# Consistency, Availability, and Convergence

Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin

**Abstract**

We examine the limits of consistency in highly available and fault-tolerant distributed storage systems. We introduce a new property—*convergence*—to explore the these limits in a useful manner. Like consistency and availability, convergence formalizes a fundamental requirement of a storage system: writes by one correct node must eventually become observable to other connected correct nodes. Using convergence as our driving force, we make two additional contributions. First, we close the gap between what is known to be impossible (i.e. the consistency, availability, and partition-tolerance theorem) and known systems that are highly-available but that provide weaker consistency such as causal. Specifically, in an asynchronous system, we show that *natural causal* consistency, a strengthening of causal consistency that respects the real-time ordering of operations, provides a tight bound on consistency semantics that can be enforced without compromising availability and convergence. In an asynchronous system with Byzantine-failures, we show that it is impossible to implement many of the recently introduced *forking*-based consistency semantics without sacrificing either availability or convergence. Finally, we show that it is not necessary to compromise availability or convergence by showing that there exist practically useful semantics that are enforceable by available, convergent, and Byzantine-fault tolerant systems.

## 1 Introduction

This paper examines the limits of consistency in highly available and fault-tolerant distributed storage systems. The tradeoffs between consistency and availability [6, 24, 38] have been widely used in guiding system design. The consistency, availability, partition-tolerance (CAP) theorem [24] is often cited as the reason why systems designed for high availability, such as Dynamo [19] and Cassandra [13], choose to enforce the very weak *eventual consistency* [56] semantics [13, 19, 56]. Conversely, the CAP theorem has guided designers of Amazon SimpleDB [4] to renounce high availability when strong consistency is required [15].

This paper extends the current understanding of these tradeoffs in two ways. First, it strengthens the CAP theorem by characterizing precisely the limits of consistency that can be achieved, not just what cannot be. Second, it moves beyond the confines of omission failures and rigorously explores for the first time the availability vs. consistency tradeoff in environments where nodes can be Byzantine. Our results can help the designers of highly available systems like Dynamo and Cassandra understand the extent by which they can offer stronger consistency guarantees. Similarly, in Byzantine-fault tolerant systems like Depot [41], our results can guide the choice of consistency that can be achieved without compromising availability.

A central challenge in trying to identify the limits of achievable consistency is that consistency can be "artificially" strengthened by ruling out certain executions. For example, the idea that something like causal consistency is the best one can hope for without compromising availability has been around for a long time, but a system in which reads by a node return only the writes issued by that node can be highly available while offering a consistency that is stronger than causal. Yet, this semantic feels artificial because it fails to deliver on an intuitively basic requirement: nodes should be able to observe another node's writes.

We formalize this intuition by introducing a new property: *convergence*. Convergence addresses the fundamental demand that writes by one correct node must eventually become observable to other connected correct nodes. By insisting on consistency semantics that admit convergent implementations, we can rule out

extremely strong, but trivial and practically useless consistency semantics in a principled way. Similarly, we can use convergence to rule out useless implementations of otherwise strong and useful consistency semantics. For example, we can use convergence to rule out an implementation of sequential consistency that avoids propagating writes by forcing nodes to return stale writes, as long as doing so does not violate sequential consistency.

Enforcing an explicit convergence requirement leads us to two contributions. First, in asynchronous systems with unreliable networks we close the gap between what is known to be impossible (i.e. CAP [17, 24, 53]) and known systems that are highly-available but provide weaker consistency such as causal [2, 8, 25, 32, 39, 49, 52]. In particular, we show that no consistency stronger than *natural causal* consistency (NC), a strengthening of causal consistency that respects the real-time ordering of operations, can be provided in an *always-available* and *one-way convergent* system. An *always-available* system allows reads and writes to complete regardless of which messages are lost and which nodes fail. A *one-way convergent* system guarantees that if node $p$ can receive messages from node $q$, then eventually $p$'s state reflects updates known to $q$. We also show that NC consistency is achievable using an always-available and one-way convergent implementation.

Second, in systems that can suffer Byzantine failures, we show that *fork-causal* consistency [40] cannot be provided in an always-available and one-way convergent system if nodes can be Byzantine. Notice that this result rules out always-available and one-way convergent implementations of many recently proposed *forking consistency* semantics [10, 12, 36, 37, 42, 48]. In these systems, a faulty node can cause correct nodes to become *permanently partitioned* in that *forked* correct nodes cannot observe each other's writes. Using this result, we argue that the design of new consistency semantics and systems should be done with an eye for convergence. We further show that indeed, convergent and practically useful consistency semantics (such as Depot's *FJC* consistency [41]) can be provided without compromising availability or tolerance to Byzantine faults.

In the rest of this paper, we first define the CAC (consistency, availability, and convergence) properties (§3). Then we explore the CAC trade-offs in omission- and Byzantine-fault tolerant systems (§4 and §5). Finally we discuss the implications of CAC theory and results (§6), summarize related work (§7), and conclude (§8).

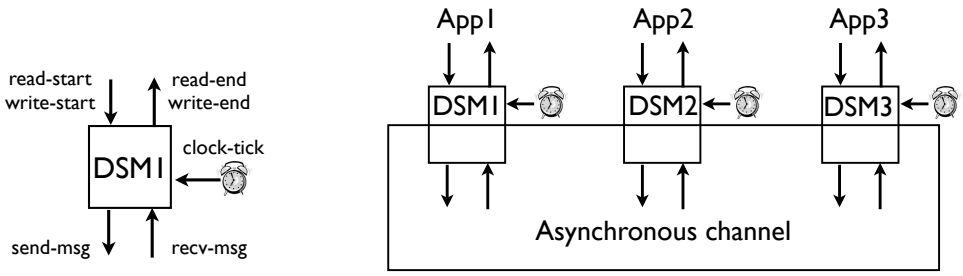
## 2 Framework and assumptions

Figure 1 illustrates the basic components of our framework. In our framework, a *storage implementation*, also referred to as a node, processes an *application's* request to read and write *objects* that are replicated across a set of *distributed storage implementations*. A storage implementations uses a *local-clock* to obtain the timing information and the *channel* to exchange messages with other storage implementations.

We assume an asynchronous system and we model this assumption in two ways. First, the channel models a faulty network which is permitted to reorder, drop, or delay messages. Second, the storage implementation does not have access to any global or real-time clock and each storage implementation's local-clock can run arbitrarily fast or slow compared to the similar local clocks at other storage implementations. Third, an input provided by the application can take an arbitrary amount of time to reach the storage implementation and vice-versa. Figure 1(a) shows a storage implementation with its input and output events.

In addition to asynchrony, nodes in our framework can operate in a variety of modes that we describe next.

- *No failures.* In this mode, nodes do not fail. Note that the network may still drop messages sent by nodes thereby preventing them from communicating.
- *Omission failures.* In this mode, nodes can fail by crashing or they may fail to send or receive one or more messages.




(a) A Storage implementation (b) A distributed storage implementation

Figure 1: (a) A storage implementation and (b) a distributed storage implementation (b) constructed by connecting several storage implementations through an asynchronous channel. Note that the asynchronous channel also controls the operation of local clocks at the various implementations.

- *Byzantine failures.* In this mode, the failed nodes can behave arbitrarily [35].

Because we assume a faulty network that can drop messages, omission failures do not introduce any new challenge. Therefore, all of our results for non-Byzantine failures apply to both omission failure and no failure modes. Unless otherwise stated, we are considering the no failure/omission failure mode. We explicitly specify the failure mode when we consider Byzantine failures.

For ease of explanation, we abstract the environment of a storage implementation—application, channel, and local-clock—using a graph called *environment graph*. An environment graph precisely captures the inputs to a particular storage implementation enabling us to compare the state of different storage implementations using their respective environments. Using an environment graph, we construct the *run* of a distributed storage implementation, consisting of multiple storage implementations. A run shows when various events were issued and which responses (if any) were generated by the storage implementations. Finally, we define an *execution* in our framework, that we use later to describe various consistency semantics. We expand on our framework below.

## 2.1 Storage implementation

A *storage implementation* is a deterministic I/O automaton with input events `read-start (objId,uid)`, `write-start (objId,uid,value)`, `clock-tick ()`, `recv-msg (nodeId,m)` and output events `read-complete (uid,wl)`, `write-complete (uid)`, `send-msg (nodeId,m)` where `oId` denotes the object identifier, `value` denotes the value being written, `uid` denotes unique identifier assigned by the application to each read and write operation, `nodeId` denotes the unique identifier of the storage implementation, `m` denotes content of the message, `wl` denotes the writeList which is a set of tuples of the form `(uid,d)` indicating that multiple values can be returned on a read [19, 41]. Reads can also return $\perp$ (the implementation returns an empty `writeList`), if no appropriate write is found.

As we discuss below (§3.1), allowing a read to return multiple values provides a clean way to handle *logically concurrent updates* without making restrictive assumptions about conflict resolution. Various conflict resolution strategies can then be layered above our implementation.

We assume that the automaton implements a *classical* memory system whose logical state is not changed by reads.

We assume that an implementation orders operations and is oblivious to the actual values being written to objects and that reads return values written by write operations. Therefore to avoid ambiguity, when discussing the result of a read, our formalism focuses on the write operation that wrote the value that the read returns [22].

A *distributed storage implementation* consists of a collection of storage implementations that communi-

cate through an asynchronous *channel*. Figure 1 shows a storage implementation and a distributed storage implementation connected through an asynchronous channel.

## 2.2 Application

An application issues `read-start` and `write-start` input events to a particular storage implementation and gets `read-end` and `write-end` outputs as responses. We assume that events issued by an application are asynchronously received by the store at a later time. Similarly, events produced by the storage implementation are received by the application at a later time.

We further assume that the workload issued by an application is not restricted in any way. Any application can issue reads/writes to any object. However, we do assume that each application issues at most one outstanding operation at a time. We make this assumption for simplicity but we conjecture that issuing multiple outstanding requests will not change the fundamental tradeoffs that our results present.

## 2.3 Channel

The channel models the network in our framework. Because we assume an asynchronous model with an unreliable network, messages may be dropped, reordered, or delayed for an arbitrary but finite duration by the channel. The channel controls the behavior of the network by issuing `recv-msg` events to various storage implementations and receiving `send-msg` events from them.

## 2.4 Local-clock

Each storage implementation has access to an unsynchronized local-clock. The local-clock is the only source of timing information available to a storage implementation; the implementation does not have access to the real-time clock. The local-clock issues `clock-tick` events to the implementation which can be used by the implementation to schedule or trigger other events. We assume that the local-clocks at different node can run arbitrarily fast or slow compared to each other.

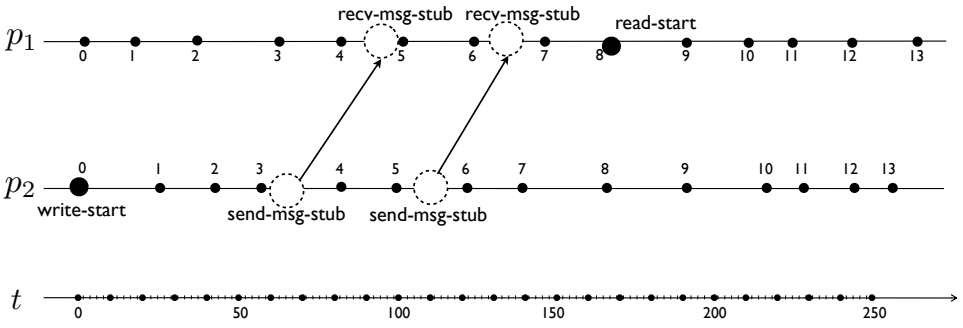
In contrast, to reason about consistency, we assume that an oracle has access to a real-time clock which it can use to identify the real-time(s) at which various events occur. As explained later in Section 2.6, these real-time(s) can be used to specify the consistency condition. We assume that the real-time clock accessible to the oracle has infinite precision. We do not care about the precision of of the local-clock as our proofs do not rely on them.

Finally, we emphasize that storage implementations cannot see the real-time reported by the real-time clock; they can only see their respective local-clock time. We do not guarantee any correspondence between the local-clock time and the real-time owing to the asynchronous nature of our system.

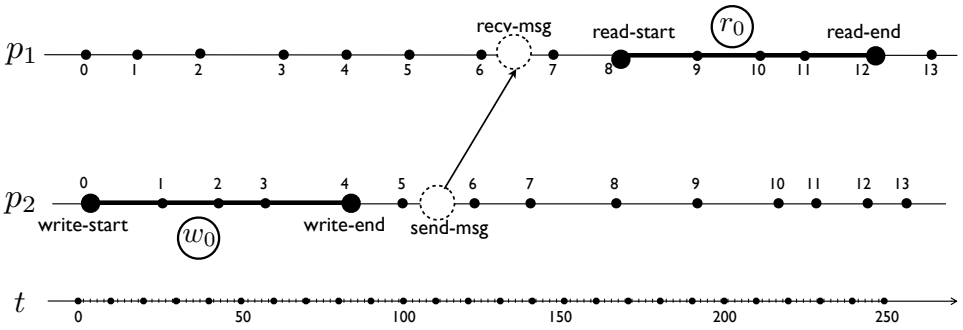
## 2.5 Environment

We describe the environmental conditions (behavior of the channel, local-clocks, and applications) through an *environment graph*. In particular, an environment graph specifies (1) the behavior of the local clocks by indicating when `clock-tick` events are issued, (2) the behavior of the application by indicating when `read-start and write-start` events are issued, and (3) the behavior of the asynchronous channel by indicating which transmitted messages are received and when, and which are dropped.

There is an issue when specifying the behavior of a channel—we don't know a priori when and which messages will be sent by a storage implementation. Therefore, we label messages using a combination of the `clock-tick` value and an identifier of the storage implementation that produced the message. For example, if the channel wants to deliver the message that was produced by storage implementation $p_1$ at local time $t_1$ for storage implementation $p_2$ at local time $t_2$, then the corresponding environment graph will contain a `send-msg-stub` at $p_1$ at local time $t_1$ connected using a directed edge to a `recv-msg-stub` at $p_2$ at local time $t_2$.




(a) An environment graph




(b) A run graph

Figure 2: (a) A sample environment graph and (b) a run of a causally consistent implementation constructed using the environment graph in (a). In the environment graph, the channel allows the delivery of messages sent from $p_2$ to $p_1$ at local-clock time(s) 3 and 5 as indicated by the `send-msg-stub` vertices. As the corresponding `recv-msg-stub` vertices indicate, these messages are delivered at local-clock time(s) 4 and 6 respectively. All other messages are dropped as indicated by the lack of any other `send-msg-stub` and `recv-msg-stub` vertices. The `local-clock` events in the graph depicts how the local-clocks at different nodes can run at different speeds and may differ from the real-time clock. The environment graph also indicates the application events: $p_2$ issues a write operation $w_0$ as indicated by the `write-start` event and $p_1$ issues a read operation $r_0$ as indicated by the `read-start` event. All operations operate on the same object $X$, which is not shown in the figure for brevity. In the corresponding run, node $p_2$'s write $w_2$ completes as indicated by the `write-end` at local-clock time 4. $p_2$ then sends an update to node $p_1$ at time 5 which $p_1$ receives and applies. $p_1$'s later issues a read $r_0$ which also completes as shown by the corresponding `read-end` event at local-clock time 12. Note that the unused `send-msg-stub` and `recv-msg-stub` vertices of an environment graph are removed from the corresponding run graph.

$$\textcircled{r_0}\ (p_1, X, w_0, p_1 : 8, 165, 250)$$

$$\textcircled{w_0}\ (p_2, X, \text{“}A\text{”}, p_2 : 0, 0, 85)$$

Figure 3: The execution corresponding to the run shown in Figure 2(b). The execution has only two vertices corresponding to the write operation by $p_2$ and the read operation by $p_1$. Each of these vertices contain the relevant details from the corresponding run vertices.

An environment graph is a directed acyclic graph with `read-start`, `write-start`, `clock-tick`, `send-msg-stub`, and `recv-msg-stub` vertices. It contains edges connecting successive events at each storage implementation and edges connecting the send of a message to its receive if the message was successfully delivered. Thus, an environment graph provides the complete specification of inputs to a distributed storage implementation, and the state of a distributed storage implementation is a function of the environment graph that defines its input. Figure 2(a) illustrates a sample environment graph.

### 2.6 Run

Generating inputs to a distributed storage implementation using an environment graph and obtaining the output produces a *run* of the distributed storage implementation. We represent a run using another directed acyclic graph that is similar to an environment graph, but augmented to include `write-end` and `read-end` vertices and `send-msg` and `recv-msg` vertices in place of `send-msg-stub` and `recv-msg-stub` vertices. Figure 2(b) illustrates the run graph produced using a causally consistent implementation [3] and the environment graph shown in Figure 2(a).

For vertices $u$ and $v$, we use the notation $u \prec_\rho v$ to denote that there exists a path from $u$ to $v$ in the run $\rho$. For operations $u$ and $v$, we use the notation $u \prec_\rho v$ to denote that there exists a path from the end of operation $u$ to the start of operation $v$ in the run $\rho$

### 2.7 Execution

We represent the application's view of a run of distributed storage implementation using an *execution*. An *execution* consists of a set of read and write operations. Intuitively, an execution eliminates the details of a run that are not needed for defining consistency while retaining the essential details. An execution is represented using a set of read and write operations that carry the following fields:

```
Read = (nodeId, objId, wl, uid, startTime, endTime)
Write = (nodeId, objId, value, uid, startTime, endTime)
```

Most of the fields above are taken from the fields of the start and end event of the corresponding operation in the run. For example, the `nodeId` corresponds to the identifier of the storage implementation at which the corresponding start and end events occur, *startTime* corresponds to the real-time at which the *read-start*/*write-start* event occurs in the run.

The real-times (`startTime and endTime`) of an operation reflect when that operation is issued at the application and when the operation is reported to be complete to the application respectively. Because propagation of an application's read/write request to the storage implementation and vice-versa can take some positive, non-zero time depending on the scheduling and propagation delays, we require that for each operation $o$, $o_{startTime} < o_{storeStartTime} < o_{storeEndTime} < o_{endTime}$, where $o_{storeStartTime}$ and $o_{storeEndTime}$ denote the real-time at which the request and response event are processed at the storage implementation. These real-time assignments are useful for characterizing semantics like linearizability where the real-time ordering of operations must be respected [29]. We require that each operation takes positive, non-zero time to complete.

# 3 Consistency, Availability, and Convergence (CAC)

We next introduce the CAC properties—consistency, availability, and convergence—using the framework introduced in the previous section. Consistency and availability are well-studied properties. We formalize existing intuition about these properties in a relatively standard way. For convergence, we provide a new formalism and new definitions because it is a novel property that has not been described in prior work.

## 3.1 Consistency

Consistency restricts the order in which reads and writes appear to occur. Formally, a consistency semantic is a *test* on an *execution*—if an execution $e$ passes the test for consistency $C$, we say $e$ is *C-consistent*.

We say that a consistency semantics $C_s$ is *stronger* than another consistency semantics $C_w$ if the set of executions accepted by $C_s$ is a strict subset of the set of executions accepted by $C_w$. We say that $C_s$ is *equivalent* to $C_w$ if they admit the same set of executions ($C_s = C_w$). We say that two non-equivalent consistency semantics are *incomparable* if neither of them is stronger. Intuitively, a stronger consistency semantics permits fewer behaviors and therefore, is easier to reason about. Hence, a stronger consistency semantics is preferable over a weaker one.

We say that an implementation $I$ *enforces* a consistency semantics $C$ if any execution $e$ produced by $I$ is accepted by $C$ regardless of the *workload*—the sequences of read and write operations issued by nodes—and the *environment*—which messages are dropped, delayed, duplicated, and delivered and how quickly local clocks run relative to each other and to global time.

We next define the notation that we use to formally define consistency semantics and to prove our results.

DEFINITION 3.1. **Precedes.** *Let $H$ be directed acyclic graph. We say that the graph $H$ imposes a partial order that we denote by $\prec_H$. We say that $v$* precedes *$u$ in graph $H$ (denoted by $v \prec_H u$) if there exists a path from $v$ to $u$ in $H$.*

DEFINITION 3.2. **Concurrent vertices.** *We say two vertices $u$ and $v$ are concurrent in $H$ (denoted by $v||_H u$) if neither of them precedes the other in $H$.*

**Causal consistency.** Causal consistency [2, 31] captures the requirement that operations that depend on each other are observed by nodes in the same order. The following definition states this requirement formally.

DEFINITION 3.3. *An execution $e$ is* causally consistent *if there exists a directed acyclic graph $G$, called a happens before (HB) graph, containing a read/write vertex for each read/write operation in $e$ such that $G$ satisfies the following consistency check:*

**C1** Serial ordering at each node. *The ordering of operations at any node is reflected in $G$. Specifically, if $v$ and $v'$ are vertices corresponding to operations by the same node, then $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.*

**C2** A read returns the latest preceding concurrent writes. *For any vertex $r$ corresponding to a read operation of object* objId*, $r$'s* writeList *$wl$ contains all writes $w$ of* objId *that precede $r$ in $G$ and that have not been overwritten by another write of* objId *that both follows $w$ and precedes $r$:*

$$w \in r.wl \Leftrightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : (w \prec_G w' \prec_G r \wedge w'.objId = r.objId))$$

Note that our definition *separates consistency from conflict resolution* for dealing with conflicting, concurrent writes to the same object. Some causally consistent systems employ a conflict resolution algorithm to pick a winner (e.g., highest-node-ID wins) or to merge conflicting writes (e.g., allowing a directory to include all differently-named files concurrently created in the directory) [2, 8, 25, 30, 32, 39, 55]. Instead, our definition models the fundamental causal-ordering abstraction shared by all these approaches without attempting to impose a particular conflict resolution strategy. In our approach, logically concurrent writes

are returned to the reader, who can then apply any standard or application-specific conflict resolver to pick a winner, or merge concurrent writes [8, 19, 41]. This logic is encoded in an application-specific conflict-handler layered over the consistency algorithm.

We further note that this formulation of causal consistency is incomparable with Ahamad el al's [3] formulation for causal consistency. Our definition separates consistency from conflict resolution because the former orders a system's reads and writes [1] and may identify a set of writes as logically concurrent [33] while the latter defines how the system handles concurrent writes of the same object [30, 51, 55]. This separation collapses many otherwise incomparable variations of causal consistency into a single consistency semantic that *exposes* concurrency but does not try to resolve it. This separation makes our formulation more flexible as it can be combined with a suitable conflict resolver to create the same behavior as Ahamad's causal consistency. Similarly, combined with a suitable conflict resolver, our formulation can simulate a consistency semantics strictly stronger than Ahamad's causal consistency by disallowing certain conflict resolution strategies.

Finally, we emphasize that a HB graph is a hypothetical construct created after-the-fact, while performing the consistency checks. An implementation need not maintain any data structure to store or update a HB graph. Instead, an execution is considered $C$-consistent if an oracle, with the knowledge of the entire execution, can produce a $C$-consistent HB graph for that execution.

### 3.2 Availability

Availability, informally, refers to an implementation's ability to ensure that read and write operations complete. The availability of an implementation is formally defined by describing the environmental conditions (network and local-clocks) under which all issued operations complete. An implementation $I$ is *available* under an environment graph $\psi$ if $\psi$ produces a *available* run on $I$. In an available run, each `read-start` input event has a corresponding `read-end` output event and each `write-start` input event has a corresponding `write-end` output event.

Now, we can compare the availability of two implementations. An implementation $I$ is *more available* than implementation $I'$ if the set of environment graphs under which $I$ is available is a strict superset of the set of environment graphs under which $I'$ is available. Intuitively, a more available implementation is preferable over a less available one because the more available implementation can serve reads and writes under a more diverse set of conditions.

DEFINITION 3.4. *An implementation is* always-available *if, for any workload, all reads and writes can eventually complete regardless of which messages are lost and which nodes can communicate. In terms of our environment graph abstraction, we can say that an implementation $I$ is* always available *if any environment graph can can be extended to form an available environment graph for $I$ by adding only the* `clock-tick` *events at each node. An implementation in which one storage implementation needs to communicate with another before processing some read or write request is not always-available; there exist environments in which a storage implementation cannot complete such read or write requests because it is partitioned from others.*

### 3.3 Convergence

Informally, *convergence* refers to an implementation's ability to ensure that writes issued by one node are observed by others. Convergence can be formally defined by describing the set of environmental conditions (network and local clocks) under which nodes can observe each other's writes.

We formalize convergence to better explore the fundamental tradeoff between safety (consistency) and liveness (availability and convergence). Absent convergence conditions, both strong consistency and high availability can be simultaneously achieved. In particular, a problem with focusing on only strong consistency and high availability is that systems that fail to propagate information among nodes (i.e. are not

convergent) may technically have very strong consistency and availability. For example, consider a gossip-based system that provides causal consistency but restricts a node to share its updates only if it has performed a prime number of writes. Similarly, a causally consistent system may completely eliminate communication among its nodes so that each read of an object returns the latest write of that object by the reader. Each of these systems offer semantics that are technically stronger than causal consistency; each also guarantees that reads and writes are always available; yet, these semantics also feel "artificial" or less useful than causal. Convergence allows us to formalize this intuition.

Like consistency and availability, different convergence properties of varying strength exist. We describe some of these properties below.

**Eventual consistency.** A simple convergence property is *eventual consistency*, which is commonly defined as follows: the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value [56] . This formulation defines a weak convergence property; for example, it makes no promises about intervals in which some nodes are partitioned from others. Therefore, an implementation may provide such eventual consistency in less robust, yet legal ways. For example, an implementation may designate a special master node responsible for resolving conflicts and assigning a global sequence number for ordering updates [55].

**Pairwise convergence.** Most systems designed for high-availability are interested in ensuring liveness (in form of availability and convergence) despite failure of an arbitrary subset of nodes. In these systems, it is desired that a correct connected subset of nodes are always able to attain eventual consistency among themselves. Below, we define a strengthening of eventual consistency for such systems. The basic idea behind this notion is simply that any pair of nodes $s$ and $d$ should be able to converge without requiring communication with any other node.

DEFINITION 3.5. Pairwise converged. *We say that nodes $s$ and $d$ have* pairwise converged *if $s$ and $d$ are in a state in which the reads of the same object by $s$ and $d$ returns the same result.*

Now we can say that a system ensures pairwise convergence if it ensures that $s$ and $d$ can become pairwise converged through communication between $s$ to $d$:

DEFINITION 3.6. Pairwise convergent. *A system is* pairwise convergent *if for any nodes $s$ and $d$, if $s$ and $d$ issue no writes and receives no messages from other nodes, then eventually they will exchange a set of messages such that receiving these messages causes $s$ and $d$ to become pairwise converged.*

**One-way convergence.** To maximize liveness, we would like to say that any subset of connected nodes should converge on a common state. For example, we want to model the anti-entropy approach used in systems like Bayou [49] and Dynamo [19]. We therefore define *one-way convergence*.

The basic idea behind this notion is simply that any pair of nodes $s$ and $d$ should be able to converge with two steps of one-way communication: first $s$ sends updates to $d$, next $d$ sends updates to $s$; henceforth, both nodes would read the same values for all objects. To this end, we first define an intermediate state where $d$ has received whatever updates it needs from $s$. The defining property of this state is that, intuitively, once $d$ is in it, it suffices for $d$ to send updates to $s$ for $d$ and $s$ to converge to a common state. We now make this intuition more precise.

DEFINITION 3.7. Semi-pairwise converged. *We say that node $s$ has* semi-pairwise converged *with node $d$ if $s$ and $d$ are in a state such that if they issue no writes and communicate with no other nodes, then eventually $d$ will send a set of messages such that if $s$ receives these messages, then subsequent reads of the same object by $s$ and $d$ will return the same result.*

Now we can say that a system provides one-way convergence if it ensures that $s$ and $d$ can become semi-pairwise converged through communication from $s$ to $d$:

DEFINITION 3.8. One-way convergent. *A system is* one-way convergent *if for any nodes $s$ and $d$, if $s$ issues no writes and receives no messages, then eventually $s$ will send a set of messages such that if $d$ receives*

*these messages, then $s$ will have semi-pairwise converged with $d$.*

Note that the preconditions in the above definitions may seem hard to establish, but they conform to our goal of establishing a set of really weak convergence conditions that any practical system must satisfy. Indeed, there exist environment graphs and runs where these preconditions are satisfied. Broadly speaking, our intent is to argue that if a system cannot ensure convergence even in such unlikely conditions, then it is less likely to be useful in practice.

# 4 CAC limits with network failures

We begin by considering environments where only network failures occur. We first introduce *natural causal* consistency for this environment (§4.1). We then show that natural causal consistency is optimal for such environments. Towards that goal, we show two key results: (1) First, we show that semantics stronger than natural causal consistency cannot be provided by one-way convergent and always-available implementations (§4.2), (2) Second, we show that natural causal consistency can be provided by always-available and one-way convergent implementations.

## 4.1 Consistency semantics for network failures

Causal consistency is a widely used in systems where high availability is desired [8, 39, 55]. We defined causal consistency in Section 3.1. Unfortunately, causal consistency is not quite strong enough to represent a tight bound on consistency semantics that are enforceable by always-available and one-way convergent implementations. In this section, we define natural causal consistency, a strengthening of causal consistency for environments with network failures We later use natural causal consistency to establish our desired tight bound.

**Natural causal consistency.** Natural causal consistency strengthens causal consistency to ensure that the logical happens before (HB) order on operations does not violate their real-time assignment: operations that happen later in time are not ordered before earlier operations. The following definition expresses this constraint.

DEFINITION 4.1. *An execution $e$ is said to be* naturally causally consistent *if its happens before (HB) graph satisfies the following checks (Note that the NC1 and NC2 checks are identical to the C1 and C2 checks for causal consistency).*

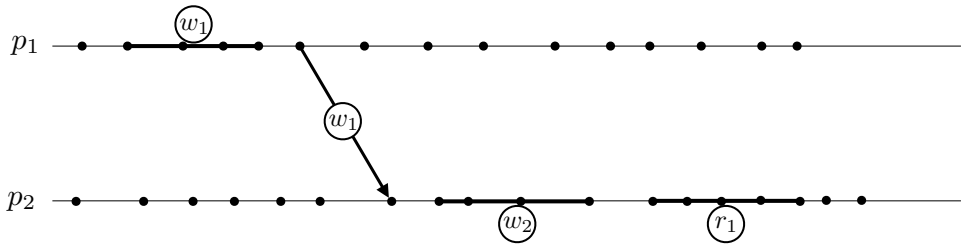
**NC1** Serial ordering at each node. *The ordering of operations at any node is reflected in $G$. Specifically, if $v$ and $v'$ are vertices corresponding to operations by the same node, then $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.*

**NC2** A read returns the latest preceding concurrent writes. *For any vertex $r$ corresponding to a read operation of object* objId*, $r$'s* writeList $wl$ *contains all writes $w$ of* objId *that precede $r$ in $G$ and that have not been overwritten by another write of* objId *that both follows $w$ and precedes $r$:*

$$\forall r, \forall w, w \in r.wl \Leftrightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : (w \prec_G w' \prec_G r \wedge w'.objId = r.objId))$$

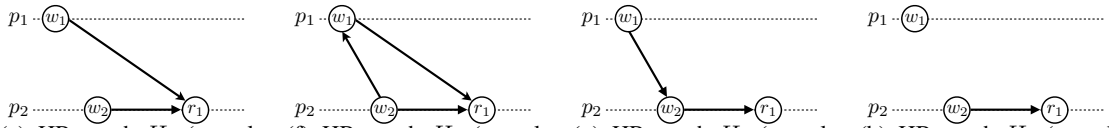
**NC3** Time does not travel backward. *For any operations $u$, $v$: $u.endTime \leq v.startTime \Rightarrow v \not\prec_G u$.*

Natural causal consistency is not a new semantics. Although we are the first to formally define natural causal consistency, it appears that most systems that claim to enforce causal consistency actually enforce the stronger natural causal consistency semantics, sometimes modified to support a system-specific conflict resolution policy [2, 8, 25, 30, 32, 39, 41, 55]. This observation should not be surprising—although a later operation can be considered *concurrent* to an earlier operation in the HB graph, it would indeed be strange for a practical implementation to order an operation that occurred later in real-time *before* an earlier




(a) A causal run. Only the relevant details are shown. Message stubs and start and end vertices are omitted.


(b) Execution $e_1$ (c) Execution $e_2$ (d) Execution $e_3$




(e) HB graph $H_1$ (causal and natural causal)




(f) HB graph $H_2$ (causal but not natural causal)




(g) HB graph $H_3$ (causal and natural causal)




(h) HB graph $H_4$ (causal and natural causal)

Figure 4: (a) A causal run, its corresponding executions (b,c,d), and their corresponding causal HB graphs (e,f,g,h). Only the relevant details are shown. All operations operate on the same object. $p_1$ issues a write $w_1$ and communicates this write to $p_2$. At a later time, $p_2$ issues a superseding write $w_2$ to the same object. Finally, $p_2$ issues a read $r_1$ to the same object. In a causal implementation, the read can return all three possible responses: $\{w_1, w_2\}, \{w_1\}, \{w_2\}$ corresponding to three different executions (labeled as $e_1$, $e_2$, and $e_3$ respectively). Figures e-h show four different causal HB graphs ($H_1, H_2, H_3, H_4$) that correspond to these three executions (both $H_3$ and $H_4$ correspond to $e_3$). In contrast, an natural causal implementation does not admit the execution $e_2$ because $H_2$ is not a valid natural causal HB graph—it violates the NC3 condition by ordering a later operation $w_2$ before an earlier operation $w_1$.

operation. Our definition of natural causal consistency simply makes the *time does not travel backward* condition explicit whereas all existing systems enforce this condition implicitly without including it in their consistency definition.

This clear and explicit addition of the *time does not travel backward* condition to the causal consistency definition has two benefits. First, following the common wisdom that stronger consistency is better, it is useful to know the strongest consistency that an implementation is enforcing so that the application writers can exploit this property in reducing their application's complexity. For example, applications can use the stronger consistency to limit the type of conflicts they need to handle.

Secondly, the stronger natural causal consistency semantics is essential to prove a tight bound as we show later in our proof in Section 4.2. Absent this requirement, an implementation could hide logical concurrency by transparently and undetectably resolving conflicts by imposing an unnatural order on operations in certain scenarios. In particular, the implementation can choose to return only a subset of the logically concurrent set of writes on a read. The NC3 condition ensures that writes that are logically concurrent should be exposed as concurrent writes to the application so that a uniform and application-specific conflict resolution policy can be enforced.

We illustrate the effect of NC3 condition with the help of an example. Consider the run in Figure 4(a). $p_1$ issues a write $w_1$ to an object $X$ and completes it. It then sends a message containing the write $w_1$ to $p_2$ that the channel delivers. At a later time, $p_2$ issues and completes another write $w_2$ to the same object. Finally, $p_2$ issues a read $r_1$ to $X$. In a causal implementation, the read can return all three possible responses: $\{w_1, w_2\}, \{w_1\}$, $\{w_2\}$ corresponding to three different executions (labeled as $e_1$, $e_2$, and $e_3$ respectively) as shown in Figure 4(b,c,d). Figure 4(e,f,g,h) shows four different causal HB graphs ($H_1, H_2, H_3, H_4$) that correspond to these three executions (both $H_3$ and $H_4$ correspond to $e_3$). In contrast, a natural causal implementation does not admit the execution $e_2$ because $H_2$ is not a valid natural causal HB graph—it violates the NC3 condition by ordering a later operation $w_2$ before an earlier operation $w_1$.

## 4.2 CAC impossibility result

Using the natural causal consistency defined above, we show in Theorem 4.2 that in an asynchronous system, it is impossible to provide any consistency stronger than natural causal while ensuring always-availability and one-way convergence. The theorem holds even if all nodes are assumed to be correct.

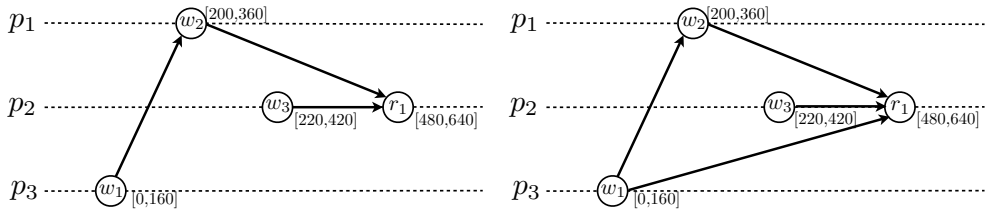
We prove this theorem by showing that we can take any system that claims to provide consistency stronger than natural causal consistency and force-feed it a workload under which it must either (i) block reads or writes (sacrificing always-availability); (ii) fail to propagate updates among connected nodes (sacrificing one-way convergence); or (iii) violate natural causal (showing that it is not, in fact, stronger than natural causal).

THEOREM 4.2. **CAC-impossibility theorem.** *No consistency semantics stronger than natural causal consistency can be enforced by a one-way convergent and always-available distributed storage implementation.*

*Proof.* By way of contradiction, suppose a one-way convergent and always-available distributed storage implementation, $I_{SC}$, enforces a semantics $SC$ that is stronger than natural causal consistency. Let $e$ be a natural causal execution that $I_{Sc}$ does not admit. We will construct a run of $I_{SC}$ that produces the rejected execution $e$, thereby contradicting the claim that $I_{SC}$ enforces $SC$. The proof goes through the following stages.

In Stage 1, we use the HB graph $G$ for $e$ to construct a *direct* HB graph $H$ for $e$ such that $H$ contains a directed edge for every pair of *non-local* vertices (i.e. vertices at different nodes) that are connected via a directed path in $G$. Note that there must exist a naturally causally HB graph $G$ for $e$ because $e$ is assumed to be naturally causally consistent.

In Stage 2, we use $H$ to construct an augmented run $\rho_a$ (and the corresponding execution $e_a$) by issuing operations at nodes in $I_{SC}$ and by controlling the behavior of the network and of the local clocks at each




(a) A natural causal HB graph (b) A direct HB graph

Figure 5: (a) A natural causal HB graph $G$ and (b) its corresponding direct HB graph $H$ generated in Stage 1 of Theorem 4.2 proof. A direct edge is added from $w_1$ to $r_1$ in $H$; $w_1$ had an indirect path to $r_1$ in $G$. For brevity, only the relevant details are shown in the HB graphs. All operations are performed on the same object. The values in square brackets indicate the execution interval in real-time for each operation.

node. Augmented run $\rho_a$ and augmented execution $e_a$ contain a few additional read operations (beyond those present in $H$), called *augmented reads*, that we issue to inspect the state of our implementation during the execution.

In Stage 3, we argue that the augmented execution $e_a$ is *similar* to the desired execution $e$. In particular, we show that (1) both $e_a$ and $e$ contain the same set of writes and $e_a$ contains all the reads present in $e$, (2) reads common to $e_a$ and $e$ will return identical responses in both these execution, and (3) operations common to $e$ and $e_a$ will start and end at the same real-time in both these executions.

Finally, to complete the proof, we argue that we can remove the additional reads added in Stage 2, to obtain a run $\rho_r$ that produces an execution $e_r$ that we prove is identical to our desired execution $e$. The argument behind this claim follows from the observation that the implementation $I_{SC}$ models a classical memory system, in which the outcome of a read operation is not influenced by prior reads. Hence, removing a few reads from $\rho_a$ should not influence the outcome of remaining operations. Furthermore, because $e_a$ is identical to the desired execution $e$ in all the unremoved operations, removing the additional reads from $\rho_a$ must produce the desired execution $e$.

We next describe these stages in more detail.

**Stage 1: Creating a direct HB graph for the execution $e$**

In Stage 1, we use $G$, the natural causal HB graph for $e$, to construct $H$, a direct HB graph for $e$. The direct HB graph $H$ contains a directed edge connecting an operation $u$ to an operation $v$ whenever $u$ and $v$ occur at different nodes and are connected via a directed path in $G$. The construction is simple and is illustrated in Figure 5.

1. First, copy all the vertices and edges from $G$ to $H$.
2. Second, for every operation $v \in G$ and for each operation $u \prec_G v$ such that $u$ and $v$ occur at different nodes, add a directed edge from $u$ to $v$ in $H$.
3. Finally, remove all non-local edges (i.e. edges that connect vertices at different nodes) of $G$ from $H$.

COROLLARY 4.3. **Direct HB graph equivalence.** *It is easy to see that the partial order $\prec_H$ imposed by the graph $H$ is the same as the partial order $\prec_G$ imposed by the graph $G$: $H$ contains the same vertices as $G$ and two vertices $u$ and $v$ have a directed path between them in $G$ iff they have a directed path between them in $H$. Therefore, if $G$ is a HB graph for the execution $e$, then $H$ must also be a HB graph for $e$.*

**Stage 2: Constructing augmented run $\rho_a$ and execution $e_a$**

In Stage 2, we produce the run $\rho_a$ and the corresponding execution $e_a$ such that $e_a$ is *similar* to our desired execution $e$. We produce $\rho_a$ by using the graph $H$ to issue a series of reads and writes while

**ALGORITHM 1:** Algorithm for assigning real-time intervals to operations.

1 Let $T$ be a time-sensitive topological sort of $H$ constructed by traversing the directed HB graph $H$ such that whenever there is a choice between multiple vertices that can be traversed next, the vertex with smallest $startTime$ is chosen
2 $\Delta := \mathbf{min}_{\forall v}((v.endTime - v.startTime)/N, \forall_{u:v \prec_T u}(u.endTime - v.startTime)/N)$ where $N$ is the total number of vertices in $G$
3 **for** $v : T$ **do**
4 $v.assignedStartTime := \mathbf{max}(v.startTime, prec(v).assignedEndTime)$ where $prec(v)$ denotes the operation immediately preceding $v$ in $T$
5 $v.assignedEndTime := v.assignedStartTime + \Delta$
6 $ts_{v,1} := (v.assignedStartTime + v.assignedEndTime)/2$
7 **for** $i \leftarrow 2$ **to** $\infty$ **do**
8 $ts_{v,i} := (ts_{v,i-1} + v.assignedEndTime)/2$

controlling the network and local clocks. We issue a few additional augmented reads to inspect the state of the implementation.

In order to produce an execution that is similar to the reference execution $e$, we need to ensure two construction invariants.

***Real-time(s) match invariant*** First, we must ensure that the real-time execution interval of a non-augmented operation at the storage implementation falls between the application observed execution interval of that operation in the reference execution. By enforcing this invariant and by controlling the propagation of messages, we can force (shown in Stage 4) the application observed execution interval of each non-augmented operation in our augmented run to be identical to its corresponding execution interval in the reference execution.

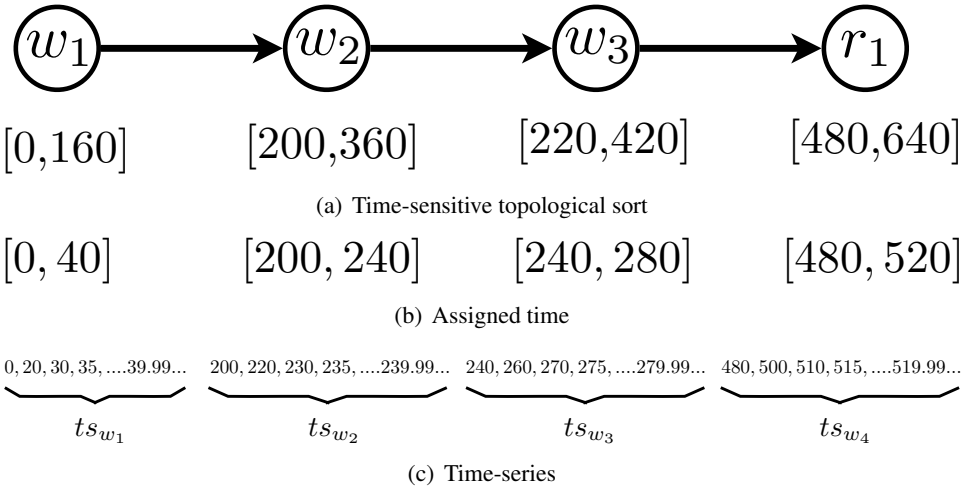
***Correct writes returned invariant*** Second, we must ensure that the set of writes returned by a non-augmented read is identical to the set of writes returned by the corresponding read in the reference execution.

Our construction proceeds in two phases to enforce these invariants. In the first phase, we identify the real-time(s) at which we execute our operations so that the real-time(s) match invariant is satisfied. In the second phase, we issue these operations by controlling the network and local-clocks. We prove in Stage 3 that our construction enforces the two invariants stated above and in Stage 4, we show that enforcing these invariants is sufficient to prove that the reference execution $e$ is accepted by the always-available, one-way convergent distributed storage implementation $I_{SC}$.

*Phase 1 (Timestamp assignment).* Phase 1 assigns the store execution interval to each operation while maintaining the real-time(s) match invariant stated above (as proved in Lemma 4.7 below): the store execution interval for an operation $v$, denoted by ( `v.assignedStartTime`, `v.assignedEndTime`), falls between its corresponding execution interval (`v.startTime`, `v.endTime`) in the reference execution $e$.

Algorithm 1 describes our approach to assign the real-time intervals to each operation. For simplicity, our algorithm assigns non-overlapping real-time intervals to different operations of our execution (including operations by different nodes). We start by constructing a *time-sensitive topological sort* $T$ of graph $H$ such that $T$ satisfies the following *time-sensitivity property*: $u \prec_T v \Rightarrow u \prec_H v \vee$ `u.startTime` $\leq$ `v.startTime`. We build such a topological sort by traversing $H$ in a way that whenever there is a choice of multiple vertices that can be traversed next, the vertex with smallest `startTime` is chosen. Its easy to see that our traversal strategy leads to the creation of a time-sensitive topological sort of HB graph $H$. Figure 6(a) shows a time-sensitive topological sort for our example scenario.

Let $N$ denote the number of vertices in $H$. We first identify the length of the largest real-time interval $\Delta$, such that we can assign a non-overlapping real-time interval of length $\Delta$ to each operation while maintaining the real-time(s) match invariant. Note that because $T$ is time-sensitive, we are guaranteed that $\Delta$, as computed in step 2 in Algorithm 1, will be positive (Lemma 4.4). We then schedule an operation at the earliest




(a) Time-sensitive topological sort

(b) Assigned time

(c) Time-series

Figure 6: (a) A time-sensitive topological sort for the HB graphs shown in Figure 5, (b) its corresponding time-assignment computed using $\Delta = 40$, and (c) time-series, as computed by the Algorithm 1 in Stage 2 of the proof of Theorem 4.2 proof.

time permitted by its real-time execution interval and after the previous operation has finished. Figure 6(b) shows the assigned times for operations in our example scenario.

We ensure that an operation and all its associated events finish in no more than $\Delta$ time. For each operation $v$, we create an infinite series of increasing timestamps, denoted by $ts_{v,i}$ $(0 < i)$, such that these timestamps fall inside $v$'s assigned real-time interval in steps $4$ and $5$ in Algorithm 1. Steps $6-8$ in Algorithm 1 show how to generate these timestamps. We denote the series of timestamps for a vertex $v$ as $ts_v$. We use these timestamps to schedule all the events that we need to execute for operation $v$ and we force the implementation to complete operation $v$ in its assigned real-time interval. Figure 6(c) shows the time-series generated for operations in our example scenario.

LEMMA 4.4. **Delta positive.** *Let $T$ be the time-sensitive topological sort of the natural causal HB graph $H$ with $N$ vertices.*
$\Delta := \mathbf{min}_{\forall v}((v.endTime - v.startTime)/N, \forall_{u:v \prec_T u}(u.endTime - v.startTime)/N)$ *is a positive value.*

*Proof.* If $\Delta$ is chosen from the first term in the `min` function, then $\Delta$ must be positive and non-zero from the assumption that each operation takes positive, non-zero time to complete. Next we consider the case in which $\Delta$ is chosen from the second term in the `min` function. From the time-sensitivity property of the topological sort $T$, we have: $v \prec_T u \Rightarrow v \prec_H u \vee v.startTime < u.startTime$. We will show that if $v \prec_T u$ then $u.endTime - v.startTime$ would be positive. Consider two cases. If $v \prec_H u$, then $u.endTime - v.startTime$ would be positive from the *time does not travel backward* property of natural causal consistency. Alternatively, if $v.startTime < u.startTime$, then $v.startTime < u.endTime$ would be positive again because $u.endTime > u.startTime$. Hence, $\Delta$ must be a positive value. □

LEMMA 4.5. **Timestamps ordered.** *Timestamps produced by Algorithm 1 for a time-sensitive topological sort $T$, satisfy the following property: $u \prec_T v \vee (u = v \wedge i < j) \Rightarrow ts_{u,i} < ts_{v,j}$.*

*Proof.* Let $u_1, u_2, ..., u_k$ be the topological sort $T$. From Step 4 and from knowing that $\Delta > 0$, we get the following ordering property: $u_1.assignedStartTime < u_1.assignedEndTime \leq u_2.assignedStartTime < u_2.assignedEndTime \leq ... \leq u_k.assignedStartTime < u_k.assignedEndTime$. From Step $6-8$ in Algorithm 1, we observe that the timestamps $ts_v$ for a vertex $v$ must be monotonically increasing and should

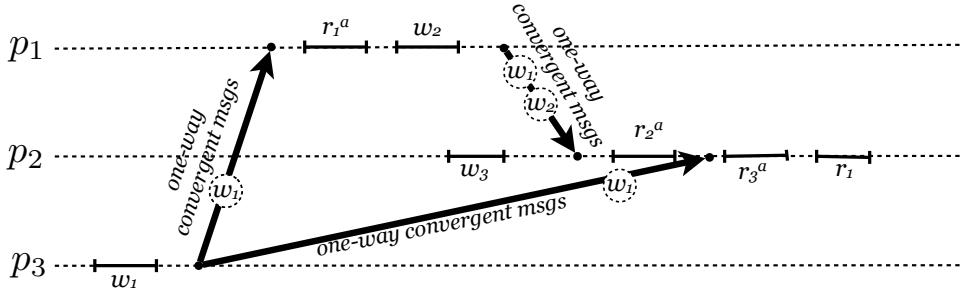



Figure 7: Schedule of operations and message exchanges generated by Phase 2 of Stage 2 of the proof of Theorem 4.2 proof for our example scenario. Thick arrows indicate the one-way convergent messages sent from one node to another. An operations execution interval is denoted by thick lines.

be confined between $v.assignedStartTime$ and $v.assignedEndTime$. Combining the ordering property with this observation about $ts_v$, we get our desired result. □

LEMMA 4.6. **AssignedStartTime progression.** *For every $v$, $v.assignedStartTime = u.startTime + j \cdot \Delta$, where $j$ is a non-negative integer smaller than $k$, the position of $v$ in the topological sort $T$, and $u$ is either equal to $v$ or $u \prec_T v$.*

*Proof.* We show this result by induction on the value of $k$. When $k = 1$, the result follows trivially. Let the result be true for $k = m$. We next show that the result is also true for $k = m + 1$. Consider two cases on the execution of `max` function in Step 4 in Algorithm 1. If $v.startTime$ was chosen in the `max` function, then the desired result follows immediately. If $prec(v).assignedEndTime$ was chosen in the `max` function, then by induction, $prec(v).assignedStartTime = u.startTime + j \cdot \Delta$, where $j < m$ and $u$ will either equal to $prec(v)$ or $u \prec_T prec(v)$. Therefore, $prec(v).assignedEndTime = u.startTime + j \cdot \Delta$, where $j < (m + 1)$ and $u \prec_T v$. Since we have $prec(v).assignedEndTime = v.assignedStartTime$, the desired result follows in this case too. □

LEMMA 4.7. **Assigned time inequality.** *For any operation $v$, we must have $v.startTime \leq v.assignedStartTime < v.assignedEndTime \leq v.endTime$.*

*Proof.* Its easy to see that $v.startTime \leq v.assignedStartTime$ from the Step 4 in Algorithm 1. From Lemma 4.4, it follows that $v.assignedStartTime < v.assignedEndTime$.

We next show that $v.assignedEndTime \leq v.endTime$. We know from Lemma 4.6 that $v.assignedStartTime = u.startTime + j \cdot \Delta$, where $j$ is a non-negative integer smaller than $k$, the position of $v$ in the topological sort $T$, and $u$ is either equal to $v$ or $u \prec_T v$. From Step 5 in Algorithm 1, we get $v.assignedEndTime = u.startTime + j \cdot \Delta$, where $0 < j \leq k$. Now, $k \leq N$, so using this we get the inequality: $v.assignedEndTime \leq u.startTime + N \cdot \Delta$. We know that $\Delta \leq (v.endTime - u.startTime)/N$ from Step 2 in Algorithm 1. Using this value of $\Delta$ in our inequality, we get: $v.assignedEndTime \leq v.endTime$, our desired result. □

*Phase 2 (Issuing operations to produce the augmented execution $e_a$).* Now we issue operations based on $T$ to produce the augmented execution $e_a$. Let $v$ iterate over $T$. We start by resetting our real-time clock to 0. For each vertex $v$ at a node $p_v$:

1. Wait until the real-time clock reads $ts_{v_1}$; because timestamps of successive vertices in $T$ are monotonically increasing (Lemma 4.5), this Step does not block indefinitely. Now, we perform the events described below (i.e. `read-start`, `write-start`, `clock-tick`, `recv-msg`, `send-msg`, `read-complete`, and `write-complete`) at consecutive timestamps from $ts_v$.

2. For each non-local incoming edge to $v$ from a write $w$ to object $o$, do the following: (a) deliver the messages that were sent when the outgoing edges of vertex $w$ were processed (see Step 4 below) and (b) add an additional read $r_o$ to object $o$ at node $p_v$ and wait until this read finishes.
3. Perform $v$'s operation at node $p_v$. Wait until the operation completes. (Because $I_{SC}$ is always available, the operation must eventually complete).
4. For each outgoing edge to vertex $v'$ at node $p_{v'}$, perform the following Steps: wait until $p_v$ sends the set of messages $M_{p_v,p_{v'}}$ that are sufficient to bring $d$ into a semi-pairwise converged state with $s$. From the one-way convergence requirement, $p_v$ must eventually send such messages. Buffer $M_{p_v,p_{v'}}$ for delivery in Step (1) when $v'$, the end point of this outgoing edge is processed.

Let $\rho_a$ be the run produced by the above construction and let $e_a$ be the corresponding execution. Let $\psi_a$ denote the environment graph that models the input events generated by the above algorithm. $\psi_a$ contains vertices for `read-start`, `write-start`, and `clock-tick` events and we add `send-msg-stub` and `recv-msg-stub` vertices for the send and receive of each message in the above execution. We draw edges connecting consecutive events at a given node and edges that connect the send of a message to its corresponding receive.

**Stage 3: Augmented execution $e_a$ is similar to the reference execution $e$**

In this stage of the proof, we argue that the augmented execution $e_a$ constructed above is similar to the reference execution $e$ by showing that $e_a$ satisfies the three requirements of similarity: (1) both $e_a$ and $e$ contain the same set of writes and $e_a$ contains all the reads present in $e$, (2) reads common to $e_a$ and $e$ will return identical responses in both these execution, and (3) operations common to $e$ and $e_a$ will start and end at the same real-time in both these executions. Its easy to see that all the operations that are present in reference execution $e$ must also be present in augmented execution $e_a$ (requirement 1) since we issued each of these operations in our construction.

We then prove other requirements for similarity. In particular, we show that writes that precede an operation in $G$ must precede that operation in any happens before graph $G_a$ for $e_a$. Similarly, due to the real-time constraint, concurrent writes returned on a read in $e$ cannot be ordered in $G_a$. Using these observations, we can show that unaugmented reads in $e_a$ must return the same set of writes in both $e$ and $e_a$ (requirement 2). We then show that by controlling the network and local-clocks, we can force unaugmented operations to start and end at the same real time in both the augmented and reference executions (requirement 3).

We start by showing a very basic requirement for deterministic execution: writes precede the reads that return them. Intuitively, there must be a causal path (containing message transfers and local state) from the write to the read that returns that write. The following lemma formalizes this intuition.

LEMMA 4.8. **Writes must have a communication path to the reads that return them.** *In augmented execution $e_a$, a write $w$ appears in the* writeList *$wl$ of a read $r$ only if $w$ precedes $r$ in the run $\rho_a$. ($w \in r.wl \Rightarrow w \prec_{\rho_a} r$.)*

*Proof.* Since an implementation can only read values produced by writes, there must exist a communication path from $p_w$ after the issue of write $w$ to $p_r$ prior to the issue of read $r$. Therefore, we must have $w \prec_{\rho_a} r$. □

LEMMA 4.9. **Precedence in the run implies precedence in the HB graph.** *Consider two non-augmented vertices $u$ and $v$. $u$ precedes $v$ in run $\rho_a$ only if $u$ precedes $v$ in $G$.*

*Proof.* Follows from construction Stage 2: a path from $u$ to $v$ can exist only if $u \prec_G v$. □

We next show a key property of one-way convergent natural causal implementations that we use later to argue that dependent writes in our reference HB graph must remain dependent in any natural causal HB graph for our augmented run.

LEMMA 4.10. **Reads return preceding writes in absence of superseding writes.** *Let $\nu$ be a run of a naturally causally-consistent implementation. Let $w$ be a write to an object $o$ and $r$ be a read to $o$ such that $p_w$ is semi-pairwise converged with $p_r$ in $\nu$. If there does not exist another write $w'$ that precedes $r$ and succeeds $w$ in $\nu$ (i.e. $\nexists w' : w \prec_\nu w' \prec_\nu r$), then we must have $w \in r.wl$.*

*Proof.* Consider a prefix $\nu_{pre}$ of $\nu$ that consists of only the events and edges that have a path to the start of $r$ or $w$ in $\nu$. Now extend this prefix to form $\nu'$ as described next. (1) Add `clock-tick` events to ensure termination of $r$ using the availability requirement of $I_{SC}$. (2) Wait until $p_r$ sends messages to force $p_w$ into a pairwise converged state and deliver these messages at $p_w$. Because $p_w$ is semi-pairwise converged with $p_r$ in $\nu$ and our implementation is one-way convergent, $p_r$ must eventually send such messages. (3) Add a read $r'$ at $p_w$ to the object that $r$ is reading. (4) Finally, add `clock-tick` events to ensure termination of $r'$. From the one-way convergence requirement, we must have $r'.wl = r.wl$.

It suffices to show that $w \in r'.wl$. Suppose, for the sake of contradiction, that $w \notin r'.wl$. We now make the following sequence of claims.

**Claim 1.** *There exists a write $w'$ such that* **(a)** *$w \prec_J w' \prec_J r'$ in any natural causal HB graph $J$ for $\nu'$ and* **(b)** *$w' \|_{\nu'} w$.* We note that in any natural causal HB graph $J$ for $\nu'$, we must have $w \prec_J r'$ (From NC1—both $r'$ and $w$ occur at the same node and $w$ was completed first). So, $w \notin r'.wl$ implies that from NC2, there exists a write $w'$ to object $o$ such that $w \prec_J w' \prec_J r'$ (**Claim 1a**) and $w' \in r'.wl$. But reads $r$ and $r'$ must return identical response, therefore, we must have $w' \in r.wl$. Applying Lemma 4.8 to $w' \in r.wl$, we get that $w' \prec_{\nu'} r$. We know that from our assumption that there does not exist any write $w'$ such that $w \prec_\nu w' \prec_\nu r$. Because $\nu'$ does not contain any new writes or any new paths between existing writes, it follows that there does not exist any write $w'$ such that $w \prec_{\nu'} w' \prec_{\nu'} r$.

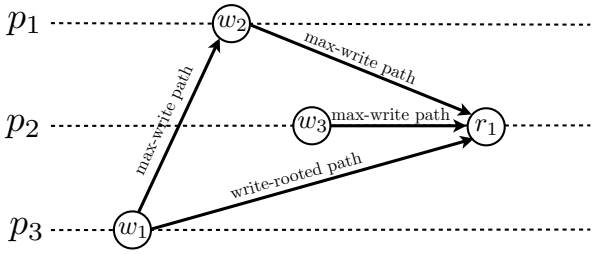
Consider cases on the precedence of $w$ and $w'$ in $\nu$. $w \prec_\nu w'$ would imply $w \prec_\nu w' \prec_\nu r$ contradicting our assumption that there does not exist any write $w'$ that precedes $r$ and succeeds $w$. $w' \prec_\nu w$ is ruled out by the claim that $w \prec_J w'$ where $J$ satisfies NC3 condition of natural causal HB graphs. Therefore , $w'$ must be concurrent to $w$ in $\nu$ and $\nu'$ (**Claim 1b**).

**Claim 2.** *There exists a run $\nu''$ such that* **(a)** *in $\nu''$, $w$ starts after $w'$ finishes, and* **(b)** *$\nu''$ is indistinguishable to the implementation from the run $\nu'$.* If $w$ and $w'$ are concurrent in $\nu'$, we can construct a run $\nu''$ in which $w$ starts after $w'$ finishes as follows: add $|w'.endTime - w.startTime|$ to all the vertices that are reachable from the $w$'s `write-start` vertex in $\nu'$ (**Claim 2a**). Because these timestamps are not visible to the implementation, $\nu''$ is indistinguishable to the implementation from $\nu'$ (**Claim 2b**).

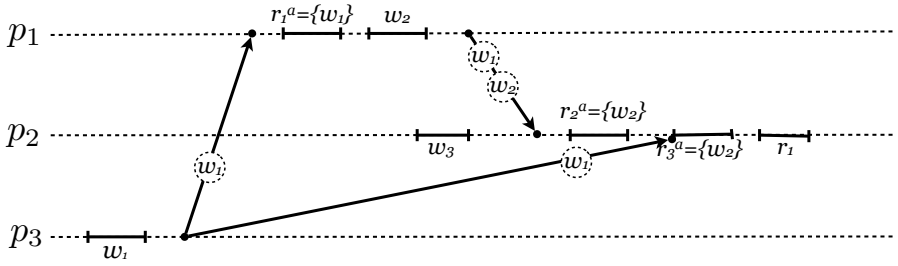
These two claims together create a contradiction as follows. The run $\nu''$ is indistinguishable to the implementation from the run $\nu'$. Hence, the run $\nu''$ should still satisfy Claim 1a. In addition, the run $\nu''$ also satisfies Claim 2a. But, Claim 1a ($w \prec_J w' \prec_J r'$) and Claim 2a ($w$ starts after $w'$ finishes) cannot be simultaneously satisfied by a run of a natural causal-consistent implementation owing to the NC3 condition (*time does not travel backwards*). By contradiction, our assumption that $w \notin r'.wl$ must be wrong. Hence, we must have $w \in r'.wl$. Furthermore, because $r$ and $r'$ were issued in a pairwise converged state, we have $r.wl = r'.wl$. Hence, $w \in r.wl$ in the run $\nu''$ and in run $\nu'$.

We next claim that $w \in r.wl$ even in the run $\nu$. The argument behind this claim relies on the deterministic nature of our implementation. In particular, we argue that when $r$ is issued, node $p_r$ receives that same set of messages and observes the same set of events in the run $\nu'$ as it observes in the run $\nu$. Therefore, the response of the implementation to the read $r$ in both these runs should be identical. Hence, $w \in r.wl$ in run $\nu$. □

Next, we apply the lemma proved above along directed paths from a write $w$ to a read $r$ in our augmented run $\rho_a$ such that the $r$ and $w$ satisfy the following condition: there does not exist a write that supersedes




(a) Write-rooted paths and max-write paths in $H$




(b) Augmented reads on max-write paths

Figure 8: (a) A set of write-rooted and max-write paths and (b) a set of augmented reads on max-write paths and their result. Figure (a) shows a set of write-rooted paths: $w_1 \prec_H w_2$, $w_1 \prec_H r_1$, $w_2 \prec_H r_1$, and $w_3 \prec_H r_1$. Only paths $w_1 \prec_H w_2$, $w_2 \prec_H r_1$, and $w_3 \prec_H r_1$ are max-write paths. As Figure (b) illustrates and Lemma 4.13 proves, the augmented reads on such max-write paths return the preceding write on that path. Augmented read $r_1^a$ must include the write $w_1$ in its `writeList`. Similarly, the augmented read $r_3^a$ must include the write $w_2$ in its `writeList`. On the other hand, for a non-max-write path $w_1 \prec_H r_1$, $r_3^a$ is not guaranteed to include $w_1$ in its `writeList` and indeed, $r_1$'s `writeList` does not include $w_1$ in our illustration.

$w$ and precedes $r$ in the run $\rho_a$. To identify such reads and writes, we first define *write-rooted* paths and *max-write* paths.

DEFINITION 4.11. **Write-rooted path.** *We say that a path $v_1 \prec_G v_2 \prec_G \ldots \prec_G v_k$ is* write-rooted *if whenever $p_{v_m} \neq p_{v_{m+1}}$, $p_{v_m}$ is a write operation.*

DEFINITION 4.12. **Max-write path.** *We say that a write-rooted path is the* max-write *path for vertices $u$ and $v$ in graph $G$, if it contains the maximum number of non-terminal writes among all the write-rooted paths from $u$ to $v$ in $G$.*

LEMMA 4.13. **Write dependencies are preserved in the augmented HB graph.** *If a write $w$ precedes an operation $u$ in $e$'s HB graph $G$ then $w$ precedes $u$ in any HB graph $G_a$ for the augmented execution $e_a$. ($w \prec_G u \Rightarrow w \prec_{G_a} u$)*

*Proof.* From Corollary 4.3, $w$ precedes $u$ in $G$ implies that $w$ must precede $u$ in $H$, the direct HB graph constructed using $G$. Let $P_{max}$ denote the max-write path from $w$ to $u$ in $H$. There must exist one such write-rooted path because while converting a general HB graph into a direct HB graph in Stage 1, we must have created at least one such write-rooted path by adding an edge from the write $w$ to the vertex $u$.

We complete the remainder of this proof by using mathematical induction on the number of writes in the max-write path ($P_{max}$) from $w$ to $u$. We first consider the base case where the number of non-terminal writes in $P_{max}$ is 0. If $w$ and $u$ occur on the same node, then the claim follows from NC1. Consider the

case when $p_w \neq p_u$. In this case, $H$ must have a direct edge from $w$ to $u$ from construction in Stage 1. Processing this edge from $w$ to $u$ in Stage 2 involved performing one-way convergence from $p_w$ to $p_u$ and inserting before $u$, an augmented read $r_f$ at $p_u$ to the object that $w$ was writing.

Consider the run $\nu$ constructed after adding the augmented read $r_f$. It contains a write $w$ and a read $r_f$ such that $p_w$ has semi-pairwise converged with $p_{r_f}$ (Stage 2, Phase 2, Step 2a). Furthermore, both the read $r_f$ and write $w$ are operating on the same object $o$ (Stage 2, Phase 2, Step 2b). Finally, because the path from $w$ to $u$ was a max-write path with no non-terminal writes, it follows that there does not exist a write $w'$ such that $w \prec_H w' \prec_H u$ (from definition of max-write). Applying Lemma 4.9, we get that there does not exist a write $w'$ such that $w \prec_\nu w' \prec_\nu r_f$. Now we can apply Lemma 4.10 to get that $w \in r_f.wl$.

From NC2, we must have $w \prec_{G_a} r_f$ and from NC1, we have $r_f \prec_{G_a} u$, it follows by transitivity that $w \prec_{G_a} u$.

After completing the base case, we now prove the induction Step. Suppose that our lemma holds true for max-write paths containing up to $k(\geq 0)$ non-terminal writes. We show that the lemma must also hold for max-write paths with $k+1$ non-terminal writes. Consider the last non-terminal write $w_l$ on the max-write path. We note that a max-write path with $k+1$ non-terminal writes must consist of max-write path with $k$ vertices ending at $w_l$ and a max-write path with 0 non-terminal vertices from $w_l$ to $u$. Now, from mathematical inducting, we must have $w \prec_{G_a} w_l$ and $w_l \prec_{G_a} u$. Using transitivity of $\prec_{G_a}$ relation, we get our desired result: $w \prec_{G_a} u$. Therefore, by induction, Lemma 4.13 is true. □

Now we show that if a read appears in both $e$ and $e_a$, then it must return identical responses in both these executions.

LEMMA 4.14. **Common reads return identical responses.** *For every read $r \in e$ with writeList $wl$ in $e$ and $wl_a$ in $e_a$, $wl = wl_a$*

*Proof.* Consider the following two cases:
**Case 1:** $w \in wl_a \wedge w \notin wl$**:** From Lemma 4.8 and Lemma 4.9, $w \in wl_a \Rightarrow w \prec_G r$, so for $r$ to not return $w$ there must exist a $w'$ such that $w \prec_G w' \wedge w' \prec_G r$. But from Lemma 4.13 , $w \prec_G w' \wedge w' \prec_G r \Rightarrow w \prec_{G_a} w' \wedge w' \prec_{G_a} r$ so $r.wl_a$ could not include $w$ (from NC2). Contradiction.
**Case 2:** $w \in wl \wedge w \notin wl_a$**:** From NC2, $w \prec_G r$, and from Lemma 4.13, $w \prec_{G_a} r$. So, from NC2, for $r$ to not return $w$ in $e_a$, there must exist $w'$ such that $r$ returns $w'$ in $G_a$ and $w \prec_{G_a} w'$. From Case 1, we know that $w' \in wl$. Combining these two observations, it must be the case that $w||_G w'$ ($w$ is concurrent to $w'$ in $G$) whereas $w \prec_{G_a} w'$.

As in the proof of Lemma 4.10, because $w$ and $w'$ are concurrent in $G$, they must be concurrent in $\rho_a$ from construction. Hence, as before, we can construct a different environment graph $\rho'_a$ in which $w$ starts after $w'$ finishes in real-time. Because the implementation does not have access to real-time, it must produce identical responses in both $\rho_a$ and $\rho'_a$. In particular, the write lists $wl_a$ and $wl'_a$ returned respectively by read $r$ in $\rho_a$ and $\rho'_a$ must be identical. However this cannot be, since by NC3 we cannot have $w \prec_{G'_a} w'$ in any HB graph $G'_a$ for the execution for the run $\rho'_a$. Contradiction. □

LEMMA 4.15. **Feasible real-time assignment.** *For each operation $v$ in the run $\rho_a$, $\psi_a$, $v.startTime < v.storeStartTime < v.storeEndTime < v.endTime$.*

*Proof.* We know that $v.assignedStartTime < v.assignedEndTime$ from Step 5 in Algorithm 4.4 and from knowing that $\Delta > 0$ (Lemma 4.4). Combing this observation with Step 6 in Algorithm 1, we get that $ts_{v,1} > v.assignedStartTime$. From construction in Phase 2 of Stage 2, we know that $v.storeStartTime = ts_{v,1}$. Combining them, we get $v.assignedStartTime < v.storeStartTime$.

Its easy to see that $v.storeEndtime < v.assignedEndTime$ because we create an infinite series of timestamps between $v.assignedStartTime$ and $v.assignedEndTime$ (Step $6-8$, Algorithm 1), and we force the implementation to finish $v$ before moving on to the timestamps beyond $v.assignedEndTime$

(we only advance timestamps in construction Step 1, in Stage 2). Of these infinite timestamps, only a finite number will be used to perform the operation $v$. Hence, we get $v.storeEndTime < v.assignedEndTime$.

Combining the inequalities above, we get: $v.assignedStartTime < v.storeStartTime < v.storeEndTime < v.assignedEndTime$. Combining this result with Lemma 4.7, we get our desired result. □

Lemma 4.14 shows that the reads common to $e_a$ and $e$ return identical set of writes. Lemma 4.15 shows that by controlling the delivery of application events to the storage implementation and vice-versa, we can ensure that the store execution time-interval of an unaugmented operation in $e_a$ falls between its execution interval in $e$. Therefore, by controlling the delivery of these events between the application layer and the storage implementation layer, we can ensure that the execution interval of unaugmented operations in $e_a$ is identical to their execution interval in $e$. Recall that controlling the delay between the application observed `startTime` of an operation, which is reported in th execution, and the store observed `storeStartTime` is possible because we assume that these layers communicate asynchronously (Section 2.7). Similarly, we can also control the delay between the store observed `storeEndTime` for an operation and the application observed `endTime` for an operation.

So far, we have shown that the augmented execution $e_a$ matches the reference execution $e$. We next show how we can construct an execution $e_r$ that is *identical* to the reference execution $e$.

**Stage 4: $I_{SC}$ accepts the execution $e$.**

In this stage, we complete our proof by showing that the always-available, and one-way convergent distributed storage implementation $I_{SC}$ accepts the execution $e$. In stages 1 through 3, we showed a construction to produce an augmented execution $e_a$ that is similar to our desired reference execution $e$. We next show the construction to produce a run $\rho_r$ that produces an execution $e_r$ that is identical to the reference execution $e$.

We construct the run $\rho_r$ by eliminating the augmented reads from $\rho_a$ that we added in Stage 2. Because our implementation is assumed to be classical and hence not influenced by reads, $\rho_r$ should produce an execution $e_r$ that must be identical to $e$: (1) the executions $e_r$ and $e$ contain the same set of operations, (2) reads common to both these executions return the same response in both $e$ and $e_r$, and (3) operations start and end at the same real-time in both the executions $e$ and $e_r$. Therefore, Theorem 4.2 holds. □

In this section, we showed that consistency semantics stronger than natural causal consistency cannot be enforced by an always-available and one-way convergent implementation. The key idea that made this result possible is our use of one-way convergence as a requirement on the implementations that we consider. In the next section, we prove that natural causal consistency is indeed achievable by one-way convergent and always-available implementations. Both these results together complete the proof for our claim that natural causal consistency provides a tight bound on consistency semantics that are enforceable by always-available and one-way convergent implementations.

## 4.3 Natural causal consistency is enforceable by an always-available and one-way convergent implementation

We next show that natural causal consistency can be enforced using an always-available and one-way convergent implementation. This result is trivial because, as argued earlier, many existing systems such as Bayou [55] and PRACTI [8] provide natural causal consistency. However, for completeness, Theorem 4.16 proves this result. Because the proof for environments without Byzantine nodes follows from the proof for environments with Byzantine nodes, we avoid presenting the complete proof here. Instead, Theorem 4.16 derives its result from Theorem 5.39.

THEOREM 4.16. *Natural causal consistency can be enforced by an always-available and one-way convergent implementation.*

*Proof.* Follows from Theorem 5.39 (proved later) which shows that *view-fork-join-causal (VFJC)* consis-

tency can be provided using an always-available and one-way convergent implementation and from noting that in absence of Byzantine nodes, the VFJC consistency reduces to natural causal consistency. □

# 5 CAC limits with Byzantine failures

The previous section considered only network failures. In this section, we consider a Byzantine failure model. We introduce *fork-causal (FC)* consistency, *fork-join-causal (FJC)* consistency, and *view-fork-join-causal (VFJC)* consistency semantics for Byzantine failure environments that require high availability. We then show that fork-causal consistency and stronger consistency semantics cannot be implemented without sacrificing availability or convergence. Finally, we show that VFJC consistency *can* be enforced by a one-way convergent and always-available implementation.

## 5.1 Consistency semantics for Byzantine failures

In this section, we introduce a number of consistency semantics that are designed for a environments that must ensure high availability despite the potential presence of Byzantine participants. We start by defining *fork-causal* consistency, inspired by SUNDR's [36] fork-linearizable consistency, for environments that desire high availability. We find that while fork-causal consistency fulfills our availability goals, it does not guarantee convergence. Therefore, we further weaken fork-causal consistency to define Depot's *fork-join-causal* consistency designed for environments desiring both convergence and high availability. Finally, in an attempt to find the strongest achievable consistency semantics that still fulfills our convergence and availability goals, we define *view-fork-join-causal* consistency, a strengthening of fork-join-causal consistency, which limits the set of forks that faulty nodes can introduce.

**Fork-causal consistency.** Fork-causal consistency ensures that each correct node sees a causally consistent subset of the global execution. Such a consistency semantics is appealing because applications designed for causal consistency will continue to be safe under fork-causal consistency. Fork-causal consistency maintains the key property of causal consistency: reads return the most recent concurrent writes, while weakening the other properties to account for Byzantine faulty nodes as described below.

DEFINITION 5.1. *An execution $e$ is* fork-causally consistent (FC-consistent) *if there exists a directed acyclic graph $G$, called a HB (happens before) graph, containing vertices for operations by correct nodes and vertices for write operations by faulty nodes that were returned on reads by correct nodes, such that $G$ satisfies the following consistency check.*

**FC1** Serial ordering of operations by correct nodes. *The operations of a correct node are totally ordered in $G$. This total ordering of operations by a correct node $p$ must be consistent with the real-time at which these operations were issued by $p$. Specifically, if $p$ is a correct node and $v$ and $v'$ are vertices corresponding to operations by $p$, then $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.*

**FC2** A read returns the latest preceding concurrent writes. *For any vertex $r$ corresponding to a read operation of object* objId*, $r$'s* writeList $wl$ *contains all writes $w$ of* objId *that precede $r$ in $G$ and that have not been overwritten by another write of* objId *that both follows $w$ and precedes $r$:*

$$\forall r, \forall w, w \in r.wl \Leftrightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : (w \prec_G w' \prec_G r \wedge w'.objId = r.objId))$$

**FC3** Serial ordering of each node's operations that are observed by a correct node. *An operation $u$ is said to be* observed *by a correct node $p$ in $G$ if either $p$ executes $u$ or if $p$ executes an operation $v$ such that $u \prec_G v$. For any operation $o$ by a correct node and for all operations $u_1$ and $u_2$ by a node $p$, $u_1 \prec_G o \wedge u_2 \prec_G o \Rightarrow u_1 \prec_G u_2 \vee u_2 \prec_G u_1$.*

As the name suggests, fork-causal consistency induces *forks* that we define next.

DEFINITION 5.2. **Fault-tree.** *A fault-tree for a faulty node $f$ is a directed tree consisting of operations by $f$.*

DEFINITION 5.3. **Fork.** *In a fault-tree for a faulty node, we call a directed path from the root to a leaf, a* fork. *The total set of forks induced by a faulty node is the union of the set of forks induced by each of its fault-trees.*

OBSERVATION 5.4. **Fork-causal fault-tree creation property.** *For each faulty node in a fork-causally consistent execution $e$, the fork-causal HB graph for $e$ induces a set of fault-trees over the operations of $f$ that are observed by some correct node.*

The fault-tree creation property states that the operations of a faulty node $f$ that are observed by correct nodes in a fork-causal HB graph can be made to form a set of fault-trees. To construct the fault-tree set for a faulty node, we first identify the vertices that need to be included in the tree—i.e. the operations of $f$ that are observed by some correct node. Next, we connect these vertices using the HB graph as follows. For each vertex $u, v$ that are selected in the first step, we add an edge from $u$ to $v$ if $u$ precedes $v$ in the HB graph. The resulting graph is a fault-tree for the given fork-causal HB graph.

The fault-tree creation property follows directly from the FC3 property of fork-causal consistency as two branches of a fault tree, a.k.a. forks, never join in a fork-causal HB graph. Indeed, such joined branches would violate the FC3 property—serial ordering for each node's operations that are observed by a correct node. More concretely, for two branches to join, there must be an operation $v$ with two parents $v_1$ and $v_2$ such that $v_1$ and $v_2$ are concurrent. Furthermore, $v$ must be observed by some correct node (say $p$)—precondition for this property. But, $v_1$ and $v_2$ are also observed by $p$ (because of the definition of observed) contradicting the FC3 property. Therefore, from FC3, all FC HB graphs must satisfy the fault-tree creation property.

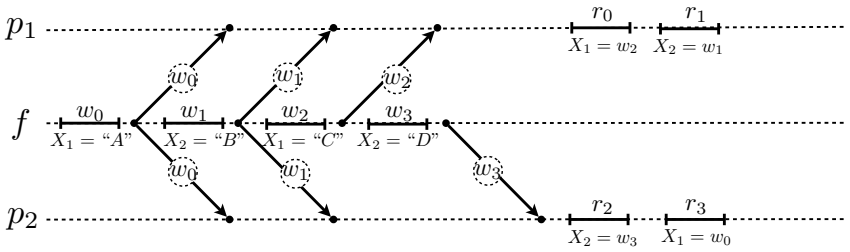
*Comparison with causal consistency.* Causal consistency enforces conditions that are analogous to those enforced by FC consistency, but it requires them to hold for operations issued by *all* nodes, not just correct ones. In particular, while FC consistency imposes the serial ordering constraint on only the operations by correct nodes, causal consistency imposes this constraint on all operations. In addition, because faulty nodes can behave arbitrarily, fork-causal consistency makes no guarantees about the results of reads issued at faulty nodes.

Figure 9(a) shows a run that is fork-causally consistent but not causally consistent. In this example, node $f$ is faulty and produces four writes $w_0$, $w_1$, $w_2$, and $w_3$. Node $p_1$ observes $w_0$, $w_1$, and $w_2$ but not $w_3$, and node $p_2$ observes $w_0$, $w_1$, and $w_3$ but not $w_2$. Figure 9(b) shows the corresponding execution. As Figure 9(c) illustrates, we can produce an edge assignment and observer graph that passes all tests for FC consistency by dispensing with the serial ordering constraint at the faulty node. Conversely, it is impossible to produce an edge assignment to produce an observer graph $G'$ that passes the causal consistency checks.

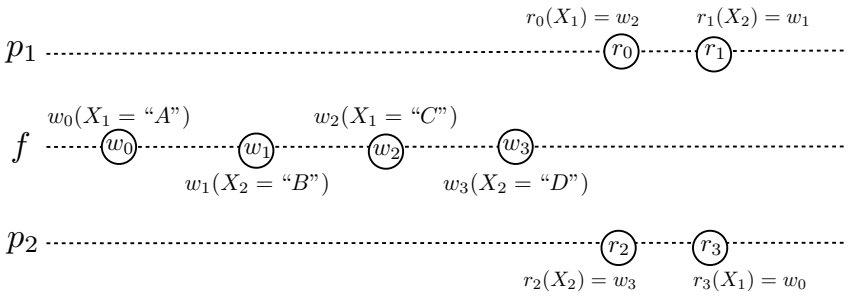
Figure 10(a) shows the fault-tree for node $f$ and Figure 10(b,c) shows the corresponding forks. Note that in fork-causal consistency, no correct node can observe more than one fork.

**Fork-join-causal consistency.** Unfortunately, as we prove later (Section 5.2), a fundamental limitation of the existing forking-based consistency semantics [10, 12, 36, 37, 42, 48] is that they compromise convergence by preventing forked nodes from observing each other's updates. Fork-causal consistency also relies on forking to handle Byzantine failures. Hence, it also suffers from the same limitation. The Depot system [41] overcomes this limitation by permitting forks to be *joined* to enforce a slightly weaker, but convergent, fork-join-causal consistency as its correctness requirement. We next define fork-join-causal consistency.

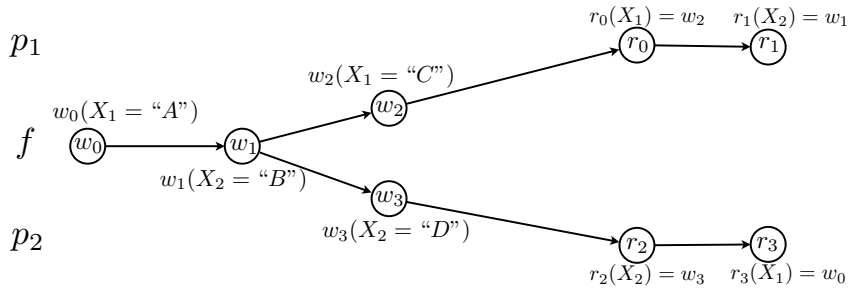
DEFINITION 5.5. *An execution $e$ is* fork-join-causally consistent (FJC-consistent) *if there exists a directed acyclic graph G, called a HB (happens before) graph, containing a vertex for every operation by a correct node and a vertex for every write operation by a faulty node that is returned on a read by a correct node,*




(a) Run




(b) Execution




(c) Fork-causal HB graph

Figure 9: Illustration of fork-causal consistency and its comparison with fork-causal consistency. (a) A run with a faulty node $f$ that issues two non-serial writes $w_2$ and $w_3$, (b) the corresponding execution, and (c) a fork-causal happens before graph for this execution. The figures show only the relevant fields of various vertices for brevity. In the run, the faulty node $f$ issues non-serial writes $w_2$ and $w_3$ and exposes these writes to correct nodes $p_1$ and $p_2$ respectively. There is no causally consistent happens before graph for this execution because in the execution $w_2$ and $w_3$ are not serially ordered according to any possible history of node $f$—$p_1$ observed $w_2$ without observing $w_3$ and $p_2$ observe $w_3$ without observing $w_2$. The HB graph is fork-causally consistent because fork-causal consistency does not require total ordering of faulty node $f$'s operations.

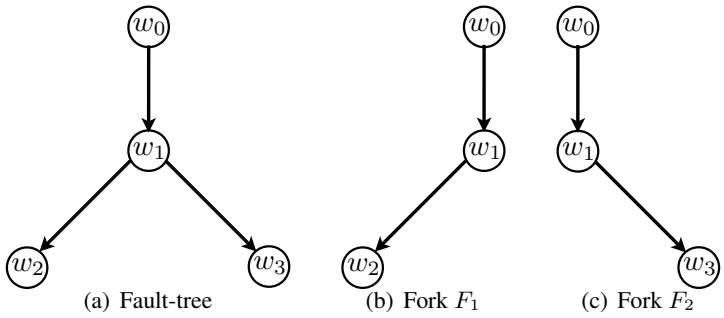



Figure 10: (a) A fault-tree, and its two forks (b,c).

*such that G satisfies the following consistency check.*

**FJC1** Serial ordering of operations by correct nodes. *The operations of a correct node are totally ordered in G. This total ordering of operations by a correct node p must be consistent with the real-time at which these operations were issued by p. Specifically, if v and v′ are operations by the p, then* $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.

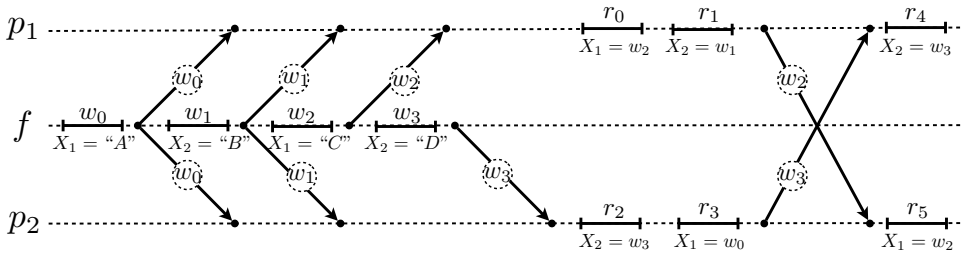
**FJC2** A read return the latest preceding concurrent writes. *For any vertex r corresponding to a read operation of object* objId*, r's* writeList $wl$ *contains all writes w of* objId *that precede r in G and that have not been overwritten by another write of* objId *that both follows w and precedes r:*

$\forall r, \forall w, w \in r.wl \Leftrightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : (w \prec_G w' \prec_G r \wedge w'.objId = r.objId))$

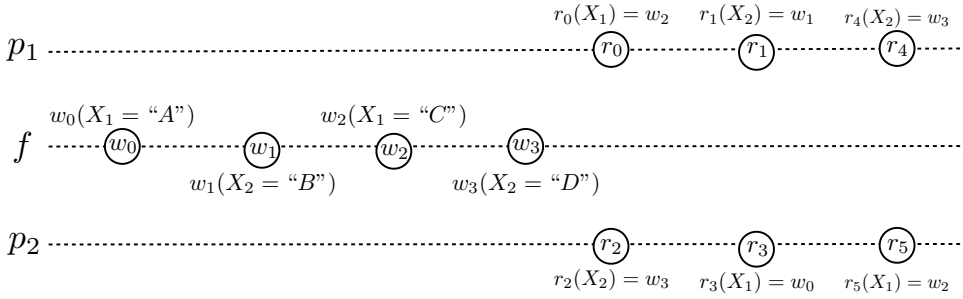
*Comparison with fork-causal consistency.* Fork-join-causal consistency weakens fork-causal consistency by eliminating the FC3 condition to ensure convergence. The FC3 condition requires that operations by any node seen by each correct node must be serially ordered. Hence, correct nodes that have seen non-serial writes by the same faulty node cannot observe each other's operations without violating FC3. By eliminating the FC3 requirement, FJC consistency permits such forked correct nodes to join these forks at the cost of observing concurrent forked writes by a faulty node, a behavior not possible in fork-causal consistency. FJC implementations, such as Depot, expose such forked writes as multiple concurrent writes by different virtual nodes that correspond to the same faulty node.

Figure 11 illustrates this difference. Figure 11 extends the scenario illustrated in Figure 9. The faulty node $f$ issues writes $w_0, w_1, w_2$ and $w_3$ and exposes $w_0$, $w_1$, and $w_2$ to $p_0$ and $w_0$, $w_1$, and $w_3$ to $p_1$ thereby forking correct nodes $p_1$ and $p_2$. Later, $p_1$ and $p_2$ communicate and exchange forked writes $w_2$ and $w_3$ such that both $p_1$ and $p_2$ observe both the forks, a phenomenon prohibited by fork-causal consistency. This *joining* of forks enables $p_1$ and $p_2$ to converge to a common state in which reads to identical objects return identical responses. No fork-causal happens before graph exists for this execution because in any such happens before graph, both $w_2$ and $w_3$ must be concurrent and yet be observed by correct nodes $p_1$ and $p_2$; a violation of the FC3 condition of fork-causal consistency. FJC omits the FC3 condition for faulty nodes and hence admits a happens before graph in which writes $w_2$ and $w_3$ by the faulty node $f$ can be concurrent and yet be observed by correct nodes $p_1$ and $p_2$.

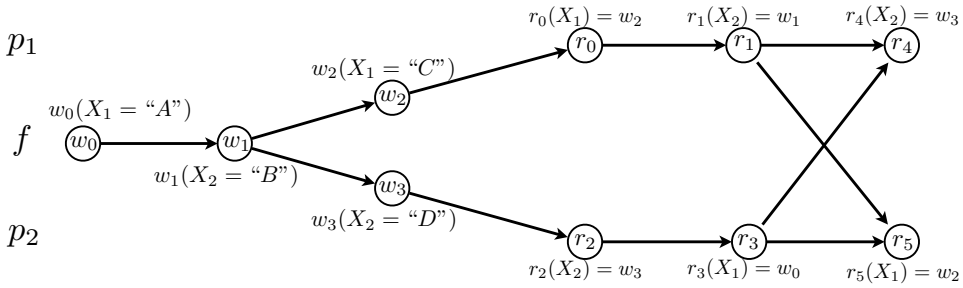
**View-fork-join-causal consistency.** We designed *view-fork-join-causal* consistency with the goal of identifying the strongest, always-available, and one-way convergent consistency semantics that can be achieved in presence of Byzantine participants. While we fail to show a tight bound, VFJC consistency provides an effective and useful lower bound on the strength of consistency semantics that can be achieved in such environments. As we show later (Section 5.2), neither causal consistency nor fork-causal consistency is




(a) Run




(b) Execution




(c) Fork-join-causal HB graph

Figure 11: Illustration of fork-join-causal consistency and its comparison with fork-causal consistency. (a) A run with a faulty node $f$ that issues two non-serial writes $w_2$ and $w_3$, (b) the corresponding execution, and (c) a fork-join-causal happens before graph for this execution. The figures show only the relevant fields of various vertices for brevity. In the run, the faulty node $f$ issues non-serial writes $w_2$ and $w_3$ and exposes these writes to correct nodes $p_1$ and $p_2$ respectively. Later, $p_1$ and $p_2$ exchange forked writes $w_2$ and $w_3$. There is no fork-causally consistent happens before graph for this execution because in the execution $w_2$ and $w_3$ are observed by both the correct nodes $p_1$ and $p_2$ violating the FC3 condition of fork-causal consistency. The HB graph is fork-join-causally consistent because fork-join-causal consistency does not require total ordering of faulty node $f$'s operations that are observed by a correct node.

achievable in such environments. Furthermore, Depot has shown that FJC consistency is indeed achievable. VFJC consistency further tightens the gap between Depot's FJC consistency that permits nodes to converge, and fork-causal, causal, and natural causal consistency that are stronger but not achievable.

Like FJC consistency, VFJC consistency permits forks to be joined, thereby allowing convergent implementations. However, unlike Depot's FJC consistency, which admits an unbounded number of forks, VFJC consistency limits the set of forks that are admitted in an execution to only those that are unavoidable without compromising liveness. This restriction on the set of admitted forks effectively forces VFJC implementations to force the eviction of faulty nodes and limit the damage such nodes can cause in an execution.

To describe the checks for limiting the set of admissible forks, we use a set of *view* vertices in addition to the read and write vertices in a VFJC HB graph. In addition, VFJC consistency includes natural causal consistency's *time does not travel backward requirement* and fork-causal consistency's fault-tree creation property. We next define VFJC consistency and then contrast VFJC consistency with FJC consistency. We first define a few terms to simplify the description of VFJC consistency checks.

DEFINITION 5.6. **Non-local edge.** *Two vertices $v_1$ and $v_2$ are said to be connected through a non-local directed edge from $v_1$ to $v_2$ if $v_1$ and $v_2$ correspond to operations occurring at different nodes, $v_1$ precedes $v_2$, and there does not exist a vertex $u$ such that $v_1$ precedes $u$ and $u$ precedes $v_2$.*

DEFINITION 5.7. **Local edge.** *Two vertices $v_1$ and $v_2$ are said to be connected through a local directed edge from $v_1$ to $v_2$ if $v_1$ and $v_2$ correspond to operations by the same node $p$, $v_1$ precedes $v_2$, and there does not exist a vertex $u$ corresponding to an operation by $p$ such that $v_1$ precedes $u$ and $u$ precedes $v_2$.*

DEFINITION 5.8. **Edge.** *Two vertices $v_1$ and $v_2$ are said to be connected through a directed edge from $v_1$ to $v_2$ (denoted by $v_1 \rightarrow v_2$) iff $v_1$ and $v_2$ are connected through either a local or a non-local directed edge.*

DEFINITION 5.9. **Projection of a vertex $v$ in a graph $G$.** *The projection $G_v$ of a vertex $v$ in graph $G$ consists of the subgraph rooted at $v$ with all the vertices $v' : v' \prec_G v \vee v' = v$ and edges connecting these vertices.*

DEFINITION 5.10. **Correct node in a subgraph.** *We say that a node $p$ is correct in a graph $H$ if all the vertices of node $p$ in graph $H$ are totally ordered. In a happens before graph, these vertices may include read, write, or view vertices.*

DEFINITION 5.11. *An execution $e$ is* view-fork-join-causally consistent (VFJC-consistent) *if there exists a directed acyclic graph G, called a HB (happens before) graph, containing a vertex for every operation by a correct node, a vertex for every write operation by a faulty node that is returned on a read by a correct node, and a set of view vertices to restrict the set of admitted forks, such that $G$ satisfies the consistency checks enumerated below.*

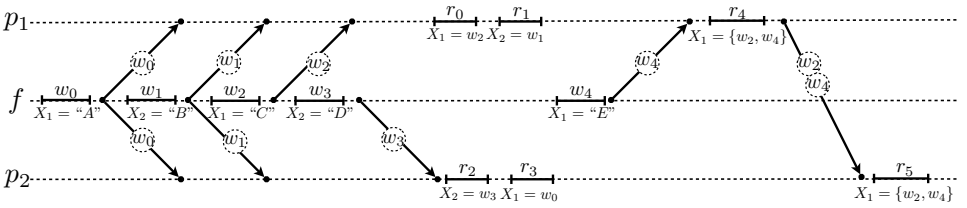
**VFJC1** Serial ordering of operations by correct nodes. *All vertices of a correct node are totally ordered in $G$. This total ordering of vertices of a correct node $p$ must be consistent with the real-time at which the corresponding operations were issued by $p$. Specifically, if $v$ and $v'$ are operations by the $p$, then $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.*

**VFJC2** A read return the latest preceding concurrent writes. *For any vertex $r$ corresponding to a read operation of object* objId*, $r$'s* writeList $wl$ *contains all writes $w$ of* objId *that precede $r$ in $G$ and that have not been overwritten by another write of* objId *that both follows $w$ and precedes $r$:*

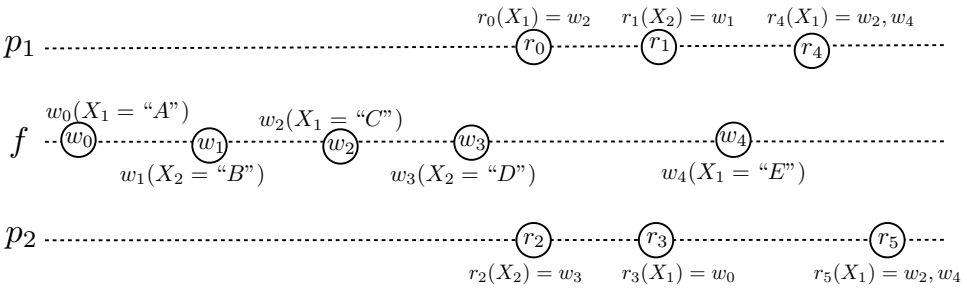
$\forall r, \forall w, w \in r.wl \Leftrightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : (w \prec_G w' \prec_G r \wedge w'.objId = r.objId))$

**VFJC3** Sharing with correct nodes. *If there exists an edge from a vertex $v_1$ at node $p_1$ to a vertex $v_2$ at node $p_2$ then both $p_1$ and $p_2$ are correct in the projection $G_{v_2}$.*

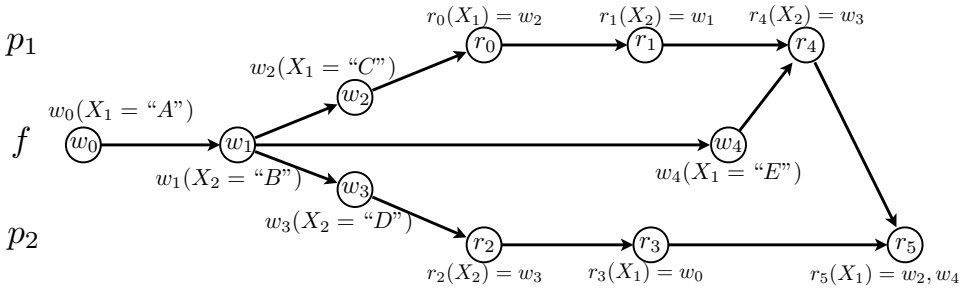
**VFJC4** Time does not travel backward. *For any read/write operations $u$, $v$ by correct nodes:*




(a) Run




(b) Execution




(c) Fork-join-causal but not view-fork-join-causal HB graph

Figure 12: Illustration of view-fork-join-causal consistency and its comparison with fork-join-causal consistency. (a) A run with a faulty node $f$ that issues three non-serial writes $w_2$, $w_3$, and $w_4$, (b) the corresponding execution, and (c) a fork-join-causal happens before graph for this execution. The figures show only the relevant fields of various vertices for brevity. In the run, the faulty node $f$ first issues two non-serial writes $w_2$ and $w_3$ and exposes these writes to nodes $p_1$ and $p_2$ respectively. Subsequently, $f$ issues another write $w_4$ concurrent to $w_2$ and $w_3$ and exposes it to $p_1$. There is no view-fork-join-causally consistent happens before graph for this execution because node $p_1$ has accepted two forks from the faulty node $f$; in a VFJC-consistent execution, correct nodes only accept forks from nodes that are known to be correct. The HB graph is fork-causally consistent because fork-join-causal consistency does not restrict the set of forks that are accepted by correct nodes.

$$u.endTime < v.startTime \Rightarrow v \not\prec_G u.$$

Next, we make the following observation about VFJC HB graphs.

OBSERVATION 5.12. **VFJC Fault-tree creation property.** *For each faulty node in a view-fork-join-causally consistent execution $e$, the VFJC HB graph for $e$ induces a set of fault-trees over the operations of $f$ that are observed by some correct node. This property follows directly from the acyclic nature of a HB graph and the* sharing with correct nodes *property (VFJC3) of VFJC HB graphs—a non-tree graph would violate the VFJC3 property of VFJC consistency.*

*Comparison with fork-join-causal consistency.* VJFC consistency strengthens FJC consistency with the goal of providing a tighter bound for achievable, always-available, and one-way convergent consistency semantics in environments with Byzantine participants. As explained earlier, VFJC consistency admits a limited set of unavoidable forks, in contrast to the unbounded number of forks admitted by FJC consistency. Intuitively, the VFJC3 property—sharing with correct nodes—ensures that the correct nodes only share information with nodes that appear to be correct. Though this check does not guarantee that a correct node never receives updates from a faulty node, it gets close. In particular, the VFJC3 property ensures that in an always-available and one-way convergent (or pairwise convergent) implementation, if a correct node $p_1$ accepts updates from a faulty node $f$ in an execution $e$ then, $p_1$ cannot distinguish $e$ from some other execution $e'$ in which $f$ is correct. We use this intuition to prove a bound on the number of forks that VFJC consistency admits (Theorem 5.49 in Section 5.4).

In addition to limiting the set of admissible forks, VFJC consistency includes the *time does not travel backward* requirement of natural causal consistency to ensure that the precede relation does not violate the real-time assignment of operations issued by correct nodes.

Figure 12 illustrates this difference. Figure 12 extends the scenario illustrated in Figure 9 to form a run which is FJC-consistent but not VFJC-consistent. Like before, the faulty node $f$ issues writes $w_0, w_1$, $w_2$, and $w_3$ and exposes $w_0$, $w_1$, and $w_2$ to $p_0$ and $w_0$, $w_1$, and $w_3$ to $p_1$ thereby forking nodes $p_1$ and $p_2$. Later, the faulty node $f$ creates another fork by issuing another write $w_4$ and sending it to $p_1$. In a FJC consistent execution, $p_1$ can accept this new write because FJC consistency does not preclude accepting multiple forks from a known faulty node. Later, read $r_4$ by node $p_1$ returns writes $w_2$ and $w_4$, from two different forks of the faulty node $f$. In contrast, the VFJC3 condition of VFJC consistency prohibits the correct node $p_1$ from directly accepting two forks from the faulty node $f$. Therefore, no VFJC happens before graph for this execution can exist.

Note that this situation can be easily avoided. The correct node $p_1$ can be configured to not accept new updates from a known faulty node and as soon as node $p_1$ sees the write $w_4$ concurrent to other writes from node $f$, it can infer that $f$ is faulty and reject write $w_4$ coming directly from the faulty node $f$.

A more complicated situation can arise when node $p_1$ is also faulty and colludes with the faulty node $f$. In this case $p_1$ can accept both the forks, knowing that they are from the faulty node $f$ and can later try to expose these forks to the correct node $p_2$. In a FJC consistent execution, $p_2$ can accept both the forks leading to an execution where a faulty node has colluded with another faulty node to expose 3 forks to a correct node.

In contrast, in a VFJC consistent execution, the correct node $p_2$ cannot accept 3 forks in presence of just 2 faulty nodes. VFJC consistency admits only the unavoidable set of forks and is able to minimize the number of forks observed by correct nodes in a given execution. In a VFJC-consistent execution, $p_1$ will have to produce proof that it accepted both the forks without knowing that $f$ was faulty and $p_1$ cannot produce such a proof because $p_1$ must have realized that $f$ is faulty when $p_1$ was accepting the second fork. More concretely, no VFJC consistent happens before graph can exist for this run. We present the argument for this claim next. For the read $r_5$ to return both $w_4$ and $w_2$, there must be path from $w_4$ to $r_5$. The existence of such a path implies that there must be a *fault-edge* from $w_4$ to either $p_1$ or to $p_2$. Let $v$ be the vertex at which such an edge terminates. This fault-edge connecting $w_4$ to $p_1$ (or $p_2$) must violate the VFJC3 condition (*sharing*

*with correct nodes*) because $f$ must be faulty in the projection of $v$, the other end point of this fault-edge. Hence, no VFJC consistency HB graph can exist for this run.

Note that the run shown in Figure 11 is both FJC consistent and VFJC consistent because in that run, all edges satisfy the VFJC3 condition.

## 5.2 CAC Byzantine impossibility results

In this section, we show impossibility results for environments with Byzantine nodes. We start by showing that causal consistency is not achievable using an always-available and pairwise convergent implementation if Byzantine failures can happen (Theorem 5.13). We then strengthen our impossibility result by showing that fork-causal consistency, a weakening of causal consistency designed for environments with Byzantine faulty nodes, is also not achievable using an always-available and pairwise convergent implementation (Theorem 5.14). Note that because pairwise convergence is less convergent than one-way convergence, by showing that no pairwise convergent implementation can provide a certain consistency, we also implicitly prove that no one-way convergent implementation can provide our desired consistency.

THEOREM 5.13. **Causal not achievable.** *Causal consistency and stronger semantics are not enforceable using an always-available and pairwise convergent distributed storage implementation if nodes can be Byzantine.*

*Proof.* Let $S$ be a consistency semantics which is at least as strong as causal consistency. Suppose, for the sake of contradiction, that an always-available and pairwise convergent implementation $I_S$ enforces $S$. Consider an execution of three nodes $p_1, p_2$, and $f$. Nodes $p_1$ and $p_2$ are correct and node $f$ is faulty. In particular, $f$ simulates the actions of two correct nodes, $f_1$ and $f_2$, both with the identity of $f$. Execute the following sequence of operations. Assume that the network drops any messages not described below.

1. Issue a write $w_a$ to an object $a$ at $f_1$ and let it complete. Similarly, issue a write $w_b$ to an object $b$ at $f_2$ and let it complete.
2. Now, let $f_1$ become pairwise converged with $p_1$ by permitting sufficient exchange of messages between $p_1$ and $f_1$. Similarly, let $f_2$ become pairwise converged with $p_2$.
3. Issue reads $r_{a,p_1}$ followed by $r_{b,p_1}$ at $p_1$ and corresponding reads $r_{a,f_1}$, and $r_{b,f_1}$ at $f_1$. Similarly, issue reads $r_{b,p_2}$ and $r_{a,p_2}$ in that order at $p_2$ and corresponding reads $r_{b,f_2}$ and $r_{a,f_2}$ at $f_2$.

We argue that the read $r_{a,p_1}$ at $p_1$ must return the write $w_a$ and the read $r_{b,p_1}$ at $p_1$ must return $\perp$. Because the faulty node $f$ is behaving like a correct node $f_1$ in its interactions with $p_1$, the execution involving the correct node $p_1$ and the faulty node $f_1$ is indistinguishable to the implementation from an execution involving two correct nodes ($p_1$ and $f_1$). Therefore, we can apply the *serial ordering for operations by correct nodes* on the *correct* node $f_1$'s reads to argue that the read $r_{a,f_1}$ must return $w_a$ and the read $r_{b,f_1}$ must return $\perp$. Furthermore, because the reads were issued after $p_1$ and $f_1$ were pairwise converged, the reads $r_{a,p_1}$ and $r_{b,p_1}$ should return the same set of writes as the reads $r_{a,f_1}$ and $r_{b,f_1}$ respectively. Hence, the read $r_{a,p_1}$ at $p_1$ must return the write $w_a$ and the read $r_{b,p_1}$ at $p_1$ must return $\perp$. Similarly, we can argue that the read $r_{b,p_2}$ at $p_2$ must return $w_b$ and the read $r_{a,p_2}$ must return $\perp$.

We argue that no causal HB graph exists for this execution. For the sake of contradiction, assume that a causal HB graph $G$ exists. Both writes $w_1$ and $w_2$ were issued by the same node $f$ and therefore, from the *serial ordering at each node (C1)* requirement, either $w_1$ precedes $w_2$ in $G$ or vice-versa. Suppose that $w_1$ precedes $w_2$ in $G$. In our execution, $p_2$ has observed $w_1$ without observing $w_2$—its read $r_{b,p_2}$ returned $w_2$ but a later read $r_{a,p_2}$ returned $\perp$—violating the *reads return the latest preceding concurrent writes (C2)* requirement of causal consistency. Similarly, $w_2$ preceding $w_1$ is not possible either. Hence, by contradiction, no causal HB graph for this execution can exist and hence, no always-available and pairwise convergent implementation can enforce causal consistency or stronger semantics if nodes can be Byzantine. □

Next, we strengthen our impossibility result by showing that fork-causal consistency, a weakening of causal consistency designed for environments with Byzantine nodes, is also not enforceable using always-available and pairwise convergent implementations. Because fork-causal consistency is weaker than many of the recently proposed forking based consistency semantics [10, 11] designed to tolerate Byzantine nodes, this result establishes that none of these semantics can be enforced without compromising either convergence or availability.

THEOREM 5.14. **Fork-causal not achievable.** *Fork-causal consistency and stronger semantics are not enforceable using an always-available and pairwise convergent distributed storage implementation.*

*Proof.* Let $S$ be a consistency semantics which is at least as strong as fork-causal consistency. Suppose, for the sake of contradiction, that an always-available and pairwise convergent implementation $I_S$ enforces $S$. Consider an execution of three nodes $p_1, p_2$, and $f$. Nodes $p_1$ and $p_2$ are correct and node $f$ is faulty. In particular, $f$ simulates the actions of two correct nodes, $f_1$ and $f_2$, both with the identity of $f$. Execute the following sequence of operations. Assume that the network drops any messages not described below.

1. Issue a write $w_a$ to an object $a$ at $f_1$ and let it complete. Similarly, issue a write $w_b$ to an object $b$ at $f_2$ and let it complete.
2. Now, let $f_1$ become pairwise converged with $p_1$ by permitting sufficient exchange of messages between $p_1$ and $f_1$. Similarly, let $f_2$ become pairwise converged with $p_2$.
3. Issue reads $r_{a,p_1}$ followed by $r_{b,p_1}$ at $p_1$ and corresponding reads $r_{a,f_1}$, and $r_{b,f_1}$ at $f_1$. Similarly, issue reads $r_{b,p_2}$ and $r_{a,p_2}$ in that order at $p_2$ and corresponding reads $r_{b,f_2}$ and $r_{a,f_2}$ at $f_2$.

We first argue that the read $r_{a,p_1}$ at $p_1$ must return the write $w_a$ and the read $r_{b,p_1}$ at $p_1$ must return $\perp$. Because the faulty node $f$ is behaving like a correct node $f_1$ in its interactions with $p_1$, the execution involving the correct node $p_1$ and the faulty node $f_1$ is indistinguishable to the implementation from an execution involving two correct nodes ($p_1$ and $f_1$). Therefore, we can apply the *serial ordering for operations by correct nodes* on the *correct* node $f_1$'s reads to argue that the read $r_{a,f_1}$ must return $w_a$ and the read $r_{b,f_1}$ must return $\perp$. Furthermore, because the reads were issued after $p_1$ and $f_1$ were pairwise converged, the reads $r_{a,p_1}$ and $r_{b,p_1}$ should return the same set of writes as the reads $r_{a,f_1}$ and $r_{b,f_1}$ respectively. Hence, the read $r_{a,p_1}$ at $p_1$ must return the write $w_a$ and the read $r_{b,p_1}$ at $p_1$ must return $\perp$. Similarly, we can argue that the read $r_{b,p_2}$ at $p_2$ must return $w_b$ and the read $r_{a,p_2}$ must return $\perp$.

We claim that the implementation cannot enforce pairwise convergence between $p_1$ and $p_2$ in this state. The primary reason behind this claim is that correct nodes $p_1$ and $p_2$ have observed inconsistent histories that cannot be reconciled without requiring them to observe the concurrent writes $w_1$ and $w_2$ issued by the same node $f$: a violation of the *serial ordering for operations seen by correct node* property that must be enforced by fork-causal consistency and stronger semantics.

We prove our claim by arguing that no fork-causal HB graph exists for this execution. For the sake of contradiction, assume that the implementation $I_S$ can attain pairwise convergence. Let $p_1$ and $p_2$ exchange enough messages to become pairwise converged. Issue reads $r'_{a,p_1}$ and $r'_{a,p_2}$ to object $a$ at $p_1$ and $p_2$ respectively and let these reads finish. Consider the admissible HB graphs for this execution. We argue that in any fork-causal HB graph for this execution, $w_a$ and $w_b$ must be shown as concurrent. Graphs that order these writes ($w_1 \prec w_2$ or $w_2 \prec w_1$) will not be admissible because $p_1$ has observed $w_1$ without observing $w_2$—its reads returned $w_1$ but not $w_2$, while $p_2$ has observed $w_2$ without observing $w_1$—its reads returned $w_2$ but not $w_1$.

Now consider the response returned to reads $r'_{a,p_1}$ and $r'_{a,p_2}$. From one way convergence requirement, both these reads must return the same answer. Furthermore, because only one write $w_a$ has been issued to object $a$, the reads can either return $w_a$ or $\perp$.

If these reads returns $\perp$, then no fork-causal HB graph can satisfy the *reads return the most recent concurrent write property* (FC2) at correct node $p_1$ because a later read $r'_{a,p_1}$ by $p_1$ is returning an answer

than is older than that returned by an earlier read $r_{a,p_1}$ by $p_1$. Conversely, if the reads return $w_a$, then from FC2, we must have $w_a \prec r_{a,p_2}$. Hence, no fork-causal HB graph can satisfy the *serial ordering for operations seen by correct node* at correct node $p_2$ because $p_2$ has observed concurrent operations $w_a$ and $w_b$ from the faulty node $f$. □

Although for simplicity we state the above result in the context of always available, one-way-convergent implementations, the limit is more restrictive. A similar result holds for implementations that allow operations under quorums that are not guaranteed to overlap in at least one correct node. We call such quorum systems *disjoint quorum systems*. Note that any two quorums are allowed to overlap as long as all the overlapping nodes are allowed to be faulty. Such systems include implementations of forking semantics [10, 12, 36, 37, 42, 48], where a potentially faulty server forms the quorum, BFT replicated state machines, which allow smaller quorums [37, 54], and decentralized systems like Depot [41] and Bayou [49], which allow disconnected operations.

For such disjoint quorum systems, we define a weaker availability guarantee, called *disjoint-quorum-availability*. In particular, we say that an implementation is *disjoint-quorum-available* under a quorum $Q$ of servers, if every operation performed by a correct client $c$ eventually completes if every node $p \in Q$ is correct and nodes in $Q \cup c$ can communicate with each other. Similarly, we define a weaker convergence guarantee, called *disjoint-quorum-convergence*, where if a quorum of servers and a set of correct clients can exchange messages, they should be able to converge to a common state where reads return identical response at converged clients. In addition, disjoint-quorum-convergence requires that correct clients should be able to attain eventual consistency.

Now we can show the following theorem:

THEOREM 5.15. *Fork-causal and stronger consistency semantics are not achievable in a disjoint-quorum-convergent and disjoint-quorum-available distributed storage implementation subject to Byzantine node failures.*

*Proof.* (Sketch) Consider two elements $Q_1, Q_2$ of the disjoint quorum system. Consider an execution in which each node $s \in Q_1 \cap Q_2$ is faulty and simulates the behavior of two nodes $s_1$ and $s_2$. Now, we consider three clients $p_1$, $p_2$, and $f$ where client $f$ is faulty and simulates the behavior of two correct clients: $f_1$ and $f_2$ in quorums $Q_1$ and $Q_2$ respectively. Now, as before, we perform write operations at $f_1$ and $f_2$ and use disjoint-quorum-availability to ensure their completion. Next, we force $f_1$ to attain quorum-convergence with $p_1$ using quorum $Q_1$ and similarly force $f_2$ to attain quorum-convergence with $p_2$ using quorum $Q_2$. Finally, we issue reads at $p_1$ and $p_2$ and force these reads to complete using disjoint-quorum-availability. To complete the proof, we argue, as in the proof of Theorem 5.14, that this system cannot attain eventual consistency because correct nodes $p_1$ and $p_2$ have observed inconsistent writes. □

While all these results concern environments with faulty clients, our results can easily be extended environments where clients are assumed to be correct. In such environments, we can prove the following result.

THEOREM 5.16. *Fork-linearizability is not achievable in an eventually consistent and disjoint-quorum-available distributed storage implementation subject to Byzantine server failures.*

*Proof.* Consider two elements $Q_1, Q_2$ of the disjoint quorum system. Consider an execution in which each node $s \in Q_1 \cap Q_2$ is faulty and simulates the behavior of two nodes $s_1$ and $s_2$. Now, we consider two clients $p_1$ and $p_2$ access quorums $Q_1$ and $Q_2$ respectively. Drop all the messages sent from nodes in $Q_1$ to nodes in $Q_2$ or vice-versa for the entirety of this execution. Furthermore, we only permit $p_1$ to communicate with nodes in $Q_1$ and $p_2$ to communicate with nodes in $Q_2$ respectively. Let $p_1$ issue a write $w$ to an object $o$ using the quorum $Q_1$. From the quorum-availability requirement, $w$ should eventually complete. Now, let $p_2$ issue a read $r$ to the object $o$ using the quorum $Q_2$ and let the $r$ complete.

We argue that the read $r$ cannot return the write $w$ because $p_2$ is not aware of $w$ and to $p_2$, the execution is indistinguishable from another execution where the write $w$ was not issued. Since, no other writes were issued during the execution, $r$ must return $\perp$.

To complete the proof, we argue that this system cannot attain eventual consistency without compromising fork-linearizability [46] as no *forking tree* can be constructed for this execution. For the sake of contradiction, assume that the system can attain eventual consistency. We issue reads $r_1$ and $r_2$ at clients $p_1$ and $p_2$ respectively in the eventually consistent state and let these operations complete. Consider the *happens before* (HB) ordering on the *forking group* containing the client $p_1$.

We argue that $w$ must precede $r_1$ in the HB order as $w$ completed before $r_1$ begin and both $w$ and $r_1$ are issued by the same client. Since, no other writes were issued in this execution, $r_1$ must return $w$. Because the system is an eventually consistent state, $r_2$ must also return $w$. From the requirements of fork-linearizability, $w$ must precede $r_2$ in the HB relation. Furthermore, fork-linearizability requires that all operations that happened before $r_2$ must be totally ordered in an order consistent with their real-time. Therefore, $w$, $r_2$, $r$ must be totally ordered in an order consistent with their real-time order: $w$, $r$, $r_2$. Unfortunately, for this HB order to be acceptable for this execution, $r$ must have returned $w$. But $r$ returned $\perp$.

Therefore, by contradiction, no fork-linearizable HB order for this execution can exist and the system cannot be both fork-linearizable and eventually consistent. □

**Forking vs. convergence.** The Byzantine CAC theorem and the follow-up results motivate the need for convergence by showing that existing forking semantics [10, 12, 36, 37, 42, 48] cannot be enforced by available and convergent implementations. The implementations of these forking semantics restrict misbehavior by clients or servers to *forking*: showing divergent histories to different nodes. Using the notion of forking, these implementations can provide greater fault-tolerance than traditional approaches [14, 16, 26, 28, 43]. Unfortunately, because existing semantics permit fork-detection but not fork-repair, they compromise convergence and have limited usefulness in practical settings. This conflict between convergence and forking in existing forking semantics prompts the question: do there exist useful forking based consistency semantics that can be enforced by convergent and available implementations?

We argue that indeed, there exist practically useful semantics that can be enforced by always-available, one-way convergent, and Byzantine-fault tolerant implementations. Depot's *fork-join-causal (FJC)* consistency [41] is an example of such semantics. FJC consistency departs from existing forking semantics by allowing correct clients to *join forks*, that is, to incorporate the divergence into a sensible history, allowing them to ensure convergence in the face of faults. Specifically, a correct node regards a fork as logically concurrent updates by two *virtual nodes*. At that point, correct nodes can handle forking by faulty nodes using the same techniques [9, 20, 30, 51, 55] that they need anyway to handle a better understood problem: logically concurrent updates during disconnected operation.

In the next section, we provide a one-way convergent, and always-available implementation for VFJC consistency.

### 5.3 VFJC is enforceable by an always-available and one-way convergent implementation

Theorem 5.14 rules out implementations of fork-causal consistency and stronger semantics that can be implemented by an always-available, one-way or pairwise convergent implementations. We next answer the question: What consistency *can* be provided using an always-available and one-way or pairwise convergent implementation? In Section 5.1, we introduced view-fork-join-causal (VFJC) consistency that we believe is close to the strongest enforceable semantics using an always-available, and one-way convergent implementation in an environment with Byzantine nodes. In this section, we show that VFJC is enforceable using an always-available and one-way convergent implementation. We show this result by construction by providing an implementation that enforces VFJC and is always-available and one-way convergent. Note that because, every one-way convergent implementation is also pairwise convergent, it follows that VFJC consistency can

```
1   Messages
2     Update := {NodeID nodeID, OID oid, Value value,
3                Set⟨Hash⟩ prevUpdate}σ_nodeID
4     // prevUpdate is a set containing hashes
5     // of updates that were superseded by u

7   State (at each node p)
8     State := {Set⟨Hash⟩lastWrites, Update lastLocalWrite,
9     HashMap{oid, ⟨Hash⟩}store, Map{Hash, Update}log,
10    NodeID myNodeID, Set⟨NodeID⟩ nodes, (K_p, K_u)}

12    // lastWrites: Set storing the hash of
13    // last unsuperseded non local updates
14    // lastLocalWrite: last local update
15    // store: Map storing the set of most
16    // recent update hashes for every object oid
17    // log: map of updates received indexed
18    // by their hashes
19    // (K_p, K_u): RSA key pair for signing

21    MyState := local state

23  Methods
24  synchronized function read(OID oid):
25    return MyState.store{oid}

27  synchronized function write(OID oid, Value value):
28    writeInt(MyState, oid, value)
29    createView(MyState)

31  function writeInt(State s, OID oid, Value value):
32    create a new signed update u such that
33    u.prevUpdates := s.LastWrites ∪ s.LastLocalWrite
34    u.oid := oid
35    u.value := value
36    u.nodeID := myNodeID
37    return intApply(s, u)

39  function send():
40    while(true)
41      synchronized
42        Let T be a list of all updates sorted
43        in an order in which they were added
44        to MyState.Log
45      foreach p ∈ MyState.nodes send T to p
46      sleep 30

48  synchronized function pktApply(Set⟨Update⟩pkt):
49    if(pkt contains only view updates)
50      return false
51    State testState = MyState.copy()
52    if(intPktApply(testState, pkt)
53      intPktApply(MyState, pkt)
54    else return false
55    return true

57  function intPktApply(State s, Set⟨Update⟩pkt):
58    status = true
59    foreach w ∈ pkt
60      status = status ∧ intApply(s, w)
61    status = status ∧ lastWriteViews(s)
62    if (status ∧ createView(s))
63      return true
64    else return false

66  function lastWritesViews(State s)
67      // return true if all the updates in
68      // state.lastWrites are view updates

70  function intApply(State s, Update u):
71    if(s.log.containsKey(Hash(u)))
72      return true
73    else if(verify(s,u))
74      // update state
75      foreach l ∈ s.lastWrites
76        if(prec(s, l, u))
77          s.lastWrites := s.lastWrites − {l}
78      if(u.nodeID = s.myNodeID)
79        s.LastLocalWrite := u
80      else
81        s.lastWrites := s.lastWrites ∪ {u}
82      s.log{Hash(w)} = u
83      foreach l ∈ s.Store[u.oid]
84        if(prec(s, l, u))
85          s.Store{u.oid} := s.Store{u.oid} − {l}
86      s.Store{u.oid} := s.Store{u.oid} ∪ {u}
87    else
88      return false

90  function verify(State s, Update u):
91    return signed(u.nodeID, u) ∧
92           historyLocal(s, u) ∧
93           noFaultyChild(s, u)

95  // returns if the history of u is present
96  // in local history of state s
97  function historyLocal(State s, Update u):
98      foreach v ∈ u.prevUpdates
99        if(s.log.containsKey(Hash(v)))
100         return false
101     return true

103 // returns false if none of the immediate
104 // child or creator of w is faulty
105 function noFaultyChild(State s, Update u):
106     if(u.nodeID ∉ getFaulty(s, u))
107       foreach c ∈ u.prevUpdates
108         if(c.nodeID ∈ getFaulty(s, u))
109           return true
110     return false

112 function getFaulty(State s, Update u):
113     // return the set of nodeIds that have
114     // concurrent updates that precede u
115     // in State s

117 // returns true if u occurs in history of v
118 function prec(State s, Update u, Update v):
119   if (v = ⊥) return false
120   foreach H_w ∈ v.prevUpdates
121    w = s.log{H_w}
122    if (u = w)
123      return true
124    else if (prec(s, u, w))
125      return true
126   return false

128 // creates a view
129 function createView(State s):
130   return writeInt(s, VIEW, VIEW)
```

Figure 13: A VFJC-consistent implementation.

also be implemented by an always-available and pairwise convergent implementation.

### 5.3.1 VFJC protocol

This protocol is based on three key ideas which are oriented towards ensuring the desired properties of always-availability, one-way convergence, and VFJC consistency. For always-availability, we require each node to maintain enough state so that it can serve reads of arbitrary objects without needing to contact other nodes. Similarly, a node should also be able to perform a write without consulting other nodes. For one-way convergence, a node should be able to receive messages from other correct nodes such that processing these messages makes the receiver aware of all the operations that the sender is aware of. Finally, to ensure VFJC consistency, we must ensure that each message carries enough information to guard against faulty nodes and correct nodes must stop communicating with nodes that are deemed to be faulty.

Figure 13 describes our VFJC implementation. For simplicity of the existentiality proof, the implementation described here is inefficient. Depot shows an efficient implementation of a similar protocol.

At the core of our VFJC protocol is the standard log-exchange protocol similar to the one used in Bayou [49], PRACTI [8], or Depot where each write produces an *update* with an object identifier and the object value (lines 27-37). In addition, each update contains a representation of its history to determine precedence of updates. Our implementation makes this history tamper-evident by signing each update and attaching to each update $u$, a cryptographic hash of the most recent update(s) that $u$ supersedes (line 33). When computed recursively, the cryptographic hash of this recent update set is sufficient to encode the entire history on which an update purports to depend.

Each node also maintains a *store* (line 9) with the most recent update(s) to each object (lines 83-86) and a *log* (line 9) that stores all the updates that the node has received or created (line 82). On a read of an object $o$, the node returns the most recent update(s) to $o$ from its local store without requiring any communication (lines 24-25). Conversely, on a write, an update is created and added to the local store and the local log at the issuing node (lines 27-29). Our implementation is able to ensure availability because in our implementation, correct nodes do not need to communicate with other nodes to complete reads or writes.

Nodes periodically exchange updates from their local logs; newly received updates are appended to the local log and then used to update the local store of a node, replacing any old updates that precede the new update (lines 48-55). Each node in our implementation periodically sends its log to all other nodes to ensure one-way convergence (lines 39-46).

Our protocol includes a number of mechanisms to safeguard against Byzantine faulty nodes (lines 90-93). First, correct nodes check the received updates against their local history to prevent omission and reordering (lines 97-101) and to detect concurrent updates from the same node, a.k.a. *forks* (lines 111-114). Second, nodes that produce forks are considered to be faulty (lines 111-114) and correct nodes stop communicating directly with such nodes (line 93). Third, nodes ensure that updates created by known faulty nodes are not accepted unless some other potentially correct node has accepted them without knowing that they were created by a faulty node (line 61). This relaxation of update checks (compared to existing fork-X protocols [10, 12, 36, 37, 42, 48]) allows forked correct nodes to *join* forks and ensure one-way convergence in our protocol. Conceptually, a node joins a fork by treating forked writes by a faulty node as concurrent writes by different *virtual* correct nodes [41]. This approach is appealing because we can reduce the problem of Byzantine faults to the well-studied problem of handling concurrency and conflicts in optimistic consistency systems [9, 20, 27, 30, 51, 55]. Finally, to bound the number of forks, a node issues a *view* update when it receives updates from another node. Intuitively, a view update is a proof issued by a node that none of the updates it accepted were issued by known faulty nodes (line 29).

We next prove that our implementation ensures the CAC properties—Consistency (VFJC consistency), availability (always-availability), and convergence (one-way convergence). We prove this claim by showing that any run of our implementation in Figure 13 is VFJC-consistent. Let $\rho$ be a run of our implementation and $e$ be the corresponding execution. Recall that an execution $e$ consists of a set of read and write operations

with a few additional fields associated with each operation. Each write operation includes the `nodeId`, `objId`, `value`, `startTime`, and `endTime` fields. The `startTime` and `endTime` are chosen using the real-time when the operations start and end. Likewise, each read operation includes the `nodeId`, `objId`, `writeList`, `startTime`, and `endTime` fields.

We prove that the execution $e$ is VJFC-consistent by describing the construction to produce a VFJC HB graph for $e$. We first define a precede relation ($\prec$) on the read and write operations in $e$ (§5.3.2). We use this partial order to prove that our implementation is (1) consistent (§5.3.4, Lemma 5.39)—by constructing a VFJC HB graph , always-available (§5.3.5, Lemma 5.34)—by showing that operations do not block, and one-way convergent (§5.3.6, Lemma 5.38)—by showing that connected nodes can exchange updates (Lemma 5.37) and nodes with identical updates return identical responses on reads (Lemma 5.36).

### 5.3.2 Defining a VFJC HB graph and precede relation

We define a HB graph by defining a precede ($\prec$) relation. Recall that VFJC HB graph contains vertices, called views, in addition to the vertices for read and write operations. So, we first describe the process of creating these view vertices from a run of our implementation. A view vertex (also referred to as a view operation) corresponds to an invocation of the `createView` function in our execution and contains only the `nodeId` field. We use the term vertices and operations interchangeably in our discussion.

We next define a set of *accepted* operations that will be included in the BFJC HB graph. As per the requirement for VFJC consistency, we must ensure that all the operations by correct nodes are part of our HB graph. In addition, we must ensure that writes by faulty nodes that have been returned by some read are also included in the HB graph. Definition 5.17 describes the set of accepted operations precisely.

DEFINITION 5.17. *A write operation $w$ (or a view operation $v$) is* accepted *by a correct node $p$ if $p$ has added the corresponding update $w$ (or $v$) to the log at its local state. Note that we make a distinction between when an update is added to the copy of the log (e.g. line 52) vs when the update is added to the actual log (e.g. line 53). Also note that a node directly adds the updates it has created to the actual log (lines $28-29$). A read operation $r$ is* accepted *by a correct node $p$ if $r$ is issued by $p$.*

Intuitively, our goal in defining the HB graph is to capture the data flow in an execution by identifying the operations that precede a given operation. We use two techniques to accomplish this goal. First, we use the *history* of a write or a view to determine the set of write and view operations that precede it. As indicated in pseudocode line 3, each update has a $prevUpdates$ field that contains hashes of all immediately superseded updates. This field can be used to compute all the superseded updates using recursion as shown in Definition 5.18.

However, reads do not have a corresponding update. Hence, to order reads with respect to writes, views, and other reads, we define an *accept time* for each node $p$ and each operation $u$ in our system. Intuitively, the accept time $ts_{p,u}$ indicates at which the node $p$ completes processing of operation $u$. Using the accept time, we can order reads after all previously accepted operations. We next define the history of an operation and the accept time of accepted operations for each operation-node pair.

DEFINITION 5.18. *For an update $u$, define* history *of $u$ (denoted by $H_u$) as a set of updates such that $\forall v \in H_u \exists w_1, w_2, ..., w_k : H(v) \in w_1.prevUpdates, H(w_1) \in w_2.prevUpdates, ..., H(w_k) \in v.prevUpdates$. $H(u)$ denotes the cryptographic hash of update $u$.*

DEFINITION 5.19. *The* accept time *for an operation $v$ that is accepted by a correct node $p$ is denoted by $ts_{p,v}$, and is defined as follows:*

- *For a read $r$ issued by a correct node $p$, $ts_{p,r}$ indicates the real time when the monitor lock was acquired in Step 24 (get method call) at $p$.*
- *For a write or view $w$ accepted by a correct node $p$, $ts_{p,w}$ indicates the real time when the $intApply$ method was called to add $w$ to $p$'s log.*

- *For all other operation-node tuples $(v, p)$, $ts_{p,v}$ is undefined.*

Recall that nodes issue at most one outstanding request in our model (§2). Hence, we have the following corollaries that will prove useful in our proof later.

COROLLARY 5.20. *For a read or write operation $u$, $u.startTime < ts_{p_u,u} < u.endTime$ (Recall that $p_u$ denotes the node at which the operation $u$ was issued).*

COROLLARY 5.21. *For operations $u, v$ issued at the same correct node $p$, $u.startTime < v.startTime \Leftrightarrow ts_{p,u} < ts_{p,v}$.*

After having defined the set of operations that will be included in the HB graph and the ordering relations (accept time and history) that we will use to define our precedes relation, we next define our precede relation in two steps. First, we define a $PRED$ set for each operation in our execution. Then, using these $PRED$ sets, we define the precede relation as the transitive closure of the set containment relation on $PRED$ sets.

DEFINITION 5.22. *The set $PRED_v$ for an operation $v$ accepted by a correct node is defined as follows:*

1. *If $v$ is a write/view operation accepted by some correct node then $PRED_v$ includes all $u : H(u) \in v.prevUpdates$.*
2. *If $v$ is a read operation issued by a correct node $p$, then $PRED_v$ includes all operations $u$ accepted by $p$ such that $ts_{p,u} < ts_{p,v}$.*
3. *If $v$ is a write operation issued by a correct node $p$, then $PRED_v$ includes all read operations $u$ accepted by $p$ such that $ts_{p,u} < ts_{p,v}$.*
4. *For any other operation $u$, $PRED_u = \phi$.*

*We denote $u \in PRED_v$ as $u \prec_{PRED} v$.*

DEFINITION 5.23. *Define $u$* precedes *$v$ ($u \prec v$) if*

1. *$u \prec_{PRED} v$, or*
2. *$u \prec_{PRED} w$ and $w \prec v$.*

Given the definition of our precedence relation (Definition 5.23) and our $PRED$ sets (Definition 5.22), it is easy to see the corollary stated below. Intuitively, this corollary states that each directed path from a read $r$ at a node $p$ to a non-local operation $o$ at a node $q(\neq p)$ must go through a write $w$ at $p$. We use this corollary in our proof below to eliminate reads from paths in our HB graph. The paths without reads are based solely on the *history* of updates (Definition 5.22) and hence easier to analyze; the history of an update is verified by a correct node before accepting it.

COROLLARY 5.24. **Writes follow reads in any directed path.** *Consider a directed path $r \prec_{PRED} u_1 \prec_{PRED} ... \prec_{PRED} u_k$. $p_r \neq p_{u_k} \Rightarrow \exists l : 1 \leq l < k \wedge p_{u_l} = p_r \wedge u_l$ is a write/view operation.*

### 5.3.3 Properties of our precedence relation

We now prove a number of properties of our precedence relation that we use later to prove the safety and liveness properties of our implementation. We start by showing that our precedence relation is acyclic, a key requirement for a VFJC HB graph.

LEMMA 5.25. **The precedence relation ($\prec$) is acyclic.**

*Proof.* For the sake of contradiction, assume that a path $C$ exists with a cycle: $u_1 \prec_{PRED} u_2 \prec_{PRED} ... \prec_{PRED} u_k \prec_{PRED} u_1$. First consider the case when $C$ has no reads. Without loss of generality, assume that $u_1$ was accepted first by a correct node $p$. But for $u_1$ to be accepted by a correct node $p$, the

*historyLocal* check should have passed for $u_1$. However, for the *historyLocal* check to pass, $p$'s log must contain $u_k$, contradicting the assumption that $u_1$ was accepted first.

Now consider a cycle with at least one read $u_i$. We show that such a cycle is not possible either. First, we note that $p_{u_i}$ must be correct because the $PRED$ set for reads by faulty nodes is empty. We further note that $p_{u_i}$ must have accepted $u_{i-1}$ (from the definition of the $PRED$ set for reads). We now split this cycle into the maximum number of sub-paths $C_1, C_2, ..., C_l$ such that

1. Each sub-path starts and ends at an operation issued by some correct node.
2. No operation by a correct node is present inside any sub-path (i.e. operations that are not end-points of a sub-path must be issued by faulty nodes).
3. Each sub-path ends at an operation at which the next sub-path begins.

Suppose the above procedure produces $l$ sub-paths. Because the cycle $C$ contains a read, it must contain at least two operations that are issued by a correct node (from Corollary 5.24, a read must be followed by a write by the same node). Hence, our sub-path construction is feasible and it must produce at least two sub-paths ($l \geq 2$).

From the definition of $PRED$ (Definition 5.22[2]), each sub-path of length greater than 2 must contain only write/view operations at all positions except, perhaps, the last position. The internal operations must be writes/views as they must be issued by faulty nodes as per the sub-path construction constraint 2. The first operation cannot be a read because it is included in the $PRED$ set of an operation by a faulty node; reads are only included in the $PRED$ set of operations by correct nodes.

Let $s_k, e_k$ denote the starting and end vertices of a sub-path $C_k$. Now, we make the following two claims about the accept time(s) of operations in these sub-paths that together show that a cycle with one or more read operations is not possible:

LEMMA 5.26. **Accept times within a sub-path are in an increasing order.** *For each sub-path $C_k$, we show that $p_{e_k}$ accepts $s_k$ and $e_k$, and $ts_{p_{e_k},s_k} < ts_{p_{e_k},e_k}$.*

*Proof.* The proof for this lemma is as follows. There are four types of sub-paths: $r \prec_{PRED} r$, $r \prec_{PRED} w$, and $w \prec_{PRED} w \prec_{PRED} ... \prec_{PRED} w$, $w \prec_{PRED} w \prec_{PRED} w \prec_{PRED} ... \prec_{PRED} r$, (from the construction of sub-paths), where $w$ denotes a view/write operation and $r$ denotes a read operation. In the first two cases, the desired result follows from the definition of $PRED$ and from the fact that the $PRED$ set of reads is only non-empty for reads issued by correct nodes.

Next, consider a sub-path $C_k$ of the third type: $w_1 \prec_{PRED} w_2 \prec_{PRED} ... \prec_{PRED} w_i$. We must have $w_1 \in w_2.prevUpdates, w_2 \in w_3.prevUpdates, ..., w_{i-1} \in w_i.prevUpdates$ from the definition of $PRED$ for writes. Now, for the *historyLocal (Step* 97*)* check to pass, $w_1$ must have been accepted by $p_k$ prior to accepting $w_2$. Similarly $w_2$ must have been accepted before $w_3$ and so on. Therefore, we must have $ts_{p_{e_k},w_1} < ts_{p_{e_k},w_2} < ... < ts_{p_{e_k},w_i}$. Hence, we get our desired result.

Finally, consider the last type of sub-path: $w \prec_{PRED} w_1 \prec_{PRED} ...w_k \prec_{PRED} r$. If this sub-path contains only two elements, $w$ and $r$, then the desired result follows from the definition of $PRED$ for reads. If the sub-path contains more than two elements, then $ts_{p_r,w_k} < ts_{p_r,r}$, from the definition of $PRED$. And, $ts_{p_r,w} < ts_{p_r,w_1} < ... < ts_{p_r,w_k}$ following the argument similar to the third case: each write/view must have been accepted before the prior write/view. Combining these, we get our desired result. □

LEMMA 5.27. **Accept times across sub-paths are in a non-decreasing order.** *For consecutive sub-paths, $C_k$, $C_{k+1}$, $ts_{p_{e_k},e_k} \leq ts_{p_{e_{k+1}},s_{k+1}}$.*

*Proof.* We start by noting that $e_k = s_{k+1}$—requirement 3 in the sub-path construction. If $p_{e_k} = p_{e_{k+1}}$ then the above claim follows trivially as $ts_{p_{e_k},e_k} = ts_{p_{e_{k+1}},s_{k+1}}$. If $p_{e_k} \neq p_{e_{k+1}}$ then $e_k$ cannot be a read operation from Corollary 5.24. Hence $e_k$ must be a view/write operation. In this case, we must have

$ts_{p_{e_k},e_k} < ts_{p_{e_{k+1}},e_k}$ as it will take some finite time for $e_k$ to propagate from the issuing node $p_{e_k}$ to the receiving node $p_{e_{k+1}}$. □

Combining Lemma 5.26 and Lemma 5.27, we get that $ts_{p_{e_1},s_1} < ts_{p_{e_1},e_1} \leq ts_{p_{e_2},s_2} < ts_{p_{e_2},e_2} \leq ... \leq ts_{p_{e_l},s_l} < ts_{p_{e_l},e_l} \leq ts_{p_{e_1},s_1}$. Since $l \geq 2$, the above real time assignment is not feasible and hence, by contradiction, no cycles with one or more read can exist. Combining this result with the argument earlier that cycles without reads are not possible, we get that no cycles can exist in our HB graph built using the precede relation. □

We next show updates performed by correct nodes supersede all prior updates accepted by that correct node. This lemma will help us in proving the *serial ordering at correct nodes* and *reads return the latest preceding concurrent writes* requirements of VFJC consistency.

LEMMA 5.28. **New updates supersede older ones.** *Let $w_1$ be a write/view operation accepted by a correct node p, and $w_2$ be a write/view operation issued by p. Then, $ts_{p,w_1} < ts_{p,w_2} \Rightarrow w_1 \in H_{w_2}$.*

*Proof.* If $w_1 \in w_2.prevUpdates$, then the desired result follows. Now, consider the case when $w_1 \notin w_2.prevUpdates$. $p$ had accepted $w_1$ when it issued $w_2$ and hence $w_1$ must have been added to the *lastWrites*. So, for $v$ to be removed from $lastWrites$, $p$ must have applied a series of updates $v_1, v_2, ..., v_l$ such that $w_1 \in v_1.prevUpdates, v_1 \in v_2.prevUpdates, ..., v_l \in w_2.prevUpdates$. Hence, $w_1 \in H_{w_2}$. □

Our next lemma show that our precedence relation respects the serial ordering requirement of VFJC consistency.

LEMMA 5.29. **Serial ordering at correct nodes.** *If $p_v = p_{v'}$ and $p_v$ is correct, then $ts_{p_v,v} < ts_{p_v,v'} \Leftrightarrow v \prec v'$.*

*Proof. "if".* $ts_{p_v,v} < ts_{p_v,v'} \Rightarrow v \prec v'$. Recall that we assume that no outstanding operations are issued at any node (Section 2). If either $v$ or $v'$ is a read, then the desired result follows from Definition 5.22[2,3]. Otherwise if both $v$ and $v'$ are non-reads then $ts_{p_v,v} < ts_{p_v,v'} \Rightarrow v \in H_{v'} \Rightarrow v \prec v'$ (using Lemma 5.28 and Definition 5.22[1]).
*"only if".* $ts_{p_v,v} < ts_{p_v,v'} \Leftarrow v \prec v'$. Both $v$ and $v'$ are performed by the same correct node and we know that operations performed by a node have non-overlapping execution times in our execution. Therefore we must have either $ts_{p_v,v} < ts_{p_v,v'}$ or $ts_{p_v,v'} < ts_{p_v,v}$. If $ts_{p_v,v} < ts_{p_v,v'}$ then the desired follows. If not, then by the "if" part shown above, we must have $v' \prec v$. Combining $v' \prec v$ with $v \prec v'$, we get that our precedence relation must be cyclic violating Lemma 5.25. Therefore, by contradiction, this scenario is not possible. □

LEMMA 5.30. **Precedes implies containment in history.** *Let $w_2$ be a write/view operations accepted by a correct node p, then $w_1 \prec w_2 \Rightarrow w_1 \in H_{w_2}$.*

*Proof.* Consider the path with fewest reads: $w_1 \prec_{PRED} u_1 \wedge u_1 \prec_{PRED} u_2 \wedge ...u_k \prec_{PRED} w_2$. If the operations in this path are writes then Lemma 5.30 follows from the definition of history (Definition 5.18) and the definition of $PRED$ for writes (Definition5.22[1]).

In the remaining part of this proof, we will show by contradiction that there must exist a path with no reads. Suppose, for the sake of contradiction, assume that the path with fewest reads has $m(> 0)$ reads. Let $u_l$ be one of the reads. Using Corollary 5.24, there must exist a later write $u_j$ such that $p_{u_j} = p_{u_l} = p$, where $p$ is correct (only reads from correct nodes are included in the definition of $PRED$) and $ts_{p,u_l} < ts_{p,u_j}$ (from serial ordering at correct nodes—Lemma 5.29). Now, consider the following two cases on $u_{l-1}$.

$u_{l-1}$ **is a read.** In this case, $p_{u_{l-1}} = p$ because only local reads are included in the $PRED$ set for reads. Furthermore, we must also have $u_{l-1} \prec_{PRED} u_j$ from the definition of $PRED$ for writes. Now we can construct an alternative path from $w_1$ to $w_2$ that does not have the read $u_l$: $w_1 \prec_{PRED} u_1 \wedge u_1 \prec_{PRED} u_2 \wedge ... \wedge u_{l-1} \prec_{PRED} u_j \wedge ...u_k \prec_{PRED} w_2$. This path will have one less read

than the earlier path because we have removed the read $u_l$, contradicting the assumption that our path has fewest reads.

**$u_{l-1}$ is not a read.** In this case, we must have $u_{l-1} \in H_{u_j}$ (from Lemma 5.28). From the definition of $PRED$ (Definition 5.22[1]) and precedes (Definition 5.23), it follows that we must have a set of writes $v_1 ... v_m$ such that $u_{l-1} \prec_{PRED} v_1 \prec_{PRED} ... \prec_{PRED} v_m \prec_{PRED} u_j$. Again, we can construct an alternative path from $w_1$ to $w_2$ with at least one less read than the earlier path, contradicting the assumption that our path has fewest reads.

Hence, by contradiction, it must be the case that the path with fewest reads has no reads in which case our desired result follows. □

LEMMA 5.31. **Correct nodes accept operations in $\prec$ order.** *If $w$ is a write/view and $v$ is an operation accepted by a correct node $p$, then $w \prec v \Rightarrow ts_{p,w} < ts_{p,v}$.*

*Proof.* By induction on the number of reads in the path $C_{min}$ with the fewest reads: $w \prec_{PRED} u_1 \prec_{PRED} u_2 \prec_{PRED} ... \prec_{PRED} u_k \prec_{PRED} v$. If all the operations are writes (base case), then Lemma 5.31 is true because of the *historyLocal* check in the *verify* function in the pseudocode: $p$ will only accept $v$ when it has already accepted $u_k$, and $p$ will only accept $u_k$ when it has already accepted $u_{k-1}$ and so on. Hence, $p$ must have accepted $w$ before $v$ which gives us our desired result.

Suppose the lemma holds true for up to $m$ reads by induction hypothesis. Consider the case when the path with fewest reads has $m + 1$ reads. We examine two cases.

**Case 1: $v$ is a read.** If the number of elements in the path is two, the result follows from the definition of $PRED$ for reads. If the number of elements is more than two, we have $u_k \in PRED_v$. Because $v$ is a read operation, it follows from the definition of $PRED$ for reads that $ts_{p,u_k} < ts_{p,v}$. We also have $w \prec u_k$ with one less read in the path. Hence, from the induction hypothesis, we have $ts_{p,w} < ts_{p,u_k}$. Combining these two results, we get our desired result.

**Case 2: $v$ is not a read.** From Lemma 5.30, we get that there must exist writes $u_1 ... u_m$ such that $w \prec_{PRED} u_1 \prec_{PRED} ... \prec_{PRED} u_m \prec_{PRED} v$ giving a path with no reads. Hence, from the induction hypothesis, we have $ts_{p,w} < ts_{p,u_k}$.

Combining these cases, we get our desired result. □

### 5.3.4 Consistency

We next prove that our protocol enforces VFJC consistency by showing that any execution of our protocol can be mapped to a VFJC HB graph.

LEMMA 5.32. **VFJC consistency lemma.** *Every execution of the pseudocode in Figure 13 is VFJC consistent.*

*Proof.* Let $e$ be an arbitrary execution of the pseudocode in Figure 13. We next describe the construction of a VFJC HB graph for $e$. Consider the graph $G$ constructed using the precedence relation as follows:

1. $G$ contains vertices for all operations accepted by some correct node. In particular, $G$ contains vertices for all operations issued by a correct node and writes/views by faulty nodes that have been accepted by some correct node.

2. $G$ contains a directed edge from a vertex $u$ to $v$ if $u \rightarrow v$.

We make the following observation about our constructed graph $G$.

OBSERVATION 5.33. *It is easy to see that for any two vertices $u$ and $v$, $u \prec_G v \Leftrightarrow u \prec v$. We do not introduce paths between vertices that do not have paths and likewise, we do not disrupt connectivity between vertices that were connected in the precedence relation. Given this equivalence, we use $u$ precedes $v$ ($u \prec v$) to imply that $u$ precedes $v$ in the HB graph $G$ ($u \prec_G v$) and vice-versa.*

Now, we need to show that the resulting graph is acyclic and upholds the conditions of a VFJC HB graph.

**$G$ is acyclic.** Follows from Lemma 5.25 which showed that the precedence relation is acyclic and from Observation 5.33.

**Serial ordering for operations by correct nodes.** If $v.nodeID = v'.nodeID$ and $nodeID_v$ is correct, then $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$. Let $p = nodeID_v = nodeID_{v'}$. Now, from Corollary 5.20, we know that $v.endTime < v'.startTime \Rightarrow ts_{p,v} < ts_{p,v'}$. Furthermore, because our implementation issues at most one request at a time, it follows that $ts_{p,v} < ts_{p,v'} \Rightarrow v.startTime < v'.startTime$. Combining these results, we get the following temporal equivalence property: $ts_{p,v} < ts_{p,v'} \Leftrightarrow v.startTime < v'.startTime$.

Now, combining Lemma 5.29 with Observation 5.33, we get the following serial ordering property based on accept times: $ts_{p,v} < ts_{p,v'} \Leftrightarrow v \prec_G v'$. Combing this serial ordering property with the temporal equivalence property derived in the previous paragraph, we get our desired serial ordering property based on the $startTime(s)$ of operations.

**Reads return the latest preceding concurrent writes.** For any vertex $r$ corresponding to a read operation of object *objId*, $r$'s *writeList* $wl$ contains all writes $w$ of *objId* that precede $r$ in $G$ and that have not been overwritten by another write of *objId* that both follows $w$ and precedes $r$:

$w \in r.wl \Leftrightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : (w \prec_G w' \prec_G r \wedge w'.objId = r.objId))$

**"if".** $w \in r.wl \Rightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : w \prec_G w' \prec_G r \wedge w'.objId = r.objId.$

First, we will prove that $w$ precedes $r$ and $w$ updates the object with id $objId$. A correct node $p$ issues a read $r$ to object id $objId$ that returns a set $wl$ of writes. Reads to an object id $oId$, return the value of the $store$ map for key $oId$ (line 25). We make two observations. First, $w$ must be a write to the object id $objId$ because only writes to object id $objId$ are added to the $store$ map entry for $objId$ (line 86). Second, $w \in r.wl$ implies that $w$ must have been accepted by $p$ before it issued $r$ (i.e. $ts_{p,w} < ts_{p,r}$); writes are first added to the log and then to the store (as described in the *applyInt* function) and the monitor lock must be released before processing a subsequent read $r$. From the definition of $PRED$ for reads (Definition 5.22[2]), $ts_{p,w} < ts_{p,r} \Rightarrow w \prec_{PRED} r$. From the definition of precede, we have $w \prec r$ and hence, $w \prec_G r$.

Next, we will show that there does not exist another write $w'$ to the object Id $objId$ such that $w$ precedes $w'$ and $w'$ precedes $r$. Suppose, for the sake of contradiction, there exists $w'$ such that $w \prec_G w' \prec_G r$. From Observation 5.33 we must have $w \prec w' \prec r$ and from Lemma 5.31, $w'$ should have been accepted by $p$ after it has accepted $w$ but before it issued $r$. Furthermore, $w \prec w' \Rightarrow w \in H_{w'}$ from Lemma 5.30. Hence the $prec$ invocation at line 84 must have returned true. Therefore, by pseudocode line 85, $p$ would have removed $w$ from the $store$ map entry for object id $objId$ on accepting $w'$ and would not have returned $w$ on read $r$ contradicting the assumption that $w$ was returned on the read $r$. Hence, by contradiction, this case is true.

**"only if".** $w \in wl \Leftarrow w \prec_G r \wedge \nexists w' : w \prec_G w' \prec_G r$. For the sake of contradiction assume that the read $r$'s writeList $wl$ does not include the write $w$. Let $p$ be the correct node that issued $r$. $w \prec_G r$ implies $w \prec r$ (Observation 5.33), which in turn implies that, from Lemma 5.31, $p$ received $w$ before issuing read $r$. Therefore, by the pseudocode Step 86, $w$ must have been added to the $store$ map with the key as $objId$. Suppose that a subsequent write $w'$, accepted by $p$, removes $w$ from the $store$ at pseudocode line 86. For this removal, we must have $w.objId = objId$ and the $prec$ invocation at line 85 must

have returned true. Therefore, by the $PRED$ construction for writes/views (Definition 5.22[1]), we must have $w \prec w'$ and hence $w \prec_G w'$. Also, by construction (Definition 5.22[2]), $w' \prec r$ and hence $w' \prec_G r$. But the precondition in our desired result precludes such a write $w'$. Therefore, by contradiction, the read $r$ must have returned the write $w$.

**Sharing with correct nodes.** If there exists an edge from a vertex $v_1$ at $p_1$ to a vertex $v_2$ at $p_2$ then both $p_1$ and $p_2$ are correct in the projection of vertex $v_2$. Recall that a node $p$ is correct in the projection of a vertex $v$ if vertices of node $p$ in the subgraph $G_v$ are totally ordered.

We first show that $p_1$ must be correct in the view of vertex $v_2$. Suppose, for the sake of contradiction, that $p_1$ is faulty in the view of vertex $v_2$. $p_1$ cannot be a correct node because a correct node must be correct in the projection of all vertices—$VFJC1$ requirement of the VFJC consistency. Recall that only writes and views from faulty nodes are included in the HB graph. Therefore, $v_1$ must be a write or a view vertex. Now, consider the following two cases:

1. $v_2$ **is a view/write operation**: We claim that we must have $v_1 \in v_2.prevUpdates$; otherwise there exists some other write/view operation $v$ such that $v_1 \prec v \prec v_2$ violating the assumption that $v_1$ and $v_2$ are connected through an edge. Now, consider when a correct node $p$ accepted $v_2$. $p$ must have checked that nodes that issued updates in $v_2.prevUpdates$ are correct in the projection of $v_2$ from the check in *noFaultyChild* function. Therefore, none of the updates in $v_2.prevUpdates$ can be from a node that is faulty in the projection of $v_2$.

2. $v_2$ **is a read operation**: $p_2$ must be correct because we do not include read operations by faulty nodes. Since, $p_2$ is correct and $p_1$ is faulty, they can only be connected by a non-local edge. Now, consider when $v_1$ was accepted by $p_2$. After processing the packet that contains $v_1$, $p_2$ must have performed a view operation $v$ (From Step 62) and hence $v_1 \prec v$ (from Lemma 5.28). The read $v_2$ must be performed after the view operation completes. Hence, we must have that $v \prec v_2$ (from the definition of $PRED$ for reads). Combining these two observations, we must have $v_1 \prec v \prec v_2$, contradicting the assumption that $v_1$ and $v_2$ are connected through a non-local edge.

Next, we show that $p_2$ must be correct in the view of vertex $v_1$. Again, for the sake of contradiction, assume that $p_2$ is faulty. As in the previous case, $p_2$ must be a faulty node for this contradiction to be true; correct nodes are correct in all views. So, $v_2$ must be a write/view operation from our construction of HB graph. Furthermore, if $p_2$ is faulty in the view of $v_1$, it must also be faulty in the view of vertex $v_2$; subgraph $G_{v_2}$ is a superset of the subgraph $G_{v_1}$. Now, we argue that $v_2$ cannot be accepted by any correct node (owing to line 106), contradicting the assumption that $v_2$ is included in the HB graph.

□

### 5.3.5 Availability

We next show that our implementation is always-available for operations issued by correct nodes. To do so, we show that all invocations of $read$ and $write$ methods in our pseudocode complete successfully. The intuition behind this claim is that our protocol does not force nodes to communicate with other nodes to complete an operation: reads can be completed by just reading the local state while writes require both reading and writing the local state. Therefore, operations by correct nodes must complete. Lemma 5.34 formalizes this intuition.

LEMMA 5.34. **VFJC availability theorem.** *The pseudocode in Figure 13 is always-available for reads and writes by correct nodes.*

*Proof.* We first observe that invocations of the $write$ and $read$ methods do not require any communication with other nodes before returning in our implementation described in Figure 13. Hence, a node does not have to block waiting for a message from another node. Furthermore, because nodes perform a bounded

number of operations while holding the shared lock, they must release the lock in a finite and bounded amount of time. This argument suffices for reads because as per our definition of accept (Definition 5.19), all reads created by a correct node are accepted. We need to show one additional property for writes: the update corresponding to a write and the view update created after applying the write will be accepted by the issuing correct node.

We first claim that the *lastWrites* set at a correct node is always empty after the lock is released in our pseudocode. The proof for this claim is as follows. Initially, the *lastWrites* set must be empty. After a successful invocation of the $write$ method, it will again be empty as the new view update (line 29) will supersede all previous entries. Similarly, after a successful packet receipt, the *lastWrites* set will again be empty as the new view update (line 62) will supersede all previous entries.

Next we claim that the new write update and the following view update created on a $write$ method invocation will be successfully applied. (1) The *historyLocal* check (line 92) will pass as the *prevUpdates* is assigned from *lastWrites* and *lastLocalWrites*; both these sets contain updates that have already been accepted. (2) The *noFaultyChild* check (line 93) will pass because the only update in the *prevUpdates* is from the correct node that is creating the update children. (3) Finally, the *signed* check (line 91) will pass because a correct node will correctly sign the update. Hence, the update and the view update will both be accepted and applied to the log, ensuring that subsequent reads return the values written by these writes. □

### 5.3.6 Convergence

Finally, we show that our implementation is one-way convergent. We designed our protocol to permit any pair of correct connected nodes to exchange updates. Furthermore, in our protocol, different correct nodes use the same $prec$ relation to decide precedence. The former property ensures that correct connected nodes will receive the same set of updates while the latter ensures that nodes that have received identical set of updates will return identical responses on reads.

Our proof roughly corresponds to the intuition above. We first show that nodes that have observed identical set of updates will return identical responses on reads of identical objects (Lemma 5.36). Next, we show that any pair of correct connected nodes can achieve a state where they have observed identical set of updates (Lemma 5.37). Combining these results, we get that any pair of correct connected nodes can achieve one-way convergence (Lemma 5.38).

LEMMA 5.35. **Identical updates imply identical precedence.** *Let $e$ be an execution in which correct nodes $p$ and $q$ have accepted identical updates. Let $p$ and $q$ issue reads $r_p$ and $r_q$ respectively to an object $o$. For a view/write operation $u$, $u$ precedes $r_q$ iff $u$ precedes $r_p$.*

*Proof.* We first prove the "only if" part of this lemma. $u \prec r_q$ implies $ts_{q,u} < ts_{q,r_q}$ (Using Lemma 5.31: correct nodes accept operations in $\prec$ order). $ts_{q,u} < ts_{q,r_q} \Rightarrow ts_{p,u} < ts_{p,r_p}$ (from the assumption that they have accepted identical updates when they issued $r_p$ and $r_q$ respectively). $ts_{p,u} < ts_{p,r_p} \Rightarrow u \prec r_p$ from the definition of the $PRED$ set for reads (Definition 5.22[2]). Similarly, we can prove the "if" part. □

LEMMA 5.36. **Identical updates imply identical responses.** *Correct nodes $p$ and $q$ that have accepted identical updates, return identical responses on reads of the same object. ($w \in r_p.writeList \Leftrightarrow w \in r_q.writeList$).*

*Proof.* Let $e$ be an execution in which correct nodes $p$ and $q$ have accepted identical updates. Now, let $p$ and $q$ issue reads $r_p$ and $r_q$ respectively to an object $o$. We first prove the "only if" part of this lemma.

Suppose, for the sake of contradiction, that $w \in r_p.writeList \wedge w \notin r_q.writeList$. We know from VFJC2, that $w \prec r_q$ and using Lemma 5.35, we get that $w \prec r_p$. So, for $r_p$ to not return $w$, there must exist a write $w'$ to object $o$ such that $w \prec w' \prec r_p$ (from VFJC2—reads return the latest preceding concurrent writes) . Again using Lemma 5.35, we get that $w' \prec r_q$. But this contradicts VFJC2 for the read $r_q$ performed by the correct node $q$: there exists a write $w'$ such that $w \prec w' \prec r_q$ and yet, the read $r_q$ to object

$o$ returned $w$. Therefore, by contradiction, no such $w'$ can exist and all the writes in $r_p.writeList$ must also be present in the $r_q.writeList$. Similarly, we can prove the "if" part of this lemma and obtain our desired result. □

LEMMA 5.37. **Correct connected nodes accumulate updates.** *A correct node $q$ processes an update packet $T$ from a correct node $p$ that generated the packet $T$ when it has accepted updates $U_p$. We claim $U_p \subseteq U_q$ after $q$ processes the packet $T$ irrespective of whether $T$ was accepted or rejected by $q$.*

*Proof.* Let $U_{pre}$ be the set of updates that $q$ has accepted prior to receiving $T$. If $U_p \subseteq U_{pre}$ then the desired result follows as updates are never removed from a node's log.

Consider the case when $U_p \not\subseteq U_{pre}$. Its easy to see that the desired condition ($U_p \subseteq U_q$) must become true if the packet is successfully applied; all the updates in $U_p$ will be added to $U_q$, thereby forcing the desired condition. We next show that the packet must indeed be successfully applied. By way of contradiction, consider the reasons for rejection of the packet $T$.

**An update fails the *verify* check (line 73) in the *intApply* function (line 60).** Let $u$ be the first update (if any) that fails the *verify* check in the *intApply* function. $p$ is a correct node and it accepted the update $u$. Therefore, $u$ must have a valid signature that the correct node $q$ should also be able to verify. Hence, the *signed* check must pass (line 91). The *historyLocal* check (line 92) must also be satisfied for the following reason. When $p$ accepted $u$, its log included all updates included in the $prevUpdates$ set of $u$. Hence, all these updates must be present in the list $T$. Furthermore, all updates in the $prevUpdates$ set of $u$ must be ordered before $u$ in $T$; $p$ accepted these updates prior to accepting $u$ and $T$ is sorted based on the accept time of updates. Finally, because $u$ is the first rejected update, it follows that all updates in the $prevUpdates$ set of $u$ were accepted and added to the log. Therefore, the *historyLocal* check cannot fail for $u$. The *noFaultyChild* check (line 93) must also pass because $u$'s children (updates whose hashes are included in $u.prevUpdates$) must be correct in the projection of $u$ as otherwise, the correct node $p$ would have rejected $u$. Hence, by contradiction, no such update $u$ can exist.

**The *lastWriteViews* check fails at Step 61.** We next show that this is not possible. We first claim that each correct node satisfies the following *last-write-view* invariant: at each correct node $r$, there must exist a view update $v$ such that for all updates $v(\neq u)$ in $r$'s log, $prec(u, v) = true$. The argument behind this claim is as follows. If the last operation executed at $r$ was a write, then the result follows from pseudocode Step 29 and the availability requirement that the view operation will not be rejected. If the last operation was a successful packet application ($pktApply$—line 48), then the result follows from pseudocode Step 60. Reads and failed packet applications do not result in any state change and hence do not affect the validity of the last-write-view invariant. Therefore, the last-write-view invariant must hold true.

Now we argue that the *lastWriteViews* check must pass. Consider when $q$ processes $p$'s packet: the *lastUpdate* will consist of a view update from the correct node $p$ and (optionally) a view update from $q$. Hence, the *lastWriteView* check at node $q$ must pass as the *lastWrite* set and *lastLocalWrite* set will together contain at most two updates from $p$ and $q$, both of which will be view updates from the last-write-view invariant.

**The *intApply* check (line 70) invoked from the $createView$ method (line 62) fails.** We show that the *intApply* check invoked from the line 62 at the correct node $q$ for the application of packet $T$ must succeed. Let $v$ be the new view update created in the invocation of the $createView$ method. From line 33, $v$'s $prevUpdates$ set will contain updates from the *lastWrite* set and the *lastLocalWrite*. As argued above, the *lastWrite* set and the *lastLocalWrite* set at $q$ will together contain at most two view updates from $p$ and $q$. Hence the *noFaultyChild* check must pass for $v$ as both $p$ and $q$ are correct.

Similarly, the *historyLocal* check must also pass because all the updates in *prevUpdates* set of the new view operation have already been applied to *log*. Finally, the *signed* check must also pass because the view update will be properly signed by the correct node $q$ who creates the update.

□

LEMMA 5.38. **VFJC convergence theorem.** *The pseudocode listing 1 is one-way convergent.*

*Proof.* Lemma 5.37 shows that by two one-way transfers of update packets, any two correct connected nodes can reach a state where both of them have identical set of updates in their logs. Lemma 5.36 shows that correct nodes that have accepted identical updates will return identical responses on reads. Intuitively, we can combine these two lemmas to argue that our system is one-way convergent because it can force two correct connected nodes to attain a converged state, in which reads to identical objects return identical responses, by using two one-way transfers of update packets. We next make this intuition precise.

In particular, we argue that by one one-way transfer of updates from $p$ to $q$, two correct connected nodes $p$ and $q$ can reach an intermediate state with the following property. In this intermediate state, $q$ will eventually send an update packet to $p$ such that after applying $q$'s packet, both $p$ and $q$ will have accepted an identical set of updates and hence will return identical responses on reads of identical objects (from Lemma 5.36). Therefore, the intermediate state must be a semi-pairwise converged state for $p$ and $q$ and our implementation must be one-way convergent. □

### 5.3.7 CAC achievability with Byzantine participants

THEOREM 5.39. **Byzantine CAC achievability theorem.** *View fork join causal (VFJC) consistency can be enforced by an always-available and one-way convergent implementation in presence of Byzantine faulty participants.*

*Proof.* Consider the implementation described by the pseudocode in Figure 13. Lemma 5.38 shows that the implementation is one-way convergent, Lemma 5.34 shows that the implementation is always-available, and Lemma 5.39 shows that the implementation admits VFJC consistent executions. Furthermore, the implementation admits Byzantine faulty participants. Combining these results, we get that VFJC is enforceable by an always-available, one-way convergent implementation in presence of Byzantine participants. □

The following corollary follows from knowing that in absence of Byzantine nodes, VFJC consistency reduces to natural causal consistency.

COROLLARY 5.40. **CAC achievability.** *Natural causal consistency can be enforced by an always-available and one-way convergent implementation if all failures are omission failures.*

Next we note that a one-way convergent implementation is also pairwise convergent. Therefore, the following corollaries follows.

COROLLARY 5.41. **Pairwise Byzantine CAC achievability.** *View fork join causal (VFJC) consistency can be enforced by an always-available and pairwise convergent implementation in presence of Byzantine faulty participants.*

COROLLARY 5.42. **Pairwise CAC achievability.** *Natural causal consistency can be enforced by an always-available and pairwise convergent implementation if all failures are omission failures.*

## 5.4 Bounding forks in VFJC consistency

In this section, we show that VFJC consistency can bound the number of forks, in addition to the set of forks, that are admitted in an execution. Our proof exploits the properties of the VFJC HB graph to identify a bound on the number of forks that correct nodes can observe.

We first define precisely what it means for a correct node to observe a fork. Recall that a fork is a directed path from a root of the fault-tree to the leaf and the fault-tree is derived from the HB graph. We next define a few terms that help simplify the description of our proof.

DEFINITION 5.43. **Observing a fork.** *A fork $f$ is said to be* observed *by a correct node $p$ if $p$ has observed a vertex $v$ that belongs to the fork $f$ such that $v$ does not belong to any other fork.*

DEFINITION 5.44. **Disconnected HB graph.** *A HB graph in which there are no outgoing edges originating at correct nodes is called a* disconnected *HB graph. Note that in a disconnected HB graph, no path exists from a vertex at a correct node to another vertex at some other correct node.*

DEFINITION 5.45. **Maximally-forked HB graph.** *A HB graph that has the maximum number of forks for a given number of correct and faulty nodes is called a* maximally-forked *HB graph.*

We next derive a few lemmas to aid the derivation of our upper bound. We start by showing that it is sufficient to limit our search for maximally-forked HB graphs to disconnected HB graphs because there exists a disconnected HB graph that is maximally-forked (Lemma 5.46). We then argue that in a maximally-forked HB graph, correct nodes must observe independent forks (Lemma 5.47). Finally, we derive a recursion on the fork bound of HB graphs by showing that a maximally-forked HB graph with one correct node consists of subgraph that are also maximally forked (Lemma 5.48). Using this recursion, we derive our fork-bound theorem for VFJC consistency (Theorem 5.49).

LEMMA 5.46. **There exists a maximally-forked and disconnected HB graph.** *For any number of correct and faulty nodes, there exists a maximally-forked and disconnected HB graph.*

*Proof.* Consider an arbitrary HB graph that is maximally-forked. Now remove the outgoing edges from the correct nodes to construct a disconnected HB graph. It is easy to see that this transformation does not reduce the number of forks observed by correct nodes. The number of forks is defined by the total number of branches of faulty nodes observed by all correct nodes taken together. While this transformation may affect the number of forks observed by a particular correct node, it does not affect the overall set of forks observed by all correct nodes taken together; the path from the faulty node that created a fork to the first correct node that observed that fork is still preserved. Note that the resulting graph is disconnected by construction and maximally-forked from the argument presented above. Hence the desired result. □

We next derive the upper bound on the number of forks that can be observed by correct nodes in a VFJC execution. Let $MaxFork(c, f)$ denotes the maximum number of distinct forks (from any faulty node) observed by correct nodes in any execution of a system with $c$ correct nodes and $f$ Byzantine faulty nodes.

LEMMA 5.47. **Correct nodes observe independent forks in a maximally-forked HB graph.** $MaxFork(c, f) \leq c \cdot MaxFork(1, f)$.

*Proof.* We know from Lemma 5.46 that there exists a disconnected HB graph for each combination of correct and faulty nodes that is maximally-forked. So, we focus our attention on disconnected HB graphs in this proof. Let $G$ be a disconnected HB graph that is maximally-forked for $c$ correct and $f$ faulty nodes. We now make two observations. First, because $G$ is maximally-forked, no two correct nodes should observe the same fork; exposing the same fork to multiple correct node does not add to the total number of forks. Second, because the HB graph is disconnected, to each correct node it appears like a HB graph for 1 correct node and $f$ faulty node.

Combining these observations, we make two claims. First, each correct node $p$ will observe at most $MaxFork(1, f)$ forks with $f$ faulty nodes. Second, because forks produced for different correct nodes must be different in a maximally-forked HB graph, we can have a maximum of $c \cdot MaxFork(1, f)$ forks in a maximally-forked HB graph with $c$ correct nodes and $f$ Byzantine faulty nodes. Hence, the desired result. □

LEMMA 5.48. **MaxFork recursion lemma for VFJC HB graph.** $MaxFork(1, f) \leq 2 \cdot MaxFork(1, f-1) + 1$.

*Proof.* Let $G$ be a maximally-forked, disconnected, and VFJC-consistent HB graph. Let $u$ be the last operation from a faulty node that was observed by the lone correct node $p$. Let $v$ be the earliest vertex at $p$ such that $u \rightarrow v$. Let $G^v$ denote the projection of the graph $G$ such that $G^v$ contains all vertices $w : w \prec_G v$ and all edges connecting these vertices. Since $p$ is the only correct node in $G$, and $u$ is the last operation accepted by $p$ in $G$, we know that (1) $p_u$ must be correct in $v$ (Definition 5.11[VFJC3]), (2) all operations by $p_u$ in $G$ must be totally ordered and precede $u$.

Observe that $G_v$ looks like a HB graph with 2 correct nodes and $f-1$ faulty nodes. Hence, from Lemma 5.47, the maximum number of forks that these two correct nodes can observe in such a HB graph is $2 \cdot MaxFork(1, f-1)$. It follows that that maximum number of forks from the $f-1$ faulty nodes that the correct node $p$ can observe must also be bounded by $2 \cdot MaxFork(1, f-1)$. Finally, note that $p$ observes a fork from $p_u$, without recognizing it as a fork yet, in addition to the $2 \cdot MaxFork(1, f-1)$ forks that it did recognize. Combining these observations, the maximum number of forks that a correct node can observe in presence of $f$ Byzantine faulty nodes is $2 \cdot MaxFork(1, f-1) + 1$. □

THEOREM 5.49. **VFJC bounds forks.** *In a VFJC HB graph for $f$ faulty nodes and $c$ correct nodes, at most $c \cdot (2^f - 1)$ distinct forks can be observed by correct nodes.*

*Proof.* From Lemma 5.48, we have $MaxFork(1, f) = 2 \cdot MaxFork(1, f-1) + 1$. Solving this recursive relation using the base case of $MaxFork(1, 1) = 1$, we get $MaxFork(1, f) \leq 2^f - 1$. Combining this with the result of Lemma 5.47, we get $MaxFork(c, f) \leq c \cdot (2^f - 1)$. □

# 6 Discussion

We started this research with the goal of strengthening the CAP theorem to prove the conjecture that causal consistency is the strongest semantics achievable with high availability. We also wanted to prove a similar bound using fork-join-causal consistency for environments with Byzantine nodes. Ultimately, our final results differ from our intended results in several ways that we discuss below.

- Why did we introduce the CAC formulation instead of using the CAP formulation? Why did we introduce an explicit convergence requirement and why did we omit the partition-tolerance requirement?
- Why causal consistency is not the strongest achievable consistency? Why did we introduce natural causal consistency?
- Why did we use one-way convergence instead of two-way convergence or eventual consistency?
- Why do we not have a tight bound for environments with Byzantine failures?

We now discuss the key insight behind these differences and sketch a few interesting questions that are open for further research.

**Why CAC and not CAP?** The CAP result talks about consistency, availability, and partition-resilience. Why do we instead talk about the CAC properties? We discuss a few aspects that motivated this divergence from the CAP result.

First, the CAP (consistency, availability, partition-resilience) formulation mixes properties (consistency and availability) with the system model (network reliability assumptions). In our formulation, we decouple the model from the properties so that we can separately consider bounds on properties achievable under both omission and Byzantine failure models.

Furthermore, in our formulation, always availability corresponds to the AP (availability despite partitions) properties in the CAP formulation.

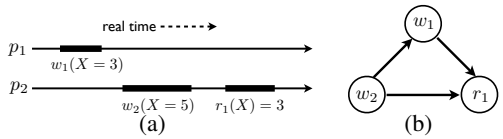


Figure 14: A causal execution (a) and its corresponding HB graph (b). Note that no NC-consistent HB graph for this execution exists as the implementation has picked an earlier (in global time) write as the winner.

Second, we explicitly add convergence to the availability and consistency properties in our tradeoff. Convergence property is the key innovation of this work and we need convergence to explore a wide range of consistency semantics in a useful manner. The CAP theorem does not explicitly consider *convergence* because it is concerned with linearizability as its consistency requirement and linearizability embeds a strong convergence requirement; completed operations become immediately visible to all subsequent reads. In contrast, even slightly weaker consistency semantics such as sequential consistency can be implemented in a non-convergent way by returning stale responses on reads. The need for an explicit convergence requirement becomes even more apparent when we examine weaker semantics like causal consistency.

Finally, we note a key similarity between CAP and CAC tradeoffs: both these results assume an arbitrary workload. Both CAP and CAC results can be strengthened, if workload can be restricted. For example, one can indeed provide strong consistency despite network partitions if objects are partitioned among nodes and a node is only permitted to issue reads and writes to objects in its partition.

**Open questions.** There are strong consistency semantics that are incomparable with linearizability but that seem "unnatural" [22] (e.g., always return the original value of an object regardless of what writes occur.) Perhaps convergence provides a principled way to prefer some strong semantics over others? Is linearizability the strongest semantic for some natural convergence requirement?

As the CAP theorem indicates, different system models yield different tradeoffs; a *CAC-M* framework would generalize CAC by exposing the model as a parameter. CAP and this paper showed consistency-availability tradeoffs for an unreliable network with Byzantine and omission failures. What other bounds on consistency exist under different availability requirements (e.g., wait, lock, or obstruction freedom), convergence requirements (e.g., two-way convergent or eventually consistent), or environment models? For example, we conjecture that weak fork-linearizability [10] is the strongest consistency semantics that can be enforced with a Byzantine faulty server if wait-freedom is desired when the server is correct.

**Why is causal not the strongest?** Here we illustrate the happens before graph for an execution that is causal but not natural causal. As before, only the relevant details are shown for brevity. An implementation could produce this execution by, for example, deciding that a write by the node with the lowest nodeID dominates other causally-concurrent updates.

The technical strengthening of natural causal consistency is needed to ensure that the proof works out. However, we find that the conjecture that causal consistency is the strongest achievable consistency semantics is still right in its spirit. natural causal consistency is a natural strengthening of causal consistency which is implemented by all the practical systems that we are aware of [8, 39, 41, 55] and we see no obvious reason why an implementation would prefer to implement causal consistency over the stronger natural causal consistency semantics.

**Open questions:** Are there other natural strengthening of causal that are incomparable with natural causal but still highly available and usefully convergent? Is there a way to characterize different families of strengthening into some design space of metrics or tradeoffs?

**Why one-way convergence?** The weaker *pairwise convergence* property requires a pair of nodes to converge only when bidirectional communication is possible.

We focused on one-way convergence because it is needed in theory and useful in practice. In our theory, natural causal is not the strongest always-available consistency semantics under pairwise convergence

because nodes could conspire to impose order among logically-concurrent updates while exchanging those updates. In practice, one-way convergence captures the spirit of anti-entropy protocols [8, 19, 49]. Although most implementations use bidirectional communication, the communication from the update-receiver to the update-sender is just a (significant) performance optimization used to avoid redundant transfers of updates already known to the receiver. One-way convergence is also important in protocols that transmit updates via delay tolerant networks [20, 49, 57].

**Open questions:** How do the CAC tradeoffs vary as we weaken convergence requirements? Although we know that we can strengthen consistency if we only require pairwise convergence, so far we have only identified "artificial" and not-obviously-useful variations. Are there useful, stronger consistency guarantees that can be provided with weaker, but still useful, convergence requirements?

**Why no tight bound for Byzantine?** We would have liked to show that VFJC consistency, which minimizes the number of forks admitted in an execution, is the strongest available and convergent semantics in the presence of Byzantine nodes. Unfortunately, this conjecture is false—an implementation can enforce technically stronger, albeit unnatural, consistency semantics that disallow certain VFJC consistent executions.

For example, consider a VFJC consistent execution $e$ in which a faulty node $f$ issues two logically concurrent writes $w_1$ and $w_2$ to an object $o$. Consider an implementation $I$ which is derived from an always-available and one-way convergent implementation for VFJC consistency $I_{VFJC}$. $I$ behaves like $I_{VFJC}$ in all the executions except for the execution $e$ described above. For execution $e$, $I$ suppresses the actual concurrency and pretends that one of the writes (say $w_1$) precedes the other (say $w_2$). Such an execution, in which the faulty node issues two serial writes $w_1$ and $w_2$ is already admitted by both $I_{VFJC}$ and $I$. Because $e$ is admitted by VFJC consistency, $I$ can enforce a slightly stronger, *VFJC-limited* consistency, using a highly available and one-way convergent implementation— $I$ has the same availability or convergence of as $I_{VFJC}$.

Fortunately, in the absence of Byzantine nodes, we can use the C3 constraint of natural causal consistency to rule out such strengthenings by arguing that the implementation must respect the real-time order of operations issued by correct nodes. Unfortunately, we cannot apply a similar argument for operations issued by Byzantine nodes as these operations do not have a well defined `startTime` or `endTime`. Therefore, we cannot construct a similar proof for environments with Byzantine nodes.

**Open questions.** Do useful strengthenings of VFJC consistency exist? Is there a tight bound for a strongest (hopefully useful!) consistency semantic that can be achieved with high availability and useful convergence?

# 7 Related work

Several prior efforts have explored the limits of consistency when other properties are desired. The CAP tension between consistency and availability in systems with unreliable networks is well known [17, 24, 53]. Similarly, there is a fundamental tension between strong consistency and performance [38]. Frigo defined the weakest "reasonable" memory model by imposing specific constraints, such as *constructability*, *nondeterminism confinement*, and *classicality* [23]. Frigo also noted the problem of defining "trivial yet strong" semantics that we address by using convergence properties [23].

To ensure availability, many systems have implemented causal consistency and weaker semantics with various conflict resolution policies [2, 8, 18, 19, 25, 49, 50, 52]. To avoid picking a winner among different conflict resolution policies, we follow some previous systems [8, 19, 41, 50] and permit a read to return a set of concurrent writes.

As noted above, most systems that claim to implement causal consistency actually implement stronger semantics (e.g. NC). Lloyd et al. [39] explicitly note that their system's *causal+* semantics are stronger than causal consistency. In particular, these semantics enforce a variation of causal consistency in which writes to each key are totally ordered. This natural strengthening has been implemented by a number of

past systems [7, 47] by, for example, associating a Lamport clock [33] with each write and then imposing a highest-accept-stamp-wins conflict resolution policy.

Traditionally, Byzantine faults have been addressed using quorums [44, 45], sometimes within state machine replication systems [5, 14, 26]. For sufficiently large quorums, traditional semantics such as regular, safe, or atomic registers [34] or linearizability [29] can be enforced, while weaker forking consistency variations can be enforced on smaller quorums or even individual machines [10, 12, 36, 37, 42, 48]. Whereas traditional semantics are unavailable when quorums are unresponsive, many forking semantics allow faulty nodes to introduce *permanent* partitions among correct nodes [41]. In this paper, we show that this problem is fundamental to these semantics.

Cachin et al. [10, 11] expose the tradeoff between consistency and availability in forking semantics by showing that neither *fork-sequential* consistency [48], nor *fork-* linearizable* consistency [37] can be enforced by a wait-free implementation using a Byzantine faulty server.

*Eventual propagation* [21] requires all the nodes to observe all the updates but, unlike convergence, does not demand different nodes to order updates or converge to a common state.

# 8 Conclusion

This paper examines the tradeoff between consistency and availability in fault-tolerant distributed systems. In environments with network failures, we strengthen the CAP theorem to show that *natural causal* consistency, a strengthening of causal consistency that respects the real-time ordering of operations, is the strongest semantics achievable while retaining strong liveness guarantees. Similarly, we show that in the Byzantine failure model, fork-causal [41] or stronger semantics cannot be implemented without compromising liveness. The key to both these results is the use of *convergence* as a liveness requirement. Convergence precludes uninteresting semantics that gain their strength by disallowing nodes from observing each other's writes. Finally, we show the existence of consistency semantics that are enforceable by convergent and always-available implementations in both omission- and Byzantine-failure prone environments.

# References


[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.

[2] M. Ahamad, J. Burns, P. Hutto, and G. Neiger. Causal memory. In *WDAG*, pages 9–30, 1991.

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, 1995.

[4] Amazon SimpleDB. `http://aws.amazon.com/simpledb`.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine Replication Under Attack. In *DSN*, 2008.

[6] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12:91–122, May 1994.

[7] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication (extended version). http://www.cs.utexas.edu/users/dahlin/papers/PRACTI-2005-10-extended.pdf, Oct. 2005.

[8] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[9] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM (CACM)*, 25(4), 1982.

[10] C. Cachin, I. Keidar, and A. Shraer. Fail-Aware Untrusted Storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2009.

[11] C. Cachin, I. Keidar, and A. Shraer. Fork sequential consistency is blocking. *Inf. Process. Lett.*, 109:360–364, March 2009.

[12] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*, 2007.

[13] The Apache Cassandra Project. `http://cassandra.apache.org`.

[14] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4), 2002.

[15] Choosing Consistency - All Things Distributed. `http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html`, Feb 2010.

[16] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[17] B. Coan, B. Oki, and E. Kolodner. Limitations on database availability when networks partition. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*, 1986.

[18] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[19] 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 *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[20] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed filesystem for challenged networks in developing regions. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[21] R. Friedman, R. Vitenberg, and G. Chockler. On the composability of consistency conditions. *Information Processing Letters*, 86(4):169 – 176, 2003.

[22] M. Frigo. The weakest reasonable memory model. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Jan. 1998.

[23] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1998.

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

[25] R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, 1992.

[26] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2010.

[27] R. Guy, J. Heidemann, W. Mak, T. Page, G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer*, 1990.

[28] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[29] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), July 1990.

[30] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–5, Feb. 1992.

[31] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 43–57, New York, NY, USA, 1990. ACM.

[32] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

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

[34] L. Lamport. On interprocess communication, 1985.

[35] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[36] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[37] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[38] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton, 1988.

[39] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[40] P. Mahajan, S. Lee, J. Zheng, L. Alvisi, and M. Dahlin. ASTRO: Autonomous and trustworthy data sharing. Technical Report TR-09-29, University of Texas at Austin, Department of Computer Sciences, Oct. 2008.

[41] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–12, Berkeley, CA, USA, 2010. USENIX Association.

[42] M. Majuntke, D. Dobre, M. Serafini, and N. Suri. Abortable fork-linearizable storage. In *OPODIS*, 2009.

[43] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.

[44] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pages 203–213, 1998.

[45] J. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine quorums. In *Symposium on Distributed Computing (DISC)*, Oct. 2002.

[46] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 108–117, New York, NY, USA, 2002. ACM.

[47] A. muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, 2002.

[48] A. Oprea and M. Reiter. On consistency of encrypted files. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2006.

[49] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[50] V. Ramasubramanian, T. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[51] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the USENIX Summer Technical Conference*, 1994.

[52] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for distributed workstation environments. *IEEE Transactions on Computers*, 39(4):447–459, Apr. 1990.

[53] A. Siegel. *Performance in Flexible Distributed File Systems*. PhD thesis, Cornell, 1992.

[54] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[55] 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 *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[56] W. Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, Jan. 2009.

[57] R. Wang, S. Sobti, N. Garg, E. Ziskind, J. Lai, and A. Krishnamurthy. Turning the postal system into a generic digital communication mechanism. In *Proceedings of the ACM SIGCOMM*, pages 159–166, 2004.