Bulletin of the Technical Committee on

Data Engineering

March 2016 Vol. 39 No. 1 IEEE Computer Society

Letters

Special Issue on Data Consistency across Research Communities

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

Evolution of the Bulletin

This issue continues with the use of dvipdfm as the program used to translate the latex generated dvi file to pdf. This has required additional work for bulletin editors and authors, as there is increased opportunities for conflicting style files and ongoing difficulties with figures. We are still sorting out the "rules of the road" for this. The gain is clear– each individual paper now has correct page numbering, and eventually, the compilation of the issue should be a bit easier as well. In the short term, however, both authors and editors have had to sweat through increased difficulties resulting from bringing the issue together. I want to thank authors for their patience, and particularly thank Bulletin editors for their willingness to go the extra mile in successfully publishing the Bulletin.

The Current Issue

Distributed information systems are verging on being everywhere. This is a result of a combination of factors, many related to their rapidly emerging deployments in the cloud. There are, of course, specialized distributed systems, think Facebook, but also distributed platforms. This kind of wide-scale distribution is a relatively recent artifact of the paradigm shift to the cloud. Earlier, for example, most instances of two phase commit were applied to local clusters of machines. But the scale has definitely changed.

The CAP theorem suggests that we may need to give up one of availability, partition tolerance, and consistency when building and deploying distributed information systems. One cannot, at least successfully, argue with math proofs, so the CAP theorem will not be repealed. However, as this issue illustrates, there is more to the situation than necessarily meets the eye. And implementors are now including several forms of consistency, including transactional consistency, in the systems being researched and deployed.

The current issue captures some of the diversity current in the consistency. What should make the issue of particular interest is that issue editor Bettina Kemme has reached out to researchers outside of the database community in an effort to expose both the distributed systems and database communities to the wide variety of work being pursued. Given the high importance and double digit growth of cloud systems and applications, readers should find the issue a gold mine of both varieties of consistency and approaches to realizing consistency. I thank Bettina for her very successful effort in producing the issue, and strongly recommend it to Bulletin readers.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

For decades the database community has looked at *consistent data replication and distribution* – most often in the context of the transactions offering the isolation and atomicity properties. The proliferation of distributed and replicated database systems, be it within a cloud, across data centres, or in mobile environments, has led to a rich research literature offering manyfold solutions to distributed transaction management and replica control. In the March Issue 2015 of the IEEE Bulletin, I invited experts in this area, mainly from the database community, to present their views and work on data consistency in the cloud, with a focus on transaction management. In contrast, this issue aims in giving an insight into how other research communities address the challenges of data consistency as this is such a cross-cutting concern that requires solutions from many areas of computer science: distributed systems, distributed algorithms, storage systems, operating systems, computer architecture and programming language, to name a few.

While each of the articles looks at the problem with the perspective of a specific research discipline it becomes apparent that there is a common thread of main building blocks that are fundamental to providing consistency in distributed and replicated environments. Examples are well-known protocols such as consensus, state-machine replication, or 2-phase commit, correctness criteria such as serializability, linearizability, causal and eventual consistency, and properties such as scalability, availability, and throughput and response time performance. This shows the importance of a cross-disciplinary approach to data consistency.

The first set of papers sheds light on the challenge to actually understand what "consistency for replicated data" means, as there are many possible interpretations and trade-offs. In the first article, Aguilera and Terry look at the notions of consistency across disciplines. The authors identify two broad types of consistency, namely state- and operation consistency, and show how they map to the many existing definitions of consistency developed for distributed systems, database systems, and computer architecture. In the second article, Guerraoui *et al.* provide a comprehensive overview of the trade-offs between many different properties in replicated systems such as the level of data consistency offered, performance in terms of latency and/or scalability, and other aspects such as availability and robustness to failures and churn. In the next article, Zhang *et al.* offer a detanglement of the often intertwined tasks of distributed concurrency control, distributed commit, and replication protocols that offer availability and consistent data despite failures. In particular, the authors provide a detailed discussion of when and when not operations need to be ordered across all replicas. In the following article, Vukolić takes a different viewpoint, focusing on strongly consistent protocols but looks how they can be implemented not only in software, but also in hardware, and for blockchains.

From there, the next two articles look at consistency issues from a programming language perspective. Burckhardt and Protzenko describe the concept of abstract executions to specify the behaviour of a replicated system. Such a specification approach helps in investigating fundamental properties of replication solutions. Sivaramakrishnan *et al.* present Quelea, a declarative programming model for eventually consistent data stores that allows for expressing precise high-level consistency guarantees without the need to know implementation-specific low-level data store semantics.

The last three articles present propose concrete frameworks and solutions for data distribution and replication. Altınbüken and van Renesse present a software-defined framework for building large-scale distributed systems that handles dynamic change and growth by allowing components to be transformed dynamically, be it to support replication, batching, sharding or encryption. Balegas *et al.* present a solution to geo-replication that merges weak and strong consistency solutions in order to find the right balance between consistency and performance. Finally, Bezerra *et al.* looks at performance aspects of strong consistency solutions based on the state machine approach at scale.

I hope you enjoy this collection of articles as much as I did!

Bettina Kemme
McGill University
Montreal, Canada

The many faces of consistency

Marcos K. Aguilera
VMware Research Group

Douglas B. Terry
Samsung Research America

Abstract

The notion of consistency is used across different computer science disciplines from distributed systems to database systems to computer architecture. It turns out that consistency can mean quite different things across these disciplines, depending on who uses it and in what context it appears. We identify two broad types of consistency, state consistency *and* operation consistency*, which differ fundamentally in meaning and scope. We explain how these types map to the many examples of consistency in each discipline.*

1 Introduction

Consistency is an important consideration in computer systems that *share* and *replicate* data. Whereas early computing systems had private data exclusively, shared data has become increasingly common as computers have evolved from calculating machines to tools of information exchange. Shared data occurs in many types of systems, from distributed systems to database systems to multiprocessor systems. For example, in distributed systems, users across the network share files (e.g., source code), network names (e.g., DNS entries), data blobs (e.g., images in a key-value store), or system metadata (e.g., configuration information). In database systems, users share tables containing account information, product descriptions, flight bookings, and seat assignments. Within a computer, processor cores share cache lines and physical memory.

In addition to sharing, computer systems increasingly replicate data within and across components. In distributed systems, each site may hold a local replica of files, network names, blobs, or system metadata—these replicas, called caches, increase performance of the system. Database systems also replicate rows or tables for speed or to tolerate disasters. Within a computer, parts of memory are replicated at various points in the cache hierarchy (L1, L2, L3 caches), again for speed. We use the term replica broadly to mean any copies of the data maintained by the system.

In all these systems, data sharing and replication raise a fundamental question: what should happen if a client modifies some data items and simultaneously, or within a short time, another client reads or modifies the same items, possibly at a different replica?

This question does not have a single answer that is right in every context. A consistency property governs the possible outcomes by limiting how data can change or what clients can observe in each case. For example, with DNS, a change to a domain may not be visible for hours; the only guarantee is that updates will be seen eventually—an example of a property called eventual consistency [23]. But with flight seat assignments, updates must be immediate and mutually exclusive, to ensure that no two passengers receive the same seat—an example

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

of a strong type of consistency provided by serializability [5]. Other consistency properties include causal consistency [13], read-my-writes [21], bounded staleness [1], continuous consistency [1, 25], release consistency [10], fork consistency [16], epsilon serializability [18], and more.

Consistency is important because developers must understand the answer to the above fundamental question. This is especially true when the clients interacting with the system are not humans but other computer programs that must be coded to deal with all possible outcomes.

In this article, we examine many examples of how consistency is used across three computer science disciplines: distributed systems, database systems, and computer architecture. We find that the use of consistency varies significantly across these disciplines. To bring some clarity, we identify two fundamentally different types of consistency: state consistency and operation consistency. State consistency concerns the state of the system and establishes constraints on the allowable relationships between different data items or different replicas of the same items. For instance, state consistency might require that two replicas store the same value when updates are not outstanding. Operation consistency concerns operations on the system and establishes constraints on what results they may return. For instance, operation consistency might require that a read of a file reflects the contents of the most recent write on that file. State consistency tends to be simpler and application dependent, while operation consistency tends to be more complex and application agnostic. Both types of consistency are important and, in our opinion, our communities should more clearly disentangle them.

While this article discusses different forms of consistency, it focuses on the *semantics* of consistency rather than the *mechanisms* of consistency. Semantics refer to what consistency properties the system provides, while mechanisms refer to how the system enforces those properties. Semantics and mechanisms are closely related but it is important to understand the former without needing to understand the latter.

The rest of this article is organized as follows. We first explain the abstract system model and terminology used throughout the article in Section 2. We present the two types of consistency and their various embodiments in Section 3. We indicate how these consistency types occur across different disciplines in Section 4.

2 Abstract model

We consider a setting with multiple *clients* that submit *operations* to be executed by the system. Clients could be human users, computer programs, or other systems that do not concern us. Operations might include simple read and write, read-modify-write, start and commit a transaction, and range queries. Operations typically act on data items, which could be blocks, files, key-value pairs, DNS entries, rows of tables, memory locations, and so on.

The system has a *state*, which includes the current values of the data items. In some cases, we are interested in the consistency of client caches and other replicas. In these cases, the caches and other replicas are considered to be part of the system and the system state includes their contents.

An operation execution is not instantaneous; rather, it *starts* when a client submits the operation, and it *finishes* when the client obtains its response from the system. If the operation execution returns no response, then it finishes when the system is no longer actively processing it.

Operations are distinct from operation executions. Operations are static and a system has relatively few of them, such as read and write. Operation executions, on the other hand, are dynamic and numerous. A client can execute the same operation many times, but each operation execution is unique. While technically we should separate operations from operation executions, we often blur the distinction when it is clear from the context (e.g., we might say that the read operation finishes, rather than the execution of the read operation finishes).

3 Two types of consistency

We are interested in what happens when shared and replicated data is accessed concurrently or nearly concurrently by many clients. Generally speaking, consistency places constraints on the allowable outcomes of operations, according to the needs of the application. We now define two broad types of consistency. One places constraints on the state, the other on the results of operations.

3.1 State consistency

State consistency pertains to the state of the system; it consists of properties that users expect the state to satisfy despite concurrent access and the existence of multiple replicas. State consistency is also applicable when data can be corrupted by errors (crashes, bit flips, bugs, etc), but this is not the focus of this article.

State consistency can be of many subcategories, based on how the properties of state are expressed. We explain these subcategories next.

3.1.1 Invariants

The simplest subcategory of state consistency is one defined by an invariant—a predicate on the state that must evaluate to true. For instance, in a concurrent program, a singly linked list must not contain cycles. In a multiprocessor system, if the local caches of two processors keep a value for some address, it must be the same value. In a social network, if user x is a friend of user y then y is a friend of x. In a photo sharing application, if a photo album includes an image then the image's owner is the album.

In database systems, two important examples are uniqueness constraints and referential integrity. A *uniqueness constraint* on a column of a table requires that each value appearing in that column must occur in at most one row. This property is crucial for the primary keys of tables.

Referential integrity concerns a table that refers to keys of another table. Databases may store relations between tables by including keys of a table within columns in another table. Referential integrity requires that the included keys are indeed keys in the first table. For instance, in a bank database, suppose that an accounts table includes a column for the account owner, which is a user id; meanwhile, the user id is the primary key in a users table, which has detailed information for each user. A referential integrity constraint requires that user ids in the accounts table must indeed exist in the users table.

Another example of state consistency based on invariants is *mutual consistency*, used in distributed systems that are replicated using techniques such as primary-backup [2]. Mutual consistency requires that replicas have the same state when there are no outstanding updates. During updates, replicas may diverge temporarily since the updates are not applied simultaneously at all replicas.

3.1.2 Error bounds

If the state contains numerical data, the consistency property could indicate a maximum deviation or error from the expected. For instance, the values at two replicas may diverge by at most ϵ. In an internet-of-things system, the reported value of a sensor, such as a thermometer, must be within ϵ from the actual value being measured. This example relates the state of the system to the state of the world. Error bounds were first proposed within the database community [1] and the basic idea was later revived in the distributed systems community [25].

3.1.3 Limits on proportion of violations

If there are many properties or invariants, it may be unrealistic to expect all of them to hold, but rather just a high percentage. For instance, the system may require that at most one user's invariants are violated in a pool of

a million users; this could make sense if the system can compensate a small fraction of users for inconsistencies in their data.

3.1.4 Importance

Properties or invariants may be critical, important, advisable, desirable, or optional, where users expect only the critical properties to hold at all times. Developers can use more expensive and effective mechanisms for the more important invariants. For instance, when a user changes her password at a web site, the system might require all replicas of the user account to have the same password before acknowledging the change to the user. This property is implemented by contacting all replicas and waiting for replies, which can be an overly expensive mechanism for less important properties.

3.1.5 Eventual invariants

An invariant may need to hold only after some time has passed. For example, under eventual consistency, replicas need not be the same at all times, as long as they *eventually* become the same when updates stop occurring. This eventual property is appropriate because replicas may be updated in the background or using some anti-entropy mechanism, where it takes an indeterminate amount of time for a replica to receive and process an update. Eventual consistency was coined by the distributed systems community [23], though the database community previously proposed the idea of reconciling replicas that diverge during partitions [9].

State consistency is limited to properties on state, but in many cases clients care less about the state and more about the results that they obtain from the system. In other words, what matters is the behavior that clients observe from interacting with the system. These cases call for a different form of consistency, which we discuss next.

3.2 Operation consistency

Operation consistency pertains to the operation executions by clients; it consists of properties that indicate whether operations return acceptable results. These properties can tie together many operation executions, as shown in the examples below.

Operation consistency has subcategories, with different ways to define the consistency property. We explain these subcategories next.

3.2.1 Sequential equivalence

This subcategory defines the permitted operation results of a concurrent execution in terms of the permitted operation results in a sequential execution—one in which operations are executed one at a time, without concurrency. More specifically, there must be a way to take the execution of all operations submitted by any subset of clients, and then *reduce* them to a sequential execution that is correct. The exact nature of the reduction depends on the specific consistency property. Technically, the notion of a correct sequential execution is system dependent, so it needs to be specified as well, but it is often obvious and therefore omitted.

We now give some examples of sequential equivalence.

Linearizability [12] is a strong form of consistency. Intuitively, the constraint is that each operation must appear to occur at an instantaneous point between its start and finish times, where execution at these instantaneous points form a valid sequential execution. More precisely, we define a partial order $<$ from the concurrent execution, as follows: $op_1 < op_2$ iff op_1 finishes before op_2 starts. There must exist a legal total order T of all operations with their results, such that (1) T is consistent with $<$, meaning that if $op_1 < op_2$ then op_1 appears before op_2 in T, and (2) T defines a correct sequential execution. Linearizability has been traditionally used

to define the correct behavior of concurrent data structures; more recently, it has also been used in distributed systems.

Sequential consistency [14] is also a strong form of consistency, albeit weaker than linearizability. Intuitively, it requires that operations execute as if they were totally ordered in a way that respects the order in which each client issues operations. More precisely, we define a partial order $<$ as follows: $op_1 < op_2$ iff both operations are executed by the same client and op_1 finishes before op_2 starts. There must exist a total order T such that (1) T is consistent with $<$, and (2) T defines a correct sequential execution. These conditions are similar to linearizability, except that $<$ reflects just the local order of operations at each client. Sequential consistency is used to define a strongly consistent memory model of a computer, but it could also be used in the context of concurrent data structures.

The next examples pertain to systems that support *transactions*. Intuitively, a transaction is a bundle of one or more operations that must be executed as a whole. More precisely, there are special operations to start, commit, and abort transactions; and operations on data items are associated with a transaction. The system provides an isolation property, which ensures that transactions do not significantly interfere with one another. There are many isolation properties: serializability, strong session serializability, order-preserving serializability, snapshot isolation, read committed, repeatable reads, etc. All of these are forms of operation consistency, and several of them are of the sequential equivalence subcategory. Here are some examples, all of which are used in the context of database systems.

Serializability [5] intuitively guarantees that each transaction appears to execute in series. More precisely, serializability imposes a constraint on the operations in a system: the schedule corresponding to those operations must be equivalent to a serial schedule of transactions. The serial schedule is called a serialization of the schedule.

Strong session serializability [8] addresses an issue with serializability. Serializability allows transactions of the same client to be reordered, which can be undesirable at times. Strong session serializability imposes additional constraints on top of serializability. More precisely, each transaction is associated with a session, and the constraint is that serializability must hold (as defined above) and the serialization must respect the order of transactions within every session: if transaction T_1 occurs before T_2 in the same session, then T_2 is not serialized before T_1.

Order-preserving serializability [24], also called strict serializability [6, 17] or strong serializability [7], requires that the serialization order respect the real-time ordering of transactions. More precisely, the constraint is that serializability must hold and the serialization must satisfy the requirement that, if transaction T_1 commits before T_2 starts, then T_2 is not serialized before T_1.

3.2.2 Reference equivalence

Reference equivalence is a generalization of sequential equivalence. It defines the permitted operation results by requiring the concurrent execution to be equivalent to a given reference, where the notion of equivalence and the reference depend on the consistency property. We now give some examples for systems with transactions. These examples occur often in the context of database systems.

Snapshot isolation [4] requires that transactions behave identically to a certain reference implementation, that is, transactions must have the same outcome as in the reference implementation, and operations must return the same results. The reference implementation is as follows. When a transaction starts, it gets assigned a monotonic start timestamp. When the transaction reads data, it reads from a snapshot of the system as of the start timestamp. When a transaction T_1 wishes to commit, the system obtains a monotonic commit timestamp and verifies whether there is some other transaction T_2 such that (1) T_2 updates some item that T_1 also updates, and (2) T_2 has committed with a commit timestamp between T_1's start and commit timestamp. If so, then T_1 is aborted; otherwise, T_1 is committed and all its updates are applied instantaneously as of the time of T_1's commit timestamp.

Interestingly, the next two properties are examples of reference equivalence where the reference is itself defined by another consistency property. This other property is in the serial equivalence subcategory in the first example, and it is in the reference equivalence subcategory in the second example.

One-copy serializability [5] pertains to a replicated database system. The replicated system must behave like a reference system, which is a system that is not replicated and provides serializability.

One-copy snapshot isolation [15] also pertains to a replicated system. The requirement is that it must behave like a system that is not replicated and that provides snapshot isolation.

3.2.3 Read-write centric

The above subcategories of operation consistency apply to systems with arbitrary operations. The read-write centric subcategory applies to systems with two very specific operations: read and write. These systems are important because they include many types of storage systems, such as block storage systems, key value storage systems, and processors accessing memory. By focusing on the two operations, this subcategory permits properties that directly evoke the semantics of the operations. In particular, a write operation returns no information other than an acknowledgment or error status, which has no consistency implications. Thus, the consistency properties focus on the results of reads. Common to these properties is the notion of a read *seeing* the values of a set of writes, as we now explain. Each read is affected by some writes in the system; if every write covers the entire data item, then writes overwrite each other and the read returns the value written by one of them. But if the writes update just part of a data item, the read returns a combination of the written values in some appropriate order. In either case, the crucial consideration is the set of writes that *could* have potentially affected the read, irrespective of whether the writes are partial or not; we say that the read *sees* those writes. This notion is used to define several known consistency properties, as we now exemplify.

Read-my-writes [21] requires that a read by a client sees at least all writes previously executed by the same client, in the order in which they were executed. This property is relevant when clients expect to observe their own writes, but can tolerate delays before observing the writes of others. Typically, read-my-writes is combined with another read-write consistency property, such as bounded staleness or operational eventual consistency, defined below. By combined we mean that the system must provide both read-my-writes and the other property. Read-my-writes was originally defined in the context of distributed systems [21], then used in computer architecture to define memory models [19].

Bounded staleness [1], intuitively, bounds the time it takes for writes to be seen by reads. More precisely, the property has a parameter δ, such that a read must see at least all writes that complete δ time before the read started. This property is relevant when inconsistencies are tolerable in the short term as defined by δ, or when time intervals smaller than δ are imperceptible by clients (e.g., δ is in the tens of milliseconds and clients are humans). Bounded staleness was originally defined in the context of database systems [1] and has been used more recently in the context of cloud distributed systems [20].

Operational eventual consistency is a variant of eventual consistency (a form of state consistency) defined using operation consistency. The requirement is that each write be eventually seen by all reads, and if clients stop executing writes then eventually every read returns the same latest value [22].

Cache coherence originates from computer architecture to define the correct behavior of a memory cache. Intuitively, cache coherence requires that reads and writes to an individual data item (a memory location) satisfy some properties. The properties vary across the literature. One possibility [11] is to require that, for each data item: (1) a read by some client returns the value of the previous write by that client, unless another client has written in between, (2) a read returns the value of a write by another client if the write and read are sufficiently separated in time and if no other write occurred in between, and (3) writes are serialized.

3.3 Discussion

We now compare state consistency and operation consistency in terms of their level of abstraction, complexity, power, and application dependence.

3.3.1 Level of abstraction

Operation consistency is an end-to-end property, because it deals with results that clients can observe directly. This is in contrast to state consistency, which deals with system state that clients observe indirectly by executing operations. In other words, operation consistency is at a higher level of abstraction than state consistency. As a result, a system might have significant state inconsistencies, but hide these inconsistencies externally to provide a strong form of operation consistency.

An interesting example is a storage system with three servers replicated using majority quorums [3], where (1) to write data, the system attaches a monotonic timestamp and stores the data at two (a majority of) servers, and (2) to read, the system fetches the data from two servers; if the servers return the same data, the system returns the data to the client; otherwise, the system picks the data with the highest timestamp, stores that data and its timestamp in another server (to ensure that two servers have the data), and returns the data to the client. This system violates mutual consistency, because when there are no outstanding operations, one of the servers deviates from the other two. However, this inconsistency is not observable in the results returned by reads, since a read filters out the inconsistent server by querying a majority. In fact, this storage system satisfies linearizability, one of the strongest forms of operation consistency.

3.3.2 Complexity

Operation consistency is more complex than state consistency. With state consistency, developers gain a direct understanding of what states they can expect from the system. Each property concerns specific data items that do not depend on the execution. As a result, state consistency is intuitive and simple to express and understand. Moreover, state consistency can be checked by analyzing a snapshot of the system state, which facilitates debugging.

By contrast, operation consistency properties establish relations between operations that are spread over time and possibly over many clients, which creates complexity. This complexity makes operation consistency less intuitive and harder to understand, as can be observed from the examples in Section 3.2. Moreover, checking operation consistency requires analyzing an execution log, which complicates debugging.

3.3.3 Power

Operation consistency and state consistency have different powers. Operation consistency can see all operations in the system, which permits constraining the ordering and results of operations. If the system is deterministic, operation consistency properties can reconstruct the state of the system from the operations, and thereby indirectly constrain the state much like state consistency. But doing so is not generally possible when the system is non-deterministic (e.g., due to concurrency, timing, or external events).

State consistency, on the other hand, can see the entire state of the system, which permits constraining operations that might break the state. If the system records all its operations in its state, then state consistency can indirectly constrain the results of operations much like operation consistency.[1] However, it is often prohibitive to record all operations so this is only a theoretical capability.

[1] It is even possible to constrain all operations of the entire execution, though enforcing such constraints would be hard.

3.3.4 Application dependence

State consistency tends to be application dependent, because the properties concern state, and the correct state of a system varies significantly from application to application. As a result, developers need to figure out the right properties for each system, which takes time and effort. Moreover, in some cases there are no general mechanisms to enforce state consistency and developers must write application code that is closely tied to each property. There are two noteworthy exceptions: mutual consistency and eventual consistency. These properties apply broadly to any replicated system, by referring to the replicated state irrespective of the application, and there are general replication mechanisms to enforce such properties.

Operation consistency is often application independent. It achieves application independence in two ways. First, some properties factor out the application-specific behavior, by reducing the behavior of the system under concurrent operations to behavior under sequential operations (as in the sequential equivalence subcategory), or behavior under a reference (as in the reference equivalence subcategory). Second, some properties focus on specific operations, such as read and write, that apply to many systems (as in the read-write centric subcategory). Theoretically, operation consistency *can* be highly application dependent, but this is not common. An example might be an email system accessible by many devices, where each operation (read, delete, move) might have different constraints on their response according to their semantics and the expectations of users.

3.3.5 Which type to use?

To decide what type of consistency to use, we suggest taking a few things into consideration. First, think about the negation of consistency: what are the inconsistencies that must be avoided? If the answer is most easily described by an undesirable state (e.g., two replicas diverge), then use state consistency. If the answer is most easily described by an incorrect result to an operation (e.g., a read returns stale data), then use operation consistency.

A second important consideration is application dependency. Many operation consistency and some state consistency properties are application independent (e.g., serializability, linearizability, mutual consistency, eventual consistency). We recommend trying to use such properties, before defining an application-specific one, because the mechanisms to enforce them are well understood. If the system requires an application specific property, and state and operation consistency are both natural choices, then we recommend using state consistency due to its simplicity.

4 Consistency in different disciplines

We now discuss what consistency means in each discipline, why it is relevant in that discipline, and how it relates to the two types of consistency in Section 3. We also point out concepts that are considered to be consistency in one discipline but not in another.

4.1 Distributed systems

In distributed systems, consistency refers to either state or operation consistency. Early replication protocols focused on providing mutual consistency while many cloud distributed systems provide eventual consistency. These are examples of state consistency. Some systems aim at providing linearizability or various flavors of read-write centric consistency. These are examples of operation consistency.

Consistency is an important consideration in distributed systems because such systems face many concerns that preclude or hinder consistency: clients separated by a slow network, machines that fail, clients that disconnect from each other, scalability of the system to a large number of clients, and high availability. These concerns can make it hard to provide strong levels of consistency, because consistency requires client coordination that

may not be possible. As a result, distributed systems may adopt weaker levels of consistency, chosen according to the needs of applications.

Cloud systems, an interesting type of distributed system, face all of the above concerns with intensity: the systems are geo-distributed (distributed around the globe) with significant latency separating data centers; machines fail often because there are many of them; clients disconnect from remote data centers due to problems or congestion in wide-area links; many clients are active and the system must serve all of them well; and the system must be available whenever possible since businesses lose money during downtime. Because of these challenges, cloud systems often resort to weak levels of consistency.

4.2 Database systems

In database systems, consistency refers to state consistency. For example, consider the ACID acronym that describes the guarantees of transactions. The "C" stands for consistency, which in this case means that the database is always in a state that developers consider valid: the system must preserve invariants such as uniqueness constraints, referential integrity, and application-specific properties (e.g., x is a friend of y iff y is a friend of x). These are flavors of state consistency.

The "A" stands for atomicity and the "I" stands for isolation. Interestingly, atomicity and isolation are examples of operation consistency. Atomicity requires that a transaction either executes in its entirety or does not execute at all, while isolation requires that transactions appear to execute by themselves without much interference. There are many different levels of isolation (serializability, snapshot isolation, read committed, repeatable reads, etc), but they all constrain the behavior of operations.

Although the database systems community separates transaction isolation from consistency and atomicity, in the distributed systems community, transaction isolation is seen as a form of consistency, while in the computer architecture community, a concept analogous to isolation is called atomicity. We do not know exactly why these terms have acquired different meanings across communities. But we suspect that a reason is that there are intertwined ideas across these concepts, which is something we try to identify and clarify in this article.

Consistency is important in database systems because data is of primary concern; in fact, data could be even more important than the result of operations in such systems (e.g., operations can fail as long as data is not destroyed). Different types of consistency arise because of the different classes of invariants that exist in the database, each with its own enforcement mechanism. For example, uniqueness constraints are enforced by an index and checks in the execution engine; application-specific constraints are enforced by the application logic; and mutual consistency is enforced by the replication manager.

4.3 Computer architecture

In computer architecture, consistency refers to operation consistency. A similar concept called coherence is also a form of operation consistency. Consistency and coherence have a subtle difference. Consistency concerns the entire memory system; it constrains the behavior of reads and writes—called loads and stores—across all the memory locations; an example is the sequential consistency property. Coherence concerns the cache subsystem; it can be seen as consistency of the operation of the various caches responsible for a given memory location. Thus, coherence constrains the behavior of loads and stores to an individual memory location.

Coherence and consistency are separated to permit a modular architecture of the system: a cache coherence protocol ensures the correct behavior of the caching subsystem, while the rest of the system ensures consistency across memory accesses without worrying about the cache subsystem.

Consistency and coherence arise as issues in computer architecture because increasingly computer systems have many cores or processors sharing access to a common memory: in such systems, there are concurrent operations on memory locations and data replication across many caches, which lead to problems of data sharing.

5 Conclusion

Consistency is a concern that spans many disciplines, as we briefly described here. This concern stems from the rise of concurrency and replication across these disciplines, a trend that we expect to continue. Unfortunately, consistency is subtle and hard to grasp, and to make matters worse, it has different names and meanings across communities. We hope to have shed some light on this subject by identifying two broad and very different types of consistency—state consistency and operation consistency—that can be seen across the disciplines.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, Sept. 1990.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering*, pages 562–570, Oct. 1976.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, Jan. 1995.

[4] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD International Conference on Management of Data*, pages 1–10, May 1995.

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

[6] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, May 1979.

[7] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, Oct. 1992.

[8] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *International Conference on Data Engineering*, pages 424–435, Mar. 2004.

[9] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, June 1990.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, Sept. 2011.

[12] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[13] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *International Conference on Distributed Computing Systems*, pages 302–309, May 1990.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[15] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *ACM SIGMOD International Conference on Management of Data*, pages 419–320, June 2005.

[16] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *ACM Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[17] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[18] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, Dec. 1995.

[19] A. Tanenbaum and M. V. Steen. *Distributed systems*. Pearson Prentice Hall, 2007.

[20] D. B. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, Dec. 2013.

[21] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*, pages 140–149, Sept. 1994.

[22] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating Systems Principles*, pages 309–324, Nov. 2013.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles*, pages 172–183, Dec. 1995.

[24] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2009.

[25] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, Aug. 2002.

Trade-offs in Replicated Systems

Rachid Guerraoui Matej Pavlovic Dragos-Adrian Seredinschi
{*firstname*}.{*lastname*}@epfl.ch
EPFL

Abstract

Replicated systems provide the foundation for most of today's large-scale services. Engineering such replicated system is an onerous task. The first—and often foremost—step in this task is to establish an appropriate set of design goals, such as availability or performance, which should synthesize all the underlying system properties. Mixing design goals, however, is fraught with dangers, given that many properties are antagonistic and fundamental trade-offs exist among them. Navigating the harsh landscape of trade-offs is difficult because these formulations use different notations and system models, so it is hard to get an all-encompassing understanding of the state of the art in this area.

In this paper, we address this difficulty by providing a systematic overview of the most relevant trade-offs involved in building replicated systems. Starting from the well-known FLP result, we follow a long line of research and investigate different trade-offs, assembling a coherent perspective of these results. Among others, we consider trade-offs which examine the complex interactions between properties such as consistency, availability, low latency, partition-tolerance, churn, scalability, and visibility latency.

1 Introduction

On-line services are continuously growing in scale and are being used by more and more users worldwide. Given their unprecedented size, these services must be deployed in the form of scalable distributed systems. Examples include social networks, search and aggregation, collaborative tools, entertainment, storage and backup, and many more. Informally, to achieve success, such services must, first and foremost, provide a smooth user experience. This requirement means that these services should ensure correct, uninterrupted operation, with fast response times. Technically, this translates to the important design goal of *high availability*, which necessarily implies tolerance to faults, including network and replica failures, and low latency of user requests.

Towards achieving this design goal, the common approach is to use geo-replication, i.e., to replicate both data and computation in different geographic regions. As such, the system may isolate faults and ensure that users in unaffected regions of the world can continue to access the service with low latency. On the surface, it might seem like geo-replication successfully solves all the issues faced by these systems. Unfortunately, this is not the case. The truth is that replication brings up the burden of ensuring *data consistency* and there is a fundamental friction between consistency and availability. In such a system, trying to guarantee high availability and uphold low latency without sacrificing strong consistency is, in fact, a hopeless endeavor [1, 12, 20].

These two properties—availability and consistency—are not the only attributes which weigh upon a replicated system; they are merely the tip of the iceberg. Depending on the specifics of the higher-level application,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

many other properties are equally or even more important. Latency, throughput, partition-tolerance, churn-tolerance, scalability, bandwidth usage, and energy-efficiency, are examples of other relevant properties of a replicated system. A comprehensive overview of how all these properties blend together in real systems is impossible, given that most of them have a continuous scale and even their precise definitions vary throughout the literature. Some of these properties can be achieved at the same time, while some are inherently antagonistic. There is a long line of research which focuses on how the most important design goals interact. Even a simple replicated system with a select few design goals could encounter inescapable conflicts between these goals, and from these conflicts, an array of trade-offs arise. This article outlines the most important trade-offs, puts them in perspective, and discusses the limits involved in engineering replicated systems.

We structure our investigation on trade-offs to start as general as possible, from the most wide-ranging trade-offs. We follow the research trends over the years, as they pursue the demands of real-world applications, by progressively narrowing down the context. This directs our investigation towards several specialized trade-offs, many of which can be seen as refinements of earlier results.

1.1 Overview on Trade-offs

In the broader context of distributed systems, the properties of any algorithm can be separated into two classes: *safety* properties and *liveness* properties [26]. Informally, safety properties convey the requirement that nothing bad should ever occur, whereas liveness properties state that something good should (eventually) happen. For instance, availability of a storage service is a liveness property, while the consistency guarantees of this service are a safety property. In practice, safety has to do with the correctness of a system, while liveness refers to a system's performance (i.e., the service should deliver results and make some progress).

The separation of distributed system properties in these two categories is relevant to our discussion as most, if not all, of the trade-offs encountered in distributed systems are, at a sufficiently high level of abstraction, safety versus liveness trade-offs. This is intuitive also from a practical perspective: the stronger the correctness guarantees a system should provide, the harder it is to implement those guarantees. The price to pay is performance, since implementing stringent correctness guarantees entails bigger complexity and resource overheads, compared to more lenient guarantees.

The canonical example of a result which puts safety and liveness at odds with each other is the FLP impossibility [18], which is the first trade-off we address. This result applies specifically to the *consensus* problem in an asynchronous system, whereby replicas need to reach agreement on a single value, called decision, among multiple proposals. Briefly, FLP specifies that either safety (no process takes a wrong decision) or liveness (all correct processes eventually reach a decision) of consensus must be sacrificed if processes are prone to failure. Consensus is at the center of the state machine replication (SMR) approach, which is a general technique suitable for replicating any deterministic service [29].

After FLP and general SMR, we narrow down the context to that of a storage service, with operations restricted to `read` and `write`. In other words, in terms of semantics, we narrow our focus from a general setting towards storage-oriented services, which are an important class of replicated systems. We first examine the CAP theorem [12, 20] and then we discuss a formulation, called PACELC [1], which descends directly from CAP. In discussing these two results, the spotlight is mainly on strong forms of consistency, and how this safety property impacts liveness. Next, we confine our discussion to a specific form of consistency, namely causal consistency. Therefore, we aim our attention at the CAC result [29] and the throughput versus visibility—or shortly, TV—trade-off [6, 11].

Then, we shift the discussion to consider dynamic storage systems, where nodes may join and leave, i.e., churn, at fast rates. In such systems, churn creates further friction among the service properties, and gives rise to a trade-off between scalability, allowable churn rate, and availability, or SCA [9].

Finally, we discuss a generalization of this last result. We switch back from storage-oriented semantics to general-purpose semantics and propose a new trade-off which is applicable to any replicated system subject

	FLP (§3)	CAP (§4.1)	PACELC (§4.2)	CAC (§5.1)	TV (§5.2)	SCA (§6.1)	RCR (§6.2)
Semantics	General	← - - - - - - - - - - - - - - - Storage - - - - - - - - - - →					General
Membership	← - - - - - - - - - - - - - - Static - - - - - - - - - - - - - - →					← - - - - Dynamic - - →	
Replication	← - - - - - - - - - - - - - - Full - - - - - - - - - - - - - - →					← - - - - - Partial - - - →	

Table 1: Perspective on the discussed trade-offs.

to churn (§6.2). We do so by introducing the notion of *reconciliation mechanism*, which allows us to capture the conflicting interaction between churn and robustness (i.e., fault-tolerance) in such a system. Any system which is subject to churn has a reconciliation mechanism; reconciliation can take various forms, such as a load-balancing scheme, replica reconfiguration [21] or regeneration [35], or the cuckoo rule [4]. Reconciliation is triggered by churn events and is necessary to uphold robustness in the face of membership fluctuations. In other words, this mechanism affects both churn and robustness: for instance, it may constrain the former by curbing the churn rate; likewise, reconciliation influences the latter by ensuring either strict measures or best-effort measures to preserve robustness. Our findings point to the existence of a trade-off between reconciliation, churn, and robustness (RCR). The specifics of the reconciliation mechanism will tip the scale of this trade-off either in favor of churn or in favor of robustness.

In Table 1, we give a concise perspective on all the trade-offs we consider. We use three coarse-grained dimensions to trace these results: semantics, membership, and replication. Before diving into the main study of these trade-offs, however, we introduce some preliminary notions (§2). The first trade-off we present is the FLP result (§3), after which we move on to results on strongly consistent storage systems, namely CAP and PACELC (§4). We then study causal consistency with its associated trade-offs (§5). We also take dynamic membership into account, by investigating what happens in the presence of churn and propose a general trade-off concerning churn (§6). Finally, we conclude with a summary of the presented trade-offs (§7).

2 Preliminaries

In this section, we give the necessary background and prepare the ground for our study of trade-offs. Given that these trade-offs span across multiple system models, we describe here the minimal variations on these models which are necessary for our discussion. We introduce a general model and then present a few refinements on it, which become relevant later on in our discussion, as we report on subsequent trade-offs.

Throughout the whole article, we assume a service running as a distributed system. Initially, we consider the service state being replicated at all the nodes of this system, called *replicas*. In other words, we consider a *full* replication model, with each replica storing a complete copy of the service state. The system is asynchronous, which means that no assumptions can be made regarding relative computation or communication speeds. A common aspect of all the trade-offs we consider is that neither of them explicitly assumes synchrony. This is a pragmatic choice, especially relevant for today's online services, since the prevailing approach—as highlighted above (§1)—is to employ geographic replication, where achieving synchrony is remarkably difficult [7].

In terms of semantics, we model our service in two iterations. First, we represent the service as a general state machine, which is relevant for the opening result, the FLP impossibility. After that, all subsequent results (except RCR, §6.2) model the service as a key-value store—or alternatively, a set of shared registers [27]. Each register is accessible by a unique key and stores a data *object*, supporting `read` and `write` semantics on that object.

The set of replicas composing the distributed system is also a moving target. This is another aspect which we refine as we go along: at the beginning, our discussion centers on trade-offs that generally apply even if the replica set (i.e., membership) is *static*. Later on, we examine trade-offs which consider *dynamic* (i.e., changing in time) membership and discuss the friction between this dynamicity and some other system properties. Once

we consider that the set of replicas is dynamic, we also drop the full replication assumption. We do so to adhere to the system model of these dynamic systems, which employ *partial* replication, as is common in file sharing or peer-to-peer systems. This distinction between partial and full replication, however, is not crucial, and the same trade-off would still apply under either of these replication models.

3 FLP

As we stated earlier, FLP[1] puts safety at odds with liveness, by proving the impossibility of a deterministic algorithm solving consensus in an asynchronous system if even a single replica can crash [18]. The actual impossibility result is more precise in its definition, as we shall see; this high-level perspective, however, is instructive, and helps us tie this result together with all the other trade-offs. In this section, we first investigate this impossibility result. Then, we veer the discussion towards a practical standpoint: we study what are the implications FLP has for system designers, and examine a trade-off arising from this result.

The main insight underlying this result is the following: in an asynchronous system, message delays can be arbitrarily long, and thus it is impossible to distinguish a crashed replica from a slow one. Consider a system where all replicas are fast, except one of them which is particularly slow. On one hand, if the slow replica is falsely suspected to have crashed, it could lead to disagreement: the fast replicas will agree among themselves, while ignoring the suspected replica ,violating safety. On the other hand, if the fast replicas do not suspect the slow replica to have crashed and simply wait for it, then they run the risk of waiting indefinitely: owing to the nature of the asynchronous system, the slow replica might—in fact—have crashed, and the other replicas never reach a decision, i.e., the algorithm does not terminate, breaking liveness.

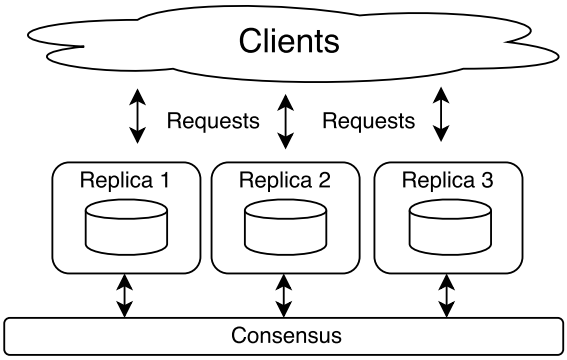


Figure 1: State machine replication: replicas use a consensus protocol to agree on a common order on operations, e.g., client requests.

FLP is an important stepping stone in our understanding of replicated systems because consensus plays a central role in *state machine replication* [29], which, in turn, is a basic and general method towards ensuring high-availability and tolerating failures. In state machine replication (SMR), the service is modelled as a deterministic state machine. All replicas hold a copy of this state machine (i.e., employing full replication) and use consensus to agree on the order of operations which they apply on the state machine (see Figure 1). This agreement step is critical, as it ensures that replicas keep their states synchronized; furthermore, the operations must be deterministic, otherwise the state at replicas could diverge. Consequently, even if a replica fails, it will not incur service downtime, since the other replicas can continue operation.

For system designers, this impossibility result has ample implications, partly due to the generality of the consensus problem, and partly because of the widespread use of SMR. Namely, other important problems in replicated systems, such as atomic broadcast or group membership, can be reduced to consensus, thus extending the reach of this result [23]. Since SMR can be used to replicate *any* deterministic service—achieving high availability for that service—it is an essential technique in replicated systems. In practice, FLP highlights the chief importance of synchrony assumptions in distributed systems, forcing engineers to reason about worst-case scenarios and to understand what conditions are necessary to be able to solve consensus.

To reach an alternative perspective on FLP, we start from our earlier observation that SMR can model *any* deterministic service. This generality of SMR is both an asset and a burden: on one hand, SMR lends itself to almost universal application; on the other hand, some services do not necessarily need this generality, and benefits

[1]The name of this result is derived from the last names of its authors: Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson.

may be reaped by restricting the allowed semantics. A notable example is storage services supporting read/write operations on single objects. These services, even with the strongest correctness condition (linearizability [25]), can be implemented despite asynchrony or failures [3]. The intuition why replicated storage is easier to implement than a general replicated state machine is that *read* and *write* are very simple operations. In particular, their outcome does not in any way depend on the previous state of the accessed object, and thus atomic *read-modify-write* behavior is not possible. Implementing an atomic transaction, for example, is not achievable using only independent read and write operations. Realizing the weaker semantics of read/write objects, we can regard FLP as a trade-off between service semantics, synchrony assumptions, and fault tolerance. Hence, in practice, we can choose to trade a certain side of the trade-off (e.g., service semantics) to the benefit of the other (e.g., synchrony assumptions).

Briefly, by lessening the requirements on semantics, a service such as a read/write storage can achieve fault-tolerance despite asynchrony. Fortunately, many applications tolerate weakened semantics and do not require a full read-modify-write model as SMR offers. For example, consider a website which recommends movies to watch based on a user's rating history. It is preferable that the website loads fast and works despite asynchrony and failures, even though it might give imperfect recommendations. It is not an issue if the recommended titles do not consider the entire rating history or the newest arrivals. For such a system, it is wise to give up read-modify-write semantics to the benefit of better performance. This subtle interplay between the semantics of a storage service and other properties is the subject of the next trade-offs which we address.

4 Strong Consistency in Replicated Storage Systems

The FLP result states that, in the given system model, replicating a general state machine is not possible. FLP does not apply, however, to storage systems with simple read/write objects, as these systems do not require solving consensus. Hence, the following question ensues: what can we guarantee in terms of storage system properties? The CAP theorem [12, 20] sheds light on this matter, stating that there is a fundamental trade-off between strong consistency, availability, and partition-tolerance of a replicated storage system. Namely, it demonstrates the impossibility of achieving all three properties at the same time. CAP was first stated by Eric Brewer [12] as a conjecture, and later proven by Gilbert and Lynch [20].

4.1 CAP: Consistency, Availability, and Partition-Tolerance

The main idea behind this result is that, in order to implement a replicated read/write object with strong consistency guarantees, communication among the replicas is necessary. If some replicas become isolated due to a network partition, upon receiving a request, these replicas have two choices: either (1) to return a value without coordinating with others, possibly violating consistency, or (2) to return no value at all, violating availability. Thus, to achieve partition-tolerance, one must sacrifice either consistency or availability. Alternatively, we can also view CAP through the lens of a safety versus liveness trade-off, because strong consistency is a classic safety property and availability is a typical liveness property.

In practice, this trade-off lies at the root of the distinction between ACID [24] and BASE [30] systems. ACID denotes a set of four properties: atomicity, consistency, isolation, and durability; it represents the gold standard in terms of transactional systems properties. Most of the traditional database management systems are ACID, offering strong consistency (isolation) guarantees. This is convenient for application developers since they are exposed to clear semantics, and do not need any explicit mechanism to deal with inconsistencies. ACID systems, however, must sacrifice availability when network partitions arise.

BASE systems (basically-available, soft state, eventually consistent), such as distributed key/value stores, are willing to trade in some data consistency guarantees in order to increase system availability and performance [30]. Intuitively, when a partition isolates a replica from the rest of the system, that replica—and all others as well—

may continue to serve client requests without employing coordination. Due to absence of coordination, the system sacrifices consistency and burdens the client with the task of dealing with inconsistencies. Both ACID and BASE systems are useful in specific scenarios, and it is actually a common practice to blend both of these models when building real-world application stacks [2].

4.2 PACELC: a Refinement of CAP

The FLP and CAP theorems are the capstones of any modern distributed system. The impossibilities which these two theorems cover are general and far-reaching. FLP proves that asynchrony and failures are a dangerous combination, prohibiting a deterministic consensus algorithm; CAP, in the same vein, teaches us that when a partition occurs in a replicated storage system, we must trade between strong consistency and availability.

PACELC [1] descends directly from CAP and suggests that even in the absence of partitions, i.e., during healthy operation, we still face a trade-off, between consistency and latency. Briefly, PACELC captures the following double trade-off: if partitions occur, then trade between availability and consistency; else trade between latency and consistency. The insight behind PACELC is that even during healthy operation, the coordination required for ensuring strong consistency among replicas results in an increased request latency.

Consider, for instance, a geo-replicated storage system with replicas spread around the globe. If this system guarantees strong consistency, then it requires synchronous inter-replica coordination, resulting in a latency penalty upwards of 100ms. In contrast, such a system could satisfy eventual consistency [15] without any synchronous coordination, and thus zero latency overhead. Low latency is a critical goal in many systems, especially so in users-facing web services, since humans perceive even a slight deterioration in latency on the order of hundreds of milliseconds [32]. Such latency numbers are common if a service uses synchronous geo-replication, given the fundamental limitation on communication speed imposed by the speed of light.

The PACELC formulation recognizes the importance of low latency in modern services, and weaves this dimension into the well-known CAP trade-off. Thus, PACELC encompasses two important trade-offs: one between consistency and availability, and another between consistency and low latency. The former is inherited directly from CAP, with the important observation that this trade-off is only relevant during partitions. The latter trade-off is relevant during normal-case operation: even in the absence of failures, as we observed in the example earlier, latency can reach hundreds of milliseconds if strong consistency is required, which can be prohibitive for some applications.

Conceptually, we can regard unavailability as high (unbounded) latency, which allows us to consolidate the PACELC formulation as a single trade-off. In this light, we see a general trade-off between consistency and latency. For a strongly consistent system, the latency depends on the coordination delay. In the extreme case of a partition—where coordination takes infinite time—the system is unavailable, with unbounded latency. Thus, to achieve low latency, one must sacrifice strong consistency in favor of weaker consistency models, which eschew the need for costly coordination. A similar latency penalty also arises in the context of consensus: even when a solution is possible (i.e., by assuming synchrony), such a solution nevertheless incurs high latency [19].

5 Causal Consistency

From the CAP impossibility result, the following question immediately follows: what is the strongest consistency that is achievable while maintaining availability and partition tolerance? The answer to this question is *causal consistency* [13]. In other words, no consistency model stronger than causal can be ensured while also satisfying availability during network partitions [29]. This model is weaker than strong consistency, but is useful in practice because it provides intuitive semantics: it ensures that, for any operation t, all the operations which causally precede t are guaranteed to take effect before t. Alternatively, causal consistency ensures three guarantees: (1) monotonic reads and writes, (2) read-your-writes, and (3) writes-follow-reads [13]. Indeed, recent work in storage systems embraces causal consistency as the sweet spot on the consistency spectrum [16].

5.1 CAC: Causal Consistency, Availability, and Convergence

The causal consistency model seems to definitively reconcile the antagonism between consistency and availability. This result, however, is not intuitive, because we can formally define even stronger models which are also available during partitions; for instance, one can define consistency models to comprise only safety properties. Such a definition provides a loophole: replicas do not need to ever coordinate, and thus it allows an implementation which is both available and consistent (stronger than causal) despite partitions. The flaw in these consistency models is that they are not useful in practice, since replicas can forever diverge. To model this aspect of usefulness, Mahajan et al. define a property called *convergence* [29].

At an abstract level, convergence captures the coordination, often called synchronization, among replicas. Coordination can be asynchronous (in the background, periodically or in batches), or synchronous (on the critical path of every operation), depending on the desired consistency level, and it is a necessary step towards achieving consistency. Put differently, convergence ensures that replicas (and clients) observe each other's updates. Some consistency protocols encompass a weak version of convergence, where coordination is asynchronous, i.e., if updates stop, then all replicas reach the same state, as in eventual consistency [15]. Other protocols, such as those guaranteeing linearizability, require a stronger variant of convergence, whereby replicas must coordinate synchronously and agree upon a total order on the sequence of updates.

Informally, the property of convergence prescribes that all replicas shall agree on a *common state*. This state should include the updates from all replicas, and, in the case of a causally consistent system, it should adhere to a partial order of operations that satisfies causality. Since the ordering is partial, this convergence property allows replicas to temporarily diverge, i.e., concurrent updates on the same object can lead to conflicting versions of that object. Such a convergence property thus allows for asynchronous coordination, while guaranteeing usefulness in the sense that the replicas eventually reach a common state, applying each other's updates in a causal order.

To sum up, we can regard this result of Mahajan et al. about causal consistency, availability, and convergence [29] as a refinement of CAP which also takes into account the notion of convergence. A system could support a consistency model which is formally stronger than causal, and also ensure high-availability (tolerate partitions), but for the price of sacrificing convergence. To maintain convergence (i.e., usefulness), causal is the strongest implementable consistency in a highly-available system [29]. In the following, we will inspect this notion of high availability in a stricter sense, and discuss causal consistency in slightly more depth.

A Closer Look at Causal Consistency:

The system model of the CAC result has an important aspect which we did not cover earlier. Specifically, the model in this trade-off makes no distinction between clients and system replicas, which means that every node in the system is both a replica of the service and a client of the service. This assumption has the veiled implication that clients are always connected to one of the replicas—a unique replica, to be precise. If, however, we choose to distinguish between the two roles of clients and replicas and draw a definite boundary between the system and its clients, then the argument for causal consistency is no longer watertight. In practice, indeed, clients are separate entities, which may lose communication to a system replica, and which, moreover, can switch from one replica to another. Such switching happens usually due to a load-balancing mechanism [2, 33].

In light of this observation, it is necessary to distinguish between two system models. First, if clients and replicas are embodied in a single, combined role, e.g., by being co-located on the same machine, then clients always access the system via the same node—their co-located replica. In such a case, we say that clients stick to the same replica, and the system provides *sticky availability* [5]. Broadly speaking, these are systems where the client is co-located with the service, e.g., in the same datacenter, rack, or machine; peer-to-peer systems also match this model, because every node is both a client and a server. In the second model, clients and replicas are distinct roles, under separate failure domains, which may be parted by network failures. In this model, clients are permitted to switch between different replicas, and we say that the system provides *high availability*. Figure 2 depicts the contrast between these two models.

The distinction between these two availability models is important because each model has its own limit in terms of strongest achievable consistency. Specifically, the causal consistency result we highlighted earlier applies to the sticky availability model. This means that clients may rely on always being able to access—i.e., stick to—the same replica. In a high availability model, this result does not hold anymore. If a client switches from a service replica to an alternative one, it could mean forsaking causal consistency: this can happen if the alternative replica lacks some of the client's updates, and thus it causes the breaking of the read-your-writes guarantee implied by causal consistency [13].

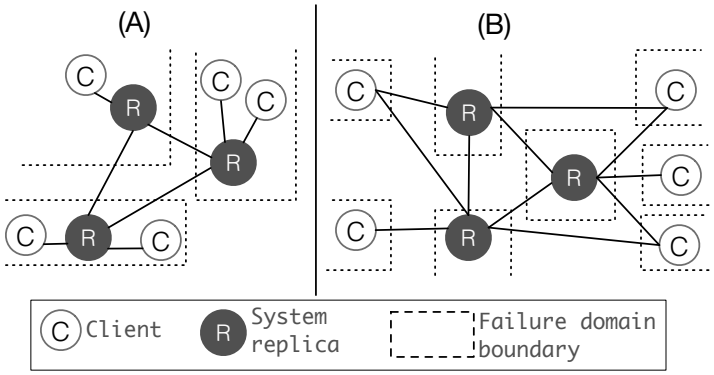


Figure 2: Contrast between a system model under sticky availability **(A)** and under high availability **(B)**. With sticky availability, clients `C` share the failure domain with one of the system replicas `R`.

Our belief is that the problem of finding the strongest consistency model achievable under high availability is very important; yet, to the best of our knowledge, this is still an open issue. At first glance, our investigation of this topic indicates that consistent prefix [34] is the answer. Under consistent prefix guarantees, a client always observes a totally ordered sequence of updates for every data object, but this sequence may lack some of the latest updates, i.e., it does not ensure freshness, and it does not capture causality constraints either. Consequently, compared to causal consistency, consistent prefix can be considered less useful in practice, since it provides neither read-your-writes nor monotonic reads across subsequent client operations. These two guarantees are important because they prohibit anomalous scenarios where the natural cause-effect relation between events is inverted, i.e., an effect may be observed to precede its respective cause [6], and such an inversion would contradict our expectations.

The consistent prefix model can be easily achieved, for instance, using a scheme where a primary replica establishes a total order on write operations and propagates this order asynchronously. The model is achievable under high availability because different replicas may reveal different states of each object. An analogy can be drawn between this consistency model and snapshot isolation [17], which is well-known in the database community, with the caveat that consistent prefix applies to read/write single-key operations [34]. In our future work, we plan to further examine this argument and also consider other alternative paths to an answer.

5.2 TV: the Throughput versus Visibility Trade-off in Causal Consistency Protocols

Despite evading the consistency/availability trade-off, causal consistency still runs into trade-offs of its own. The crux of the issues in causally-consistent systems are due to metadata management. In contrast to most other forms of consistency—such as linearizability, per-key sequential consistency, or eventual consistency—protocols for causal consistency rely on metadata to track causal dependencies between operations and establish a partial order among operations, an order which respects causality. This introduces two main difficulties: (1) each update operation generates some new metadata, and (2) this metadata has to propagate to every replica. The latter is particularly problematic and it arises because causal consistency entails full replication: every node replicates every object, otherwise availability would be sacrificed during partitions [6].

In this context, the trade-off is between the throughput of the storage system and visibility latency [6, 11]. Visibility latency denotes the delay of propagating and applying updates across nodes. Intuitively, update visibility is modeled by convergence (see §5.1), and it captures the delay of propagating both data and metadata [14]. In the case of metadata, a system can choose different tracking granularities. If metadata is fine-grained, then the system tracks causal dependencies among *individual* objects, so each replica may apply an incoming update

as soon as it satisfies the dependencies for that update. In this case, visibility latency is minimized, since only actual dependencies need to be taken into account. The throughput, however, is limited: fine-grained metadata is large and requires high bandwidth, which can lead to the metadata explosion problem [28]. If, on the other hand, metadata is coarse-grained, then the system tracks dependencies among *groups* of objects; this reduces the metadata size, requires less bandwidth, and allows the system to scale better in terms of throughput. The downside is that false dependencies may ensue if objects are grouped, causing the visibility latency to go up [11].

In order to circumvent this trade-off, storage systems have to limit the metadata size—but without introducing false dependencies, i.e., without grouping objects. One way to achieve this is through application-specific *explicit* dependencies, whereby every update includes just the actual application-level dependencies, and not the entire causal past. It is arguable, however, whether applications are equipped to handle such a mechanism, as it requires changes in the application logic [6]. Unless applications include mechanisms to track explicit dependencies among all possible combinations (and sequences) of operations, causality may be broken.

6 Dynamic systems

So far, we only considered systems with static membership. Many real-world systems, however, experience significant *churn*: nodes joining and leaving the system at run-time. This behavior is especially prominent in (but not limited to) peer-to-peer systems. There are many causes that make the membership of system dynamic: node failures and recovery, system scale-out, software or hardware maintenance, or simply the behavior of its users, all require the system to cope with membership changes. Dynamic membership, however, is another potential source of problems for system engineers, as it can interfere with other desirable system properties.

6.1 SCA: Scalability, Churn, and Availability

This trade-off, identified by Blake and Rodrigues [9], is in the context of highly available peer-to-peer storage systems. The formulation is as follows: as a system becomes more dynamic, i.e., as the set of replicas changes at a growing pace, the system becomes less scalable. This trade-off puts emphasis on the necessity of preserving data redundancy, which means that each data object should have, at all times, a minimum number of copies. As it turns out, if nodes churn at a very high rate, then it becomes impossible to maintain a minimum number of copies per each object, since the per-node bandwidth is fixed over time, posing a bottleneck. Hence, the system may only scale by either reducing availability—i.e., forsaking a minimal redundancy level—or by throttling churn—i.e., using a form of rate limiting mechanism to prevent too many nodes from joining and leaving.

The main idea behind this trade-off is based on a simple argument. A node joining the system needs to download the data it is about to store from another node. When a node departs from the system, other nodes have to create copies of the data which the departing node stored, by downloading it from available replicas. This is necessary in order to maintain an appropriate replication factor required for data availability.

To be more precise, assume a fixed amount of data to be stored by the system, while the network bandwidth allocated by each peer is also fixed. By *session time* we express the period of time which a peer spends as a member of the system. Then, higher churn translates to shorter session times, which, in turn, means that more data has to be moved around in order to maintain data availability. In the same vein, if we fix the number of peers belonging to the system at each time, then per-node bandwidth cost increases as churn goes up.

Using conservative estimates (e.g., considering only maintenance bandwidth and disregarding traffic generated by client requests) on the bandwidth needed to maintain data highly available, Blake and Rodrigues show that this bandwidth quickly becomes prohibitive with increasing churn [9]. For example, if each node provisions a bandwidth of 200KBps, then a million peers with session times of one month are necessary to store 1000TB of data with a replication factor of 20. Decreasing the session times to 1 day, only 50 TB could be maintained. Even employing optimizations such as erasure-coding or distinguishing between node departure and temporary

downtime does not make a significant difference. Thus, as a broad conclusion, highly dynamic peer-to-peer systems are unsuitable for large-scale data storage.

6.2 RCR: Robustness, Churn, and Reconciliation

In this section, we investigate how churn affects other properties, but in a more general framework, not restricted to storage, and thus extend the last trade-off we considered (SCA, §6.1). We switch back from a shared register context (with read/write semantics) to a general replication setting. We retain the other properties of the system model: *dynamic* membership and *partial* replication, since churn presumes a dynamic set of replicas. Our findings point to a trade-off between churn, robustness, and reconciliation. To explain this new trade-off, we begin with a short examination of our terms (robustness and reconciliation), and then we dive into some examples.

The common approach of applying replication in a large dynamic system is to partition the nodes into multiple *replication groups*, each group replicating a part of the system state [8, 21]. The rationale behind this separation is twofold: (1) performance, since full replication incurs substantial overhead and scales poorly; and (2) robustness, as each failure is isolated in its group, preventing a system-wide proliferation of faults. To avoid ambiguities, we distinguish between fault-tolerance (which denotes the ability of a replication group to withstand failures) and robustness (the property of the system as a whole of withstanding failures).

To ensure fault-tolerance, a replication group requires a minimal threshold of its members to be non-faulty and participate in the replication protocol [10]. In general, increasing the size of the replication group also increases its fault-tolerance, thus making the whole system more robust. Intuitively, more nodes are allowed to fail before violating the minimal threshold of non-faulty nodes. A small replication group is arguably less fault-tolerant, as fewer failures are sufficient to topple the threshold.

Our intuition is that the simple presence of churn may jeopardize robustness. The argument is as follows. Churn necessarily brings along fluctuations in the sizes of replication groups: a departing node triggers a decrease in the size of its group, while a joining node causes its hosting group to increase in size. Often, the system has limited or no control over the churning nodes. For example, in peer-to-peer systems, nodes may freely decide to join or leave at any time. Consider a small group of just three replicas storing important documents. If a replica leaves the system, the data remains present only on two other nodes; the departure or failure of one of those, under an asynchronous model, would potentially make the data unavailable, or even forever lost. To keep robustness undisturbed in the face of churn, replication groups must remain fault-tolerant despite join and leave events. This is the purpose of a *reconciliation* mechanism.

In the above example of three-way replication, reconciliation might work as follows: upon the departure of a replica, a replacement node has to take its role in the replication group, downloading the important document from the live replicas. This is essential to replenish groups and prevent them from dying out. In this case, reconciliation might involve actions such as: detecting departure, selecting a replacement node, reconfiguring the group to insert the replacement, and downloading the document (i.e., synchronizing states). Note that while this mechanism executes, the group should be locked, to prevent the remaining replicas from departing. In other words, in this case the reconciliation serializes the churn events, throttling the churn rate, to the benefit of robustness. Algorithm 1 summarizes the general pattern of group reconciliation upon churn events.

Algorithm 1 Basic interplay between churn, robustness, and reconciliation.

Require: `E` (a churn event), and G (a replication group).

```
1: G' = apply E to G
2: if robustness(G') < min_robustness then
3:     reconcile(G')
4: end if
```

To enable higher churn rates, we may speed up reconciliation by making it non-blocking. After the departure of a replica, such a best-effort reconciliation would run in the background, hoping that the replication group survives until reconciliation finishes. One may easily imagine generalizations of this approach, e.g. limiting the number of concurrent churn events or setting a lower bound on some measure of fault-tolerance beyond which

the group becomes locked. Generalizing even further, different reconciliation mechanisms may be considered to strike a trade-off between fast and robust reconciliation.

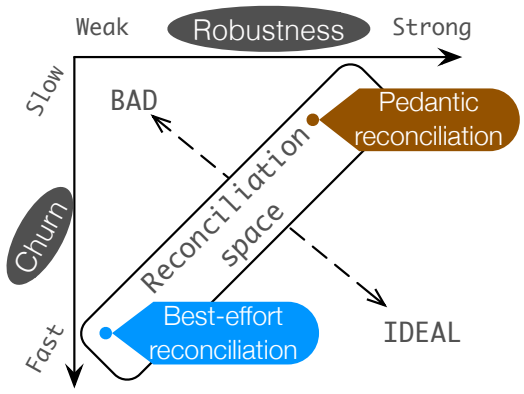


Figure 3: Robustness, churn, and reconciliation trade-off. Reconciliation mechanisms determine a trade-off between robustness (x-axis) and churn (y-axis).

We sketch this basic interaction between churn, robustness, and reconciliation in Figure 3. A system designer may pick a point on the churn-robustness scale by choosing a particular reconciliation implementation. Several notable reconciliation mechanisms are: simple data download (as in the argument for the SCA trade-off §6.1), group replenishing (as in the three-way replication we discussed earlier), or gossip-based membership (used to lazily propagate membership changes and keep the network of replicas connected, as in Dynamo [15]). Note that replenishing replication groups after node departures is not the only case where this principle applies. Counter-intuitively, even new nodes being added to a group may be threatening its fault-tolerance, such as in a Byzantine-tolerant system [22].

In general, a lightweight reconciliation mechanism favors churn: the system would not prevent or delay structural changes, and the reconciliation would be executed on a *best-effort* basis, e.g., as in the case of gossip-based systems. Such a system trades robustness for churn; specifically, in an gossip-based system, if the churn rate is too high, it may lead to cases when the network becomes disconnected, and nodes cannot reach each other, breaking robustness [36]. A *pedantic* reconciliation mechanism, on the other hand, favors robustness: all churn events become expensive operations—thus limiting the churn rate—but the system upholds robustness rigidly, as in the cuckoo rule [4].

7 Conclusion

We discussed the most important trade-offs involved in engineering modern replicated systems. We started with FLP, a fundamental result stating the impossibility of solving consensus in an asynchronous, failure-prone system. As we observed, consensus is a central building block for maintaining consistency among replicas in state machine replication (SMR), i.e., under a general semantics model. Accordingly, we can see FLP as a trade-off between strong consistency for general-purpose replication, synchrony assumptions, and fault tolerance.

We then turned our attention to storage systems with read/write semantics. Here, instead of the consensus problem, we switched to a data consistency problem. We examined the CAP theorem, which postulates that strong consistency, availability, and partition-tolerance are not achievable at the same time. We also discussed two refinements on CAP. The first, called PACELC, states that even in the absence of partitions, replicated storage systems must nevertheless face a trade-off between consistency and latency. The second, called CAC, defines the important notion of convergence. If we ignore convergence, we can devise consistency models that are rather strong while still permitting high availability, yet are useless in practice. Once we add convergence to the mix of system properties—as CAC proved—causal consistency is the strongest consistency model which escapes the CAP impossibility. Concerning causal consistency, we also discussed an essential trade-off between throughput and update visibility (TV).

We also investigated systems with dynamic membership, where churn plays an important role, and which use partial replication. We studied the trade-off between scalability, churn, and availability (SCA), which essentially states that systems susceptible to high churn—such as peer-to-peer—are not suitable for highly available large-scale storage. Finally, we presented a generalization of this trade-off, called RCR, showing that churn not only inhibits scalability, but the overall robustness of a system, in the general context of replicated systems.

References

[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, (2):37–42, 2012.

[2] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *HotOS XV*, 2015.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[4] B. Awerbuch, C. Scheideler. Towards a scalable and robust dht. *Theory of Computing Systems*, 45(2):234–260, 2009.

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *VLDB*, 7(3), 2013.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *ACM SoCC*, 2012.

[7] P. Bailis and K. Kingsbury. The network is reliable. *ACM Queue*, 12(7):20, 2014.

[8] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable State-Machine Replication. In *IEEE DSN*, 2014.

[9] C. Blake, R. Rodrigues. High availability, Scalable Storage, Dynamic Peer Networks: Pick two. In *HotOS IX*, 2003.

[10] G. Bracha, S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[11] M. Bravo, L. Rodrigues, and P. Van Roy. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Middleware Doctoral Symposium*, 2015.

[12] E. A. Brewer. Towards robust distributed systems (invited talk). In *PODC*, 2000.

[13] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (IEEE PDP)*, 2004.

[14] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo. In *SOSP*, 2007.

[16] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *ACM SoCC*, 2014.

[17] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[19] E. Gafni, R. Guerraoui, and B. Pochon. From a static impossibility to an adaptive lower bound: The complexity of early deciding set agreement. In *STOC*, 2005.

[20] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[21] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, T. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

[22] R. Guerraoui, F. Huc, A.-M. Kermarrec. Highly dynamic distributed computing with byzantine failures. In *PODC'13*.

[23] R. Guerraoui and A. Schiper. Consensus: the Big Misunderstanding. In *IEEE FTDCS*, 1997.

[24] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 1983.

[25] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[26] L. Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), 1978.

[28] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI*, 2013.

[29] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. *University of Texas at Austin*, Technical Report TR-11-22, 2011.

[30] D. Pritchett. Base: An acid alternative. *ACM Queue*, 2008.

[31] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[32] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[33] P. Shuff. Building a Billion User Load Balancer, 2013. SREcon15.

[34] D. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[35] H. Yu and A. Vahdat. Consistent and automatic replica regeneration. *ACM Transactions on Storage*, 1(1):3–37, 2005.

[36] P. Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.

When Is Operation Ordering Required in Replicated Transactional Storage?

Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports
University of Washington
{iyzhang,naveenks,aaasz,arvind,drkp}@cs.washington.edu

Abstract

Today's replicated transactional storage systems typically have a layered architecture, combining protocols for transaction coordination, consistent replication, and concurrency control. These systems generally require costly strongly-consistent replication protocols like Paxos, which assign a total order to all operations. To avoid this cost, we ask whether all replicated operations in these systems need to be strictly ordered. Recent research has yielded replication protocols that can avoid unnecessary ordering, e.g., by exploiting commutative operations, but it is not clear how to apply these to replicated transaction processing systems. We answer this question by analyzing existing transaction processing designs in terms of which replicated operations require ordering and which simply require fault tolerance. We describe how this analysis leads to our recent work on TAPIR, a transaction protocol that efficiently provides strict serializability by using a new replication protocol that provides fault tolerance but not ordering for most operations.

1 Introduction

Distributed storage systems for today's large-scale web applications must meet a daunting set of requirements: they must offer high performance, graceful scalability, and continuous availability despite the inevitablity of failures. Increasingly, too, application programmers prefer systems that support distributed transactions with strong consistency to help them manage application complexity and concurrency in a distributed environment. Several recent systems [11, 3, 8, 5, 6] reflect this trend. One notable example is Google's Spanner system [6], which guarantees strictly-serializable (aka linearizable or externally consistent) transaction ordering [6].

Generally, distributed transactional storage with strong consistency comes at a heavy performance price. These systems typically integrate several expensive mechanisms, including concurrency control schemes like strict two-phase locking, strongly consistent replication protocols like Paxos, and atomic commitment protocols like two-phase commit. The costs associated with these mechanisms often drive application developers to use more efficient, weak consistency protocols that fail to provide strong system guarantees.

In conventional designs, these protocols – concurrency control, replication, and atomic commitment – are implemented in separate layers, with each layer providing a subset of the guarantees required for distributed

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

ACID transactions. For example, a system may partition data into shards, where each shard is replicated using Paxos, and use two-phase commit and two-phase locking to implement serializable transactions across shards. Though now frequently implemented together, these protocols were originally developed to address separate issues in distributed systems. In our recent work [23, 24], we have taken a different approach: we consider the storage system architecture as a whole and identify opportunities for cross-layer optimizations to improve performance.

To improve the performance of replicated transactional systems, we ask the question: *do all replicated operations need to be strictly ordered*? Existing systems treat the replication protocol as an implementation of a persistent, ordered log. Ensuring a total ordering, however, require cross-replica coordination on every operation, increasing system latency, and is typically implemented with a designated leader, introducing a system bottleneck. A recent line of distributed systems research has developed more efficient replication protocols by identifying cases when operations do not need to be ordered – typically by having the developer express which operations commute with others [4, 14, 18, 15]. Yet it is not obvious how to apply these techniques to the layered design of transactional storage systems: the operations being replicated are not transactions themselves, but coordination operations like PREPARE or COMMIT records.

To answer this question, we analyzed the interaction of atomic commitment, consistent replication, and concurrency control protocols in common combinations. We consider both their requirements on layers below and the guarantees they provide for layers above. Our analysis leads to several key insights about the protocols implemented in distributed transactional storage systems:

- The concurrency control mechanism is the *application* for the replication layer, so its requirements impact the replication algorithm.
- Unlike disk-based durable storage, replication can *separate* ordering/consistency and fault-tolerance guarantees with different costs for each.
- Consensus-based replication algorithms like Paxos *are not* the most efficient way to provide ordering.
- Enforcing consistency at every layer *is not necessary* for maintaining global transactional consistency.

We describe how these observations led us to design a new transactional storage system, *TAPIR* [23, 24]. TAPIR provides strict serializable isolation of transactions, but relies on a weakly consistent underlying replication protocol, *inconsistent replication* (IR). IR is designed to efficiently support fault-tolerant but unordered operations, and TAPIR uses an *optimistic timestamp ordering* technique to make most of its replicated operations unordered. This article does not attempt to provide a complete explanation of the TAPIR protocol, its performance, or its correctness. Rather, it analyzes it along with additional protocols to demonstrate the value of separating operation ordering from fault tolerance in replicated transactional systems.

2 Coordination Protocols

This paper is about the protocols used to build scalable, fault-tolerant distributed storage system, like distributed key-value stores or distributed databases. In this section, we specify the category of systems we are considering, and review the standard protocols that they use to coordinate replicated, distributed transactions.

2.1 Transactional System Model

The protocols that we discuss in this paper are designed for scalable, distributed transactional storage. We assume the storage system divides stored data across several *shards*. Within a shard, we assume the system uses replication for availability and fault tolerance. Each shard is replicated across a set of storage servers organized

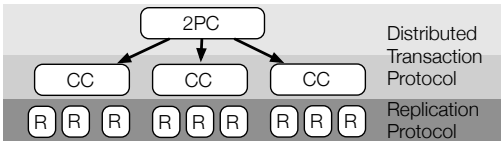


Figure 1: Standard protocol architecture for transactional distributed storage systems.

into a *replica group*. Each replica group consists of $2f + 1$ storage servers, where f is the number of faults that can be tolerated by one replica group. Transactions may read and modify data that spans multiple shards; a distributed transaction protocol ensures that transactions are applied consistently to all shards.

We consider system architectures that layer the distributed transaction protocol atop the replication protocol, as shown in Figure 1. In the context of Agrawal et al.'s recent taxonomy of partitioned replicated databases [2], these are *replicated object systems*. This is the traditional system architecture, used by several influential systems including Spanner [6], MDCC [11], Scatter [9] and Granola [7]. However, other architectures have been proposed that run the distributed transaction protocol within each site, and use replication across sites [17].

Our analysis does not consider the cost of disk writes. This may seem an unconventional choice, as synchronous writes to disk are the traditional way to achieve durability in storage systems – and a major source of their overhead. Rather, we consider architectures where durability is achieved using in-memory replication, combined with asynchronous checkpoints to disk. Several recent systems (e.g., RAMCloud [20], H-Store [21]) have adopted this approach, which offers two advantages over disk. First, it ensures that the data remains continuously available when one system fails, minimizing downtime. Second, recording an operation in memory on multiple machines can have far lower latency than synchronously writing to disk, while still tolerating common failures.

We assume that clients are application servers that access data stored in the system. Clients have access to a directory of storage servers and are able to directly map data to storage servers, using a technique like consistent hashing [10].

Transaction Model We assume a general transaction model. Clients begin a transaction, then execute operations during the transaction's *execution period*. During this period, the client is free to abort the transaction. Once the client finishes execution, it can commit the transaction, atomically and durably committing the executed operations to the storage servers.

2.2 Standard Protocols

The standard architecture for these distributed storage systems, shown in Figure 1 is layering an atomic commitment protocol, like two-phase commit, and a concurrency control mechanism, like strict two-phase locking, atop a consensus-based replication protocol, like Paxos. We briefly review each protocol.

Atomic Commitment Two-Phase Commit (2PC) is the standard atomic commit protocol; it provides all-or-nothing agreement to a single operation across a number of machines, even in the presence of failures. Distributed storage systems use 2PC to atomically commit transactions involving operations at different storage servers.

2PC achieves atomicity by having participants first *prepare* to commit the transaction, then wait until they hear a commit or abort decision from the coordinator to finish the transaction. To maintain correctness in the presence of participant or coordinator failures, 2PC requires three durable writes to an on-disk log: on prepare and commit at the participants and at the commit decision point on the coordinator.

Concurrency Control In order to support concurrent transactions at participants, distributed storage systems typically integrate 2PC with a concurrency control mechanism. Concurrency control mechanisms enable partic-

Table 1: Comparison of read-write transaction protocols in replicated transactional storage systems.

Transaction System	Replication Protocol	Read Latency	Commit Latency	Msg At Bottleneck	Isolation Level	Transaction Model
Spanner [6]	Multi-Paxos [13]	2 (leader)	4	$2n$ + reads	Strict Serializable	Interactive
MDCC [11]	Gen. Paxos [14]	2 (any)	3	$2n$	Read-Committed	Interactive
Repl. Commit [17]	Paxos [13]	$2n$	4	2	Serializable	Interactive
CLOCC [1, 16]	VR [19]	2 (any)	4	$2n$	Serializable	Interactive
Lynx [25]	Chain Repl. [22]	–	$2n$	2	Serializable	Stored procedure
TAPIR [23, 24]	Inconsistent Rep.	2 (to any)	2	2	Strict Serializable	Interactive

ipants to prepare for concurrent transactions as long as they do not violate linearizable ordering. Concurrency control mechanisms can be either pessimistic or optimistic; a standard pessimistic mechanism is strict two-phase locking (S2PL), while optimistic concurrency control [12] (OCC) is a popular optimistic mechanism. S2PL depends on durable log writes to ensure that locks persist in the presence of participant failures. OCC relies on writes to log to keep the list of accepted (prepared or committed) transactions, in order to check optimistically executed transactions for conflicts.

Consistent Replication Distributed systems have widely adopted consensus-based replication protocols like Paxos [13] or Viewstamped Replication [19]. Replication protocols execute operations at a number of replicas to ensure that the effects of successful operations persist even after a fraction of the replicas have failed. To achieve strict single-copy consistency across replicas, replicas coordinate to ensure a single order of operations across all replicas.

Distributed storage systems use these protocols to provide consistent replicated logs for two-phase commit and concurrency control. These replication protocols are often used to replace disk-based logs for durable storage; however, as we will demonstrate, they provide guarantees in a different way from disks: in a distributed system, it is possible to order operations without making them durable, and vice versa.

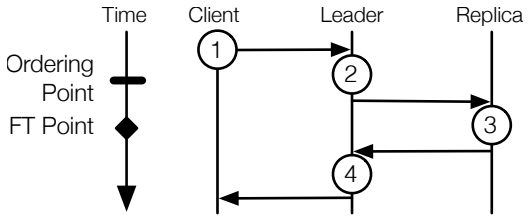
Examples Table 1 gives examples of transactional storage systems and describes the replication protocols they use in order to support particular transaction models. The table compares these systems based on the latency they require to complete a read or commit a transaction, as well as the number of messages processed by a bottleneck replica – typically the determining factor for replication system throughput.

3 Separating Ordering and Fault-Tolerance in Distributed Protocols

Traditional distributed storage systems use disk-based logs to provide two guarantees: ordering and fault tolerance. These two concepts are fundamentally intertwined in a disk-based log: the order in which operations are written to disk, in addition to marking the point at which an operation becomes fault-tolerant, defines a single global order of operations.

Modern distributed storage systems replace disk-based logs with consensus-based replication protocols to provide the same guarantees. The key difference in replicated systems is that ordering and fault tolerance each impose different costs. We argue that, as a result, they *should be separated* whenever possible. Therefore, it is important to distinguish between the following operations:

- **Logged operations** are guaranteed to be both ordered and fault-tolerant, and can be provided with a consensus-based replication protocol.
- **Fault-tolerant (FT) operations** are guaranteed to be fault-tolerant, and can be provided by quorum writes to $f + 1$ replicas.



1. Client sends an operation to the leader.
2. Leader orders the operation in the serial ordering and sends the operation to the replicas in serial order.
3. Replica records or executes the operation.
4. Leader collects responses from at least f replicas.

Figure 2: **Viewstamped Replication (VR) protocol.** The *ordering point* (black line) indicates the point in the algorithm where the ordering of the log operation is determined and the *FT point* (black diamond) is the point where the operation has been replicated and can be considered fault-tolerant. In VR, step 2 at the leader provides ordering, and step 3 at the other replicas provide durability.

- **Ordered operations** are guaranteed to have a single serial ordering. These can be provided in a number of ways, including serialization through a single server or timestamps based on loosely synchronized clocks like Spanner.

Researchers have traditionally analyzed the complexity of distributed protocols using metrics like the number of messages processed by each replica or the number of message delays per operation. We conduct our analysis by studying when logged, ordered and FT operations are required, as this ultimately gives greater insight into how to co-design the layered protocols used in distributed transactional storage systems.

At the same time, the number of logged, ordered and FT operations in each protocol gives insight into the basic complexity (i.e., message delays) and performance. FT operations require a single round-trip to multiple replicas. Ordered operations vary in cost, depending on how the ordering is provided. Logged operations are the most expensive because they provide both ordering and fault tolerance.

Two-phase commit and concurrency control mechanisms were designed for disk-based logs so they use logged operations for fault tolerance. In the following two sections, we analyze whether logged operations in these protocols require ordering or can be replaced with the cheaper FT operations.

3.1 Analysis of Standard Protocols

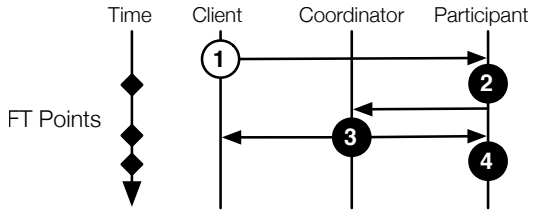
Consistent Replication We first briefly analyze the basic replication protocol used to provide logged operations. The commonly-used consensus protocols, like Multi-Paxos or VR, are leader-based. In the common, non-failure case, shown in Figure 2, these protocols provide consensus in two steps: the first step provides ordering, while the second ensures fault tolerance. Essentially, these protocols provided logged operations by combining an ordered operation followed by an FT operation. The ordered operation is expensive; it is provided by serialization through the leader, incurring a round-trip and leading to a bottleneck at the leader.

Previous work has noted the cost of providing ordering in consensus-based replication protocols. Our following analysis is orthogonal and complementary to recent work [18, 14, 15] on providing replication protocols where *some* of the operations do not require ordering.

3.1.1 Distributed Transactions

2PC uses logged operations to the replication layer in three places, shown as black numbered circles in Figure 3. The operations need to be fault-tolerant but not necessarily ordered.

To ensure that prepared transactions are not lost due to failures at the participants, the prepare operation must be an FT operation. The commit/abort operation must be an FT operation as well; it requires durability to move the participant out of the prepared state and to persist the effects of the transaction. At the coordinator, the commit/abort decision must be an FT operation to ensures that, if there is a failure at the coordinator, the



1. Client sends prepare to participants.
2. Participant prepares to commit transaction and responds to coordinator.
3. Once all participants respond, coordinator makes the commit decision and sends decision to participants and client.
4. Participant commits/aborts the transaction.

Figure 3: **Two-phase Commit (2PC) protocol**. Participants are servers involved in the transaction. Logged operations are marked as black circles with a white operation number. The blocking period represents the period when participants block on a decision from the coordinator.

participants can still find the outcome of the transactions[1]. Otherwise, participants could be blocked in the prepared state forever.

Whether the 2PC operations require ordering, besides fault tolerance, depends on whether the underlying concurrency control mechanism relies on ordered prepare and commit operations to maintain transaction ordering. In the next subsection we further analyze the ordering requirements for 2PC operations when integrating different concurrency control mechanisms.

Observation: 2PC requires at least 3 FT operations, but ordering depends on the underlying concurrency control mechanism.

3.1.2 Distributed Concurrency Control

While 2PC relies on replication for durability, the concurrency control mechanism is the application being replicated. Thus, its requirements dictate the ordering guarantees needed from the replication layer.

The number of logged operations remains the same in the integrated 2PC/concurrency control protocol. Concurrency control mechanisms do not require additional logged operations during the execution period because the transaction can always abort on failure. However, during 2PC, the concurrency control state must be logged on prepare and commit.

Different concurrency control mechanisms have different requirements for the replication layer. For S2PL, shown in Figure 4, acquiring locks during the execution phase is not a logged operation, however, *it is an ordered operation*. This ordered operation may be handled by the replication layer; a common way to implement distributed S2PL is to keep the locks at the leader of the replica group and send all operations to the leader during the execution period. The prepare operation for S2PL/2PC does not depend on ordering, so it only needs an FT operation for durability. The commit operation releases locks, so it must be an ordered and FT operation.

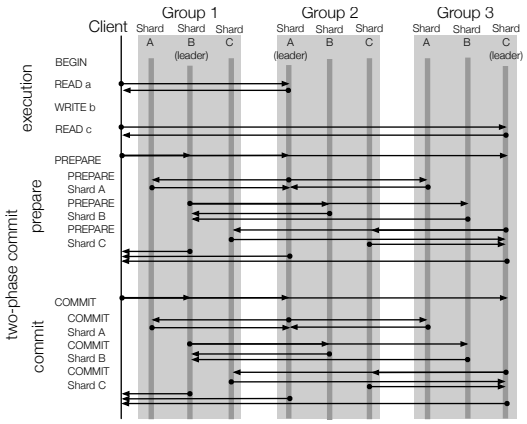
OCC/2PC, shown in Figure 5, has a different set of requirements. During execution, there are no ordered operations. Thus, distributed OCC can send reads to any replica and support buffered writes at the client[2]. Instead, OCC/2PC's prepare operation must be an ordered and FT operation. OCC requires ordering on prepare because it checks for conflicts against previously accepted (prepared or committed) transactions. The commit operations do not need ordering, but aborts do because they affect conflict checks.

S2PL and OCC maintain consistency in fundamentally different ways, therefore, they have different ordering requirements. At the participants, S2PL requires one ordered operation per lock, one FT operation and one logged operation, while OCC requires two logged operations. The coordinator still may require either an FT or a logged operation, depending on how it makes the commit/abort decision.

Observation: Optimistic concurrency control mechanisms limit the number of ordered operations.

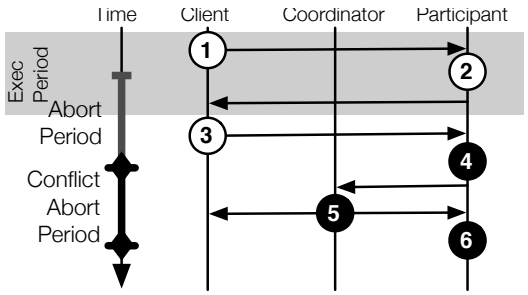
[1] If the coordinator cannot vote to abort after a successful prepare, then the coordinator does not require a logged operation

[2] A common optimization, used by Spanner [6], is to buffer writes for S2PL as well and acquire write locks on prepare. Then, the prepare for S2PL/2PC would also require ordering.



1. Client sends reads/writes to participant.
2. Participant acquires lock (and returns the value).
3. Client sends prepare to all participants.
4. Participant records locks.
5. Once all participants respond, coordinator makes the commit decision and sends decision to participants and client.
6. Participant commits the transaction and releases locks.

Figure 4: **2PC with Strict Two-Phase Locking (S2PL)**. The execution period represents when client runs the transaction, sending reads and writes to the server. With S2PL, locks block other transactions until they are released, so acquiring the lock is an ordered operation. That makes the prepare an FT operation, while the commit/abort decision at the coordinator and the commit/abort at the participants is a logged operation.



1. Client sends reads to participant.
2. Participant returns the current version.
3. Client sends prepare to all participants.
4. Participant runs OCC validation checks.
5. Once all participants respond, coordinator makes commit decision and sends decision to participants and client.
6. Participant commits the transaction and removes transaction from prepared list.

Figure 5: **2PC with Optimistic Concurrency Control (OCC)**. Instead of blocking other transactions, OCC executes optimistically, then checks for conflicts on prepare. This makes the prepare a logged operation, along with the coordinator commit/abort decision and the commit/abort at the participants.

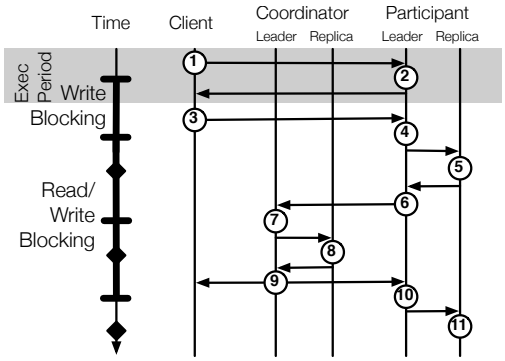
3.1.3 The Integrated Transaction Protocol

In this section, we combine 2PC and concurrency control with consistent replication into a full, integrated protocol. This integration allows us to see the consequences of requirements in the 2PC/concurrency control protocol on the replication protocol.

For simplicity, we only analyze the pessimistic, S2PL/2PC/VR protocol shown in Figure 6. We add a common optimization that Spanner uses to the protocol from the last section. We buffer writes at the client and only acquire write locks on prepare, making the prepare operation an ordered (and FT) operation. Altogether, we believe this protocol represents the typical way a distributed storage system today might provide replicated, distributed transactions.

There are a large number of ordered operations in this protocol. Reads during execution, prepares, and commits at the coordinator and the participants, all require ordered operations. These represent extra network delays and throughput bottlenecks. The protocol also has 3 FT operations.

Observation: The standard integration of S2PL/2PC/VR requires a large number of ordered (and FT) operations to provide transactional consistency.



1. Client sends reads to participant leader.
2. Participant leader acquires lock and returns value.
3. Client sends prepare to the each participant leaders.
4. Participant leader acquires write locks.
5. Participant replica acquires read and write locks.
6. Participant leader responds to coordinator.
7. Once participant leaders respond, coordinator makes commit decision.
8. Coordinator replica commits.
9. Coordinator sends commit to participant leaders and client.
10. Participant leader commits and releases locks.
11. Participant replica commits and releases locks.

Figure 6: **S2PL/VR/2PC**. We add the write-buffering optimization that Spanner uses, where write locks are acquired on prepare. Together, we believe this integrated protocol represents the typical way to provide distributed replicated transactions.

3.2 Analysis of Spanner

The recent Spanner protocol, shown in Figure 7, is Google's solution to replicated, distributed transactions. Its key contribution is TrueTime, which uses atomic clocks to provide loosely synchronized clocks with error bounds. Spanner uses TrueTime to provide ordering for read-only transactions without acquiring locks. Since read-only transactions require only ordering, and not durability, using TrueTime for ordering eliminates the need to serialize read-only transactions through a single server.

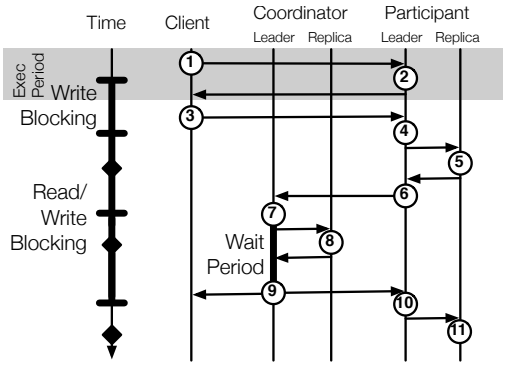
Spanner avoids serializing read-only transactions through a single server because it incurs an extra round-trip and the server can become a bottleneck. Systems that rely only on the replication layer to provide ordering, like MegaStore [3], *always incur these overheads,*. Spanner demonstrates that there are other ways to provide ordering in a distributed system that can be more effective in some cases. Of course, there is a trade-off. TrueTime requires waits and only provides consistency as long as clock skews are within error bounds.

To support read-only transactions using TrueTime, Spanner must serialize each read/write transaction at a single timestamp. While Spanner makes effective use of TrueTime for read-only transactions, it uses the standard S2PL/2PC/VR protocol for read-write transactions, with a wait on commit to accommodate clock skew. Spanner still uses a leader-based Paxos protocol and serialization through the leader for locking. Thus, for each read-write transaction, Spanner uses the same amount of distributed coordination as standard S2PL/2PC/VR to *make a single ordering decision.*

Observation: Spanner requires a large number of ordered (and FT) operations to order each transaction at a single timestamp.

3.2.1 Optimistic Spanner

As an example of how we can move and eliminate ordered operations in Spanner, we propose an alternative Spanner protocol, shown in Figure 8. This protocol has two changes from the basic Spanner protocol. First, we switch Spanner to OCC, which uses fewer ordered operations, but may have to abort. Next, we move the coordinator from one of the participants to the client, which we assume to be an application server. With these changes, we can eliminate ordered operations during execution, to allow reads from any replica for read-write transactions, and eliminate a ordered and replicated operation at the coordinator.



1. Client sends reads to participant leader.
2. Participant leader acquires lock and returns value.
3. Client sends prepare to participant leaders.
4. Participant leaders acquires write locks and select prepare timestamp.
5. Participant replicas acquire locks and record prepare timestamp.
6. Participant leaders respond with prepare timestamp.
7. Once participant leaders respond, coordinator selects a commit timestamp by taking the max of the prepare timestamps and its own local time.
8. Coordinator replicas commit at commit timestamp.
9. After waiting for the uncertainty bound, coordinator sends commit with commit timestamp to participant leaders and client.
10. Participant leaders commit at commit timestamp, and release locks.
11. Participant replicas commit at commit timestamp, and release locks.

Figure 7: **Spanner Commit Protocol.** This protocol is very similar to the S2PL/VR/2PC algorithm. The key difference is the use of timestamps from roughly synchronized clocks (TrueTime) and wait period (of double the uncertainty bound), which enables linearizable read-only transactions without locking or 2PC.

Spanner's use of TrueTime lends itself well to OCC. Like Spanner and unlike locking, OCC orders each transaction at a single timestamp, supplied by the coordinator on commit. In a distributed system, selecting a globally-relevant commit timestamp can be tricky; however, Spanner's commit timestamps work perfectly as OCC timestamps.

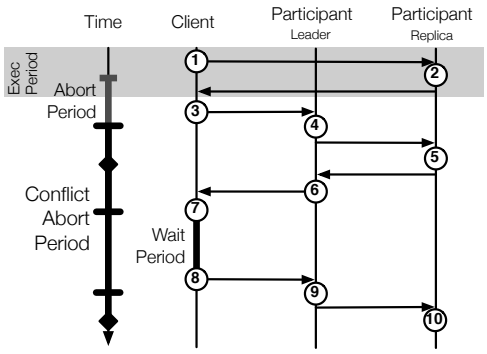
Observation: Ordered operations can be moved and eliminated in some cases while still maintaining transactional consistency.

4 Inconsistent Replication and TAPIR

Our analysis above shows that existing protocols require multiple ordered and fault tolerant operations to commit transactions in a replicated, partitioned database system. We argue that the inability to separate fault-tolerant and ordered operations is a major cause of wasted work in these cases, because extra protocol steps and increased blocking result in worse latency and throughput. Motivated by this observation, we recently designed a new transaction system based on the idea of co-designing the transaction coordination and replication protocols to support efficient, unordered operations [23, 24].

As part of this work, we have designed a new replication protocol, *inconsistent replication (IR)* that treats fault-tolerant and ordered operations separately. IR is not intended to be used by itself; rather, it is designed to be used with a higher-level protocol, like a distributed transaction protocol. IR provides fault-tolerance without enforcing any consistency guarantees of its own. Instead, it allows the higher-level protocol, which we refer to as the *application protocol*, to decide the outcome of conflicting operations. It does so using two classes of operations:

- **inconsistent** operations are fault-tolerant but not ordered: successful operations persist across failures, but operations can execute in any order.
- **consensus** operations are allowed to execute in any order, but return a single *consensus result*. Successful operations and their consensus results persist across failures.



1. Client sends reads to participant replica.
2. Participant replica returns latest version.
3. Client sends prepare to participant leaders.
4. Participant leaders run OCC validation checks and select prepare timestamps.
5. Participant replicas record prepared transaction and timestamps.
6. Participant leaders respond to client with prepare timestamps.
7. Once participants respond, client selects a commit timestamp.
8. After waiting for the uncertainty bound, client sends commit with commit timestamp to participant leaders.
9. Participant leaders commit at commit timestamp, and remove transaction from list of prepared transactions.
10. Participant replicas commit at commit timestamp.

Figure 8: **Optimistic Spanner.** We switch the Spanner protocol to OCC instead of S2PL and eliminate the coordinator. The switch to OCC enables reads to be served by any replica. If the replica is not up-to-date, the transaction will abort during the prepare phase. Eliminating the coordinator lets the client pick (and know) the commit timestamp earlier, but leaves the client with nothing to do while waiting out the uncertainty.

Both types of operations are efficient: **inconsistent** operations complete in one round trip without coordination between replicas, as do **consensus** operations when replicas agree on their results (the common case). If replicas disagree on the result of a **consensus** operation, the application protocol on the client is responsible for *deciding* the outcome of the operation in an application-specific way.

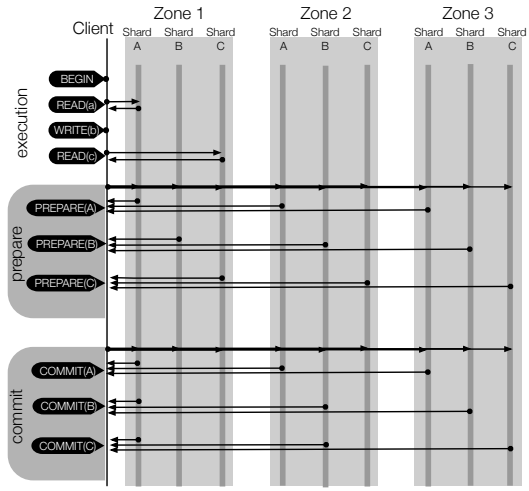
TAPIR is designed to be layered atop IR in a replicated, transactional storage system. Figure 9 demonstrates how coordination in TAPIR works using the same transaction as Figure 6. TAPIR obtains greater performance because IR does not require any leaders or centralized coordination.

As we noted in our discussion of Spanner above, using optimistic concurrency control instead of locking makes it possible to reduce the number of ordered operations. TAPIR uses this approach, and takes it further using *optimistic timestamp ordering*. The client selects a timestamp using its local clock, and proposes that as the transaction's timestamp in its prepare operation. Participant replicas accept the transaction's prepare only if both of two conditions hold: they have not processed any transactions with a higher timestamp, and the transaction passes an OCC validation check with all previously prepared transactions. This reduces the number of ordered operations to one.

Realizing this technique requires careful protocol design. In particular, TAPIR must be able to handle inconsistent results from its replication layer, select timestamps in a way that ensures progress, and tolerate failures of client-coordinators. We do not attempt to describe these protocols here in detail; the interested reader is referred to our recent SOSP paper [23] and its accompanying technical report [24].

5 Conclusion

Many partitioned replicated databases treat the replicated storage much as they might use a traditional disk: as a persistent, ordered log. Unlike a disk, however, replicated distributed systems are able to achieve fault tolerance without ordering operations (and vice versa). We have presented a framework for analyzing transaction coordination protocols in terms of the number of fault tolerant and ordered operations, and argue that separating them in this way provides insight into the fundamental costs of different approaches like two-phase locking and optimistic concurrency control in distributed systems. As a concrete example, we discuss our recent work on



1. Client sends reads to participant replica.
2. Participant replica returns latest version.
3. Client selects a proposed timestamp and sends prepare to all participants as **consensus** operation.
4. Participant runs OCC validation against operations it has previously seen, using the client's proposed timestamp.
5. If replicas return conflicting results, TAPIR uses IR's slow path to resolve the conflict; a prepare is successful only if a majority of replicas voted prepare.
6. Client sends commit (or abort) operation to all participants as an IR **inconsistent** operation.
7. Participant commits or aborts the transaction at commit timestamp.

Figure 9: *Example read-write transaction in TAPIR.* TAPIR executes the same transaction pictured in Figure 6 with less redundant coordination. `Read`s go to the closest replica and `Prepare` takes a single round-trip to all replicas in all shards.

the TAPIR protocol, which uses optimistic concurrency control and optimistic timestamp ordering to eliminate most of its ordered operations, allowing it to use a replication protocol (IR) that provides fault tolerance without ordering.

Acknowledgements

This work was supported by the National Science Foundation under grants CNS-0963754, CNS-1217597, CNS-1318396, CNS-1420703, and CNS-1518702, by NSF, IBM and MSR Ph.D. fellowships, and by Google. We also appreciate the support of our local zoo tapirs, Ulan and Bintang.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *Proc. of SIGMOD*, 1995.

[2] D. Agrawal, A. E. Abbadi, and K. Salem. A taxonomy of partitioned replicated cloud-based database systems. *IEEE Data Engineering Bulletin*, 38(1):4–9, Mar. 2015.

[3] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, 2011.

[4] L. Camargos, R. Schmidt, and F. Pedone. Multicoordinated Paxos. Technical report, University of Lugano Faculty of Informatics, 2007/02, Jan. 2007.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. of VLDB*, 2008.

[6] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, 2012.

[7] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proc. of USENIX ATC*, 2012.

[8] R. Escriva, B. Wong, and E. G. Sirer. Warp: Multi-key transactions for key-value stores. Technical report, Cornell, Nov 2013.

[9] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *Proc. of SOSP*, 2011.

[10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of STOC*, 1997.

[11] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proc. of EuroSys*, 2013.

[12] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 1981.

[13] L. Lamport. Paxos made simple. *ACM SIGACT News*, 2001.

[14] L. Lamport. Generalized consensus and Paxos. Technical Report 2005-33, Microsoft Research, 2005.

[15] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. of OSDI*, 2012.

[16] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proc. of ECOOP*, 1999.

[17] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. E. Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. of VLDB*, 2013.

[18] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[19] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[20] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. of SOSP*, 2011.

[21] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. of VLDB*, 2007.

[22] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of OSDI*, 2004.

[23] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. of SOSP*, Oct. 2015.

[24] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication (extended version). Technical Report 2014-12-01 v2, University of Washington CSE, Sept. 2015. Available at http://syslab.cs.washington.edu/papers/tapir-tr-v2.pdf.

[25] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proc. of SOSP*, 2013.

Eventually Returning to Strong Consistency

Marko Vukolić
IBM Research - Zurich
mvu@zurich.ibm.com

Abstract

Eventually and weakly consistent distributed systems have emerged in the past decade as an answer to scalability and availability issues associated with strong consistency semantics, such as linearizability.

However, systems offering strong consistency semantics have an advantage over systems based on weaker consistency models, as they are typically much simpler to reason about and are more intuitive to developers, exhibiting more predictable behavior. Therefore, a lot of research and development effort is being invested lately into the re-engineering of strongly consistent distributed systems, as well as into boosting their scalability and performance.

This paper overviews and discusses several novel directions in the design and implementation of strongly consistent systems in industries and research domains such as cloud computing, data center networking and blockchain. It also discusses a general trend of returning to strong consistency in distributed systems, when system requirements permit so.

1 Introduction

Strong consistency criteria, and, in particular, *linearizability* [16], have for years been the gold standard in distributed and concurrent data management. Linearizability has been favored by developers and users alike, as it brings a powerful abstraction that dramatically reduces the complexity of reasoning about data consistency in a distributed system. Specifically, linearizability requires every read/write[1] operation to appear to take place instantaneously at some point between operation's invocation and response. As a result, consistency-wise, linearizability reduces a distributed system to a centralized one — which developers and users have been traditionally accustomed to.

However, although very intuitive to understand, the strong semantics of linearizability make it challenging to implement. This is captured by the *CAP* theorem [6], an assertion that binds strong consistency (linearizability) to the ability of a system to maintain a non-trivial level of availability when confronted with network partitions. In a nutshell, the *CAP* theorem, formally proven in [15], states that in the presence of network partitions, a distributed storage system has to sacrifice either availability or (strong) consistency.

In response, many eventually and weakly consistent distributed systems have recently emerged as an answer to the scalability and availability issues associated with strong consistency semantics. In particular, eventually

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] In this paper, we denote operations that modify the state of a distributed data management system as *write* operations and those that do not as *read* operations.

consistent systems were pioneered already in the 1990s [34], but became popular in the past decade with the advent of cloud computing [36], where scalability and availability requirements are often put before consistency requirements. Roughly speaking, eventual consistency requires replicas in a distributed data management system to eventually converge to identical copies in the absence of further writes.

Over the years, many different consistency notions between strong and weak consistency have been proposed (eventual consistency being one of them). In non-transactional systems alone (i.e., systems in which operations are performed on one data object at a time), more than 50 different consistency flavors have been proposed [35]. Furthermore, dozens of additional consistency notions were proposed in the context of transactional database systems (see e.g., [1]). This multitude of consistency notions as well as the complexity and subtleties separating different nuances of consistency have contributed to the fact that strong consistency remains the preferred correctness condition of a distributed system for vast majority of practitioners and users [33, 9].

Therefore, it is not surprising that a lot of recent and ongoing research has focused on exploring how to make strongly consistent systems scale as much as possible and how to boost their performance. In this paper, we briefly overview a subset of recent research efforts, specifically those focused around the following axes:

- *Consistency hardening (in software).* First, in Section 2, we discuss recent efforts that turn eventually consistent cloud storage systems into strongly consistent ones in a scalable way, effectively *hardening* their consistency notion.
- *Strong consistency in hardware.* Then, in Section 3, we discuss recent efforts that aim at boosting the performance of strongly consistent system by implementing them using modern hardware technologies such as FPGAs, RDMA, Infiniband, etc.
- *Scaling strong consistency for blockchains.* Finally, in Section 4, we discuss ongoing trends in blockchains and cryptoledgers (such as Bitcoin [23]), related to moving from eventually consistent consensus protocols (e.g., a proof-of-work consensus protocol of Bitcoin) to strongly consistent ones. We also highlight the main scalability challenges in this context.

2 Consistency hardening (in software)

As already discussed, eventual consistency is used very often in cloud storage systems. In particular, eventual consistency established itself as a go-to consistency model for very-large-scale object storage systems that provide general-purpose storage to web applications at low cost, typically through a REST interface. For instance, eventual consistency is offered by commercial services based on popular open-source technologies such as Openstack Swift (e.g., IBM Softlayer Object Storage), as well as on proprietary solutions, such as Amazon S3.

Eventual consistency of such storage services increases the complexity of value-added services built on top of them. For instance, multi-cloud storage solutions built on top of cloud object stores only (e.g., [3, 4]) can at best offer *consistency proportionality*, in the sense that the consistency of value-added depends on the consistency of the underlying clouds [4].

To rectify this, recent systems, such as Hybris [13] and SCFS [5], propose to strengthen the consistency of eventually consistent cloud storage systems by relying on a small portion of metadata that is kept strongly consistent. In the following, we briefly outline this technique, called *consistency hardening* [13] (or, alternatively, *consistency anchoring* [5]).

To achieve consistency hardening, systems such as Hybris build on established architectural decision to separate data and metadata (control) planes in distributed storage systems (see HDFS [31] as an example). Then, in addition to typical storage system metadata, such as version control numbers, Hybris adds a cryptographic hash of an object stored in an eventually consistent cloud store. In a sense, Hybris maintains hashes of objects

in a strongly consistent way, while keeping the bulk data separately in eventually consistent clouds. Then on reading data objects from eventually consistent cloud stores, Hybris compares this data to the hash stored in a strongly consistent metadata store, detecting potential inconsistencies and allowing re-tries, effectively masking the temporary inconsistencies. Whereas Hybris keeps hashes in a strongly consistent, Zookeeper [17] cluster on a private cloud, hashes and metadata can be kept in any strongly consistent smaller-scale data store. A Hybris performance evaluation [13] attests that consistency hardening achieves very good performance and scales easily to tens of thousands of operations.

Note that Hybris and SCFS are data-agnostic, in the sense that they rely on system-architectural decisions rather than exploiting semantic aspects of stored data to harden consistency. This is different from other approaches to "putting more strength" into eventual consistency that actually exploit data semantics to turn weaker consistency notions into stronger ones. In particular, Shapiro et al. [30] propose *strong eventual* consistency exploiting conflict-free replicated data types (CRDTs) (e.g., those data types in which operations commute) to boost the consistency guarantees of eventual consistency. In short, an eventually consistent system satisfies strong eventual consistency if correct data replicas that have delivered the same updates also have equivalent state [30].

3 Strong consistency in hardware

As data centers grow in size and volume, with services often running on hundreds to thousands of machines, they increasingly depend on a strongly consistent coordination (or metadata) service. Earlier, we have already mentioned that modern, strongly consistent coordination services (e.g., Zookeeper) easily achieve a throughput of tens of thousands operations per second when run on commodity hardware, combined with reasonable latencies on the order of milliseconds. However, these performance numbers are not sufficient for the high demand of data-center applications which often leads to relaxing consistency, which in turn requires building more complex logic in data-center applications and services to deal with inconsistencies.

To rectify this, a lot of research effort has recently been devoted to speeding up strongly consistent services using modern hardware readily available in data centers. This hardware includes, but is not limited to, field-programmable gate arrays (FPGAs), Remote Direct Memory Access (RDMA), Infiniband and 40/100 Gbps Ethernet (40/100 GbE) networking, etc. In particular, the focus of this research has been on implementing consensus, total order (atomic) broadcast [7] and state-machine replication [29], i.e., conceptually equivalent abstractions on top of which any strongly consistent service can be built. In the following, we briefly describe some of the prominent systems that delegate strong consistency to hardware.

Recently, Istvan et al. [18] demonstrated a Zookeeper-like system in which Zookeeper atomic broadcast [19] is entirely implemented in FPGAs. This implementation of Zookeeper atomic broadcast comes in two network flavors: TCP and a custom-built messaging protocol for boosting performance even further. On 40GbE networking, atomic broadcast of [18] achieves peak throughputs of nearly 4 million operations per second for the custom-built protocol and around 2.5 million operations per second for the TCP variant. The system further exhibits very low latencies on the order of few microseconds, without significant tail latencies. These performance numbers are very promising for enabling strong consistency at data-center scale. For example, it is not difficult to see how systems such as Hybris (see Section 2) would profit from more than two orders of magnitude better performance of their strongly consistent component when implemented in modern hardware instead in software.

DARE [26] is another recent example of a strongly consistent system exploiting modern hardware. It implements a state-machine replication protocol similar to Raft [25]. DARE is optimized for one-sided RDMA, and achieves consensus latency of less than 15 μs with 0.5-0.75 million operations per second running over an Infiniband network. Similarly, FaRM [14] is designed for RDMA over 40GbE and Infiniband. FaRM is a distributed main-memory key value store with consensus-based replication that achieves very high throughput

(up to 10 million requests per second per node for a mixed read/write workload).

Finally, besides these implementations that exploit modern hardware, another interesting and emerging research direction is using software-defined networking (SDNs). Examples include NetPaxos [12], Speculative Paxos [27] and an implementation of Paxos [20] in switches [11]. Although these SDN-based systems typically achieve one to two orders of magnitude worse performance than the hardware implementations discussed above, they get some of the benefits of implementing strongly consistent services closer to hardware, while maintaining a higher level of programming abstraction.

4 Scaling strong consistency for blockchains

Many different *blockchains* or *distributed ledgers*, led by Bitcoin [23], have been emerging in recent years. Although initially reserved for cryptocurrencies (such as Bitcoin itself), blockchain technology is maturing very fast and is embracing all types of asset transactions, ranging from simple currency transactions á la Bitcoin, to complex transactions containing "smart contracts" i.e., custom code executed in a context of a modern blockchain such as Ethereum [38]. Regardless of the type of the transaction, a blockchain should enforce strong consistency, or total order among transactions (at least for transactions that depend on each other and may conflict) to prevent issues such as asset double-spending.

Even though Bitcoin and similar alternative cryptocurrencies (i.e., *altcoints*) boast distributed consensus [23], this is not the classical consensus that underlies total order broadcast and state-machine replication, but rather a sort of an *eventual consensus*. Indeed, when participants in the Bitcoin network try to solve the difficult cryptographic puzzle (i.e., *mine* a block using proof-of-work (PoW)) [23] in an attempt to agree on the next block of transactions, more than one participant may actually mine the next candidate block. Conflicts between candidate blocks are resolved later on by conflict-resolution rules, such as the longest (most difficult) branch rule of Bitcoin, or alternative rules such as the GHOST rule [32].

However, regardless of conflict resolution, classical PoW consensus remains only eventual. This might come as a surprise, having in mind the requirement that blockchain should enforce total order to prevent asset double-spending, as discussed above. The fact that PoW consensus is only eventual is often informally referred to as absence of *consensus finality* in PoW blockchains; in short, *consensus finality* requires that a valid block can never be removed from the blockchain once appended to it [37].

To cope with the absence of consensus finality of PoW eventual consensus and to eliminate the enormous computational overhead of PoW-based consensus [24], blockchain and distributed ledger communities are increasingly turning back to classical, strongly-consistent distributed consensus to power the blockchain [10, 37]. In the case of the trust model of blockchain, this implies the use of Byzantine fault-tolerant (BFT) consensus, in which consensus participants can exhibit arbitrary or Byzantine [21] behavior. As classical BFT consensus is strongly consistent, it also guarantees consensus finality to blockchains based on it. This trend in blockchain research and development of moving from eventually-consistent PoW consensus to classical, strongly-consistent BFT consensus is exemplified by practical systems such as the consensus protocol underlying the Ripple network[2], or Openblockchain[3], an open-source proposal from IBM for the Linux Foundation Hyperledger project[4].

This shift to strong consistency in blockchains comes with a set of challenges. In particular, one of the key challenges for BFT consensus protocols is scalability in terms of the number of nodes (N). Specifically, whereas PoW eventual consensus scales easily with additional nodes, this is less obvious for BFT consensus protocols which often involve $O(N^2)$ message complexity [8], although deterministic protocols with $O(N)$ amortized message complexity exist [28, 2] and randomized protocols with $O(N)$ worst-case message complexity have been

[2] `https://ripple.com.`

[3] `https://github.com/openblockchain.`

[4] `https://www.hyperledger.org`

proposed [22]. For more detailed information on the scalability challenges in BFT-based blockchains, and for a general comparison between PoW (eventual) and BFT (strong) consensus, we refer the reader to [37].

5 Conclusion

The trend of moving from weak, e.g., eventual, consistency to strong consistency is affecting many industries and research communities, such as cloud computing, data center networking and blockchain. In this paper we gave a brief overview of the reasons underlying these trends and of techniques and systems that aim at making strong consistency practical in these important domains.

References

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Cambridge, MA, USA, March 1999. Also available as Technical Report MIT/LCS/TR-786.

[2] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12:1–12:45, January 2015.

[3] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolić, and Ido Zachevsky. Robust data sharing with key-value stores. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2012*, pages 1–12, 2012.

[4] Alysson Neves Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[5] Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin, and Paulo Veríssimo. SCFS: A shared cloud-backed file system. In *USENIX Annual Technical Conference (ATC), 2014*, pages 169–180, 2014.

[6] Eric A. Brewer. Towards robust distributed systems (abstract). In *ACM Symposium on Principles of Distributed Computing (PODC)*, page 7, 2000.

[7] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[8] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[10] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gun Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains (a position paper). In *3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[11] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 2016.

[12] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.

[13] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: Robust hybrid cloud storage. In *ACM Symposium on Cloud Computing (SOCC)*, pages 12:1–12:14, 2014.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[15] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[16] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[17] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[18] Zsolt Istvan, David Sidler, Gustavo Alonso, and Marko Vukolić. Consensus in a box: Inexpensive coordination in hardware. In *USENIX symposium on Networked systems design and implementation (NSDI)*, 2016.

[19] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[20] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

[21] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[22] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Cryptology ePrint Archive 2016/199*, 2016.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[24] Karl J. O'Dwyer and David Malone. Bitcoin mining and its energy footprint. In *IET Irish Signals & Systems Conference*, 2014.

[25] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[26] Marius Poke and Torsten Hoefler. DARE: high-performance state machine replication on RDMA networks. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2015.

[27] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX symposium on Networked systems design and implementation (NSDI)*, 2015.

[28] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 88–102, 2005.

[29] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[30] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400, 2011.

[31] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE Symposium on Mass Storage Systems and Technologies, (MSST)*, pages 1–10, 2010.

[32] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in Bitcoin. In *Financial Cryptography and Data Security (FC)*, pages 507–527, 2015.

[33] Michael Stonebraker. Stonebraker on NoSQL and enterprises. *Communications of the ACM*, 54(8):10–11, 2011.

[34] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–183, 1995.

[35] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (to appear). Also available as arXiv pre-print http://arxiv.org/abs/1512.00168.*

[36] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, October 2008.

[37] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *IFIP WG 11.4 Workshop on Open Research Problems in Network Security (iNetSec)*, 2015.

[38] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2015.

Abstract Specifications for Weakly Consistent Data

Sebastian Burckhardt
Microsoft Research
Redmond, WA, USA

Jonathan Protzenko
Microsoft Research
Redmond, WA, USA

Abstract

Weak consistency can improve availability and performance when replicating data across slow or intermittent connections, but is difficult to describe abstractly and precisely. We shine a spotlight on recent results in this area, in particular the use of abstract executions *to specify the behavior of replicated objects or stores. Abstract executions are directed acyclic graphs of operations, ordered by visibility and arbitration. This specification approach has enabled several interesting theoretical results, including systematic investigations of the correctness and optimality of replicated data types, and has shed more light on the nature of the fundamental trade-off between availability and consistency.*

1 Introduction

Many organizations operate services that are used by a large number of clients connecting from a range of devices, possibly on multiple continents. Unfortunately, performance and availability of such services can suffer if the network connections (either between servers, or between the servers and the clients) are slow or intermittent. A natural solution is to replicate data. For example, clients may cache a replica on the device so it remains accessible even if communication with the server is temporarily unavailable. Or, data may be geo-replicated across data-centers to maintain low access times and preserve availability under network partitions when serving clients in multiple continents.

Unfortunately, this plan introduces some amount of inconsistency: if a device is operating offline, the device-local replica can temporarily diverge from the server replica; and if we want to avoid waiting for slow intercontinental communication whenever we read or write, then the replicas in multiple continents are not completely synchronized at all times.

The crux is that in any distributed system with slow or intermittent communication, the replication of data that is both read and written is inherently a double-edged sword. The positive effects (improved availability and latency) cannot effectively be separated from the negative effects (expose the application to inconsistent states). Where availability and latency are crucial, we must thus face the challenge of weak consistency. This well-known fact has been popularized as a fundamental problem by the CAP theorem [3, 11]. In the context of cloud storage, Amazon's Dynamo system [13] has garnered much attention and encouraged other storage solutions supporting weak consistency [2, 19] that are in common use today.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 The Specification Problem

Despite their popularity, weakly consistent storage systems are difficult to use. Particularly frustrating is the lack of useful specifications. Typical specifications are either too weak, or too detailed, and usually too vague, to serve their intended purpose: giving the programmer a useful, simple way to visualize the behavior of the system. Consider the notion of *eventual consistency*, which was pioneered by the Bayou system [37] in the context of replicas on mobile devices. It specifies that

> if clients stop issuing update requests, then the replicas will eventually reach a consistent state.

This specification is not actually strong enough to be useful. For one, it is not enough to reach an arbitrary consistent state: this state also must make sense in the context of what operations were performed (for example, one would expect a key-value store to only contain values that were actually written). Similarly, it matters what values are observed not just after, but also before convergence is achieved. Finally, a useful system must guarantee sensible behavior even if updates never stop, which is typical for online services.

In the absence of a clear, abstract specification, application programmers typically base their design on a vague description of some replication algorithm, including numerous low-level details about message protocols, quorum sizes, and background processes. This lack of abstraction makes it difficult to ensure that the application program functions correctly, and leads to non-portable, hard-to-read application code that heavily depends on the particulars of the chosen system.

1.2 Overview

The goal of this article is to put a spotlight on recent, theoretically-oriented work that addresses the "specification problem" for weakly consistent data. We explain the key concept of *abstract executions* that enable us to separate the "what" from the "how". Abstract executions generalize operation sequences (which are suitable for specifying the semantics of single-copy, strongly consistent data) to graphs of operations ordered by visibility and arbitration relations (which are suitable for specifying the semantics of replicated, highly available data). We then mention a few results that demonstrate how the use of abstract executions for devising specifications has opened the door to a systematic study of implementations of replicated data types, including correctness and optimality proofs.

2 Abstract Executions

A specification of a data object or a data store must describe what data is being stored, and how it is being read and updated. We do not distinguish between objects and stores here, but treat them both simply as an abstract data type. First, we introduce two sets Val and Op as follows.

Val contains all data *values* we may ever want to use. For example, Val may include both simple data types such as strings, integers, or booleans, and structured data such as arrays, records, relational databases, or XML documents.

Op contains all *operations* of all data types we may ever want to use. Op may range from simple read and write operations to operations that read and/or write structured data, and may include application-defined operations such as stored procedures.

Operations can include input parameters, and all operations return a value (sometimes a simple ok to acknowledge completion). For an operation $o \in \mathsf{Op}$ and value $v \in \mathsf{Val}$ we write $o : v$ to represent the *invocation* of the operation together with the returned value. For example, when accessing a key-value store, we may observe the following sequence of invocations:

$$wr(A,1):\mathsf{ok}, \quad wr(B,1):\mathsf{ok}, \quad rd(A):1, \quad wr(B,2):\mathsf{ok}, \quad rd(B):2$$

2.1 Sequential Data Types

Invocation sequences capture the possible sequential interactions with the object. We can use such sequences to define the observable behavior of the object as a black-box.

Definition 1: A sequential data type specification is a set of invocation sequences.

Example 1: Define the **register** data type to contain the sequences where (a) each invocation is of the form $wr(v):\texttt{ok}$ or $rd:v$ (where v ranges over some set of values), and (b) each read returns the value last written, or `null` if the location has not been written.

Example 2: Define the **counter** data type with an increment operation *inc* that returns the updated counter value as the set of all sequences of the form $inc:1, inc:2, \cdots inc:n$ (for $n \in \mathbb{N}$).

Example 3: Define the **key-value store** data type to contain the sequences where (a) each invocation is of the form $wr(l,v):\texttt{ok}$ or $rd(l):v$ (where l ranges over some set of keys and v ranges over some set of values), and (b) each read returns the value last written to the same location, or `null` if the location has not been written.

Sequential data types are also used to define strong consistency models. Sequential consistency [15] and Linearizability [12] both require that for all concurrent interactions with an object, there exists a sequential witness that "justifies" the observed values. Thus, any system guaranteeing strong consistency must somehow guarantee a globally consistent linear order of operations.

2.2 Replicated Data Types

What distinguishes a replicated data type [6] from a sequential or concurrent data type is that *all operations are always available* regardless of temporary network partitions. Concretely, it requires that, at any location and time, any operation can be issued and returns a value immediately. All communication is asynchronous: the propagation of updates is a background process that takes a variable amount of time to complete.

Abstract Executions. For a replicated data type, a simple invocation sequence is not sufficient to define its semantics. For example, if two different replicas simultaneously update the data, the version history forks and is no longer linear. Thus, the key to a specification of a replicated data type is to generalize from an invocation *sequence* to an invocation *graph*, which we call an *abstract execution*. We define abstract executions as directed acyclic graphs like the ones shown in Fig. 1: each vertex is an invocation, edges represent *visibility*, and the subscripts represent *arbitration* timestamps.[1] The generalization of Definition 1 is now straightforward.

Definition 2: A replicated data type specification is a set of abstract executions.

Instead of ordering all invocations as a sequence, an abstract execution graph contains two ordering relations on the vertices, visibility (which is partial, and shown as arrows) and arbitration (which is total, and shown as timestamps). Roughly, visibility captures the fact that update propagation is not instantaneous, and arbitration is used for resolving conflicts consistently.

Consider a counter as in Example 2 that provides a single increment operation *inc* that returns the updated value. Then, a possible abstract execution could look as in Fig. 1(a). The idea is that the current value of the counter at any point of the execution is the number of visible increments (and thus zero if no increments are visible).

Example 4: Define the **replicated counter** data type to contain all abstract executions such that the value returned by each increment matches the number of increments that are visible to it, plus one.

[1] For formal definitions of abstract executions see [4, 6, 1].

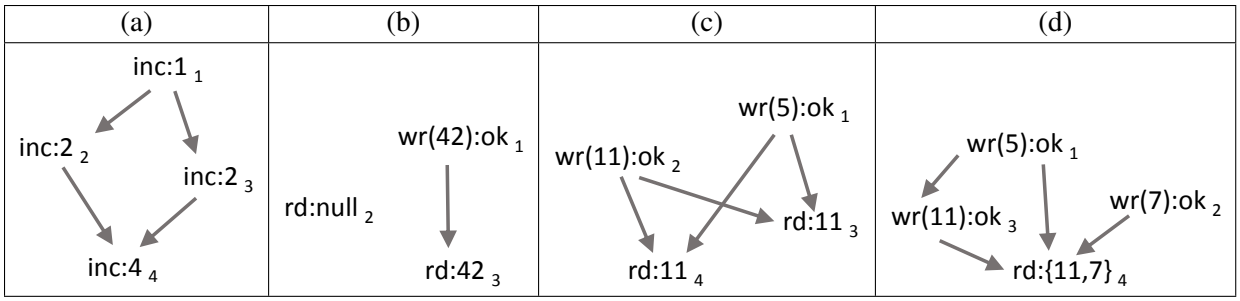


Figure 1: Three abstract executions; (a) an execution of the replicated counter, (b, c) executions of the last-writer-wins register, (d) an execution of the multi-value register

Note that the arbitration order does not appear in the definition. It is in fact irrelevant here: its function is to resolve conflicts, but for the counter, all operations are commutative, therefore we do not need to make use of the arbitration order at all. However, this is not true for all data types. Consider a register as in Example 1. Two possible abstract execution are shown in Fig. 1(b,c). Note that if more than one write is visible to a read then we must consistently pick a winner. This is what the arbitration can do for us: if there are multiple visible writes, the one with the highest timestamp wins.

Example 5: Define the **last-writer-wins register** data type to contain all abstract executions such that the value returned by each read matches the value written by the visible write with maximal arbitration, or `null` if no writes are visible.

We can think of visibility and arbitration as two aspects of the execution that were synonymous in sequential executions, but are no longer the same for replicated data types. Specifically, even if an operation A is ordered before operation B by its arbitration timestamp, this does not necessarily imply that A is also visible to B.

Both the replicated counter and the last-writer-wins register take a very simple approach to conflict resolution. For the former, conflicts are irrelevant, and for the latter, it is assumed that the latest write is the only one that matters. However, this may not be enough. Sometimes, conflicts may need to be detected and handled in an application-specific way. The multi-value register data type (introduced by Dynamo [13]) achieves this by modifying the semantics of a read operation: instead of returning just the latest write, it returns all conflicting writes. An example execution is shown in Fig. 1(d).

Example 6: Define the **multi-value register** data type to contain all abstract executions such that each read returns the set of values written by visible writes that are maximal, i.e. which are not visible to any later visible write.

A multi-value register can be used by the application programmer to detect conflicting writes; once a conflict is detected, it can be resolved by writing a new value. It is typically offered as part of a key-value store, each value being a multi-value register.

2.3 System Guarantees

Abstract executions allow us to clearly articulate the values an operation may or may not return in a given context. Beyond that, they also allow us to formalize system-wide safety and liveness guarantees. Thus, we can precisely describe the behavior of a large class of differing implementations and consistency protocols.

Eventual Visibility. Perhaps the most important liveness property we expect from a replicated data type is that *all operations become eventually visible*. Technically, we can specify this by requiring that in all infinite abstract

executions, all operations are visible to all but finitely many operations. Note that this definition is effective even in executions where updates never stop.

Ordering Guarantees. To write correct applications, we may want to rely on some basic ordering guarantees. For example, we may expect that an operation performed at some replica is visible to all subsequent operations at the same replica. This is sometimes called the read-my-writes guarantee [17].

Causality. Causality means that if operation A is visible to operation B, and operation B is visible to operation C, then operation A should be visible to operation C as well. Thus, it corresponds simply to transitivity of the visibility relation. Typically, causal consistency is defined in a way that also implies the read-my-writes guarantee.

Multiple vs. single objects. Consistency guarantees may vary depending on whether we consider an individual object, or a store containing multiple objects. For example, some stores may guarantee causality between all accesses to one object, but not for accesses across multiple objects.

3 Protocols

Abstract executions represent the propagation of updates using an abstract visibility relation, without fixing a particular relationship to the underlying message protocols. This means that the same specifications can be implemented by a wide variety of protocol styles. The following categories are used to organize the replication protocol examples in [4].

Centralized. In a centralized protocol, replication is asymmetric, using a primary replica (perhaps on some highly reliable infrastructure) and secondary replicas (perhaps cheaper and less reliable). Eventual visibility in that case means that all updates are sent to the primary, and then back out to the secondaries.

Operation-Based. In an operation-based protocol, all replicas are equal. After performing an update operation, a replica broadcasts information about the operation to all other replicas. All operations are delivered to all replicas eventually, which guarantees eventual visibility. However, messages are not delivered in a consistent order.

Epidemic. In an epidemic protocol, all replicas are equal. However, replicas do not broadcast individual messages for each operation. Rather, they periodically send messages containing their entire state, which summarizes the effect of *all* operations they are aware of (thus, this is sometimes called a *state-based* protocol). Visibility of operations can thus spread via other nodes (indirectly like an infectious disesase). Eventual visibility can be guaranteed even if many messages are lost, as long as the network remains fully connected.

4 Results

Abstract executions as a specification methodology have enabled systematic research on weakly consistent data, and led to several theoretical results (both positive and negative).

Correctness Proofs. One can prove that a protocol implementation satisfies a specification by showing that for each concrete execution, there exists a corresponding abstract execution that satisfies the data type specification, eventual visibility, and any other system guarantees we wish to validate. This approach is used in [4, 6] to prove correctness of several optimized protocols that implement various replicated data types (counters, registers, multi-value registers, and key-value stores) using various architectures (centralized, broadcast-based, and epidemic) and for differing system guarantees (causality, session guarantees).

Bounds on Metadata Complexity. An important design consideration is the space required by *metadata*. We

define the *metadata overhead* of a replica to be the ratio of the size of all information stored, divided by the size of the current data content. In Burckhardt et al. [6], lower bounds for the worst-case metadata overhead are derived; specifically it is shown that epidemic counters, epidemic registers, and epidemic multi-value registers have a worst-case metadata overhead of at least $\Omega(n)$, $\Omega(\lg m)$, and $\Omega(n \lg m)$, respectively (where n is the number of replicas and m is the number of operations performed). Since these bounds match the metadata overhead of the best known implementations, we know that they are asymptotically optimal.

Strongest Always-Available Consistency. Another interesting question is how to maximally strengthen the consistency guarantees without forsaking availability. In [1], it is shown that a key-value store with multi-value-registers cannot satisfy a stronger consistency guarantee than observable causal consistency (OCC), a slightly stronger variant of causal consistency.

5 Living with Weak Guarantees

Many storage systems provide only the bare minimum: eventual visibility, without transaction support, and without inter-object causality. In such cases, it can be daunting to ensure applications preserve even basic data integrity. However, data abstraction can go a long way. For example, simple collection types such as sets and lists may be supported as replicated data types.

Even more powerful is the concept that application programmers can raise the abstraction level further, by defining their own custom data types. Taken to the extreme, we could consider the entire store to be a single database object, with operations defined by stored procedures, and with update propagation corresponding to queued transaction processing [2].

Operational Consistency Models. So far, we have defined replicated data types as a set of abstract executions, and have defined those sets by stating the properties its members must satisfy. This is known as the *axiomatic* approach. Another common approach is to use *operational* consistency models, i.e. define the set of possible executions by specifying an abstract machine that generates them. Operational models can augment our intuitions significantly, and make good mental reference models for visualizing executions. For example, the popular TSO memory model can be defined in both ways, axiomatically, or as a simple operational model [8]. The global sequence protocol (GSP) operational model [7] is an adaptation of the TSO model that defines a generic replicated data type, supporting both weak and strong consistency, using a simple centralized protocol.

6 Related Work

Protocols for replication have been around for a long time; recent interest in specialized protocols was spurred by the comprehensive collection by Shapiro et al. [16]. It describes counters, registers, multi-value registers, and a few other data types, and introduces the distinction between operation-based and state-based protocols. It does not however specify the behavior of these implementations abstractly.

The use of event graphs and relations for specifying consistency models can be traced back to the common practice of axiomatic memory consistency models, with a long history of publications that we shall elide here as they are only marginally relevant in this context. We refer the interested reader to [6] for some comparisons to the C++ memory model.

The use of a visibility relation is similar to the use of justification sets by Fekete et al. [9]. The use of both visibility and arbitration relations appears first in Burckhardt et al. [5], in the context of defining eventually consistent transactions. This basic idea was later significantly expanded and used to specify replicated data types, verify implementations, and prove optimality [6]. The latter paper also coins the term "abstract execution". An equivalent formalization is used in [1], where arbitration is represented implicitly as a sequence. A slight

generalization of abstract executions that can capture both weak and strong consistency models appears in [4], which also contains a variety of protocol examples and correctness proofs.

References

[1] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. In *Principles of Distributed Computing (PODC)*, pages 385–394, 2015.

[2] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 2nd ed. edition, 2009.

[3] E. A. Brewer. Towards robust distributed systems (abstract). In *Principles of Distributed Computing (PODC)*, 2000.

[4] S. Burckhardt. *Principles of Eventual Consistency*. Foundations and Trends in Programming Languages, 2014.

[5] S. Burckhardt, M. Fähndrich, D. Leijen, and M. Sagiv. Eventually consistent transactions. In *European Symposium on Programming (ESOP), (extended version available as Microsoft Tech Report MSR-TR-2011-117), LNCS*, volume 7211, pages 64–83, 2012.

[6] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. *SIGPLAN Not.*, 49(1):271–284, Jan. 2014.

[7] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 568–590, 2015.

[8] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[9] A. D. Fekete and K. Ramamritham. Consistency models for replicated data. In *Replication*, volume 5959 of *LNCS*, pages 1–17. Springer, 2010.

[10] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[11] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] R. Klophaus. Riak core: Building distributed applications without shared state. In *Commercial Users of Functional Programming (CUFP)*. ACM SIGPLAN, 2010.

[14] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[17] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems (PDIS)*, 1994.

[18] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symposium on Operating Systems Principles (SOSP)*, 1995.

Representation without Taxation: A Uniform, Low-Overhead, and High-Level Interface to Eventually Consistent Key-Value Stores

KC Sivaramakrishnan*
University of Cambridge, UK
sk826@cl.cam.ac.uk

Gowtham Kaki
Purdue University, USA
gkaki@purdue.edu

Suresh Jagannathan
Purdue University, USA
suresh@cs.purdue.edu

Abstract

Geo-distributed web applications often favor high availability over strong consistency. In response to this bias, modern-day replicated data stores often eschew sequential consistency in favor of weaker eventual consistency (EC) data semantics. While most operations supported by a typical web application can be engineered, with sufficient care, to function under EC, there are oftentimes critical operations that require stronger consistency guarantees. A few off-the-shelf eventually consistent key-value stores offer tunable consistency levels to address the need for varying consistency guarantees. However, these consistency levels often have poorly-defined ad hoc *semantics that is usually too low-level from the perspective of an application to relate their guarantees to invariants that must be respected by the application. Moreover, these guarantees are often defined in way that is strongly influenced by a specific implementation of the data store. While such low-level implementation-dependent solutions do not readily cater to the high-level requirements of an application, relying on ill-defined guarantees additionally complicates the already hard task of reasoning about application semantics under eventual consistency.*

In this paper, we describe Quelea*, a declarative programming model for eventually consistent data stores. A novel aspect of* Quelea *is that it abstracts the actual implementation of the data store via high-level programming and system-level models that are agnostic to a specific implementation of the data store. By doing so,* Quelea *frees application programmers from having to reason about their application in terms of low-level implementation specific data store semantics. Instead, programmers can now reason in terms of an abstract model of the data store, and develop applications by defining and composing high-level replicated data types.* Quelea *is equipped with a formal specification language that is capable of expressing precise semantics of high-level consistency guarantees (*e.g.*, causal consistency) in the abstract model. Any eventually consistent key-value store can support* Quelea *by implementing a thin shim layer and a choosen set of high-level consistency guarantees on top of its existing low-level interface. We describe one such instantiation on top of Cassandra, that includes support for causal and sequential consistency guarantees, and coordination-free transactions. We present a case study of a large web application benchmark to demonstrate* Quelea*'s practical utility.*

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was done at Purdue University, USA.

1 Introduction

Eventual consistency facilitates high availability, but can lead to surprising anomalies that have been well-documented [4, 15, 18, 8, 12]. While applications can often tolerate many of these anomalies, there are some that adversely effect the user experience, and hence need to be avoided. For instance, a social network application can tolerate out-of-order delivery of unrelated posts, but causally related posts need to be delivered in causal order; *e.g.,* a comment cannot be delivered before the post itself. The view count of a video on YouTube need not necessarily reflect the precise count of the number of views, but it should not appear to be decreasing. A bank account application may not always show the accurate balance in an account, but neither should it let the balance go below zero, nor should it admit operations that would lead it to display a negative balance.

Bare eventual consistency is often too weak to ensure such high-level application invariants; stronger consistency guarantees are needed. To help applications enforce such high-level invariants, off-the-shelf replicated data stores, such as Cassandra and Riak, offer tunable consistency levels on a per-operation basis: applications can specify the consistency level for every read and write operation they perform on the data store. However, consistency levels offered by these off-the-shelf stores are often defined at a very low-level. For example, consistency levels in Cassandra and Riak assume the values of ONE, TWO, QUORUM, ALL etc., describing how many physically distributed nodes comprising the store must respond before a read or write operation is declared successful. It is not immediately apparent what permutation of these low-level consistency guarantees would let the application enforce its high-level level invariants. For instance, what should be the consistency level of reads and writes to the `posts` table to guarantee causal order delivery of posts in the aforementioned social network application?

Furthermore, the semantics of low-level consistency guarantees are not uniform across different store implementations. For instance, while QUORUM means *strict quorum* (i.e., Lamport's quorum [11]) in the case of Cassandra, it means a *sloppy quorum* [8] in Riak. Complicating matters yet further, consistency semantics is often imprecisely, or even inaccurately, defined in the informal vendor-hosted documentations. For instance, Datastax's Cassandra documentation [7] claims that one can achieve "strong consistency" with "quorum reads and writes" in Cassandra. While this claim appears reasonable superficially (because a pair of quorum operations are serialized at least at one node), it is incomplete, at best, and inaccurate at worst.[1] Another example of a low-level consistency enforcement construct with vaguely defined semantics is Cassandra's Compare-and-Set (CAS) operation, which is advertised as a "lightweight transaction" and exposed as a conditional write query (*e.g.,* `INSERT INTO users VALUES ... IF NOT EXISTS`). The addition of CAS to Cassandra was coupled with the introduction of a new consistency level named SERIAL. Surprisingly, SERIAL is not a valid query-level consistency parameter for a write (conditional or not), while the others (*e.g.,* ONE) are valid.[2] Furthermore, Cassandra accepts a new *protocol-level* consistency parameter for a CAS operation that can be set to SERIAL, but its informal description doesn't explain how this parameter interacts with the query-level consistency parameter. The only way to unravel this complexity is to understand low-level details of the operator's underlying Paxos-based implementation. Mired in this quagmire of low-level implementation details, it is easy to lose track of our original goal - ensuring the high-level semantics guarantees required by the application are met as efficiently as possible by the implementation.

In this paper, we describe QUELEA, a declarative programming framework for eventually consistent data stores that was built to address the issues discussed above. QUELEA can be realized as a thin layer on top of any off-the-shelf eventually consistent key-value store, and as such, provides a uniform implementation-independent

[1] The devil is in the details of the timestamp-based last-writer-wins conflict resolution strategy in Cassandra, which need not necessarily pick the last writer due to inevitable clock drift across nodes. [9] presents a counterexample.

[2] Given the advertised use cases for lightweight transactions (such as maintaining uniqueness of usernames), one might expect a CAS to be SERIAL by default. It is therefore unintuitive that CAS accepts a consistency parameter, at least to the developers of *cassandra-cql*, a popular Haskell library for programming with Cassandra, whose API for CAS operation incorrectly hardcodes the parameter to SERIAL. This bug has been reported and fixed.

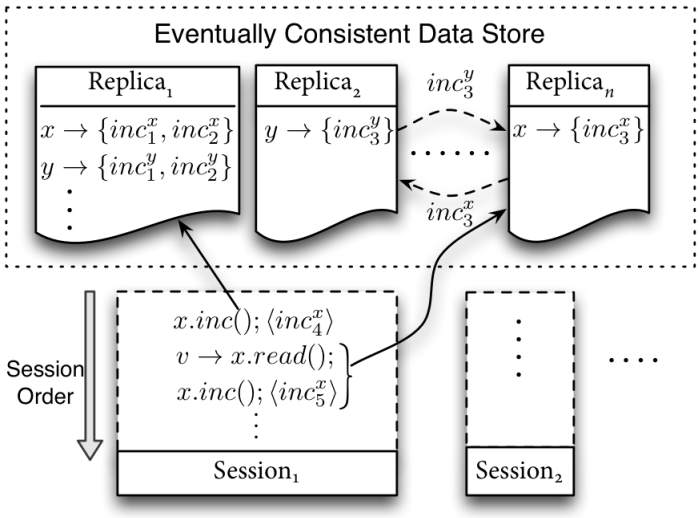


Figure 1: QUELEA system model.

interface to the data store. QUELEA programmers reason in terms of an abstract system model of an eventually consistent data store (ECDS), and any functionality offered by the store in addition to bare eventual consistency, including stronger consistency guarantees, transactions with tunable isolation levels etc., is required to have a well-defined semantics in the context of this abstract model. We show that various high-level consistency guarantees (eg., causal consistency) and various well-known isolation levels for transactions (eg., read committed) indeed enjoy such properties. QUELEA is additionally equipped with an expressive specification language that lets data store developers succinctly describe the semantics of the functionality they offer. A similar specification language is exposed to application programmers, who can declare the consistency requirements of their application as axiomatic specifications. Specifications are constructed using primitive consistency relations such as *visibility* and *session order* along with standard logical and relational operators. A novel aspect of QUELEA is that it can directly compare the specifications written by application programmers and data store developers since both are written within the same specification language, and can thus automatically map application requirements to the appropriate store-level guarantees. Consequently, QUELEA programmers can write portable code that automatically adapts to any data store that can express its functionality in terms of QUELEA's abstract system model.

Another key advantage of QUELEA is that it allows the addition of new replicated data types to the store, which obviates the need to support data types with application-specific semantics at the store level. Replicated data types (RDTs) are ordinary data structures, such as sets and counters, but with their state replicated across multiple replicas. As such, they offer useful high-level abstractions to build applications on top of weakly consistent replication. Weak consistency admits the possibility of concurrent conflicting updates to the state of the data structure at different replicas. The definition of an RDT must therefore specify its convergence semantics (i.e., *how* conflicting updates are resolved), along with its consistency properties (i.e., *when* updates become visible). QUELEA achieves a clean separation of concerns between these two aspects of the RDT definition, permitting *operational* reasoning for conflict resolution, and *declarative* reasoning for consistency. The combination of these techniques enhances overall programmability and simplifies reasoning about application correctness.

2 System Model

Figure 1 summarizes the abstract system model of a data store exposed to the QUELEA programmer. The store is a collection of *replicas*, each storing a set of *objects* $(x, y, \ldots)$ of a replicated data type. For the sake of an

example, let x and y represent objects of an *increment-only counter* replicated data type (RDT) that admits *inc* and *read* operations. The state of an RDT object is represented as the set of all updates (effectful operations, or simply *effects*) performed on the object. In Fig. 1, the state of x at replica 1 is the set $\{inc_1^x, inc_2^x\}$, where each inc_i^x denotes an *inc* effect on x.

Clients interact with the data store via concurrent *sessions*, where each session is a sequence of operations that a client invokes on any of the objects contained in the store. Note that clients have no control over which replica an operation is applied to; the data store may choose to route the operation to any replica in order to minimize latency, load balance, etc. For example, the *inc* and *read* operations invoked by the same session on the same object, may be applied on different replicas because replica 1 (to which the *inc* operation is applied, say) might be unreachable when the client invokes a subsequent *read*.

When an operation is applied to a replica, it is said to witness the state of its object at that replica. For example, *x.inc* applied to replica 1 witnesses the state of x as $\{inc_1^x, inc_2^x\}$. We say that the effects inc_1^x and inc_2^x are *visible* to the effect (inc_4^x) of *x.inc*, written logically as $\mathsf{vis}(inc_1^x, inc_4^x) \wedge \mathsf{vis}(inc_2^x, inc_4^x)$, where vis stands for the irreflexive and asymmetric visibility relation between effects over the same object. The notion of visibility is important since the result of an operation often depends on the set of visible effects[3]. For instance, a *read* on x applied to the last replica in Fig. 1 returns 1 since it only witnesses the effect (inc_3^x) of a single *x.inc* operation.

A visibility relation between two effects implies that the former operation has happened before the latter (since the latter has witnessed the effect of the former). However, visibility is not enough to capture a happens-before order between operations. As Fig. 1 demonstrates, a pair of operations from the same session, although one happens before the other, need not be visible to each other. To capture happens-before, we define an irreflexive transitive *session order* relation that relates the effects of operations arising from the same session. For example, in Fig. 1, inc_4^x and inc_5^x are in session order (written logically as $\mathsf{so}(inc_4^x, inc_5^x)$).

The effect added to a particular replica is asynchronously sent to other replicas, and eventually merged into all other replicas. Observe that this model is independent of the resolution strategy for concurrent conflicting updates, and instead preserves *every* update. Update conflicts are resolved when an operation reduces over the set of effects on an object at a particular replica. The model, however, admits all the inconsistencies associated with eventual consistency, some of which could adversely impact the usability of the application. We call such unacceptable inconsistencies as *anomalies*. Stronger consistency guarantees are needed to prevent unwanted anomalies.

In the next section we concretize, in Quelea, the *counter* application described informally above, followed by the anomalies the application admits under our model. Next, we show that strengthening the model with a few simple guarantees is enough to prevent these anomalies. We introduce a specification language that lets us naturally express such additional requirements. Finally, we show that well-known high-level consistency guarantees have precise semantics under our model, and hence can be expressed as formulas in our specification language. This makes it straightforward to compare application requirements with consistency guarantees, and determine the appropriate consistency semantics required to prevent anomalies.

3 Programming with Quelea

3.1 RDT Definition

Figure 2 shows the implementation of a counter RDT in Quelea. The data type `Ctr` represents the counter's effect type. Every RDT in Quelea is associated with an effect type that specifies the effects allowed on the objects of that type. An RDT is defined merely a list of its effects. The type of an RDT operation is an instance of the following type, written in Haskell syntax as:

```
type Operation e a r = [e] → a → (r, Maybe e)
```

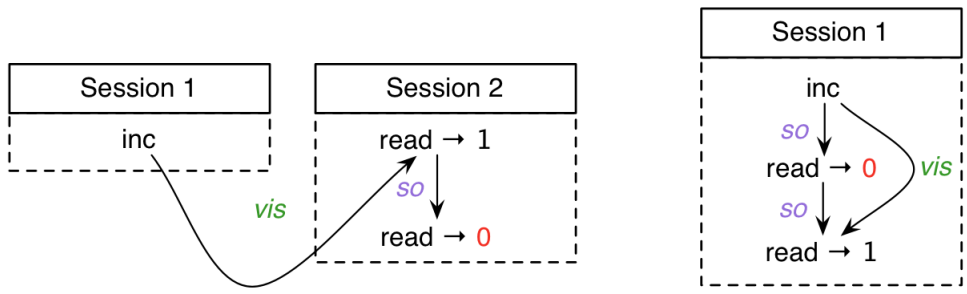
[3]We abuse the visibility relation by informally extending it to operations (including read-only operations, which produce no effects).

```haskell
-- CtrEff is the data type of counter effects.
-- Inc is its only inhabitant.
data CtrEff = Inc
-- Counter (rather, its state) is defined simply
-- as a list of counter effects.
type Counter = [CtrEff]
-- A read operation reads the value of the counter
-- by counting the number of Inc effects.
read :: Counter → () → (Int, Maybe Ctr)
read hist _ = (length hist, Nothing)
-- An inc operation simply generates an Inc effect.
inc :: Counter → () → ((), Maybe Ctr)
inc hist _ = ((), Just Inc)
```

Figure 2: Definition of a counter expressed in Quelea (Haskell syntax).

An operation on an RDT accepts a list of effects (the *history* of updates representing the state of the object at some replica), and an input argument, and returns a result along with an optional effect. While a read-only operation (eg., `read`) generates no effect (i.e., it returns `Nothing`), a write-only operation (eg., `inc`) returns a new effect.

3.2 Anomalies under Eventual Consistency



(a) Monotonic Violation (b) Missing Update

Figure 3: Anomalies possible under eventual consistency for the counter read operation.

Observe that the counter RDT does not admit a decrement operation. Therefore, the value of a counter should appear to be monotonically increasing. Indeed, this property is what makes the RDT useful to implement, for example, a video view counter on YouTube. Unfortunately, the monotonicity invariant can be violated in an eventually consistent execution.

Consider the execution shown in Figure 3a. Session 1 performs an `inc` operation on the counter, while Session 2 performs two `read` operations. The first `read` witnesses the effect of `inc` from Session 1, hence returns 1. The second `read`, however, does not witness `inc`, possibly because it was served by a replica that has not yet merged the `inc` effect. It returns 0, thus violating the monotonicity invariant.

In order for counter's value to appear monotonically increasing, the second `read` should also witness the effect of `inc`, because it was witnessed by the preceding `read`. Because eventually consistent `read` operations do not ensure this property, `read` operations need to be executed at a stronger consistency level. The choice of consistency level must guarantee that if a `read` witnesses an `inc` effect, all the subsequent `read` operations on the same counter object also witness that effect. If we let sameobj relate effects over the same object, we can

$$
\begin{array}{llll}
a, b \in \texttt{EffVar} & & \mathsf{Op} \in \texttt{OperName} & \\
\psi & \in \texttt{Spec} & ::= & \forall(a:\tau).\psi \mid \forall a.\psi \mid \pi \\
\tau & \in \texttt{EffType} & ::= & \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in \texttt{Prop} & ::= & \mathsf{true} \mid R(a,b) \mid \pi \vee \pi \mid \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in \texttt{Relation} & ::= & \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \mid R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

Figure 4: Syntax (Context-Free Grammar) of the specification language.

formalize this requirement using visibility (vis) and session order (so) relations:

$$\forall(a:\texttt{inc}),(b,c:\texttt{read}).\ \mathsf{vis}(a,b) \ \wedge\ \mathsf{so}(b,c) \ \wedge\ \mathsf{sameobj}(b,c) \Rightarrow \mathsf{vis}(a,c)$$

The above formula is in fact a valid specification that can be associated with counter RDT operations in Quelea. Note that the specification captures the guarantees required to enforce the monotonicity invariant with respect to the abstract model of the store described in § 2. In particular, the specification does not refer to the low-level details of any specific data store. Our observation is that if consistency levels can also be specified in a similar manner, we can eliminate the need for the application programmer to understand the low-level nuances of the data store to enforce the required invariants; understanding their semantics in the abstract model is enough. In the following sections, we demonstrate that this is indeed possible. We first introduce our specification language.

3.3 Specification Language

The syntax of our specification language is shown in Figure 4. The language is based on first-order logic (FOL), and admits prenex universal quantification over typed and untyped effect variables. We use a special effect variable ($\hat{\eta}$) to denote the effect of the *current operation* - the operation for which a specification is being written. The type of an effect is simply the name of the operation (eg: `inc`) that induced the effect. We admit disjunction in types to let an effect variable range over multiple operation names. The specification $\forall(a : \tau_1 \vee \tau_2).\ \psi$ is just syntactic sugar for $\forall a.(\mathsf{oper}(a,\tau_1) \vee \mathsf{oper}(a,\tau_2)) \Rightarrow \psi$. An untyped effect variable ranges over all operation names.

The syntactic class of relations is seeded with primitive vis, so, and sameobj relations, and also admits derived relations that are expressible as union, intersection, or transitive closure of primitive relations. Commonly used derived relations are the *same object session order* ($\mathsf{soo} = \mathsf{so} \cap \mathsf{sameobj}$), *happens-before order* ($\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^{+}$) and the *same object happens-before order* ($\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^{+}$). For example, the same object session order (soo) can be used to succinctly represent the specification of counter RDT's consistency requirement:

$$\forall(a:\texttt{inc}),(b,c:\texttt{read}).\ \mathsf{vis}(a,b) \ \wedge\ \mathsf{soo}(b,c) \Rightarrow \mathsf{vis}(a,c)$$

Same object happens-before order (hbo) captures causal order among operations on same object. For example, if an `inc` is visible to a `read`, and the `read` precedes another `read` in session order (all operations on the same counter object), then the `inc` and the second `read` are related by hbo, although they may not be directly related by vis or soo. The causal relationship is transitive, via the first `read`. As such, hbo is useful to capture the causal consistency condition, which requires an operation to witness all the causally preceding operations perfomed on the same object (i.e., operations that precede the current operation in hbo). This condition is formalized in the following section.

3.4 Consistency Guarantees

To help programmers eliminate certain classes of anomalies in their applications, Terry *et al.* equip their data store Bayou [17] with four incomparable consistency levels called *session guarantees* [18]. While Terry *et al.*

realize efficient implementations of session guarantees making use of low-level properties of their store, the semantics of these guarantees can nonetheless can be captured succinctly within QUELEA's abstract model thus:

$$\begin{aligned}
\text{Read Your Writes (RYW)} &::= \forall a, b.\ \mathsf{soo}(a,b) \Rightarrow \mathsf{vis}(a,b)\\
\text{Monotonic Reads (MR)} &::= \forall a, b, c.\ \mathsf{vis}(a,b) \wedge \mathsf{soo}(b,c) \Rightarrow \mathsf{vis}(a,c)\\
\text{Monotonic Writes (MW)} &::= \forall a, b, c.\ \mathsf{soo}(a,b) \wedge \mathsf{vis}(b,c) \Rightarrow \mathsf{vis}(a,c)\\
\text{Writes Follow Reads (WFR)} &::= \forall a, b, c, d.\ \mathsf{vis}(a,b) \wedge \mathsf{vis}(c,d) \wedge (\mathsf{soo} \cup =)(b,c) \Rightarrow \mathsf{vis}(a,d)
\end{aligned}$$

Consider a Monotonic Reads (MR) session guarantee. The semantics of MR guarantees that if the effect of an operation a is visible to the effect of b, and b precedes c in (same object) session order, then a will also be made visible to c. Recall that this is precisely the guarantee required by the counter to enforce monotonicity. In fact, by restricting the bound variable a in MR's specification to range over `inc` effects, and bound variables b and c to range over `read` effects, we can easily conclude that executing `read` at an MR consistency level is sufficient to enforce the monotonicity invariant.

Like MR, the semantics of other session guarantees are categorically stated by their specifications. Read-Your-Writes (RYW), for example, guarantees that an operation (b) witnesses the effect of every preceding operation (a) in the session. A `read` operation executed at RYW consistency level therefore witnesses every previous `inc` operation from the same session. This guarantee is necessary to avoid the anomaly shown in Fig. 3b, where a `read` that succeeds an `inc` fails to witness the effect of `inc`, but a later `read` witnesses the effect. The anomaly can also be avoided by running `inc` and `read` under the QUORUM consistency level offered by some off-the-shelf key-value stores, but doing so requires non-trivial reasoning over the semantics of quorum operations (*e.g.*, strict/sloppy) and conflict resolution strategies (eg., LWW) to arrive at this conclusion. In contrast, reasoning with high-level consistency guarantees, such as MR and RYW, circumvents this complexity.

The precise characterization of guarantees as specifications facilitates the use of automatic analyses to determine if a consistency level meets application requirements. For instance, consider a data store that offers the following three consistency levels:

$$\begin{aligned}
\text{Causal Visibility (CV)} &::= \forall a, b, c.\ \mathsf{hbo}(a,b)\ \wedge\ \mathsf{vis}(b,c) \Rightarrow \mathsf{vis}(a,c)\\
\text{Causal Consistency (CC)} &::= \forall a, b, c.\ \mathsf{hbo}(a,b) \Rightarrow \mathsf{vis}(a,b)\\
\text{Strong Consistency (SC)} &::= \forall a, b.\ \mathsf{sameobj}(a,b) \Rightarrow \mathsf{vis}(a,b) \vee \mathsf{vis}(b,c)
\end{aligned}$$

It is not immediately apparent which among CV, CC and SC meet the requirements of a counter (CR). Fortunately, we can leverage the power of automated theorem provers (*e.g.*, Z3) to prove that the specification of CR is stronger than CV, but weaker than CC and SC, thus letting us deduce that counter's requirements can be met both under causal and strong consistency levels. A theorem prover can also be used to prove that among CC and SC, CC is weaker. Assuming that weaker guarantees incur lower cost to enforce availability, it is reasonable to conclude that counter's `read` operations should be executed under causal consistency to enforce monotonicity.

The analysis described informally above is formalized as a *classification scheme* [15] in QUELEA. This scheme completely automates the choice of consistency levels in QUELEA, thus eliminating the need for programmers to understand the semantics of different consistency levels. The ease of reasoning with precisely stated high-level guarantees demonstrates the advantage of exposing the functionality of the data store via QUELEA, as against the low-level ad hoc interfaces currently offered by many ECDS.

4 Transactions

Real-world applications often need to perform atomic operations that span multiple objects. An example is a `transfer` operation on bank accounts, which needs to perform a `withdraw` operation on one bank account, and a `deposit` operation on another. Transactions are usually the preferred means to compose sets of operations into a single atomic operation. However, a classical ACID transaction model requires inter-replica

coordination, leading to the loss of availability. To address this problem, several recent systems [16, 3, 1] have proposed *coordination-free transactions* that offer atomicity, remain available under network partitions, but only provide weaker isolation guarantees. Several variants of coordination-free transactions have subtly different isolation semantics and widely varying runtime overheads. Fortunately, the semantics of a large subset of such transactions can be captured elegantly in the abstract model of Quelea. Towards this end, we extend the contract language with a new term under quantifier-free propositions - $\mathsf{txn}\ S_1\ S_2$, where S_1 and S_2 are sets of effects, and which introduces a new primitive equivalence relation sametxn that holds for effects from the same transaction. $\mathsf{txn}\{a,b\}\{c,d\}$ is just syntactic sugar for $\mathsf{sametxn}(a,b) \wedge \mathsf{sametxn}(c,d) \wedge \neg\mathsf{sametxn}(a,c)$, where a and b considered to belong to the *current* transaction. Since atomicity is a defining characteristic of a transaction, we extend our formal model with the following axiom, which guarantees that a transaction is visible in its entirety, or it is not visible at all:

$$\forall a,b,c.\ \mathsf{txn}\{a,b\}\{c\} \wedge \mathsf{sameobj}(a,b) \wedge \mathsf{vis}(a,c) \Rightarrow \mathsf{vis}(b,c)$$

4.1 Isolation Requirements

§ 3.2 demonstrates how the consistency requirements of a counter RDT's `read` operation can be expressed in Quelea. In a similar manner, Quelea's specification language allows applications to declare isolation requirements for their transactions.

Consider an implementation of a bank account RDT in Quelea with three operations – `withdraw`, `deposit`, and `getBalance`, each with straightforward semantics. Additionally, we define two transactions – `save(amt)`, which transfers `amt` from current (c) to savings (s), and `totalBalance`, which returns the sum of the balances of individual accounts. Our goal is to ensure that `totalBalance` returns the result obtained from a consistent snapshot of the object states. The Quelea code for these transactions is given below:

```
save amt = atomically do
    b ← withdraw c amt
    -- b is true iff withdraw succeeds.
    when b $ deposit s amt
```

```
totalBalance = atomically do
    b1 ← getBalance c
    b2 ← getBalance s
    return b1 + b2
```

The `atomically` construct ensures that the effects of the operations are made visible atomically. However, atomicity itself is insufficient in this case, as it does not guarantee that both `getBalance` operations (in `totalBalance`) witness the effects of `save`. Consequently, `getBalance` on `s` may not witness the `deposit` on `s` from `save`, although `getBalance` on `c` witnesses the `withdraw` on `c`, resulting in `totalBalance` reporting an inconsistent balance.

Observe that the aforementioned anomaly can be averted by requiring that both `getBalance` operations in a `totalBalance` transaction witness the same set of `save` actions. This requirement can be captured succinctly in our specification language:

$$\forall(a,b : \texttt{getBalance}),(c : \texttt{withdraw} \vee \texttt{deposit}),(d : \texttt{withdraw} \vee \texttt{deposit}).$$
$$\mathsf{txn}\{a,b\}\{c,d\} \wedge \mathsf{vis}(c,a) \wedge \mathsf{sameobj}(d,b) \Rightarrow \mathsf{vis}(d,b)$$

The key idea in the above definition is to use the txn primitive to relate operations on different objects performed in a single transaction.

4.2 Isolation Guarantees

The isolation semantics of a transaction determines how the transaction witnesses the effects of previously committed transactions[4]. As mentioned previously, the isolation semantics of many variants of coordination-free transactions can be expressed in QUELEA's specification language. For demonstration, we pick three well-understood coordination-free transactions – Read Committed (RC) [2], Monotonic Atomic View (MAV) [1] and Repeatable Read (RR) [2], and express their isolation semantics in QUELEA.

ANSI RC isolation level guarantees that a transaction only witnesses the effects of committed transaction:

$$\forall a, b, c.\ \mathsf{txn}\{a\}\{b, c\} \wedge \mathsf{sameobj}(b, c) \wedge\ \mathsf{vis}(b, a) \Rightarrow \mathsf{vis}(c, a)$$

Note that RC is the dual of atomicity; it can be guaranteed with no additional effort if all transactions in the system are atomic.

MAV semantics ensures that if some operation in a transaction T_1 witnesses the effects of another transaction T_2, then subsequent operations in T_1 will also witness the effects of T_2:

$$\forall a, b, c, d.\ \mathsf{txn}\{a, b\}\{c, d\}\ \wedge\ \mathsf{so}(a, b)\ \wedge\ \mathsf{vis}(c, a)\ \wedge\ \mathsf{sameobj}(d, b) \Rightarrow \mathsf{vis}(d, b)$$

MAV semantics is useful for maintaining the integrity of foreign key constraints, materialized views and secondary updates [1].

ANSI RR semantics requires that the transaction witness a snapshot of the data store state. More concretely, RR requires that if an operation in transaction T_1 witnesses the effects of transaction T_2, then all the remaining operations should also witness the effects of T_2:

$$\forall a, b, c, d.\ \mathsf{txn}\{a, b\}\{c, d\}\ \wedge\ \mathsf{vis}(c, a)\ \wedge\ \mathsf{sameobj}(d, b) \Rightarrow \mathsf{vis}(d, b)$$

Note that this is precisely the semantics required by the `totalBalance` transaction to ensure that it returns a balance that reflects a consistent snapshot of the data store. Hence, it is sufficient to execute `totalBalance` at RR isolation level.

The ease of reasoning with precisely stated high-level guarantees thus extends to transactions as well. Furthermore, the ability to express the isolation requirements of an application, along with the semantics of various isolation guarantees provided by the store in the QUELEA's specification language allows us to define a classification scheme [15], similar to the one in § 3.4, to automatically map requirements to guarantees.

5 Implementation

The abstract model of QUELEA, optionally extended with a chosen set of high-level consistency and isolation guarantees, can be instantiated on top of any eventually consistent key-value store. We now describe a reference implementation of QUELEA on top of Cassandra's key-value store [10]. Our implementation supports CV, CC, and SC consistency levels (§ 3.4) for data type operations, and RC, MAV, and RR isolation levels (§ 4.2) for transactions. This functionality is implemented entirely on top of the standard interface exposed by Cassandra. From an engineering perspective, leveraging an off-the-shelf data store enables an implementation comprising roughly only 2500 lines of Haskell code, which is packaged as a library [13].

5.1 Object State

Cassandra adopts a data model similar to that of BigTable [5]. Each row is identified by a composite *primary key*, whose first component is the *partition key*, and remaining components are *clustering columns*. Like Dynamo [8], Cassandra uses consistent hashing on partition keys to map rows to machines in the cluster that maintain a

[4]It is informative to compare isolation with atomicity. While the former constrains how the current transaction witnesses the effects of other transactions, the later determines how other transactions witness the effects of the current transaction.

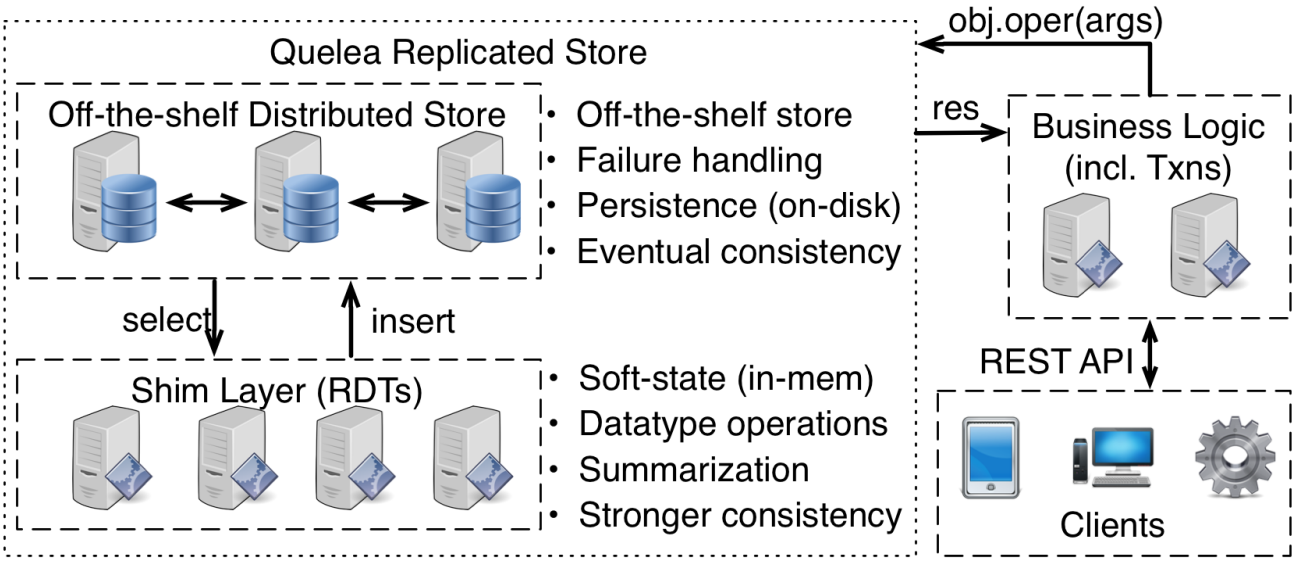


Figure 5: Implementation Model.

replica. Hence, rows with the same partition key (together referred to as a *partition*) are mapped to the same machine[5]. Within each partition, rows are clustered and sorted on the values of their clustering columns. QUELEA relies on these properties to minimize the latency of querying object state stored in Cassandra.

Recall that the state of an object in QUELEA is represented as a set of effects. An effect generated as a result of executing an effectful operation (eg., `inc` or `withdraw`) inserts a new row $(o, s, i, txn, val, deps)$, where o is the identifier of the object on which the operation is performed, s is the identifier of the session that issued the operation, and i is operation's sequence number within its session. The optional *txn* column identifies the transaction (if any). The column *val* is the value associated with the effect (eg: `Withdraw 50`). *deps* is the set of identifiers of *dependencies* of this operation and is defined as $deps(e) = \{e_1 \mid \mathsf{vis}(e_1, e) \wedge \neg(\exists e_2.\mathsf{vis}(e_1, e_2) \wedge \mathsf{vis}(e_2, e))\}$. The primary key of this row is a composite of (o, s, i), making o the partition key, and s and i clustering columns. Thus, effects on the same object belong to the same partition, minimizing the latency of reading its state. Within the partition, effects are clustered (and sorted), first on s, then on i. Consequently, range queries on s and i, such as the set of effects that precede a given effect in the same session (i.e., effects with same o and s, but lesser i), are efficient. This data model has been crafted to enable efficient implementations of consistency guarantees, such as session guarantees described in § 3.4.

5.2 Operation Consistency

The consistency semantics of replicated data types are implemented and enforced in the *shim layer* above Cassandra. The overall system architecture is shown in Fig. 5. Note that the shim layer node simply acts as a soft-state write-through cache and can safely be terminated at any point. Similarly, new shim layer nodes can be spawned on demand.

The shim layer maintains a causally-consistent in-memory state of a subset of objects in the system. A causally-consistent state of an object is a subset of effects on the object that are closed under the hbo relation. In other words, the shim layer includes an effect e over an object o, only if it also includes all effects e' over o that happened before e (i.e., $\mathsf{hbo}(e', e)$). Since all read operations are served by the shim layer, a read operation only ever witnesses a causally-consistent state of its object. Recall that this is precisely the CV consistency level

[5]Dynamo's consistent hashing maps a partition key to a *vnode*, which, in common case, maps to a single physical machine.

described in § 3.4. Thus, CV is the default consistency level in this implementation of Quelea.

The shim layer nodes periodically fetches updates from the backing store, thereby ensuring that later operations witness more recent CV-consistent state. For causally consistent operations, however, updates need to be fetched on-demand. For example, if a causally consistent operation op on o is the i^{th} operation in session s, and if the effect of $i-1^{th}$ operation in s is not present in the shim layer state of o, then a blocking read query for a row with primary key $(o, s, i-1)$ needs to be issued by the shim layer.

Strongly consistent operations are performed after obtaining a distributed global lock. Distributed lock is implemented with the help of Cassandra's conditional updates (lightweight transactions). To prevent deadlocks due to crash failures of the lock owner, the lock is leased only for a pre-determined amount of time. Lease functionality is implemented using Cassandra's support for expiring columns.

5.3 Transactions

Multi-key coordination-free transactions are implemented in Quelea by exploiting the shim layer's default CV consistency guarantee. Recall that the shim layer does not include an effect unless all its dependencies are also included. For every transaction, Quelea instantiates a special transaction marker effect m that is included as a dependence to every effect generated in the transaction. Importantly, the marker m is not inserted into the backing store until and unless the transaction finishes execution. Now, any replica which includes one of the effects from the transaction must include m, and transitively must include every effect from the transaction. This ensures atomicity and satisfies the RC requirement.

MAV semantics is implemented by keeping track of the set of transaction markers M witnessed by the transaction, and before performing an operation at some replica, ensuring that M is a subset of the transaction markers included at that replica. If not, the missing effects are synchronously fetched. RR semantics is realized by capturing an optimized snapshot of the state of some replica; each operation from an RR transaction is applied to this snapshot state. Any generated effects are added to this snapshot.

5.4 Summarization

Observe that the state of an object in Quelea grows monotonically as more effectful operations are performed on the object. Unbounded growth of state would make querying prohibitively expensive at some point. To keep state size in check, Quelea summarizes object state both in the shim layer node and the backing store, typically when the number of effects on an object crosses a tunable threshold. Quelea's summarization is similar in nature to the *major compaction* operation on *SSTables* in BigTable. While BigTable's compaction summarizes reads and writes at the storage layer, Quelea summarizes effects with application-specific semantics at the application layer. Hence, the semantics of summarization in Quelea is application-specific. Quelea therefore requires its applications to implement a special `summarize` function that is called whenever the state needs to be summarized. For example, a bank account RDT's `summarize` function may summarize a state comprising multiple `withdraw` and `deposit` effects into a single `deposit` effect, thus drastically reducing the number of effects that need to be kept track of by the shim layer and the store.

6 Evaluation

We have used Quelea to implement various applications, which include individual RDTs as well as larger applications composed of several RDTs. In order to ensure certain high-level invariants, these applications require various kinds of consistency and isolation guarantees from the data store. For instance, the microblogging application requires causal consistency for its `getTweet` operation to ensure the causal order delivery of tweets (§ 1). RUBiS [14], an eBay-like auction site, requires MAV isolation level for its `cancelBid` transaction to maintain the integrity of data in its materialized views. If an application developer were to implement these

applications on top of the default interfaces exposed by a typical off-the-shelf data store, she would either have to rely on ill-specified low-level functionality of the store, or build the required high-level functionality herself. With QUELEA, we were able to derive the high-level guarantees required by these applications by merely stating their requirements as specifications with respect to the abstract model[6].

For performance evaluation, we commissioned Amazon EC2 instances, and deployed QUELEA applications in an environment similar to the production environment of medium-scale web applications. We then ran these applications on realistic workloads constructed from standard benchmarks, such as YCSB [6] and RUBiS bidding mix. Our experiments [15] show that (a). QUELEA's expressive programming model incurs no more than 30% (resp. 20%) of latency (resp. throughput) overhead when compared to the native implementation on top of Cassandra (both run under default consistency levels) with 512 concurrent clients, and (b). with a distribution of 50% SC, 25% CC, and 25% EC operations, QUELEA incurred 41% (18%) higher latency (lower throughput) than 100% EC operations, when compared to 162% (52%) higher latency (lower throughput) incurred by 100% SC operations. These experiments demonstrate that the performance penalty of supporting QUELEA on top of an existing key-value stores is within reasonable limits, and that there is a significant performance incentive for applications to rely on QUELEA to enforce their high-level invariants rather than choosing SC for every operation.

7 Conclusion

Although modern web applications settle for eventual consistency in return for high availability, they nonetheless need stronger consistency guarantees to support small, yet significant, fraction of their functionality. The de facto interfaces exposed by well-known eventually consistent data stores are often ill-specified and too low-level from the perspective of an application developer. There currently exists a wide chasm between the high-level application-specific consistency requirements, and low-level store-specific tunable consistency levels. This paper describes QUELEA, a programming framework and a system that proposes to address this gap by lifting and standardizing the interface to an eventually consistent data store. We present many examples that demonstrate the advantage of reasoning with high-level interface exposed by QUELEA, when compared to the low-level interfaces offered by off-the-shelf data stores. We realize an instantiation of QUELEA on top of Cassandra, and illustrate the performance implications of using QUELEA, for both, applications and data stores.

References

[1] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[3] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually Consistent Transactions. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 67–86, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of*

[6] Both specifications combined span less than 5 lines. Source code available at `https://github.com/kayceesrk/Quelea/tree/master/tests`

the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[7] datastax developer blog. lightweight transactions in cassandra 2.0, 2016. URL `http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0`. accessed: 2016-02-14 2:30:00.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[9] Jepsen. Cassandra, 2016. URL `https://aphyr.com/posts/294-jepsen-cassandra`. accessed: 2016-02-14 2:40:00.

[10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[11] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[13] Quelea. Programming with Eventual Consistency over Cassandra, 2016. URL `https://hackage.haskell.org/package/Quelea`. Accessed: 2016-02-14 12:20:00.

[14] RUBiS. Rice University Bidding System, 2014. URL `http://rubis.ow2.org/`. Accessed: 2014-11-4 13:21:00.

[15] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM.

[16] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[17] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[18] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

Ovid: A Software-Defined Distributed Systems Framework to support Consistency and Change

Deniz Altınbüken, Robbert van Renesse
Department of Computer Science
Cornell University
{deniz,rvr}@cs.cornell.edu

Abstract

We present Ovid, a framework for building large-scale distributed systems that have to support strong consistency and at the same time need to be able to evolve quickly as a result of changes in their functionality or the assumptions they made for their initial deployment. In practice, organic growth often makes distributed systems increasingly more complex and unmanageable. To counter this, Ovid supports transformations*, automated refinements that allow distributed systems to be developed from simple components. Examples of transformations include replication, batching, sharding, and encryption. Refinement mappings prove that transformed systems implement the specification. The result is a* software-defined distributed system*, in which a logically centralized controller specifies the components, their interactions, and their transformations. Such systems can be updated on-the-fly, changing assumptions or providing new guarantees while keeping the original implementation of the application logic unchanged.*

1 Introduction

Building distributed systems is hard, and evolving them is even harder. A simple client-server architecture may not be able to scale to developing workloads. Sharding, replication, batching, and similar improvements will be necessary to incorporate over time. Next, it will be necessary to support online configuration changes as hardware is being updated or a new version of the system is being deployed. Online software updates will be necessary for bug fixes, new features, enhanced security, and so on. All this is sometimes termed "organic growth" of a distributed system. While we have a good understanding of how to build a strongly consistent service based on techniques such as state machine replication and atomic transactions, we do not have the technology to build *consistent systems* that are comprised of many services and that undergo organic growth.

In this paper we describe the design of *Ovid*, a framework for building, maintaining, and evolving distributed systems that have to support strong consistency. The framework leverages the concept of refinement [15] or, equivalently, backward simulation [20]. We start out with a relatively simple specification of *agents*. Each agent is a simple self-contained state machine that transitions in response to messages it receives and may produce output messages for other agents. Next, we apply *transformations* to agents such as replication or sharding. Each

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

transformation replaces an agent by one or more new agents. For each transformation we supply a refinement mapping [15] from the new agents to the original agent to demonstrate correctness, but also to be able to obtain the state of the original agent in case a reconfiguration is necessary. Transformations can be applied recursively, resulting in a tree of transformations.

A collection of agents itself is a state machine that transitions in response to messages it receives and may produce output messages for other agents. Consequently, a collection of agents can be considered an agent itself. When an agent is replaced by a set of agents, the question arises what happens to messages that are sent to the original agent. For this, each transformed agent has one or more *ingress agents* that receive such incoming messages. The routing is governed by routing tables: each agent has a routing table that specifies, for each destination address, what the destination agent is. Inspired by software-defined networks [7, 21], Ovid has a logically centralized controller, itself an agent, that determines the contents of the routing tables.

Besides routing, the controller determines where agents run. Agents may be co-located to reduce communication overhead, or run in different locations to benefit performance or failure independence. In order for the controller to make placement decisions, agents are tagged with location constraints. The result is what can be termed a "software-defined distributed system" in which a programmable controller manages a running system.

Ovid supports on-the-fly reconfiguration based on "wedging" agents [5, 1]. By wedging an agent, it can no longer make transitions, allowing its state to be captured and re-instantiated for a new agent. By updating routing tables the reconfiguration can be completed. This works even for agents that have been transformed, using the refinement mapping for retrieving state.

This paper is organized as follows. We start with discussing related work in Section 2. In Section 3 we present a system model for Ovid, including agents, communication, and transformation. Section 4 presents various examples of transformation. In Section 5 we discuss how Ovid supports online configuration changes. We describe how agents are scheduled and how routing is done in Section 6. We conclude in Section 7.

2 Related Work

There has been much work on automating distributed system creation and verification as well as implementing long-lived distributed systems that can function in evolving environments. In this section we look at both approaches and present related work.

2.1 Automated Distributed System Creation and Verification

Mace [13] is a language-based solution to automatically generate complicated distributed system deployments using high-level language constructs. Mace is designed as a software package that comprises a compiler that translates high-level service specifications to working C++ code. In Mace, a distributed system is represented as a group of nodes, where each node has a state that changes with message or timer events. To construct a distributed system using Mace, the user has to specify handlers, constants, message types, state variables and services in a high-level. The compiler then creates a working distributed application in C++ according to the specifications provided.

CrystalBall [28] is a system built on top of the Mace framework to verify a distributed system by exploring the space of executions in a distributed manner and having every node predict the outcome of their behavior. In CrystalBall, nodes run a state exploration algorithm on a recent consistent snapshot of their neighborhood and predict possible future violations of specified safety properties, in effect executing a model checker running concurrently with the distributed system. This is a more scalable approach compared to running a model checker from the initial state of a distributed system and doing exhaustive state exploration.

Similarly, other recent projects have been focusing on verifying distributed systems and their components automatically. In [23] Schiper et al. use the formal EventML [22] language to create specifications for a Paxos-

based broadcast protocol that can be formally verified in NuPRL [8]. This specification is then compiled into a provably correct and executable implementation automatically and used to build a highly available database.

In [26], Wilcox et al. present a framework, namely Verdi, for implementing practical fault-tolerant distributed systems and then formally verifying that the implementations meet their specifications. Verdi provides a Coq toolchain for writing executable distributed systems and verifying them, a mechanism to specify fault models as network semantics, and verified system transformers that take an existing system and transform it to another system that makes different assumptions about its environment. Verdi is able to transform systems to assume different failure models, even if it is not able to transform systems to provide new guarantees.

IronFleet [9] proposes building and verifying distributed systems using TLA-style state-machine refinements and Hoare-logic verification. IronFleet employs a language and program verification toolchain Dafny [16] that automates verification and it enables proving safety and liveness properties for a given distributed system.

Systems like CrystalBall, Verdi, and IronFleet and languages like EventML can be used in combination with Ovid to build provably correct large-scale infrastructure services that comprise multiple distributed systems. These systems can be employed to prove the safety and liveness properties of different modules in Ovid, as well as the distributed systems that are transformed by Ovid. This way, large-scale infrastructure systems that are built as a combination of multiple provably correct distributed systems can be constructed by Ovid.

2.2 Implementing Evolving Distributed Systems

One approach in implementing evolving distributed systems is building reconfigurable systems. Reconfigurable distributed systems [4, 6, 12, 14] support the replacement of their sub-systems. In [3], Ajmani et al. propose automatically upgrading the software of long-lived, highly-available distributed systems gradually, supporting multi-version systems. In the infrastructure presented, a distributed system is modeled as a collection of objects. An object is an instance of a class. During an upgrade old and new versions of a class and their instances are saved by a node and both versions can be used depending on the rules of the upgrade. This way, multi-versioning and modular upgrades are supported in the object-level. In their methodology, Ajmani et al. use transform functions that reorganizes a node's persistent state from the representation required by the old instance to that required by the new instance, but these functions are limited with transforming the state of a node, whereas we transform the distributed system as a whole.

Horus [25, 17] and Ensemble [10, 24] employ a modular approach to building distributed systems, using *micro-protocols* that can be combined together to create protocols that are used between components of a distributed system. Specific guarantees required by a distributed system can be implemented by creating different combinations of micro-protocols. Each micro-protocol layer handles some small aspect of guarantees implemented by a distributed system, such as fault-tolerance, encryption, filtering, and replication. Horus and Ensemble also support on-the-fly updates [18, 19].

Prior work has used refinement mappings to prove that a lower-level specification of a distributed system correctly implements a higher-level one. In [2], Aizikowitz et al. uses refinement mappings to show that a distributed, multiple-server implementation of a service is correct if it implements the high-level, single-server specification. Our work generalizes this idea to include other types of system transformations such as sharding, batching, replication, encryption, and so on.

3 System Model

3.1 Agents

A system consists of a set $\mathcal{A}$ of *agents* $\alpha, \beta, ...$ that communicate by exchanging messages. Each agent has a unique identifier. A message is a pair $\langle$*agent identifier*, *payload*$\rangle$. A message sent by a correct agent to another

```
agent KeyValueStore :
  var map
  initially : ∀k ∈ Key : map[k] = ⊥
  transition ⟨g, p⟩ filter g ≠ ⊥:
    if p.type = PUT:
      map[p.key] := p.value
    elif p.type = GET:
      SEND ⟨p.replyAgentID, map[p.key]⟩
```

Figure 1: Pseudocode for a key-value store agent. The key-value store keeps a mapping from keys to values and maps a new value to a given key with the PUT operation and returns the value mapped to a given key with the GET operation.

```
agent Client :
  var result
  transition ⟨g, p⟩:
    if g = ⊥:
      SEND⟨ 'KVS', ⟨type : GET,
                    key : 'foo',
                    replyAgentID : 'client'⟩⟩
    else:
      result := p
```

Figure 2: Pseudocode for a client agent that requests a key mapping from the key-value store agent with a GET operation on the key 'foo'.

correct agent is eventually delivered, but multiple messages are not necessarily delivered in the order sent and there is no bound on latency.

We describe the state of an agent by a tuple $(\mathcal{ID}, \mathcal{SV}, \mathcal{ST}, \mathcal{IM}, \mathcal{PM}, \mathcal{OM}, \mathcal{WB}, \mathcal{RT})$:

- $\mathcal{ID}$: a unique identifier for the agent;
- $\mathcal{SV}$: a collection of *state variables* and their values;
- $\mathcal{TF}$: a *transition function* invoked for each input message;
- $\mathcal{IM}$: a collection of input messages that have been received;
- $\mathcal{PM}$: a subset of $\mathcal{IM}$ of messages that have been processed;
- $\mathcal{OM}$: a collection of all output messages that have been produced;
- $\mathcal{WB}$: a *wedged bit* that, when set, prevents the agent from transitioning. In particular, no more input messages can be processed, no more output messages can be produced, and the state variables are immutable;
- $\mathcal{RT}$: a routing table that maps agent identifiers to agent identifiers.

Agents that are faulty make no more transitions (*i.e.*, their wedged bit is set) and in addition stop attempting to deliver output messages. Assuming its $\mathcal{WB}$ is clear, a correct agent eventually selects a message from $\mathcal{IM}\backslash\mathcal{PM}$ (if non-empty), updates the local state using $\mathcal{TF}$, and adds the message to $\mathcal{PM}$. In addition, the transition may produce one or more messages that are added to $\mathcal{OM}$. Optionally, the transition function may specify a *filter predicate* that specifies which of the input messages are currently of interest. Such transitions are atomic. $\mathcal{IM}$ initially consists of a single message $\langle \perp, \perp \rangle$ that can be used for the agent to send some initial messages in the absence of other input. Note that a message with a particular content can only be processed once by a particular agent; applications that need to be able to exchange the same content multiple times are responsible for adding additional information such as a sequence number to distinguish the copies.

See Figure 1 for an example of an agent that implements a key-value store: a mapping from keys to values. (Only $\mathcal{SV}$ and $\mathcal{TF}$ are shown.) For example, the transition is enabled if there is a PUT request in $\mathcal{IM}\backslash\mathcal{PM}$, and the transition simply updates the map but produces no output message. The agent identifier in the input message is ignored in this case. If there is a GET request in $\mathcal{IM}\backslash\mathcal{PM}$, the transition produces a response message to the agent identifier included in the payload p. The command **SEND** $\langle g', p' \rangle$ adds message $\langle g', p' \rangle$ to $\mathcal{OM}$. In both cases the request message is added to $\mathcal{PM}$.

```
agent ShardIngressProxy :
  transition ⟨g, p⟩:
    if p ≠ ⊥ ∧ P(p.key):
      SEND ⟨ 'KVS/ShardEgressProxy1', ⟨g, p⟩⟩
    elif p ≠ ⊥ ∧ ¬P(p.key):
      SEND ⟨ 'KVS/ShardEgressProxy2', ⟨g, p⟩⟩
```

Figure 3: Pseudocode for a client-side sharding ingress proxy agent for the key-value store. The ingress proxy agent mimics `'KVS'` to the client and forwards client requests to the correct shard depending on the key.

```
agent ShardEgressProxy :
  transition ⟨g', ⟨g, p⟩⟩:
    SEND ⟨g, p⟩
```

Figure 4: Pseudocode for a server-side sharding egress proxy agent for the key-value store. The egress proxy agent mimics the client to a shard of the key-value store and simply forwards a received client request.

Figure 2 gives an example of a client that invokes a `GET` operation on the key 'foo'. The routing table of the client agent must contain an entry that maps `'KVS'` to the identifier of the key-value store agent, and similarly, the routing table of the key-value store agent must contain an entry that maps `'client'` to the identifier of the client agent.

We note that our specifications are *executable*: every state variable is instantiated at an agent and every transition is local to an agent.

3.2 Transformation

The specification of an agent α can be *transformed*— replacing it with one or more new agents in such a way that the new agents collectively implement the same functionality as the original agent from the perspective of the other, unchanged, agents. In the context of a particular transformation, we call the original agent *virtual*, and the agents that result from the transformation *physical*.

For example, consider the key-value store agent of Figure 1. We can *shard* the virtual agent by creating two physical copies of it, one responsible for all keys that satisfy some predicate $P(key)$, and the other responsible for the other keys. That is, P is a binary hashing function. To glue everything together, we add additional physical agents: a collection of *ingress proxy agents*, one for each client, and two *egress proxy agents*, one for each server.[1] An ingress proxy agent mimics the virtual agent `'KVS'` to its client, while an egress proxy agent mimics the client to a shard of the key-value store. The routing table of the client agent is modified to route messages to `'KVS'` to the ingress proxy. Figures 3 and 4 present the code for these proxies. Note that the proxy agents have no state variables. Moreover, Figure 5 illustrates their configuration as a directed graph. Every physical agent is pictured as a separate node and agents that are co-located are shown in the same rectangle representing a box. The directed edges between nodes illustrate the message traffic patterns between agents. Lastly, the dotted line is used to separate two abstraction layers from each other.

A transformation is essentially a special case of a refinement in which an *executable* specification is refined to another executable specification. To show the correctness of refinement, one must exhibit:

- a mapping of the state of the physical agents to the state of the virtual agent,
- a mapping of the transitions of the physical agents to transitions in the virtual agent, or identify those transitions as *stutter transitions* that do not change the state of the virtual agent.

In this example, a possible mapping is as follows:

[1] For this particular example, it would be possible to not use server-side egress proxies and have the client-side ingress proxies send directly to the shards. The given solution is chosen to illustrate various concepts in our system.

- $\mathcal{ID}$: the identifier of the virtual key-value store agent is unchanged and constant;
- $\mathcal{SV}$: the map of the virtual agent is the union of the two physical shards;
- $\mathcal{TF}$: the transition function of the virtual agent is also as specified;
- $\mathcal{IM}$: the set of input messages of the virtual agent is the union of the sets of input messages of all client-side proxies;
- $\mathcal{PM}$: the set of processed messages of the virtual agent is the union of the set of processed messages of the two shards;
- $\mathcal{OM}$: the set of output messages of the virtual agent is the union of the sets of output messages of the two shards;
- $\mathcal{WB}$: the wedged bit is the logical 'and' of the wedged bits of the shard agents (when both shards are wedged, the original agent can no longer transition either);
- $\mathcal{RT}$: the routing table is a constant.

In addition, the transitions of physical agents map to transitions in the virtual agent as follows:

- receiving a message in one of the $\mathcal{IM}$s of the ingress proxy agents maps to a message being added to the $\mathcal{IM}$ of the virtual agent;
- each $\mathcal{TF}$ transition in the physical shards maps to the corresponding transition in the virtual key-value store agent. In addition, adding a message to either $\mathcal{PM}$ or $\mathcal{OM}$ in one of the shards maps to the same transition in the virtual agent;
- setting the $\mathcal{WB}$ of one of the shards so that both become set causes the $\mathcal{WB}$ of the virtual agent to become set;
- clearing the $\mathcal{WB}$ of one of the shards when both were set causes the $\mathcal{WB}$ of the virtual agent to become cleared;
- any other transition in the physical agents is a stutter.

The example illustrates *layering* and *encapsulation*, common concepts in distributed systems and networking. Figure 5 shows two layers with an abstraction boundary. The top layer shows an application and its clients. The bottom layer multiplexes and demultiplexes. This is similar to multiplexing in common network stacks. For example, the EtherType field in an Ethernet header, the protocol field in an IP header, and the destination port in a TCP header all specify what the next protocol is to handle the encapsulated payload. In our system, agent identifiers fulfill that role. Even if there are multiple layers of transformation, each layer would use, uniformly, an agent identifier for demultiplexing.

The example specifically illustrates sharding, but there are many other kinds of transformations that can be applied in a similar fashion, among which:

- *State Machine Replication*: similar to sharding, this deploys multiple copies of the original agent. The proxies in this case run a replication protocol that ensures that all copies receive the same messages in the same order;
- *Primary-Backup Replication*: this can be applied to applications that keep state on a separate disk using read and write operations. In our model, such a disk is considered a separate agent. Fault-tolerance can be achieved by deploying multiple disk agents, one of which is considered primary and the others backups;
- *Load Balancing*: also similar to sharding, and particularly useful for stateless agents, a load balancing agent is an ingress proxy agent that spreads incoming messages to a collection of server agents;

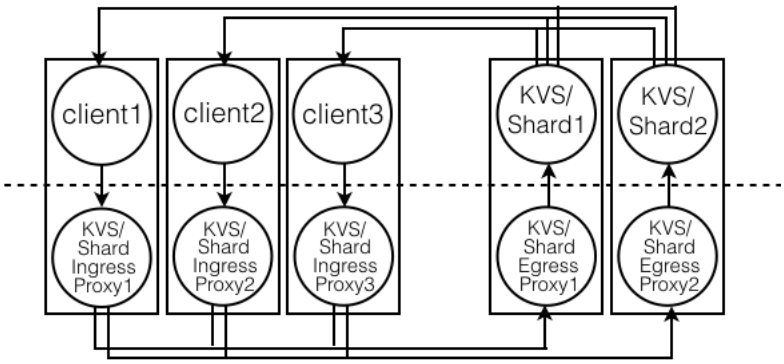


Figure 5: Configuration for the key-value store that has been transformed to be sharded two-ways.

- *Encryption, Compression, Batching, ...*: between any pair of agents, one can insert a pair of agents that encode and decode sequences of messages respectively;
- *Monitoring, Auditing*: between any pair of agents, an agent can be inserted that counts or logs the messages that flow through it.

Above we have presented transformations as refinements of individual agents. In limited form, transformations can sometimes also be applied to sets of agents. For example, a *pipeline of agents* (in which the output of one agent form the input to the next) acts essentially as a single agent, and transformations that apply to a single agent can also be applied to pipelines. Some transformations, such as Nysiad [11], apply to particular configurations of agents. For simplicity, we will focus here on transformations of individual agents only, but believe the techniques can be generalized more broadly.

3.3 Agent Identifiers and Transformation Trees

Every agent in the system has a unique identifier. The agents that result from transformation have identifiers that are based on the original agent's identifier, by adding new identifiers in a `'path name'` style. Thus an agent with identifier `'X/Y/Z'` is part of the implementation of agent `'X/Y'`, which itself is part of the implementation of agent `'X'`, which is a top level specification. In our running example, assume the identifier of the original key-value store is `'KVS'`. Then we can call its shards `'KVS/Shard1'` and `'KVS/Shard2'`. We can call the server proxies `'KVS/ShardEgressProxy1'` and `'KVS/ShardEgressProxy2'` respectively, and we can call the client proxies `'KVS/ShardIngressProxy1'`, `'KVS/ShardIngressProxy2'`, ...

The client agent in this example still sends messages to agent identifier `'KVS'`, but due to transformation the original `'KVS'` agent no longer exists physically. The client's routing table maps agent identifier `'KVS'` to `'KVS/ShardIngressProxy`*X*`'` for some *X*. Agent `'KVS/ShardIngressProxy`*X*`'` encapsulates the messages it received from its client and sends it to agent identifier `'KVS/ShardEgressProxy1'` or `'KVS/ShardEgressProxy2'` depending on the hash function. Assuming those proxy agents have not been transformed themselves, there is again a one-to-one mapping to from agent identifiers to corresponding agent identifiers. Each egress proxy ends up sending to agent identifier `'KVS'`. Agent identifier `'KVS'` is mapped to agent `'KVS/Shard1'` at agent `'KVS/ShardEgressProxy1'` and to agent `'KVS/Shard2'` at agent `'KVS/ShardEgressProxy2'`. Note that if identifier *X* in a routing table is mapped to an identifier *Y*, it is always the case that *X* is a prefix of *Y* (and is identical to *Y* in the case the agent has not been refined).

Given the original specification and the transformations that have been applied, it is always possible to determine the destination agent for a message sent by a particular source agent to a particular agent identifier.

```
agent KeyValueStore :
  var : map, counter
  initially : ∀k ∈ Key : map[k] = ⊥ ∧ counter = 0

  transition ⟨g, ⟨c, p⟩⟩ filter c = counter:
    if p.type = PUT:
      map[p.key] := p.value
    elif p.type = GET:
      SEND ⟨p.replyAgentID, map[p.key]⟩
    counter := counter + 1
```

Figure 6: Pseudocode for a determistic key-value store agent that handles requests in order using a counter.

```
agent Numberer :
  var counter
  initially : counter = 0

  transition ⟨g, p⟩ filter g ≠ ⊥:
    SEND ⟨g, ⟨counter, p⟩⟩
    counter := counter + 1
```

Figure 7: Pseudocode for the numbering agent that numbers every message before it forwards it to its destination.

This even works if agents are created dynamically. For example, if a new client `'client2'` is added to our example, and sends a message to agent identifier `'KVS'`, we can determine that agent `'KVS'` has been transformed and thus a new client-side ingress proxy agent has to be created, and appropriate agent identifier to agent identifier mappings must be added. The client's request can now be delivered to the appropriate shard through the ingress proxy agent. The shard sends the response to agent identifier `'client2'`. In this case the new client itself has not been transformed, and so the mapping for agent identifier `'client2'` at the shard can be set up to point directly to agent `'client2'`. Should agent `'client2'` itself have been transformed, then upon the KVS shard sending to agent identifier `'client2'` the appropriate proxy agents can be instantiated on the server-side dynamically as well.

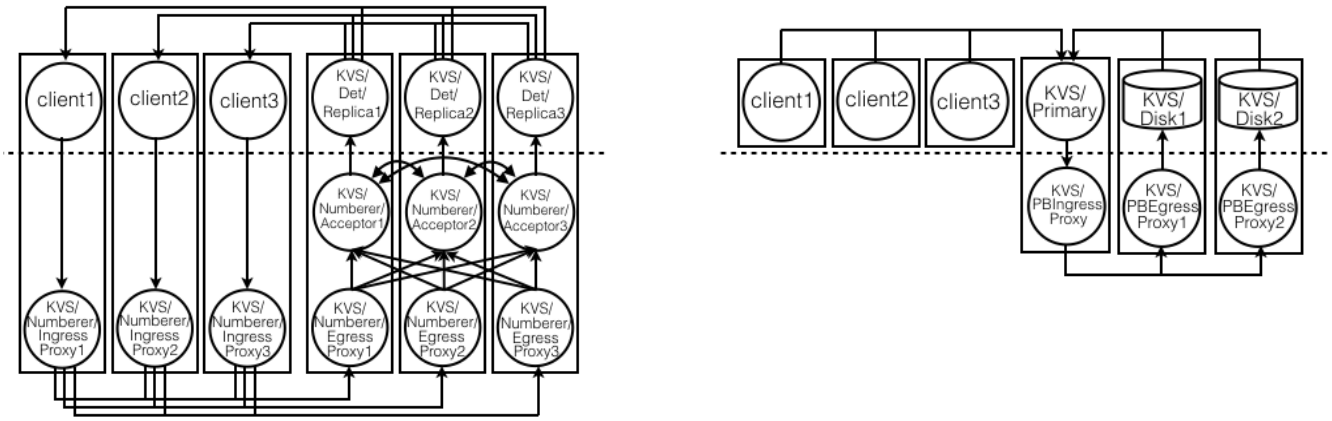
We represent the specification of a system and its transformation in a *Logical Agent Transformation Tree* (LATT). A LATT is a directed tree of agents. The root of this tree is a virtual agent that we call the *System Agent*. The "children" of this root are the agents before transformation. Each transformation of an agent (or set of agents) then results in a collection of children for the corresponding node.

This technique presents, to the best of our knowledge, the first general solution to composing transformed agents, such as a replicated client interacting with a replicated server. While certain special case solutions exist (for example, the Replicated Remote Procedure Call in the Circus system [27]), it has not been clear how, say, a client replicated using the replicated state machine approach would interact with a server that is sharded, with each shard chain replicated. At the same time, another client may have been transformed in another fashion, and also has to be able to communicate with the same server. The resulting mesh of proxies for the various transformations is complex and difficult to implement correctly "by hand." The Ovid framework makes composition of transformed agents relatively easy.

3.4 Ordering

In our system, messaging is reliable but not ordered. The reason is clear from the running example: even if input and output were ordered, that ordering is lost (and unnecessary) if the key-value store is sharded. Maintaining the output ordering would require additional complexity. We did not want to be tied to maintaining a property that would be hard to maintain end-to-end in the face of transformations.

However, for certain transformations it is convenient if, or even necessary that, input and output are ordered. A canonical example is state machine replication, which only works if each replica processes its input in the same order. Agents that need such ordering should require, for example, that messages are numbered or tagged with some kind of ordering dependencies. In case there cannot be two different messages with the same number,



(a) KVS transformed with state machine replication.

(b) KVS transformed with primary-backup replication.

Figure 8: The key-value store can be transformed to tolerate failures using different replication techniques.

an agent will make deterministic transitions. For example, Figure 6 shows the code for a deterministic version of the key-value store that can be replicated using state machine replication. If unreplicated, messages can be numbered by placing a *numbering agent* (Figure 7) in front of it that numbers messages from clients. When replicated with a protocol such a Paxos, Paxos essentially acts as a fault-tolerant numbering agent.

We can consider the pair of agents that we have created a refinement of the original `'KVS'` agent, and identify them as `'KVS/Deterministic'` and `'KVS/Numberer'`. By having clients map `'KVS'` to `'KVS/Numberer'`, their messages to `'KVS'` are automatically routed to the numbering agent.

4 Transformation Examples

Agents can be transformed in various ways and combined with other agents in various ways, resulting in complex but functional systems. In this section we go through some examples of how agents can be transformed to obtain fault-tolerance, scalability, and security.

To visualize transformations and the resulting configuration of a system, we use graphs with directed edges. The edges represent the message traffic patterns for a particular client and a particular server. Messages emanating from other clients, which themselves may be transformed in other ways, may not follow the same paths. The dotted lines are used to separate different abstraction layers from each other.

4.0.1 State Machine Replication

State Machine Replication is commonly used to change a non-fault tolerant system to a fault-tolerant one by creating replicas that maintain the application state. To ensure consistency between the replicas, they are updated using the same ordered inputs. As alluded to before, we support state machine replication by two separate transformations. First, we transform a deterministic agent simply by generating multiple copies of it. Second, we refine the numbering agent and replace it with a state machine replication protocol such as Paxos.

We start with the deterministic key-value store agent in Figure 6 and create copies of it. We can call its replicas `'KVS/Deterministic/Replica1'`, `'KVS/Deterministic/Replica2'`, and so on, but note that they each run identical code to the virtual `'KVS/Deterministic'` agent. Next, we take the `'KVS/Numberer'` agent, and replace it with a fault-tolerant version. For example, in order to tolerate f acceptor failures using the Paxos protocol, we may deploy `'KVS/Numberer/AcceptorX'` for $X \in [0, 2f]$.

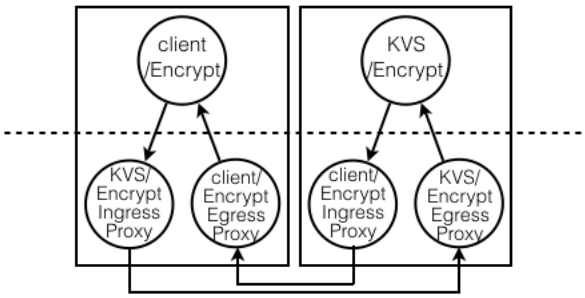


Figure 9: The key-value store can be transformed to accept encrypted traffic from clients by adding an ingress proxy on the client-side that encrypts client messages before they are sent and an egress proxy on the server-side that decrypt a message using the key shared between proxies. The reverse traffic is encrypted by transforming the clients in the same fashion.

As before, we will also deploy a client-side ingress proxy `'KVS/Numberer/IngressProxy`*C*`'` for each client *C*. Figure 8a shows the resulting system.

The technique can be combined with sharding. For example, we can first shard the key-value store and then replicate each of the shards individually (or a subset of the shards if we so desired for some reason). Alternatively, we can replicate the key-value store first, and then shard the replicas. While the latter is less commonly applied in practice, it makes sense in a setting where there are multiple heterogeneous data centers. One can place a replica in each datacenter, but use a different number of shards in each datacenter. And if one so desired, one could shard the shards, replicate the replicas, and so on.

4.0.2 Primary-Backup Replication

In primary-backup replication, a primary front-end handles requests from clients and saves the application state on multiple backup disks before replying back to the clients. When the primary fails, another primary is started with the state saved in one of the backup disks and continues handling client requests. To transform the key-value store to a primary-backup system, our example needs a new level of indirection, where the front-end KVS agent that handles the client requests itself is the client of a back-end disk that stores the application state. This new layering is necessary to introduce multiple back-up disks.

Accordingly, the primary-backup transformation of the KVS agent creates new proxies that enable the primary KVS agent, denoted as `'KVS/Primary'` to refer to backup disks as `'disk'` by having an entry in its routing table that maps `'disk'` to `'KVS/PBIngressProxy'`. Figure 8b shows this transformation. The `'KVS/PBIngressProxy'` can then send the update to be stored on disk to the disk proxies, denoted `'KVS/PBEgressProxy`*X*`'`, which in turn store the application state on their local back-end disk `'KVS/Disk`*X*`'`. This way, the key-value store is transformed with primary-backup replication without requiring the clients and the KVStore agents to change.

Similar to state machine replication, the fault-tolerance level of the transformed KVStore agent depends directly on the guarantees provided by primary-backup replication and the number of back-end disks that are created. As a result, because primary-backup replication can tolerate *f* failures with *f+1* back-end disks, the transformed KVStore in Figure 8b can tolerate the failure of one of the back-end disks.

4.0.3 Encryption, Compression, and Batching

One type of transformation that is supported by Ovid is adding an encoder and decoder between any two agents in a system, in effect processing streams of messages between these agents in a desired way. This transformation can be used to transform any existing system to support encryption, batching, and compression.

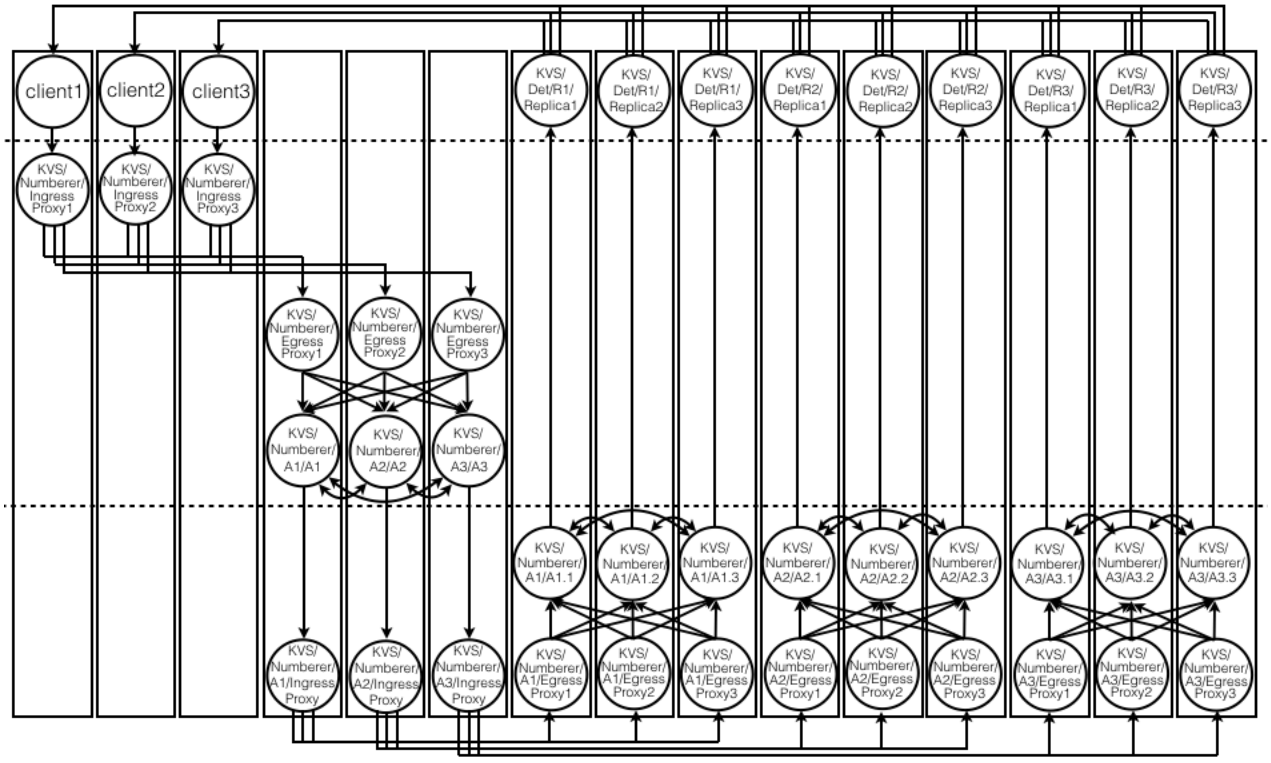


Figure 10: A crash fault-tolerant key-value store can be made to tolerate Byzantine failures by applying the Nysiad transformation, which replaces the replicas of the key-value store agent.

The encryption transformation is an example of these types of transformations. Encryption can be used to implement secure distributed systems by making traffic between different components unreadable to unauthorized components. To implement encryption in a distributed system, the requests coming from different clients can be encrypted and decrypted using unique encryption keys for clients. The transformation for encryption in Ovid follows this model and creates secure channels between different agents by forwarding messages to encryption and decryption proxies that are created during the transformation.

Figure 9 shows how the key-value store agent is transformed to support secure channels between the key-value store and the clients. Note that, in this example the traffic from the key-value store to client is encrypted, as well as the traffic from the client to the key-value store. When the client sends a message to the key-value store, the message is routed to `'KVS/EncryptIngressProxy'`, where it is encrypted and sent to `'KVS/EncryptEgressProxy'`. The egress proxy decrypts the message using the key shared between the proxies and forwards it to the key-value store to be handled. Virtually, `'KVS/EncryptIngressProxy'` and `'KVS/EncryptEgressProxy'` are separate entities than the client and the key-value store, but physically `'KVS/EncryptIngressProxy'` is co-located with the client, and `'KVS/EncryptEgressProxy'` is co-located with the key-value store shown as `'KVS/Encrypt'`. After the request is handled by the key-value store and a reply is sent back to the client, the reply follows a route symmetrical to the one from the client to the key-value store, since the client is transformed in the same fashion.

Batching and compression follow the same method: To achieve better performance in the face of changing load, multiple requests from a client can be batched together or compressed by an encoding agent and sent through the network as a single request. Then on the server side, these requests can be unbatched or decompressed accordingly and handled.

4.0.4 Crash Fault-Tolerance to Byzantine Fault-Tolerance with Nysiad

Many evolving distributed systems need to be transformed multiple times to change the assumptions they make about their environment or to change the guarantees offered by the system. For instance, a key-value store that has been transformed to handle only crash failures would not be able to handle bit errors in a large datacenter deployment. Ovid can solve this problem by transforming the crash fault-tolerant key-value store to tolerate Byzantine failures using the Nysiad [11] transformation. Figure 10 shows the transformation of the crash fault-tolerant key-value store of Figure 8a to a Byzantine fault-tolerant key-value store. This transformation replaces the replicas of the deterministic key-value store agent, namely `'KVS/Deterministic/Replica1'`, `'KVS/Deterministic/Replica2'`, and `'KVS/Deterministic/Replica3'`. A Byzantine failure is now masked as if it were a crash failure of a deterministic key-value store agent replica.

5 Evolution

A deployed system evolves. There can be many sources of such evolution, including:

- client agents come and go;
- in the face of changing workloads, applications may be elastic, and server agents may come an go as well;
- new functionality may be deployed in the form of new server agents;
- existing agents may be enhanced with additional functionality;
- bug fixes also lead to new versions of agents;
- the performance or fault tolerance of agents may be improved by applying transformations to them;
- a previously transformed agent may be "untransformed" to save cost;
- new transformations may be developed, possibly replacing old ones.

Our system supports such evolution, even as it happens "on-the-fly." As evolution happens, various correctness guarantees must be maintained. Instrumental in this are agents' wedged bits: we have the ability to temporarily halt an agent, even a virtual one. While halted, we can extract its state, transfer it to a new agent, update routing tables, and restart by unwedging the new agent. It is convenient to think of *versions* of an agent as different refinements of a higher level specification. Similarly, an old version has to be wedged, and state has to be transferred from an old version of an agent to a new one. The identifiers of such new agents should be different from old ones. We do this by incorporating version numbers into the identifiers of agents. For example, `'KVS:v2/IngressProxy:v3:2'` would name the second incarnation of version 3 of an ingress proxy agent, and this refines version 2 of the virtual KVS agent.

Obtaining the state of a wedged physical agent is trivial, and starting a physical agent with a particular state is also a simple operation. However, a wedged virtual agent may have its state distributed among various physical agents and thus obtaining or initializing the state is not straightforward. And even if a virtual agent is wedged, not all of its physical agents are necessarily wedged. For example, in the case of the sharded key-value store, the virtual agent is wedged if all shards are wedged, but it does not require that the proxies are wedged. Note also that there may be multiple levels of transformation: in the context of one transformation an agent may be physical, but in the context of another it may be virtual. For example, the shards in the last example may be replicated.

We require that each transformation provide a function *getState* that obtains the state of a virtual agent given the states of its physical agents. It is not necessarily the case that all physical agents are needed. For example, in

the case of a replicated agent, some physical agents may be unavailable and it should not be necessary to obtain the state of the virtual agent. Given the state of a virtual agent, a transformation also needs to provide a function that instantiates the physical agents and initializes their state.

While this solution works, it can lead to a significant performance hiccup. We are designing a new approach where the new physical agents can be started without an initial state before the virtual agent is even wedged. Then the virtual agent is wedged (by wedging the physical agents of its original transformation). This is a low-cost operation. Upon completion, the new physical agents are unwedged and "demand-load" state as needed from the old physical agents. Once all state has been transferred, the old physical agents can be shut down and garbage collected. This is most easily accomplished if the virtual agent has state that is easily divided into smaller, functional pieces. For example, in the case of a key-value store, each key-value pair can be separately transferred when needed.

6 Running Agents

6.1 Boxing

An agent is a logical location, but it is not necessary to run each agent on a separate host. Figure 5 illustrates *boxing*: co-locating agents in some way, such as on the same machine or even within a single process.

A *box* is an execution environment or virtual machine for agents. For each agent, the box keeps track of the agent's attributes and runs transitions for messages that have arrived. In addition, each box runs a *Box Manager Agent* that supports management operations such as starting a new agent or updating an agent's routing table.

Boxes also implement the reliable transport of messages between agents. A box is responsible for making sure that the set of output messages of an agent running on the box is transferred to the sets of input messages of the destination agents. For each destination agent, this assumes there is an entry for the message's agent identifier in the source agent's routing table, and the box also needs to know how to map the destination agent identifier to one or more network addresses of the box that runs the destinating agent. A box maintains the Box Routing Table (BRT) for this purpose. For now, we assume that each BRT is filled with the necessary information.

The box transmits the message to the destination box. Upon arrival, the destination box determines if the destination agent exists. If so, the box adds the message to the end of the agent's input messages, and returns an acknowledgment to the source box. The source box keeps retransmitting (without timeout) the message until it has received an acknowledgment. This process must restart if there is a reconfiguration of the destination agent. An output message can be discarded only if it is known that the message has been processed by the destination agent—being received is not a safe condition.

6.2 Placement

While resulting in more efficient usage of resources, boxing usually leads to a form *fate sharing*: a problem with a box such as a crash is often experienced by all agents running inside the box. Fate sharing makes sense for agents and its proxies, but it would be unwise to run replicas of the same agent within the same box.

We use *agent placement annotations* to indicate placement desirables and constraints. For example, agent α can have a set of annotations of the following form:

- α.SameBoxPreferred(β): β ideally is placed on the same box as α;
- α.SameBoxImpossible(β): β cannot run on the same box as α.

Placing agents in boxes can be automatically performed based on such annotations using a constraint optimizer. Placement is final: we do not currently consider support for *live migration* of agents between boxes. It is,

however, possible to start a new agent in another box, migrate the state from an old agent, and update the routing tables of other agents.

6.3 Controller Agent

So far we have glossed over several administrative tasks, including:

- What boxes are there, and what are their network addresses?
- What agents are there, and in which boxes are they running?
- How do agents get started in the first place?
- How do boxes know which agents run where?

As in other software-defined architectures, we deploy a logically centralized controller for administration. The controller itself is just another agent, and has identifier "controller". The controller agent itself can be refined by replication, sharding and so on. For scale, the agent may also be hierarchically structured. But for now we will focus on the high-level specification of a controller agent before transformation. In other words, for simplicity we assume that the controller agent is physically centralized and runs on a specific box.

As a starting point, the controller is configured with the LATT (Logical Agent Transformation Tree), as well as the identifiers of the box managers on those boxes. The BRT (Box Routing Table) of the box in which the controller runs is configured with the network addresses of the box managers.

First, the controller sends a message to each box manager imparting the network addresses of the box in which the controller agent runs. Upon receipt, the box manager adds a mapping for the controller agent to its BRT. (If the controller had been transformed, the controller would need to send additional information to each box manager and possibly instantiate proxy agents so the box manager can communicate with the controller.)

Using the agent placement annotations, the controller can now instantiate the agents of the LATT. This may fail if there are not enough boxes to satisfy the constraints specified by the annotations. Instantiation is accomplished by the controller sending requests to the various box managers.

Initially, the agents' routing tables start out containing only the "controller" mapping. When an agent sends a message to a particular agent identifier, there is an "agent identifier miss" event. On such an event, the agent ends up implicitly sending a request to the controller asking it to resolve the agent identifier to agent identifier binding. The controller uses the LATT to determine this binding and responds to the client with the destination agent identifier. The client then (again, implicitly) adds the mapping to its routing table.

Next, the box tries to deliver the message to the destination agent. To do this, the box looks up the destination agent identifier in its BRT, and may experience a "BRT miss". In this case, the box sends a request to the controller agent asking to resolve that binding as well. The destination agent may be within the same box as the source agent, but this can only be learned from the controller. One may think of routing tables as caches for the routes that the controller decides.

7 Conclusion

Ovid is an agent-based software-defined distributed systems platform that supports the complexity involved in building and maintaining large-scale distributed systems that have to support consistency and organic evolution. Starting from a simple specification, transformations such as replication and sharding can be applied to satisfy performance and reliability objectives. Refinement mappings allow reasoning about correctness, and also support capturing state from distributed components that have to be updated or replaced. A logically centralized controller simplifies the management of the entire system and allows sophisticated placement policies to be implemented.

We currently have a prototype implementation of Ovid that supports agents and transformations written in Python. Next, we want to develop tools to verify performance and reliability objectives of a deployment. We then plan to do evaluations of various large-scale distributed systems developed using Ovid.

Acknowledgments

The authors are supported in part by AFOSR grants FA2386-12-1-3008, F9550-06-0019, by the AFOSR MURI Science of Cyber Security: Modeling, Composition, and Measurements as AFOSR grant FA9550-11-1-0137, by NSF grants CNS-1601879, 0430161, 0964409, 1040689, 1047540, 1518779, 1561209, and CCF-0424422 (TRUST), by ONR grants N00014-01-1-0968 and N00014-09-1-0652, by DARPA grants FA8750-10-2-0238 and FA8750-11-2-0256, by MDCN/iAd grant 54083, and by grants from Microsoft Corporation, Infosys, Google, Facebook Inc., and Amazon.com.

References

[1] Hussam Abu-Libdeh, Robbert van Renesse, and Ymir Vigfusson. Leveraging sharding in the design of scalable replication protocols. In *Proceedings of the Symposium on Cloud Computing*, SoCC '13, Farmington, PA, USA, October 2013.

[2] Jacob I. Aizikowitz. *Designing Distributed Services Using Refinement Mappings*. PhD thesis, Cornell University, Ithaca, NY, USA, August 1989. Also available as technical report TR 89-1040.

[3] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 452–476, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] Christophe Bidan, Valrie Issarny, Titos Saridakis, and Apostolos Zarras. A dynamic reconfiguration service for corba. In *4th International Conference on Distributed Computing Systems*, ICCDS'98, pages 35–42. IEEE Computer Society Press, 1998.

[5] Ken Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.

[6] Toby Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT-LCSTR-303, MIT, 1983.

[7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. ACM.

[8] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[9] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '15. ACM, October 2015.

[10] Mark Garland Hayden. *The Ensemble System*. PhD thesis, Cornell University, Ithaca, NY, USA, 1998. AAI9818467.

[11] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, pages 175–188, Berkeley, CA, USA, 2008. USENIX Association.

[12] Christine R. Hofmeister and James M. Purtilo. A framework for dynamic reconfiguration of distributed programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, ICDCS 1991, pages 560–571, 1991.

[13] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[14] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[15] Leslie Lamport. Specifying concurrent program modules. *Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[16] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, Germany, 2010. Springer-Verlag.

[17] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating System Principles*, Kiawah Island Resort, SC, USA, December 1999.

[18] Xiaoming Liu and Robbert van Renesse. Fast protocol transition in a distributed environment. In *19th ACM Conference on Principles of Distributed Computing (PODC 2000)*, Portland, OR, USA, July 2000.

[19] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta properties. In *International Workshop on Applied Reliable Group Communication at the International Conference on Distributed Computing Systems*, ICDCS 2011, Phoenix, AZ, USA, April 2001.

[20] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: Ii. timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[22] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using eventml. *ECEASST*, 72, 2015.

[23] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Marck Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 395–406, Washington, DC, USA, 2014. IEEE Computer Society.

[24] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Software Practice and Experience*, 28(9):963–979, August 1998.

[25] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[26] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed system. In *Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI 2015, Portland, OR, USA, June 2015.

[27] J.C.P. Woodcock and Carroll Morgan. Refinement of state-based concurrent systems. In *VDM '90 VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 340–351. Springer Berlin Heidelberg, 1990.

[28] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

Geo-Replication: Fast If Possible, Consistent If Necessary*

Valter Balegas[1], Cheng Li[2], Mahsa Najafzadeh[4], Daniel Porto[3], Allen Clement[2,5], Srgio Duarte[1], Carla Ferreira[1], Johannes Gehrke[6], João Leitão[1], Nuno Preguia[1], Rodrigo Rodrigues[3], Marc Shapiro[4], Viktor Vafeiadis[2]

[1]NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa [2]Max Planck Institute for Software Systems (MPI-SWS)

[3]INESC-ID / IST, University of Lisbon [4]Inria & Sorbonne Universits, UPMC Univ Paris 06, LIP6

[5]Currently at Google [6]Microsoft and Cornell University

Abstract

Geo-replicated storage systems are at the core of current Internet services. Unfortunately, there exists a fundamental tension between consistency and performance for offering scalable geo-replication. Weakening consistency semantics leads to less coordination and consequently a good user experience, but it may introduce anomalies such as state divergence and invariant violation. In contrast, maintaining stronger consistency precludes anomalies but requires more coordination. This paper discusses two main contributions to address this tension. First, RedBlue Consistency enables blue operations to be fast (and weakly consistent) while the remaining red operations are strongly consistent (and slow). We identify sufficient conditions for determining when operations can be blue or must be red. Second, Explicit Consistency further increases the space of operations that can be fast by restricting the concurrent execution of only the operations that can break application-defined invariants. We further show how to allow operations to complete locally in the common case, by relying on a reservation system that moves coordination off the critical path of operation execution.

1 Introduction

A geo-replicated system maintains copies of the service state across geographically dispersed locations. Geo-replication is not only employed today by virtually all the providers of major Internet services, who typically manage several data centers spread across the globe, but is also accessible to anyone outsourcing their computational needs to cloud providers, since cloud services allow computations or VMs to be instantiated in different data centers.

There are two main reasons for deploying geo-replicated systems. The first reason is disaster tolerance, i.e., the ability to tolerate the unplanned outage of an entire data center, due to catastrophic events such as natural disasters [1]. The second reason is to reduce the latency between the users and the machines that provide the service. The importance of this aspect is demonstrated by several studies that point out an inverse correlation between response times and user satisfaction for important Internet services such as search [30].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Student author are followed by faculty author names, both in alphabetical order. Cheng Li and Valter Balegas are the lead authors of the work.

However, there is a fundamental tension between latency and consistency: intuitively, ensuring strong consistency requires coordination between replicas before returning a reply to the user, while, alternatively, a fast response can be given without replica coordination, but only ensuring weak consistency guarantees. This tension has led the providers of global-scale Internet services to choose, for some parts of their services, storage systems offering weak consistency guarantees such as eventual consistency [13], and, for other components, systems with strong consistency such as serializability [10].

This paper revisits in a unified way two of our recent results in trying to achieve a balance between performance and consistency, by devising methods to build geo-replicated systems that introduce a small amount of coordination between replicas to achieve the desired semantics, i.e., systems that are fast when possible and consistent when necessary [21, 7]. In the first result, we improve the performance of geo-replicated systems by (1) allowing different operations to execute in either a weakly consistent (fast) or strongly consistent (slow) manner; and (2) identifying a set of principles for making safe use and increasing the space of fast operations. In the second result, we further increase the space of operations that can execute fast by (1) identifying the operations that can break application invariants when executing concurrently; and (2) deploying concurrency control mechanisms that remove coordination from the critical path of operation execution, while preserving invariants.

We start the presentation by laying out our terminology and system model in Section 2. We present an initial approach based on a coarse-grained classification into strong and weak consistency in Section 3. One key aspect of this approach is operation commutativity, and we explain how to achieve it using CRDTs in Section 4. Then we present an approach that makes use of fine-grained coordination between pairs of operations in Section 5. We discuss related work in section 6 and conclude in Section 7.

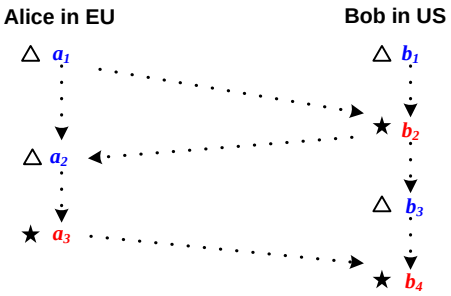
2 System model

Our system model is that of a fully replicated distributed system, where replicas are located in different data centers. Each replica follows a deterministic state machine: there is a set of operations $\mathcal{U}$, which manipulate a set of reachable system states $\mathcal{S}$. Each operation u is initially submitted at a given replica (preferably in the closest data center), which we call the *origin replica* of u. When the remaining replicas receive a request to replicate this operation, they will apply the operation against their local state.

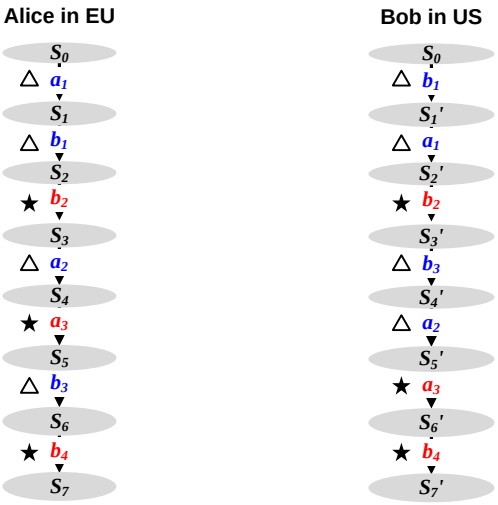
Throughout our explanation we will highlight two important properties that the replicated system should obey. First, there is the **state convergence** property, which says that all the sites that have executed the same set of operations against the same initial state are in the same final state. This is important to prevent a situation where the system quiesces (no more updates are received) and read-only queries return different results depending on which sites the users are connected to. The second property is to **preserve application-specific invariants**, which comprise a specification for the behavior of the system. To define these, we introduce the primitive *valid*(S) to be *true* if state S obeys these invariants and *false* otherwise.

3 Mixing consistency levels in RedBlue consistency

In this section, we present a hybrid consistency model called RedBlue consistency, where weakly consistent operations, labeled blue, can be executed at a single replica and propagated in the background, with mostly no coordination with concurrent actions at other replicas, while others, labeled red, require a stronger consistency level and thus require cross-replica coordination. RedBlue consistency is one of several systems that propose labeling operations according to their consistency levels [18, 33, 21, 36], but improves on these systems by offering a precise method for labeling operations.



(a) RedBlue order O of operations



(b) Serializations of O

Figure 1: RedBlue order and serializations for a system spanning two sites. Operations marked with ★ are red, and operations marked with △ are blue. Dotted arrows in (a) indicate the partial ordering of operations.

3.1 Defining RedBlue consistency

RedBlue consistency relies on three components: (1) a partitioning of operations into weakly consistent blue operations whose order of execution can vary from site to site, and red operations that must be executed in the same order at all sites, (2) a RedBlue order, which defines a partial order of operations where red operations have to be ordered with respect to each other, and (3) a set of site-specific serializations (i.e., total orders) in which the operations are locally applied. More precisely:

Definition 1 (RedBlue consistency): A replicated system is *RedBlue consistent* if each site i applies operations according to a linear extension of a RedBlue order O of the operations that were invoked, where O is a partial order among those operations with the requirement that red operations are totally ordered in O.

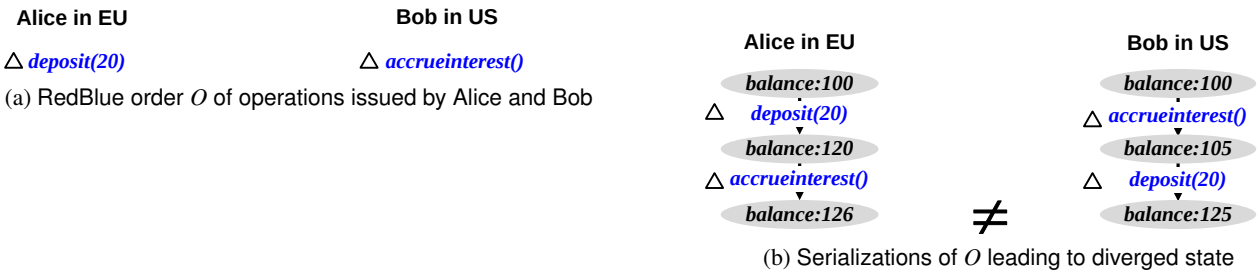
Figure 1 shows a RedBlue order and two serializations, i.e., the linear extensions of that order in which operations are applied at two different sites. In systems where every operation is labeled red, RedBlue consistency is equivalent to serializability [10]; in systems where every operation is labeled blue, RedBlue consistency becomes a form of causal consistency [37, 23, 25], since the partial order conveys the necessary causality between operations.

When applying RedBlue consistency to an application, we would like to label all operations blue to obtain best performance. However, this could lead to state divergence and invariant violation, when operations are not commutative. We describe a set of sufficient conditions to guide the classification of operations in order to safely use weak consistency when possible.

3.2 Ensuring state convergence

In the context of RedBlue consistency, we can formalize state convergence as follows:

Definition 2 (State convergence): A RedBlue consistent system is state convergent if all serializations of the underlying RedBlue order O reach the same state S w.r.t. any initial state S_0.



(a) RedBlue order O of operations issued by Alice and Bob

(b) Serializations of O leading to diverged state

Figure 2: A RedBlue consistent account with initial balance of 100 and final diverged state

.

To find a correct labeling for maintaining state convergence while providing low latency access, we describe a simple banking example, in which users may share an account that is modified via three operations, namely `deposit`, `withdraw` and `accrueinterest`[1]. Keeping in mind that one of the goals of RedBlue consistency is to make the target service as fast as possible, we tentatively label all these operations blue. According to this labeling result, we construct a RedBlue order of deposits and interest accruals made by two users Alice and Bob and two possible serializations applied at both branches of the bank, as shown in Figure 2. This example shows that the labeling of these operations as described is not state convergent. This is because RedBlue consistency allows the two sites to execute blue operations in a different order, but two of the blue operations in the example are non-commutative, namely `deposit` and `accrueinterest`. To prevent this situation, a sufficient condition to guarantee state convergence in a system supporting RedBlue consistency is that every blue operation commutes with all other operations, blue or red.

However, when applying this condition to the banking example, it implies that we need to label all three operations red (`deposit`, `withdraw` and `accrueinterest`). This is equivalent to running the system under serializability, which requires coordination across replicas for executing all these operations. To address the problem that it is difficult to find operations that commute with all other operations in the system, we observe that, in many cases, while operations may not be commutative, we can make the changes they induce on the system state to commute. In the banking example, we can engineer `accrueinterest` commute with the remaining two operations by first computing the amount of interested accrued at the primary replica and then treating that value as a deposit.

To exploit this observation and increase operation commutativity, we propose a change to our original system model, where we split each original application operation u into two components: a *generator operation* g_u with no side-effects, which is executed only at the primary site against some system state S and produces a *shadow operation* $h_u(S)$, which is executed at every site (including the primary site). The generator operation decides which state transitions should be made while the shadow operation applies the transitions in a state-independent manner.

3.3 Preserving invariants

Although the concept of shadow operation helps produce more commutative operations, labeling too many shadow operations as blue may introduce the problem of breaking application invariants. In the banking example, assuming that the shared bank account has an initial balance of 100, if both Alice and Bob withdraw 70 and 60 respectively, the final balance would be -30. This violates the invariant that a bank balance should never be negative. To determine which shadow operations can be safely labeled blue, we begin by defining that a shadow operation is invariant safe if, when applied to a valid state, it always transitions the system into another valid state. This allows us to define the following sufficient condition: a RedBlue consistent system preserves

[1] `accrueinterest` computes a new balance by multiplying the old balance value and (1 + `interest rate`).

invariants (meaning that all its sites are always in valid states) if all shadow operations that are not invariant safe are labeled red (i.e., strongly consistent).

3.4 What can be blue? What must be red?

In summary, the two conditions above lead to the following procedure for deciding which shadow operations can be blue or must be red if a RedBlue consistent system is to provide both state convergence and invariant preservation:

1. For any pair of non-commutative shadow operations h_u and h_v, label both h_u and h_v red.
2. For any shadow operation h_u that is not invariant safe, label h_u red.
3. Label all remaining shadow operations blue.

4 State convergence

In the previous section we discussed how RedBlue consistency achieves state convergence by relying on shadow operations that commute with each other. With this approach, defining a new operation also implies writing one or more commutative shadow operations, each of which corresponds to a distinct side effect. The major challenge of doing this manual work is that, in an application with a large number of operations, this process may be complex and error-prone.

We now discuss an alternative principled approach to create commutative operations by design. Our approach builds on conflict-free replicated data types (CRDTs) [31], which are specially-designed data structures that can be replicated and modified concurrently, and include mechanisms to merge concurrent updates in a deterministic way. Application operations consist of updates to these elementary data types, thus guaranteeing state convergence.

4.1 CRDTs

A CRDT is a data type that can be replicated at multiple replicas. As such, it defines an interface with a set of operations to read and to modify its state. A CRDT replica can be modified by locally executing an update operation. When different replicas of the same object are modified concurrently, they temporarily diverge. CRDTs have built-in support for achieving strong eventual consistency [31], in which all replicas will eventually reach the same (equivalent) state after applying the same set of updates, without relying on a distributed conflict arbitration process.

Two main flavors of CRDTs have been studied in the literature: operation-based CRDTs and state-based CRDTs. For each of these, sufficient conditions for achieving strong eventual consistency have been established.

In operation-based CRDTs (or commutative replicated data types), updates are propagated by broadcasting operations to every replica in causal order. Interestingly, this proposal matches the operation execution decomposition presented in RedBlue consistency (Section 3), where operations are divided in two components, a *generator* operation that executes in the local replica, has no side effect and produces a *shadow* operation, which is propagated and executed in all replicas. The two types of operations are analogous to *prepare* and *downstream* operations in the context of operation-based CRDTs, respectively, with the main difference that shadow operations are assigned a consistency level in RedBlue consistency. Similarly to the consequence of commutative shadow operations, the replicas of an operation-based CRDT converge to the same state after executing the same set of updates (in any order that respects causality) if the execution of any two concurrent downstream operations commutes [31].

SQL type	CRDT	Description
FIELD*	LWW	Use last-writer-wins to solve concurrent updates
	NUMDELTA	Add a delta to the numeric value
TABLE	AOSET, UOSET, AUSET, ARSET	Sets with restricted operations (add, update, and/or remove). Conflicting operations are logically executed by timestamp order.

Table 1: Commutative replicated data types (CRDTs) for relational data. * FIELD covers primitive types such as integer, float, double, datetime and string.

A state-based CRDT (or convergent replicated data type) defines, in addition to the operations to read and update its state, an operation to merge the state of two replicas. Replicas synchronize by exchanging the full replica states: when a new state is received, the new updates are incorporated in the local replica by executing the merge function. It has been shown that the replicas of a state-based CRDT converge to the same state after all replicas synchronize (directly or indirectly) if: (1) all the possible states of an object are partially ordered, forming a join-semilattice; (2) the merge operation between two states is the semilattice join; and (3) an update monotonically increases the state according to the defined partial order [31].

4.2 Examples

CRDTs have been used in a number of research systems, such as Walter [35] and SwiftCloud [39], and commercial systems, such as Riak [2] and SoundCloud [3]. These systems include CRDTs that implement several data types, such as registers, counters, sets, maps, and flags. For each such data type, it is possible to define and implement different semantics to handle concurrent updates, leading to different CRDTs. These semantics define which is the final state of a CRDT when concurrent updates occur. For example, for sets, it is possible to define an *add-wins* semantics, where, in the presence of a concurrent add and remove of some element e, the final state will contain e (or, more precisely, there exists an add of e that does not happen before a remove of e). It is also possible to define a *remove-wins* semantics, where the remove will win over a concurrent add. Other semantics can also be implemented, such as a *last-writer-wins* strategy where an element will belong to the set or not depending on which was the last operation executed, according to the order among operations.

When creating an application, an application developer must select the CRDT with the most appropriate semantics for its goal. For example, in the bank account example, the balance of an account can be modeled as a counter and the set of accounts of a client can be maintained in an add-win set or map CRDT.

In general, an application operation will manipulate multiple data objects. When using CRDTs, it is possible to maintain replicas of these objects in multiple nodes. An operation can execute by accessing a single replica of each object it accesses. These updates can later be propagated to other nodes, with CRDT rules guaranteeing that the replicas of each object will converge to the same state. By propagating the updates to all objects modified in an operation atomically, it is possible to guarantee that all effects of an operation are observed at the same time.

CRDTs for relational databases In relational databases, it is also possible to model data using CRDTs. Table 1 presents the mapping proposed in SIEVE [20]. Regarding table fields, we defined only two CRDTs. The *LWW* CRDT can be used with any field type and implements a *last-writer-wins* strategy for defining the final value of a field. The *NUMDELTA* CRDT can be used with numeric fields, and transforms each update operation in a downstream operation that adds or subtracts a constant to the value of the field. This can be used to support account balances, counters, etc.

A database table can be seen as a set of tuples. In the general case, and following the semantics of the *ARSET* CRDT, when concurrent *insert*, *update* and *delete* operations occur, the following rules can be used: (1) concurrent inserts of tuples with the same key are treated as an insert followed by a sequence of updates;

(2) for concurrent updates, the rules defined for fields are used to deterministically define the final value; (3) a delete will only take effect if no concurrent update or insert was executed.

While using CRDTs guarantees that all replicas converge to the same state, it does not guarantee that the convergence rules executed independently by different CRDTs maintain application invariants. Next, we show how we can address this problem by restricting the concurrent execution of operations that can break application invariants.

5 Preserving invariants with minimal coordination

As mentioned before, in the banking example, the `withdraw` operation, despite being commutative, cannot execute under weak consistency, as the concurrent execution of multiple withdrawals can break the invariant that the account balance cannot be negative. To avoid the possibility of breaking the invariant, RedBlue consistency would label all withdrawals as red, requiring replicas to coordinate the execution of every withdraw operation. In practice, however, only in a few cases the cumulative effects of all concurrent withdrawals will surpass the actual balance of the account.

To relieve the strong constraint imposed by RedBlue consistency, we propose a more efficient coordination plan: given some account balance, replicas can coordinate beforehand to split the balance among them. Until a replica consumes its allocated share of the balance, it can execute operations locally, without coordination with other replicas, with the guarantee that the balance will not become negative, i.e., the application invariant will not be broken.

The above idea has been previously explored in the context of escrow transactions [9, 27]. We revisit and generalize the concept of escrow transactions, to allow replicas to assess the safety of operations without coordination when executing operations. In our generalization, when replicas cannot ensure an operation is safe by reading local state, they contact remote peers to update their vision of the database to decide the fate of the operation. In addition, we discuss how we avoid the coordination across sites for all red operations, which is required for totally ordering them. Instead, we identify a small set of coordination requirements between operations, and show how to enforce those rules at runtime.

5.1 Explicit Consistency in a nutshell

We present a new consistency model, called Explicit Consistency, that extends RedBlue consistency to avoid the coordination of red operations when possible. The idea is that instead of labeling shadow operations as red or blue, programmers specify the application invariants. The system must execute operations while guaranteeing that these invariants are not broken.

To this end, we propose the following methodology for creating applications that adhere to Explicit Consistency. First, programmers must specify the application invariants and operation effects. Second, we provide a tool to analyze the specification of the application and identify the pairs of conflicting shadow operations. Non-conflicting shadow operations execute without any restrictions, as blue operations. We include a library of CRDTs to help programmers define commutative operations. Third, for each pair of conflicting shadow operations, the programmer can use a specialized concurrency control mechanism that restricts the concurrent execution of these operations. This mechanism executes coordination outside of the critical path of operation execution, allowing these operations to execute locally without the need to coordinate with other replicas.

The following sections provide additional details on these steps to use the Explicit Consistency model.

5.2 Application specification

Programmers specify application invariants and the post-conditions of shadow operations as first order logic expressions. Invariants must be written as universally quantified formulas in prenex normal form, while the

```
01: //Invariant declaration
02: @Invariant( "forall(aId : Aid) :- balance(aId) >= 0" )
03: @Invariant( "forall(cId : Cid, aId : Aid) :- userAccount(cId, aId) =>
04:                                              registeredUser(cId)")
05: public interface Bank{
06:
07:   @decrement(balance(accountId), amount)
08:   boolean withdraw( CId clientId, Aid accountId, Int amount);
09:
10:   @true(userAccount(clientId, accountId))
11:   boolean assignAccount( CId clientId, AId accountId);
12:
13:   @false(userAccount(clientId, accountId))
14:   boolean closeAccount( CId clientId, AId accountId);
16:
17:   @false(registeredUser(clientId))
18:   boolean endContract( CId clientId);
19:
20: }
```

(a) Specification written with Java Annotations.

Operations X *Operations*		*Conflict*
withdraw(cId, aId, amount)	withdraw(cId, aId, amount)	**Non-idempotent**
...		
assignAccount(cId,aId)	closeAccount(cId, aId)	**Opposing**
...		
assignAccount(cId,aId)	endContract(cId)	**Conflict**
...		

(b) Conflicting pairs of operations for the Bank example.

Figure 3: Bank application specification and analysis results.

grammar for specifying applications post-conditions is restricted to predicate assignments, that assert the truth value of some predicate, and function clauses, which define the relation between the value of some predicate before and after the execution of the operation.

The code snippet in figure 3(a) shows the specification of the banking application. We extended this example to illustrate different invariant violations. In the extended version, clients must have a valid contract with the bank to be able to access an account. Clients might have multiple accounts and must close all of them before finishing the contract. In Line 2, the invariant guarantees that an account balance is never negative. In line 3, the invariant states that, for every open account, the account holder must be registered with the bank.

5.3 Analysis

The analysis checks which are the shadow operations whose concurrent execution might produce a database state that is invalid with respect to the declared invariants. Conceptually, for each pair of operations and for every valid state where these operation can execute, the algorithm verifies if the execution of both operations will lead to a state that is not valid according to the invariants of the application. Obviously, checking every pair of operations in every valid state exhaustively is unfeasible. Instead, our algorithm relies in the Z3 satisfiability modulo theory (SMT) solver to perform this verification efficiently. A full description of the algorithm is given in our prior publication [7].

Figure 3(b) summarizes the conflicts in the example of Figure 3(a): two concurrent successful withdrawals might make the balance negative (non-idempotence); assigning and removing an account concurrently for the same user might leave the system in an inconsistent state, because each shadow operation writes different values for the predicate *userAccount*(*cId*, *aId*) (opposing post-conditions); and finally, the pair *createAccount*(*cId*, *aId*) and *endContract*(*cId*) might violate the integrity constraint of line 3, because a new account is being added to a user that is ending a contract with the bank.

5.4 Code instrumentation

After identifying which operations can lead to conflicts, the programmer must instrument the application to avoid them.

Some conflicts can be handled by simply relying on CRDTs to automatically solve them. For example,

our analysis can report that operations have opposing post-conditions: e.g., operations `assignAccout` and `remAccount` assign the value *true* and *false* to predicate $userAccount(cId, aId)$. In this situation, the programmer can choose a preferred value for the predicate and use a CRDT that automatically implements the selected decision[2].

Other conflicts must be handled by restricting the concurrent execution of operations that can cause invariants to be broken. To this end, we provide a set of specialized reservation-based concurrency control mechanisms.

For conflicts on numeric invariants, like the one that withdraw causes, we support an escrow reservation for allowing some decrements of numeric values to execute without coordination. In an escrow reservation, each replica is assigned a budget of decrements, based on the initial value of the data. In our example, when a replica receives a withdraw request, if the local budget is sufficient, the generator operation executes immediately without coordination, generating a shadow operation that decrements the balance. This local execution is safe, guaranteeing that the invariant still holds after executing all concurrent operations, because the sum of the budgets of all replicas is equal to the value of the initial value. If the local budget is not enough to satisfy the request, the replica needs to contact remote replicas to increase its budget, until it can satisfy the request. If that is not possible, because there are not enough resources globally, then the generator operation fails, generating no shadow operation.

For conflicts on generic invariants, we include a multi-value lock reservation. This lock can be in one of the following three states: (1) shared forbid, giving the shared right to forbid some action to occur; (2) shared allow, giving the shared right to allow some action to occur; (3) exclusive allow, giving the exclusive right to execute some action. The idea is that, for a conflicting pair of operations, (o_1, o_2), the lock will be associated with the execution of one of the operations, say o_1. To execute o_1, a replica must hold the lock in the shared allow mode. This right can be shared by multiple replicas. To execute o_2, a replica must hold the lock in the shared forbid mode. As before, when executing the generator operation, if the replica already holds the necessary locks (in the required mode to execute the operation), it can execute locally and generate the corresponding shadow operation. If not, it must contact other replicas to obtain the necessary locks.

Besides these two locks, we also proposed other locks that can efficiently restrict the concurrent execution of operations that conflict in other types of invariants, including conditions on the number of elements that satisfy a given condition and disjunctions. In a related work, Gotsman et. al. [16] have shown how to prove that a given set of locks is sufficient for maintaining invariants.

6 Related work

Many cloud storage systems supporting geo-replication have emerged in recent years. Some of these systems offer variants of eventual consistency, where operations produce responses right after being executed in a single data center (usually the closest one) and are replicated in the background, so that user observed latency is improved [13, 23, 24, 4, 19]. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [23, 4, 14, 6]; supporting limited transactions where a set of updates are made visible atomically [24, 5]; supporting application-specific or type-specific reconciliation with no lost updates [13, 23, 35, 2], etc.

While some systems implement eventual consistency by relying on a simple last-writer-wins strategy, others have explored the semantics of applications (and data types). Semantic types [15] have been used for building non-serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [31] explore commutativity for enabling the automatic merge of concurrent updates to the same data types.

Eventual consistency is insufficient for some applications that require some operations to execute under strong consistency for correctness. To this end, several systems support strong consistency. Spanner provides

[2]In our experience, boolean predicates can be implemented using Set CRDTs with add-wins and remove-wins policies to enforce that the corresponding predicate becomes true or false respectively.

strong consistency for the whole database, at the cost of incurring coordination overhead for all updates [12]. Transaction chains support transaction serializability with latency proportional to the latency to the first replica that the corresponding transaction accesses [40]. MDCC [17] and Replicated Commit [26] propose optimized approaches for executing transactions but still incur inter-data center latency for committing transactions.

Some systems combine the benefits of weak and strong consistency models by allowing both levels to coexist. In Walter [35], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [37] and Pileus [36] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [11] and DynamoDB [34] also combine weak consistency with per-object strong consistency relying on conditional writes, where a write fails in the presence of concurrent writes. RedBlue consistency also combines weak and strong consistency in the same system. Unlike other systems, RedBlue consistency splits operations into generator and shadow parts to allow more operations to commute, and define a procedure to help programmers labeling shadow operations as weak or strong.

Escrow transactions [27] offer a mechanism for enforcing numeric invariants under concurrent execution of transactions. By enforcing local invariants in each transaction, they can guarantee that a global invariant is not broken. This idea can be applied to other data types, and it has been explored for supporting disconnected operation in mobile computing [38, 28, 32]. Balegas et al. [8] proposed the bounded counter CRDT that can be used to enforce numeric invariants in weakly consistent cloud databases. The demarcation protocol [9] aims at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [22] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages. Indigo builds on similar ideas for enforcing application invariants, but it is the first piece of work to provide an approach that, starting from application invariants expressed in first-order logic, leads to the deployment of the appropriate techniques for enforcing such invariants in a geo-replicated weakly consistent data store. Gotsman et. al. [16] propose a proof rule for establishing that the use of a given set of techniques is sufficient to ensure the preservation of invariants.

The static analysis of code is a standard technique used extensively for various purposes, including in a context similar to ours. SIEVE [20] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a RedBlue system [21]. Roy et al. [29] present an analysis algorithm that describes the semantics of transactions. These works are complementary to ours, since the proposed techniques could be used to automatically infer application side effects.

7 Conclusion

In this paper we summarized two of our recent results in addressing the fundamental tension between latency and consistency in geo-replicated systems. First, RedBlue consistency [21] offers fast geo-replication by presenting sufficient conditions that allow programmers to safely separate weakly consistent (fast) operations from strongly consistent (slow) ones in a coarse-grained manner. To increase the space of potential fast operations and simplify the programmer's task of defining commutative operations, we propose the use of conflict-free replicated data types. Second, Explicit Consistency [7] enables programmers to make fine-grained decisions on consistency level assignments by connecting application invariants to ordering conflicts between pairs of operations, and explores efficient reservation techniques for coordinating conflicting operations with low cost.

Acknowledgments

The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant. This research was also supported in part by EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PEst-OE/ EEI/ UI0527/ 2014, NOVA LINCS (UID/CEC/04516/2013), and INESC-ID (UID/CEC/50021/2013).

References

[1] 7 Data Center Disasters You'll Never See Coming. http://www.informationweek.com/cloud/7-data-center-disasters-youll-never-see-coming/d/d-id/1320702. Accessed Feb-2016.

[2] Using data types – riak documentation. http://docs.basho.com/riak/latest/dev/using/data-types/. Accessed Feb-2016.

[3] Consistency without Consensus: CRDTs in Production at SoundCloud. http://www.slideshare.net/InfoQ/consistency-without-consensus-crdts-in-production-at-soundcloud Accessed Feb-2016.

[4] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *EuroSys '13*, 85–98, 2013. ACM.

[5] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD '14*, 27–38, 2014. ACM.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *SIGMOD '13*, 761–772, 2013. ACM.

[7] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *EuroSys '15*, 6:1–6:16, 2015. ACM.

[8] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguica. Extending eventually consistent cloud databases for enforcing numeric invariants. In *SRDS '15*, 31–36, Sept 2015.

[9] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

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

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *OSDI '12*, 251–264, 2012. USENIX Association.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP '07*, 205–220, 2007. ACM.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *SOCC '13*, 11:1–11:14, 2013. ACM.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[16] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *POPL 2016*, 371–384, 2016. ACM.

[17] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *EuroSys '13*, 113–126, 2013. ACM.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[19] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. In *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[20] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *ATC '14*, 281–292, 2014. USENIX Association.

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *OSDI '12*, 265–278, 2012. USENIX Association.

[22] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. In *NSDI '14*, 2014. USENIX Association.

[23] 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 *SOSP '11* , 401–416, 2011. ACM.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *NSDI '13*, 313–328, 2013. USENIX Association.

[25] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI*, 2010.

[26] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9):661–672, 2013.

[27] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[28] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *MobiSys '03*, 43–56, 2003. ACM.

[29] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD '15*, 1311–1326, 2015.

[30] E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at velocity web performance and operations conference. `http://slideplayer.com/slide/1402419/`, 2009.

[31] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *SSS '11*, 386–400, 2011. Springer-Verlag.

[32] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers. In *Middleware '08*, 42–61, 2008. Springer-Verlag New York, Inc.

[33] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually Consistent Byzantine-Fault Tolerance. In *NSDI'09*, 169–184, 2009.

[34] S. Sivasubramanian. Amazon DynamoDB: A Seamlessly Scalable Non-relational Database Service. In *SIGMOD '12*, 729–730, 2012. ACM.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-replicated Systems. In *SOSP '11*, 385–400, 2011. ACM.

[36] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *SOSP '13*, 309–324, 2013. ACM.

[37] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP '95*, 172–182, 1995. ACM.

[38] G. D. Walborn and P. K. Chrysanthis. Supporting Semantics-based Transaction Processing in Mobile Database Applications. In *SRDS '95*, 31–40, 1995. IEEE Computer Society.

[39] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Middleware '15*, 75–87, 2015. ACM.

[40] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *SOSP '13*, 276–291, 2013. ACM.

Strong Consistency at Scale

Carlos Eduardo Bezerra
University of Lugano (USI)
Switzerland

Le Long Hoang
University of Lugano (USI)
Switzerland

Fernando Pedone
University of Lugano (USI)
Switzerland

Abstract

Today's online services must meet strict availability and performance requirements. State machine replication, one of the most fundamental approaches to increasing the availability of services without sacrificing strong consistency, provides configurable availability but limited performance scalability. Scalable State Machine Replication (S-SMR) achieves scalable performance by partitioning the service state and coordinating the ordering and execution of commands. While S-SMR scales the performance of single-partition commands with the number of deployed partitions, replica coordination needed by multi-partition commands introduces an overhead in the execution of multi-partition commands. In the paper, we review Scalable State Machine Replication and quantify the overhead due to replica coordination in different scenarios. In brief, we show that performance overhead is affected by the number of partitions involved in multi-partition commands and data locality.

1 Introduction

In order to meet strict availability and performance requirements, today's online services must be replicated. Managing replication without giving up strong consistency, however, is a daunting task. State machine replication, one of the most fundamental approaches to replication [1, 2], achieves strong consistency (i.e., linearizability) by regulating how client commands are propagated to and executed by the replicas: every non-faulty replica must receive and execute every command in the same order. Moreover, command execution must be deterministic.

State machine replication yields configurable availability but limited scalability. Since every replica added to the system must execute all requests, increasing the number of replicas results in bounded improvements in performance. Scalable performance can be achieved with state partitioning (also known as *sharding*). The idea is to divide the state of a service in multiple partitions so that most commands access one partition only and are equally distributed among partitions. Unfortunately, most services cannot be "perfectly partitioned", that is, the service state cannot be divided in a way that commands access one partition only. As a consequence, partitioned systems must cope with multi-partition commands. Scalable State Machine Replication (S-SMR) [3] divides the service state and replicates each partition. S-SMR relies on an atomic multicast primitive to consistently order commands within and across partitions. Commands that access a single partition are multicast to the concerned partition and executed as in classical SMR; commands that involve multiple partitions are multicast to and executed at all concerned partitions.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Atomic multicast ensures that commands are consistently ordered within and across partitions—in brief, no two replicas deliver the same commands in different orders. However, simply ordering commands consistently is not enough to ensure strong consistency in Scalable State Machine Replication since the execution of commands can interleave in ways that violate strong consistency. In order to avoid consistency violations, S-SMR implements execution atomicity. With execution atomicity, replicas coordinate the execution of multi-partition commands. Although the execution of any two replicas in the same partition does not need coordination, a replica in one partition must coordinate its execution with a replica in every other partition involved in a multi-partition command.

Execution atomicity captures real-time dependencies between commands, typical of strong consistency criteria such as linearizability and strict serializability. In both linearizability and strict serializability, if one operation precedes another in real time (i.e., the first operation finishes before the second operation starts), then this dependency must be reflected in the way the two operations are ordered and executed. Respecting real-time dependencies leads to replicated systems that truly behave like single-copy systems, and thus are easier to program. Serializability does not need execution atomicity but may lead to non-intuitive behavior. We show in the paper that execution atomicity is not as expensive to implement in a partitioned system as one might expect.

S-SMR has proved to provide scalable performance, in some cases involving single-partition commands, with improvements in throughput that grow linearly with the number of partitions [3]. Forcing replicas in different partitions to coordinate to ensure strong consistency may slow down the execution of multi-partition commands, since each replica cannot execute at its own pace (i.e., a fast replica in a partition may need to wait for a slower replica in a different partition). This paper reviews the Scalable State Machine Replication approach and takes a close look at replica coordination in the execution of multi-partition commands. For multi-partition commands that require replicas in different partitions to exchange data as part of the execution of the command, execution atomicity does not affect performance. For multi-partition commands that do not require data exchange, execution atomicity has an impact on performance that depends on the number of partitions involved in the command and on data locality. For example, in workloads that do not experience locality, the overhead in throughput introduced by execution atomicity in commands that span two partitions is around 32% in executions with 16 partitions; in workloads with data locality, this overhead is around 27%.

The remainder of the paper is structured as follows. Section 2 presents the system model and definitions. Sections 3 and 4 recall classical state machine replication and scalable state machine replication. Section 5 describes our experimental evaluation. Section 6 reviews related work and Section 7 concludes the paper.

2 System model and definitions

2.1 Processes and communication

We consider a distributed system consisting of an unbounded set of client processes $C = \{c_1, c_2, ...\}$ and a bounded set of server processes (replicas) $\mathcal{S} = \{s_1, ..., s_n\}$. Set $\mathcal{S}$ is divided into disjoint groups of servers $\mathcal{S}_1, ..., \mathcal{S}_k$. Processes are either *correct*, if they never fail, or *faulty*, otherwise. In either case, processes do not experience arbitrary behavior (i.e., no Byzantine failures). Each server group S_i contains at least $f + 1$ correct processes, where f is the number of faulty processes.

Processes communicate by message passing, using either one-to-one or one-to-many communication. The system is asynchronous: there is no bound on message delay or on relative process speed. One-to-one communication uses primitives $send(p, m)$ and $receive(m)$, where m is a message and p is the process m is addressed to. If sender and receiver are correct, then every message sent is eventually received. One-to-many communication relies on reliable multicast and atomic multicast,[1] defined in Sections 2.2 and 2.3, respectively.

[1]Solving atomic multicast requires additional assumptions [4, 5]. In the following, we simply assume the existence of an atomic multicast oracle.

2.2 Reliable multicast

To reliably multicast a message m to a set of groups γ, processes use primitive reliable-multicast(γ, m). Message m is delivered at the destinations with reliable-deliver(m). Reliable multicast has the following properties:

- If a correct process reliable-multicasts m, then every correct process in γ reliable-delivers m *(validity)*.
- If a correct process reliable-delivers m, then every correct process in γ reliable-delivers m *(agreement)*.
- For any message m, every process p in γ reliable-delivers m at most once, and only if some process has reliable-multicast m to γ previously *(integrity)*.

2.3 Atomic multicast

To atomically multicast a message m to a set of groups γ, processes use primitive atomic-multicast(γ, m). Message m is delivered at the destinations with atomic-deliver(m). Atomic multicast ensures the following properties:

- If a correct process atomic-multicasts m, then every correct process in γ atomic-delivers m *(validity)*.
- If a process atomic-delivers m, then every correct process in γ atomic-delivers m *(uniform agreement)*.
- For any message m, every process p in γ atomic-delivers m at most once, and only if some process has atomic-multicast m to γ previously *(integrity)*.
- No two processes p and q in both γ and γ' atomic-deliver m and m' in different orders; also, the delivery order is acyclic *(atomic order)*.

Atomic broadcast is a special case of atomic multicast in which there is a single group of processes.

3 State machine replication

State machine replication is a fundamental approach to implementing a fault-tolerant service by replicating servers and coordinating the execution of client commands against server replicas [1, 2]. The service is defined by a state machine, which consists of a set of *state variables* $\mathcal{V} = \{v_1, ..., v_m\}$ and a set of *commands* that may read and modify state variables, and produce a response for the command. Each command is implemented by a deterministic program. State machine replication can be implemented with atomic broadcast: commands are atomically broadcast to all servers, and all correct servers deliver and execute the same sequence of commands.

We consider implementations of state machine replication that ensure linearizability. Linearizability is defined with respect to a sequential specification. The *sequential specification* of a service consists of a set of commands and a set of *legal sequences of commands*, which define the behavior of the service when it is accessed sequentially. In a legal sequence of commands, every response to the invocation of a command immediately follows its invocation, with no other invocation or response in between them. For example, a sequence of operations for a read-write variable v is legal if every read command returns the value of the most recent write command that precedes the read, if there is one, or the initial value, otherwise. An execution $\mathcal{E}$ is linearizable if there is some permutation of the commands executed in $\mathcal{E}$ that respects (i) the service's sequential specification and (ii) the real-time precedence of commands. Command C_1 precedes command C_2 in real-time if the response of C_1 occurs before the invocation of C_2.

In classical state machine replication, throughput does not scale with the number of replicas: each command must be ordered among replicas and executed and replied by every (non-faulty) replica. Some simple optimizations to the traditional scheme can provide improved performance but not scalability. For example, although

update commands must be ordered and executed by every replica, only one replica can respond to the client, saving resources at the other replicas. Commands that only read the state must be ordered with respect to other commands, but can be executed by a single replica, the replica that will respond to the client.

4 Scalable State Machine Replication

In this section, we describe an extension to SMR that under certain workloads allows performance to grow proportionally to the number of replicas [3]. We first recall S-SMR and then discuss some of its performance optimizations.

4.1 General idea

S-SMR divides the application state $\mathcal{V}$ (i.e., state variables) into k partitions $\mathcal{P}_1, ..., \mathcal{P}_k$, where for each $\mathcal{P}_i$, $\mathcal{P}_i \subseteq \mathcal{V}$. Moreover, we require each variable v in $\mathcal{V}$ to be assigned to at least one partition and define $part(v)$ as the partitions that hold v. Each partition $\mathcal{P}_i$ is replicated by servers in group $\mathcal{S}_i$. For brevity, we say that server s belongs to $\mathcal{P}_i$ with the meaning that $s \in \mathcal{S}_i$, and say that client c multicasts command C to partition $\mathcal{P}_i$ meaning that c multicasts C to group $\mathcal{S}_i$.

To execute command C, the client multicasts C to all partitions that hold a variable read or updated by C. Consequently, the client must be able to determine the partitions that may be accessed by C. Note that this assumption does not imply that the client must know all variables accessed by C, nor even the exact set of partitions. If the client cannot determine a priori which partitions will be accessed by C, it must define a superset of these partitions, in the worst case assuming all partitions. For performance, however, clients must strive to provide a close approximation to the command's actually accessed partitions. We assume the existence of an oracle that tells the client which partitions should receive each command.

Upon delivering command C, if server s does not contain all variables read by C, s must communicate with servers in other partitions to execute C. Essentially, s must retrieve every variable v read in C from a server that stores v (i.e., a server in a partition in $part(v)$). Moreover, s must retrieve a value of v that is consistent with the order in which C is executed, as we explain next.

In more detail, let op be an operation in the execution of command C. We distinguish between three operation types: $read(v)$, an operation that reads the value of a state variable v, $write(v, val)$, an operation that updates v with value val, and an operation that performs a deterministic computation.

Server s in partition $\mathcal{P}_i$ executes op as follows.

i) op is a $read(v)$ operation.
If $\mathcal{P}_i \in part(v)$, then s retrieves the value of v and sends it to every partition $\mathcal{P}_j$ that delivers C and does not hold v. If $\mathcal{P}_i \notin part(v)$, then s waits for v to be received from a server in a partition in $part(v)$.

ii) op is a $write(v, val)$ operation.
If $\mathcal{P}_i \in part(v)$, s updates the value of v with val; if $\mathcal{P}_i \notin part(v)$, s executes op, creating a local copy of v, which will be up-to-date at least until the end of C's execution.

iii) op is a computation operation.
In this case, s executes op.

It turns out that atomically ordering commands and following the procedure above is not enough to ensure linearizability [3]. Consider the execution depicted in Figure 1 (a), where state variables x and y have initial value of 10. Command C_x reads the value of x, C_y reads the value of y, and C_{xy} sets x and y to value 20. Consequently, C_x is multicast to partition $\mathcal{P}_x$, C_y is multicast to $\mathcal{P}_y$, and C_{xy} is multicast to both $\mathcal{P}_x$ and $\mathcal{P}_y$. Servers in $\mathcal{P}_y$ deliver C_y and then C_{xy}, while servers in $\mathcal{P}_x$ deliver C_{xy} and then C_x, which is consistent with

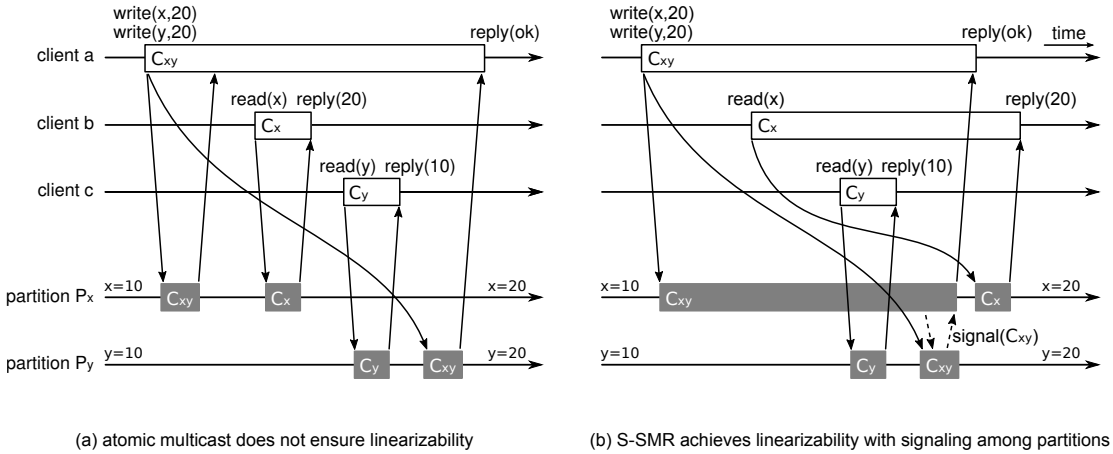


Figure 1: Atomic multicast and S-SMR. (To simplify the figure, we show a single replica per partition.)

atomic order. In this execution, the only possible legal permutation for the commands is C_y, C_{xy}, and C_x, which violates the real-time precedence of the commands, since C_x precedes C_y in real-time.

Intuitively, the problem with the execution in Figure 1 (a) is that commands C_x and C_y execute "in between" the execution of C_{xy} at partitions $\mathcal{P}_x$ and $\mathcal{P}_y$. In S-SMR, we avoid such cases by ensuring that the execution of every command is atomic. Command C is *execution atomic* if, for each server s that executes C, there exists at least one server r in every other partition in $part(C)$ such that the execution of C at s finishes only after r starts executing C. More precisely, let $start(C, p)$ and $end(C, p)$ be, respectively, the time when server p starts executing command C and the time when p finishes C's execution. Execution atomicity ensures that, for every server s in partition $\mathcal{P}$ that executes C, there is a server r in every $\mathcal{P}' \in part(C)$ such that $start(C, r) < end(C, s)$. Intuitively, this condition guarantees that the execution of C at s and r overlap in time.

Replicas can ensure execution atomicity by coordinating the execution of commands. After starting the execution of command C, servers in each partition send a $signal(C)$ message to servers in the other partitions in $part(C)$. Before finishing the execution of C and sending a reply to the client that issued C, each server must receive a $signal(C)$ message from at least one server in every other partition that executes C. Because of this scheme, each partition is required to have at least $f + 1$ correct servers, where f is the maximum number of tolerated failures per partition; if all servers in a partition fail, service progress is not guaranteed.

Figure 1 (b) shows an execution of S-SMR. In the example, servers in $\mathcal{P}_x$ wait for a signal from $\mathcal{P}_y$, therefore ensuring that the servers of both partitions are synchronized during the execution of C_{xy}. Note that the outcome of each command execution is the same as in case (a), but the execution of C_x, C_y and C_{xy}, as seen by clients, now overlap in time with one another. Hence, there is no real-time precedence among them and linearizability is not violated.

4.2 Performance optimizations

The scheme described in the previous section can be optimized in many ways. In this section, we briefly mention some of these optimizations and then detail caching.

- Server s does not need to wait for the execution of command C to reach a $read(v)$ operation to only then multicast v to the other partitions in $part(C)$. If s knows that v will be read by C, s can send v's value to the other partitions as soon as s starts executing C.

- The exchange of objects between partitions serves the purpose of signaling. Therefore, if server s sends variable v's value to server r in another partition, r does not need to receive a signal message from s's

partition.

- It is not necessary to exchange each variable more than once per command since any change during the execution of the command will be deterministic and thus any changes to the variable can be applied to the cached value.

- Even though all replicas in all partitions in $part(C)$ execute C, a reply from a replica in a single partition suffices for the client to finish the command.

Server s in partition $\mathcal{P}$ can cache variables that belong to other partitions. There are different ways for s to maintain cached variables; here we define two techniques: conservative caching and speculative caching. In both cases, the basic operation is the following: When s executes a command that reads variable x from some other partition $\mathcal{P}_x$, after retrieving the value of x from a server in $\mathcal{P}_x$, s stores x's value in its cache and uses the cached value in future read operations. If a command writes x, s updates (or creates) x's local value. Server s will have a valid cache of x until (i) s discards the entry due to memory constraints, or (ii) some command not multicast to $\mathcal{P}$ changes the value of x. Since servers in $\mathcal{P}_x$ deliver all commands that access x, these servers know when any possible cached value of x is stale. How servers use cached entries distinguishes conservative from speculative caching.

Servers in $\mathcal{P}_x$ can determine which of its variables have a stale value cached in other partitions. This can be done by checking if there was any command that updated a variable x in $\mathcal{P}_x$, where such command was not multicast to some other partition $\mathcal{P}$ that had a cache of x. Say servers in $\mathcal{P}_x$ deliver command C, which reads x, and say the last command that updated the value of x was C_w. Since $x \in \mathcal{P}_x$, servers in $\mathcal{P}_x$ delivered C_w. One way for servers in $\mathcal{P}_x$ to determine which partitions need to update their cache of x is by checking which destinations of C did not receive C_w. This can be further optimized: even if servers in $\mathcal{P}$ did not deliver C_w, but delivered some other command C_r that reads x and C_r was ordered by multicast after C_w, then $\mathcal{P}$ already received an up-to-date value of x (sent by servers in $\mathcal{P}_x$ during the execution of C_r). If servers in $\mathcal{P}$ discarded the cache of x (e.g., due to limited memory), they will have to send a request for its value.

Conservative caching: Once s has a cached value of x, before it executes a $read(x)$ operation, it waits for a cache-validation message from a server in $\mathcal{P}_x$. The cache validation message contains a set of pairs (var, val), where var is a state variable that belongs to $\mathcal{P}_x$ and whose cache in $\mathcal{P}$ needs to be validated. If servers in $\mathcal{P}_x$ determined that the cache is stale, val contains the new value of var; otherwise, $\perp$, telling s that its cached value is up to date. If s discarded its cached copy, it sends a request for x to $\mathcal{P}_x$. If it is possible to determine which variables are accessed by C before C's execution, all such messages can be sent upon delivery of the command, reducing waiting time; messages concerning variables that could not be determined a-priori are sent later, during the execution of C, as variables are determined.

Speculative caching: It is possible to reduce execution time by speculatively assuming that cached values are up-to-date. Speculative caching requires servers to be able to rollback the execution of commands, in case the speculative assumption fails to hold. Many applications allow rolling back a command, such as databases, as long as no reply has been sent to the client for the command yet. The difference between speculative caching and conservative caching is that in the former servers that keep cached values do not wait for a cache-validation message before reading a cached entry; instead, a $read(x)$ operation returns the cached value immediately. If after reading some variable x from the cache, during the execution of command C, server s receives a message from a server in $\mathcal{P}_x$ that invalidates the cached value, s rolls back the execution to some point before the $read(x)$ operation and resumes the command execution, now with the up-to-date value of x. Server s can only reply to the client that issued C after every variable read from the cache has been validated.

5 Performance evaluation

In this section, we present the results found with Chirper, a scalable social network application based on S-SMR. We compare the performance impact of the S-SMR signaling mechanism, by running experiments with, and experiments without signaling turned on. This was done for different workloads and numbers of partitions. In Section 5.1, we describe the implementation of Chirper. In Section 5.2, we describe the environment where we conducted our experiments. In Section 5.3, we report the results.

5.1 Chirper

We implemented Chirper, a social network application similar to Twitter, in order to evaluate the performance of S-SMR. Twitter is an online social networking service in which users can post 140-character messages and read posted messages of other users. The API of Chirper includes: post (user publishes a message), follow (user starts following another user), unfollow (user stops following someone), and getTimeline (user requests messages of all people whom the user follows).

Chirper partitions the state based on user id. A function $f(uid)$ returns the partition that contains all up-to-date information regarding user with id uid. Taking into account that a typical user probably spends more time reading messages (i.e., issuing getTimeline) than writing them (i.e., issuing post), we decided to optimize getTimeline to be single-partition. This means that, when a user requests his or her timeline, all messages should be available in the partition that stores that user's data, in the form of a materialized timeline (similarly to a materialized view in a database). To make this possible, whenever a post request is executed, the message is inserted into the materialized timeline of all users that follow the one that is posting. Also, when a user starts following another user, the messages of the followed user are inserted into the follower's materialized timeline as part of the command execution; likewise, they are removed when a user stops following another user. Because of this design decision, every getTimeline request accesses only one partition, follow and unfollow requests access objects on at most two partitions, and post requests access up to all partitions.

One detail about the post request is that it needs access to all users that follow the user issuing the post. Since the Chirper client cannot know for sure who follows the user, it keeps a cache of followers. The client cache can become stale if a different user starts following the poster. To ensure linearizability when executing a post request, the Chirper server checks if the command is sent to the proper set of partitions. If this is the case, the request is executed. Otherwise, the server sends a $retry(\gamma)$ message, where γ is the complete set of additional partitions the command must be multicast to. Upon receiving the $retry(\gamma)$ message, the Chirper client multicasts the command again, now with the destination that includes all partitions in γ. This repeats until all partitions that contain followers of the poster deliver the command. This is guaranteed to terminate because partitions are only added to the set of destinations for retries, never removed. Therefore, in the worst case scenario, the client will retry until it multicasts the post request to all partitions of the system.

Moreover, in order to observe the impact of the signaling mechanism described above, we also introduced these two commands: Follow-noop and Unfollow-noop, which demonstrate pure signal exchange, since they do not change the structure of the social network (i.e., do not change the list of followers and followed users).

5.2 Environment setup and configuration parameters

All experiments were conducted on a cluster that had two types of nodes: (a) HP SE1102 nodes, equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b) Dell SC1435 nodes, equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve 2920-48G gigabit network switch, and the Dell nodes were connected to another, identical switch. Those switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 7.1 with kernel 3.10 and had the OpenJDK Runtime Environment 8 with the 64-Bit Server

VM (build 25.45-b02). We kept the clocks synchronized using NTP in order to measure latency components involving events in different computers.

For the experiments, we used the following workloads: Timeline (composed only of getTimeline requests), Post (only post requests), and Follow/unfollow (50% of follow-noop requests and 50% of unfollow-noop).

5.3 Results

For the Timeline workload, the throughput and latency when signaling was turned on and off are very similar in both workloads without and with locality (Figures 3 and 2, respectively). This happens because getTimeline requests are optimized to be single-partition: all posts in a user's timeline are stored along with the User object. Every getTimeline request accesses a single User object (of the user whose timeline is being requested). The structure and content of the network do not change, and partitions do not need to exchange either signal or data. Thus, signals do not affect the performance of Chirper when executing getTimeline requests only.

In the Post workload, every command accesses up to all partitions, which requires partitions to exchange both data and signals. In the execution with only one partition, where signals and data are not exchanged anyway, turning off signaling does not change performance. With two partitions or more, signals and data are exchanged across partitions. However, we can see that the signaling does not affect the performance of Chirper significantly. This happens because the Post command changes the content of the User object (changes in timeline), so in both tests, partitions have to exchange data anyhow. For this reason, even when signaling is turned off, the partitions still have to communicate. As a result, the throughput and delay with signaling on or off are similar.

In the Follow/unfollow workload, each command accesses up to two partitions in the system, requires partitions to exchange signal (no data exchanged since the noop command does not change the content of the network). With one partition, Chirper performs in a similar way to that observed with the Post workload in the experiment with one partition. With two partitions or more, performance of Chirper decreased when signaling was turned on. This was expected since the partitions started to exchange the signals: partitions had to wait for signal during the execution of the command.

For both Post and Follow/unfollow workloads, datasets with locality (Figure 2) result in higher throughput and lower latency than datasets without locality (Figure 3).

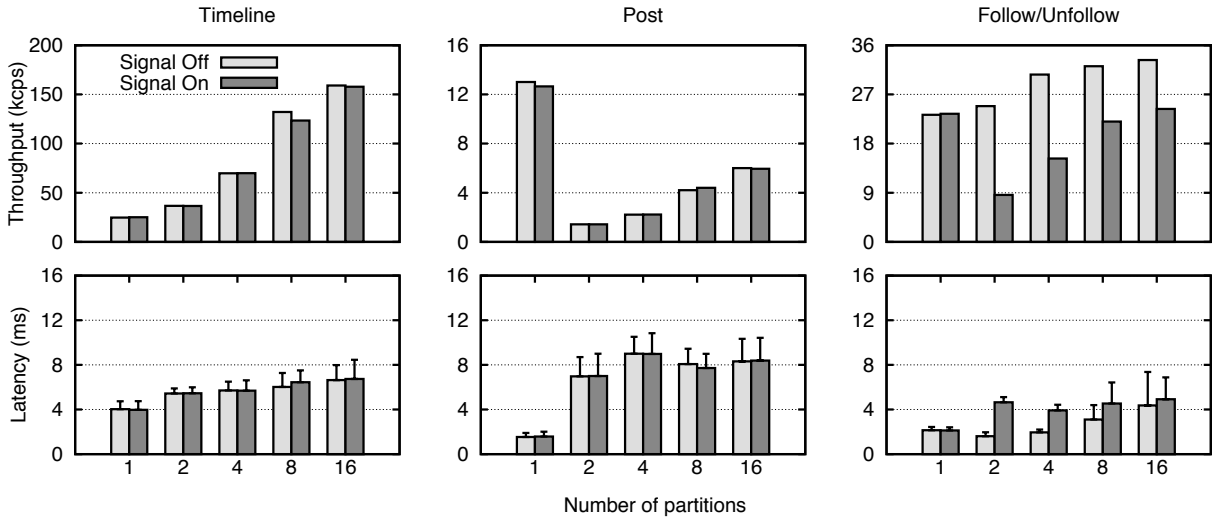


Figure 2: Results of Chirper running with signaling turned on and off, on a dataset with locality. Throughput is shown in thousands of commands per second (kcps). Latency is measured in milliseconds (bars show average and whiskers show 95-th percentile).

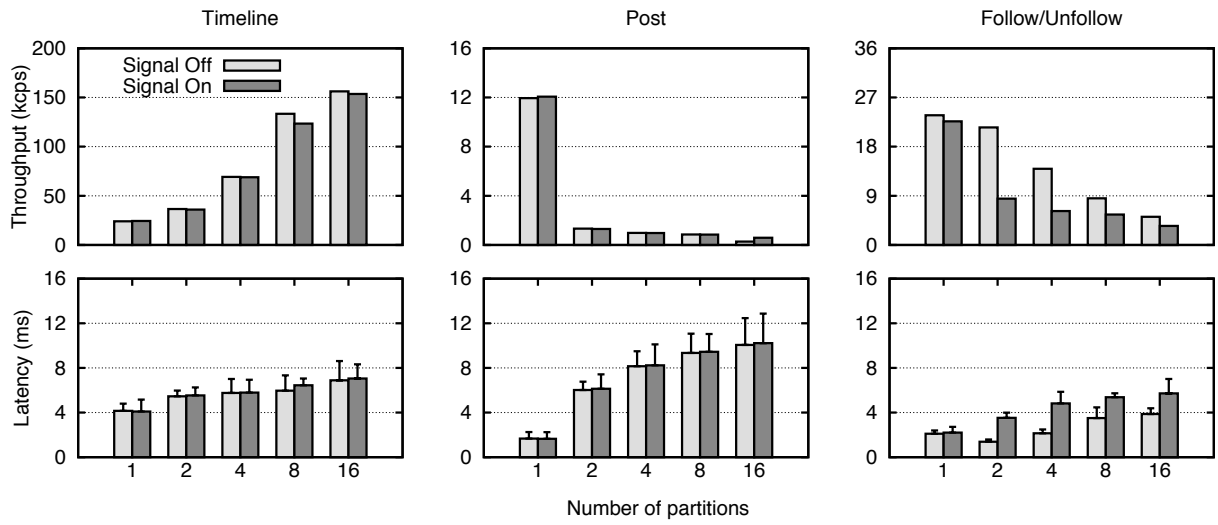


Figure 3: Results of Chirper running with signaling turned on and off, on a balanced dataset. Throughput is shown in thousands of commands per second (kcps). Latency is measured in milliseconds (bars show average and whiskers show 95-th percentile).

6 Related work

State machine replication is a well-known approach to replication and has been extensively studied (e.g., [1, 2, 6, 7, 8]). State machine replication requires replicas to execute commands deterministically, which implies sequential execution. Even though increasing the performance of state machine replication is non-trivial, different techniques have been proposed for achieving scalable systems, such as optimizing the propagation and ordering of commands (i.e., the underlying atomic broadcast algorithm). In [9], the authors propose to have clients send their requests to multiple computer clusters, where each such cluster executes the ordering protocol only for the requests it received, and then forwards this partial order to every server replica. The server replicas, then, must deterministically merge all different partial orders received from the ordering clusters. In [10], Paxos [11] is used to order commands, but it is implemented in a way such that the task of ordering messages is evenly distributed among replicas, as opposed to having a leader process that performs more work than the others and may eventually become a bottleneck.

State machine replication seems at first to prevent multi-threaded execution since it may lead to non-determinism. However, some works have proposed multi-threaded implementations of state machine replication, circumventing the non-determinism caused by concurrency in some way. In [8], for instance, the authors propose organizing each replica in multiple modules that perform different tasks concurrently, such as receiving messages, batching, and dispatching commands to be executed. The execution of commands is still sequential, but the replica performs all other tasks in parallel.

Some works have proposed to parallelize the execution of commands in SMR. In [7], application semantics is used to determine which commands can be executed concurrently without reducing determinism (e.g., read-only commands can be executed in any order relative to one another). Upon delivery, commands are directed to a parallelizer thread that uses application-supplied rules to schedule multi-threaded execution. Another way of dealing with non-determinism is proposed in [6], where commands are speculatively executed concurrently. After a batch of commands is executed, replicas verify whether they reached a consistent state; if not, commands are rolled back and re-executed sequentially. Both [7] and [6] assume a Byzantine failure model and in both cases, a single thread is responsible for receiving and scheduling commands to be executed. In the Byzantine

failure model, command execution typically includes signature handling, which can result in expensive commands. Under benign failures, command execution is less expensive and the thread responsible for command reception and scheduling may become a performance bottleneck.

Many database replication schemes also aim at improving the system throughput, although commonly they do not ensure strong consistency as we define it here (i.e., as linearizability). Many works (e.g., [12, 13, 14, 15]) are based on the deferred-update replication scheme, in which replicas commit read-only transactions immediately, not necessarily synchronizing with each other. This provides a significant improvement in performance, but allows non-linearizable executions to take place. The consistency criteria usually ensured by database systems are serializability [16] or snapshot isolation [17]. Those criteria can be considered weaker than linearizability, in the sense that they do not take into account real-time precedence of different commands among different clients. For some applications, this kind of consistency is good enough, allowing the system to scale better, but services that require linearizability cannot be implemented with such techniques.

Other works have tried to make linearizable systems scalable [18, 19, 20]. In [19], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In [20], a partitioned variant of SMR is proposed, supporting single-partition updates and multi-partition read operations. It relies on total order: all commands have to be ordered by a single sequencer (e.g., a Paxos group of acceptors), so that linearizability is ensured. The replication scheme proposed in [20] does not allow multi-partition update commands. Spanner [18] uses a separate Paxos group per partition. To ensure strong consistency across partitions, it assumes that clocks are synchronized within a certain bound that may change over time. The authors say that Spanner works well with GPS and atomic clocks.

Scalable State Machine Replication employs state partitioning and ensures linearizability for any possible execution, while allowing throughput to scale as partitions are added, even in the presence of multi-partition commands and unsynchronized clocks.

7 Final remarks

This paper described S-SMR, a scalable variant of the well-known state machine replication technique. S-SMR differs from previous related works in that it allows throughput to scale with the number of partitions without weakening consistency. We evaluate S-SMR with Chirper, a scalable social network application. Our experiments demonstrate that throughput scales with the number of partitions, with nearly ideal (i.e., linear) scalability for workloads composed solely of single-partition commands. Moreover, the results show replica coordination, needed to ensure linearizability, has a relatively small cost (in throughput and latency) and this cost decreases with the number of partitions. For multi-partition commands that already require data exchange between partitions, this extra cost is virtually zero.

Acknowledgements

This work was supported in part by the Swiss National Science Foundation under grant number 146404.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[3] C. E. Bezerra, F. Pedone, and R. van Renesse, "Scalable state machine replication," *DSN*, pp. 331–342, 2014.

[4] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty processor," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[6] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pp. 237–250, USENIX Association, 2012.

[7] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, 2004.

[8] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pp. 266–275, IEEE Computer Society, 2013.

[9] M. Kapritsos and F. Junqueira, "Scalable agreement: Toward ordering as a service," in *Proceedings of the Sixth Worshop on Hot Topics in System Dependability*, HotDep '10, pp. 1–8, USENIX Association, 2010.

[10] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-Paxos: Offloading the leader for high throughput state machine replication," in *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pp. 111–120, IEEE Computer Society, 2012.

[11] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[12] P. Chundi, D. Rosenkrantz, and S. Ravi, "Deferred updates and data placement in distributed databases," in *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pp. 469–476, IEEE Computer Society, 1996.

[13] T. Kobus, M. Kokocinski, and P. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pp. 286–296, IEEE Computer Society, 2013.

[14] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, pp. 1–12, IEEE Computer Society, 2012.

[15] A. Sousa, R. Oliveira, F. Moura, and F. Pedone, "Partial replication in the database state machine," in *Proceedings of the IEEE International Symposium on Network Computing and Applications*, NCA '01, pp. 298–309, IEEE Computer Society, 2001.

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

[17] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. Armendáriz-Iñigo, "Snapshot isolation and integrity constraints in replicated databases," *ACM Transactions on Database Systems*, vol. 34, no. 2, pp. 11:1–11:49, 2009.

[18] J. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems*, vol. 31, no. 3, pp. 8:1–8:22, 2013.

[19] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in Scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 15–28, ACM, 2011.

[20] P. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, DSN '11, pp. 454–465, IEEE Computer Society, 2011.

Call for participation

32nd IEEE International Conference on Data Engineering
May 16-20, 2016, Helsinki, Finland

www.icde2016.fi

The annual ICDE conference addresses research issues in designing, building, managing, and evaluating advanced data systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences.

Venue: ICDE 2016 is to be held in Helsinki, hosted by Aalto University School of Science.

Conference Events

- Research papers
- Industrial Papers
- Demos
- Keynotes
- Tutorials
- Panels
- Workshops
- Posters

General Chairs

Boris Novikov, Saint Petersburg University, Russia
Eljas Soisalon-Soininen, Aalto University School of Science, Finland

Affiliated Workshops

- CloudDM–Workshop on Cloud Data Management
- HDMM 2016–Health Data Management and Mining
- DESWeb 2016–7th International Workshop on Data Engineering meets the Semantic Web
- HardDB 2016–Big Data Management on Emerging Hardware
- KEYS 2016–The Fourth International Workshop on Keyword Search and Data Exploration on Structured Data

Aalto University
School of Science

IEEE computer society

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

TCDE Chair

Kyu-Young Whang
KAIST
371-1 Koo-Sung Dong, Yoo-Sung Ku
Daejeon 305-701, Korea
kywhang@cs.kaist.ac.kr

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398