A Unified and Understandable Summary of Non-transactional Consistency Levels and Protocols for Distributed Replication

Guanzhou Hu
UW–Madison
guanzhou.hu@wisc.edu

1 INTRODUCTION

The crucial first step towards designing distributed replication protocols and building reliable distributed storage systems is to define their consistency semantics[1]. However, apart from the purely formal summary by Viotti and Vukolić [50], there has been no unified, condensed definition of existing consistency levels in the context of distributed replication systems. This is largely due to the rich and convoluted history of research that contributed to this field. Many of the fundamental works stemmed from various research areas, including distributed system modeling [21, 28, 32, 34, 35, 39, 48], multiprocessor shared memory consistency [1–3, 29, 41, 44, 47], network reliability modeling [10, 13, 17, 19, 22], and database transaction processing [23, 25, 43]. They use different names within different contexts, leading to plentiful but sometimes blurry terminology when applied to distributed replication.

This report presents a unified, understandable, and sufficiently complete summary of non-transactional consistency levels in the context of a shared object store service. Compared to previous work [50], this report aims to achieve the following three goals. First, we propose a minimal yet self-contained theoretical framework – the *Shared Object Pool* (SOP) model – which unifies the definition of common consistency levels in a way that is understandable to protocol designers and system engineers. Second, we restrict our discussion to a set of selected non-transactional consistency levels that are interesting for real object storage implementations. Third, to further improve understandability, we use examples extensively to explain the practical differences between consistency levels, and give a mapping between consistency levels and well-known replication protocols.

Section 2 describes our problem model setup, defines logical ordering and validity constraints, and explains the meaning of non-transactional consistency within this context. Section 3 describes all levels of ordering validity constraints. Section 4 presents the hierarchy of selected consistency levels, dissects their ordering validity constraint guarantees, and explains their practical differences. Section 5 presents the availability upper bound of each level in the presence of network partitioning. Section 6 gives examples of distributed replication protocols/systems that belong to one or multiple consistency levels.

2 PROBLEM MODEL

We model our problem setup as a conceptual storage service, which we term a Shared Object Pool (SOP). In this section, we define the SOP model and explain the meaning of consistency.

[1]By consistency, we refer to the constraints that restrict which orderings of operations on shared data objects are considered valid, as defined in §2. This is not to be confused with the "C" in ACID [23, 25], which refers to application-level integrity invariants. In fact, consistency in our context maps to the "I" (*isolation*) in ACID, as we explain in §2.

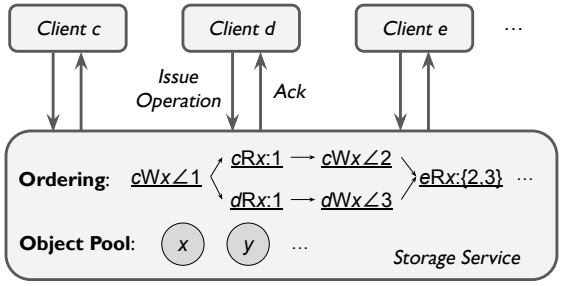


Figure 1: Shared Object Pool (SOP) Model.

2.1 Shared Object Pool (SOP) Model

We consider a storage *service* shared by multiple *clients*, as shown in Figure 1. The service appears to be a pool of *objects*. Each object has a unique name and contains a *value*. The only way to learn about an object's value is through the result of a client read operation, which we introduce below. Objects are not necessarily stored as physical bytes on physical machines; in other words, the SOP model is entirely conceptual and is agnostic to any actual design of protocols and implementation of systems.

Clients are single-threaded entities that invoke **operations** on the service. When a client c issues an operation p, it will block until the acknowledgement of p by the service. An operation is of one of the following three types:

- **Read** (R): we use $|c\mathrm{R}x{:}v|$ to denote client c reading object x and getting the result value v upon acknowledgement. A read operation may also return a set of values, or some arbitrarily reduced value by applying a function f to a set of values. We denote this as $|c\mathrm{R}x{:}f(\{v_1, v_2\})|$, or just $|c\mathrm{R}x{:}\{v_1, v_2\}|$ for short.
- **Write** (W): we use $|c\mathrm{W}x\angle v|$ to denote client c overwriting object x's value with value v.
- **Read-Modify-Write** (RMW): we use $|c\mathrm{RMW}x{:}v\angle v'|$ to denote a compound read-modify-write operation on object x, which reads the value of x, getting v, and writes back a new value v' based on some arbitrary computation over the result of the read. One representative RMW operation is *conditional write*, e.g., *compare-and-swap* (CAS), which reads the current value, compares it against a given value v, and writes a new value v' if the comparison shows equality or writes $v' = v$ back otherwise.

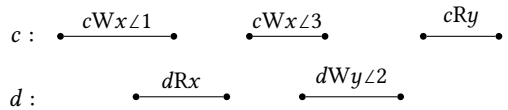
The service maintains a possibly partial **ordering** O of all operations that have been acknowledged. The ordering O captures all dependencies between operations enforced by the service and therefore materializes the result of each operation. Given a *workload* of operations, whether an ordering is acceptable or not is decided by its *validity constraints*. Modeling the ordering validity constraints

guaranteed by the service effectively models its interface semantics, hence its *consistency level*. The following three subsections explain the meaning of workload, ordering, and consistency, respectively.

2.2 Physical Timeline Workload

In the SOP model, each client is a single-threaded entity. For a concrete collection of client operations, we can visualize the *physical timeline* T of when each operation is issued and acknowledged. Every row represents a client, while the x-axis represents the real-world time at which an operation is issued or acknowledged.

For example, below is a physical timeline of two clients, c and d, performing operations on two objects, x and y:



A physical timeline effectively depicts a concrete history of client activity. We can think of it as a specific "workload" that drives the storage service. Given a physical timeline, the storage service chooses a final ordering (from the set of all possible valid orderings allowed by its consistency level) that connects together all operations in the timeline.

Results of read values in R and RMW operations are *not* included in the physical timeline workload. Rather, they are materialized in the final ordering decided by the service. Everything else about client operations activity are included in the physical timeline.

Values of writes are part of the workload. Although we use concrete numeric values as examples throughout this report, they can also be symbolic values that capture the program logic of client applications. For instance, $|d\mathrm{W}y\angle 2|$ in the example above may be $|d\mathrm{W}y\angle v|$, where v is a symbolic value that represents applying a function over the return value of d's preceding read of object x. The write value of a RMW operation is typically a symbolic value that depends on the result of the read.

2.3 Definition of Ordering

An *ordering* is a *directed acyclic graph* (DAG), where nodes are operations in a physical timeline workload. Each operation that has been acknowledged appears exactly once in an ordering. Pending operations that have not been acknowledged are not interesting in our definition of consistency and are thus not explicitly discussed. A directed edge connecting two operations represents an "ordered before" relationship between the two.

We say an operation op_1 is *ordered before* op_2 (denoted $op_1 \rightsquigarrow op_2$) in ordering O iff. there exists either an edge in O pointing from op_1 to op_2, or an operation op' such that $op_1 \rightsquigarrow op'$ and $op' \rightsquigarrow op_2$ (transitivity). If neither operation is ordered before the other, that is, $op_1 \not\rightsquigarrow op_2$ and $op_2 \not\rightsquigarrow op_1$, then we say op_1 and op_2 are *unordered* with each other (denoted $op_1 \not\leftrightsquigarrow op_2$).

Given a physical timeline, an ordering is *valid* on the timeline with respect to a consistency level if it satisfies the validity constraints enforced by that level. We will explain validity constraints in detail in Section 3.

Early Literature Terminology. Similar definitions of "ordered before" relationship have appeared in many early literature [6, 20, 28, 32, 34], where it was often termed "happens before" relationship. Unordered operations in a partial ordering was often termed "concurrent" operations. In this report, we use "ordered before" and "happens before" interchangeably, and use "unordered" and "concurrent" interchangeably. We would like to emphasize that an ordering is a logical representation of how the storage service thinks client operations take effect. It is completely decoupled from the physical timeline workload.

2.4 Meaning of Consistency

The **consistency level** of the storage service is determined by *which orderings of operations are considered valid* given any physical timeline workload. In other words, the consistency level enforces *what validity constraints must be held* on the ordering given any workload. A stronger consistency level imposes more constraints than a weaker one and therefore disallows more orderings, exposing an interface that is more restrictive in the protocol design space and in the meantime easier to use by clients. In contrast, a weaker consistency level relaxes certain constraints and opens up new opportunities in the protocol design space, however providing weaker semantic guarantees for clients.

An ordering represents *logical* dependencies among operations, similar to Lamport's definition of logical clock [34], and does not necessarily capture physical time relationships in the timeline. In fact, whether physical time is respected or not is one of the validity constraints that differentiate several consistency levels, as we demonstrate in Section 4. Our SOP model shares similarities with the specification framework for replicated data types proposed by Burckhardt et al. [20]; the differences are that we simplify the notion of ordering (at the cost of being less expressive in corner cases) and cover stronger consistency levels (rather than focusing only on causal and eventual consistency models).

Note that the SOP model is oblivious to any system design and implementation details of the service, including but not limited to how the service is constructed out of servers, what the network topology looks like, and how are client-server connections established. These internal design choices should not affect the interface semantic exposed to clients. Different replication protocols may make different assumptions of the system and end up providing different consistency levels and availability guarantees. Section 5 presents the availability upper bound of each level in a system of symmetric replicas, and Section 6 gives a mapping between well-known replication protocols and consistency levels.

We only consider a *non-transactional* storage service interface, where each operation touches exactly *one* object. Transactional operations, which group multiple single-object operations together, open up a new dimension in the consistency level space and are essential to distributed database systems. A common practice in modern database systems is to deploy *sharded concurrency control* mechanisms atop replicated data objects, effectively layering transaction isolation guarantees separately from single-object consistency. In spite of this, transaction isolation levels can indeed be integrated into the same unified theoretical framework with single-object consistency [7, 30] (because they are both rooted in the validity of orderings). We leave such integration as future work.

Early Literature Terminology. In early literature on shared memory consistency, operations are further decomposed into *events* [28]. The invocation and acknowledgment of an operation are considered two separate events. All events form a strictly serial sequence, named a *history*. Consistency levels are then defined on the validity of well-formed histories. In this report, we simplify this notation and choose not to use the words "event" and "history". Instead, we consider each operation *op* as a contiguous timespan from its start (when the client issues *op*) to its end (when the service acknowledges *op* and returns a result to the client). When discussing ordering of operations, we use partial ordering to depict incomparability if necessary, instead of merging them into a serial history of events. We found this approach easier to understand and visualize.

3 ORDERING VALIDITY CONSTRAINTS

In this section, we list two sets of *validity constraints* that determine which orderings are acceptable in a consistency level. Specifically, the two sets are: 1) *convergence constraints*, which bound the "shape" of the ordering, and 2) *relationship constraints*, which bound the "placement" of operations with respect to each other within the ordering given any physical timeline workload.

3.1 Convergence Constraints

The convergence constraints restrict whether a valid ordering must be a serial order or can be a partial order, and in the latter case, whether reads must observe convergent results. The three levels of convergence constraints are, from the strongest to the weakest, *Serial Order* (SO), *Convergent Partial Order* (CPO), and *Non-convergent Partial Order* (NPO).

3.1.1 Serial Order (SO)

An SO ordering must be a *total order* of operations, forming a single serial chain.

The result of a read (or RMW) on object x is determined by the latest write (of RMW) operation that *immediately precedes* the read. We say an operation op_1 immediately precedes operation op_2 iff.:

- they are on the same object x, and
- $op_1 \rightsquigarrow op_2$, and
- there is no other write (or RMW) operation op' on object x s.t. $op_1 \rightsquigarrow op' \rightsquigarrow op_2$.

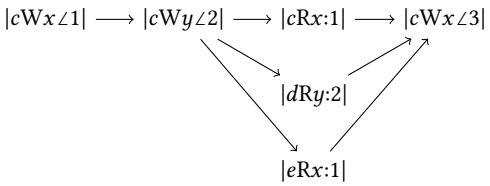
If there is no immediately-preceding operation for a read, we assume a special initial value, e.g. 0, for every object.

Below is an example ordering that satisfies SO:

$$|c\mathrm{W}x\angle 1| \longrightarrow |d\mathrm{W}x\angle 2| \longrightarrow |c\mathrm{R}x{:}2| \longrightarrow |d\mathrm{W}y\angle 2| \longrightarrow |c\mathrm{R}y{:}2|$$

SO is the strongest convergence constraint that any consistency level can enforce. Every operation has a relative position w.r.t. any other operation in the total order. It implies that the service must maintain a centralized view, e.g. a *log*, of all operations [35, 36]; an operation from a client can never be acknowledged solely on its own will.

Cluster of Reads. We make one exception to the seriality of operations in an SO ordering: any cluster of pure read operations in between two writes are allowed to be unordered with each other. For example, the following ordering is a valid SO ordering:



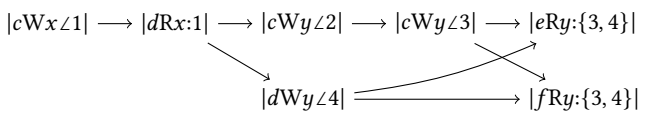
Without loss of generality, in this report, we always present a serial chain when giving SO ordering examples for clarity.

3.1.2 Convergent Partial Order (CPO)

A CPO ordering can be a *partial order* of operations. Writes may be unordered with some other operations, forming branches.

In addition, the result of a read must be *strongly convergent* [50], meaning that it must observe all operations to the same object that immediately precede it. If multiple operations with different values to the same object all immediately precede the read and they are unordered with each other, then the read must return the set of all these values (or a reduced value over the set by applying a *convergent* reduction function, as described in Section 2.1).

Below is an example ordering that satisfies CPO (but does not satisfy SO):



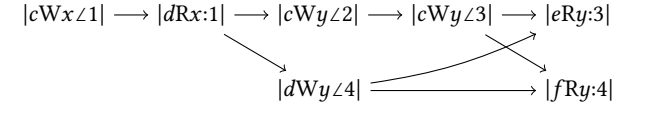
Notice how certain operations are unordered with each other, for example, $|c\mathrm{W}y\angle 2| \not\rightsquigarrow |d\mathrm{W}y\angle 4|$ and $|c\mathrm{W}y\angle 3| \not\rightsquigarrow |d\mathrm{W}y\angle 4|$. Also notice that $|e\mathrm{R}y{:}\{3,4\}|$ and $|e\mathrm{R}y{:}\{3,4\}|$ must observe both values 3 and 4.

CPO opens the opportunity to allow temporarily diverging states of object values, as long as they collapse into a convergent state at some read. This typically gives protocol designers more space to improve the scalability and availability of the service.

3.1.3 Non-convergent Partial Order (NPO)

An NPO ordering can be a partial order of operations, just like in CPO. Furthermore, reads (and RMWs) do not have to be convergent. They are allowed to only observe a *subset* of values from immediately-preceding operations, or apply a *diverging* reduction function that may produce different values on different clients given the same set of input values. Reads still have to be *well-formed*, meaning they cannot observe values that come from nowhere[2].

Below is an example ordering that satisfies NPO (but does not satisfy CPO):



Notice that $|e\mathrm{R}y{:}3|$ is now allowed to only observe value 3 and know nothing about the existence of value 4; similarly for $|f\mathrm{R}y{:}4|$.

[2]For more complex object types such as counters or queues, this means values observed must all obey *return value consistency* of the object semantic [50]. We assume return value consistency for all consistency levels discussed in this report.

NPO allows clients to observe forever-diverging values of the same object. Without careful assistance from the relationship constraints side, a service that only guarantees NPO can hardly provide a reasonable consistency semantic.

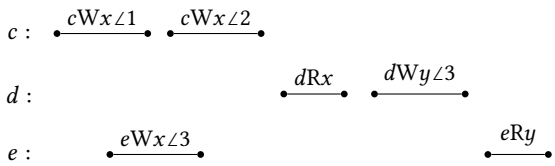
3.2 Relationship Constraints

The relationship constraints restrict how operations are placed with respect to each other in the final ordering. More specifically, they determine what properties in the physical timeline workload must be reflected in the ordering. The four levels of relationship constraints are, from the strongest to the weakest, *Real-Time* (RT), *Causal* (CASL), *First-In-First-Out* (FIFO), and *None*.

3.2.1 Real-Time (RT)

In an RT ordering, if operation op_1 ends before operation op_2 starts in *physical time* (regardless of whether they come from different clients or are on different objects), then the ordering must contain $op_1 \rightsquigarrow op_2$.

For example, given the physical timeline below:



The following is an ordering that is SO and RT:

$$|cWx\angle 1| \rightarrow |eWx\angle 3| \rightarrow |cWx\angle 2| \rightarrow |dRx{:}2| \rightarrow |dWy\angle 3| \rightarrow |eRy{:}3|$$

And the following is an ordering that is CPO and RT:

$$|cWx\angle 1| \longrightarrow |cWx\angle 2| \longrightarrow |dRx{:}\{2,3\}| \longrightarrow |dWy\angle 3| \longrightarrow |eRy{:}3|$$
$$|eWx\angle 3| \nearrow$$

RT is the strongest relationship constraint that any consistency level can enforce. For each client, its operations exhibit the same order as how the client issues them, because an operation naturally finishes before the start of the one that follows it on the same client. Across different clients, RT ensures that an operation observes all operations that have been acknowledged before its start.

The RT guarantee implies that the service must deploy a mechanism to synchronize across all clients' operations; an operation from a client can never be acknowledged solely on its own will.

3.2.2 Causal (CASL)

The causal guarantee relaxes RT by allowing more cases of reordering between cross-client operations. If operation op_2 *causally depends on* operation op_1 [3, 38, 39], then the ordering must contain $op_1 \rightsquigarrow op_2$. Specifically, op_2 causally depends on op_1 iff.:

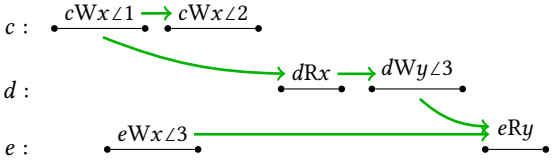
- op_1 and op_2 are from the same client and op_2 follows op_1, or
- op_1 is a write (or RMW), op_2 is a write (or RMW), and op_2 returns the written value of op_1, or
- there is an operation op' s.t. op_2 causally depends on op' and op' causally depends on op_1 (transitivity).

For instance, the following is an SO ordering that satisfies CASL (but does not satisfy RT), given the same example timeline presented in the RT section:

$$|eWx\angle 3| \rightarrow |cWx\angle 1| \rightarrow |dRx{:}1| \rightarrow |dWy\angle 3| \rightarrow |eRy{:}3| \rightarrow |cWx\angle 2|$$

Notice that $|cWx\angle 2|$ ends before $|dRx{:}1|$ starts in physical time, yet $|cWx\angle 2| \not\rightsquigarrow |dRx{:}1|$ in the ordering. Also notice that e's read $|eRy{:}3|$ causally depends on d's write $|dWy\angle 3|$ (and therefore traces back to c's first write $|cWx\angle 1|$), but has nothing to do with c's second write $|cWx\angle 2|$. In other words, the potential "cause" of e reading value 3 out of y (and any future operations made by e after this read, if any) traces back to c's write of value 1 to x.

We can in fact visualize the causal dependencies captured by this ordering by drawing arrows that represent potential causality between operations on the timeline:



The following is another valid ordering that is CPO and CASL on the same timeline example, in which case $|dRx{:}\{1,3\}|$ also observes $|eWx\angle 3|$, setting up an additional causal dependency:

$$|cWx\angle 1| \rightarrow |dRx{:}\{1,3\}| \rightarrow |dWy\angle 3| \rightarrow |eRy{:}3| \rightarrow |cWx\angle 2|$$
$$|eWx\angle 3| \nearrow$$

CASL is weaker than RT. For each client, its own operations still exhibit the same order as how the client issues them. Across different clients, however, CASL is less restrictive than RT. An operation op_2 (or a group of operations) from a client can be reordered before another operation op_1 from a different client, even though op_1 is ahead of op_2 in physical time, as long as op_2 does not causally depend on op_1. This tolerates certain forms of divergence and allows certain operations to be processed concurrently without knowing the existence of others.

Session Guarantees. A popular approach to interpreting causality, as first described in [48], is to think from each client's perspective (termed a *session*) and decompose the CASL constraint into four *session guarantees*:

- *Read My Writes*: if a write op_1 and a read op_2 are from the same client and op_2 follows op_1, then op_2 must observe op_1.
- *Monotonic Writes*: writes by a client must happen in the same order as they are issued by the client.
- *Monotonic Reads*: if two reads are from the same client, then the latter read cannot observe an older state prior to what the former read has observed. This means if a client issues a read op_1 followed by another read op_2, then op_2 must be ordered after all writes that op_1 observes. In this report, we assume a slightly stricter version of this guarantee, where op_2 must be ordered after the read op_1 itself.
- *Writes Follow Reads*, i.e., *Session Causality*: if a client issued a read op' that observed a write op_1, and later issues a write op_2,

then op_2 must become visible after op_1. In this report, we assume a slightly stricter version of this guarantee, where op_2 must be ordered after the read op' itself[3].

The CASL guarantee can be defined exactly as the combination of the four session guarantees [11, 30].

3.2.3 First-In-First-Out (FIFO)

The FIFO guarantee further relaxes CASL by removing write causality dependencies across clients. Specifically, if a read operation op_r from client c observes a write op_w by a different client, now write operations from client c following op_r are allowed to be ordered before op_r and op_w. In other words, writes by different clients do not have to maintain their causality order anymore.

For instance, the following is an SO ordering that satisfies FIFO (but does not satisfy CASL), given the same example timeline presented in the RT section:

$$|e\mathrm{W}x\angle 3| \rightarrow |d\mathrm{W}y\angle 3| \rightarrow |e\mathrm{R}y{:}3| \rightarrow |c\mathrm{W}x\angle 1| \rightarrow |d\mathrm{R}x{:}1| \rightarrow |c\mathrm{W}x\angle 2|$$

Notice that $|d\mathrm{W}y\angle 3|$ is now ordered before $|c\mathrm{W}x\angle 1|$ and $|d\mathrm{R}x{:}1|$, breaking the causality chain. Imagine that another client f is reading objects x and y, it may then observe d's write to y before seeing c's write to x. This may lead to counter-intuitive results for client applications, for example, letting a user see an updated profile page before observing that the user has been removed from the access control list.

The name FIFO comes from the fact that this level of relationship constraint is equivalent to the following definition: writes from each client must be observed by everyone in the same order as they are issued by the client. Writes from different clients are allowed to be unordered with each other. It is as if each client pushes its own writes into a separate FIFO queue.

The FIFO guarantee can be defined exactly as the combination of the *Read My Writes*, *Monotonic Writes*, and *Monotonic Reads* session guarantees [30]. It relaxes CASL by removing *Writes Follow Reads*: a write operation can now get reordered before reads that precede it on the same client, as well as any writes from other clients observed by those reads.

3.2.4 None Relationship

An ordering could of course place no restrictions on the relative positions of operations. In this case, even operations issued by the same client may get arbitrarily reordered. For example, writes by the same client may be visible to another client in a different order than issued, or that a client's read may fail to observe its own preceding write.

This level of relationship constraint demands the least amount of synchronization across operations. Every operation may be processed in a completely asynchronous manner.

4 CONSISTENCY LEVELS

We present the hierarchy of useful consistency levels and dissect each level's ordering validity constraints. We first explain the most common consistency levels, namely *linearizability*, *sequential consistency*, *causal+ consistency*, and *eventual consistency*, followed by more subtle levels. We provide examples along the way to help demonstrate their practical differences.

Figure 2 presents the hierarchy of selected consistency levels. Arrows represent a "stronger than" relationship, where the source level is strictly more restrictive than and implies the destination level. Table 1 defines all these consistency levels by listing their ordering validity constraints.

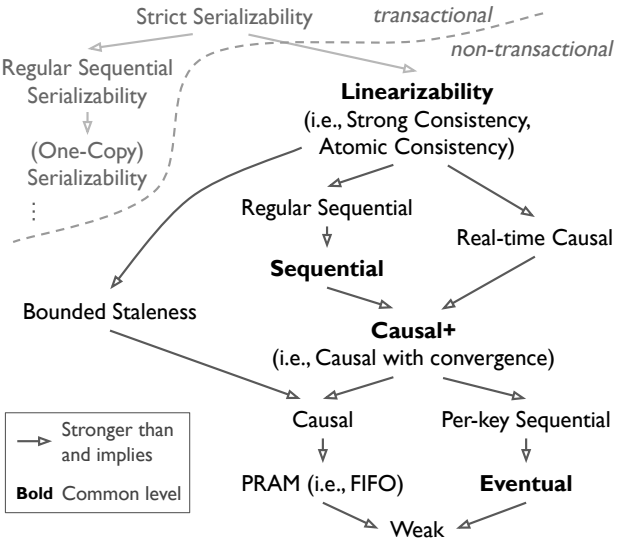


Figure 2: Hierarchy of Selected Consistency Levels.

Consistency Level	Convergence	Relationship
Linearizability	SO	RT
Regular Sequential	SO	RT-W & CASL-R
Sequential	SO	CASL
Bounded Staleness	NPO	Bounded-CASL
Real-time Causal	CPO	RT′
Causal+	CPO	CASL
Causal	NPO	CASL
PRAM	NPO	FIFO
Per-key Sequential	CPO	CASL-per-key
Eventual	CPO	None
Weak	NPO	None

Table 1: Ordering Validity Constraints of Consistency Levels.

4.1 Linearizability

The strongest non-transactional consistency level is *linearizability*, as defined by Herlihy and Wing in [28]. In our model, a *linearizable* ordering can be defined as one that satisfies both SO and RT constraints given a physical timeline. It is a serial total order, where each operation is ordered before all operations that start after its acknowledgment in real time. A service that provides linearizability

[3]Having the slightly stricter versions of *Monotonic Reads* and *Writes Follow Reads* allows us to simplify the notion of causality and use a single ordering instead of two (i.e., *visibility order* and *arbitration order* [50]) to define the selected consistency levels.

is one that always gives a linearizable ordering for any physical timeline workload.

Such a service must maintain some form of a serial log of all operations, where each operation has a specific relative position w.r.t. others. All clients agree on that same order of operations. Furthermore, the service must keep a record of the acknowledgment of each operation, so as to properly order all operations that start after its acknowledgment to satisfy the real-time property.

Linearizability is often referred to as *strong consistency*, due to the fact that it is the strongest possible non-transactional consistency level. Linearizability is sometimes also referred to as *atomic consistency* [28, 41], because a service that provides linearizability appears to be a piece of shared memory where every client operation is an atomic memory operation. This convenient atomicity semantic makes linearizability one of the easiest consistency levels to reason about and verify against; we can just think of the service as a single piece of memory and apply client operations as they arrive, ignoring all internal details about complicated distributed system implementation.

State Machine Replication (SMR). Since the ordering is a serial total order, it is natural to model the object pool as a *state machine* and model client operations as state-transfer *commands*. The service acts as a coordinated set of replicated state machines (typically by replicating the log of operations) and applies committed commands in the decided serial order. This resembles the well-known *State Machine Replication* (SMR) model [33, 46], which is widely used in modeling distributed replication systems[4].

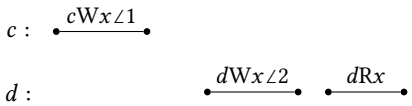
Our *Shared Object Pool* (SOP) model is equivalent to the SMR model if we put some restrictions on both sides. Specifically, an SOP model where only SO orderings are accepted is equivalent to an SMR model where the state is a collection of objects and where there are three types of commands, namely R, W, and RMW operations. The SOP model is more expressive than the SMR model in the aspect that it inherently allows partial orderings, which helps us incorporate consistency levels that do not guarantee SO. The SMR model is more expressive than the SOP model in the aspect that it allows more general state machines with custom states and custom commands, not only reads and writes.

4.2 Sequential Consistency

Sequential consistency was first defined by Lamport in [32]. The original interpretation of this level was that all clients agree on the same *sequence* of operations applied by the service, where operations from each client appear in the sequence in the same order as they are issued by the client, hence the name "sequential consistency". In our model, a service that provides sequential consistency always gives an ordering that is SO and CASL for any physical timeline workload.

Compared to linearizability, since the ordering does not have to be RT anymore, sequential consistency allows the service to move an operation (or a group of operations) backward in time, reordering it before another group that does not causally precede it. This property is sometimes referred to as *unstable ordering* [9, 12], in contrast to *stable ordering* provided by linearizability.

For example, given the following physical timeline:



A linearizable ordering must be SO and RT:

$$|c\mathrm{W}x\angle 1| \longrightarrow |d\mathrm{W}x\angle 2| \longrightarrow |d\mathrm{R}x{:}2|$$

While a sequentially consistent protocol is allowed to give the following ordering that is SO and CASL:

$$|d\mathrm{W}x\angle 2| \longrightarrow |c\mathrm{W}x\angle 1| \longrightarrow |d\mathrm{R}x{:}1|$$

The reordering is allowed because client d did not issue any read on object x before $|d\mathrm{W}x\angle 2|$ that observed value 1 written by client c. Therefore, there is no causal dependency from client c's write $|c\mathrm{W}x\angle 1|$ to client d's write $|d\mathrm{W}x\angle 2|$.

At first glance, it may be hard to tell the exact differences between linearizability and sequential consistency. Attiya and Welch presented a quantitative analysis of the performance implications of these two levels, showing that linearizability is strictly more expensive to implement than sequential consistency for common object types in systems without perfectly synchronized clocks. What semantic power do we lose if we relax the real-time guarantee and allow certain reordering? The following paragraphs explain their three practical differences: 1) sequential consistency does not capture external causality dependencies, 2) sequential consistency is non-local, and 3) it takes extra care to add read-modify-write (RMW) operation support to a sequentially-consistent protocol.

External Causality Dependencies. So far we have assumed that all clients communicate only with the service and there are no *external* communication channels between clients that bypass the service, as depicted in Figure 1. However, in real distributed systems such as cloud databases [16, 24, 31, 49], clients of a replicated storage service may be part of a higher-level system. It is common for clients to coordinate with each other through external causality dependencies, which are impossible for the service to capture without using physical time.

In the example above, it could be that client c first issues a write of value 1 to object x and waits for its acknowledgment. It then sends a message to client d through an external inter-client channel saying "I have finished my write to x and you can go ahead to operate on x". Client d then issues its own write of value 2 and expects to read out value 2. However, since the message from c to d is external to the service, a sequentially consistent service may reorder d's write ahead of c's, and return value 1 for d's read. Figure 3 demonstrates this phenomenon.

A service that provides linearizability will be able to capture such implicit external dependencies, because of the real-time property (as $|d\mathrm{W}x\angle 2|$ starts after $|c\mathrm{W}x\angle 1|$'s acknowledgment in physical time).

[4] We would like to clarify another closely related term – *consensus*. A consensus protocol, e.g. Paxos [35, 36], operates at a lower level than a replication protocol; it is used to achieve agreement on a single value (or a sequence of values in optimized variants) among a set of message-passing processes. A complete SMR protocol, e.g. Multi-Paxos [36] or Raft [45], typically builds on top of or inherently integrates a consensus protocol.

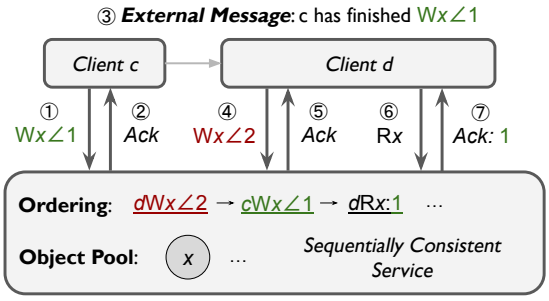
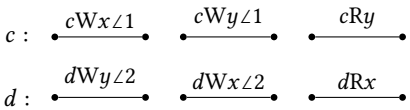


Figure 3: Demonstration of External Causality Dependencies.

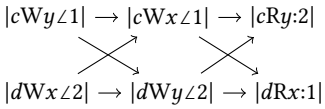
We say that linearizability captures *external causality dependencies*[5], while sequential consistency does not.

Implementation Locality. Herlihy and Wing have proven in [28] that a protocol that implements sequential consistency for each object does not necessarily guarantee overall sequential consistency across all operations. Formally, we say that sequential consistency is *non-local*: it is possible for an ordering to be SO and CASL on each object, while not SO or CASL overall.

For example, given the following physical timeline:



The following ordering is SO and CASL on each object (i.e., the *subordering* on object x and y are both SO and CASL), but the overall ordering is CPO and FIFO:



Notice that given the result of d reading 1 out of x and c reading 2 out of y, it is impossible to resolve an SO and CASL ordering across all six operations. This implies that a protocol that guarantees sequential consistency on each object may fail to come up with a global sequence of operations. In fact, such a protocol provides *per-key sequential consistency* (see Section 4.5.6).

In contrast, a service that provides linearizability on a per-object basis is guaranteed to provide overall linearizability [6, 28]. We say that linearizability is *local*, allowing modular implementation and verification. The above example can only return value 1 for c's read and value 2 for d's read with such a service.

Support for RMW Operations. A protocol that implements sequential consistency for only read (R) and write (W) operations may take advantage of the unstable ordering of writes to speed up the processing of writes. *Shared register* protocols [5, 9] are the primary examples of this category. We describe shared registers in more detail in Section 6.

[5]Note that this is not to be confused with *external consistency* in distributed transaction processing [15, 18], which means that transactions are processed in the same order as they commit, i.e., an enhanced version of *strict serializability*.

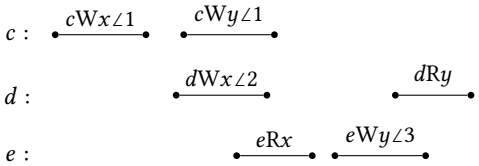
Adding support for read-modify-write (RMW) operations to such protocols is not a trivial task. In particular, we cannot simply treat RMW operations in the same way as pure writes, because RMWs require a stable base value to determine the result of the read. Systems that demand compare-and-swap (CAS) operations, such as the *LogOnce* operation in [24], may have to go for a service that provides linearizability (or *regular sequential consistency* [26], as will be described in Section 4.5.1). Gryff [12] is a recent protocol that adds RMW support atop shared registers.

4.3 Causal+ Consistency

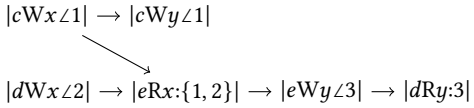
If a global total order is not required, it may be desirable to further relax sequential consistency and embrace the family of causal consistency levels. Causal consistency stems from the definition of *causal memory* [3]. Lloyd et al. pointed out in [38] that distributed replication protocols typically implement a slightly stronger version of causal consistency, which they term *causal+ consistency*. It is essentially causal consistency with *convergence* guarantee.

In our model, a service that provides causal+ consistency always gives an ordering that is CPO and CASL. Compared to sequential consistency, the ordering does not have to be a serial total order but instead may leave certain operations unordered with each other. This opens opportunities to improve the scalability of a replication protocol. However, all causal dependencies still have to be reflected in the decided ordering.

For example, given the following physical timeline:



A service that provides causal+ consistency may give the following ordering that is CPO and CASL:



Notice that $|cWx∠1|$ and $|dWx∠2|$ are unordered with each other, and $|eRx:\{1,2\}|$ observes the values of both writes, hence causally depends on both. $|eWy∠3|$ follows e's read and hence causally depends on both writes as well. $|dRy:3|$ observes the result of e's write and hence continues this causal dependency chain. In contrast, $|cWy∠1|$ is dangling and has not been observed by any reader.

Interpreting A Partial Ordering. Assuming that we are designing a replication protocol atop a set of replica nodes, an intuitive way to interpret a partial ordering in the SOP model is to think from each replica's perspective. Replicas are free to apply an arbitrary order between operations that are unordered with each other in the ordering decided by the consistency level. Figure 4 demonstrates this perspective.

With a consistency level that always gives an SO ordering, all replicas agree on the same sequence of operations. With a consistency level that allows CPO or NPO ordering, replicas may apply

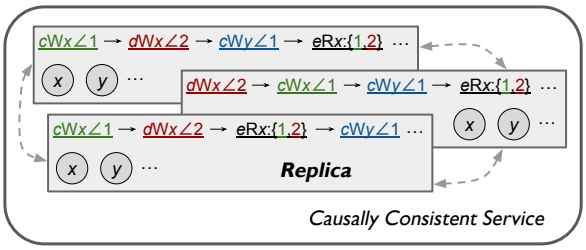
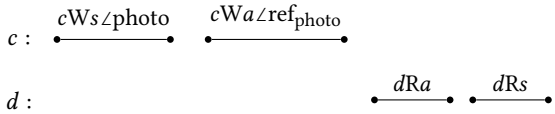


Figure 4: Partial Ordering Interpretation with Replicas.

operations in different orders, as long as everyone is coherent with the required validity constraints. This removes the need to coordinate a global sequence for operations that do not causally depend on each other, and is the root source of the scalability benefits of causal+ or weaker consistency levels. Lloyd et al. showed that causal+ and weaker consistency levels are achievable in *available, low-latency, partition-tolerant, and scalable* (ALPS) systems, while linearizability and sequential consistency are not [38].

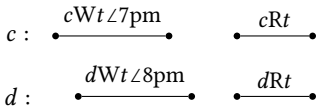
Why Causality. The causal property is desirable in many application scenarios. For example, [38] describes a scenario where client c is sharing a photo with client d by first uploading the photo to an image store s and then adding a reference to the photo to the album a. Client d then checks c's album and, upon seeing a new reference, goes to fetch the referenced photo:



For consistency levels that do not honor causal dependencies, such as per-key sequential consistency or eventual consistency, it is possible for d to observe a new reference out of album a but fail to see the new photo from store s (because $|c\mathrm{W}s\angle\mathrm{photo}| \not\rightsquigarrow |d\mathrm{R}s{:}\mathrm{nil}|$ in the decided ordering). Causal and thus causal+ consistency prevents this type of counter-intuitive phenomenon, because causal dependencies will force $|c\mathrm{W}s\angle\mathrm{photo}| \rightsquigarrow |d\mathrm{R}s{:}\mathrm{photo}|$ since $|c\mathrm{W}a\angle\mathrm{ref}_{\mathrm{photo}}| \rightsquigarrow |d\mathrm{R}a{:}\mathrm{ref}_{\mathrm{photo}}|$.

Why Convergence. Compared to regular causal consistency, causal+ consistency demands a *convergent conflict resolution* mechanism for conflicting values observed. In other words, all read operations that observe the same set of unordered values on an object must resolve into the same return value. Examples of such conflict resolution mechanisms include *last-writer-wins*, *taking-the-max*, and *taking-the-sum*.

Without the convergence guarantee, regular causal consistency is allowed to forever return different values for reads on the same object from different clients. This may be undesirable in many application scenarios. For example, [38] describes a scenario where two clients, c and d, happen to concurrently update the time for a reminder event t:



Original causal consistency may yield the following NPO ordering, letting both c and d falsely believe that their own update is the finalized one, even though they have indeed observed both writes:

$$|c\mathrm{W}t\angle 7\mathrm{pm}| \longrightarrow |c\mathrm{R}t{:}7\mathrm{pm}|$$
$$|d\mathrm{W}t\angle 8\mathrm{pm}| \longrightarrow |d\mathrm{R}t{:}8\mathrm{pm}|$$

Causal+ consistency guarantees that c and d agree on the same time value after they have observed both writes. Assuming a last-writer-wins conflict resolution policy, the service may check the acknowledgment timestamp of both writes and determine that the reduced value should be 8pm:

$$|c\mathrm{W}t\angle 7\mathrm{pm}| \longrightarrow |c\mathrm{R}t{:}f(\{7\mathrm{pm}, 8\mathrm{pm}\}) = 8\mathrm{pm}|$$
$$|d\mathrm{W}t\angle 8\mathrm{pm}| \longrightarrow |d\mathrm{R}t{:}f(\{7\mathrm{pm}, 8\mathrm{pm}\}) = 8\mathrm{pm}|$$

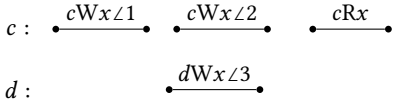
With a service that provides linearizability or sequential consistency, conflicts are avoided altogether by enforcing an SO ordering. However, as previous paragraphs have explained, such protocols lose the benefits of scalable implementation and are not achievable in ALPS systems [38].

4.4 Eventual Consistency

Eventual consistency, as the name suggests, is a consistency level that only requires reads on an object to return a consistent value if no updates are being made to the object [51]. There is no relationship constraint between operations, meaning that any pairs of operations issued by the same client are allowed to get reordered in the final ordering. Eventual consistency is widely adopted in geo-scale systems where the demand for high performance and scalability outweighs the need for data consistency.

Eventual Convergence. Although eventual consistency is sometimes used interchangeably with weak consistency, it does impose one requirement on the service: the decided ordering must be *convergent*. In other words, after all the writers on an object become inactive and after all the writes become visible to readers, reads on the object must all return the same value. In our model, this is captured by the CPO constraint.

For example, given the following physical timeline:



An eventually consistent service is allowed to produce the following CPO ordering:

$$|c\mathrm{W}x\angle 2| \longrightarrow |c\mathrm{W}x\angle 1| \longrightarrow |c\mathrm{R}x{:}\{1,3\}|$$
$$|d\mathrm{W}x\angle 3| \longrightarrow$$

Notice that $|c\mathrm{W}x\angle 2|$ is allowed to be ordered before $|c\mathrm{W}x\angle 1|$, violating the FIFO property. In real implementations, eventually consistent systems typically process every write operation in an asynchronous manner to maximize concurrency. Also notice that $|c\mathrm{R}x{:}\{1,3\}|$ must return a convergent value over the set $\{1,3\}$.

Quiescent Consistency. A closely related, vaguely defined term is *quiescent consistency*, as mentioned in [27]. In a commonly accepted definition of quiescent consistency, special periods of physical time are identified, during which no write operations are happening. Every such contiguous time period is called a *quiescence period*. Each quiescence period orders all operations that are acknowledged ahead of the period before operations that start after the period. With this definition, quiescent consistency is weaker than eventual consistency, because it makes no guarantees if a system-wide quiescence period never appears [50].

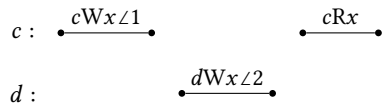
4.5 Other Consistency Levels

In this section, we briefly describe the rest of the selected consistency levels other than the four most common ones. These levels explore different combinations of convergence and relationship constraints to refine the consistency level spectrum.

4.5.1 Regular Sequential Consistency

Helt et al. formalized the notion of *regular sequential consistency* in a recent work [26]. Regular sequential consistency takes the middle ground between linearizability and sequential consistency. It combines the strengths of both by imposing different levels of relationship constraints for *read-only* operations and write operations. Specifically, all write operations (and RMWs) must honor the real-time property (denoted RT-W), while read operations are allowed to travel back in time as long as they still honor causality (denoted CASL-R).

For example, given the following physical timeline:



A service that provides regular sequential consistency may give the following SO ordering, where c's read travels back in time:

$$|c\mathrm{W}x\angle 1| \rightarrow |c\mathrm{R}x{:}1| \rightarrow |d\mathrm{W}x\angle 2|$$

Invariant-equivalence to Linearizability. It is shown in [26] that regular sequential consistency is *invariant-equivalent* to linearizability, meaning that: 1) it is *local* (see Section 4.2 for the definition of implementation locality) and 2) it inherently supports RMW operations thanks to stable ordering of writes. However, it does not guarantee to capture *external causality dependencies*, making it still slightly weaker than linearizability. If external causality is not an issue, a linearizable replicated storage system can seamlessly adopt regular sequential consistency to improve the performance of read-only operations.

The transactional version of this consistency level is *regular sequential serializability* [26], where read-only transactions are allowed to get reordered in the serialization sequence, while all other transactions must honor RT. Similar properties have been exploited in transactional database systems that use certain *Timestamp Ordering* (T/O) optimistic concurrency control mechanisms [53].

4.5.2 Real-time Causal Consistency

Real-time causal consistency is a strengthening of causal+ consistency by bringing back a relaxed version of the real-time property. On top of causal+, real-time causal further requires that: if operation op_1 is acknowledged before the start of op_2 in physical time, then $op_2 \not\rightsquigarrow op_1$ in the final ordering. Notice that this is a weaker constraint than what we have defined as RT, since RT would enforce $op_1 \rightsquigarrow op_2$. We denote this weaker constraint as RT′.

The discussion around causality and convergence is tightly coupled with availability guarantees. Assuming that the system is composed of a set of symmetric message-passing replica nodes, Mahajan et al. have proven in [39] that real-time causal consistency is the strongest possible level that is achievable in an *always-available, one-way convergent* system (which is implied by our definition of *sticky available* in Section 5).

Fork-based Consistency Models. A family of fork-based consistency models has been developed to deal with Byzantine faults in a system containing untrusted replica nodes. For example, a *fork-linearizable* system ensures that if any two replicas have observed different orderings (i.e., *forked* by an adversary, even for one operation), then their writes will never be visible to each other afterward (i.e., they cannot be *joined* again). *Fork causal consistency* is a family of consistency levels that weaken causal consistency to tolerate Byzantine replicas and enforce causal consistency among correct replicas [40]. It is proven that fork causal consistency is unachievable in an always-available, one-way convergent system, while *bounded fork-join causal* consistency, a refinement of fork causal, is achievable in such a system [39].

4.5.3 Causal Consistency

Section 4.3 has explained the most essential pieces of causal and causal+ consistency. To recap, a service that provides *causal consistency* must give an ordering that is NPO and CASL given any physical timeline workload. Such an ordering captures all the potential causality dependencies between operations. Compared to causal+, original causal consistency does not demand *convergent conflict resolution*, meaning that different clients are allowed to forever retrieve different values from reads on the same object.

As mentioned in Section 3.2.2, causal consistency can be defined exactly as the combination of the four session guarantees [11, 30].

4.5.4 Bounded Staleness

Although causal consistency enables the powerful abstraction of causal dependency, it does not provide any guarantee on the "timeliness" of when writes become visible to reads. *Bounded staleness* is a vaguely-defined family of consistency levels that typically strengthen causal consistency by adding *recency* guarantees [42].

Bounded staleness levels put an extra constraint on the *delay* between the acknowledgment of a write by client c on object x and when reads from other clients on x must reflect the effect of the write. The delay constraint may be expressed in the following ways: 1) at most j more write operations by client c, or 2) at most k more updates on object x, or 3) at most a physical time interval t, or 4)

a mixture of the three, e.g., whichever is reached first. We use the name Bounded-CASL to broadly refer to the combination of the CASL relationship guarantee with any delay constraint.

Because of the extra delay constraint, bounded staleness levels are incomparable with both sequential and causal levels, because they both do not express any recency requirements.

4.5.5 PRAM Consistency

Pipeline Random Access Memory (PRAM) consistency [37], or simply *FIFO consistency*, is a weaker consistency level than causal consistency, where causality across clients is not captured. It was originally defined within a shared memory system context. In our model of distributed replication, it is a consistency level that requires NPO and FIFO ordering.

Using the notion of session guarantees, PRAM consistency can be defined exactly as the combination of *Monotonic Writes*, *Monotonic Reads*, and *Read My Writes* [30]. It does not enforce *Writes Follow Reads*, hence not capturing cross-client causality.

Consistent Prefix. The combination of *Monotonic Writes* and *Monotonic Reads* are sometimes referred to as *Consistent Prefix* [42]. This name comes from the fact that, for every writer, all clients will observe a monotonically-growing prefix of its writes.

Although Figure 2 does not include bounded staleness and consistent prefix because of their vague definitions, we can easily derive a strength rank of these two levels w.r.t. causal and PRAM consistency: any Bounded Staleness configuration > Causal > PRAM > Consistent Prefix.

4.5.6 Per-key Sequential Consistency

As Section 4.2 pointed out, sequential consistency is *non-local*, meaning that a protocol that enforces SO and CASL ordering on a per-object basis (termed CASL-per-key) does not necessarily guarantee a global SO and CASL ordering across all operations. In fact, such a protocol implements *per-key sequential consistency*.

This consistency level was first studied in the PNUTS system [14], a highly-concurrent data serving system that provides per-record consistency. However, modern distributed systems typically have complicated client-side logic layered on top of a non-transactional object store, where each client is interested in way more than one object. This makes the object-key-oriented consistency level less appealing than session-oriented causality levels. The photo-album case described in Section 4.3 would be a good example that demonstrates the limitations of per-key sequential consistency.

4.5.7 Weak Consistency

Weak consistency is at the bottom of the consistency level spectrum and is weaker than all other consistency levels. In our model, weak consistency can be defined as enforcing an NPO and None-relationship ordering. It can simply be interpreted as "not providing any consistency guarantees at all". Note that this level is irrelevant to *weak ordering* in shared memory consistency [29, 44].

4.5.8 Mixed/Hierarchical Consistency Levels

So far, we have assumed a single conceptual storage service without making any assumptions on the internal implementation of the service. Real distributed systems may, however, contain multiple layers or scopes of sub-services, each providing a different consistency level semantic. For example, Cosmos DB [42] provides a stronger consistency guarantee for clients within the same *region* than those distributed across multiple regions, effectively exposing a 2-layer consistency model. Given the implementation details of a system, we can always refine the consistency level spectrum and define mixed or hierarchical consistency levels that are composed of multiple basic levels.

Yu and Vahdat [52] proposed a continuous consistency model for replicated services, where consistency is defined as a 3-tuple, (*numerical error*, *order error*, and *staleness*), named a *conit*. This leads to a fairly fine-grained consistency spectrum and allows applications to dynamically balance consistency and performance.

4.5.9 Memory Consistency Models

Distributed replication consistency is tightly related to early works in multiprocessor shared memory consistency. Hill defined *hardware memory consistency model* as the interface contract for memory in a shared memory multiprocessor, where instructions may be executed out-of-order [29]. Many of the consistency levels described in this report originated from multiprocessor shared memory consistency models. The three primary examples are sequential consistency, which was first defined by Lamport in [32], causal consistency, which was adapted from causal memory [3], and PRAM consistency, which was adapted from FIFO processor consistency [2]. Other relaxed memory consistency models and techniques, such as *weak ordering*, *acquire/release consistency*, *entry consistency*, *cache coherence*, and *memory fences/barriers* [29, 44], are out of the scope of this report.

5 AVAILABILITY GUARANTEES

Besides consistency, *availability* is also an important part of the interface contract between a distributed storage service and clients. Availability is not implementation-oblivious; the meaning of fault-tolerance and availability can only be defined given a specific system model. In this section, we consider a simple system model of symmetric replicas and analyze the best possible availability guarantee that each consistency level can provide in such a system.

5.1 Symmetric Replicas System Model

We consider a fault-tolerant system implementation of the object store service composed of a set of *symmetric replica servers*, similar to what Figure 4 depicts. Each replica node holds a complete copy of all objects and can communicate with any other replica through message-passing over the network. Clients establish connections to one (or more) replica nodes, issue operations, and wait for acknowledgments.

Data Partitioning. Since we only consider non-transactional workloads, this symmetric model can be easily extended to incorporate *data partitioning* (or called *partial replication*), where each node is responsible for a subset of objects. For each object, only the set of nodes that hold the object is under consideration for availability of that object.

Client-side Caching. A client may act as a partial replica server by doing *client-side coherent caching* w.r.t. the consistency level for

its reads and writes [8, 48]. In this case, we count the client itself as a valid partial replica. We leave a more detailed discussion on client-side caching as future work.

5.2 Meaning of Availability

We say a system of symmetric replicas provides high **availability** if, in the presence of arbitrarily long network partitions between arbitrary replicas, every client that can establish connection to one (or a specific set of) non-failing replica(s) of an object eventually gets acknowledged for all operations it issues on that object [7].

Failure Model. We allow the following two types of failures:

- *Fail-stop*: a replica server may crash and stop responding to any incoming messages at any time point.
- *Network partition*: the network channel between any two groups of replicas may break. A failed replica is indistinguishable to its peers from a completely network-partitioned replica or a replica that is just running very slowly.

We do *not* consider *Byzantine* faults [13, 17, 39], where a server node may maliciously send out inconsistent information, or where the content of network messages may get tampered with. We leave the integration of Byzantine faults as future work.

Availability Levels. We consider three coarsely-defined levels of availability guarantees in the presence of arbitrarily long network partitions between arbitrary pairs of replicas [30]:

- *Totally available*: every client that can contact at least one non-failing replica of an object eventually receives responses that honor the consistency level for operations on that object.
- *Sticky available*: a client maintains *stickiness* if it keeps contacting the same replica for each of its operations on an object. The system is sticky available if every client that sticks to a non-failing replica of an object eventually receives responses that honor the consistency level for operations on that object.
- *Weakly available*: the system does not guarantee progress under arbitrary network partitions.

Note that the "weakly available" category can be further decomposed into finer-grained, protocol-specific availability levels if we can bound the number of failures to a certain quantity. For example, most state machine replication protocols are available when at least a majority of nodes are healthy and can communicate with each other. We mention such availability implications in Section 6. Also, extra care needs to be taken to define reasonable transactional availability guarantees [7], which is out of the scope of this report.

5.3 Availability Upper Bounds

Brewer's famous *CAP theorem* states that a distributed system cannot achieve Consistency, Availability, and network Partition-tolerance all at the same time [10]. This informal description is often misunderstood and taken in an overly restrictive form. A more precise statement would be that a distributed system cannot achieve linearizability, total/sticky availability, and network partition-tolerance all at the same time. This statement has been proven by Gilbert and Lynch in [19].

By relaxing linearizability to weaker consistency levels, it is often (but not always) possible to derive a replication protocol that guarantees sticky or even total availability under arbitrary network partitions. Table 2 lists the availability upper bound of each of the selected consistency levels.

Consistency Level	Availability Upper Bound
Linearizability	Weakly available
Regular Sequential	
Sequential	
Bounded Staleness	
Real-time Causal	Sticky available
Causal+	
Causal	
PRAM	
Per-key Sequential	
Session Guarantees:	
Read My Writes	
Writes Follow Reads	Totally available
Monotonic Reads	
Monotonic Writes	
Eventual	
Weak	

Table 2: Availability Upper Bound of Consistency Levels.

Most of these availability bounds have been proven in previous literature [7, 39]. Linearizability, regular sequential consistency, and bounded staleness are obviously weakly available because of the RT constraint or the delay constraint: clients connecting to servers separated on opposite sides of a network partition have no way of knowing the acknowledgment time of operations made on the other side, unless operations on that side are blocked indefinitely. Sequential consistency cannot be sticky available because of its non-locality, as counter-examples similar to the one presented in Section 4.2 can be constructed; in contrast, per-key sequential is sticky available. Bailis et al. have proven that the writes follow reads, monotonic reads, and monotonic writes session guarantees are totally available, while read my writes requires stickiness [7]. Causal and PRAM consistency are therefore both sticky available. Mahajan et al. have proven that real-time causal is as available as causal consistency (given one-way convergence, which is assumed in our model) [39]. Causal+ is also sticky available following this result. Eventual and weak consistency are both totally available: clients can make progress on any live server.

Limitations. The availability upper bounds presented here are rather coarse-grained and do not capture everything about availability. First, they say nothing about *recency* guarantees, i.e., how stale are read results allowed to be. For example, although causal consistency is sticky available, a network partition may indefinitely prevent writes made on one side from being visible to readers on the other side. Bounded staleness levels would thus all be weakly available in our definition. Second, these availability bounds also do not consider *partial* network partitions, where certain pairs nodes cannot directly communicate with each other, but some indirect multi-hop paths are still available. Alfatafta et al. discussed partial network partitions and mechanisms to exploit available indirect paths in [4].

6 PROTOCOL EXAMPLES

TODO: expand this section...

- Consensus primitives: Paxos, Speculative-Paxos, Ben-Or algorithm, randomized consensus, ABD shared register, CASPaxos, weighted voting, quorum consensus
- Linearizable SMR protocols & systems:
  - Leader-based protocols atop Paxos: Multi-Paxos, Cheap-Paxos, Fast-Paxos, Vertical-Paxos, FPaxos, Generalized-Paxos, Viewstamped replication, Raft, Paxos summary
  - Multi-leader protocols (optimizing latency/contention): Mencius, EPaxos, WPaxos, AllConcur, RIFL, Atlas
  - Chain-topology protocols (optimizing throughput): Chain replication, CRAQ, Ring-Paxos, Multi-Ring-Paxos, ChainPaxos
  - SMR with Error Correction Code (ECC) techniques: RS-Paxos, CRaft
  - SMR with randomized consensus: Rabia
  - Scalability-focused optimizations: HP-SMR, S-SMR, DS-SMR, P-SMR, OP-SMR, Insanely-scalable SMR, Compartmentalization
  - Durability-oriented optimizations: Durable SMR, SDPaxos, Consistency-aware durability, Skyros Nil-externality
  - Fail-slow tolerance enhancements: SDPaxos, Copilots
  - Bypassing consensus by exploiting hardware assumptions: NOPaxos, FLAIR
- Causal consistency protocols & systems: Lazy replication, Bayou, COPS, ChainReaction, Bolt-on causal consistency, Occult, TCC-Store, UniStore
- Deployed systems:
  - Coordination service & Master replication: Chubby, ZooKeeper, Niobe, GFS
  - Industrial transactional database systems: Spanner, F1, FaRM, FaRMv2, Aurora, TiDB, FoundationDB, CockroachDB
  - Systems with weaker/continuous consistency guarantees: PRACTI, TACT, PNUTS, Dynamo, Cassandra, DynamoDB
  - Deployed system measurement & surveys: Consensus in the cloud, Paxi, Cloud OLTP eval
- Programming abstractions, tools, & services: CORFU, Tango (shared log abstraction), uKharon (membership service), Nifty (masking partial network partitioning), virtual synchrony, Sinfonia, Derecho, DepFast (general libraries)

REFERENCES

[1] S.V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: a tutorial. *Computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

[2] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. 1993. The power of processor consistency. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993 (Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993)*. Association for Computing Machinery, Inc, 251–260. https://doi.org/10.1145/165231.165264 Funding Information: * Thk work was supported in part by the National Science Foundation under grants CCR-8619S86, CCR-8909663j and CCR-9106627. Authors' address: College of Computing, Georgia Institute of Technology Atlanta, Georgia 30332-0280. t Tlds author was supported in part by a scholarship Hariri Foundation. Publisher Copyright: © 1993 ACM.; 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993 ; Conference date: 30-06-1993 Through 02-07-1993.

[3] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (01 Mar 1995), 37–49. https://doi.org/10.1007/BF01784241

[4] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 20, 18 pages.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (jan 1995), 124–142. https://doi.org/10.1145/200836.200869

[6] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (may 1994), 91–122. https://doi.org/10.1145/176575.176576

[7] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (nov 2013), 181–192. https://doi.org/10.14778/2732232.2732237

[8] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

[9] Alysson Bessani, Paulo Sousa, and Miguel Correia. 2010. Active Quorum Systems. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability* (Vancouver, BC, Canada) *(HotDep'10)*. USENIX Association, USA, 1–8.

[10] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '00)*. Association for Computing Machinery, New York, NY, USA, 7. https://doi.org/10.1145/343477.343502

[11] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. 2004. From session causality to causal consistency. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings.* 152–158. https://doi.org/10.1109/EMPDP.2004.1271440

[12] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 591–618.

[13] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[14] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1 (2008), 1277–1288.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson 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. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. https://doi.org/10.1145/2491245

[16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (apr 1985), 374–382. https://doi.org/10.1145/3149.214121

[18] David Kenneth Gifford. 1981. *Information Storage in a Decentralized Computer System.* Ph. D. Dissertation. Stanford, CA, USA. AAI8124072.

[19] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (jun 2002), 51–59. https://doi.org/10.1145/564585.564601

[20] Alexey Gotsman, Hongseok Yang, Marek Zawirski, and Sebastian Burckhardt. 2014. Replicated Data Types: Specification, Verification, Optimality. In *41st Symposium on Principles of Programming Languages (POPL)* (41st symposium on principles of programming languages (popl) ed.). ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/replicated-data-types-specification-verification-optimality/

[21] V. Gramoli, N. Nicolaou, and A.A. Schwarzmann. 2021. *Consistent Distributed Storage.* Morgan & Claypool Publishers. https://books.google.com/books?id=bWiKzgEACAAJ

[22] Jim Gray. 1985. *Why Do Computers Stop and What Can Be Done About It?* HP. Retrieved 2023-01-05 from https://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf

[23] Jim Gray. 1988. *The Transaction Concept: Virtues and Limitations.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 140–150.

[24] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. 2022. Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation. *Proc. VLDB Endow.* 16, 2 (nov 2022), 379–392. https://doi.org/10.14778/3565816.3565837

[25] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (dec 1983), 287–317. https://doi.org/10.1145/289.291

[26] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 163–179. https://doi.org/10.1145/3477132.3483566

[27] M. Herlihy and N. Shavit. 2011. *The Art of Multiprocessor Programming*. Elsevier Science. https://books.google.com/books?id=pFSwuqtJgxYC

[28] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

[29] M.D. Hill. 1998. Multiprocessors should support simple memory consistency models. *Computer* 31, 8 (1998), 28–34. https://doi.org/10.1109/2.707614

[30] JEPSEN. 2016. *Jepsen Consistency Models*. JEPSEN. Retrieved 2023-01-05 from https://jepsen.io/consistency

[31] Donald Kossmann, Tim Kraska, and Simon Loesing. 2010. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 579–590. https://doi.org/10.1145/1807167.1807231

[32] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[33] Leslie Lamport. 1978. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks* 2 (August 1978), 95–114. https://www.microsoft.com/en-us/research/publication/implementation-reliable-distributed-multiprocess-systems/

[34] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[35] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].* (May 1998). https://www.microsoft.com/en-us/research/publication/part-time-parliament/ ACM SIGOPS Hall of Fame Award in 2012.

[36] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[37] Richard J. Lipton and Jonathan Sandberg. 1988. PRAM: A Scalable Shared Memory. (08 1988).

[38] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[39] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. 2012. Consistency, Availability, and Convergence. (05 2012).

[40] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.* 29, 4, Article 12 (dec 2011), 38 pages. https://doi.org/10.1145/2063509.2063512

[41] Jenny Mankin. 2007. Memory Consistency Models: A Survey in Past and Present Research. (2007).

[42] Microsoft. 2022. *Consistency levels in Azure Cosmos DB*. Microsoft. Retrieved 2023-01-06 from https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels

[43] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (dec 1986), 378–396. https://doi.org/10.1145/7239.7266

[44] David Mosberger. 1993. Memory Consistency Models. *SIGOPS Oper. Syst. Rev.* 27, 1 (jan 1993), 18–26. https://doi.org/10.1145/160551.160553

[45] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[46] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (dec 1990), 299–319. https://doi.org/10.1145/98163.98167

[47] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.

[48] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems* (Autin, Texas, USA) *(PDIS '94)*. IEEE Computer Society Press, Washington, DC, USA, 140–150.

[49] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[50] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (jun 2016), 34 pages. https://doi.org/10.1145/2926965

[51] Werner Vogels. 2008. Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs Between Consistency and Availability. *Queue* 6, 6 (oct 2008), 14–19. https://doi.org/10.1145/1466443.1466448

[52] Haifeng Yu and Amin Vahdat. 2002. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *ACM Trans. Comput. Syst.* 20, 3 (aug 2002), 239–282. https://doi.org/10.1145/566340.566342

[53] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935