the database to ensure a set of proprieties, usually referred to as the ACID proprieties (ACID stands for atomicty, consistency, integrity, and durability). One of the strongest consistency criteria supported by databases is known as *serializability*. In short, serializability ensures that the concurrent execution of transactions yields the same results as a some serial executions of the same transactions. Ensuring serializability requires the database to execute some form of concurrency control. The most common approach to concurrency control consists in using locks to prevent concurrent transactions to observe inconsistent values.

Even in centralized databases some DBMS opt for a lower isolation levels, that offer weaker consistency models, to improve the performance of concurrent transactions. One of these weaker forms of consistency is Snapshot isolation (SI). SI guarantees that all reads made in a transaction will see a consistent snapshot of the database, and that the transaction itself will successfully commit only if none of the updates it has made conflicts with updates performed concurrently by other transactions (i.e., transactions that have executed concurrently based on the same snapshot).

Serializability and snapshot isolation levels are often explained in terms of the *anomalies* they prevent (where an anomaly is an observed state that would never occur if transactions where executed instantaneously, one after the other). Two anomalies that are relevant in this context are the *write skew* and the *lost update*. A write skew occurs when a transaction T_1 reads an object written by concurrent transaction T_2 and T_2 also reads an object written by T_1. A lost update occurs when one transaction T_1 reads a given data item, subsequently a second concurrent transaction T_2 updates the same data item, then T_1 also modifies that data item without taking into account the value written by T_2 (and thus, the system behaves as if the update by T_2 has never occurred). Serializability prevents both the write skew and the lost update anomalies. SI only prevents the lost update anomaly.

Unfortunately, in a distributed setting, these two isolation models can only be enforced if at least a majority of nodes are able to coordinate, given that both require transactions to be totally ordered. Therefore, these models cannot be enforced without compromising the availability of the system.

2.3 Isolation Levels that Support High Availability

In order to ensure that transactions are highly available, we need to define isolation levels that can be implemented with little coordination among replicas. We use the study by Bailis *et al.* [3] on highly available transactions to enumerate a number of isolation levels that can be enforced with minimal coordination.

As with serializability and snapshot isolation, several of these weak isolation models are defined in terms of the anomalies they prevent. In this context, we identify the following additional anomalies.

Dirty Writes A Dirty Write anomaly occurs when two concurrent transactions update two or more objects and these updates are applied in different orders at different objects [15]. This is illustrated in Figure 2.2, where T_1 updates x before T_2 and T_2 updates y before T_1.

T1	T2
$W(x_1)$	
	$W(x_2)$
	$W(y_2)$
$W(y_1)$	

Figure 2.2: Dirty Write anomaly

Dirty Reads A Dirty Read anomaly occurs when one transaction reads a value that has been written by another transaction that is still running and has not committed yet. Consider the example depicted in Figure 2.3. In this example Dirty Read anomaly occurs if T_3 read $x = 1$ or $x = 3$ in the case that T_2 aborted.

T1	T2	T3
$W_x(1)$	$W_x(3)$	$R_x()$
$W_x(2)$		

Figure 2.3: Dirty Read anomaly

Read Skew A Read Skew anomaly occurs when in a same transaction same read yields to a different result. Consider the example depicted in Figure 2.4. In this example Read Skew anomaly occurs if the reads over the item x in the transaction T_2 return two different results.

T1	T2
	Rx()
Wx(1)	
	Rx()

Figure 2.4: Read skew anomaly

With the help of the anomalies identified above, we can now list a number of relevant weak isolation models.

2.3.1 Read Uncommitted (RU)

Read Uncommitted is an isolation level that only prevents the “Dirty Writes” anomaly. It does not prevent other anomalies as it makes no attempt to prevent transactions from reading uncommitted values before a transaction has finished. Dirty writes can be avoided by defining a total order among transactions and ensuring that updates are applied to all objects according to that order.

2.3.2 Read Committed (RC)

Read Committed is an isolation level that ensures that transactions never access uncommitted or intermediate versions of data items. RC prevents both "Dirty Writes" and "Dirty Reads" anomalies. Dirty Reads can be prevented by not allowing the client to write in the database uncommitted data. Therefore, other transactions will never read uncommitted data. This can be achieved by requiring the client to buffer his writes until commit time or by having the servers to buffer multiple uncommitted values the same data and to only apply those writes when the corresponding commit is received.

2.3.3 Cut Isolation (CI)

Under Cut isolation, if a transaction reads the same data more than once, it sees the same value each time. This isolation level prevents "Read Skew" anomaly, if this property holds on data items, it is called Item Cut Isolation (I-CI), and if it holds when a transaction does a predicate-based read (e.g., SELECT... WHERE P) it is called Predicate Cut Isolation (P-CI). However this isolation level allows "Dirty Writes" and "Dirty Reads" anomalies.

Cut Isolation (CI) could be achieved by the transaction caching reads locally at the client and then reading from the cache so that the values do not change in the same transaction unless the transaction itself overwrites them. Alternatively the reads could be stored in multiple versions at the server, and the transaction would only read from this versions, this could be achieved by assigning the transaction to a group of servers (transaction group) that would store this version and the following reads would only read from this group of server until the end of the transaction. The cache and multiple versions of objects are garbage collected at the end of the transaction.

2.3.4 Monotonic Atomic View (MAV)

Under Monotonic Atomic View (MAV), once some of the effects of a transaction T_i are observed by another transaction T_j, after that, all effects of T_i are observed by T_j. That is, if a transaction T_j reads a version of an object that transaction T_i wrote, then a later read by T_j cannot return a value whose later version is installed by T_i. MAV prevents "Dirty Reads" by guaranteeing all or nothing visibility of transactions. However allows "Dirty Writes" anomalies.

MAV could be achieved using lightweight locks and/or concurrency control over data items [16]. This approach of achieving MAV does not satisfy high availability, because the system could stall in the presence of extended network partitions and has a significant impact on the system's throughput. There are enumerate alternatives to mitigate this problem and do an implementation of MAV without the use of locks [3, 17] this systems store every data object that was ever written and replicas then gossip information about versions they have observed and construct a lower bound on the versions that can be

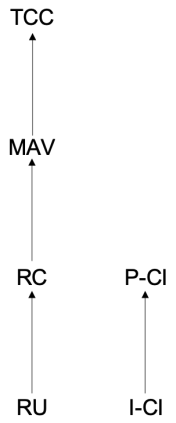


Figure 2.5: Relation between the guarantees of isolation levels

found on every replica. At the start of every transaction the client chooses a timestamp that is lower or equal to the lower global bound, and during the transaction, the replicas return data items that have a timestamp lower than the chosen timestamp.

2.3.5 Transactional Causal Consistency (TCC)

Causal consistency is defined for individual operations, regardless of how they are related, allowing sequences of read and write operations which results may not be as expected by programmers. Consider for example two write transactions $T^1 = w_a^1(k_a), w_b^1(k_b)$ and $T^2 = w_a^2(k_a), w_b^2(k_b)$ where $T^1 \rightsquigarrow T^2$ and two read transactions $T^3 = r_b^3(k_b), r_a^3(k_a)$ and $T^4 = r_a^4(k_a), r_b^4(kb)$. Causal consistency guarantees that in case that T^3 reads the value of k_b written by T^2 then, later it should read k_a written by T^2 (and not the previous value is written by T^1). This guarantee results from the fact that $w_a^1(k_a) \rightsquigarrow w_b^1(k_b) \rightsquigarrow w_a^2(k_a) \rightsquigarrow w_b^2(k_b)$ being independent from the way that operations are ordered in a transaction. However, causal consistency allows that T^3 reads a value written by T_1 in k_b and afterward reads a value written in k_a by T^2. Also, it allows that T^4 to read a value written by T^1 in K_a afterward reads a value in K_b written by T^2. Neither of these sequences violates causal consistency; However, introduces unexpected behavior.

Consider social network application, where the friendship relations are symmetric, and for that, it needs to ensure the following invariant. If user u_1 is visible in the friends' list of u_2, then u_2 needs to be also visible in the fiends' list of u_1. Consider that k_i stores the friends' list of u_i and that the transaction T^1 establishes a new relation of friendship between u_a and u_b, and that T^2 erases that relation. In this case, both T^3 and T^4 would read a state that would violate the proposed invariant.

Now consider an application that manages a shared folder, in which k_a registers which users have access to the folder and k_b stores the content of the folder. Consider that T^2 that excludes a user from the

list and afterward writes a new document to the folder witch that user should no longer have access. The sequences described above of the transaction T^4 would allow the transaction to read the old access-list and the new document, the application is unable to enforce the access restriction intended by the user.

TCC as MAV avoids some anomalies described above by ensuring that all the effects of one write transaction are visible to other transactions or none at all. However in contrast to MAV, clients read and write transactions must respect causal consistency. For example if two write transaction $T^1 = w_a^1(k_a), w_b^1(k_b)$ and $T^2 = w_c^2(k_c), w_d^2(k_d)$ where $T^1 \rightsquigarrow T^2$ and a read transaction $T^3 = r_a^3(k_a), r_c^1(k_c)$. In MAV is possible that T^3 returns a older value of k_a and a new value of k_b or vice verse as it only guarantees that after T^3 reads a value installed by T^1 or T^2, all the effect from T^1 or T^2 are observed by T^3. However, MAV does not capture the relation between transactions. And so it is possible that T^3 will observe the effect of T^2 without observing the effects of T^1, thus MAV does not capture the causal relation between transactions. In TCC the transaction must respect causal consistency, and so T^3 should not observe the effects of T^2 without also observing the effects of T^1. Making TCC a combination between MAV and causal consistency.

It should be noted that several systems extend causal consistency with support for read-only transactions [14, 18–20]. These systems avoid the anomaly illustrated by T^4, although not the anomaly illustrated by T^3. Thus, TCC is stronger than causal consistency with support for read transactions, which in turn is stronger than causal coherence with no support for transactions. The interested reader may find a hierarchical comparison of the various consistency models that have been proposed in the literature in [5]. However, we emphasize that TCC remains weaker than snapshot isolation that, in turn is weaker than serializability. These last two consistency models oblige to sort the write operations in a total order, which is not the case with the TCC.

Figure 2.5 summarizes all the guarantees and the relation among the different isolation levels mentioned above.

2.4 Systems that Support Transactions

In this section, we survey relevant systems from the related work that address the problem of supporting transactions in distributed storage systems. For each of them, we start by explaining how the transactions are implemented, the isolation level and session guarantees that they grant. Finally, we will analyze the chosen strategy for propagating the transaction's updates.

2.4.1 COPS

COPS [14] is a distributed key-value storage system that has been designed to run across a small number of datacenters. It implements a lock-free read-only transaction algorithm that provides clients

with a causal+ consistent view of multiple keys in a distributed key-value store in at most two rounds of local operations.

COPS assumes that each datacenter is fully replicated and linearizable. Moreover, clients only communicate with their local datacenter which makes COPS sticky available.

In COPS each client is responsible for maintaining its causal history. This history maintains all the client's direct and indirect (i.e., transitive) dependencies to the reads and writes operations it has performed. At the server side, each item is stored together with its list of dependencies, where each entry in the list contains the key and the version of the corresponding dependence.

Every time a client reads an item, the client appends the item's key to its history with that item's list of dependencies. When a client wants to write an item, first it computes the nearest dependency list and then sends it with the update. If the update was successful a new version is created and, it returns the new version number that the client then adds to its history with that item's list of dependencies. After the write finishes the datastore start to replicate it asynchronously to the other datacenters. When a server receives a remote write it delays it until all the writes dependencies are visible.

To retrieve multiple values in a causal+ consistent manner, a client issues read-only transaction with the desired set of item keys. The COPS as previously mentioned implements the read-only transactions algorithm in at most two rounds. In the first round, it issues n concurrent read operations to the local cluster, one for each key listed in the read-only transaction operation. Because COPS clients commit writes locally and the local data store is linearized, the local data store guarantees that each of these explicitly listed keys' dependencies are already satisfied, therefore the reads on them will immediately return. The first round of reads returns the corresponding items and the items' versions and the list of these items dependencies. The causal dependencies for each item are satisfied if either the client did not request the dependent key, or if it did, the version it retrieved was greater or equal than that item's version in the dependency list.

For all the items that not satisfy this condition, the client issues a second round of concurrent read operations for the greatest version in any dependency list from the first round. This only happens when there are write operations racing the reads of the first round. These versions satisfy all the causal dependencies because they are greater or equal to the needed versions. Furthermore, the second round of reads does not introduce new dependencies because dependencies are transitive and all the new retrieved items depend on the items from the first round which enables the read-only transaction to finish in at most two rounds of reads.

Although how the client migration between datacenters are not specified in COPS, it is still possible for a client to migrate from one datacenter to another. Since clients hold their entire causal history, once they arrive at the new datacenter, they can wait until all the required writes are made visible in the new datacenter before issuing new operations.

2.4.2 RAMP

RAMP [21] is an algorithm that enforces a lock-free read-only transaction that guarantees the MAV isolation model among transactions within a datacenter. The authors presented tree variations of the algorithm, that differ between each other by the size of the metadata and the number of rounds needed to return a read-only transaction. For brevity, we will focus on the version that optimizes the number of rounds needed to finish the read-only transaction because, in our algorithms, we will also favor solutions with smaller number of communication rounds (instead of algorithms that use smaller metadata size but more rounds of communication). In this version, the read-only transactions are guaranteed to finish in at most two round of communication even in the presence of racing write and read operations.

RAMP assume that each datacenter contains all the data (i.e., they consider full replication) but the the set of items is spread over multiple servers.

RAMP implements write-only transactions that use a two-phase commit protocol that ensures that if a write of a write-only transaction is visible in one partition, all the other writes of that transaction are present in the corresponding partitions. This helps to ensure that clients do not stall due to reading an item written by a transaction that the effects are not yet present in all the corresponding partitions. After receiving a write, the server creates a new version and responds to the client with a prepared message. The new version is only made visible if the server receives a commit message from the client. After one server receives the commit message, the transaction is guaranteed to be committed in the rest of the servers and cannot be aborted.

In RAMP the client is responsible for guaranteeing that the read-only transaction returns a MAV. In the first round, the client issues n concurrent read operations over the requested item set. The read operation of the first round returns the item, the transaction number of the write transaction and a list of the keys of the other items that the write transaction modified. After all the reads have returned, if the client has read a version of an item that is included in a write transaction with a higher transaction number, the client begins the second round of reads for the specific missing version number. This scenario only occurs if the reads are concurrent with the writes or if the commit message from the client has not yet arrived at the corresponding server (but has arrived to at least another server).

A possible optimization is the following: if read request is receive for a version that is not yet visible, it is safe to make that version immediately visible. This is safe because a client only requests that version if the corresponding transaction has already committed (and, therefore, it is safe to make the version visible).

2.4.3 Cure

Cure [5] was the first system to successfully support read and write operations within the same transaction while ensuring TCC. Furthermore, Cure ensures that reads and writes can be executed using two round of communications. In previous systems, such as COPS and RAMP, read operations need at most two rounds of communication while guaranteeing only causal consistency or MAV isolation respectively.

Cure has been designed for a geo-replicated key-value store. Cure assumes that the full set of items is replicated across different datacenters. Moreover, each datacenter is partitioned, where each partition stores a non-overlapping subset of the key-space. The client executes all the operations of a transaction in its local datacenter (i.e., Cure implements sticky availability).

Each partition holds two vector clocks of size equal to the number of datacenters in the system. One vector clock (PVC) is responsible for tracking the remote updates received from the replicated partitions in remote datacenters. The other is responsible for maintaining the latest *globally stable snapshot (GSS)* known to that partition. The GSS is maintained by the partitions within the same datacenter exchanging their PVCs. The PVC contains the physical clock values of the commit timestamps, and it is updated in the corresponding entry when a local or a remote commit is received. Also if no local commits are received within a threshold, the replica sends a heartbeat to the other replicas.

Transactions use a two-phase commit protocol where one of the participating partitions of the transaction is assigned as the transaction coordinator which is responsible for committing the client transaction. Before starting a transaction, the client gets a transaction timestamp from the coordinator. This timestamp is the max between the entry of that datacenter in the last GSS seen by the client and the current coordinator physical clock. To ensure that the client's causal remote dependencies are satisfied, the coordinator stalls the operations until the client's last seen GSS is lower or equal to the coordinator's GSS. All the client future reads during the transaction must return version lower than the transaction timestamp and the client buffers all of its writes locally until the commit phase. During the commit phase, the client sends the buffered write set to the coordinator that will then propagate to the corresponding replicas. The replicas will prepare the new version and respond to the coordinator with their current physical time. The coordinator after receiving all the clock values chooses the maximum as the commit timestamp and sends it to the corresponding replicas. As the client's dependencies are all locally satisfied the effect of the transaction will be immediately visible to the client.

After locally committed the values are asynchronously propagated to the other replicas. The remote transaction effects become visible when the GSS advances past their commit timestamp. This ensures that all causally related transactions are already visible locally because they have a smaller commit timestamp.

In the presence of network partition between datacenters, the observed transaction from remote datacenters will be delayed until the network recovers, while local updates will continue to be made

visible.

The client migration is not defined in Cure, however its possible to assume that the client could migrate to any datacenter. When the client tries to start a transaction, the coordinator will stall the client until all the causal dependencies are satisfied.

2.4.4 Eiger

Eiger [8] is a distributed key-value storage system that extends the ideas behind COPS. In addition to lock-free read-only transactions, Eiger introduces write-only transactions which ensure TCC isolation among transactions. Like COPS and Cure, clients transactions are performed in a single datacenter (to which the client is attached) and the key-value store is fully replicated across datacenters.

Eiger's read-only transaction algorithm has the same properties as COPS's, however, the implementation is different, namely Eiger uses logical time instead of explicit dependencies to enforce causal consistency. Each partition in a datacenter keeps an earliest valid time Earliest Valid Time (EVT) and its current Logical Valid Time (LVT). EVT is the partition's logical time when it committed the last visible operation. As in COPS, each client is responsible for maintaining its causal history.

The read-only transaction return in at most tree rounds. The first round consists of reading from the partitions that contain the target data objects optimistically. The partitions return the current visible value, the EVT and its LVT. Once all the first round reads return, this metadata is used to check the consistency. All values are consistent if $the\ maximum\ EVT \leq the\ minimum\ LVT$. If not, the transaction issues a second round of reads to the partitions that returned inconsistent values. In the second round, the client issues a read to a specific timestamp that satisfies $minimum\ LVT \geq the\ maximum\ EVT$, this ensures that all the reads return and are consistent with the previously read values. It is possible that the second round read, requests a key that has a pending commit value, this means that the LVT of the visible key is lower than the LVT requested by the client. In this case, the partition is unsure which value to return and must do a commit check with the coordinator partition to ensure which value to return without breaking causality.

The second round should be rare as it only occurs if a concurrent write operation is committed in the target partitions during the first round of reads.

Eiger's write-only transactions allow the client to write atomically across multiple data objects across multiple partitions without the use of locks. The write-only transactions are separated between local write-only transactions and replicated write-only transactions. The local version is used between the client and the local datacenter and the replicated version is used between datacenters to replicate the local write-only transactions. Both use a Two Phase Commit (2PC) protocol and assign one of the target partitions of the transaction as the coordinator. In the local write-only transactions, the coordinator first prepares the writes by sending a prepare message to all the target partitions. These partitions create

a new version of the data items and mark it as pending and respond with a yes vote to the coordinator. When the coordinator receives all the votes, it sends a commit message and make the new value visible and respond with an Acknowledge (ACK) to the coordinator. When the coordinator receives all the ACK, the coordinator ends the transaction and asynchronously propagates the transaction write by running a replicated write-only transactions. Upon receiving a replicated write-only transaction, the transaction coordinator must first check if all the dependencies are locally visible. When the dependency check returns, the coordinator proceeds with a local write-only transaction.

The client migration could be supported in the same way as the COPS because the client keeps its entire causal history and its operations could be stalled until all the dependencies are satisfied.

It is important to note that in more than 10% write workloads Eiger has a very poor performance [5]. This is probably due to the overhead of the dependency checks of the remote updates and the read transaction probably need always two round of communication.

2.4.5 Clock-SI

Clock-SI [22] is an algorithm that enforces Snapshot isolation (**SI**) using loosely synchronized clocks in a partitioned Data Stores. Most of the previous solutions that implemented SI in a distributed system relied on a centralized timestamp authority that managed the assignment of commit timestamps. However, this centralized authority introduced a single point of failure and could be a bottleneck in heavy workloads. Clock-SI overcame this problem by removing the timestamp authority, using instead loosely synchronized clocks to order transactions commits.

Clock-Si's read transactions return a consistent view of multiple keys across multiple partitions from a consistent snapshot. Moreover, as the Cure's read transactions, also return in two round of communication.

However, using clocks to achieve a consistent snapshot manifests as a challenge for two major reasons. First, clock skew can cause snapshot unavailability, this occurs when a partition P_1 issues a read transaction T_1 with the snapshot timestamp of t to a partition P_2 that is in snapshot $t - \theta$, where θ is the amount the P_2 clock is behind P_1's. Therefore the Snapshot t is not yet available in P_2, and if a local write transaction in P_2 commits between $t - \theta$ and t this change must be included in T_1's snapshot. Second, a pending commit of a write transaction can cause a snapshot to be unavailable. If a write transaction T_1 with a snapshot timestamp t that updated the value of x, and started a commit operation at t' and finished the commit at t'', where $t < t' < t''$, if a read transaction T_2 starts with snapshot timestamp between t' and t'' and tries to read x, it should not return the value written by T_1 because it is not certain if the commit will succeed, however we also cannot return the earlier value, because, if T_1's commit succeeds, this older value will not be part of a consistent snapshot at t''.

Both examples are instances of a situation where the snapshot specified by the snapshot timestamp

of a transaction is not yet available. Clock-SI deals with this problem by delaying the read operation until the snapshot becomes visible. In both cases, delaying a read operation does not introduce deadlocks, an operation waits only for a finite time until a commit operation completes, or a clock catches up. A possible optimization to reduce the probability and the duration of the read operation being delayed is by assigning a slightly older snapshot timestamp to the read transactions, having a cost that the read transaction return more stale data. This could the achieved by assigning to the read transaction a snapshot timestamp lower than the most recent timestamp snapshot by Δ. If we want the most recent data, the Δ needs to be set to 0. On the other hand, if we want to reduce the probability of the read operation being delayed, we could set the Δ to the maximum between the time required to commit a transaction to stable storage synchronously plus one round-trip network latency, and the maximum clock skew minus one-way network latency between two partitions.

The write transactions in Clock-SI are very similar to the Cure's [5], and the only difference is that the Clock-SI concurrent write transaction over the same items abort this is to ensure a much stronger isolation model (**SI**) than the Cure's transactional causal consistency.

2.4.6 Orbe

Orbe [18] is a distributed key-value store that provides read-only transactions using loosely synchronized clocks with causal consistency. Orbe requires two round of messages to execute a read-only transaction in the failure-free mode while COPS requires maximum two rounds. Also, Orbe only tracks the nearest dependencies at the client side, compared to COPS that the client has to track all the dependencies explicitly.

The key-value store is fully replicated across datacenters. Each datacenter is divided into N partitions, one per server and there are M different replicas. Clients performs the operations in a local datacenter.

The client keeps track of its causal history by maintaining a dependency matrix (DM_c) with N rows and M columns and a physical dependency time (PDT_c). PDT_c is the most recent update timestamp that the client depends on its session.

Each partition maintains a version vector (VV), which consists of M non-negative integer elements, that correspond to a logical clock of updates received from the replicas and the local updates. Also, every partition maintains a physical vector clock (PVV) with one entry for each replica containing the corresponding last seen physical clock value, to maintain this clock every partition sends periodically a heartbeat containing its current physical clock value.

When the client issues an update to an object, the client sends with the request its PDT_c and DM to the local server responsible for the partition. The partition upon receiving the request waits until its clock is greater than the PDT_c, this ensures that all the causal dependencies of the client are satisfied

before the update occurs. Then the partition creates a new version of the data object timestamped with the current physical clock value and the logical clock value. Attached to the new version is the DM_c, that afterward will be used on replication. The timestamp is then returned to the client that updates its PDT_c to the maximum between the received physical timestamp and its current PDT_c.

After the update to an object occurs, the server propagates the new version with the corresponding metadata. The receiving server uses the DM and the VV to verify the dependencies. It looks at the corresponding row of the same replicas and compares it to the VV. If all the values in VV are greater than or equal to the corresponding entries in the DM the server can go to the next step. If not, it means that the operation depends on other operations that the current datacenter may not store. So, the server contacts the corresponding servers, making an explicit dependency check. If they fulfill the dependencies, the remote update can be made visible and the corresponding vectors (VV and PVV) entries updated.

When the client issues a read-only transaction, it sends a read set to one partition. The partition upon receiving the request associates to the transaction a snapshot timestamp, similarly to [22] the Snapshot Timestamp (ST) is the current physical clock value minus Δ, where Δ is some positive number usually the time of one network round-trip. This reduces the probability of the read operation being delayed and the duration of the delay. If the partition does not contain an item from the transaction read set, the partition reads from another local partition that contains the item. However, before that partition could read the item it must first delay the read until two conditions hold: first the transaction ST must be lower than the partition current clock value, second the transaction ST must be lower than the partition's PVV. When the two conditions hold, the partition responds with the highest version of the item that is lower than ST. After all the reads finish, the client receives a set of values from the requested items and updates the Physical Dependency Time (PDT) to the maximum of all retrieved items timestamp and the current PDT.

However, this creates a problem in the face of failing servers or network partition between datacenter, as the transaction read is delayed until $ST < PVV$, this means if a partition fails and stops sending heartbeats the transaction needs to be stalled until the partition recovers, because the replica is uncertain that all the dependencies are satisfied. To mitigate this problem when a transaction is delayed past some threshold, the transaction ST is changed to a lower value. This will return a more stale data in favor of reducing the delay of the read operation. However, if the downtime is very high, no useful progress will be made until the partition recovers, so to mitigate this problem Orbe changes to a failure mode. This mode uses a two-round read similar to Eiger [8] ensuring that more recent data is returned and the system progresses.

2.4.7 GentleRain

GentleRain [20] is a causally consistent key-value store that implements causal consistent read-only transactions. The contribution of this system is that it uses a physical timestamp to track dependencies which reduce the communication and storage overhead and eliminates dependency check messages for updates improving throughput compared to COPS [14] and Orbe [18].

The clients only have to store two timestamps, a PDT as in Orbe and the last seen Global Stable Time (GST).

The server only maintains a physical vector clock (PVV) with one entry for each replica containing the corresponding last seen physical clock value and a GST. The lowest entry in the PVV is designated as Local Stable Time (LST), that is a lower bound of all the updates visible in all the replicas.

GST is the lower bound on the minimum LST of all partitions in the same datacenter. This value could be calculated by the partitions at the same datacenter periodically exchanging between each other their LST. However, merely exchanging the LST between partitions could be a bottleneck and limit the scalability in the presence of a high number of partitions. To efficiently derive the GST, GentleRain uses a tree. Child nodes send their LST to the parent node, upon receiving all the LST from the children it sends the lowest one to its parent node and this process repeats until the root node receives all the LST. The root node then calculates the GST and pushes it down the tree, thus saving the number of messages needed to calculate the GST.

GentleRain's update operation the client only sends its PDT. The server stalls the update until its clock is greater than the PDT received. Afterward, creates a new version assigned with the update timestamp and updates the PVV. The server returns the update timestamp to the client and propagates the update to the replicas.

The server upon receiving a remote update, creates a new version. However, this version is not visible to local clients until the partition GST becomes greater than its update timestamp.

When the client issues a read-only transaction to any server in the client's local datacenter, this server will act as the transaction coordinator. The request contains the item set, the client's GST_c and PDT_c. If the coordinator's GST is lower than the GST_c, it updates the GST and starts requesting the item with a timestamp lower than the GST to the corresponding partitions. The coordinator returns the collected item values with the maximum update timestamp and the maximum GST. Upon receiving the reply, the client updates its dependency time and the GST_c.

However using the read operation it is possible that a client could read an item that is not yet in the GST, and so the read-only transactions will violate the causal consistency. If the PDT_c and GST_c are only apart by some defined threshold, the coordinator delays the read until the $PDT_c < GST_c$, in the case that the difference is greater than the threshold then the coordinator will use the protocol used in Eiger [8] for causally consistent read-only snapshots.

2.4.8 Yesquel

Yesquel [23] is a key-value store that in the other end of the spectrum from the system mentioned above because in opposite to all the systems that offer very high availability sacrificing all the interesting features that relational databases provided such as ACID transactions, joins, between clauses, and others. The Yesquel provides all the features of a SQL relational database, however scales as well as NoSQL in the same type of workload. To support this type of operations efficiently in a distributed fashion, Yesquel implements a Distributed Balanced Tree (DBT) heavily inspired in a B+Tree. DBT consists of a tree where the nodes are spread across servers, the leaf nodes of the tree contain the values and the keys, and the interior nodes stores keys and pointers to other nodes.

Clients cache the inner nodes of the tree locally. This works because B+tree have a large fan-out and relatively few inner-nodes. Caching the tree allows the clients to search in the tree locally and only need to contact the leaf node, and so reducing the number of messages needed. However, the cache of the client becomes outdated as the tree is dynamic and changes over time as the nodes are split or merged due to insertion, deletion or replication. To solve this problem each node holds a fence interval, that is the lowest and the highest value key that the node contains, before fetching from the leaf node the client request a fence interval test, if the key is in the fence interval the client can proceed with the operation. If not the client goes up the tree to test the parent nodes until it finds a node that contains the wanted key, at the same time the client updates its tree locally in the cache, due to the nature of this type of trees the upper nodes rarely change and in the majority of time it can rebuild only a part of the cache without needing to fetch the whole tree again. When the client finds the node that satisfies the fence interval, it starts going down the tree and updating its local cache until reaching the leaf node.

Unlike a normal B+tree that split due to a tree node being overfull or empty, the Yesquel also slips overloaded nodes. This operation is called load splits. Yesquel estimates the workload of the node by keeping track of the number of accesses to partitions. If a node is overloaded, the node is split according to the estimated workload as each node receives approximately half of the workload. However is possible that one key is extremely popular, and no optimal split could be done, so the key is replicated and the node is split according to that key, to the lower bits of the key are attached random numbers and the client when searching for key also attaches some random generated number allowing to split the load between replicated keys.

Each client has a query processor making processing capacity increase linearly with the number of clients, which allows the system to scale well even with a high number of clients.

Yesquel's read-only transaction do not block or abort. To improve latency, the client acts as the transaction coordinator. As the state of the coordinator is irrelevant for the transaction outcome, the system can recover from fails by running periodically a function that checks the pending transaction and aborts them in cases of detecting falling servers or coordinator. Yesquel uses physical clocks to order

operations similarly to the clock-Si, however, uses a much stronger 2PC with locking for write operations. Making Yesquel perform very poorly in write-heavy workloads.

2.4.9 ChainReaction

ChainReaction [19] is a geo-distributed key-value data store that offers causal+ consistency, as the name suggests it was developed on top of chain replication [24]. ChainReaction supports causal read-only transactions. However in contrast to other systems such as COPS that assume linearizability inside the datacenter. That is all the reads are performed in the tail of the chain. In contrast, ChainReaction allows the read operation in the middle of the chain without breaking causal consistency and so improving the overall throughput.

The client tracks its causal history by maintaining a table with one entry for each item viewed during the client session. This entry contains the item's version and the chain index vector. The chain index vector contains an identifier that captures the position on the chain from where this item was last read, one entry per datacenter. All the read operation that the client does must have this metadata attached to the request.

The client update request must contain the key of the object, the new value, and a compression of all the read objects by the client since the last update. This update is forwarded to the chain head node that assigns a new version number to the update and then propagates to the following nodes. When the update is replicated at k nodes then the update is denominated as a *k-stable* update. Only then, that the update result is returned to the client with the new version number and index of the last node that turned the update *k-stable*. When the update reaches the tail of the chain, it is denominated *DC-Write-Stable*.

After the chain head assigns a new version number to the update, this update is scheduled for replication and then propagated in batches to the other datacenters. To ensure that the update respects the causal history of the client, it is only visible after all the versions of the objects that this update depends are DC-Write-Stable in that datacenter.

When the client issues a read operation to the local datacenter and the chain length of the datacenter is equal to the chain index entry, every node in that chain can answer the request without needing to wait for a remote update. Otherwise, the request could be answered by all the nodes until the one that is specified in the chain index. However, as the updates are propagated asynchronously, it is possible that the item version that the client wants to read is not yet present in the head of the chain. So the head of the corresponding object in that datacenter needs to wait for the remote update or redirect the client to a datacenter that has that version number of the object.

The implementation of read-only transaction uses a sequencer process per datacenter. This sequencer is used to order all the put operation and reads that are part of a read-only transaction. The sequence number process maintains a different sequence number for each chain. To ensure that the client

read objects respecting causal+ consistency even when updates are applied concurrently and without blocking update operations, ChainReaction keeps multiple versions of the same object. When the client issues a read transaction, it first must request the sequence number for each chain that contains the targeted objects. This sequence number is assigned by the sequencer and issues an individual read for the heads of the chain that contain the objects. The object returned has the last update sequence number, lower than the number assigned to the transaction.

Due to the asynchrony of the system is possible that the sequencer will order a put operation before the transaction and the value is not yet available in the head of the chain. This could be mitigated by stalling the read transaction until the write becomes visible or a timeout occurs leading to a transaction abort. It is also possible that the client has a dependency on an object read in a different datacenter that is not yet visible. In this case, the transaction is aborted and retried in a two-phase procedure. First, all the dependencies that have failed are verified by using a blocking read operation that blocks until the tail of the chain contains the object makes visible the version from which the client is dependent. Afterward, it reissues the transaction, and it is guaranteed to succeed, it is important to note that this case is very slow compared to the normal version.

2.4.10 Wren

Wren [25] is a distributed key-value storage system that extends the ideas behind Cure [5]. Cure enforces causality and atomicity at the cost of potentially blocking read operations. For example, the snapshot assigned to the client's transaction may be ”in the future” concerning the snapshot installed by a server from which T read an item. Also, Eiger's second-round read suffers from the same problem. As the client can read a partially committed value that is still pending at the other partitions, instead of blocking the reads, Eiger issues a commit check to the coordinator to know which versions it can safely return. Wren avoids blocking by providing to a transaction a snapshot that only includes writes of transactions that have been installed at all partitions.

Each partition holds two timestamps. One LST such that all transactions with a commit timestamp lower than or equal to LST have been installed at all partitions. The other is responsible for tracking the remote updates received from the replicated partitions in remote datacenters. The last one is called Remote Stable Time (RTS). Periodically partitions in the same datacenter exchange the lowest prepare timestamp minus one and the latest commit timestamp received from a replicated partition. LST and RTS are set to the minimum received from all local partitions.

The read transaction fallows the logic as Cure. However, as the clients are assigned snapshots slightly in the past, it does not satisfy Monotonic Reads as the snapshot assigned to the client will not contain the client's latest writes. To avoid blocking clients until it is snapshot becomes visible, Wren exploits the fact that the only client dependencies that may not be in the local stable snapshot are the

Table 2.1: Systems that offer causal consistency. R represents the number of read rounds and V the number of rounds that return values. NBR and WTX respectively represent non-blocking reads and write transactions. N represents the number of partitions, M the number of data center and ts the physical clock value.

System	R	V	NBR	WTX	Metadata Size	Strategy	Concistency
Ramp [21]	≤ 2	≤ 2	✓	✓	O(k)	Explicit Check	MAV
ChainReaction [19]	≥ 1	≥ 1	✗	✗	O(M)	Sequencer	Causal
Orbe [18]	2	1	✗	✗	O(N x M)	Stabilization	Causal
GentleRain [20]	2	1	✗	✗	1 ts	Stabilization	Causal
COPS [14]	≤ 2	≤ 2	✓	✗	O(|deps|)	Explicit Check	Causal
Cure [5]	2	1	✗	✓	O(M)	Stabilization	TCC
Eiger [8]	≤ 3	≤ 2	✓	✓	O(|deps|)	Explicit Check	TCC
Wren [25]	2	1	✓	✓	2 ts	Stabilization	TCC
FastCCS	≤ 2	≤ 2	✓	✓	O(N)	Stabilization	TCC
Yesquel [23]	1	1	✗	✓	O(k)	Explicit locks	SI
Clock-SI [22]	2	1	✗	✓	1 ts	Stabilization	SI

items that the client wrote. Therefore, Wren implements a client-side cache that contains the client's more recent writes. As the LST progresses and the client writes become visible, it can safely remove the writes from the cache. This solution presents several problems, namely, storage and computation overhead, to maintaining the cache.

The write transaction follows the same logic as Cure.

2.5 Comparison

The implementation of different forms of transactions in systems with low consistency has been much studied in recent years, and it is possible to find different approaches to the problem in the literature, that were described in detail in Chapter 2. In this section, we will compare and discuss how the concepts and ideas presented in the related work have inspired us in the design of FastCCS. However, this comparison is not trivial due to different systems having different types of consistency and transaction isolation guarantees, so to make a reasonable comparison between the systems, this section will be subdivided into comparing the different techniques, the types of transactions and consistency guarantees. Table 2.1 summarizes the information of this comparison.

Strategies can be classified into four broad categories according to the technique they use to ensure that the transaction is performed in a consistent cut, namely: stabilization, serialization, explicit dependency check, and explicit locks. Next, we will briefly discuss each of these techniques.

- The strategy in which the coherent cut is obtained by stabilization, do it by ordering the transactions in total order, based on a timestamp. A partition can satisfy a read operation made by a transaction that executes at the instant t when it knows that it has already become aware of the effect of all write transactions that were executed with timestamp lower than t. Systems such as Orbe [18],

GentleRain [20], and Cure [5], use physical clocks as the timestamp to order the transactions. Using the physical clock has the advantage of them monotonically increasing without the presence of events. Unfortunately, since it is impossible to synchronize clocks with complete accuracy, partitions need to exchange information to know which timestamps are in the past of all other nodes. Moreover, it is possible due to clock skew that a client reads a value that the timestamp is the ”future” for some partition, forcing the partition to stall the operation until its clock catches up. Wren [25] uses similar strategies though using hybrid clocks.

- In systems where the consistent cut is obtained by serialization, a centralized component is used, which takes cognizance of all transactions, and orders them in full. This solution has the disadvantage of creating a bottleneck in the system, which limits the scalability of the system. One of the systems that implement this strategy is ChainReaction [19].

- Systems that use explicit consistent cut verification require that all write transactions to be associated with metadata that captures the client's causal past. System like COPS [14], Eiger [8] and Ramp [21] use this strategy. However, the client must piggyback causal dependency information and execute expensive dependency checks across partitions.

- Systems that use explicit locks to read from consistent cut significantly limit the throughput of write operations due to the need to abort transactions in the presence of a deadlock or concurrent accesses to the same data items. Making them incompatible with high availability. One of the systems that implement this strategy is Yesquel [19].

Some systems try to pursue general-purpose transactions. In their pursuit of general transactions, these systems all choose consistency models that cannot guarantee low-latency operations. These include Yesquel and Clock-SI. Other systems such as Wren and Cure, try to give the give to the client a much weaker form of general-purpose transaction guaranteeing only TCC. In these types of transactions, the client is responsible for caching its writes and execute all read operations from a specific snapshot. However, the pre-set of the snapshot has a cost. In this case, one round of communication and was previously shown [7] this cost is not negligible.

Other systems take one step back and implement only read transactions without implementing any write transactions. Some of these systems are ChainReaction, Orbe, GentleRain, Cops. Dropping write transactions for lower latency introduces several artifacts that make it more difficult for programmers to reason about. These types of transactions are incompatible with TCC.

Eiger and Ramp have a much moderate approach. These systems separate read from write transactions. Thus, they can optimize read transactions without introducing too much latency. More, over it is compatible with TCC.

Now we will discuss how FastCCS places in the related work. The goal of this thesis is to create a system that satisfies the following goals. Our system must support scalable and low latency transactions. Making two strategies incompatible with our requirements. Explicit locks as this strategy are incompatible with low latency and serialization due to the limitation of the scalability. We opted not to use explicit dependencies as it would introduce too much metadata overhead at the client-side [5], due to the client needing to piggyback its operations. So, we chose stabilization as a strategy to be implemented. However, trying to avoid some downfalls of previous systems that implemented this strategy with physical clocks. We opted to use logical vector clocks that avoids blocking the client reads due to clock skew.

For the type of transaction, as we mentioned, one of our goals is low latency. General transactions have the disadvantage of requiring two rounds of communication. Moreover, as we FastCCS typical workload would be read-heavy, it would introduce unnecessary latency; however, as we want to provide TCC to avoid some anomalies related to write operations. Thus, we opted for read-only and write-only transactions, as they are compatible with TCC.

Summary

This chapter has introduced the consistency guarantees that apply individual read and write operations and those that apply to transactions. Usually, distributed storage systems prefer weak consistency models, as they offer the lowest impact on performance. However, they are difficult to reason with, even more, when interacting with multiple objects simultaneously that lead to unexpected behaviors. While many cloud stores offer different forms of consistency, few apply this consistency over an operation that interacts with multiple keys. In the next chapter, a novel system that extends TCC is proposed, aimed at offering low latency and high throughput.

3

FastCCS: Fast Causal Consistent Snapshot

Contents

This chapter introduces FastCCS, that offers low latency Transactional Causal Consistency while preserving system concurrency. Section 3.1 expresses the goals that need to be fulfilled. Section 3.2 overviews the design of FastCCS, highlighting each component and their interaction. Section 3.2.2 describes the system metadata. Section 3.3 details the different protocols used in FastCCS. Section 3.4 provides a proof of correctness for our algorithms. The library used by clients to access the distributed key-values store when using FastCCS is described in Section 3.5. Section 3.6 describes how partitions are implemented in FastCCS. Section 3.7 describes how the system limits the number of old versions. Section 3.8 describes the behavior of FastCCS in the case of failing partitions or network partitions. Finally, Section 3.9 describes the implementation of the system.

3.1 Goals

In many cloud applications, read operations dominate the workload of the system [7]. For example, 99.8 % of Facebook distributed database operations [26] are reads and the latency of those operations is particularly important because a client request can lead to thousands of reads and some of these reads need to be done in sequence, and the critical path can reach dozens of reads [27].

Therefore, it is crucial to offer an algorithm that supports non-blocking read operations with as fewest as possible rounds of communication.

FastCCS focuses on providing read and write transactions for cloud application without increasing significantly the overall latency experience by the user compared to eventual consistency, however providing to the client higher consistency guarantees.

3.2 Design

This section starts with a small introduction of the different system components, and then it overviews the metadata that is used in the system to implement TCC.

3.2.1 System Components

Clients connect to a nearby datacenter, and applications strive to handle requests entirely within that datacenter. Inside the datacenter, client requests are served by a front-end web server. Front-ends serve requests by reading and writing data to and from storage tier nodes.

In order to scale, the storage cluster in each datacenter is typically partitioned across 10s to 1000s of machines. As a primitive example, Server 1 might store and serve user profiles for people whose names start with ‘A’, Server 2 for ‘B’, and so on.

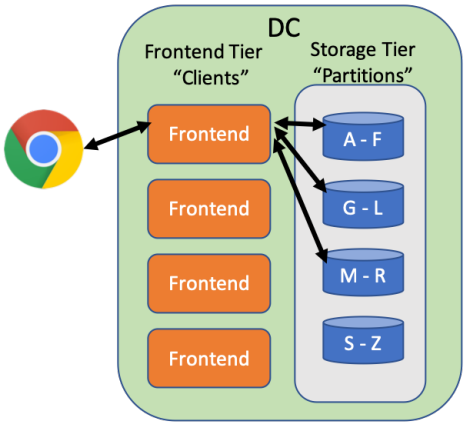


Figure 3.1: FastCCS architecture.

As a storage system, FastCCS' clients are the front-end web servers that issue read and write operations on behalf of the human users. When we say, "a client writes a value", we mean that an application running on a web or application server writes into the storage system. The architecture is represented in Figure 3.1.

3.2.2 Metadata

In the recent works was proven that it is impossible to achieve one round read transaction while supporting TCC or as they were called in the literature *fast transaction* without compromising availability [28]. Thus, the read transaction protocol needs at least two rounds of communication always to return a consistent causal view over multiple keys. The optimal solution is to, in the majority of cases, only needing one round of communication and the second round for the cases that it fails. The success rate of the first round will influence the overall latency that the client experiences. So we can derive a two-step algorithm in which the first part is optimistic, that reads the most recent available values, then has a function that verifies if the first round was successful or not. The second part is for the times that the optimistic read fails. The more metadata that the system stores with the versions, the more precise is the assessment if the first round was successful. For example, if the system stores little metadata, the system could conclude that the first round was unsuccessful and need to execute a second round that would introduce more latency, even if, in reality, the result was causally consistent. This is called a *false dependency*. Moreover, the metadata needs to be precise enough to determine the second round read condition, or else, more rounds would be needed to return a consistent result. On the other end of the spectrum, the system could store more metadata that would reduce the false dependencies at the cost of a heavier first round and more storage overhead. Other solutions may include fetching more than one

version of the keys in the first round, the more versions that are fetched, the lower the probability of the second round occurring, however, with the cost of more expensive first-round reads. It becomes clear that exits a balance between the number of rounds of communication and the size of the metadata.

FastCCS implements more precise metadata with the intent to reduce to the minimum the number of second rounds without too large sized metadata that would detriment the overall performance of the system. We will now describe the metadata implemented in FastCCS.

FastCCS attaches to each read or write transaction a timestamp, that is materialized in the form of a vector clock of size N. Where N is the number of partitions in a datacenter, every time a write transaction changes the value of a key, it is created a new version of that key, which is stored with the commit timestamp of that transaction. The client also maintains a vector clock timestamp vc of size N of the last operation that it performed. This vector clock tracks the client's causal past. When a new version of a key is created, it is passed by three states of knowledge: pending, confirmed, visible. A timestamp associated with a pending transaction is temporary and will be updated when the transaction is confirmed. A key passes to the state of visible when all the keys in that write transaction are confirmed in all participating partitions. Every partition i keeps a sequence number that is incremented every time that a new version of a key is created in that partition. This sequence number is denominated as sn_i. Every partition i also maintains snapshot vector clock timestamp svc_i of size N, where every entry holds the maximum value of sn_j observed by the partition i, denominated as snapshot vector clock. Every key modified by a transaction with a commit timestamp lower or equal to the snapshot vector clock, are guaranteed to be confirmed in all the partitions. Every partition i updates the value of the entry $svc_i[i]$ every time a transaction is confirmed in that partition. Moreover, partitions periodically exchange the value entry of $svc_i[i]$. Every time that the snapshot vector clock is updated, the partition checks if there exists transactions in the state confirmed with a commit timestamp lower or equal than the svc_i and passes them to the visible state accordingly.

3.3 Protocols

In this section, there is a detailed description of FastCCS's protocols, accompanied by algorithms 1 and 2, which are referenced through the text. The algorithms show the pseudocode for the protocols that run in the client proxy and partitions, respectively. Figure 3.2 depicts the execution flow of a write-only transaction, where the client c_w issues a write transaction T_W to keys a and b, that are installed respectively in partition p_A and partition p_B. Figure 3.3 depicts the execution flow of a read-only transaction, where the client c_R issues a read transaction T_R to keys a and b, that are installed respectively in partition p_A and partition p_B.

Algorithm 1 Client code

```
    ▷ State kept by the client
1:  vc_c
    ▷ Handles a read transaction request
2:  function READ_ONLY_TRANS(requests)
    ▷ Round 1
3:      for r ∈ requests do                          ▷ Send first round requests in parallel
4:          send ⟨ multiget_slice r, vc_c ⟩ to r.p
5:          receive ⟨ multiget_slice val[r] ⟩ from r.p
6:      repeat ← ∅
7:      for r ∈ requests do
8:          vc_c max(val[r].ctm[k], vc_c[k]), ∀k ∈ N
9:          for r' ∈ requests do
10:             if val[r].svc < val[r'].ctm then
11:                 repeat ← repeat ∪ r
12:                 break
13:     if repeat = ∅ then
14:         return val
15:     else
    ▷ Round 2
16:         for r ∈ repeat do                        ▷ Send second round requests in parallel
17:             send ⟨ multiget_slice_by_time r, vc_c ⟩ to p_i
18:             receive ⟨ multiget_slice_by_time_resp val[r] ⟩ from p_i
19:         return val
    ▷ Handles a write transaction request
20: function WRITE_ONLY_TRANS(requests)
21:     cp ← CHOSE_RANDOMLY_COORDINADOR_PARTITION(request)
22:     for r ∈ requests do                          ▷ Send requests in parallel
23:         send ⟨ transactional_batch_mutate r, vc_c, cp, SIZE(requests), tx_id ⟩ to r.p
24:         receive ⟨ transactional_batch_mutate_resp ct ⟩ from r.p
25:         if ct ≠ ∅ then
26:             vc_c ← ct
```

3.3.1 Write-Only Transactions

The write transaction T is executed in two rounds of communication, in the following way. Let $\mathcal{P}(T)$ be the set of partitions that store keys modified by T. In the first round, the client chooses one of the partitions as the transaction coordinator as $cp \in \mathcal{P}(T)$ (Algorithm 1, line 21). The client sends to each partition $\forall i \in \mathcal{P}(T)$ the new value of the modified keys and the coordinating identifier of the transaction, the vc_c and the number of participants in the transaction (Algorithm 1, line 23) (Figure 3.2, step 1) and waits for the responses (Algorithm 1, line 24). Each partition, upon receiving this message, increments its sequence number sn_i (Figure 3.2, step 2) and creates a new pending version of the key to which it assigns a temporary commit timestamp ct_i where $ct_i[j] = -1, i \neq j$ and $ct_i[i] = sn_i$ (Algorithm 2, line 18 - line 21). If the partition is the transaction coordinator, the partition waits for the other partitions to respond to $\mathcal{P}(T)$ otherwise the value of sn_i is returned to the coordinator (Algorithm 2, line 26) (Figure 3.2, step 3).

Algorithm 2 Partition p_i code

▷ State kept by the partition
svc, sn, $pending$, $versions$
▷ Handles a read transaction request
upon receive ⟨ **multiget_slice** r, vc_c⟩ from c **do**
  $resp \leftarrow \emptyset$
  $ctm \leftarrow \emptyset$
  $svc_i[k] \leftarrow max(svc_i[p], vc_c[p]), \forall p \in N$
  for $key \in r$ **do**
    $version \leftarrow \{w \in versions[key] \wedge svc \geq w.ct\}$
    $ctm[p] \leftarrow max(ctm[p], version.ct[p]), \forall p \in N$
    $resp \leftarrow resp \cup version$
  send ⟨ **multiget_slice_resp** $resp, ctm, svc$ ⟩ to c
▷ Handles a second round read transaction request
upon receive ⟨ **multiget_slice_by_time** r, vc_c⟩ from c **do**
  $svc_i[k] \leftarrow max(svc_i[p], vc_c[p]), \forall p \in N$
  for $key \in r$ **do**
    $version \leftarrow \{w \in versions[key] \wedge vc_c \geq w.ct\}$
    $ctm[p] \leftarrow max(ctm[p], version.ct[p]), \forall p \in N$
    $resp \cup version$
▷ Handles clients write transaction request
upon receive ⟨ **batch_mutate** $r, vc_c, cp, size, tx_{id}$⟩ from c **do**
  $ct[j] \leftarrow vc_c[j], i \neq j, \forall p \in N$
  $ct[i] \leftarrow$ INCREMENT_AND_GET(sn)
  $versions[r.key] \leftarrow versions[r.key] \cup \langle r.value, ct\rangle$
  $pending \leftarrow pending \cup \langle tx_{id}, \langle r, ct, c\rangle\rangle$
  if $cp = p_i$ **then**
    $prepared = \emptyset$
    $pending[tx_{id}] \leftarrow pending[tx_{id}] \cup \langle size, prepared\rangle$
  else
    send ⟨ **prepared** ct[i],tx_{id} ⟩ to cp
upon receive ⟨ **prepared** sn_t, tx_{id}⟩ from p_j **do**
  $pending[tx_{id}].ct[p_j] \leftarrow sn_t$
  $pending[tx_{id}].prepared \leftarrow pending[tx_{id}].prepared \cup p_j$
  if SIZE$(pending[tx_{id}].prepared) = pending[tx_{id}].size$ **then**
    for $p_j \in pending[tx_{id}].prepared$ **do**
      send ⟨ **commit** $pending[tx_{id}].ct, tx_{id}$ ⟩ to p_j
    COMMIT_LOCAL(tx_{id})
    send ⟨ **batch_mutate_resp** $pending[tx_{id}].ct$ ⟩ to $pending[tx_{id}].c$
upon receive ⟨ **commit** ct, tx_{id}⟩ from cp **do**
  COMMIT_LOCAL(tx_{id})
  send ⟨ **batch_mutate_resp** $\emptyset$ ⟩ to pending[tx_{id}].c
function COMMIT_LOCAL(tx_{id})
  wait for $svc[i] + 1 = ct[i]$
  for $key \in pending[tx_{id}].r$ **do**
    $version \leftarrow w \in versions[key] \wedge w.ct[i] = ct[i]$
    $version.ct \leftarrow ct$
  $pending \leftarrow pending \setminus tx_{id}$
  $svc[i] \leftarrow ct[i]$

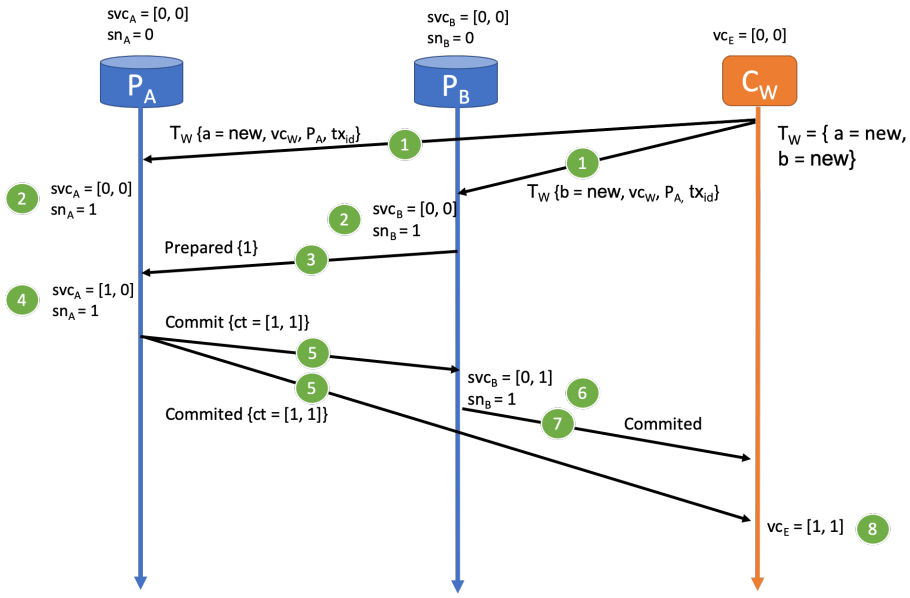


Figure 3.2: FastCCS write transaction. The numbers represent the steps in the algorithm. The client c_W issues a write transaction T_W to partitions p_A and p_B.

The second round begins when the coordinator receives a response from all partitions (Algorithm 2, line 33) (Figure 3.2, step 4). The coordinator creates a commit timestamp for the ct_T transaction, where $ct_T[i] = max(vc_c[i], sn_i), \forall i \in \mathcal{P}(T)$. The coordinator sends this value to all partitions in $\mathcal{P}(T)$ (Figure 3.2, step 5). Receiving the value of ct_T, the partition passes all versions of keys modified by transaction T to the committed state, associating with these versions the value of the final timestamp. Finally, each partition waits until $svc_i[i] + 1 \geq ct_T[i]$ (Algorithm 2, line 39) (Figure 3.2, step 6), at which point it is sure that all transactions with a commit timestamp lower than ct_T are already committed in the partition i. When this condition is met, a response is sent to the client (Figure 3.2, step 7). The write transaction is considered terminated when the client receives a response from all the participating partitions, ensuring that all values that the transaction wrote are already committed on all partitions. The coordinator sends the transaction ct_T to the client. The client adopts ct_T as its new client timestamp ($vc_c = ct_T$) (Figure 3.2, step 8).

3.3.2 Read-Only Transactions

The read transaction T runs in at most two rounds of communication as follows. Let $\mathcal{P}(T)$ be the set of partitions that store keys read by T.

In the first round, the client sends to each $i \in \mathcal{P}(T)$ partition the keys it wants to read , along with its vc_c (Figure 3.3, step 1) and waits for a response from all partitions in $\mathcal{P}(T)$. Each partition, upon receiving this message, updates its snapshot vector clock by making $svc_i[k] = max(svc_i[k], vc_c[k]), \forall k \in$

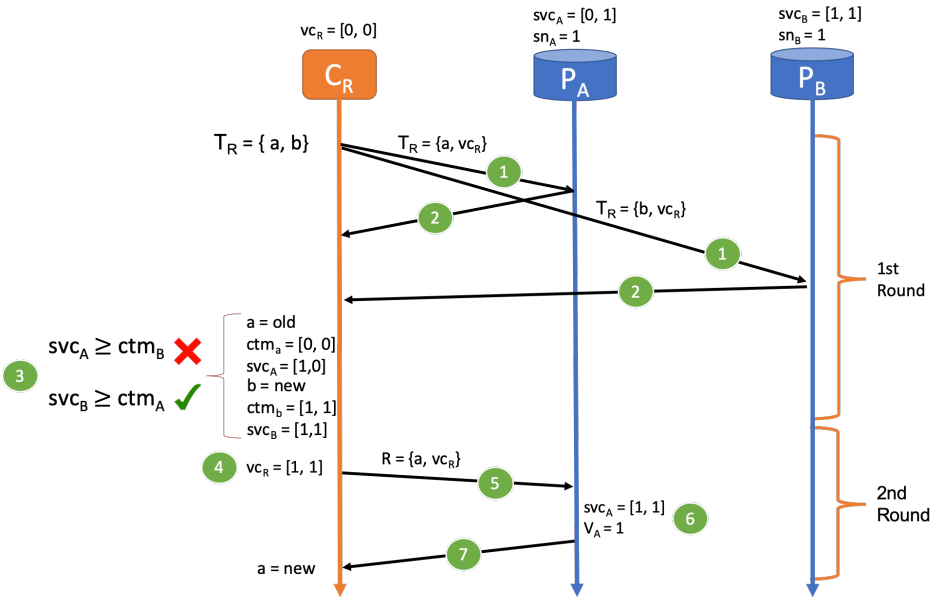


Figure 3.3: FastCCS read transaction. The numbers represent the steps in the algorithm. The client c_R issues a read transaction T_R to partitions p_A and p_B.

N (Algorithm 2, line 5). This is possible since vc_c represents the maximum timestamp the client has ever seen, and if the effects of a transaction are already visible it means that the transaction has already been committed on all partitions and thus making possible to advance svc to vc_c safely. This ensures that all transaction effects that the client has observed in the past will be visible, thus ensuring atomicity and causality.

Partitions return newer versions with $ct_T \leq svc$, svc, and the maximum commit timestamp read ctm, where $ctm[k] = max(ctm[k], ct[k]), \forall k \in N$ (Algorithm 2, line 5) (Figure 3.3, step 2).

When the client receives a response from all partitions. Client checks if $svc_i \geq ct_j, \forall i, j \in \mathcal{P}(T) \wedge i \neq j$ (Algorithm 1, line 10)(Figure 3.3, step 3) and updates its $vc_c = max(ctm_i[k], vc_c[k]), \forall i \in \mathcal{P}(T) \wedge \forall k \in N$ (Figure 3.3, step 4). If the condition is met, the client ends the transaction and returns the values. If the condition is not met, the client has read of a snapshot that may or may not be consistent. Therefore, the client must start a second round of read operations on partitions that did not meet the previous condition by sending their new vc_c (Figure 3.3, step 5). Partitions receiving a request from the second reading round return the latest versions with $ct_T \leq vc_c$, and update their snapshot vector clock similar to the first round (Figure 3.3, steps 6 and 7).

The second round ensures that a coherent causal snapshot will be returned. Moreover, the stabilization protocol ensures that all versions that the client read in the first round are already installed on all partitions, so it is not necessary to block reads until the new version is installed. Since the client reads versions that satisfy $ct_T \leq vc_c$, no extra rounds of communication will be required because the reads returned in the first round have a $ct_T \leq vc_c$.

3.3.3 Stabilization

For the write transaction effect to become visible, the commit timestamp of the transaction must be lower or equal to the snapshot vector clock. For that reason, it is essential to implement an algorithm that guarantees the progression of the snapshot vector clock. FastCCS implements the following algorithm.

Each partition i is responsible for updating its snapshot vector clock locality svc_i. Let T be a write transaction that modified one or more keys stored in partition i and which sequence number assigned to the transaction in that partition is $svc_i[i]+1$. When the write transaction T goes from pending state to the confirmed state and the sequence number assigned is $svc_i[i] + 1$, the partition increments its value of $svc_i[i]$. Periodically, each partition i exchanges between all the other partitions its value of $svc_i[i]$. When a partition j receives the value of $svc_i[i]$, j updates its svc_j to $svc_j[i] = max(svc_j[i], svc_i[i])$. Also the clients help to progress the svc when they present their client timestamp, this is possible because the client only observes versions that are committed in all partitions so the partition if receives a $ct > svc$ can safely update the svc to the ct. It is important to note that how partitions propagate their svc entry does not influence the correctness of the algorithm and it is even possible that only the clients help to progress the snapshot vector clock, with the cost of greater visibility latency.

3.4 Correctness

We provide an informal proof that FastCCS implements TCC by showing that the snapshot read by a transaction is causally consistent and respects the atomicity of committed transactions. This proof is based upon the proof presented in [5].

Proposition 1. If an update u_2 depends on an update u_1, then $u_2.ct > u_1.ct$.

An update u_2 depends on u_1 if the client's previous read transaction of u_2 read from a snapshot that contains u_1. From Alg. 1 line 8, the following inequality holds: $vc_c \geq u_1.ct$. Since the commit vector clock entry for each participating partition is generated by a sequence number sn attributed by the partition to that transaction and sn is monotonically increasing. Moreover, from Alg. 2 line 18 all entries in the ct corresponding to the not modified partitions will be equal to the entries of the vc_c corresponding to that partitions. It's guaranteed that if the u_2 updates keys in the same partition as u_1, from Alg. 2 line 19 the sequence number will be greater than the sn attributed to u_1 in that partition, and for the keys updated in different partition from u_1 will be equal or greater than the vc_c, as the commit timestamp is greater than vc_c and, vc_c is greater than $u_1.ct$, so $u_2.ct > u_1.ct$.

Proposition 2. A partition snapshot vector clock svc_i implies that the partition p_i has received all updates with commit vector clock $\leq svc_i$.

Now we show that there are no pending updates with $ct \leq svc_i$. When updating $svc_i[i]$, the partition i finds the minimum prepared time stamps from the transactions in the prepared phase. As sn_i is

monotonically increasing and the $svc_i[i]$ represents the minimum prepared time minus 1 (Alg. 2 line 39 and line 44), it is guaranteed that future transactions will receive a commit time which is greater than or equal to this minimum prepared timestamp, guaranteeing that the partition has already committed all updates for the snapshot svc_i.

Proposition 3. Reads return values from a causally consistent snapshot.

Proposition 1 and 2, together guarantee that by including all updates in a partition which have a commit vector clock $\leq$ snapshot vector clock, the read returns values from a causally consistent snapshot. This is true even when reading from multiple partitions because it reads from the same snapshot.

Proposition 4. Reading from a snapshot respects atomicity.

Atomicity is not violated even though updates are made visible independently by each partition. All updates from a transaction belong to the same snapshot because they receive the same commit vector clock. The visibility of an update is delayed until the same snapshot is available in all (accessed) partitions, thus reading all or no updates from a transaction.

FastCCS implements TCC, as every transaction reads from a causally consistent snapshot (Proposition 3) that includes all effects (Proposition 4) of its causally dependent transactions.

3.5 Client Library

This library is responsible for handling the splitting, routing, and re-assembly of the transactional requests. As the transaction can span multiple partitions, the client library is responsible for splitting the client's request, and so hiding the internal structure of the replicas by forwarding requests to the correct partitions. For knowing the correct partition, it uses a hashing function that is provided by Cassandra [29]. Moreover, to hide the internal algorithm of the transaction and to give an illusion to the client that the transaction is executed as a single operation. The client library is responsible for the re-assembly of the client request, and in case of Eiger [8] and FastCCS to check if the first round of parallels reads was successful.

Also, the client library keeps track of the causal dependencies of the client to avoid introducing false dependencies between thread-of-execution done on behalf of different clients. Each client has a unique id that it uses to communicate with the client library so that the dependencies of different clients (e.g., operations done on behalf of c_1 are not entangled with operations of c_2). This unique id is also used to generate the transaction id, by simply appending the unique client id to a sequence number.

3.6 Data Partitions

Partitions store the keys and the values associated with those keys. Given that FastCCS relies on the second round of the algorithm to return a consistent snapshot, it is possible that the targeted snapshot was overwritten by concurrent write transactions, forcing the transaction to execute more rounds of reads to return a consistent snapshot. So, to allow the transaction always to return a consistent snapshot and to mitigate possible race conditions between write and read transactions, the system must keep old versions of the values. To satisfy this requirement, partitions implement multi-version concurrency control (MVCC) [1], making that if the partition needs to update a value of the key, it will not overwrite the previous value instead will create a new version of that key.

3.7 Garbage Collection of Obsolete Versions

To limit the number of versions that are maintained in the system, partitions need to implement some sort of garbage collection mechanism in order to delete versions that are no longer necessary. One way to garbage-collect old versions is as follows. Partitions periodically exchange among them the lower snapshot read by any active transaction; versions that are older than this snapshot can be safely deleted. However, this approach requires more communication among partitions, increasing the overall load on the network. Furthermore, in the presence of network partitions, garbage collection may stall, because partitions would not be able to compute the lower snapshot. FastCCS implements the garbage collection mechanism proposed in Eiger [8] that circumvents these problems by assuming that servers can have their physical clocks loosely synchronized. This garbage collection strategy limits old versions in two ways. First, the transaction has a timeout that specifies its real-time duration. If the timeout fires, the client library restarts the transaction. Thus, servers only need to store values that were overwritten during this timeout period. Second, the partition only retains values that could be requested in the second round. Thus, a server only keeps versions that are newer than those returned in a first-round within the timeout duration. This mechanism requires nodes to maintain additional metadata for each version of the data, namely the last time at which that version has been accessed.

3.8 Faults

In this section, we examine the behavior of FastCCS in the face of faults. In a cloud environment, faults of components such as servers, network equipment, or even power outage are common [3]. So the design of the system needs to take in to account these types of scenarios.

Single server failures are common, FastCCS mitigates single server failures by creating logical

servers composed by multiple physical servers. This can be implemented by running a Paxos [30] group. For example, three server Paxos groups can withstand one server failure in the group.

If a client fails during the read transaction, this operation does not change the state of the system, and the client does not hold any meaningful state. So the system can proceed normally.

If a client fails during the write transaction depending on which state the write transaction is, there are two different outcomes to the write transaction. First, if the client fails during the first communication round and only a portion of the targeted partitions receive the write operation, after the timeout of the transaction triggers, the coordinator sends an abort operation to the participating partitions. If a non-coordinating partition that already voted for the commit timestamp and the timeout triggers. The partition sends a check commit to the coordinating partition to confirm if it is safe to abort the transaction. Second, if the client fails after the first communication round the write proceeds as usual, and will be committed.

If a network partitioning occurs between the servers running the partitions, the client reads will continue to return results, however returning stale results. Clients executing write transactions will stall until the partition is resolved.

3.9 Implementation

In order to evaluate the proposed system, it needs to be compared to other solutions that employ different techniques. The chosen ones were: Eiger [8], for offering read and write transaction with multiple rounds of reads (at most three) and Wren [25] for comparing if two round of reads has a higher overhead than reading in at most two rounds although with higher metadata overhead. In addition to the causally consistent systems, FastCCS is also compared with a system that only offers eventual consistency as these types of storage systems are predominant in cloud applications. It is expected that eventual consistency should offer the best latency, as it makes updates visible as soon as they are received and only needs one round to satisfy read transactions without any metadata that normally be used to enforce consistency. We have implemented FastCCS as a modification of Eiger's fork of Cassandra. Wren has also been implemented by modifying Eiger's fork of Cassandra. For Eiger, we have just used the original code. For eventual consistency, we used the original fork of Cassandra from which Eiger is based. Our prototype of FastCCS added and modified 3000 lines of Java code to the existing 75000 lines of code in Cassandra. Alongside the implementation of the prototypes in Cassandra, we have also implemented all four prototypes in PeerSim. This implementation uses approximately 2000 lines of Java code. As described in the next chapter, simulations are used in the evaluation to estimate the performance of the algorithms in settings with large numbers of nodes.

Summary

This chapter has addressed the design an implementation of FastCCS, a cloud store that offers TCC without introducing high latency overhead while preserving the causal consistency of the system, by introducing a protocol which supports non blocking read transactions with as few rounds of communication as possible. The next chapter presents a comprehensive evaluation of the proposed solution's prototype.

4

Evaluation

Contents

This chapter describes the experimental setting we have used to assess the performance of FastCCS in comparison with other systems previously proposed in the literature and presents the results that we have obtain from our experiments. The chapters starts by stating the main goals of the evaluation exercise (Section 4.1). Next, Section 4.2 describes the experimental setting that we have used to conduct the experiments. The presentation of the experimental results starts in Section 4.3, with the analysis of each system operation latency experienced by the client. Then, Section 4.4 explores the impact that each system has in the throughput, when compared to eventual consistency. Section 4.5 looks on how each system can scale horizontally with the increase in the number of partitions. Finally, Section 4.6 discusses the bottlenecks observed in different systems.

4.1 Goals of the Evaluation

The main goal of the evaluation is to understand the impact of implementing causal consistent transactions and how it affects the performance of the system compared to other alternatives that offer lower consistency guarantees and no isolation levels between transactions such as eventual consistency. We evaluate different qualities such as operation latency, throughput, and scalability.

The evaluation also aims at understanding how FastCCS performs when compared to other systems that have been previously proposed in the literature, namely: Eiger [8], Wren [25], and a system that offers only eventual consistency. The latter works as a common baseline, as it provides the best achievable result for each test, giving insights regarding the overheads imposed by the systems that offer stronger guarantees.

4.2 Experimental Setting

In our evaluation, we use two different experimental testbeds that complement each other, namely, we perform experiments using simulations and we perform experiments in a real deployment on Amazon Web Services (AWS) [31]. Simulations allow us to experiments with system sizes that we cannot afford to deploy in AWS. The AWS deployment allow us to assess the performance of FastCCS in a realistic setting, and also to offer some form of validation of the results obtained using simulations.

Simulations have been performed using PeerSim [32], which was augmented with some modules that help in increasing the fidelity of the results. Namely, PeerSim has been configured with extensions that simulate First In, First Out (FIFO) point-to-point channels with configurable network latency and finite bandwidth. Each node has a bandwidth limit of $1\ Gb/s$, and each message is randomly delayed with an average value of $0.5\ ms$ (latency observed between servers in an AWS data center [33]). Since the performance of the various algorithms depends fundamentally on the number of rounds and the

Table 4.1: Parameters of the dynamic workload generator.

Parameter	Default	Range
Keys/Read	5	2-64
Keys/Write	5	2-64
Partitions	8	2 - 64
Value Size (B)	128	2 - 1024
Write fraction	0.05	0.01-0.5

need to wait for a causal cut, to speed up simulations we do not aim at capturing CPU utilization or overhead of the disk access time. This allow us to quickly obtain results that approximate well enough the real setting. However, as we will note later, in some systems, the CPU can also be a bottleneck, in particular in systems that have to manage large amounts of metadata. These effects are only captured in the real deployment.

For the real deployment we have built a prototype of our system using Cassandra, a well known key-value store which is very used in the industry. For running the experiments, the prototype was deployed in AWS. In these deployments, each partition run within m4.2xlarge instance with 8 vCPUs and 32 GB of memory. Each client machine runs a client library that issues read and write transactions eagerly. In order to benchmark the proposed architecture we used the a modified version of the stress test of Cassandra. The modifications we have introduced were the minimum necessary to make it work with the new client library.

Unless specified, the experiments presented in this chapter have as default parameters presented in Table 4.1. Clients populate the key-value store before running the experiments. The experiments run in five trials of 1 minute, and we have ignored the first and last quarters of each trial to avoid experimental artifacts.

Clients are configured to execute requests to the system in closed-loop, waiting for the response to the previous request before placing a new request. When a client issues an operation, it chooses with a certain probability whether it corresponds to a write or read transaction (different experiments can use different read/write ratios). The keys accessed by each request are selected using a Gaussian distribution (the number of keys accessed also varies with the experience).

4.3 Latency

The first experiment compares the latency observed by the clients when they perform read and write operations using different systems. It is expected that read operations present a lower overall latency when compared to write operations. In fact, our target systems have been designed to optimize the latency of read operations, which are assumed to be the most frequent operations.

Figure 4.1 depicts the Cumulative Distributed Function (CDF) of the latency observed by the clients

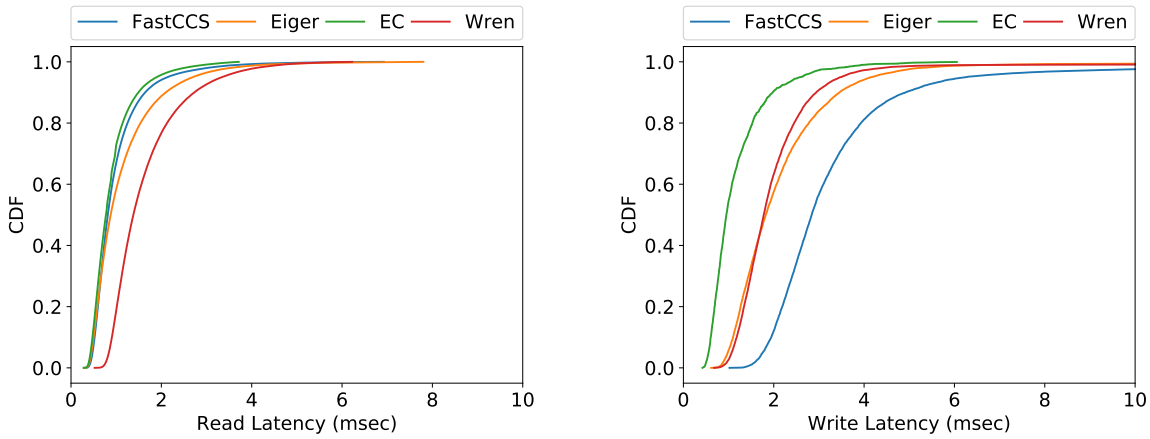
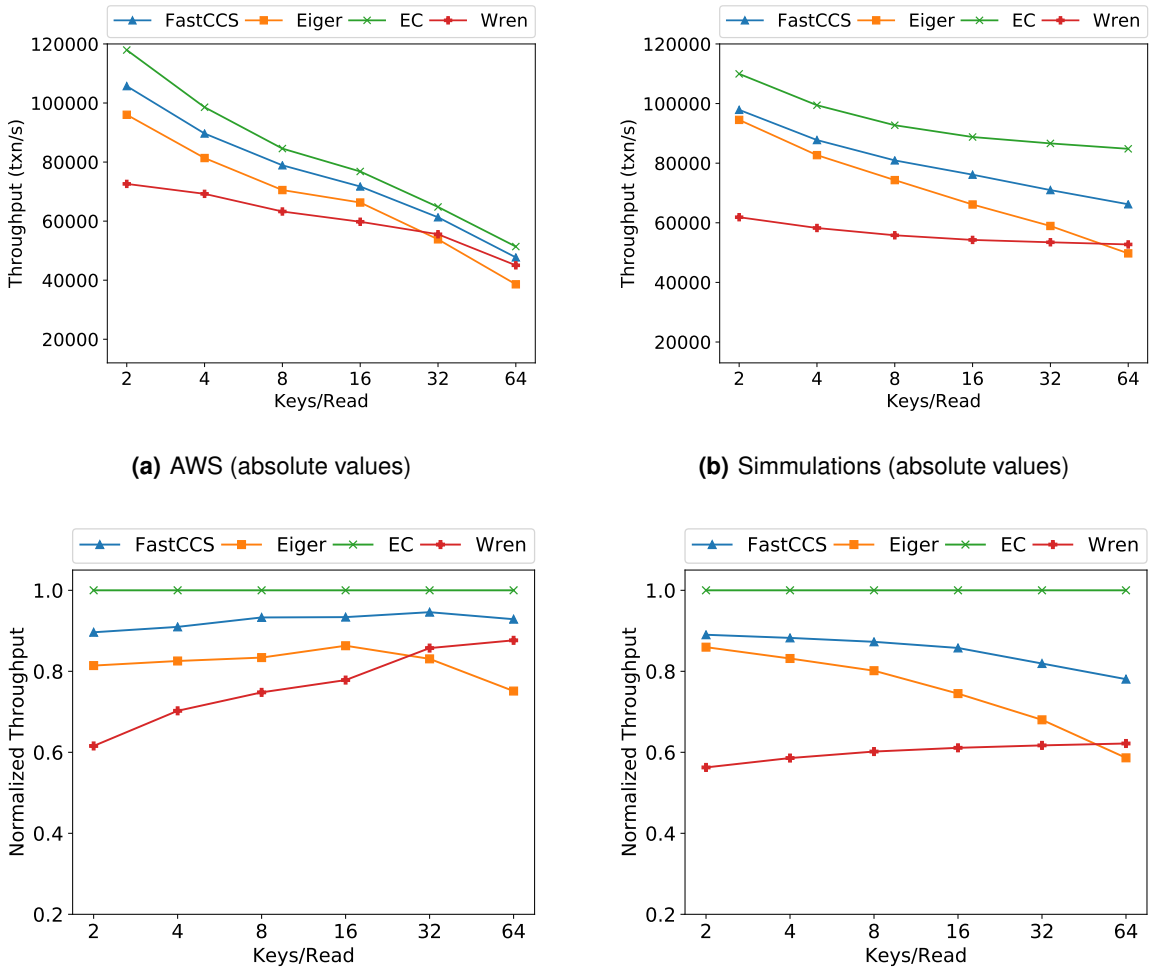


Figure 4.1: Cumulative distribution function for each system's read and write latencies.

for both read and write operations. These numbers have been collected using the deployment on AWS.

We start by discussing the performance of reads. Not surprisingly, the lowest latency if offered by a system that offers only eventual consistency. This can be explained by the fact that this system is the one that requires less metadata and less coordination (operations only need one round of parallel requests to return a result). Interestingly, FastCCS closely follows the performance of an eventual consistent system for read operations. This happens because FastCCS implements a good tradeoff between metadata and accuracy: it does not require clients to maintain complex dependency trees but avoids most false positives, and therefore allows most reads to execute in a single communication round. Eiger has less precise metadata compared to FastCCS, which leads to more false dependencies which, in turn, often generate a second round of reads. Moreover, in Eiger, a write transaction becomes visible as soon as the partition receives the commit, so the second round of reads may target a pending commit value; the partition needs to issue a request to the coordinator of that transaction to confirm if it is safe to return the new version. Eiger also needs to maintain a dependency tree in the client, which adds additional overhead in the client, that introduces even more latency. Finally, Wren presents a read latency that is higher than all the other systems because it always needs two rounds to offer causal consistent.

Looking at the latency of write operations, the tradeoff implemented by FastCCS becomes clearer. In order to favor read transactions, FastCCS sacrifices write latency, as the client waits for the update of the write transaction to be installed in all partitions. Thus, FastCCS is particularly well suited for read-heavy workloads, where slower write operations are not able to have a significant impact on the overall throughput significantly, as we will see next.



(a) AWS (absolute values)

(b) Simmulations (absolute values)

(c) AWS (normalized values)

(d) Simmulations (normalized values)

Figure 4.2: Read throughput as a function of Tx length: AWS vs simulations.

4.4 Throughput

This section analyzes the throughput achieved by the different systems. We measure throughput as the number of transactions executed by the client per second. We presents the results using absolute values and also results normalized with regard to the throughput achieved by the eventually consistent system; the later makes easier to assess the overhead incurred when one increase the consistency guarantees to TCC. Furthermore, in this section we present results obtained both using simulations and the real deployment on AWS. This allows us to assess how accurate is our simulation environment.

4.4.1 Effect of Tx Lenght on Read Transactions

Figure 4.2 presents the throughput of the different systems, when executing transactions with a write/read ratio of 0.05, as a function of the objects accessed in each *read* transaction. As in can be observed, FastCCS is able to closely follow the performance of system that only offers eventual consistency, with a penalty in the order of 10%. Both Eiger and Wren are much worse. For transactions that touch a small number of items, the number of communication rounds of each protocol dominates the performance and, therefore, Wren exhibits a poor throughput. However, for transactions that touch a lot of objects, the performance of Wren starts approximating the performance of other systems, even surpassing Eiger for long transactions. This is due to a combination of three factors.

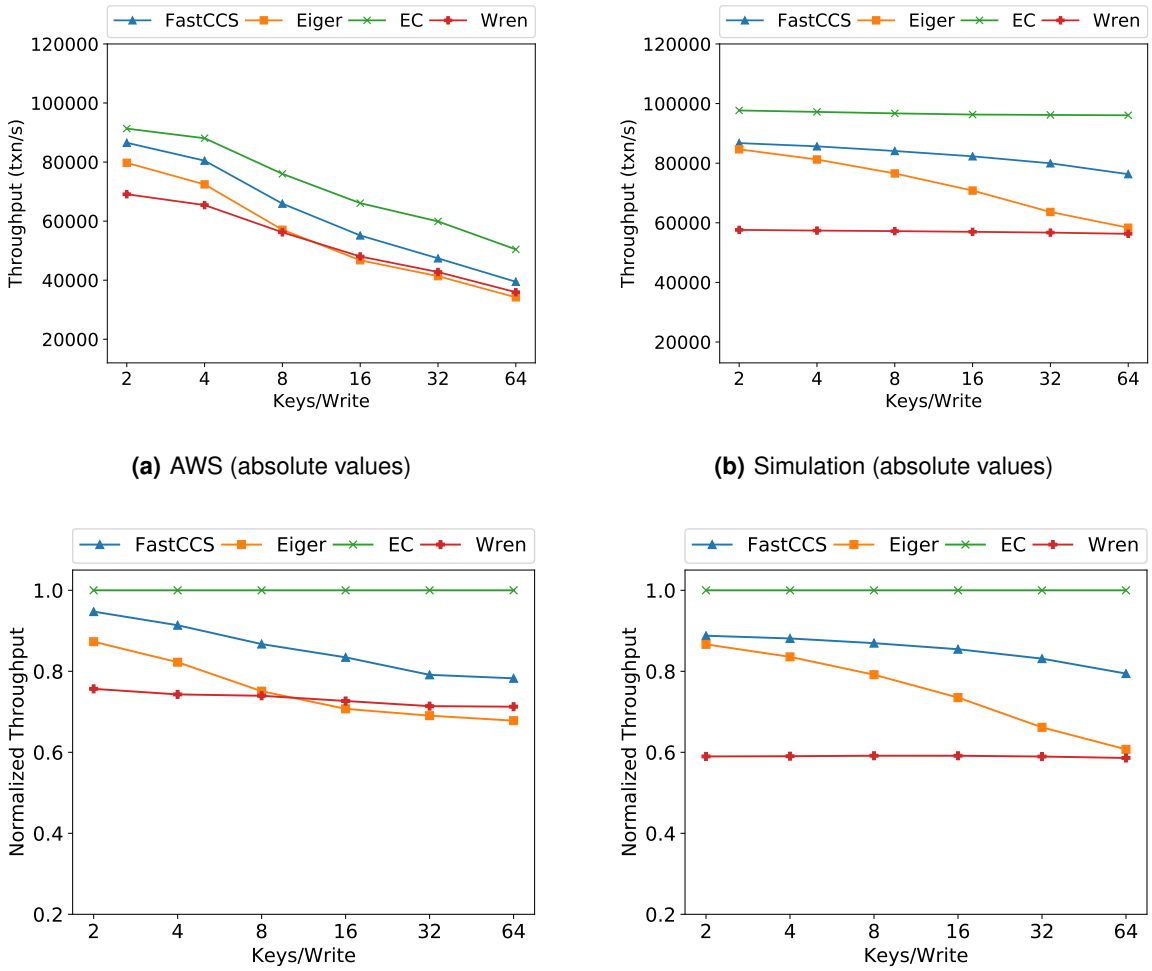
- First, a large number of keys is penalizing for Eiger. In fact, in Eiger, when more keys are accessed in the first round it becomes more likely that the first round will return an inconsistent view, which subsequently increases the likelihood of reading from a key that has a pending version, which subsequently increases the number of commit checks. Moreover, if the number of keys per request increases, the second round is more expensive as more keys will need to be returned in the second round of reads.

- Second, as the number of keys increase, the costs associated with CPU utilization and with disk access time start to become dominant, and the overhead of the additional round introduced by Wren becomes less relevant.

- Third, we have also noticed that as the read transaction size influences the number of dependencies in the client, and this dependency tree is only cleared when the client issues a write transaction. Moreover, as the client has a low probability of issuing a write transaction, the overhead of maintaining the dependency tree increases, which negatively influences the client's performance.

In the case of FastCCS, when the read transaction size increases, the second round also becomes more likely and more expensive as it needs to return more values. However, as the probability of the second round due to false positives is low, and there is no need for a third round, FastCCS is still able to offer overall better performance than Eiger.

The figures obtained with the simulated environment show that the simulator can capture with reasonable accuracy the performance of the different systems for small transactions. For large transactions, the simulator no longer provides an accurate estimate of the performance; this is due to the fact that the simulator is not able to simulate CPU or disk usage.



(a) AWS (absolute values)

(b) Simulation (absolute values)

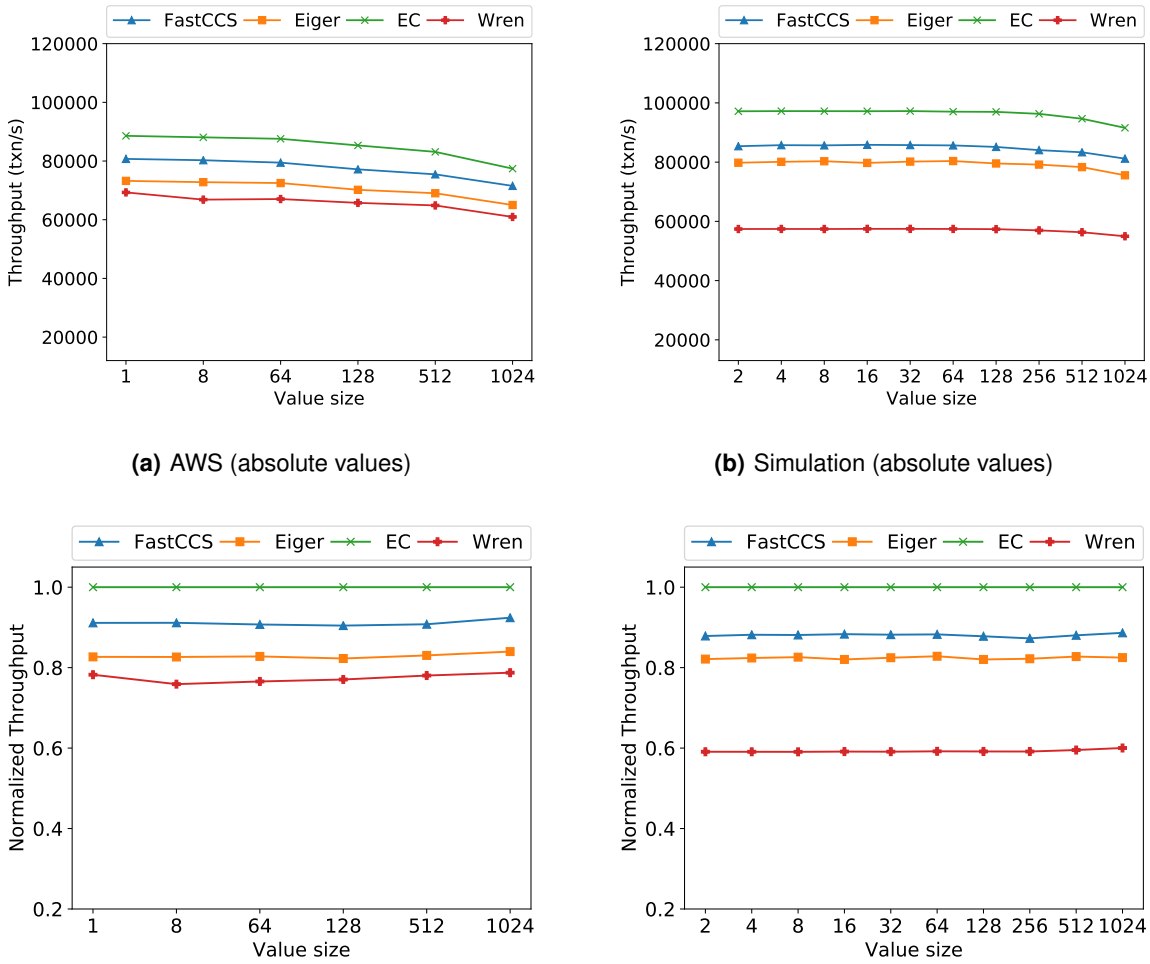
(c) AWS (normalized values)

(d) Simulation (normalized values)

Figure 4.3: Write throughput Tx length: AWS vs simulations.

4.4.2 Effect of Tx Lenght on Write Transactions

Figure 4.3 presents the throughput of the different systems, when executing transactions with a write/read ratio of 0.05, as a function of the objects accessed in each *write* transaction. We recall that FastCCS has been optimized for read transactions. Therefore, its performance on write transactions is substantially worse than that of a system that only offers eventual consistency. Nevertheless, FastCCS is still able to offer a better performance than Eiger and Wren. Note that when the size of write transactions increases, the probability of having more partitions participating in the transaction increases, leading to a higher probability of having the first round of reads to return inconsistent view. Moreover, the higher number of participating partitions drastically increases the number of commit checks that need to be performed, which further increases latency.



(a) AWS (absolute values)

(b) Simulation (absolute values)

(c) AWS (normalized values)

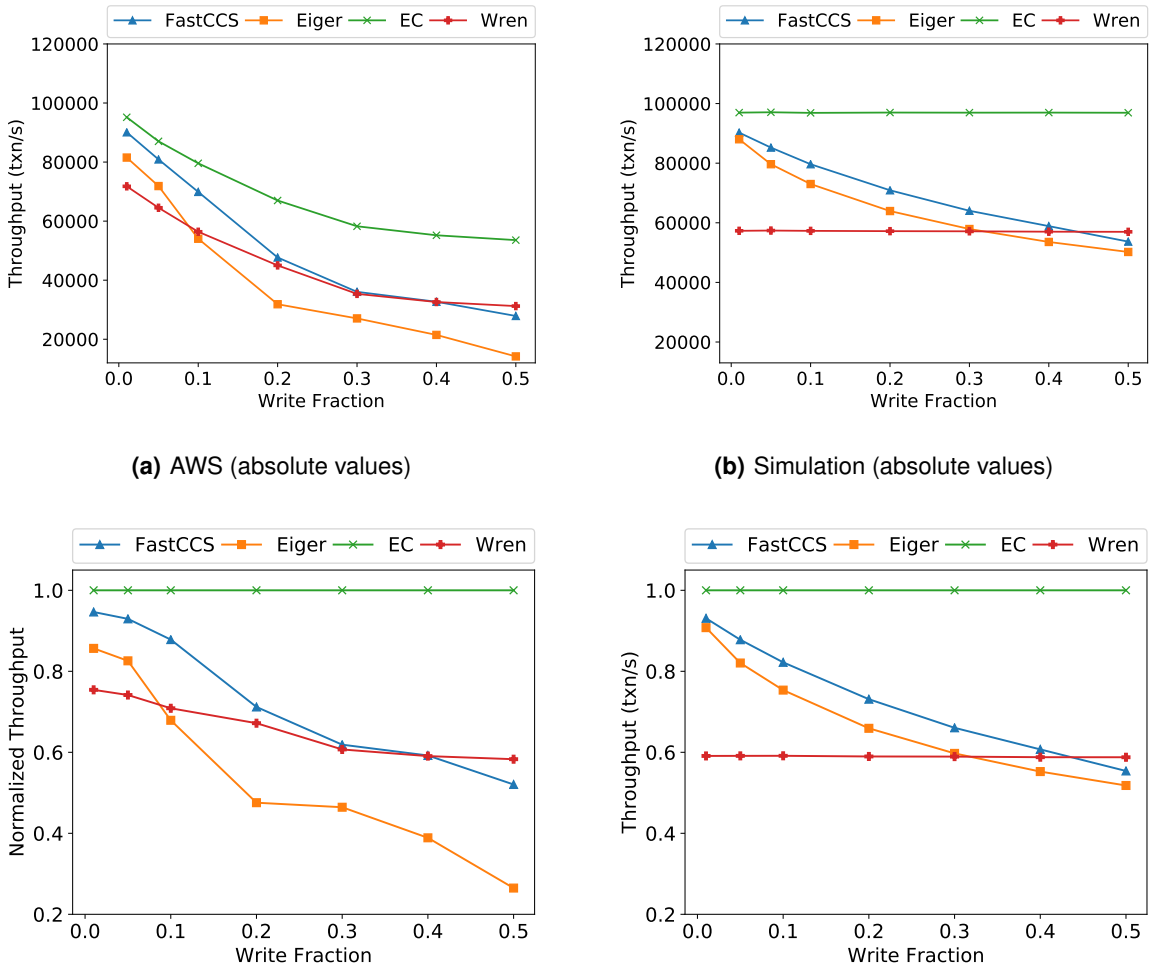
(d) Simulation (normalized values)

Figure 4.4: Read throughput as a function of the item size: AWS vs simulations.

When looking at the simulations results, it can be observed that the relative performance of Wren is again underestimated by the simulator. As before, the CPU and disk overheads, which are not accurately captured by the simulator, affect equally all systems, reducing the negative impacts that result from the coordination overhead.

4.4.3 Effect of the Item Size

Figure 4.4 presents the throughput of the different systems, when executing transactions with a write/read ratio of 0.05, as a function of the size of objects accessed in each transaction (all transactions access 5 objects). As in previous experiments, FastCCS is the system that better approximates the performance of a eventually consistent system. It is interesting to note that, for very large items, the costs involved



(a) AWS (absolute values)

(b) Simulation (absolute values)

(c) AWS (normalized values)

(d) Simulation (normalized values)

Figure 4.5: Throughput as a function of the write/read ratio: AWS vs simulations.

in the coordination become less relevant when compared with data transfer and disk access times. Therefore, for object sizes bigger than 512 bytes the performance of FastCCS, Eiger, and Wren start approximating the performance of an eventual consistent system. The results obtained with the simulation are roughly consistent with the results obtained with AWS although, as in the previous cases, the fact the simulator considers the communication overhead but not the disk access overhead, penalizes Wren (and, in a smaller extent, FastCCS) which perform comparatively better in practice than the simulations predict.

4.4.4 Effect of the Write/Read Ratio

Figure 4.5 presents the throughput of the different systems, when executing transactions with different write/read ratios (all transactions access 5 objects of 128 bytes).

The figure unveils an interesting limitation of Eiger. For residual write ratios (for instance, 0.01%), Eiger and FastCCS have almost the same number of second-round reads. Therefore, one could expect that both systems would exhibit the same performance, as predicted by our simulations. In reality, the throughput of Eiger is 10% lower that that of FastCCS in the AWS deployment. The cause for this difference lies in the way Eiger keeps dependencies in the client. Eiger maintains a tree of dependencies that can only be purged when a write transaction is executed. For small ratio of write transactions the dependency tree keeps increasing and the CPU utilization at the client becomes a bottleneck.

The figure also shows throughput of FastCCS decreases, as the percentage of write transaction increase. This is due to the higher write transaction cost and a higher probability of a second round. In fact, it is possible to observe that for large write/read ratios, Wren eventually outperforms the FastCCS. This happens because write operations terminate earlier in Wren. Interestingly, the simulations can also capture this fact, despite the limitations in accuracy previously discussed. Nevertheless, for most realistic write/read ratios, FastCCS outperforms both Eiger and Wren.

4.5 Scalability

The ability to distribute the data across different servers is essential for scalability. Thus, all cloud storage systems split the data into logical partitions and then let a different set of servers handle each partition. If a transaction accesses data that are in different partitions, coordination among different servers is required. This experiment focuses on understanding how the system can scale horizontally. We do so by studying the effect of the number of partitions on the system throughput.

Given that the number of communication rounds used by each protocol has a direct impact on the achievable throughput, we also show the average number of communication round used by the different systems. For these experiments, we have been able to deploy the system on AWS using up to 64 partitions. Unfortunately, we could not afford to run experiments on a real deployment using a larger number of machines, as this was outside our budget. Therefore, we resorted to simulations to estimate the performance of the system in scenarios that go up to 1024 machines.

4.5.1 Horizontal Scaling

We start by showing that FastCCS is able to horizontally scale and sustain additional clients as more servers are added to the system. For this experiment, we augmented the number of partitions and

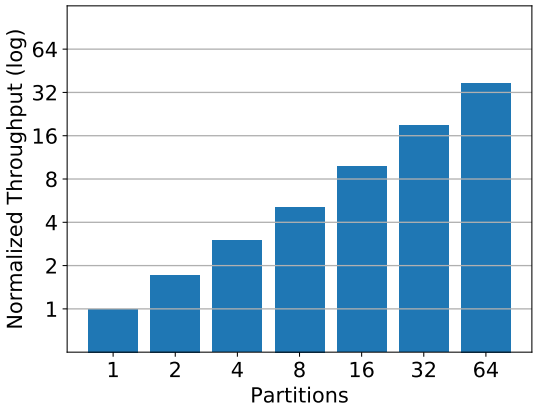


Figure 4.6: Normalized throughput of FastCCS changing the total number of Partitions (AWS) and proportionally changing the number of clients and keys. Bars are normalized against 1 partition.

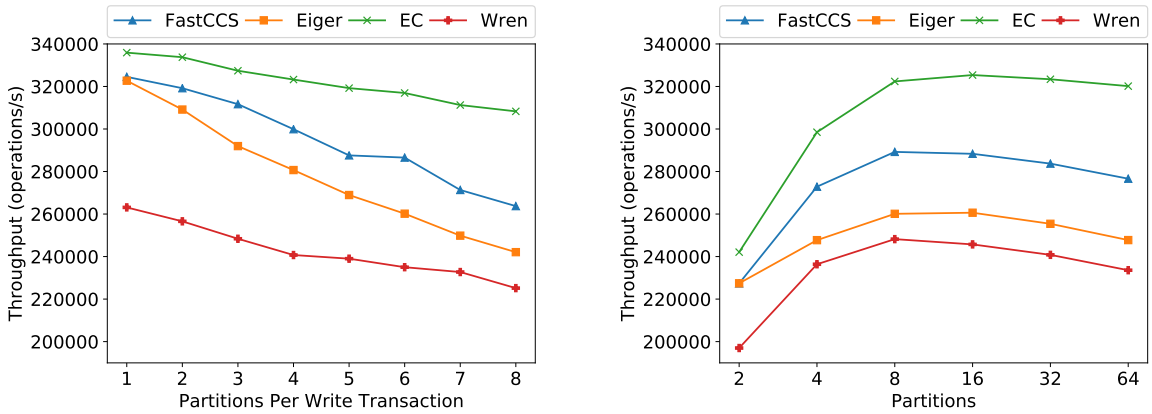
augmented proportionally the number of clients, such that the operations submitted by N client machines are fully loading N partitions. Moreover, we proportionally increase the number of keys to avoid any artifacts due to an increase in concurrency between clients. Transactions always access 5 objects selected at random.

Figure 4.6 shows the throughput for FastCCS as we scale the number of partitions from 1 to 64 (note that both axes are in log scale). The bars show the throughput normalized against the throughput of 1 partition. FastCCS scales out as the number of partitions increases. However, this increase is not linear from 1 to 8 partitions. The configuration with 1 partition has the benefits of batching: all operations that involve multiple keys are executed on a single machine. As the number of partitions increases, the transactions span across multiple partitions, and thus the system is no longer able to exploit batching effectively. This effect was also present in the original evaluation of Eiger [8].

As we increase the number of partitions, the differences due to lack of batching no longer become relevant. In fact, in a system with many partitions, most transactions tend always to access 5 different partitions. Nevertheless, the ability of the system to scale perfectly is limited due to a number of overheads that are associated with the maintenance of multiple partitions, such as background stabilization procedures or increased size of metadata. Next, we describe a number of experiments that provide some insights for the causes of the observed impairments to horizontal scaling.

4.5.2 Partition Overhead

In order to better understand the sources of overhead that become visible when many partitions are used, we have run a series of experiments where we increase the number of partitions while keeping the workload constant. Figure 4.7 shows the results obtained with the deployment at AWS. We have performed two experiments. In the first experiment (4.7(a)), we fixed the total number of partitions to



(a) Partitions per Write

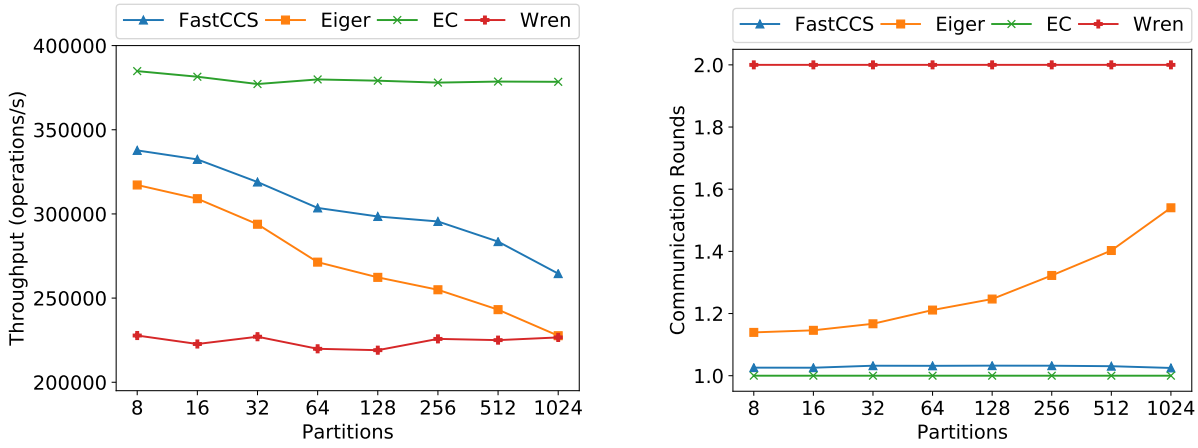
(b) Total Number of Partitions

Figure 4.7: Changing the Number of Partitions (AWS).

8, and we varied the number of partitions accessed by each transaction. In the second experiment (4.7(b)), transaction access 5 objects at random, and we have changed the total number of partitions (note that, the higher the total number of partitions is, the more likely is that a transaction touches 5 different partitions).

We start by discussing the effect on the number of partitions accessed by each transaction. Naturally, even the eventual consistent system degrades its performance as the number of partitions accessed by a given transaction increases, as more servers need to be contacted. This happens because Cassandra maintains a number of background bookkeeping tasks that become heavier with the number of nodes increase. Still, not surprisingly, the eventually consistent system is the one that is less affected by an increase in the number of partitions, and it does not require nodes to coordinate, and the amount of metadata maintained is minimal. The performance of FastCCS, Eiger, and Wren follow a similar trend when the number of partitions accessed by each transaction increases. Because these protocols require more coordination among partitions, the effect of using a larger number of partitions is more noticeable than in an eventually consistent system. For instance, with Eiger, because the metadata used to capture causal dependencies has low accuracy, the number of times the protocol requires a second or even a third round of communication. FastCCS has the same coordination overhead as Eiger but, due to more precise metadata, it is less likely to require a spurious second round of reads.

We now discuss the effect of changing the total number of partitions. In this experiment, we have kept the workload constant across all experiments. When there are few partitions, served do not have enough capacity to serve all the clients' requests. Therefore, the throughput of the system is limited by the lack of capacity of the servers. As we increase the number of partitions, we are able to distribute the load of the clients among different servers, and the throughput increases. This growth stops when the



(a) Throughput

(b) Average Number of Rounds

Figure 4.8: Changing the Number of Partitions (Simulations).

number of partitions reaches 16. In this scenario, the capacity of the servers is no longer a bottleneck, and adding more servers does not provide any help (we recall that the workload is fixed). From this point, we can start observing the effect of some amount of overhead that is induced by having a large number of partitions. This overhead as two main sources: one source are background activities, such as the stabilization protocol used by FastCCS and Wren, and another source is the increased size of the metadata (for instance, larger vector clocks). The combined effect of these two factors slows down the system when the workload remains constant (naturally, the larger system could sustain a higher peak workload, but this is not depicted in these plots). Note, however, that even the eventually consistent system experiences this effect, given that the underlying key value store also has some bookkeeping tasks whose overhead increases with the number of servers.

In order to estimate the performed of the system for a higher number of partitions, we needed to resort to simulations. The results are depicted in Figure 4.8. Note that, as before, we assume that the workload is fixed. Thus, the maximum throughput is bounded by the number of clients. Therefore, adding more partitions only increases the overhead generated by background tasks, such as the stabilization protocols, and larger metadata (i.e., larger clocks). The simulations show a trend that is aligned with the results obtained with the real experiment. However, as we have seen in previous experiments, the results from the simulator are an upper bound on the real performance, because the simulator is not considering CPU utilization, which also increases as the metadata increases.

We have also measured the average number of rounds required by read operations in the different systems, as the number of partitions grows (Figure 4.8(b)). This is interesting because it highlights that the overhead induced by a large number of participants manifests in different ways for different protocols. For Eiger, the loss of performance can be mainly attributed to the fact that the average number of rounds

increases with the number of partitions, which affects the throughput of the system. Thus, because Eiger uses less metadata than FastCCS, it needs to perform a second round more often. FastCCS, instead, does not suffer from this problem. Most reads can be performed in one round, no matter how large is the system. Unfortunately, this positive feature is obtained at the cost of using more metadata, which grows linearly with the system size. This amount of metadata also affects negatively the performance of the system (even if the results are still better than Eiger).

4.6 Discussion

The hypothesis that motivated the design of FastCCS was that the number of rounds required per read operation has a significant impact on the performance of systems offering TCC. Therefore we expected to outperform previous systems if we could reduce the average number of rounds required by read operations. Our experimental results broadly confirm that our hypothesis was correct. In FastCCS, most reads are performed in a single round, while in Eiger, depending on the scenarios, from 20% to 50% of the reads require more than one round, and with Wren, all reads require two rounds. Experimental results show that the resulting performance of the different systems is inversely proportional to the average number of rounds. FastCCS is the system that better approximates the performance of a weakly consistent system and, in some favorable cases, such has read-heavy scenarios, that are the most frequent in most cloud based applications, presents a small overhead, that can be as low as 6%. Other systems, and Wren in particular offer much larger penalties.

Our experimental evaluation also provides interesting insights into the effect of metadata size on the performance of systems that implement TCC. FastCCS is able to reduce the average number of rounds by using more metadata. The size of this metadata grows linearly with the number of partitions used in the system, and, for a large number of partitions, the cost of processing and transferring the metadata has a negative impact on performance. In our experiments, for most scenarios, FastCCS was still able to outperform other solutions despite this drawback. However, this suggests that further research is needed to understand what is the best tradeoff between metadata size and the number of rounds.

Interestingly, our evaluation has also unveiled that the amount of metadata keep at the client proxy, even if it is not exchanged with a server, can also be an impairment to performance. Eiger maintains at each client a dependency tree to ensure that the values returned in the first round of read are causal consistent with the causal past of the client. The cost of maintaining this tree can be large and negatively impacts Eiger performance in read-heavy scenarios. Highlighting the tradeoff between the amount of metadata and the final performance is quite subtle in real systems, due to the conflicting forces involved.

Summary

In this chapter, we have presented the experimental evaluation of FastCCS, detailing the tests that were conducted in order to assess its performance. The results show that FastCCS has the potential to reduce latency compared to other systems in the literature with little overhead (5%) compared with eventual consistent systems. This section also discussed some limitations of current TCC implementations and FastCCS limitations that may justify the observed results. The next chapter ends this thesis by reporting the most important findings, as well as sharing some ideas for future work.

5

Conclusion

Contents

5.1 Conclusions

Stronger consistency models ease the life of programmers, making it more easy to reason about. However, as stated by Amazon and Google, the increase in user perceived latency leads to a concrete revenue loss. For example, Amazon estimates that a 100ms latency increase leads to 1% revenue loss [34]. So it is important to find consistency models that can offer low latency. This thesis has described the design, implementation, and evaluation of FastCCS. By using more precise metadata, we showed that FastCCS reduces the number of communication rounds needed to implement TCC. In fact, to our knowledge, FastCCS is the first system that implements TCC with at most two rounds of communication, reducing the overall latency that the client experiences.

An experimental evaluation compares FastCCS with other two state of the art solutions, as well as an eventual consistent system. When compared with other solutions that implement TCC, FastCCS outperforms all the other system, that confirms that a lower number of rounds positively influence the overall throughput of the system.

5.2 System Limitations and Future Work

Although this work was focused on experiments within one datacenter, in a real setting, partitions would be replicated across multiple datacenters. It would be interesting to develop FastCCS further to introduce replication or even partial replication and to study what modification would be necessary to satisfy thous requirements. Finally, as mentioned in Section 4.5.2, the increase of the metadata, limits in some ways scalability by partitioning. It would be interesting to study the effects of vector clock compression and how it would affect the overall performance of the system.

Bibliography

[1] P. A. Bernstein, P. A. Bernstein, and N. Goodman, “Concurrency control in distributed database systems,” *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, 1981.

[2] C. H. Papadimitriou, “Serializability of concurrent database updates,” *Journal of the ACM*, vol. 26, no. 4, 1979.

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Highly available transactions: Virtues and limitations,” in *Proc. of the 39th International Conference on Very Large Data Bases*, Trento, Italy, Aug. 2013.

[4] W. Vogels, “Eventually consistent,” *Communications of the ACM*, vol. 52, no. 1, 2009.

[5] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, “Cure: Strong semantics meets high availability and low latency,” in *Proc. of the 36th IEEE International Conference on Distributed Computing Systems*, Nara, Japan, Jun. 2016.

[6] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, “Challenges to adopting stronger consistency at scale,” in *Proc. of 15th Workshop on Hot Topics in Operating Systems*, Kartause Ittingen, Switzerland, May 2015.

[7] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, “The SNOW theorem and latency-optimal read-only transactions,” in *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, Nov. 2016.

[8] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Stronger semantics for low-latency geo-replicated storage,” in *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, IL, Apr. 2013.

[9] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, 1990.

[10] L. Lamport, “On interprocess communication,” *Distributed Computing*, vol. 1, no. 2, 1986.

[11] ——, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, 1978.

[12] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, 1995.

[13] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak, “From session causality to causal consistency,” in *Proc. of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Coruña, Spain, Feb. 2004.

[14] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops,” in *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011.

[15] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ansi sql isolation levels,” in *Proc. of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

[16] S. Das, D. Agrawal, and A. El Abbadi, “G-store: A scalable data store for transactional multi key access in the cloud,” in *Proc. of the 1st ACM Symposium on Cloud Computing*, Indianapolis, IN, Jun. 2010.

[17] N. Preguica, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro, “Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine,” in *Proc. of the 33rd IEEE International Symposium on Reliable Distributed Systems Workshops*, Nara, Japan, Oct. 2014.

[18] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, “Orbe: Scalable causal consistency using dependency matrices and physical clocks,” in *Proc. of the 4th ACM Annual Symposium on Cloud Computing*, San Jose, CA, Oct. 2013.

[19] S. Almeida, J. a. Leitão, and L. Rodrigues, “Chainreaction: A causal+ consistent datastore based on chain replication,” in *Proc. of the 8th ACM European Conference on Computer Systems*, Prague, Czech Republic, Apr. 2013.

[20] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, “Gentlerain: Cheap and scalable causal consistency with physical clocks,” in *Proc. of the 5th ACM Annual Symposium on Cloud Computing*, Seattle, WA, Nov. 2014.

[21] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Scalable atomic visibility with RAMP transactions,” *ACM Transactions on Database Systems*, vol. 41, no. 3, 2016.

[22] J. Du, S. Elnikety, and W. Zwaenepoel, “Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks,” in *Proc. of the 32nd IEEE International Symposium on Reliable Distributed Systems*, Braga, Portugal, Oct. 2013.

[23] M. K. Aguilera, J. B. Leners, and M. Walfish, “Yesquel: Scalable sql storage for web applications,” in *Proc. of the 25th Symposium on Operating Systems Principles*, Monterey, CA, Oct. 2015.

[24] R. Van Renesse and F. B. Schneider, “Chain replication for supporting high throughput and availability,” in *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[25] K. Spirovska, D. Didona, and W. Zwaenepoel, “Wren: Nonblocking reads in a partitioned transactional causally consistent data store,” in *Proc. of the 48th IEEE International Conference on Dependable Systems and Networks*, Luxembourg City, Luxembourg, Jun. 2018.

[26] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, “TAO: Facebook’s distributed data store for the social graph,” in *Proc. of the USENIX Annual Technical Conference*, San Jose, CA, Jun. 2013.

[27] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, “Challenges to adopting stronger consistency at scale,” in *Proc. of the 15th USENIX Workshop on Hot Topics in Operating Systems*, Kartause Ittingen, Switzerland, May 2015.

[28] D. Didona, P. Fatourou, R. Guerraoui, J. Wang, and W. Zwaenepoel, “Distributed transactional systems cannot be fast,” in *Proc. of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, Phoenix, AZ, Jun. 2019.

[29] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” in *Proc. of the 3rd ACM Workshop on Large Scale Distributed Systems and Middleware*, Big Sky, MT, Oct. 2009.

[30] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, no. 2, 1998.

[31] Amazon, “Amazon Elastic Compute Cloud (EC2),” https://aws.amazon.com/ec2/.

[32] A. Montresor and M. Jelasity, “PeerSim: A scalable P2P simulator,” in *Proc. of the 9th International Conference on Peer-to-Peer*, Seattle, WA, Sep. 2009.

[33] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Highly available transactions: Virtues and limitations,” in *Proc. of the 39th International Conference on Very Large Data Bases*, Trento, Italy, Aug. 2013.

[34] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie, “How far can client-only solutions go for mobile browser speed?” in *Proc. of the 21st International Conference on World Wide Web*, Lyon, France, Apr. 2012.