Consistency, Availability, and Convergence

Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin

Department of Computer Science, The University of Texas at Austin

Technical Report (UTCS TR-11-22)

Abstract

We examine the limits of consistency in fault-tolerant distributed storage systems. In particular, we identify fundamental tradeoffs among properties of *consistency*, *availability*, and *convergence*, and we close the gap between what is known to be impossible (i.e. CAP) and known systems that are highly-available but that provide weaker consistency such as causal. Specifically, in the asynchronous model with omission-failures and unreliable networks, we show the following tight bound: **No consistency stronger than Real Time Causal Consistency (RTC) can be provided in an always-available, one-way convergent system** and **RTC can be provided in an always-available, one-way convergent system.** In the asynchronous, Byzantine-failure model, we show that it is impossible to implement many of the recently introduced *fork*-based consistency semantics without sacrificing either availability or convergence; notably, proposed systems allow Byzantine nodes to permanently partition correct nodes from one another. To address this limitation, we introduce *bounded fork join causal semantics* that extends causal consistency to Byzantine environments while retaining availability and convergence.

[*] All correspondence should be addressed to: E-mail: princem@cs.utexas.edu; Address: Prince Mahajan, 1 University Station, #C0500, Austin, TX 78712-0233; Telephone: 512.366.0512

1 Introduction

In this paper, we examine the limits of consistency in fault-tolerant distributed storage systems. We identify fundamental tradeoffs between the safety property of *consistency* and the liveness properties of *availability* and *convergence*, where *consistency* constrains the order that reads and writes may appear to occur, *availability* requires reads and writes to complete, and *convergence* requires connected nodes to observe one another's updates.

In asynchronous systems with omission failures and unreliable networks, we close the gap between what is known to be impossible (i.e. CAP [10, 17, 40]) and known systems that are highly-available but provide weaker consistency such as causal [1, 4, 18, 23, 29, 36, 39]. In particular, we show the following tight bound:

- No consistency stronger than *real time causal* (RTC) consistency, a strengthening of causal consistency, can be provided in an *always-available*, *one-way convergent* system.
- *RTC* can be provided in an *always-available*, *one-way convergent* system.

An *always available* system allows reads and writes to complete regardless of which messages are lost and a *one-way convergent* system guarantees that if node p can receive from node q, then eventually p's state reflects updates known to q.

In systems that can suffer Byzantine failures, we do not establish a tight bound, but we come close. First, we show the following lower bound:

- No system in which nodes can be Byzantine can enforce *fork causal* or stronger consistency in an always-available, one-way convergent system.

Notice that this result rules out always-available and convergent implementations of many recently proposed *fork-X* semantics [6, 8, 26, 27, 31, 35]. 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.

Second, we show that, fortunately, it is not necessary to sacrifice liveness to have sensible, well-defined consistency semantics despite Byzantine failures:

- *Bounded fork join causal* can be provided to correct nodes in an always available, one-way convergent system with an arbitrary number of Byzantine nodes.

Fork causal [30] and *bounded fork join causal* (BFJC) are weakenings of causal consistency that deal with Byzantine nodes. The basic idea is to treat inconsistent writes by a Byzantine node as concurrent writes by multiple virtual nodes. BFJC limits the number of forks accepted from a faulty node and thus bounds the number of virtual nodes needed to represent each faulty node.

In addition to the four specific results above, a central contribution of this paper is the formalization of the convergence property and the CAC trade-offs. The trade-offs between consistency and availability are well known [10, 17, 40], but absent a convergence requirement, there exist many semantics that (1) are stronger than causal consistency, (2) are always available to reads and writes, and (3) "feel" somehow "artificial" or "less useful" than causal. Convergence eliminates these artificial strengthenings by formalizing a natural requirement for many systems—that writes by one node become visible to others.

In the rest of this paper, we first define the CAC (consistency, availability, and convergence) properties (Section 2). Then we explore the CAC trade-offs in omission- and Byzantine-fault tolerant systems (Section 3 and Section 4). Finally we discuss the implications of CAC theory and results (Section 5), summarize related work (Section 6), and conclude (Section 7).

2 Consistency, Availability, and Convergence (CAC)

In this section, we define the CAC properties: consistency, availability, and convergence more carefully. We first state our assumptions about the environment and the implementation.

We assume an asynchronous model with an unreliable network: messages may be delayed for an arbitrary but finite duration, reordered, or dropped. Likewise, local clocks at different nodes may run at different speeds. We assume a *classical* memory system at each node whose state is not affected by reads as opposed to a *quantum* memory system whose state might change on reads [15]. 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 [15], when discussing the result of a read, our formalism focuses on the write operation that wrote the value that the read returns.

2.1 Consistency

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

An *execution* comprises of a set of nodes and a sequence of read and write operations at each node. We abstract the details of an execution and model read and write operations as follows:

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

For a read operation the writeList is a list of write operations that produced the values a read returns. Note that we permit a read operation to return multiple results, which, as we discuss below, provides a clean way to handle *logically concurrent updates* without making restrictive assumptions about conflict resolution.

The `startTime` and `endTime` fields indicate the real time at which an operation starts and finishes. Note that in an asynchronous system, this absolute global time is not visible to the nodes but our model includes it so that we can reason about semantics such as linearizability [21] and real time causal, which restrict legal orderings to be consistent with real-time clocks. Note that we assume serial execution at each node so that one operation at a node ends before the next one starts.

We say that a consistency semantics C_s is *stronger* than another consistency semantics C_w iff the set of executions accepted by C_s is a subset of the set of executions accepted by C_w ($E_{C_S} \subset E_{C_W}$ where E_C denotes the set of executions accepted by a consistency semantics C). We say that two consistency semantics are incomparable iff neither of them is stronger.

Causal consistency. An execution is *causally consistent* if there exists a directed acyclic graph G, called a HB (happens before) graph, containing a read/write vertex for each read/write operation in e and edges that impose a partial order $\prec_G$ (precede) on these vertices such that G satisfies the following consistency check:

CC1 *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 $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.

CC2 *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 systems implement a conflict resolution algorithm to pick a winner (e.g., highest-node-ID wins) or to merge conflicting writes (e.g., combining concurrent creation of differently-named files in the same directory) [1, 4, 18, 22, 23, 29, 41]. Instead, the definition above models the fundamental causal-ordering abstraction shared by all these approaches without attempting to impose a particular conflict resolution strategy—logically concurrent writes are returned to the reader, which can then apply any standard and application-specific conflict resolver to pick a winner or merge concurrent writes in a layer over the consistency algorithm [12, 30].

Real-time-causal consistency. An execution e is said to be *real time causally consistent (RTC)* if its HB graph satisfies the following check in addition to the checks for causal consistency:

CC3 *Time doesn't travel backward.* For any operations u, v: $u.endTime < v.startTime \Rightarrow v \not\prec_G u$.

RTC is not a new semantics. Although we are the first one to formally define RTC, it appears that most systems that claim to enforce causal consistency actually enforce the stronger RTC semantics, sometimes modified to support a system-specific conflict resolution policy [1,4,18,22,23,29,30,41]. This observation should not be surprising—it would indeed be strange for a practical implementation to order an operation that occurred later in real time before an earlier operation.

2.2 Availability

Availability, informally, refers to an implementation's ability to ensure that read and write operations complete. The availability of an implementation is defined by describing the environment conditions (network, local-clocks etc) under which all issued operations complete. An implementation is *always available* if for any workload, all reads and writes can complete regardless of which messages are lost and which nodes can communicate.

2.3 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 environment conditions (network, local-clocks etc) under which nodes can observe each other's writes.

We formalize convergence to expose the fundamental trade-off between safety (consistency) and liveness (availability and convergence). In particular, a challenge with defining *stronger consistency* as *accepts fewer executions* is that systems that fail to propagate information among nodes may technically have very strong consistency and availability. For example, a system could completely eliminate communication among nodes and have each read of an object return the latest write of the object by the reader; or a gossip-based causal consistency system could restrict its nodes to only send or receive messages if the sequence number of the node's last write is evenly divisible by 100. Each of these semantics is technically stronger than causal consistency and is still always available for reads and writes; yet, each of these semantics also feels "artificial" or less useful than causal. Convergence allows us to understand this trade-off.

A simple convergence property is *eventual consistency*. One common definition requires that if a system stops accepting writes and sufficient communication occurs, then the system reaches a state in which for any object o, a read of o would return the same value at all nodes. This formulation defines a weak convergence property; for example, it makes no promises about intervals in which some nodes are partitioned from others.

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 [36] and Dynamo [12].

We therefore define *one-way convergence*. The basic idea is that any pair of nodes s and d can converge with two steps of one-way communication: first s sends updates to d, next d sends updates to s, and then both nodes would read the same values for all objects.

The full definition is a little more complicated because it explicitly states that additional communication between s and d should not prevent convergence. First, we define an intermediate state where d has received whatever updates it needs from s. The defining property of this state is that, once d is in it, it suffices for d to send updates to s for d and s to converge to a common state.

DEFINITION 2.1. *Semi-pairwise converged. We say that machine s has* semi-pairwise converged *with machine d iff 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 define one-way convergence by saying that a system provides this property if it ensures that s and d can become semi pairwise converged through communication from s to d:

DEFINITION 2.2. *One-way convergent. A system is* one-way convergent *iff for any machines 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.*

3 CAC with omission failures

In this section, we consider environments where only omission and network failures occur.

3.1 Impossibility result

Theorem 3.1 shows that it is impossible to provide any consistency stronger than RTC while ensuring always availability and one-way convergence. This theorem holds under an asynchronous model with unreliable network even if 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 RTC and force-feed it a workload under which it must either block reads or writes (sacrificing always availability), fail to propagate updates among connected nodes (sacrificing one-way convergence), or violate RTC (showing that it is not, in fact, stronger than RTC).

THEOREM 3.1. *No consistency semantics stronger than real time causal consistency can be implemented using a one-way convergent and always available distributed storage implementation.*

Proof. (Sketch) By way of contradiction, suppose there exists a stronger semantics SC, implemented by a one-way convergent and always available distributed storage implementation, I_{SC}, that doesn't accept an execution e that is accepted by RTC consistency. We will construct a run of I_{SC} that produces the rejected execution e. The proof goes through the following stages.

Stage 1 We use the real time causal HB graph G for e to construct another graph H as follows. For every read/write vertex $v \in G$ and for each write $w \prec_G v$ such that w and v occur at different nodes, add a directed edge from w to v in H. Now, remove all non-local edges from H (edges that connect vertices at different nodes) that were not added in Stage 1.

Stage 2 We then use H to construct an execution e' by issuing operations at nodes in I_{SC} and by controlling the behavior of the network and local clocks at each node. When constructing e', we issue a few additional read operations e' (beyond those present in H) to inspect the state of the implementation I_{SC} during the execution. We use the always availability property to ensure that all operations complete.

Stage 3 Because SC is stronger than RTC, any execution e' of I_{SC} must also be RTC consistent. Let G' be the RTC HB graph for execution e'.

Stage 4 Using G', we show that reads in e and e' return the same set of writes.

Stages 1 and 3 are straightforward. In the rest of the section, we highlight the key intuition for Stages 2 and 4. The full proof is available in the Appendix B.

Stage 2 Use H to generate a series of reads/writes while controlling the network and local clocks such that I_{SC} produces an execution e' that we later show must be similar to e.

Let v be an iterator over a topological sort T of H. For each vertex v at a node p_v:

1. 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 v' were processed (see step 3 below), and (b) add an additional read r_o to object o at node p_v and wait until this read finishes.
2. Perform v's operation at node p_v. Wait until the operation completes (Because I_{SC} is always available, the operation must eventually complete).
3. 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 these messages for delivery in step (1) when the corresponding end point of this outgoing edge is processed.

Note that we have not yet specified when each operation will be performed in real time; we show in Appendix B (Lemma B.2) that a feasible real time assignment (that matches the $startTime$ and $endTime$ of operations in e) can be found because G satisfies the CC3 requirement of RTC.

Stage 4: Reads return the same set of writes in e and e' In this stage of the proof, we argue that writes that are dependent in G must remain dependent in any observer graph G' for execution e' and similarly, due to the real time constraint, concurrent writes returned on a read cannot be ordered in G'. Using these observations, we can show that e must match e'.

LEMMA 3.2. *If a write w precedes an operation u in G then w precedes u in G'. ($w \prec_G u \Rightarrow w \prec_{G'} u$)*

Proof. (Sketch) If w and u occur on the same node, then the claim follows from CC1. If not ($p_u \neq p_w$), let u' denote the earliest operation on p_u such that $w \prec_G u'$. By Stage 1, there exists an incoming edge from w to u' in H. Processing that edge in Stage 2 involved performing one-way convergence from p_w to p_u and inserting before u' artificial reads r_f to the object o at p_u, where o is the object that w was writing. It is easy to prove that $w \in r_f.wl$ and hence, by CC2, $w \prec_{G'} r_f$. Now, since by CC1 $r_f \prec_{G'} u$, it follows by transitivity that $w \prec_{G'} u$. □

LEMMA 3.3. *In forced execution e', a write w appears in the* writeList *of a read r only if w precedes r in G. ($w \in r.wl' \Rightarrow w \prec_G r$.)*

Proof. (Sketch) 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. By construction, such a path can exist only if $w \prec_G r$. □

LEMMA 3.4. *For every read $r \in e$ with writeList wl in e and wl' in e', $wl = wl'$*

Proof. (Sketch) Consider the following two cases:
Case 1: $w \in wl' \wedge w \notin wl$: From construction Stage 2, $w \in wl' \Rightarrow w \prec_G r$, 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 3.2 , $w \prec_G w' \wedge w' \prec_G r \Rightarrow w \prec_{G'} w' \wedge w' \prec_{G'} r$ so $r.wl'$ could not include w (from CC2). Contradiction.
Case 2: $w \in wl \wedge w \notin wl'$: From CC2 $w \prec_G r$, and from Lemma 3.3 $w \prec_{G'} r$. So, from CC2, for r to not return w in e', there must exist w' such that r returns w' in G' and $w \prec_{G'} w'$. 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 $w \prec_{G'} w'$.

Now, consider a different execution e'' in which w starts after w' finishes in real time (e'' is possible because w and w' are concurrent in G and hence also concurrent in H). Because the implementation does not have access to real time, it must produce identical responses in both e' and e''. In particular, the write lists wl' and wl'' returned respectively by read r in e' and e'' must be identical. However this cannot be, since by CC3 we can't have $w \prec_{G''} w'$ in any HB graph G'' for e''. Contradiction. □

Lemma 3.4 shows that e' and e match for reads that are common to both executions. Because our implementation is assumed to be classical and hence not influenced by reads, if we were to generate an execution with no artificial reads, it should still produce the same answer for all the reads that are present in e. Hence, repeating the construction in Stage (1) and (2) above but without adding the artificial reads must produce the execution e. Therefore, Theorem 3.1 holds. □

3.2 CAC implementation of RTC

We next show that RTC provides a tight bound for CAC semantics that are achievable using an always available and one-way convergent implementation. Theorem 3.5, which holds in an asynchronous model with unreliable network and omission failures, proves the desired result.

THEOREM 3.5. *Real time causal consistency (RTC) can be enforced by an always available and one-way convergent implementation.*

We defer the detailed proof of Theorem 3.5 to the Appendix F and instead provide the key ideas behind constructing the implementation and the associated proof here.

The implementation for RTC is similar to the log-exchange protocol used in Bayou [36] or PRACTI [4] where each write produces an update with a vector clock (or version vector), an object identifier, and the object value. The vector clock determines precedence of updates, and a local store at each node tracks the most recent update(s) to each object. 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. Similarly, on a read, an update is created and added to the local store and the local log at the issuing node. 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 causally precede the new update. Each node in our implementation periodically sends its log to all other nodes to ensure one-way convergence.

The implementation sketched above is always available because it does not require communication to ensure completion of reads/writes. It is one-way convergent because updates received from a sender can be applied at the receiver to attain the converged state, and nodes periodically broadcast all the updates from their local log. It is causally consistent because the vector clocks attached to each update ensure that the causally newest writes are returned on a read. Finally, the implementation is RTC consistent because the vector clock assignment never violates the real time requirement by assigning an older vector clock to a newer update.

4 CAC with Byzantine failures

The previous section considered an omission failure model. In this section, we consider a Byzantine failure model and show that fork-causal [30] and stronger consistency semantics cannot be implemented without sacrificing availability or convergence. We then introduce *bounded fork join causal* consistency that can be enforced by a one-way convergent and always available implementation.

4.1 Impossibility result

Informally, fork-causal consistency ensures that a correct node sees only causally consistent subset of the global execution even though the overall execution may not be causally consistent [30]. We include a definition of fork-causal consistency in the Appendix C for reference.

THEOREM 4.1. *Fork-causal and stronger consistency semantics are not achievable in an always available and one way convergent distributed storage implementation.*

Proof. (Sketch) Let S be a semantics at least as strong as fork-causal consistency that is implemented by an always available and one way convergent implementation I_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 two instances of a node, f_1 and f_2, that share the same initial state as f. Execute the following sequence of operations. Assume the network drops any messages not described below.

1. Issue and complete write w_a to object a at f_1 and write w_b to object b at f_2.
2. Now, let f_1 become semi-pairwise converged with p_1 by waiting for f_1 to send messages to p_1 and then delivering these messages at p_1. Similarly, let f_2 become semi-pairwise converged with p_2 by delivering messages from f_2 to p_2.
3. Issue r_{a,p_1} followed by r_{b,p_1} at p_1 and r_{b,p_2} followed by r_{a,p_2} at p_2. From the definition of one way convergence and the requirements of fork-causal consistency, r_{a,p_1} at p_1 must return w_a and r_{b,p_1} at p_1 must return $\perp$. Similarly, r_{b,p_2} at p_2 must return w_b and r_{a,p_2} must return $\perp$.

We claim that this implementation now cannot enforce one way convergence between p_1 and p_2. This is because p_1 and p_2 have observed inconsistent histories that cannot be reconciled without requiring correct nodes p_1 and p_2 to observe the concurrent writes w_1 and w_2 issued by a single node: a violation of the *serial ordering for operations seen by correct node* property enforced by fork-causal and stronger consistency semantics. In Appendix D, we show this result by arguing that no fork-causal HB graph exists for this execution. □

While the above result has been shown for highly available implementations, 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* to indicate that the two quorums in such a system are not required to overlap in any correct node. Note that any two quorums are allowed to overlap as long as the overlapping nodes can be faulty.

We then define a weaker notion of availability, called *quorum-available*. In particular, we say that an implementation is available under a quorum Q if any operation performed by a correct client c completes within a bounded amount of time if every node $p \in Q$ is correct and nodes in $Q \cup c$ can communicate with each other. Similarly, we define a weaker notion of convergence, called *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.

Now we can show the following theorem:

THEOREM 4.2. *Fork-causal and stronger consistency semantics are not achievable in a disjoint-quorum-convergent, disjoint-quorum-available, and eventually consistent distributed storage implementation.*

Proof. (Sketch) The proof of this theorem is very similar to the proof for Theorem 4.1. 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 operations at f_1 and f_2 and 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. Next we issue reads at p_1 and p_2 and force these reads to complete using quorum-availability. Now we can argue that this system can't attain eventual consistency because correct nodes p_1 and p_2 have observed inconsistent writes. □

COROLLARY 4.3. *Always available implementations are equivalent to the disjoint quorum system where each quorum contains a single correct node. Hence, always available implementations must either sacrifice eventual consistency or enforce semantics that are not equal to or stronger than fork-causal if nodes can be Byzantine.*

COROLLARY 4.4. *Fork-X implementations are equivalent to a disjoint quorum system where each quorum contains the server. Hence, fork-X implementations must either sacrifice eventual consistency or enforce semantics that are not equal to or stronger than fork-causal if nodes can be Byzantine.*

4.2 CAC implementation of BFJC

Theorem 4.1 rules out CAC implementations of fork-causal and stronger consistency semantics. We next answer the question: What consistency *can* be provided using an always available and one-way convergent implementation? We introduce a new consistency semantics called *bounded fork join causal (BFJC)* consistency that we believe is close to the strongest achievable CAC semantics in the Byzantine failure model. We then describe a CAC implementation of BFJC semantics.

THEOREM 4.5. *Bounded fork join causal (BFJC) consistency can be enforced by an always available and one-way convergent implementation.*

An execution e is *bounded fork join causally consistent (BFJC-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, and edges that impose a partial order $\prec_G$ (precede) on these vertices such that G satisfies the following consistency check:

BFJC1 *Serial ordering.* 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 $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.
The operations of a faulty node form a *directed tree*. In a tree that contains multiple leaf vertices (i.e. the tree isn't a chain), we call each path from the root to a leaf a *fork*.

BFJC2 *A read return the latest preceding concurrent writes.* Same as CC2.

BFJC3 *Time doesn't travel backward.* For operations u, v by correct nodes: $u.endTime < v.startTime \Rightarrow v \nprec_G u$.

BFJC4 *Bounded number of forks.* In a distributed implementation consisting of k faulty nodes and n correct nodes, the maximum number of forks possible are $n \cdot 2^{k-1}$. Note that it can easily be shown by construction that the bound on the number of forks imposed by BFJC definition is tight; one can indeed construct executions where any implementation is forced to create $n \cdot 2^{k-1}$ forks if it does not want to sacrifice always availability and one-way convergence.

BFJC is achievable using an always available available and one-way convergent implementation. We defer the pseudocode and proof to Appendix E and sketch the key ideas here.

BFJC can be implemented by extending the implementation for RTC in four key ways. First, as is common in protocols designed to tolerate Byzantine faults [6, 27, 30], updates are signed and include a tamper-evident encoding of the history seen by the node that is creating the update. The signature and encoded history guard against omission or reordering. Second, nodes check the received updates against their local history to ensure that nodes that issue concurrent updates (and thus create *forks*) are declared *faulty* and their updates are rejected. 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 their creator is faulty. This relaxation of update checks (compared to existing fork-X protocols [6, 8, 26, 27, 31, 35]) allows forked correct nodes to *join* these forks and ensure one-way convergence in our implementation. Conceptually, a node joins a fork by treating forked writes by a faulty node as concurrent writes by different *virtual* correct nodes [30]. 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 [5, 13, 20, 22, 38, 41]. Finally, to bound the number of forks, when a node receives updates from another node, it issues a *view* update to prove to other nodes that none of the updates it accepted were issued by nodes known to be faulty.

5 Discussion

Why CAC and not CAP? 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.

Additionally, CAP does not explicitly consider *convergence* because linearizability and sequential consistency embed a convergence requirement. When we examine weaker semantics like causal consistency, we find that we must explicitly consider convergence.

Open questions. There are strong consistency semantics that are incomparable with linearizability but that seem “unnatural” [15] (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?

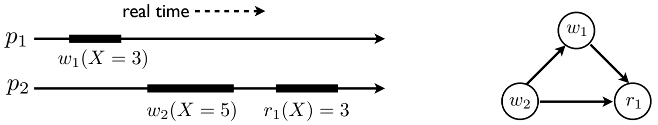
As CAP indicates, different system models yield different trade-offs; a *CAC-M* framework would generalize CAC by exposing the model as a parameter. CAP and this paper showed consistency-availability trade-offs for an unreliable network with Byzantine and omission failures. What other bounds on consistency 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 [6] 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 one-way convergence? The weaker *two-way 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, RTC is not the strongest always-available consistency semantics under two-way 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 [4, 12, 36]. 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 [13, 36, 42].

Open questions: How do the CAC trade-offs vary as we weaken convergence requirements? Although we know that we can strengthen consistency if we only require two-way 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 isn’t causal the strongest? Here we illustrate the HB graph for an execution that is causal but not RTC. 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.



(a) Causal execution (b) HB graph

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

Why no tight bound for Byzantine? We would have liked to show that BFJC is the strongest available and convergent semantics in the Byzantine model. Unfortunately, this conjecture is false—an implementation can enforce a slightly stronger, albeit unnatural, *BFJC’* semantics that disallows certain BFJC executions. For example, consider an execution e consisting of two logically concurrent writes w_1 and w_2 issued by a faulty node f to an object o. An implementation for BFJC’ can suppress the actual concurrency and pretend that one of the writes (say w_1) precedes the other (say w_2). Such a *BFJC’* implementation admits all executions admitted by BFJC except for execution e (that BFJC admits) and thus enforces stronger semantics.

The CC3 constraint of RTC helps us rule out such strengthenings in the omission-failure model. Unfortunately, operations performed by a Byzantine faulty node do not have a well-defined start and end time, so that approach does not work here.

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

6 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 [10, 17, 40]. Similarly, there is a fundamental tension between strong consistency and performance [28]. Yu and Vahdat's TACT framework provides ways to manage the trade-offs between availability, consistency, and staleness [43]. Frigo defined the weakest "reasonable" memory model by imposing specific constraints, such as *constructability*, *nondeterminism confinement*, and *classicality* [16]. Frigo also noted the problem of defining "trivial yet strong" semantics that we address by using convergence properties [16].

To ensure availability, many systems have implemented causal consistency and weaker semantics with various conflict resolution policies [1, 4, 11, 12, 18, 36, 37, 39]. To avoid picking a winner among different conflict resolution policies, we follow some previous systems [4, 12, 30, 37] 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. RTC). Lloyd et al. [29] explicitly note that their system's *causal and per-object sequential* 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 [3, 34] by, for example, associating a Lamport clock [24] with each write and then imposing a highest-accept-stamp-wins conflict resolution policy.

Traditionally, Byzantine faults have been addressed using quorums [32, 33], sometimes within state machine replication systems [2, 9, 19]. For sufficiently large quorums, traditional semantics such as regular, safe, or atomic registers [25] or linearizability [21] can be enforced, while weaker fork-X consistency variations can be enforced on smaller quorums or even individual machines [6, 8, 26, 27, 31, 35]. Whereas traditional semantics are unavailable when quorums are unresponsive, many Fork-X semantics allow faulty nodes to introduce *permanent* partitions among correct nodes [30]. In this paper, we show that this problem is fundamental to these semantics.

Cachin et al. [6, 7] expose the trade-off between consistency and availability in fork-X semantics by showing that neither *fork-sequential* consistency [35], nor *fork-* linearizable* consistency [27] can be enforced by a wait-free implementation using a Byzantine faulty server.

The BFJC consistency in this paper strengthens Depot's fork join causal consistency [30] to ensure that only a bounded number of forks are admitted. The BFJC implementation uses *views* to bound the number of forks and enforce BFJC consistency. In contrast, Depot's eviction protocol is more efficient and attains similar bound on forks but requires two way communication.

Eventual propagation [14] requires all the nodes to observe all updates but, unlike convergence, does not demand different nodes to order updates or converge to a common state.

7 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 *real time causal* consistency, a strengthening of causal consistency that respects the real time order of operations, is the strongest semantics achievable while retaining strong liveness guarantees. Similarly, we show that in the Byzantine failure model, fork-causal [30] or stronger consistency 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 introduce *bounded fork join causal* consistency and show that it can be enforced despite network failures and Byzantine nodes.

References

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

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

[3] 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.

[4] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006.

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

[6] C. Cachin, I. Keidar, and A. Shraer. Fail-Aware Untrusted Storage. In *DSN*, 2009.

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

[8] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *PODC*, 2007.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4), 2002.

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

[11] 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 *VLDB*, 2008.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[13] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed filesystem for challenged networks in developing regions. In *FAST*, 2008.

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

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

[16] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *SPAA*, 1998.

[17] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*, 2002.

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

[19] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. In *Eurosys*, 2010.

[20] 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.

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

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

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

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

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

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[27] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

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

[29] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Stronger consistency for wide-area storage. NSDI 2011 Poster Session, Mar. 2011. http://www.cs.princeton.edu/˜wlloyd/papers/widekv-poster-nsdi11.pdf.

[30] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI 2010*, Oct. 2010.

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

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

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

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

[35] A. Oprea and M. Reiter. On consistency of encrypted files. In *DISC*, 2006.

[36] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*, 1997.

[37] 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 *NSDI*, 2009.

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

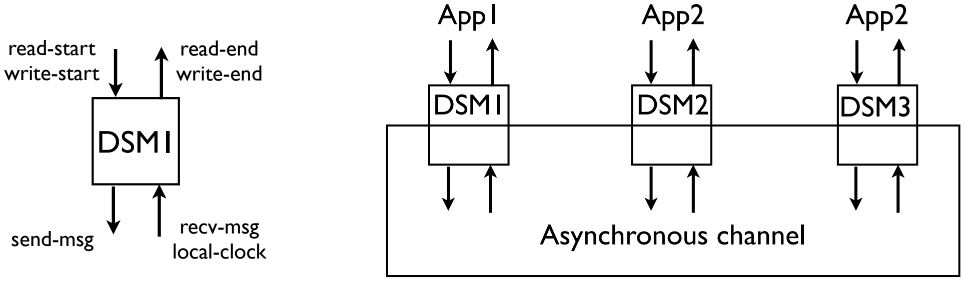
[39] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for distributed workstation environments. *IEEE Transactions on Computers*, 39(4):447–459, Apr. 1990.

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

[41] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

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

[43] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, pages 29–42, 2001.



(c) A Storage implementation

(d) A distributed storage implementation

Figure 1: A storage implementation (a) and 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 various implementations.

A Revisiting availability and convergence

In this section, we first introduce our framework and then define the availability, and convergence properties in terms of our framework. We note that this framework is a slight extension of the framework that we introduced in the main paper. This extension is useful to provide precise proofs for the results stated in the main paper. The definitions of consistency remains unchanged and therefore we don't repeat those definitions.

A.1 Terminology

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 written, `uid` denote application assigned unique identifiers assigned to each read and write operation, `nodeId` denotes the unique identifier of a storage implementation, `m` denotes content of a message, `wl` denotes a writeList which is a set of tuples of the form `(uid, d)` indicating that multiple values can be returned on a read [12, 30].

We assume that the automaton implements a *classical* memory system whose state is not changed by reads reads (as opposed to a *quantum* memory system in which some reads may change the behavior of other reads) [15]. Note that we do not preclude implementations in which a local read operation triggers communication (e.g., a local read causes a node to fetch updates from nearby reachable node) including the reception of messages that changes the system state. In these systems, it is not the *read* that changes the state of the reader, it is the reception of messages.

A *distributed storage implementation* consists of a collection of storage implementations that communicate through an asynchronous channel. Figure 1 shows a storage implementation and a distributed storage implementation connected through an asynchronous channel.

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. The channel controls the local clock at each storage implementation by issuing `local-clock` events—the implementation doesn't have access to any other clock. The channel also controls the network by issuing `recv-msg` events and receiving `send-msg` events.

Environment We assume an asynchronous model with an unreliable network: messages may be delayed for an arbitrary but finite duration, reordered, or dropped in the network. Likewise, local clocks at different storage implementations may run at different speeds.

We describe these environmental conditions 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 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.

An environment graph is a directed acyclic graph with read-start, write-start, local-clock, send-msg-stub, and recv-msg-stub vertices. It contains edges connecting successive events at each storage implementation and edges connecting 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 has been provided as input to it.

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 local-clock 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 time t_1 for storage implementation p_2 at time t_2, then the corresponding environment graph will contain a send-msg-stub at p_1 at local clock t_1 connected using a directed edge to a recv-msg-stub at p_2 at local clock t_2.

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.

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 details of a run that are not needed for defining consistency while retaining 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. nodeId corresponds to the identifier of the storage implementation at which the event is issued.

The real-time (startTime and endTime) of an operation reflects 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 layer and vice-versa can take some finite 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 occur 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 [21]. We require that each operation takes finite and positive time to complete. Note that the real time(s) associated with an event are not visible to the implementation that only has access to an unsynchronized local clock.

A.2 Availability

An implementation I is *available* under a environment graph EG if EG produces a *available* run on I. In an available run, each read-start input has a corresponding read-end output and each write-start input has a corresponding write-end output.

Now, we can compare the availability of two implementations. An implementation I is *more available* than implementation I' if the set of environment graphs S_G under which I is available is a superset of the set of environment graphs $S_{G'}$ under which I' is available.

An implementation I is *always available* iff it is available under all environment graphs that contain infinite local clock events for each machine. An implementation in which one storage implementation must communicate with another before processing a request cannot be always available because a read or write request can not complete in an environment where two storage implementations are partitioned.

A.3 Convergence

Convergence intuitively refers to the ability of a system to ensure that writes issued by one storage implementation are observed by other storage implementations. Like consistency and availability, different systems can offer different convergence properties.

We say that a run R of implementation I has *globally converged* iff for any extension R' of the run R with no `write-start` events in R'/R, reads in R'/R of identical objects that return responses, return identical responses.

Informally, an implementation is eventually consistent if sufficient communication can cause it to become globally converged. Formally, an implementation is said to be *eventually consistent* iff for each run R, there exists an extension R' of the run R that has globally converged such that R'/R comprises only of `local-clock` events, `send-msg` events, and `recv-msg` events that correspond to a subset of the `send-msg` events in R'/R.

The above property does not require that disconnected machines should be able to converge and therefore an implementation may (for example) provide eventual consistency by designating a special master machine responsible for resolving conflicts or through which all updates flow. However, we are interested in highly available systems that can tolerate failures or partitions of arbitrary subsets of machines while still allowing the remaining machines to provide useful liveness and safety properties. Therefore, we next define a strengthening of eventual consistency that defines a similar convergence property among arbitrary subsets of connected machines.

We say that an available run R of implementation I has *pairwise converged* for machines s and d iff for any extension R' of the run R that satisfies the following constraint, reads by s and d in R'/R of identical objects that return responses, return identical response: R'/R comprises only of `local-clock` events at s and d, `send-msg` events at s and d, and `recv-msg` events at s and d that correspond to a subset of the `send-msg` events in R'/R.

Informally, an implementation is pairwise convergent if sufficient communication between a pair of nodes can cause that pair of nodes to converge. Formally, an implementation is said to be *pairwise convergent* iff for each run R and for any nodes s and d, there exists an extension R' of the run R that has pairwise converged for s and d and R'/R comprises only of `local-clock` events at s and d, `send-msg` events at s and d, and `recv-msg` events at s and d that correspond to a subset of `send-msg` events R'/R.

Most practical systems rely on one-way transfer of data where data is transferred from one machine to another. We next define a convergence property useful for such scenarios.

The basic idea of one-way convergence is that any pair of nodes s and d can converge with two steps of 1-way communication: first s sends updates to d, next d sends updates to s, and then both nodes read the same values for all objects. The full definition is a little more complicated because it explicitly states that additional communication between s and d should not prevent convergence.

First, we define an intermediate state where d has converged on the appropriate state by receiving whatever it needs from s. The key property of this state is that now d can send to s and both nodes would converge to the common state:

DEFINITION A.1. *Semi-pairwise converged. We say that a run R of an implementation I has* semi pairwise converged *for nodes s and d iff for all extensions R' of run R such that R'/R contains no* `write-start` *events and all* `send-msg` *and* `recv-msg` *events in R'/R occur at node s and d and* `recv-msg` *events in R'/R correspond to some subset of* `send-msg` *events in R'/R, the following condition holds: there exists an extension R'' of run R' that has pairwise converged for s and d such that R''/R' comprises of only* `local-clock` *events at s and d,* `send-msg` *events at d, and* `recv-msg` *events at s that correspond to a subset of* `send-msg` *events in R''/R.*

Now we can define one-way convergence by saying that an implementation provides this property if it ensures that s and d can become semi pairwise converged through communication from s to d:

DEFINITION A.2. *One way convergent. An implementation is said to be* one way convergent *iff for each run R and for any nodes s and d, there exists an extension R' of R such that R'/R comprises only of* `local-clock` *and* `send-msg` *events at s, such that all extensions R'' of R' that satisfy the following constraints, are semi pairwise converged: (1) R''/R' comprises only of* `local-clock` *events at d, and (2) R''/R' contains a corresponding* `recv-msg` *event at d for every* `send-msg` *event to d at s in R'/R.*

B Nothing stronger than RTC is enforceable

THEOREM B.1. *No consistency semantics stronger than real time causal consistency can be implemented using a one-way convergent and always available distributed storage implementation.*

Proof. By way of contradiction, suppose there exists a stronger semantics SC, implemented by a one-way convergent and always available distributed storage implementation, I_{SC}, that doesn't accept an execution e that is accepted by RTC consistency. We will show that a run of I_{SC} exists that produces the rejected execution e. To do so, we will construct a run augmented with a few additional reads added to inspect the state of the implementation. We will then argue that this execution e' matches e in reads and writes that are present in both the executions. Hence, if we were to remove the additional reads, the implementation must produce the execution e as we have assumed that the state of the implementation doesn't change on reads (classical implementation assumption). We will construct a run of I_{SC} that produces the rejected execution e. The proof goes through the following stages.

Stage 1 We use the real time causal HB graph G for e to construct another graph H as follows. For every read/write vertex $v \in G$ and for each write $w \prec_G v$ such that w and v occur at different nodes, add a directed edge from w to v in H. Now, remove all non-local edges from H (edges that connect vertices at different nodes) that were not added in Stage 1.

Stage 2 We then use H to construct an execution e' by issuing operations at nodes in I_{SC} and by controlling the behavior of the network and local clocks at each node. When constructing e', we issue a few additional read operations e' (beyond those present in H) to inspect the state of the implementation I_{SC} during the execution. We use the always availability property to ensure that all operations complete.

Stage 3 Because SC is stronger than RTC, any execution e' of I_{SC} must also be RTC consistent. Let G' be the RTC HB graph for execution e'.

Stage 4 Using G', we show that reads in e and e' return the same set of writes.

Stages 1 and 3 are straightforward. In the rest of the section, we describe Stages 2 and 4 in detail.

Stage 2 Use H to generate a series of reads/writes while controlling the network and local clocks such that I_{SC} produces an execution e' that we later show must be similar to e.

Let v be an iterator over a topological sort T of H. For each vertex v at a node p_v:

1. 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 v' were processed (see step 3 below), and (b) add an additional read r_o to object o at node p_v and wait until this read finishes.
2. Perform v's operation at node p_v. Wait until the operation completes (Because I_{SC} is always available, the operation must eventually complete).

3. 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 these messages for delivery in step (1) when the corresponding end point of this outgoing edge is processed.

Stage 4: Executions e and e' match. In this stage of the proof we will argue that executions e and e' match. We first show that it is possible to control the local clocks at each node such that the $startTime$ and $endTime$ of read and write events in e and e' can be made identical. Then we will show that reads that are present e must return identical responses in both e and e'.

Feasible time assignment We next show that a feasible real time assignment (that matches the $startTime$ and $endTime$ of operations in e) can be found because G satisfies the CC3 requirement of RTC.

Let ψ be the run corresponding to the above execution. Recall that ψ will contain `read-start`, `read-end`, `write-start`, `write-end`, `send-msg`, `recv-msg`, and `local-clock` events. Now, we will assign a real time to each of these events such that each read and write event starts and finishes ($storeStartTime$ and $storeEndTime$) at some time between the specified start and end time of read/write operations in the execution e (i.e. $startTime$ and $endTime$)

Recall that we assumed that the real time clock has infinite precision and therefore, there are infinitely many timestamps between any two real timestamps. Consider a topological sort T of G such that when multiple choices are possible, the vertex with smallest $startTime$ is chosen. This topological sort has no *inconsistent assignments* because G doesn't contain any inconsistent assignments. We now use the following algorithm to assign the time stamps (s, e) to every event i:

1. set $t = \text{Min}_{\forall j}((j.endTime - j.startTime)/3 \cdot N, \forall_{i:i \prec_G j}(j.endTime - i.startTime)/3 \cdot N$ where N is the total number of vertices in G.

2. Let i be the reverse iterator on T

3. set $i.e = \text{Min}(i.endTime - t, \forall_{j \in c_i}(j.e - 3 \cdot t))$.

4. set $i.s = i.e - t$.

5. if $i.startTime < i.s < i.e < i.endTime$ continue else stop.

LEMMA B.2. *The above algorithm produces a legal time stamp assignment ($\forall i, i.startTime < i.s < i.e < i.endTime$).*

Proof. (Sketch) Suppose that no legal assignment exists and the check on step 5 fails at $i = I$. Now, from step 3 and 4, $\forall i \exists j : i.s = j.endTime - k \cdot t$ where $i \prec_G j$ and k is a finite integer such that $k < 3 \cdot N$ (this can be shown by induction). However, this value of k combined with the value of t implies that the check at step (5) can't be violated. □

We then assign the (s, e) values to the corresponding events in the run—the s value is assigned to the corresponding start event (`read-start`/`write-start`) and e value is assigned to the corresponding end event. The intermediate `local-clock` and `send-msg` and `recv-msg` events can then be assigned suitable time stamp based on a topologic sort of the run graph ψ.

Reads return the same set of writes in e and e' In this stage of the proof, we argue that writes that are dependent in G must remain dependent in any observer graph G' for execution e' and similarly, due to the real time constraint, concurrent writes returned on a read cannot be ordered in G'. Using these observations, we can show that e must match e'.

LEMMA B.3. *If a write w precedes an operation u in G then w precedes u in G'. ($w \prec_G u \Rightarrow w \prec_{G'} u$)*

Proof. If w and u occur on the same node, then the claim follows from CC1. If not, let p_w and p_u denote the distinct nodes where w and u respectively occurred, and let u' denote the earliest operation on p_u such that $w \prec_G u'$. By Stage 1, there exists an incoming edge from w to u' in H. Processing that edge in Stage 2 involved performing one-way convergence from p_w to p_u and inserting before u' artificial reads r_f to the object o at p_u, where o is the object that w was writing. It is easy to prove (see below) that $w \in r_f.wl$ and hence, by CC2, $w \prec_{G'} r_f$. Now, since by CC1 $r_f \prec_{G'} u$, it follows by transitivity that $w \prec_{G'} u$.

Next we show that $w \in r_f.wl$. Suppose $w \notin wl_{r_f}$. Consider a different environment ψ' that consists of only events that have a path to the start of u in ψ. Now extend psi' as follows: First, add local clock events to ensure termination of u using the availability requirement on I_{SC}. Second, add one way convergent communication pattern from u's machine to w's machine. Third, add a read $r_{f'}$ at $p_{w'}$ to the object o. Finally, add local clock events to ensure termination of $r_{f'}$. From the one-way convergence requirement $wl_{r_{f'}} = wl_{r_{f'}}$. Furthermore, $wl_{r_{f'}}$ must contain w as argued below:

1. Note that in any RTC HB graph J for ψ', $w \prec_J r_{f'}$ from the program order requirement.

2. So, $w \notin wl_{r_{f'}} \Rightarrow \exists w' : w \prec_J w' \wedge w' \prec_J r_{f'}$. However, from construction of psi' there doesn't exists a w' in ψ' such that $w \prec_{\psi'} w'$. So $w||_{\psi'}w'$. However, if w and w' are concurrent in ψ', we can construct psi'' in which w and w' have time stamps such that w starts after w' finishes. Because these time stamps are not visible to the implementation, I_{SC} should still produce the same response on $r_{f'}$ which is impossible without violating CC3 requirement of RTC consistency. Hence, by contradiction, we must have $w \in wl_{r_{f'}}$.

□

LEMMA B.4. *In forced execution e', a write w appears in the* writeList *of a read r only if w precedes r in G. ($w \in r.wl' \Rightarrow w \prec_G r$.)*

Proof. (Sketch) 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. By construction, such a path can exist only if $w \prec_G r$. □

LEMMA B.5. *For every read $r \in e$ with writeList wl in e and wl' in e', $wl = wl'$*

Proof. (Sketch) Consider the following two cases:
Case 1: $w \in wl' \wedge w \notin wl$**:** From construction Stage 2, $w \in wl' \Rightarrow w \prec_G r$, 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 B.3 , $w \prec_G w' \wedge w' \prec_G r \Rightarrow w \prec_{G'} w' \wedge w' \prec_{G'} r$ so $r.wl'$ could not include w (from CC2). Contradiction.
Case 2: $w \in wl \wedge w \notin wl'$**:** From CC2 $w \prec_G r$, and from Lemma B.4 $w \prec_{G'} r$. So, from CC2, for r to not return w in e', there must exist w' such that r returns w' in G' and $w \prec_{G'} w'$. 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 $w \prec_{G'} w'$.

Now, consider a different execution e'' in which w starts after w' finishes in real time (e'' is possible because w and w' are concurrent in G and hence also concurrent in H). Because the implementation does not have access to real time, it must produce identical responses in both e' and e''. In particular, the write lists wl' and wl'' returned respectively by read r in e' and e'' must be identical. However this cannot be, since by CC3 we can't have $w \prec_{G''} w'$ in any HB graph G'' for e''. Contradiction. □

Lemma B.5 shows that e' and e match for reads that are common to both executions. Because our implementation is assumed to be classical and hence not influenced by reads, if we were to generate an execution with no artificial reads, it should still produce the same answer for all the reads that are present in e. Hence, repeating the construction in Stage (1) and (2) above but without adding the artificial reads must produce the execution e. Therefore, Theorem B.1 holds. □

C Fork-causal consistency

DEFINITION C.1. *An execution e is said to be* fork-causally consistent *if there exists a directed acyclic graph G, called a HB (happens before) graph, containing a read/write vertex corresponding to each read/write operation in e, and edges connecting these vertices such that G satisfies the following consistency check.*

FC1 *Serial ordering at each correct node.* The ordering of operations by any node is reflected in G. 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:

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

FC3 *Serial ordering for operations seen by correct node.* For any operation o by a correct node and for all operations u_1 and u_2 by a node p, $u_1 \prec_G o \wedge u_2 \prec_G o \Rightarrow u_1 \prec_G u_2 \vee u_2 \prec_G u_1$.

D Byzantine impossibility result

THEOREM D.1. *Fork-causal and stronger consistency semantics are not achievable in an always available and one way convergent distributed storage implementation.*

Proof. Let S be a semantics at least as strong as fork-causal consistency that is implemented by an always available and one way convergent implementation I_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 two instances of a node, f_1 and f_2, that share the same initial state as f. Execute the following sequence of operations. Assume the network drops any messages not described below.

1. Issue and complete write w_a to object a at f_1 and write w_b to object b at f_2.
2. Now, let f_1 become semi-pairwise converged with p_1 by waiting for f_1 to send messages to p_1 and then delivering these messages at p_1. Similarly, let f_2 become semi-pairwise converged with p_2 by delivering messages from f_2 to p_2.
3. Issue r_{a,p_1} followed by r_{b,p_1} at p_1 and r_{b,p_2} followed by r_{a,p_2} at p_2. From the definition of one way convergence and the requirements of fork-causal consistency, r_{a,p_1} at p_1 must return w_a and r_{b,p_1} at p_1 must return $\perp$. Similarly, r_{b,p_2} at p_2 must return w_b and r_{a,p_2} must return $\perp$.

We claim that this implementation now cannot enforce one way convergence between p_1 and p_2. This is because p_1 and p_2 have observed inconsistent histories that cannot be reconciled without requiring correct nodes p_1 and p_2 to observe the concurrent writes w_1 and w_2 issued by a single node: a violation of the *serial ordering for operations seen by correct node* property enforced by fork-causal and stronger consistency semantics.

More precisely, suppose the implementation I_S can attain one way convergence. Let p_1 send messages necessary to p_2 to ensure one-way convergence and vice-versa. Deliver these messages and 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. Now we consider the admissible HB graphs for this execution. We first note that in any fork-causal HB graphs for this execution w_a and w_b must be shown as concurrent. Graphs that order them as $w_1 \prec w_2$ or $w_2 \prec w_1$ won't be admissible based on the reads issued by correct nodes p_1 and p_2.
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 the reads returns $\perp$, we can show that no fork-causal HB graph can satisfy the *reads return the most recent concurrent write property* at correct node p_1 because the 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. Similarly, if the reads return w_a, then we can show that 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 faulty node f. □

E BFJC is enforceable using a CAC implementation

Consider an execution of the pseudocode described in Table 1. We show that this implementation is CAC: VFJC-consistent, available, and one-way convergent.

DEFINITION E.1. *A write w (or a view 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 vs when the update is added to the actual log). A read operation r is* accepted *by a correct node p if r is issued by p. The accept time for an operation v accepted by a correct node p is denoted by $ts_{p,v}$ and 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 *w accepted by a correct node p, $ts_{p,w}$ indicates the real time when the monitor lock was acquired in step 27 (put method call) at that correct node.*
- *For a* view *v accepted by a correct node p, $ts_{p,v}$ indicates the real time at which* createView *is invoked at p.*
- *For all other operation node pairs, $ts_{p,v}$ is undefined.*

DEFINITION E.2. *For an update u, define* history *of u (denoted by H_u) as a set of updates such that $\forall v \in H_u \exists w_1, w_2, ..., w_k : H(u) \in w_1.prevUpdates, H(w_1) \in w_2.prevUpdates, ..., H(w_k) \in v.prevUpdates$. $H(u)$ denotes the cryptographic hash of update u.*

DEFINITION E.3. *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 write/view operation issued by a correct node p, then $PRED_v$ includes all reads r accepted by p such that $ts_{p,r} < ts_{p,v}$.*
3. *If v is a read operation issued by a correct node p, then $PRED_v$ includes all operations u accepted by p such that $ts_{p,u} < ts_{p,v}$.*
4. *if u is any other operation not covered by any of the earlier cases, then $PRED_u$ is the empty set.*

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

DEFINITION E.4. *Define $u \prec v$ if*

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

DEFINITION E.5. *Vertices v_1 and v_2 in $\prec$ are said to be connected through an edge (denoted by $v_1 \rightarrow v_2$) iff $v_1 \prec v_2 \wedge \nexists u : v_1 \prec u \prec v_2$.*

Table 1: Listing for BFJC implementation.

```
Messages
  Update := {NodeID $nodeID$, OID $oid$, Value $value$,
             Set$\langle Hash \rangle$ $prevUpdate\}_{\sigma_{nodeID}}$
  // prevUpdate is a set containing hashes
  // of updates that were superseded by u

State (at each node p)
  State := {Set$\langle Hash \rangle lastWrites$, Update $lastLocalWrite$,
  $HashMap\{oid, \langle Hash \rangle\}store$, $Map\{Hash, Update\}log$,
  NodeID $myNodeID$, Set$\langle NodeID \rangle$ nodes, $(K_p, K_u)\}$

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

  MyState := local state

Methods
synchronized function get(OID $oid$):
  return $MyState.store\{oid\}$

synchronized function put(OID $oid$, Value $value$):
  putInt($MyState$, $oid$, $value$)
  createView($MyState$)

function putInt(State $s$, OID $oid$, Value $value$):
  create a new signed update $u$ such that
  $u.prevUpdates := s.LastWrites \cup s.LastLocalWrite$
  $u.oid := oid$
  $u.value := value$
  $u.nodeID := myNodeID$
  return intApply($s$, $u$)

function send():
  while(true)
    synchronized
      Let $T$ be a list of all updates sorted
      in an order in which they were added
      to $MyState.Log$
    foreach $p \in MyState.nodes$ send $T$ to $p$
    sleep 30

synchronized function pktApply(Set$\langle Update \rangle pkt$):
  if($pkt$ contains only view updates)
     return false
  State testState = State.copy()
  if(intPktApply($testState, pkt$)
    intPktApply($MyState, pkt$)
  else return false
  return true

function intPktApply(State $s$, Set$\langle Update \rangle pkt$):
  status = true
  foreach $w \in pkt$
    status = status $\wedge$ intApply($s, w$)
  status = status $\wedge$ lastWriteViews($s$)
  if (status $\wedge$ createView($s$))
     return true
  else return false

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

function intApply(State $s$, Update $u$):
  if(s.log.containsKey(Hash($u$)))
     return true
  else if(verify($s, u$))
    // update state
    foreach $l \in$ s.lastWrites
      if(prec($s, l, u$))
        s.lastWrites := s.lastWrites $- \{l\}$
    if($u.nodeID = s.myNodeID$)
      $s.LastLocalWrite := u$
    else
      s.lastWrites := s.lastWrites $\cup \{u\}$
    s.log$\{Hash(w)\}$ = $u$
    foreach $l \in s.Store[u.oid]$
      if(prec($s, l, u$))
        $s.Store\{u.oid\} := s.Store\{u.oid\} - \{l\}$
    $s.Store\{u.oid\}$ := s.Store$\{u.oid\} \cup \{u\}$
  else
    return true

function verify(State $s$, Update $u$):
  return signed($u.nodeID, u$) $\wedge$
         historyLocal($s, u$) $\wedge$
         noFaultyChild($s, u$)

// returns if the history of u is present
// in local history of state s
function historyLocal(State $s$, Update $u$):
   foreach $v \in u.prevUpdates$
      if($s.log.containsKey(Hash(v))$)
         return false
   return true

// returns if any of the immediate child
// of w is faulty
function noFaultyChild(State $s$, Update $u$):
   foreach $c \in u.prevUpdates$
     if($c \in getFaulty(s, u)$)
        return true
   return false

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

// returns true if u occurs in history of v
function prec(State $s$, Update $u$, Update $v$):
  if ($v = \perp$) return false
  foreach $H_w \in v.prevUpdates$
   w = s.log$\{H_w\}$
   if ($u = w$)
       return true
   else if (prec($s, u, w$))
      return true
  return false

// creates a view
function createView(State $s$):
  return putInt($s$, VIEW, VIEW)
```

DEFINITION E.6. *An edge connecting vertices v_1 and v_2 in $\prec$ is said to be* non-local *if v_1 and v_2 correspond to operations occurring at different nodes.*

LEMMA E.7. *Consider a chain $r \prec_{PRED} u_1 \prec_{PRED} ... \prec_{PRED} u_k$. $p_r \neq p_{u_k} \Rightarrow \exists l : 1 \leq l \leq k \wedge p_{u_l} = p_r \wedge u_l$ is a write/view operation.*

Proof. (Sketch) Let l be the largest value such that $1 \leq l < k \wedge \forall n \leq l, u_n.nodeID = r.nodeID$. There must exist one such l because from Definition E.3[2,3], reads only appear in $PRED$ set of local operations. By the same logic, we claim that u_l must be a write/view operation. □

LEMMA E.8. *$\prec$ is acyclic.*

Proof. (Sketch) Consider a cycle $C : u_1 \prec_{PRED} u_2 \prec_{PRED} ... \prec_{PRED} u_k \prec_{PRED} u_1$. Suppose 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 that would have required p's log to contain u_k contradicting the assumption that u_1 was accepted first.

Now consider a cycle with at least one read u_i. u_i must be issued by a correct node since $PRED$ for reads by faulty nodes is set to empty. Furthermore, from definition of $PRED$ for reads, node that issued u_i must have accepted u_{i-1}. Break down this chain into maximum number of sub-chains $C_1, C_2, ..., C_l$ such that

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

Suppose that the above procedure produces l chains. Because the chain C contains a read, it must contain at least two operations that are issued by a correct node (from Lemma E.7, a read must be followed by a write by the same node) and hence the above breaking is possible and should produce at least two components ($l \geq 2$).

Now, from definition of $PRED$ (Definition E.3[2][3]), each sub-chain of length greater than two must contain only write/view operations at all positions except the last position. Let s_k, e_k denote the starting and end vertices of a sub-chain C_k and p_k denote the correct node at which e_k was issued. Now, we make the following two claims:

Claim 1 First, for each sub-chain C_k, we show that s_k and e_k are accepted by the node p_k and $ts_{p_k,s_k} < ts_{p_k,e_k}$.

The proof for this claim is as follows. There are four types of chains: $r \rightarrow r$, $r \rightarrow w$, $w \rightarrow r$, and $w \rightarrow w \rightarrow ... \rightarrow w$ (from the construction steps involved in breaking down a chain). In the first three cases, the desired result follows from the definition of $PRED$ and from the fact that $PRED$ is only defined to be non-empty for reads by correct node. So, we concentrate on the third case. Now, in a sub-chain C_k consisting of writes $w_1 \prec_{PRED} w_2 \prec_{PRED} ... \prec_{PRED} w_i$, we have $w_1 \in w_2.prevUpdates, w_2 \in w_3.prevUpdates, ..., w_{i-1} \in w_i.prevUpdates$ and hence, w_1 must have been accepted by p_k prior to accepting w_2 for the *localHistory* check to pass and similarly w_2 must have been accepted before w_3 and so on. Therefore, we must have $ts_{p_k,w_1} < ts_{p_k,w_2} < ... < ts_{p_k,w_i}$. Hence, we get our desired result. Note that the $w \rightarrow w \rightarrow ... \rightarrow r$ chain with length ¿ 2 isn't possible because then the penultimate w must have been issued at a correct node violating the requirement (2) in the breaking down steps above.

Claim 2 Next, we note that for consecutive sub-chains, C_k, C_{k+1}, $ts_{p_k,e_k} \leq ts_{p_{k+1},s_{k+1}}$.

This claim can be proved as follows. Recall that $e_k = s_{k+1}$—requirement 3 in the construction above). If $p_k = p_{k+1}$ then the above claim follows trivially as $ts_{p_k,e_k} = ts_{p_{k+1},s_{k+1}}$. If $p_k \neq p_{k+1}$ then e_k can't be a read operation from Lemma E.7. Otherwise, we must have $ts_{p_k,e_k} < ts_{p_{k+1},e_k}$ as it will take some finite time for the write to propagate from the issuing node to the receiving node.

Combining claims (1) and (2), we get that $ts_{p_1,s_1} < ts_{p_1,e_1} \leq ts_{p_2,s_2} < ts_{p_2,e_2} \leq ... \leq ts_{p_l,s_l} < ts_{p_l,e_l} \leq ts_{p_1,s_1}$. Since $l \geq 2$, the above real time assignment is not feasible and hence, by contradiction, no cycles with at least one read can exist. □

Lemma E.9. *Let w_1 be a write/view operation accepted by a correct node p, and w_2 by a write/view operation issued by p, then $ts_{p,w_1} < ts_{p,w_2} \Rightarrow w_1 \in H_{w_2}$.*

Proof. (Sketch) If $w_1 \in w_2.prevUpdate$, then the desired result follows.

Consider the case when $w_1 \notin w_2.prevUpdate$. 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 $w_1 \in v_1.prevUpdate, v_1 \in v_2.prevUpdate, ..., v_l \in w_2.prevUpdate$. Hence, $w_1 \in H_{w_2}$. □

Lemma E.10. *If $v.nodeID = v'.nodeID$ and $v.nodeID$ is correct, then $v.startTime < v'.startTime \Leftrightarrow v \prec v'$.*

Proof. (Sketch) *"if".* $v.startTime < v'.startTime \Rightarrow v \prec v'$. We assume that no outstanding operations are issued at each node. If either v or v' is a read, then the desired result follows from the Definition E.3 (clause 2 and 3). Otherwise if both v and v' are non-reads then $v.startTime < v'.startTime \Rightarrow v \in H_{v'} \Rightarrow v \prec v'$ (using Lemma E.9 and Definition E.3[1]).
"only if". $v.startTime < v'.startTime \Leftarrow v \prec v'$. We know that v and v' are performed by the same correct node and we know that operations performed by a node have non-overlapping times. Therefore we must have either $v.startTime < v'.startTime$ or $v'.startTime < v.startTime$. If $v.startTime < v'.startTime$ then we have achieved the desired result. If not, then by the "if" part shown above, we must have $v' \prec_{PRED} v$. Combining this with $v \prec_{PRED} v'$, we get that the observer graph must have a cycle which will violate Lemma E.8. Therefore, by contradiction, this scenario is not possible. □

Lemma E.11. *If r and w denote reads and writes/views respectively then $v \prec_{PRED} r \prec w \wedge p_r = p_w \Rightarrow v \prec_{PRED} w \vee \exists$ writes/views $w_1, w_2, ..., w_l : v \prec_{PRED} w_1 \prec_{PRED} ... \prec_{PRED} w_l \prec_{PRED} w$.*

Proof. (Sketch) Let $p = p_r = p_w$. From Lemma E.10, we have $ts_{p,r} < ts_{p,w}$. If v is a read, then from Definition E.3[2][3], we get $v \prec_{PRED} w$ and hence the desired result follows.

Suppose instead, that v is a write/view operation. In this case the desired result follows from Lemma E.9. □

Lemma E.12. *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. (Sketch) Consider the chain with fewest reads: $w_1 \prec_{PRED} u_1 \wedge u_1 \prec_{PRED} u_2 \wedge ...u_k \prec_{PRED} w_2$. If all the operations are writes then Lemma E.12 follows from the definition of history (Definition E.2) and the definition of $PRED$ for writes (DefinitionE.3[1]). Suppose that the chain has m reads. Let u_l be one of the reads. Using Lemma E.7, there must exist a later write u_j such that $p_{u_j} = p_{u_l}$. Now using Lemma E.11 we can construct an alternative chain from w_1 to u_j that comprises solely of write/view operations and doesn't have the read u_l. Repeating this process for all reads gives us a chain from w_1 to w_2 with no reads and hence, the result follows from the argument given earlier for a write only chain. □

Lemma E.13. *If w is a write/view and v is an operation accepted by a correct node p, then $w \prec v \Rightarrow w$ accepted by the node p prior to v.*

Proof. (Sketch) By induction on the number of reads in the chain with fewest reads: $w \prec_{PRED} u_1 \wedge u_1 \prec_{PRED} u_2 \wedge ...u_k \prec_{PRED} v$. If all the operations are writes (base case) then Lemma E.13 is true because of the *localHistory* check in *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. Now consider the case when the chain with fewest reads has $m + 1$ reads. Consider two cases.

Case 1: v **is a read.** If the length of chain is 2, the result follows from the definition of $PRED$ for reads. If the $length > 2$, we have $u_k \in PRED_v$. Using the induction hypothesis, we have that $w \prec u_k$ and hence w must have been accepted by p before u_k. But, from definition of $PRED$ for reads, u_k was received by p before v. Combining these two, we get our desired result.

Case 2: v **is not a read.** Suppose that u_l is the last read. If $p = p_{u_l}$, then by Lemma E.10 we get that u_l must have been accepted before v and from induction hypothesis and case 1, we have $u \prec u_l \Rightarrow u$ was accepted before u_l. Combining these two results, we get our desired result.

Now, consider the case when $p_{u_l} \neq p$. Recall that a read r is only added to the $PRED$ sets of local operations issued after r based on the construction. Using Lemma E.7, there must exist a later write u_j such that $p_{u_j} = p_{u_l}$. Now using Lemma E.11 we can construct an alternative chain from w to u_j that comprises solely of write/view operations and doesn't have the read u_l. Hence, using the induction hypothesis, the desired result must be true. □

E.1 Bounding forks

DEFINITION E.14. *An operation u is said to be* observed *by a correct node p in a view graph G iff there exists an operation v issued by p such that $u \prec_G v$.*

DEFINITION E.15. *A fork is said to be* observed *by a correct node p if its leaf vertex is observed by p.*

DEFINITION E.16. *Let $MaxFork(n, k)$ denotes the maximum number of distinct forks (from any faulty node) observed by correct nodes in a system with n correct nodes and k Byzantine nodes.*

DEFINITION E.17. *A directed acyclic graph G is called a* view graph *for an execution e if G contains a read/write vertex for every operation of a correct node and G satisfies the following conditions:*

VC1 Serial ordering. *Views, writes, and reads of correct nodes are totally ordered in an order consistent with the order in which reads and writes were issued in an execution. Specifically, if v and v' are vertices corresponding to operations by the same correct node, then $v.startTime < v'.startTime \Leftrightarrow v \prec_G v'$.*

Operations of a faulty node form a directed tree. If the tree has more than one leaf vertices (i.e. the tree isn't a total order), then we call each path from the root to the leaf in such a tree a fork. *In other words, if a faulty node behaves as a correct node and all its operations are totally ordered, then we don't consider its operations to be part of a fork.*

VC2 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: $w \in r.wl \Leftrightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : (w \prec_G w' \prec_G r \wedge w'.objId = r.objId))$*

VC3 Sharing with correct nodes. *If there exists an edge from view v_1 at p_1 to view v_2 at p_2 then both p_1 and p_2 are correct in v_2. A node p is correct in a view v if all the vertices v' of p (reads/writes/views) in v (i.e. all vertices $v' : v' \prec_G v$) are totally ordered.*

VC4 Time doesn't travel backward. *For any read/write operations u, v by correct nodes:*

$$u.endTime < v.startTime \Rightarrow v \nprec_G u.$$

VC5 Operations of a faulty node are observed by some correct node. *For any operation u by a faulty node $u \in G \Rightarrow \exists v : u \prec_G v \wedge p_v$ is correct.*

LEMMA E.18. $MaxFork(n, k) \leq n. MaxFork(1, k)$.

Proof. (Sketch) Consider a view graph that contains the maximum number of forks that are observed by correct nodes. Now remove all edges that connect two vertices at correct nodes in this graph. It is easy to see that this transformation doesn't reduce the number of forks observed by correct nodes; the number of forks is defined by the number of branches of faulty nodes. Note that while this transformation may affect the number of forks observed by a particular correct node, it doesn't 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. Next, we argue that in a view graph with maximum number of forks, no two correct node should observe the same fork; exposing the same fork to multiple correct node doesn't add to the total number of forks. Instead, the total fork count can be increased by showing different forks to different correct nodes.

Combining these observations, we make two claims. First, each correct node c will observe at most $MaxFork(1, k)$ forks with k faulty nodes. Second, because forks produced for different correct nodes must be different in an optimal view graph, we can have a maximum of $n \cdot MaxFork(1, k)$ forks in an optimal view graph with n correct nodes and k Byzantine faulty nodes. □

LEMMA E.19. $MaxFork(1, k) \leq 2. MaxFork(1, k - 1)$.

Proof. (Sketch) Let u be one of the last operation from a faulty node that was observed by the correct node c (there could be multiple such operations that were observed simultaneously). Let v be the earliest vertex at c such that $u \rightarrow v$. Let G be the original optimal view graph and G^u denote the projection of the graph G such that G^u contains all vertices $w : w \prec_G u$ and all edges connecting these vertices. Since c is the only correct node in G, and u is the last operation accepted by c in G, we know that (1) p_u must be correct in v (Definition E.17[VC3]), (2) all operations by p_u in G must be totally ordered and precede u. Now, G_u is a view graph containing $k - 1$ nodes that are allowed to create forks (since p_u must ensure that all its operations are totally ordered and hence act as a correct node). Therefore, the number of forks that c can observe through u must be $MaxFork(1, k-1)$. Consider G' obtained by removing G^u from G. G' doesn't contain any operation from p_u since u was the last operation observed by c in our view graph G. G' contains one correct node c and $k-1$ faulty nodes. Therefore, the maximum number of forks that c can observe in G' are $MaxFork(1, k - 1)$. Combining these two observations, the maximum number of forks that a correct node can observe in presence of k Byzantine faulty nodes is $2. MaxFork(1, k - 1)$. □

LEMMA E.20. *In a view graph for k faulty nodes and n correct nodes, at most $n \cdot 2^{k-1}$ distinct forks can be observed by correct nodes.*

Proof. (Sketch) From Lemma E.19, we have $MaxFork(1, k) = 2. MaxFork(1, k - 1)$. Solving this recursive relation using the base case of $MaxFork(2, 1) = 1$, we get $MaxFork(k + 1, k) \leq 2^{k-1}$. Combining this with the result of Lemma E.18, we get $MaxFork(n, k) \leq n. 2^{k-1}$. □

E.2 Enforcing BFJC consistency

LEMMA E.21. *Every execution of the pseudocode in Table 1 can be mapped to a view graph.*

Proof. (Sketch) Consider the graph G such that G contains vertices for all operations by correct node and writes/views by faulty nodes that have been accepted by some correct node. If $u \rightarrow v$ then add an edge from u to v. Now, we need to show that the resulting graph is acyclic and upholds the conditions of a view graph.

$\prec$ graph is acyclic. Follows from Lemma E.8.

Serial ordering. If $v.nodeID = v'.nodeID$ and $nodeID_v$ is correct, then $v.startTime < v'.startTime \Leftrightarrow v \prec v'$. Follows from Lemma E.10.

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:

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

1. *"if"*. $w \in r.wl \Rightarrow (w \prec_G r \wedge w.objId = r.objId) \wedge (\nexists w' : w \prec_G w' \prec_G r \wedge w'.objId = r.objId.$ A correct node p issues a read r that returns a set wl of writes. Reads to an object id oId, return the value of the store map for key oId. Therefore, $w \in r.wl$ implies that w must have been accepted by p before it issued r because 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. Therefore, $w \prec_{PRED} r$ from Definition E.3[3]. Furthermore, if there exists w' such that $w \prec w' \prec r$, then from Lemma E.13, w' should have been accepted by p after it has accepted w but before issuing r. Furthermore, $w \prec w' \Rightarrow w \in H_{w'}$ from Lemma E.12. Hence $prec(w, w')$ must have returned true. Therefore, by pseudo-code line 86, p would have removed w from the store on accepting w' and therefore not returned w on read r. Hence, by contradiction, this case is true.

2. *"only if"*. $w \in wl \Leftarrow w \prec_G r \wedge \nexists w' : w \prec_G w' \prec_G r$. Let p be the correct node that issued r. $w \prec r$ implies that from Lemma E.13, p received w before issuing read r and therefore, by the pseudocode step 87, w must have been added to the store. Suppose that a subsequent write w' accepted by p removes w at pseudocode line 86. For this removal, $prec(w, w')$ must have returned true to pass line 85. Therefore, by $PRED$ construction (Definition E.3[1]), $w \prec w'$. Also, by construction (Definition E.3[3]), $w' \prec r$, violating the second precondition. Therefore, by contradiction, no such w' can exist and hence this case is true.

Sharing with correct nodes. If there exist an edge from a vertex v_1 at p_1 to a vertex v_2 at p_2 then p_1 is correct in v_2. A node p is correct in a vertex v if all the vertices v' of p (reads/writes/views) in v (i.e. all vertices $v' : v' \prec_G v$) are totally ordered.

Suppose by contradiction that p_1 is faulty in vertex v_2. If p_1 is a correct node then it must be correct in all vertices from the $VC1$ requirement of VFJC consistency so p_1 must be faulty for this assumption to hold true. Recall that only writes and views from faulty nodes are used to construct the $prec$ relation and 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 $v_1 \in v_2.prevUpdate$ as otherwise there exist 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. It must have checked that $v_2.prevUpdates$ are correct in 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 v_2.

2. v_2 **is a read operation**: p_2 must be correct because we don't consider read operations by faulty nodes. 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 E.9. Only then can a later read be serviced. Furthermore, we must have that $v \prec v_2$ (from definition of $PRED$ for reads). Combining these two, we get that $v_1 \prec v \prec v_2$ contradicting the assumption that v_1 and v_2 are connected through a non-local edge.

Time doesn't travel backward. For any read/write operations u, v by correct nodes:

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

By way of contradiction, assume that u and v are operations by correct nodes and $u.endtime < v.startTime \wedge v \prec_G u$. u must be accepted by some correct node p. Note that from Definition E.1, for an operation u by a correct node, $u.startTime < ts_{p_u,u} < u.endTime$. Now consider cases on u:

u **is a write operation:** From Lemma E.13, u must be accepted by p_v prior to v but u was issued after v completed. Contradiction and hence this case isn't possible.

u **is a read operation:** If $p_u = p_v$, then the desired result follows from the serial ordering property. If $p_u \neq p_v$, then there must be write w such that $u \prec_G w \prec_G v$ and $p_w = p_u$ (from Lemma E.7). We know from the earlier case (when u was a write) then w must have been issued before v and from the serial ordering at node p_u, u must have been issued before w, therefore, we can't have $u.startTime > v.endTime$.

Operations of a faulty node are observed by some correct node Follows from the construction of graph G. □

THEOREM E.22. *The pseudocode in Table 1 enforces BFJC consistency.*

Proof. (Sketch) From Lemma E.21 there exists a *view graph* G for every execution e of listing in Table 1. Construct graph G' using G as follows. G' contains all the read and write vertices of G but view vertices are removed. Let $prev_u$ denote the non-view operation that immediately precedes an operation u at the node p_u. If no previous operation exists, the $prev_u$ is $\perp$. Similarly, $next_u$ denotes the earliest operation such that u precedes $next_u$ and $next_u$ is $\perp$ if no such operation exists. Now, iterate through all the view vertices v in G and do the following:

1. For each incoming edge from a vertex u to a view vertex v, add an edge from u to $next_v$ if $next_v$ is $non-\perp$. Otherwise, do nothing.

2. For each outgoing edge from the view vertex v to a vertex u, add an edge from $prev_v$ to u.

3. Remove v and all edges connecting v.

At the end of this construction, we will have graph G' with only reads and writes and with the property that $u \prec_{G'} v \Leftrightarrow u \prec_G v$. Properties BFJC1, BFJC2, and BFJC3 follow directly from properties VC1, VC2, and VC4 of the view graph G used to construct G'. Property BFJC4 follows from Lemma E.20. □

E.3 Liveness

THEOREM E.23. *The pseudocode listing 1 is available for reads and writes.*

Proof. (Sketch) All reads created by a correct node are accepted. However, for writes we need to argue that the update corresponding to a write and the view update following a write will be accepted. We first claim that *lastWrites* at a correct node is always empty after the lock is released. The proof for this claim is as follows: initially the *lastWrites* set must be empty. After a successful write, it will again be empty as the new view operation will supersede all previous entries. Similarly, after a successful packet receipt, the *lastWrites* will again be empty as the new view operation will supersede all previous entries. Based on this argument, we can claim that new write update and the following view update will be successfully applied—(1) *historyLocal* check will pass as the *prevUpdate* is assigned from *lastWrites*, a set containing updates that have already been accepted, (2) *noFaultyChild* will pass because the only update in the *prevUpdate* is from the correct node that is creating the update children, and (3) *signed* 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. □

LEMMA E.24. *Correct nodes p and q that have accepted identical updates will return identical responses on reads to any object o.*

Proof. (Sketch) Let U be the set of updates received by both these nodes. Let p and q issue reads r_p and r_q respectively to object o. Consider the set of updates $U_p : u \prec r_p$ and $U_q : u \prec r_q$. We claim that $U_p = U_q = U$. If not, then by Lemma E.13, p must have received all updates in U_p and likewise q must have received all updates in U_q. Also, from construction Definition E.3[3], $\forall u \in U, u \prec r_p$ and $\forall u \in U, u \prec r_q$ and similarly for q. Hence, we must have $U_p = U_q = U$. Given this claim, both the reads must return the same answer from the reads must return the most recent concurrent writes requirement of VFJC consistency. □

LEMMA E.25. *A correct node q processes an update packet T from a correct node p that has accepted updates U_p. We claim $U_p \subseteq U_q$ after q processes the packet T (accepts or rejects).*

Proof. (Sketch) Suppose that when the packet T was received, q has accepted U'_q updates. If $U_p \subseteq U'_q$ then the desired result follows as updates are never removed from a node's log. Consider the case when $U_p \not\subseteq U'_q$.

Let u be the first update (if any) that fails the *verify* check in the *intApply* function. The *historyLocal* will be satisfied because all updates in the history are sent in T and u is the first rejected update. Similarly, the *signed* check should be satisfied if u was accepted by a correct node p. Finally, u's children (updates whose hashes are included in $u.prevUpdate$) must be correct in u because u was accepted by a correct node p that must have performed the same check. Hence, no such update u from p will be rejected.

There are two other ways in which a packet might be rejected:

1. The *lastWriteViews* check might fail at step 61. We next show that this isn't possible.

   We first note that at p there must exist a view update v such that $\forall u \in Log_p, prec(u, v) = true$. If the last operation was a write, then the result follows from pseudocode step 29 and the availability requirement that the view operation won't be rejected. If it was a successful packet apply then the result follows from pseudocode step 60.

   Now 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 must pass as the *lastWrite* will contain at most two view updates from p and q.

2. Second, we need to show that the *intApply* at step 62 will succeed. The claim follows because, as argued above, the *lastWrite* will contain at most two view updates from p and q, both of which are correct and hence the *noFaultyChild* must pass. Similarly, the *historyLocal* check will pass because all the updates in *lastWrites* have already been applied to *log* and *signed* will pass because the update will be properly signed by the correct node q creating the view update.

□

THEOREM E.26. *The pseudocode listing 1 is one-way convergent.*

Proof. (Sketch) Follows from Lemma E.25 and Lemma E.24. □

THEOREM E.27. *Bounded fork join causal (BFJC) consistency can be enforced by an always available and one-way convergent implementation.*

Proof. (Sketch) Follows from Theorem E.26, Theorem E.23 and Theorem E.22. □

F RTC is enforceable using a CAC implementation

THEOREM F.1. *Real time causal consistency can be enforced by an always available and one-way convergent implementation.*

Proof. (Sketch) Follows from Theorem E.27 and from noting that in absence of Byzantine nodes, BFJC consistency reduces to RTC. □