Geo-Replication in Large-Scale Cloud Computing Applications

Sérgio Garrau Almeida
sergio.garrau@ist.utl.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Cloud computing applications have acquired a significant importance in the last years. Therefore, this report addresses the problems of supporting consistency and fault-tolerance in large-scale Geo-replicated systems. We are particularly interested in applications built using the cloud-computing paradigm, where data is maintained in multiple datacenters, located in geographically distant regions. Examples of such applications are Facebook, Google Apps, among others. In such systems, data is replicated in different datacenters but also internally to each datacenter, in different nodes. The report discusses the different tradeoffs among consistency, reliability, availability, and performance when implementing these systems and pinpoints some possible directions to improve the state of the art in this respect.

1 Introduction

Large-scale applications must deal with the problem of keeping multiple copies of the same data, stored in different geographic locations, and maintaining them consistent in face of concurrent access by a potentially large number of users. The problem of replication in such setting is often called *Geo-replication*; it appears with particular emphasis in the context of applications developed according to the cloud computing paradigm.

There are many examples of applications that benefit from Geo-replication. For instance, Amazon provides services to clients located anywhere in the world, using multiple datacenters located in different regions. Also Facebook and Google have similar setups, serving a very large (and growing) user base. In order to support this kind of business scale, these systems have to combine performance, high availability, and scalability, something that is not trivial to achieve in such a large-scale scenario.

Ideally, the runtime middleware support for these systems would hide from programmers, issues such as distribution, replication, and the occurrence of node or network failures. As a result, data would appear to be accessed locally, as if it was stored in the same machine where the application is launched. Furthermore, data access by one application would be isolated from concurrent access to the same data by other applications (or other instances of the same application). Therefore, there are two main features that cloud computing platforms must support: concurrency control and replication management. We briefly introduce these two aspects in the next paragraphs.

1.1 Concurrency Control

Two operations on a given object execute in serial order if one starts after the other terminates. Otherwise, the operations are said to execute concurrently. Although the outcome of a serial execution of operations is often clear to the programmer, however the result of a concurrent execution may not be intuitive. For instance, consider an empty FIFO queue object and a method *insert*. If "INSERT(b)" is executed *after* "INSERT(a)", it is clear that '*a*' should be at the head of the queue and '*b*' at the tail. But what are the legal outcomes when both operations are executed concurrently?

The problem is exacerbated when we consider sequences of operations, typically known as *transactions*. The concurrent execution of two transactions may yield many different interleavings of their individual operations. Many of these interleavings may result in outcomes that are not intended by the programmer of the transactional code (which assumed that her code would be executed in isolation or could not foresee all possible interleavings).

One way to avoid the problems above is to avoid concurrency, by forcing all operations and transactions to execute in serial order. Unfortunately this strategy may result in extremely long waiting times when the system is subject to high loads. Furthermore, it would prevent the infrastructure from fully using existing resources (for instance, by preventing operations or transactions from running in parallel in different cores or machines). This is unacceptable, in particular because many operations access independent data structures and so, their concurrent execution does not mutually interfere. Therefore, any practical system must support concurrent execution of operations and implement mechanisms to ensure that, despite concurrency, operations still return the "expected" outcomes.

A *data consistency model* defines which outcomes of a concurrent execution are deemed correct and which outcomes are deemed incorrect. The mechanisms required to prevent incorrect outcomes are known as *concurrency control mechanisms*. As we have noted before, it is often intuitive to programmers and users to consider that the system behaves *as if* all operations and/or transactions were executed in *some* serial order. This has led to the definition of data consistency models such as *linearizability* (for isolated operations) and *serializability* (for transactions). These models offer what is often called *strong consistency*. We will describe these models in detail later in the text.

Unfortunately, as we will see, implementing strong consistency in Geo-replicated systems is very expensive, as it requires tight coordination among datacenters; the required coordination is slowed down by the large network latencies observed in the *Wide Area Networks* (WANs) that connect different locations. Therefore, one has often to impose on the programmers (and users) more relaxed consistency models, that allow for more legal outcomes (even if not as intuitive) at a smaller cost.

1.2 Replication Management

Replication serves multiple purposes in a large-scale system. On one hand, it allows to distribute the load of read-only operations among multiple replicas, at the cost of requiring inter-replica coordination when updates occur. Since many workloads are read dominated, replication often pays off from the performance point of view. On the other hand, replication may increase the reliability and availability of data. If a given machine fails, data may survive if it is also stored on a different node. If data is replicated on different datacenters, it may even provide tolerance to catastrophic events that may destroy an entire datacenter (such as an earthquake, fire, etc).

Additionally, by keeping multiple copies of the data in different locations, one also increases the probability that at least one of these copies is reachable in face of failures in networking components that may, for instance, partition the network. Given that users may access data from different locations, replication allows to place data copies closer to the users, and allows them to access data with smaller latencies. It is therefore no surprise that replication is an important component of cloud computing.

The main challenge when using replication is how to maintain the replicas consistent. Again, as with concurrency, ideally it would be possible to hide replication from the programmers. In that case, the access to a replicated data item would be similar to the access to a non-replicated item, something known as *one-copy equivalence* [1]. Naturally, this requires that copies coordinate during write (and possibly read) operations. A fundamental coordination abstraction is *Distributed Consensus* [2], that allows to implement *state-machine replication* [3], a well-established strategy to keep replicas consistent.

As noted before, an essential characteristic of Geo-replicated systems is that network connections among datacenters are characterized by high latencies. This makes coordination across datacenters very expensive. Therefore, to operate with acceptable latencies, a system has to provide weaker guarantees than one-copy equivalence. Furthermore, wide area networks are subject to network partitions. In a partitioned network, replicas in different partitions cannot coordinate, and may diverge if updates are allowed. The observation of this fact has motivated the *CAP theorem* [4], that states that it is sometimes impossible to offer both consistency, availability, and partition tolerance. The problems underlying the CAP problem are thus another reason to offer weaker consistency guarantees in geo-replicated systems.

1.3 Putting Everything Together

The use of replication and the support for concurrency brings several benefits to cloud computing applications. This strongly suggests that a Geo-distributed system must provide both features. On the other hand, the challenges introduced by high-latencies in communication between datacenters and the occurrence of network partitions, make it difficult to support strong consistency and one-copy equivalence. Unfortunately, the use of weaker consistency models makes programming harder, as the allowed set of outcomes is larger, forcing the programmer to take into account a wider set of legal executions.

Moreover, the systems that are built to operate on a geo-distributed scenario usually need to support a large number of users. These users combined issue thousands of requests per second that must be answered with a reasonable short delay in order to provide a good user experience. This stresses the infrastructure to operate at a very large scale while providing low latency operations. These applications must rely on an infrastructure that allows them to manipulate the data with such low latencies. In order to satisfy these requirements, the infrastructure must employ very efficient algorithms, often resorting to the weakening of consistency guarantees.

Additionally, applications are willing to trade consistency for performance because any increase in the latency of operations could mean the loss of a percentage of their users. Therefore, a key aspect in the development of cloud-computing infrastructures is to find data storage protocols that offer a good trade-off between efficiency and consistency. This is a very active area of research, to which this work aims at contributing.

1.4 Roadmap

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present all the background related with our work. Section 4 describes the proposed architecture to be implemented and Section 5 describes how we plan to evaluate our results. Finally, Section 6 presents the schedule of future work and Section 7 concludes the report.

2 Goals

This work addresses the problem of providing geo-replication mechanisms for large-scale cloud computing applications. More precisely, we aim at:

> *Goals:* Designing and implementing a novel Geo-replication protocol that offers some useful guarantees to the programmers while maintaining the scalability and throughput required by large-scale applications.

As will be described further ahead, our protocol will be based in existing replication approaches, in order to capture the best of each one. According to the strict requirements imposed by this type of applications, the consistency model that is going to be supported must be chosen with care. We plan to perform an experimental evaluation of the protocol in order to assess the suitability of our approach to cloud-computing environments.

> *Expected results:* The work will produce i) a specification of the protocol for a single site and for a Geo-replicated scenario; ii) an implementation of both settings above, iii) an extensive experimental evaluation

using simulations. If time allows we would like to run experiments using a cluster infrastructure available at INESC-ID, with real code and dedicated load generators.

3 Related Work

This section surveys the main concepts and systems that are relevant for our work. We start by surveying different data consistency models. Then we briefly introduce the main concurrency control techniques and the main data replication techniques. Finally, we describe a number of relevant systems that implement different variants and combinations of these models and techniques.

3.1 Data Objects and Conflict Resolution

We consider that the goal of a datastore system is to store objects in a persistent way, in which objects are replicated and maintained at multiple nodes. The most simple objects are *registers*, that only export READ and WRITE operations. In a register, any update operation (i.e., a WRITE) always overwrites the effect of the previous update. However, in some cases, the system also supports higher level objects with complex semantics. For instance, consider a *set* object, where items can be added using a ADD operation. In this case, the ADD operation updates the state of the object without overwriting the previous update.

As will be discussed further ahead in the report, some consistency models allow concurrent update operations to be executed in parallel at different replicas. As a result, replicas will have an inconsistent state if operations are executed in an uncoordinated manner. The procedure to reconcile the state of different replicas is often called *conflict resolution*. Conflict resolution may be *automatically* handled by the system or it may require the intervention of some external component. In the latter case the system may only detect the conflict and handle the different states to the external component that will be in charge of generating the new reconciled state of the object (ultimately, this may require manual intervention from the end-user).

Automatic conflict resolution is, in the general case, impossible. The most common way of performing automatic conflict resolution is to use the last-writer-wins rule [5]. This rule states that two concurrent writes are ordered, giving the impression that one happened before the other (the last write overwrites the value written by the first). This policy is reasonable for a register object but may yield unsatisfactory results for more complex objects. For instance, consider again the *set* object. Consider a set that is initially empty and two concurrent updates add different items (say 'a' and 'b') to the set. If the set is seen as black box, the last-writer-wins rule would result in a set with only one item (albeit the same in all the replicas). However, if some operation semantics aware scheme exists, it could recognize that some operations are commutative, and merge the operations in a way that results in the same final state. Therefore, the ADD operation would be recognized as commutative.

3.2 Data Consistency Models

The following sections describe various consistency models that are relevant to this work. We start by discussing models that do not consider transactions, i.e., where no isolation guarantees are provided to sequences of operations. Instead, the first models only define which outcomes of individual operations are legal in face of a given interleaving of individual operations executed by different threads. Consistency models for transactional systems are surveyed in the subsequent section. Models are described by showing the outcomes of the operations that are legal under each model. Unless otherwise explicitly stated we assume that the shared data items are simple *registers*.

3.3 Non-Transactional Data Consistency Models

3.3.1 Linearizability The Linearizability or Atomic Consistency model, introduced in [6], provides the strongest guarantees to applications. As a matter of fact, this consistency model is the one that is closer to the idealized abstraction of memory where each operation occurs instantaneously and atomically. Moreover, the model assumes that operations may take some time to execute: their duration corresponds to the interval of time between its invocation and completion. However, the system behaves as if the operation took effect instantaneously, in an atomic moment within this interval.

This means that the results of a WRITE operation are necessarily visible to all other READ operations, at most when the WRITE completes. Also, if a WRITE operation is concurrent with a READ operation, the outcome of the WRITE may or may not be visible to READ. However, if it becomes visible, then it will be visible to all other read operations that are linearized after READ. This is illustrated in Figure 1a.

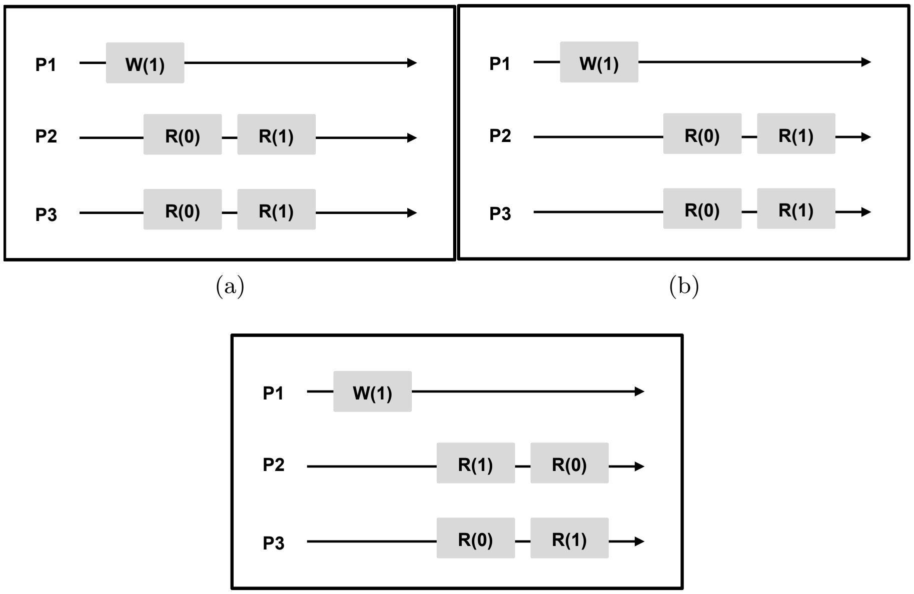
Furthermore, linearizability has a property known as composability (or locality). The latter states that a system is linearizable if it is linearizable with respect to each object. This means that a complex system built by composing several linearizable objects is also linearizable.

Unfortunately, to build a replicated fault-tolerant system that is linearizable is very expensive. One way to enforce linearizability is to resort to locking to perform writes, preventing concurrent reads while a write is in progress. This may avoid non-linearizable runs but may lead to poor performance and create unavailability problems in the presence of faults. Non-blocking algorithms exist to implement linearizable registers, but most of these algorithms require the reader to write-back the value read, effectively making every read as expensive as a write.

3.3.2 Sequential Consistency Sequential consistency [7] states that the execution of multiple threads should be equivalent to the execution of a single thread which combines all the operations of the multiple threads in some serial interleaving, as long as this interleaving respects the partial order defined by each individual thread (i.e, the serial order must respect the order of the operations as they were defined in the program).

Unlike linearizability, sequential consistency is not composable. A system where operations, made over an individual object, respect the local serial consistency is not necessarily globally sequentially consistent.

Sequential consistency is weaker than linearizability, as it does not require the outcome of a write operation to become immediately visible to other read operations (i.e., read operations may be serialized "in the past"), as can be seen on Figure 1b. Still, it requires operations to be totally ordered, which is also very expensive in a geo-replicated system.



(a) (b)

(c)

Fig. 1: Sequences of operations that are correct (assuming that the initial value of the register is 0) under: a) Linearizability; b) Sequential Consistency (not linearizable); c) Eventual Consistency (Neither linearizable nor sequentially consistent).

3.3.3 Per-record Timeline Consistency Per-record timeline consistency [8] is a relaxation of sequential consistency that implicitly makes the existence of replication visible to the programmer/user. In the first place, it only ensures sequential consistency on a per object basis. In this model, the updates made to an object have a single ordering, according to a certain timeline, that is shared by all its replicas (similar to sequential consistency). This ordering is imposed by a single replica that works as a serialization point for the updates.

This model also guarantees that the versions of the object present in a certain replica always move forward in this timeline. This way, read operations, made under this model, will return a consistent version from the ordering described above. However, and unlike sequential consistency, a read from the same thread may move back in the timeline with respect to previous reads. This may happen if the thread needs to access a different replica.

The rationale for this model is that most applications access a single object and often remain connected to the same datacenter. Therefore, in most cases users will not observe the "anomalies" allowed by the model. On the other hand, the weakening of the consistency model and the absence of conflicts (i.e. updates are serialized at a single node) allows for updates to be propagated in a lazy fashion among replicas.

3.3.4 Causal Consistency The causal consistency model [9] ensures that the observed outcomes of operations are always consistent with the "happened-before" partial order as defined by Lamport [10]. For completeness, we reproduce here the definition of the happened before order:

- Considering that A and B are two operations that are executed by the same process, if A was executed before B then they are causally related. There is a causal order between A and B , where *A happens before B*.

- If A is a write operation and B is a read operation that returns the value written by A, where A and B can be executed at different processes, then A happens before B in the causal order.

- Causal relations are transitive. Considering that A, B and C are operations, if A happens before B and B happens before C, then A happens before C.

Note that, as with sequential consistency, threads are not required to read the last value written, as read operations can be ordered in the past, as shown in Figure 2. Furthermore, and contrary to sequential consistency, concurrent writes (i.e., two writes that are not causally related) may be observed by different threads in different orders.

3.3.5 Eventual Consistency Eventual consistency [11, 12] is a term used to designate any consistency model that allows different threads to observe the results of update operations in different orders during some (possibly long) period. Although all threads are guaranteed to eventually read the same values, if write operations stop being executed. The time interval between the last write and the point when all replicas see that write is called *inconsistency window.* The length of this window depends on various factors like communication delays, failures, load of the system, network partitions, and *churn*. The latter refers to the phenomenon that happens when there are nodes joining and leaving the network at a very high rate. An example of a sequence of operations that is acceptable under eventual consistency can be observed in Figure 1c.

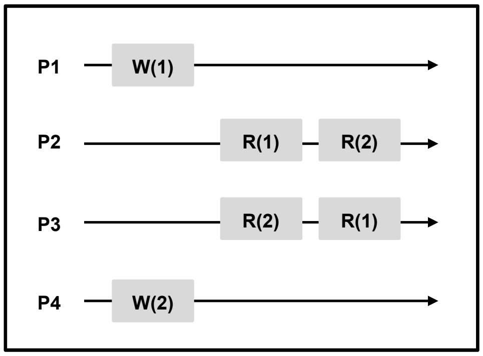


Fig. 2: Sequence of operations that are correct (assuming that the initial value is 0) under Causal Consistency (and Eventual Consistency), although they are neither linearizable nor sequentially consistent.

3.3.6 Causal+

Causal consistency guarantees that the values returned by read operations are correct according to the causal order between all operations. However, it offers no guarantees on the ordering of concurrent operations. This leads to scenario where replicas may diverge forever [9]. This happens in face of conflicting writes, which can be applied in different orders at different nodes.

Causal+ consistency appears as a model that aims at overcoming the drawback described above by adding the property of *convergent conflict handling*. This model has been suggested in a number of different papers [13, 14] and later detailed in [15].

In this consistency model, the enforcing of the convergence property can be made by using the last-writer-wins rule, as described before. Additionally, it allows semantic aware re-conciliation of inconsistent replicas, in which operations would be recognized as commutative.

3.4 Transactional Data Consistency Models

The consistency models listed previously consider operations in isolation. Transactional data consistency models consider *sequences* of operations that must be treated as a unit. These sequences of operations are named *transactions*, and are typically delimited by special operations, namely, beginTransaction and endTransaction. Transactions often serve a dual purpose: they are a unit of concurrency control and they are a unit of fault-tolerance.

For concurrency control purposes, a transactional system makes an attempt to isolate transactions from each other. As noted earlier in the text, the stronger consistency models ensure that the concurrent execution of transactions is the same as if they were executed one after the other (i.e., in a serial execution).

For fault-tolerance, transactions are a unit of *atomicity*, i.e., either all operations that constitute the transaction take effect or none does. In the former case, one says that the transaction has *committed*. In the latter case, one say that the transaction has *aborted*. A consistency model for a transactional system defines which are the valid outcomes of committed transactions.

3.4.1 Serializability Serializability [16] states that the outcome of a concurrent execution of a set of transactions must be the same as the outcomes of some serial execution of the same transactions. Moreover, the transactions issued by a process, which are included in this ordering, must respect the order defined in the program. A concurrent execution of transactions in a replicated system is serializable if the outcome is equivalent to that of some serial execution of the same transactions in a single replica.

Similarly to sequential consistency, serializability also requires concurrent transactions to be totally ordered, and therefore, consensus in a replicated system. Given the arguments captured by the CAP theorem [4], it is impossible to implement a geo-replicated system that combines consistency, availability, or even scalability under this model.

3.4.2 Snapshot Isolation Snapshot Isolation (SI) [17] is a weaker model of consistency that has been widely adopted by most database vendors (e.g. Oracle and SQLServer). According to this model, the state of the system at a given point in time includes all the updates of all transactions that have committed, as if these updates have been applied in some serial order. Furthermore, the outcome of each individual transaction is computed based on a valid system state, at some previous point in time (i.e., the outcome of a transaction never depends on partial results from on-going transactions or from updates from transactions that later abort). Furthermore, if two transactions update a common variable, one transaction needs to be executed based on a system state that already includes the outcome of the other. However, unlike with serializability, the state observed by a transaction does not necessarily reflect the updates of all transactions that have been totally ordered in the past.

The difference between snapshot isolation and serializability may be subtle and is better illustrated with an example. Consider the following execution: two transactions concurrently read an overlapping set of values from the same system state and make *disjoint* updates to different subsets of these values. Accordingly to snapshot isolation, these transactions are allowed to commit (and their updates are applied using some serial order). However, they will commit without seeing each other's updates. This "anomaly" (with regard to the idealized system) is characteristic of snapshot isolation and it is called *write skew.*

This particular example shows that SI is weaker than Serializability, as this write skew could never happen in a serializable execution.

3.4.3 Parallel Snapshot Isolation Although weaker than serializability, snapshot isolation still requires that write-write conflicts are detected. Therefore, in a replicated system, a transaction cannot be committed without previous coordination among replicas. Recent work introduced a variation of this model, named Parallel Snapshot Isolation [18]. This model consists of a relaxation of snapshot isolation aimed at geo-replicated systems that allows transactions to be applied in different orders at different nodes.

In PSI, each transaction is executed and committed at one site but different transactions may be executed in different sites. Therefore, under this model we can have concurrent transactions executing at the same site and concurrent transactions executing at different sites. Concurrency control operates differently in each of these cases.

In the case where concurrent transactions execute at the same site and there is a write-write conflict between them, then one of the transactions is required to abort. The decision on the transaction to be aborted depends on when the conflict is detected.

- If the conflict is detected when one of the transactions has already committed, the other must abort.
- If the conflict is detected and none of the transactions has committed yet, then one or both transactions may be required to abort, given that the decision policy is not deterministic.

On the other hand, if the conflicting transactions execute at different sites there are scenarios where both can commit. More precisely:

- If the conflict is detected when one of the transactions has already committed and the other has not, then the latter must be aborted.
- If the conflict is detected and both transactions have committed (in different sites), then the transactions must be ordered according to some criteria.
- If no conflict is detected, then both are committed and their updates applied in the same order at all sites.

Unfortunately, when conflicting transactions execute at different nodes their outcome is unpredictable as it depends on runtime conditions and implementation decisions related on how and when the system decides to propagate information about on-going and committed transactions among the different replicas.

3.5 Concurrency Control Mechanisms

In order to allow concurrent access by different threads, a system must provide some mechanism to coordinate them. These coordination mechanisms can be of two types: *pessimistic* or *optimistic*.

The most common way to implement pessimistic concurrency control is through the use of locking. In this approach all operations made over an object require the acquisition of a lock to the object. This grants to the thread exclusive access to a certain object, blocking the other threads that are trying to access that object. Unfortunately, the process of acquiring a lock can lead to a situation of *deadlock* in which all processes are blocked. This problem is aggravated when in distributed settings where nodes can fail, leaving objects locked. The latter issue is usually due to the complexity associated with the use of locks in programs. In order to avoid deadlocks, a programmer has to verify all possible scenarios for the execution, ensuring that two or more threads are not waiting for each other to release a certain object (i.e., all threads are ensured to progress in the execution).

On the other hand, optimistic concurrency control offers a better utilization of the computational resources by allowing the execution of concurrent operations over the same objects without requiring the usage of locking mechanisms. This concurrency control model is usually related with transactions and in order to achieve the consistency guarantees of serializability, the threads must check if there are no conflicts between the issued transactions. As described before, a conflict occurs in the two following situations: i) if two concurrent transactions write the same object (write-write conflict); ii) one transaction reads an object that was written by a concurrent transaction (read-write conflict). If there is a conflict, then one of the transactions must rollback its operations (i.e., it is aborted). Although, the transaction that was aborted is repeated later. Unfortunately, if conflicts happen frequently, the repetition of aborted transactions hinders the performance of the system (i.e., computational resources are wasted in executing operations that will be discarded). Therefore, in optimistic concurrency control, it is assumed that different threads will perform operations that do not affect each other.

3.6 Data Replication Approaches

The benefits of replication, as described previously, led to the appearance of a large number of replication approaches. The replication mechanisms that we consider relevant for the work being developed are the following: *active replication*, *primary-backup replication*, and *multi-master replication*.

Active replication: In active replication, requests made to the system are processed by all replicas. This requires that processes execute operations in a deterministic and synchronized way, receiving the same sequence of operations. In order to achieve the previous mechanism, the system must make use of an *atomic broadcast* protocol that guarantees that all replicas receive the messages in the same order and if one process receives the message then all the others receive it. The use of atomic broadcast hinders the scalability of the system since it must use a distributed consensus mechanism [2], which requires heavy communication.

Unlike the latter, the primary-backup and multi-master mechanisms assume that operations are made over a single replica. These two solutions are based on the read-one-write-all-available (ROWAA) approach [19], in which operations are assigned to a local site where they are executed and later propagated to remote sites. This way, these two mechanisms can be classified using the following parameters: *transaction location* and *synchronization strategy.*

Primary-backup replication: According to the parameters defined above, in the primary-backup approach updates are made on a single replica (master) and then propagated to the remaining replicas, while read operations can be directed to any node. This propagation can be either made in an *eager* or *lazy* way. In the eager approach, the client issues an update and the master only replies after the update has been propagated to all the replicas, often providing strong consistency. On the other hand, the lazy approach assumes that the master replies to the client after the update has been applied locally. This way, if the client issues a read operation to one of the other replicas, before the propagation of the update, she can see stale data. Additionally, in the primary-backup approach it is simple to control the concurrency of updates (serialized at the master), although a single master can be seen as a bottleneck in the system.

Multi-master replication: In multi-master replication the write and read operations can be made at any replica. Like in primary-backup, the propagation of updates is made in a similar way, either in the lazy or eager approach. However, in the eager approach the replicas must synchronize in order to decide on a single ordering of concurrent updates, which can lead to heavy communication. In the lazy approach, there are some situations where conflicts can occur, which can be hard to solve as was described before. Considering the previous description, we can state that the multi-master approach is more flexible than primary-backup and allows a better distribution of write operations among the replicas. Unfortunately, the existence of multiple masters imposes a higher level of synchronization that can lead to a large number of messages.

3.7 Existing Systems

This section will introduce some systems that are relevant for the work that is going to be developed. Most of these systems are intended to be deployed in a large-scale setting and illustrate the implementation of some consistency models described before.

3.7.1 Chain Replication Chain Replication, introduced in [20], is a primary-backup approach that allows to build a datastore that provides linearizability, high throughput, and availability. This approach assumes that replicas are organized in a *chain topology.* A chain consists of a set of nodes where each node has a successor and a predecessor except for the first (head) and last (tail) nodes of the chain.

The write operations are directed to the head of the chain and are propagated until they reach the tail. At this point the tail sends a reply to the client and the write finishes. The way updates are propagated along the chain ensures the following property, named *Update Propagation Invariant*: the history of updates of a given node is contained on the history of updates of its predecessor. Contrary to write operations, read operations are always routed to the tail which returns the value of the object. Since all the values stored in the tail are guaranteed to have been propagated to all replicas, reads are always consistent.

The failure of a node can break the chain and render the system unavailable. To avoid this problem, chain replication provides a fault-tolerance mechanism that recovers from node failures in three different situations. This mechanism assumes the fail-stop model [21] and operates as follows:

- If the head (H) of the chain fails, then the head is replaced by its successor (H^+). All the pending updates, that were in H, and were not propagated to H^+ are assumed to have failed.

- The failure of the tail is handled by replacing it by its predecessor. By the property of update propagation invariant is guaranteed that the new tail has all the needed information to continue on normal operation.

- Unlike the failures described above, the failure of a node in the middle of the chain (S) is not simple to handle. To remove S from the chain, its successor (S^+) and its predecessor (S^-) must be informed. Also, it must be guaranteed that the updates propagated to S are propagated to S^+, so that the property of update propagation invariant is not violated.

Failed nodes are removed from the chain, which shortens its size. Although, a chain with a small number of nodes tolerates fewer failures. To overcome this problem, chain replication also offers a mechanism to add nodes to a chain. The simplest way of doing this procedure is to add a node after the tail (T). In order to add a node T^+, the current tail needs to propagate all the seen updates to T^+, and is later notified that is no longer the tail.

Chain replication is considered as a Read One, Write All Available (ROWAA) approach, which implements linearizability in a very simple way. However, chain replication is not applicable to partitioned operation and has some disadvantages, as follows; The overloading of the taill with write and read operations, which hinders throughput and constitutes a bottleneck in the system; The failure of just one server makes the throughput of read and write operations, drop to zero until the chain is repaired.

Recent work [22], on top of chain replication, tries to mitigate the problem of the tail overloading by allowing reads to be routed to any node. There is also an effort in trying to improve chain replication behavior in a multiple datacenter scenario by providing different chain topologies [23].

3.7.2 Clusters of Order-Preserving Servers (COPS) COPS [15] is a datastore designed to provide causal+ consistency guarantees while providing high scalability over the wide-area. It provides the aforementioned guarantees through the use of operation *dependencies*. The dependencies of an operation are the previous operations that must take place before the former, in order to maintain the causal order of operations. These dependencies are included in the read/write requests issued by a client. Operations can only take place if the version of an object, present in a datacenter, fulfills the dependencies. Otherwise, operations are delayed until the needed version is written in the datacenter.

The authors argue that COPS is the first system to achieve causal+ consistency in a scalable way. Previous work on this topic [14, 13] described the implementation of systems that provide causal+ consistency. However, these systems do not scale and are not suitable for a Geo-replication scenario. COPS also introduces a new type of operations named as *get-transactions*. These operations allow a client to read a set of keys ensuring that the dependencies of all keys have been met before the values are returned. The usefulness of these operations can be better assessed with resort to an example: imagine that we issue two writes to objects A and B in a sequential order, however it could happen that the write on B is propagated faster among the replicas. Considering this, if we issue two consecutive reads to A and B it could happen that we see the old value of A and the new value of B, which is correct according to the causal ordering of operations but is not desirable in some applications. To overcome this situation we could use the get-transaction construct, which imposes that if we read the new version of B we must also read the new version of A (because the write on A happens-before the write on B). Additionally, in the case that we read the old version of B, then either the old or the new version of A can be observed.

The guarantees and scalability of COPS are a result of the use of dependencies. However, these dependencies are stored in the client and the system has no control over them, which may lead to problems, such as the following: The number of dependencies stored in the client can be very large, especially when using get-transactions. This constitutes a major drawback in this solution because client applications may not be able to store this amount of information.

3.7.3 Walter Recent work on geo-replication introduced a new key-value store that supports transactions, known as Walter [18]. This datastore implements parallel snapshot isolation (described previously) allowing the transactions to be asynchronously propagated after being committed at one site, by a central server. Transactions are ordered with resort to vector timestamps, assigned at their beginning, which contains an entry for each site. Each vector timestamp represents a snapshot of the data in the system.

Walter also introduces two novel mechanisms that are also used to implement PSI: preferred sites and counting sets. Each object has a preferred site that corresponds to the datacenter closer to the owner of that object. This enables transactions that are local to the preferred site to be committed more efficiently using a *Fast Commit* protocol. This protocol allows for a transaction to commit at the preferred site without contacting other sites. To commit a transaction in this way, Walter must perform two checks: check if all objects in the write-set have not been modified since the transaction started; check if object in the write-set are unlocked. If the two conditions are verified then the transaction can be committed. However, if the transaction is not local it must execute a *Slow Commit* protocol. This protocol consists in a two-phase commit protocol between the preferred sites of the objects being written.

The counting sets are a new data type, similar to commutative replicated data types [24], that allows to avoid write-write conflicts. Operations where counting sets are acceptable can be quickly committed using the Fast Commit protocol, improving the throughput in those cases.

Walter was designed to provide PSI in a scalable way considering a geo-replicated scenario. However, there are some factors that can hinder its scalability. The fact that transactions are executed and committed by a central server at each site corresponds to a bottleneck in performance. Moreover, if transactions are not local, then they must be committed by the Slow Commit protocol, which can limit the throughput and scalability.

3.7.4 Yahoo's PNUTS PNUTS [8] is a large scale data storage system, built by Yahoo, that provides storage for applications that are geographically distributed. This system tries to provide some guarantees between strong and eventual consistency, by implementing the per-record timeline consistency model. This relaxed model allows PNUTS to provide better guarantees than eventual consistency systems while maintaining the performance and scalability properties.

This system works with records as its data storage unit, which consists of a row of a common database table. To implement the aforementioned consistency model each record has its own master replica, to which all updates are routed. The master of a record is selected according to the following criteria: if a replica receives a majority of updates to a record, that have origin in the same geographic region, then that replica is selected as the master. This way, PNUTS allows an adaptive change of the master, according to the workload changes.

PNUTS provides an API containing a new set of operations that allows the achievement of different degrees of consistency. However, these operations impose over the applications, the task of explicitly including version numbers in some of their calls to the API, exposing the internal structure of the system. Moreover, the current version of PNUTS is not adaptable to network partition scenarios due to the existence of a master replica. If a partition occurs and the master is on one side of the partition, then the other side cannot reach the master and two situations could occur: i) The side of the partition that cannot reach the master will not make any updates; ii) a new master is elected (on the side that cannot access the master) and there will be two master replicas, which is not desirable.

3.7.5 Lazy Replication Lazy replication, introduced in [25], is considered as a multi-master approach that allows to build causal consistent distributed systems. This technique states that operations are only executed at one replica. The rest of the replicas are updated in the background using *lazy* gossip messages. Therefore, it aims at providing better performance and scalability at the cost of providing weaker consistency guarantees.

The causal consistency model is implemented by using *multipart timestamps*, which are assigned to each object. These timestamps consist on a vector that contains the version number of the object at each replica. This vector must be included (in the form of dependencies) in all update and query requests so that operations respect the causal ordering: an operation can only be processed at a given node if all the operations on which it depends have already been applied at that node. Furthermore, the nodes must be aware of the versions present in other replicas. This information is maintained in a table that is updated by exchanging gossip messages between replicas. When a node is aware that a version as already reached all replicas then it issues an acknowledgment message so that the gossip dissemination protocol can be stopped.

The causal consistency guarantees provided by this approach allow its adaptation to a Geo-replicated scenario while maintaining scalability. Moreover, this technique is able to handle network partitions since the replicas can synchronize the updates when network recovers from the partition, without affecting the operation of the system. However, this approach has a major drawback: large vector timestamps. These timestamps must be stored in the client and can be very large in a system that comprises hundreds of nodes.

3.7.6 Amazon's Dynamo Amazon has many services that ust be highly available and scalable in order to provide a good user experience. One of the components that helps satisfying these requirements is the Dynamo key-value store [26]. This store is based on a Distributed Hash Table that implements eventual consistency.

Dynamo provides eventual consistency guarantees in the event of failures or high load of the system. In this scenario, read and write operations are made over a number of replicas that represent a majority (quorum technique). This number can be configured by applications, however the number of read and write replicas must be higher than the existing replicas (i.e., read and write quorums must overlap). Each write creates a new version that corresponds to a vector clock, which is used in read operations to obtain the most recent version in a read quorum.

Unlike most datastores, Dynamo was designed to provide high write availability. Therefore, it allows multiple writes at multiple replicas resulting in write-write conflicts. These conflicts can be handled either automatically or by notifying the application, that then solves the conflict by other means.

3.7.7 Discussion and Comparison The previous sections have described a number of systems that use different approaches to build a distributed datastore. These systems implement different consistency models, concurrency control mechanisms, and replication mechanisms, as can be seen in Table 1.

As shown in the table, most of these solutions tend to use optimistic concurrency control in order to exploit the maximum usage of the existing resources. The studied systems that use pessimistic concurrency control (Chain Replication and PNUTS) require that all writes are serialized at a single replica (avoiding the existence of concurrent updates). According to the latter, Chain Replication is not suitable for a Geo-replicated scenario due to the low scalability of the approach. On the other hand, PNUTS makes use of relaxed consistency guarantees in order to achieve a scalable and efficient solution.

Additionally, the majority of the studied systems implement a primary-backup or multi-master replication mechanism with the exception of Dynamo. However, the latter must relax consistency guarantees in order to scale for a large number of users. This is due to the fact that an active replication mechanism that provides strong consistency requires a heavy synchronization protocol that hinders the performance of the system.

COPS and Lazy Replication are examples of the implementation of a multi-master replication approach. As we can see, both approaches require the storage of metadata (dependencies) in the client, which allows to achieve the causal ordering of operations. However, both solutions can be used in a Geo-replicated scenario at the cost of storing some (possibly large) amount of information at each client.

Systems	Consistency Model	Concurrency Control		Replication Approach		
		Optimistic	Pessimistic	Active	Primary-Backup	Multi-Master
Chain Replication	Linearizability		✓		✓(eager)	
COPS	Causal+	✓				✓(lazy)
Walter	PSI	✓			✓(lazy)	
PNUTS	Per-record timeline		✓		✓(lazy)	
Lazy Replication	Causal	✓				✓(lazy)
Dynamo	Eventual	✓		✓		

Table 1: Comparison between the presented systems.

According to the solutions described, it is reasonable to claim that most geo-replicated applications tend to provide eventual consistency guarantees [26, 27], which are the weaker of all models described. This is due to the fact that these applications cannot afford the increase in latency imposed by the implementation of stronger models. Therefore, the consistency guarantees are weakened in order to provide the maximum performance and scalability.

4 Architecture

The previous section introduced most of the context needed for achieving the goals stated in Section 2. Considering the models described, it is reasonable to say that some are acceptable in some applications and others are not.

As we can see, linearizability, serializability, sequential consistency, and snapshot isolation provide a set of guarantees that are most programmer-friendly, since they allow the programmer to build applications without having to worry about concurrency and replication issues. However, these models cannot scale to the wide area because they require the use of a expensive construct (Distributed Consensus). Other models like causal, causal+, and PSI provide weaker guarantees than the models above. Moreover, they provide some guarantees to the programmer while maintaining scalability and performance, making them attractive for Geo-replicated scenarios.

Hereupon, in our solution we intent to implement the causal+ consistency model, which is interesting because it merges the best from causal and eventual consistency: It provides stronger guarantees than eventual consistency (causal ordering of operations), while preserving the convergence property, which avoids the existence of divergent replicas (a problem of causal consistency). This model is adaptable to a large number of applications that do not require atomic consistency. However, they would benefit from the stronger consistency offered by causal+ (instead of using eventual consistency). Also, we will try to provide these guarantees in a efficient and scalable way in order to be able to port this solution for a Geo-replicated scenario.

To achieve the aforementioned performance, our design will be based on the chain replication approach. This approach constitutes a very simple and flexible way of achieving primary-backup replication and atomic consistency, while providing better throughput than other approaches [28]. More specifically, this solution aims at relaxing the consistency guarantees offered in the chain replication approach by providing causal+ consistency. Since chain replication achieves good throughput while ensuring atomic consistency, it is expected that the proposed solution will have better performance and scalability considering the consistency model being implemented.

4.1 Operation Types

Considering the comparison and discussion stated previously, in this section we describe the operations that will be supported by our datastore prototype. We intend to provide two types of operations: isolated operations and multi-object operations.

The set of isolated operations is composed by write and read operations, which are described as follows:

Write Operation: Write operations will allow a programmer to perform updates on a given object. According to the specification described in the following section, write operations are always made over the newest version of the object ensuring the guarantees imposed by causal+ consistency.

Read Operation: Read operations will enable a programmer to access the values of the stored objects that will reflect the outcome of the write operations. Like the previous, read operations are ensured to provide values that are correct according to the causal order of operations.

There are some applications where reads and writes are not enough because it could be very difficult to provide some functionalities with just these two types of operations. Thus, we think it is interesting to provide multi-object operations, which are inspired by the get-transactions supported by COPS. These operations would be something similar to a transaction but they would only allow for a single operation (read/write) over a set of objects (unlike fully-fledged transactions that allow multiple operations within the transactional environment). To this extent, we aim at providing two operations of this kind: *multi-object reads* and *multi-object writes*.

Multi-object Read Operation: This operation would allow an application to fetch a set of objects, whose values returned are ensured to fulfill all the objects dependencies (similar to get-transactions). The rational for this operation is the following: imagine a social network that allows users to share photo albums. Alice is a user of this application and she removes Bob from the list of users that can see her photos (ACL). Then Alice adds a new photo that Bob should not be able to see. In this case if two read operations were used to fetch both the ACL and the album, it could happen that Bob reads an older version of the ACL and the new version of the album. This way, Bob would see the new photo and would violate the semantics of the ACL. To support this behavior, it would be interesting to provide multi-object reads.

Multi-object Write Operation: Unfortunately, multi-object reads may not be enough to satisfy the application needs. There are some situations where applications may need to modify a set of objects that depend on each other. Considering the previous example, imagine that the application also supports friend lists, in which users can remove and add friends. Imagine that Alice is a friend of Bob, this means that her friend list has an entry for Bob and Bob's list has an entry for Alice. If Alice wants to remove Bob from her friends the application would issue a write operation to remove Bob from her list and an write operation to remove Alice from Bob's list (in a sequential order). However, if another user checks Bob's profile between the two write operations, she could see that Bob is a friend of Alice but Alice is not a friend of Bob, which makes no sense since friendship is a reciprocal relation. This way, it would be interesting to provide an operation primitive (multi-object writes) that could update a set of values that do not make sense to be updated in a separate way.

4.2 Proposed Implementation

In this section we will describe the proposed design for our prototype. To simplify the description we will only consider a single datacenter (a multi-datacenter scenario will be described in the following section). Our design is based on using a topology where nodes inside a datacenter are organized in a *chain* (Figure 3), using similar techniques to those used in chain replication (Section 3.7.1). However, in chain replication the operations can only be issued to a single node, in which writes are issued to the head and reads to the tail. This leads to an overloading of the tail because it needs to process not only all the writes that are propagated to it but also all the read requests (normally in higher number than writes). Further developments made in CRAQ [22] allow read operations to be issued to any node. However, this solution still requires some communication between the node and the tail in some situations. Specifically, if the version on the node has not yet reached the tail, then the node must contact the tail in order to serve the read request. In this case some of the load is taken away from the tail but it still gets saturated in a scenario where the number of writes is high.

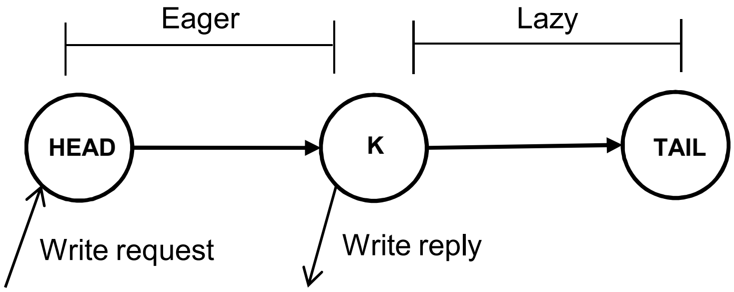


Fig. 3: Example of a write operation.

Unlike chain replication, we intend to allow a set of nodes to serve read requests, without communicating with the tail. This is possible due to the update invariant property, which can be used to show that if a node K is causally consistent, with respect to some client operations, then all predecessors of node K are also causal consistent. Moreover, the replication mechanism that is going to be employed is a hybrid between eager and lazy primary-backup replication since updates will be propagated to some nodes (not all) before replying to the client. Additionally, we will study different topologies of the chain to assess the trade-offs between flexibility and operations complexity.

In order to provide an useful datastore to applications, the design will offer the following type of operations: writes, reads, and multi-object operations. The description of these operations is as follows:

Write Operation: Like in chain replication, write operations will be directed to the head of the chain. This operation will allow a client to write a new value on a given object. This write corresponds to a new version that is going to be propagated down the chain, until it reaches node K. The latter is the kth node in the chain, as shown in Figure 3, this guarantees that the chain has a minimum replication factor equal to K. This parameter is intended to be configurable in order to provide a trade-off between availability and write latency (i.e., if we choose a bigger K we will have a bigger replication factor but also a higher write latency). When the write operation reaches node K, a reply containing the new version and some metadata (used in following read operations), is sent to the client. At the same time, the write operation is propagated further (in a lazy way) down the chain, until it reaches the tail.

Read Operation: The read operation provided by our approach will allow a client to issue read operations to a set of consistent nodes. This set of nodes is identified by the metadata that is returned in read and write operations (explained above). If the client reads a version that has already reached the tail then the set of consistent nodes contains all nodes in the chain and no metadata is returned (i.e., in the absence of the metadata the read operation can be made in any node). Otherwise, the set of consistent nodes contains the node that served the request and all its predecessors. The client is informed by piggybacking the metadata in the read result.

Multi-object Read Operation: The straightforward solution to implement this kind of operation is to retrieve the values that are at the head of each object's chain. However, this solution would have a poor performance because it does not allow for the load to be distributed among the nodes in an object's chain. Therefore, we will develop new solutions for implementing this operation in the future work.

Multi-object Write Operation: One simple way to implement this operation is to employ a two-phase commit protocol between the heads of the objects' chains. Given that a two-phase commit protocol has poor performance and low scalability, in the future work we intend to propose new protocols for multi-object write operations.

4.3 Adaptation to Geo-replication

In order to use the previous design in a Geo-replicated scenario, the chain must be adapted to multiple datacenters. One possible solution for this setting is to use the following topology: the chain would have multiple heads (i.e., one for each datacenter) and a single tail (located in one of the datacenters), as can be observed in Figure 4. However, other solutions will be discussed in the future work.

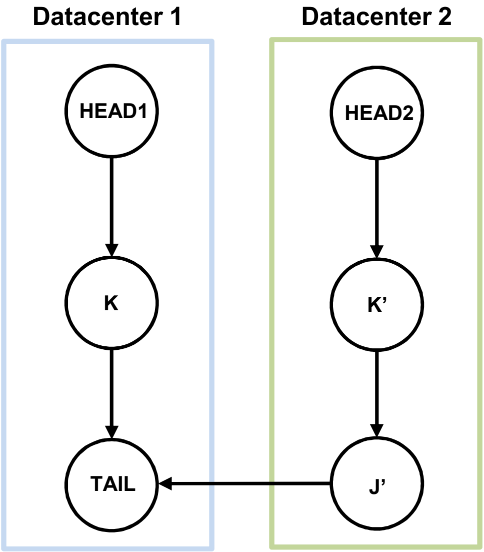


Fig. 4: Proposed topology for a geo-replicated scenario.

In order to provide the topology described above, operations would need to be modified. One of the possible solutions for these modifications are described as follows:

Write Operation: All the write operations issued by a client would be redirected to the nearest datacenter in order to experience a smaller latency. After reaching the head of the nearest datacenter, the write request would be propagated to the other datacenters’ heads. Additionally, in this scenario it is possible for concurrent writes, to the same object, to occur at different datacenters. In order to solve this write-write conflict, the system would require the existence of some synchronization mechanism to order the writes.

Read Operation: Read operations would not require extensive modifications. Only the metadata that is returned to the clients would need to reflect the state in all datacenters (larger than with a single local chain). This is useful when the nearest datacenter is unreachable and the client still wants to access data in a causal+ consistent way (on another datacenter).

Multi-Object Operations: Since multi-object operations are not yet clearly defined how they are going to be implemented to a single local chain, their adaptation to this setting will be further developed in future work.

5 Evaluation

The goal of this work is to build a datastore that provides causal+ consistency while providing good performance and scalability in a geo-replicated scenario. To assess the validity of the proposed architecture we intend to build a prototype and to evaluate the implementation experimentally. This evaluation procedure will be focused on four main aspects:

- Determine the *throughput* of read, write, and multi-object operations;
- Analyze the effects of write operations on read throughput and the effect of multi-object operations on both reads and writes;
- Determine the *latency* of each type of operations;
- Assess the *size* of the metadata that must be stored in the client in order to achieve the desired guarantees.

Moreover, we want to show the trade-off between stronger consistency guarantees and weaker guarantees, while comparing with a similar replication approach (chain replication). Additionally, if time allows, we intend to port an application to use our prototype and analyze the benefits of this change.

6 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29, 2012 Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15: Deliver the MSc dissertation.

7 Conclusions

In this report we presented the most relevant mechanisms used to build a data storage infrastructure for large-scale Geo-replicated applications.

In the first place we introduced the benefits that result from the use of replication and concurrency control and we identified consistency as its main challenge.

Then, we introduced some consistency models, which are considered relevant for the work being developed. Moreover, we surveyed the existing concurrency control and replication mechanisms. Some existing solutions were described in order to provide some awareness on the current state of art, concerning a Geo-replicated scenario.

We also proposed a design for a datastore that aims at providing some consistency guarantees to programmers while maintaining a reasonable throughput and scalability. The major challenges in this design are to avoid the storing of metadata in the client and the implementation of multi-object operations.

We concluded the report with a description of the methodologies that we expect to apply during the evaluation of the proposed architecture.

Acknowledgments I'm grateful to Professor Luís Rodrigues and to my colleagues João Leitão, Oksana Denysyuk, João Paiva, and Miguel Branco for the fruitful discussions and comments during the preparation of this report. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the INESC-ID multi-annual funding through the PIDDAC Program fund grant and via the project "HPCI" (PTDC/EIA-EIA/102212/2008).

References

1. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley (1987)
2. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16** (May 1998) 133–169
3. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys **22** (December 1990) 299–319
4. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. PODC '00, New York, NY, USA, ACM (2000) 7
5. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems **4** (June 1979) 180–209
6. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12** (July 1990) 463–492
7. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **C-28**(9) (September 1979) 690 –691
8. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment **1** (August 2008) 1277–1288
9. Ahamad, M., Neiger, G., Burns, J., Kohli, P., Hutto, P.: Causal memory: definitions, implementation, and programming. Distributed Computing **9** (1995) 37–49 10.1007/BF01784241.
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (July 1978) 558–565

11. Vogels, W.: Eventually consistent. Commun. ACM **52** (January 2009) 40–44
12. Gustavsson, S., Andler, S.F.: Self-stabilization and eventual consistency in replicated real-time databases. In: Proceedings of the first workshop on Self-healing systems. WOSS '02, New York, NY, USA, ACM (2002) 105–107
13. Belarami, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., Zheng, J.: Practi replication. NSDI '06 (May 2006) 59–72
14. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Proceedings of the sixteenth ACM symposium on operating systems principles. SOSP '97, New York, NY, USA, ACM (1997) 288–301
15. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11, New York, NY, USA, ACM (2011) 401–416
16. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Computing Surveys **13** (June 1981) 185–221
17. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: Proceedings of the 1995 ACM SIGMOD international conference on Management of data. SIGMOD '95, New York, NY, USA, ACM (1995) 1–10
18. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11, New York, NY, USA, ACM (2011) 385–400
19. Charron-Bost, B., Pedone, F., Schiper, A.: Replication theory and practice. Volume 5959. Springer Berlin (2010)
20. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6, Berkeley, CA, USA, USENIX Association (2004) 91–104
21. Schneider, F.B.: Byzantine generals in action: implementing fail-stop processors. ACM Trans. Comput. Syst. **2** (May 1984) 145–154
22. Terrace, J., Freedman, M.J.: Object storage on craq: high-throughput chain replication for read-mostly workloads. In: Proceedings of the 2009 conference on USENIX Annual technical conference. USENIX'09, Berkeley, CA, USA, USENIX Association (2009) 11–11
23. Laden, G., Melamed, R., Vigfusson, Y.: Adaptive and dynamic funnel replication in clouds. LADIS 2011 (September 2011)
24. Shapiro, M., Preguiça, N.M.: Designing a commutative replicated data type. CoRR **abs/0710.1784** (2007)
25. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Transactions on Computer Systems **10** (November 1992) 360–391
26. Hastorun, D., Jampani, M., Kakulapati, G., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: In Proc. SOSP. (2007) 205–220
27. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Operating Systems Review **44** (April 2010) 35–40
28. Jiménez-Peris, R., Patiño Martínez, M., Alonso, G., Bettina, K.: Are quorums an alternative for data replication? ACM Trans. Database Syst. **28** (September 2003) 257–294