

# ASTRO: Autonomous and Trustworthy Data Sharing

Prince Mahajan, Sangmin Lee, Jiandan Zheng, Lorenzo Alvisi and Mike Dahlin  
University of Texas at Austin

UTCS TR-09-29

*This technical report supersedes UTCS TR-08-24*

**Abstract:** We present ASTRO, the first system that supports trustworthy data sharing with strong consistency guarantees among a distributed collection of autonomous nodes. Autonomous nodes support data reads and writes without coordination with other nodes, share data opportunistically in a peer-to-peer fashion, and are mutually distrustful. ASTRO enforces *fork-causal consistency*, a new consistency semantics that, though weaker than causal consistency, provides two key properties: (1) it prevents Byzantine nodes from undetectably modifying the causal history of updates created or accepted by correct nodes, and (2) it guarantees that correct nodes accept only updates that extend the causal history of the updates they already know. We prove that fork-causal consistency is the strongest weakening of causal consistency achievable among autonomous nodes and show that it can be enforced efficiently while giving nodes the freedom to control their bandwidth and storage resources by selecting which updates they are interested in receiving.

## 1 Introduction

This paper presents the first architecture that supports trustworthy data sharing with strong consistency guarantees among a distributed collection of *autonomous* nodes. Such nodes support data reads and writes without coordination with other nodes, share data opportunistically in a peer-to-peer fashion, and are mutually distrustful.

Supporting nodes’ autonomy is critical in many practical data sharing applications—including delay tolerant networking (DTN) [2, 13, 25, 32, 39], personal data sharing [4, 31, 34], and cooperative data sharing across multiple administrative domains [1, 29]—where peers operate in settings where their connectivity, dependability, and level of mutual trust vary dramatically.

Several existing systems address a subset of these challenges, but their solutions don’t apply to autonomous nodes. For example, weak consistent replication systems support peer to peer data sharing and are highly available [8, 33, 30, 34], but are designed for environment where nodes trust each other. Systems that instead rely on a centralized server [10, 20, 23, 27, 26]—possibly implemented through a collection of untrusted servers [11, 22]—don’t support disconnected operations and peer-to-peer data sharing.

This work introduces ASTRO, a data sharing system that resolves the fundamental tension between peer-to-

peer sharing and mutual distrust by relaxing the consistency requirements on the updates accepted by autonomous nodes. Nonetheless, ASTRO guarantees that the data accessed by correct nodes satisfy a notion of consistency that is not only globally meaningful but also, in a precise way, is the strongest relaxation of causal consistency [18] achievable between autonomous nodes.

ASTRO supports autonomy along three dimensions:

1. **Limited Trust:** ASTRO provides correct nodes with sufficient local information to determine, regardless of the faults or manipulations of other nodes, whether it is safe for them to accept the updates they receive.
2. **Opportunistic Sharing:** ASTRO supports disconnected workgroups and peer-to-peer data sharing. As a result, ASTRO remains available despite network partitions or node failures.
3. **Selective Sharing:** ASTRO gives nodes control over their storage. Nodes can garbage collect their logs and decide which subset of the updates created in the system they want to receive.

ASTRO’s ability to achieve high availability while supporting partial information exchange among mutually distrustful nodes comes at a price: familiar guarantees such as *causal consistency*—the guarantee at the core of Bayou [33] and PRACTI [8]—become unattainable.

ASTRO provides instead a new consistency semantics, which we call *fork-causal consistency*. As its name suggests, fork-causal consistency weakens causal consistency by adopting an approach inspired by SUNDR’s *fork-linearizability*.

Fork-linearizability limits the degree by which a faulty server can disrupt linearizability: a faulty server can at most “fork” its history, leading different clients to observe histories that, though incompatible, are linearizable when taken in isolation. Similarly, fork-causal consistency limits how much a faulty node can harm causal consistency; a faulty node can at most fork its history, leading different peers to observe histories that, although incompatible, are causal when taken in isolation. This guarantee greatly reduces the attacks a faulty node can mount. For instance, a faulty replica can no longer expose arbitrary subsets of updates from one correct replica to another correct replica even if the faulty replica controls the only channel of communication between the correct replicas. Furthermore, ASTRO ensures that upon exchanging updates, correct nodes will be able to detect any

mutual inconsistency. We prove that when nodes are autonomous and potentially faulty or malicious, fork-causal consistency is the strongest possible weakening of causal consistency, *i.e.* it is the weakening of causal consistency that admits the smallest number of additional executions.

The main challenge in providing fork-causal consistency in ASTRO is to prevent faulty processes from exploiting nodes’ autonomy to their advantage. For instance, the requirement to support selective sharing means that, even with perfect connectivity, a given node may never receive certain updates. Faulty nodes may try to exploit partial information to create fraudulent gaps and reorderings in the updates they exchange. ASTRO addresses this challenge by supporting secure *update summaries*. Precise update summaries are equivalent to traditional updates, but ASTRO allows also imprecise summaries to be sent in lieu of updates the receiver doesn’t care for.

Each update summary carries with it a proof of its accuracy. One of the key properties of ASTRO’s summaries is that the proofs they carry are associative: the proof for the summary that covers a sequence of updates  $u_1 \dots u_n$  can be verified by combining the proofs carried by the individual updates that the summary claims to cover. This property gives ASTRO nodes maximum flexibility in verifying whether their state is consistent with the state of a node they want to exchange summaries with, and it is crucial to marry selective and opportunistic sharing data sharing with strong consistency guarantees, despite limited trust.

We have built a prototype of our system using the PRACTI framework and our evaluation demonstrates that ASTRO adds only 2% CPU and 19% bandwidth overhead over approaches that guarantee data integrity but no consistency.

In summary, we make the following contributions:

1. We propose new consistency semantics, *fork-causal consistency*, that we argue enforce the right semantics for distributed and highly available environments with potentially faulty or malicious participants.
2. We show that when nodes are autonomous and may be faulty or malicious, fork-causal consistency is the strongest possible weakening of causal consistency.
3. We introduce ASTRO, a data sharing system for autonomous nodes that guarantees fork-causal consistency while supporting opportunistic and selective data sharing among mutually distrustful nodes.
4. We develop a new construct based on Merkle trees that generates associative proofs crucial to implementing the secure update summaries on which ASTRO relies.

The remainder of this paper is organized as follows. Section 2 describes our threat model and assumptions. Section 3 introduces fork-causal consistency. Section 4 and 5 describe protocols that achieve fork-causal consistency in autonomous environments with complete and in-

complete information, respectively. Finally, we evaluate our system, discuss related work and conclude.

## 2 Model

ASTRO is designed to operate in settings where connectivity, dependability, and the level of mutual trust vary dramatically. We thus assume an asynchronous system where nodes exchange *updates* to *objects* through a network that may fail to deliver messages, delay them, duplicate them, or deliver them out of order. Additionally, connectivity between nodes may vary over time.

We assume a Byzantine failure model, in which faulty nodes may deviate arbitrarily from the protocol due to bugs, misconfiguration, or malice. We put no bound on the number of Byzantine nodes, but we assume that they cannot subvert cryptographic primitives such as digital signatures and one-way hashes. We use the notation  $\langle m \rangle_p$  to indicate that message  $m$  is signed by node  $p$ .

## 3 Consistency

Access control and consistency are two important properties that together ensure the correctness of a system. Access control prevents unauthorized nodes from reading/writing objects and is enforced using standard cryptographic techniques by signing updates, encrypting data, and rejecting unauthorized reads and writes [5, 26]. On the other hand, consistency is a contract between the system and the programmer that enables the programmer to reason about the results of operations. Intuitively, consistency constraints the ordering of events to make the results of reads and writes predictable and well-defined.

Strong consistency semantics simplify reasoning about correctness because all nodes observe similar ordering of events. For instance, linearizability [17] ensures that all nodes observe each write happening in the same order and at the same instant. Unfortunately, in a distributed system, enforcing strong consistency semantics such as linearizability or sequential consistency has high, often unacceptable costs in terms of loss of availability or performance overheads [11, 22].

As a result, distributed systems often use consistency semantics that are weaker than sequential consistency or linearizability but still provide some intuitive basis for reasoning about the system. For example, Monotonic write (FIFO/PRAM) consistency [40, 21] ensures that if a node performs write  $w_1$  followed by write  $w_2$ , then all other nodes observe this same ordering. On the other hand, monotonic read consistency [40] guarantees that if a process has observed a particular value for the object any subsequent accesses will never return any previous values. Causal consistency [7] strengthens this model to ensure that if a node observes write  $w_1$  (done by itself or by some other node) and then performs write  $w_2$ , then all other nodes that observe these dependent writes observe

them in the same order, *i.e.*  $w_1$  followed by  $w_2$ . A different approach is taken by eventual consistency, which allows transient inconsistencies but ensures that once sufficient communication has happened and no more updates occur in the system, every node agrees on the final value of all the objects.

In benign environments, consistency enables application developers to rely on invariants, thereby simplifying design and development [6, 9]. Lack of consistency can make the developer’s life very difficult; as Werner Vogels noted, “*Systems that do not guarantee monotonic write consistency are notoriously hard to program*” [3].

Consistency is also important for the users of applications. In a study of communication needs during wildfires, Taylor et al. note that getting the story right during disasters can help make the situation predictable, limit loss to property, and make the difference between life and death [39]. On the other hand, re-orderings of emails and mailing lists information in a DTN mailing system can cause delivery of messages to unintended recipients or omission of some intended recipients [32, 36]. Absence of consistency in personal data sharing settings can prevent updates from a cellphone from reaching a laptop and vice-versa. Similarly, a peer-to-peer data sharing infrastructure in critical military settings cannot tolerate the possibility of an adversary reordering messages [25].

### 3.1 Consistency vs autonomy

Given the importance of consistency, what consistency semantics are enforceable in a system with autonomous nodes? Unfortunately, by requiring opportunistic sharing, limited trust, and selective sharing, many consistency guarantees are not just expensive but simply unachievable. The CAP Theorem [15] states that linearizability and sequential consistency are unenforceable in an asynchronous network where partitions can occur and one insists on 100% availability for reads and writes as required by autonomy.

SUNDR’s fork-linearizability [27] is similarly unattainable since it must enforce linearizability for runs where nodes do not fail. Moreover, SUNDR has three additional requirements regarding the trust and availability of autonomous nodes. First, SUNDR relies on a central server through which all updates flow. Hence, even if nodes can physically communicate with each other, they cannot exchange updates if the server is unavailable or unreachable, violating opportunistic sharing. Second, although SUNDR tolerates a faulty server, it requires all clients to be correct to ensure safety, violating the limited trust requirement. Figure 1 illustrates a scenario in which faulty clients collude with the faulty server to violate fork-linearizability. Suppose in the open-source CVS example on which the SUNDR paper focuses, an attacker controlling a compromised client  $F$  can issue two updates  $u = m_{innocuous}$  (say, a

changed comment) and  $u' = m_{attack}$  (say, a backdoor or code with a buffer overflow bug that allows stack smashing and compromise of the program) such that  $m_{innocuous}$  and  $m_{attack}$  have the same version information ((0, 1, 0) in the given example). Then, the faulty server can show  $u = m_{innocuous}$  to one set of clients  $A$  (e.g., the auditor for the module in question) and show  $u' = m_{attack}$  to other clients  $C$  (e.g., non-contributing users who download the source tree to compile for production use) while still allowing  $A$  to continue to see  $C$ ’s updates and vice versa (*i.e.*, without detectably “forking” the histories). Third, a fail-stop client failure can block access to a data item if it fails at an inopportune time during an update [27].

Causal and FIFO consistency are impossible to guarantee in an autonomous environment with misbehaving or malicious nodes. Both consistency semantics require all the writes performed by a node to be observed in the same order by every node in the system. A misbehaving node, however, can create and send two different updates to two different correct nodes. By reading the received value, the correct nodes would violate causal (and FIFO) consistency—unfortunately, if the correct nodes cannot communicate directly, there is no way for them to determine whether or not the updates they receive violate consistency.

Supporting trustworthy data sharing among autonomous nodes requires operating under a fundamentally weaker system model. To cope with this challenge, we introduce a new notion of consistency.

### 3.2 Fork-causal consistency

We begin by associating with each execution an *observer graph* that captures how information flows during the execution. This is not an actual graph that our protocol maintains, but it is useful for presentation purposes. To build this graph, we ask each node to disclose the dependencies between the operations it executes. The graph has two types of vertices:

1. *Read vertices* are tuples of the form  $(n, old, wl)$ , where  $n$  denotes the node at which the read is performed,  $old$  is the identifier of the object being read, and  $wl$  denotes the list of vertices corresponding to the write operations whose value the read returns.
2. *Write vertices* are tuples of the form  $(q, old, val)$ , where  $val$  is the value written to object  $old$  by node  $q$ . Every object update creates a new write vertex.

In the absence of Byzantine nodes, the observer graph would contain one vertex for each read and write operation; a directed edge between two successive operations at each process; and a directed path between a read vertex  $r$  and the write vertex that wrote the value that  $r$  reads. It would be foolish, of course, to expect Byzantine nodes to play by these rules and disclose their dependencies accurately—all we can require is for correct nodes to cooperate.

Time	Action	Auditor(A)	Faulty Client(F)	Correct Client(C)	Server(S)'s VSL
0	initially	(0,0,0)	(0,0,0)	(0,0,0)	A:(0,0,0), F:(0,0,0), C:(0,0,0)
1	F writes u and u' with ts (0,1,0)	(0,0,0)	(0,1,0)	(0,0,0)	A:(0,0,0), F:(0,1,0), C:(0,0,0)
2	A reads u	(1,1,0)	(0,1,0)	(0,0,0)	A:(1,1,0), F:(0,1,0), C:(0,0,0)
3	A writes approval	(2,1,0)	(0,1,0)	(0,0,0)	A:(2,1,0), F:(0,1,0), C:(0,0,0)
4	C reads approval and F's write u'	(2,1,0)	(0,1,0)	(2,1,1)	A:(2,1,0), F:(0,1,0), C:(2,1,1)

Fig. 1: Illustration of an undetectable forking attack in SUNDR [27] in the presence of faulty clients. The server values denote the Version Structure List (VSL) maintained at the server, and the client values are the vector clocks maintained at the clients. The server's VSL values indicate the VSL obtained after the client has performed the operation with the same timestamp. For example, VSL at time 1 is the VSL obtained *after* F writes u.

Hence, the observer graph is only guaranteed to include, for every correct node  $p$ :

1. Read and write vertices for operations executed by  $p$ ;
2. Write vertices referenced in any read vertex of  $p$ ;
3. A directed edge between the vertices corresponding to two consecutive operations of  $p$ ;
4. A directed edge between write vertex  $w$  and any read vertex of  $p$  that reads from  $w$ .

**Definition 1.** We say that vertex  $u$  precedes vertex  $v$  in observer graph  $G$  (denoted as  $u \prec_G v$ ) if there is a directed path from  $u$  to  $v$  in  $G$ . By extension, we say that the operation corresponding to  $u$  precedes the one corresponding to  $v$ . If  $u \prec_G v$ , then  $v$  depends on  $u$ . If  $u \not\prec_G v$  and  $v \not\prec_G u$ , then we say that  $u$  and  $v$  are concurrent.

**Definition 2.** An operation  $u$  is said to be observed by a correct node  $p$  in  $G$  if either  $p$  executes  $u$  or if  $p$  executes an operation  $v$  such that  $u \prec_G v$ . We denote the projection of graph  $G$  over the set of operations observed by  $p$  as  $G^p$ .

We now define the set of executions admitted by our new notion of consistency in terms of the properties we expect of their corresponding observer graphs. In particular, we introduce two conditions that limit what Byzantine nodes can disclose (or fail to disclose) beyond the four basic requirements we listed above.

**Fork-causal consistency:** An execution  $\alpha$  is said to be *fork causally consistent* if it admits a corresponding observer graph  $G$  that satisfies the following properties:

1. For any correct node  $p$ , and for all nodes  $q$ , the precedes relation totally orders all operations of  $q$  observed by  $p$  in  $G$ .
2. If  $r = (p, old, wl)$  is a read vertex in  $G$  and  $p$  is correct, then there exists no write vertex  $w' = (q, old, val)$  in  $G$  s.t. one of the write vertices in  $wl$  precedes  $w'$  and  $w'$  precedes  $r$ . In other words,  $r$  returns the values of the latest concurrent writes to  $old$ .

It is easy to restate these requirements on the observer graph in terms of the guarantees that fork causal consistency offers to correct nodes:

**FC1:** Writes that are dependent on each other are observed in the same order by all correct nodes.

**FC1** implies that writes performed by a correct node are observed everywhere in the same total order. However, a Byzantine node  $b$  may create during an execution  $\alpha$  two writes  $b_1$  and  $b_2$  such that in no observer graph  $G$  that is admitted by  $\alpha$  either  $b_1 \prec_G b_2$  or  $b_2 \prec_G b_1$ . We call such concurrent writes *incompatible*.

**FC2:** Incompatible writes are never observed by a correct node.

**FC3:** A read  $r$  to an object  $old$  returns the values written by the most recent set of concurrent writes to  $old$  that precede  $r$ .

Fork-causal (FC) consistency is weaker than causal consistency: every causally consistent execution is also FC-consistent. In particular, FC consistency does not ask, as causal consistency does, that all operations of a node  $q$  be totally ordered. Instead, it limits its requirement to those operations that are observed by a correct node  $p$ . This weakening, when combined with **FC2**, implies that Byzantine nodes that *fork* their history by creating incompatible updates can prevent correct nodes from sharing updates past the point of incompatibility.

Nonetheless, FC consistency provides powerful guarantees: indeed, it is the strongest weakening of causal consistency achievable among autonomous nodes.

More precisely, consider a collection of autonomous nodes implementing a data sharing application  $\mathcal{A}$  that, in the absence of Byzantine faults, provides causal consistency and eventual consistency.

**Theorem:** *FC is the strongest consistency semantics achievable in  $\mathcal{A}$ .*

*Proof sketch.* We model an autonomous node  $p$  as a deterministic state machine whose state includes a representation of the observer graph induced by the operations observed by  $p$ . The state machine has three commands:  $READ(old)$ ,  $WRITE(old, val)$ , and  $SYNC_C(G)$ . The first two commands are invoked by  $\mathcal{A}$  running on  $p$  and involve local operations on object  $old$ ;  $SYNC_C$  is invoked by another autonomous node  $q$  and takes as a parameter  $q$ 's current observer graph  $G$ . When  $SYNC_C$  with  $q$ ,  $p$  learns of new updates; however, before incorporating them as part of its state,  $p$  checks whether the new observer graph that would result by joining  $G$  with  $p$ 's

current observer graph satisfies consistency condition  $C$ . To do so, it creates the hypothetical observer graph  $G'$  that would result if  $p$  were to read all the updates it received from  $q$ . If  $G'$  satisfies  $C$ , the  $p$  adopts  $G'$  as its new observer graph; if not,  $SYNC_C(G)$  fails and  $p$  rejects  $q$ 's updates. It is easy to see that if  $C$  admits an execution  $\alpha$ , then  $\alpha$  never causes a  $SYNC_C$  operation to fail at a correct process. It is also not hard to prove the following lemma: **Lemma:** If FC admits an execution  $\alpha$ , then, for each correct  $p$ , there exists a (possibly different) causally consistent execution  $\beta$  such that  $\alpha$  and  $\beta$  are indistinguishable by  $p$ .

Suppose now by contradiction that there exists a consistency condition SC for  $\mathcal{A}$  that is strictly stronger than FC but weaker than causal consistency. Then, in every execution the set of failed  $SYNC_{SC}(G)$  is a superset of the set of failed  $SYNC_{FC}(G)$ . Further, there exists at least one execution  $\gamma$  where no  $SYNC_{FC}$  fails, but  $SYNC_{SC}(G)$  fails at some correct process  $p$ . By our lemma, there exists a causally consistent execution  $\beta$  that for  $p$  is indistinguishable from  $\gamma$ . Hence, since  $SYNC_{SC}$  fails on some  $G$  in  $\gamma$ , it must also fail on the same  $G$  in  $\beta$ . But, since by assumption SC is weaker than causal consistency,  $SYNC_{SC}$  can never fail at  $p$  when executing  $\beta$ : contradiction.  $\square$

We argue that fork-causal consistency is the right semantics for data-sharing applications involving autonomous nodes for two reasons. First, as we just showed, no stronger consistency can be provided in these settings. The need for disconnected operations excludes linearizability, sequential consistency, and even the weaker fork-linearizability. Similarly, the presence of faulty nodes excludes causal consistency.

Second, fork-causal consistency is useful in many practical settings involving autonomous nodes. In executions with only fail-stop failures, fork-causal consistency converges to causal consistency, a desirable consistency semantics for autonomous nodes. In environments with buggy, misconfigured, or malicious nodes, fork-causal consistency limits damage by preventing faulty nodes from altering the causal dependencies observed by correct nodes.

## 4 ASTRO-CI

ASTRO-CI (Complete Information) is a protocol that enforces fork-causal consistency assuming *complete information*—nodes receive, process, and store all updates in the system. ASTRO-CI uses Bayou's log-exchange protocol for efficiently exchanging updates between nodes [33].

Note that simple approaches based on signed version vectors or vector clocks [35] are insufficient to guarantee fork-causal consistency. Consider, for instance, a faulty node that generates two updates, both with the same version vector or vector clock, and sends them to two different correct nodes  $p$  and  $q$ . Even after communicating

with each other,  $p$  and  $q$  won't detect these incompatible updates unless they decide to exchange and compare all the updates they know of, including those their version vectors say are already known to both of them. Besides being inefficient, this approach still falls short of providing fork-causal consistency: if all communication between two correct nodes  $p$  and  $q$  goes through a faulty node  $s$ , then  $s$  can selectively forward  $p$ 's updates to  $q$  and vice versa, violating the no-gap property.

ASTRO-CI is based on the observation that if the sender attaches all the updates it has seen to every update it creates, then the receiver can be assured of getting causal consistency in benign environments [9]. The same idea can be extended to environments with Byzantine nodes to get fork-causal consistency by detecting whether the sender and receiver are forked. So, a receiver additionally checks that the visible subset of the overall updates, received update and its history and the receiver's local history, doesn't contain a pair of incompatible updates; updates that are created by the same node but neither update occurs in the history of the other.

ASTRO-CI is an optimization of this simple idea. Instead of sending the entire history, a sender includes a *causal history summary (CHS)* to each update  $u$ . As described in the next section, a CHS provides a secure and efficient mechanism to summarize the set of updates on which  $u$  depends. Thus, an update  $u$  in our system has the following structure:  $\langle oId, lts, data, CHS \rangle_{cid}$ , where  $lts$  is the logical timestamp  $(cid, lc)$ ,  $cid$  is the creator of  $u$  and  $lc$  is the value of a counter local to  $cid$  that is incremented on every update.

### 4.1 Causal History Summary

To prevent faulty nodes from manipulating dependencies between updates, each correct node  $p$  running ASTRO-CI embeds in each update  $u$  it creates and signs the update's causal history summary (or CHS). The CHS for  $u$  is a tamper-evident summary of the set of updates that  $u$  depends upon. Its purpose, intuitively, is to offer a correct node  $q$  that receives  $u$  the ability to check whether  $u$ 's CHS is consistent with the dependencies induced by the updates that  $q$  has already observed.

Unfortunately, encoding the causal history of an update securely and efficiently is not straightforward. Simply encoding the causal history as a list of all prior updates does not scale to practical systems with a significant number of updates. Similarly, relying on a straightforward comparison of the hash of all the updates known to the creator and the receiver of  $u$  will not work in causally consistent systems, because causal consistency allows different nodes to observe different subsets of concurrent writes. Hence, different nodes may have observed different but compatible causal histories, and simple hashes across such compatible histories may not match. For example, in Figure 2 node  $b$  has updates

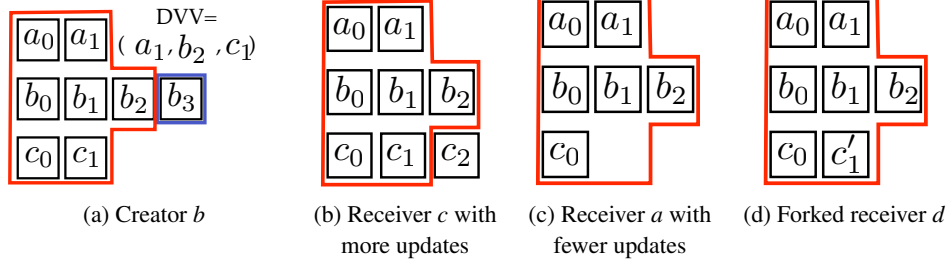


Fig. 2: Challenges of summarizing updates in a causal environment: (a) node  $b$  wants to create update  $b_3$ , (b) receiver  $c$  has seen more legal updates than  $b$ , (c) receiver  $a$  is missing some updates, and (d) receiver  $d$  is forked. The figure indicates the updates histories at each node when they receive update  $b_3$  and the outlined box indicates the updates that will be covered by the DVV( $a_1, b_2, c_1$ ) of the update  $b_3$ .

$a_0, a_1, b_0, b_1, b_2, c_0, c_1$  when it creates update  $b_3$ . However, a simple hash of these updates will not match with the hash of updates at the receiver  $c$  in Figure 2(b), even though  $c$  is causally consistent with  $b$ , because  $c$  possesses the extra update  $c_2$ . Note that fig 2(b) reflects a legal possible state in causal and fork-causal system but unfortunately, the simple hashing approach doesn't accommodate this possibility.

To address these issues, a CHS consists of a *dependency version vector (DVV)* and a *summary hash (SH)*. The DVV summarizes the updates known to the creator of  $u$ . For example, in Figure 2(a), the DVV of update  $b_3$  is  $(a_1, b_2, c_1)$ . SH is a summary hash of all updates included in the DVV: before they are hashed, updates are ordered according to their logical time stamp [33]. Note that the CHS of  $u$  summarizes the CHSs of all the updates included in the DVV—it is this recursive structure that allows a node that receives  $u$  to check whether  $u$ 's declared dependencies are consistent with what the receiver has already observed.

Going back to the example of Figure 2(b), in ASTRO-CI node  $c$ , upon receiving  $b_3$ , computes the hash of all the updates it has observed within the DVV. If the computed value matches  $b_3$ 's summary hash SH, then  $c$  accepts  $b_3$ .

Because of the DVV, every update in ASTRO-CI includes an  $O(N)$  version vector, where  $N$  is the number of nodes in the system. We reduce this overhead by observing that, since the receiver verifies new updates in causal order, the DVV created by node  $p$  only needs to include the vector entries that have changed since  $p$  created its previous update. This optimization allows nodes to include only an *Incremental DVV (IDVV)* in the (now) *Incremental CHS (ICHS)* of the updates they create.

In an experiment based on connectivity records of 100 mobile users [14], we observed an average IDVV size close to 1. Using IDVVs instead of DVVs also reduced bandwidth consumption by 95% and synchronization latency by 97%.

## 4.2 The Protocol

The creator of an update in our system simply generates the ICHS and includes it in the signed update. This update is then opportunistically propagated to other nodes

using a variant of Bayou's log exchange protocol. As in Bayou, each node maintains a version vector (CVV) that summarizes the updates it has accepted. When a node wants to receive more updates, it sends a request with its CVV to other nodes. A node  $q$  receiving such a request scans its logs to determine any updates that are not included in the requestor's CVV and sends these updates. Before accepting an update  $u$ , node  $q$  performs the following sequence of checks:

**Inclusion Check:** The inclusion check enforces **FC1** by ensuring that  $q$  has observed all the updates that  $u$  depends upon. The inclusion check involves verifying that the DVV of an update is included in  $q$ 's version vector and ensuring that the summary hash of the updates observed by  $q$  that are included in the DVV matches the summary hash of  $u$ .

**Compatibility Check:** The compatibility check enforces **FC2** by ensuring that neither  $u$  nor any of the updates that  $u$  depends upon is incompatible with any update that  $q$  has already observed. To check compatibility for updates that have passed the inclusion check, all that is required is to verify that (1) each newly received update  $u$  signed by node  $p$  has a higher logical time than any update from  $p$  previously observed by  $q$ , and (2)  $u$ 's IDVV includes the most recent update by  $p$  observed by  $q$ .

Once an update to object  $old$  has passed both checks,  $q$  uses it to update the set  $W_{old}$  that  $q$  keeps for each  $old$ . At all times,  $W_{old}$  contains the most recent updates to  $old$  that  $q$  has observed. ASTRO-CI enforces **FC3** by having each read operation that  $q$  performs on object  $old$  return the current content of  $q$ 's  $W_{old}$ .

The following theorem then follows directly:

**Theorem:** *ASTRO-CI enforces fork-causal consistency.*

ASTRO-CI supports several additional features:

**Immediate Fork Detection:** ASTRO-CI ensures that the first exchange of updates between forked nodes  $p$  and  $q$  will reveal the presence of incompatible updates, minimizing the damage caused by faulty nodes.

**Accountability:** ASTRO-CI allows a correct node that detects a fork to identify the incompatible updates and use them to generate a proof of misbehavior (POM) against the node that signed them.

**Reconciliation:** ASTRO-CI allows application-specific policies that relax fork-causal consistency to let correct nodes that observe incompatible updates reconcile their differences and proceed using the reconciled state. Reconciliation in ASTRO draws on prior mechanisms introduced by systems that support disconnected operation [8, 20, 29, 33, 34] by modeling a faulty node that forks its history into two incompatible branches as two nodes that issue conflicting updates. Choosing a policy, of course, remains a hard problem: the conflicting updates could have effects that ripple through the system—how can these issues be isolated and repaired? As with standard conflict resolution, we expect systems to adopt a range of automatic and manual conflict resolution techniques that make use of application requirements and semantics to determine an appropriate course of action such as automatically declaring one fork of the history to be the “truth” and discarding all updates from other forks, automatically merging compatible updates, or emailing a human to resolve a conflict. At some level such heuristics are unsatisfying, but we conjecture that systems like ASTRO-CI, which provide mechanisms to quickly detect faulty updates and provably identify nodes that issue them, can simplify recovery and minimize the damage from divergent histories.

## 5 ASTRO

Despite its many attractive properties, ASTRO-CI’s ability to support truly autonomous nodes is fundamentally limited by its requirement that all nodes receive and store all updates. ASTRO (Partial Information) is an extension of ASTRO-CI that eliminates the need for complete information, while still providing fork-causal consistency. ASTRO supports the exchange of partial information along the following dimensions to give nodes freedom to control their bandwidth and storage resources.

1. ASTRO supports secure, consistent checkpoints to allow log truncation/garbage collection. Rather than maintaining full logs, nodes can keep a suffix of the logs and exchange checkpoints when they need to communicate with nodes that need updates from before the truncation point [33].
2. ASTRO allows each node to specify independently an *interest set* that contains all the objects for which the node wants to receive updates. In ASTRO-CI, every node had the same interest set, which included all objects.
3. ASTRO supports also a combination of the two previous dimensions: if a node only cares about a subset of data and it needs to receive a checkpoint, it can receive a checkpoint that covers just the objects of interest.

Supporting the exchange of partial information is challenging. In benign environments, the missing information may hide some dependencies required for

enforcing causal consistency. Figure 3 shows a simple (and naive) approach to support partial information. Node  $a$  has updates  $a_0, a_1, \dots, a_8$ , which write to objects  $o_0, o_1, \dots, o_8$  respectively, and it wants to transmit these updates to node  $b$ . However,  $b$  is only interested in objects  $o_0, o_1$ , and  $o_8$ . Suppose that  $a$  simply transmits updates  $a_0, a_1, a_8$  to  $b$ . No harm is yet done: as long as  $b$  never accesses objects affected by the missing updates, it observes the same values and hence the same consistency as it would with complete information. The problems start when  $b$  synchronizes with another node  $c$  that is interested in all objects. Because  $c$  doesn’t know about the missing updates  $a_2 - a_7$ , successive reads  $o_8$  and  $o_5$  return for  $o_8$  the new value written  $a_8$  and for  $o_5$  some obsolete value written before  $a_5$ . This execution violates both causal and fork-causal consistency semantics:  $c$  observes the effect of update  $a_8$  without observing the effects of  $a_5$ , although correct node  $a$  observed  $a_5$  before  $a_8$ .

Byzantine nodes further complicate supporting the exchange of partial information for two reasons. First, in the absence of complete information omission and reordering errors may go unnoticed. Revisiting the previous example, suppose that  $a$  sends *all* updates to  $b$ , but that node  $b$  is Byzantine. Suppose that, as before, node  $c$  is interested in all updates, but that node  $b$  only exposes  $a_1, a_2, a_8$  to  $c$ , pretending to not possess  $a_2 - a_7$ . From  $c$ ’s point of view, this scenario is indistinguishable from the one in which  $b$  truly did not possess updates  $a_2 - a_7$ . Second, partial information makes CHS unusable. ASTRO-CI guards against Byzantine nodes by using CHS but, if nodes receive arbitrary subsets of the updates, the CHS that different nodes create will not match, thereby preventing even correct nodes from exchanging information.

To address these challenges, we introduce *updates summaries*. *Precise* summaries are equivalent to ASTRO-CI updates (and we will continue to represent them as such in our figures), but *imprecise* summaries can be used as a secure and tamper-evident substitute for a sequence of precise updates to objects outside of the receiver’s interest set. For example in Figure 3, node  $b$  receives summaries  $a_0, a_1, su_{2-7}, a_8$ , where  $su_{2-7}$  is an imprecise summary replacing precise summaries  $a_2 - a_7$ .

PRACTI [8] has shown that, in benign settings, simple update summaries of the form  $\langle old\_list, start\_lts, end\_lts \rangle$  can help save bandwidth and ensure causal consistency by informing the recipient that it should not read from the objects in *old\_list* until it retrieves the missing updates from some other source. The remaining of this section is dedicated to showing how ASTRO uses update summaries to support interest sets while guaranteeing fork-causal consistency between potentially Byzantine nodes. We structure our presentation in three steps. In Section 5.1 we define update summaries more precisely and give an axiomatic characterization of an *append* operator

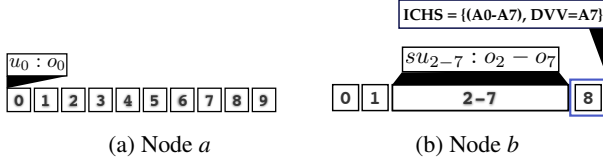


Fig. 3: The state of node  $a$  with updates  $a_0$  through  $a_8$  to objects  $o_0$  through  $o_8$  respectively and node  $b$  with a summarized update  $su_{2-7}$ , replacing updates  $a_2$  through  $a_7$ . All updates are done by node  $a$ .

that can combine summaries of consecutive sequences of updates in a single summary—deliberately postponing the details of its implementation. In Section 5.2 we discuss how our implementation of ASTRO relies on the properties of *append* to enforce fork-causal consistency. Finally, we present in Section 5.3 an implementation of *append* that guarantees the properties asserted in Section 5.1.

## 5.1 Update Summaries

A summary  $su$  is a tuple  $\langle old\_list, start\_lts, end\_lts, data, CHS \rangle_{cid}$ , where  $start\_lts$  and  $end\_lts$  denote respectively the logical timestamp of the earliest and the latest update included in the summary,  $cid$  is the creator of the updates summarized in  $su$ , and  $old\_list$  is the list of objects modified by the updates included in the summary. For precise summaries,  $start\_lts$  is the same  $end\_lts$  and  $data$  is the new value of the single object the summary covers; for imprecise summaries,  $data$  is left empty. Note that (1) all updates in a summary come from a single creator node and (2) summaries are signed. Both requirements are for ease of presentation: ASTRO can handle summaries with updates from multiple nodes and, although it only accepts summaries with the guarantees integrity and non-repudiation that signatures provides, ASTRO can leverage the tamper-evident CHS of a signed summary to serve as an implicit signature for all the summaries covered by that CHS.

We require three properties of summaries. First, for the purposes of enforcing consistency, observing a summary that correctly represents the updates it claims to cover must be equivalent to observing those individual updates. Hence, the receiver of a correct imprecise update  $su$  must be able to compute for all its updates that depend on  $su$  the same CHS that it would have computed if it had received the individual updates. Conversely, a summary that incorrectly represents the constituent updates—because it omits updates, tampers with the CHS, or misrepresents either the set of modified objects or the logical-time interval that the update spans—will, if observed, fork the receiver from those correct process who have observed the individual updates. Finally, summaries must be composable: any node  $p$  that possesses summaries—even if (implicitly) signed by some other node  $q$ —covering consecutive sequences of updates should be able to append them to create a single

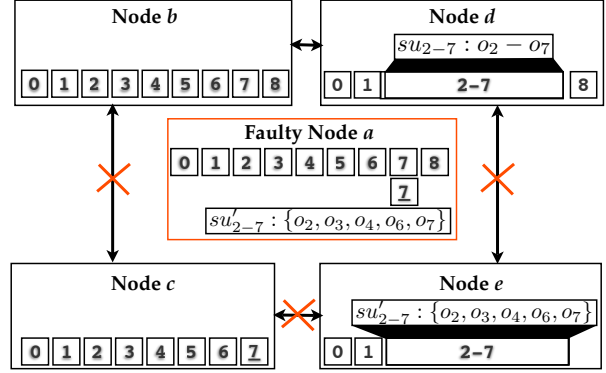


Fig. 4: Different scenarios of a faulty node exploiting partial information

new summary implicitly signed by  $q$ .

As we will see in the next section, we rely on the implementation of ASTRO to guarantee the first two properties. To address composability, we introduce instead an *append* operator  $\oplus$ . Informally, *append* accepts as input a pair of valid consecutive summaries and produces a new valid summary that encompasses both. We then say that a summary  $su$  is valid if either it is the result of an *append*, or if  $su$  satisfies the following three constraints: (1) it is signed, (2) it spans  $2^k$  updates for some  $k$  and (3)  $start\_lts$  and  $end\_lts + 1$  are multiples of  $2^k$ . Let now  $su^1$  and  $su^2$  be two valid summaries signed by  $p$ , such that that  $su^1 \prec_G su^2$  and  $start\_lts_{su^2} = end\_lts_{su^1} + 1$ . Then  $su^1 \oplus su^2 = su^3$ , where:

- $su^3$  is a valid summary signed by  $p$
- $start\_lts_{su^3} = start\_lts_{su^1}$  and  $end\_lts_{su^3} = end\_lts_{su^2}$
- $CHS_{su^3}$  includes all updates that precede the updates summarized in  $su^1$  and  $su^2$
- $old\_list_{su^3}$  is a deterministic ordering of the union of the objects in  $old\_list_{su^1}$  and  $old\_list_{su^2}$

If the input summaries don't conform to the above constraints, then *append* does not produce a valid summary. For inputs for which it does produce valid summaries, *append* offers two additional properties:

*Associativity*:  $\forall su^1, su^2, su^3, (su^1 \oplus su^2) \oplus su^3 = su^1 \oplus (su^2 \oplus su^3)$ .

*Collision resistance*:  $\forall su^1, su^2$ , it is difficult to find  $su^2$  such that  $su^1 \oplus su^2 = su^1 \oplus su^2$ .

We will rely on these properties next in describing how ASTRO works and in proving that it provides fork-causal consistency. We will discuss an implementation of *append* that guarantees these properties in Section 5.3.

## 5.2 The Protocol

The basic structure of ASTRO is similar to that of ASTRO-CI, except that now nodes share opportunistically summaries rather than updates. Of course, the presence of imprecise updates introduces some changes to how a correct node creates updates (a.k.a. precise summaries), forwards the summaries it has observed to its



peers, and verifies that observing the summaries received from a peer does not violate fork-causal consistency.

**Creating updates** As in ASTRO-CI, each precise summary  $su$  created by a correct node  $p$  contains the summary’s CHS<sup>1</sup>, consisting of a DVV and a summary hash SH. Node  $p$  computes SH in two steps. First, with the help of the *append* operator, it creates, for each peer  $q$ , a single summary encompassing all the summaries signed by  $q$  that  $p$  has observed, as reflected  $su$ ’s DVV. Then,  $p$  orders these summaries deterministically and creates a single hash SH across them.

**Opportunistic Sharing** To initiate an exchange, node  $p$  sends  $q$  its LPVV—the latest version vector for which  $p$  has observed all precise updates to the objects in  $p$  interest set. The LPVV is a special case of the CVV we used in ASTRO-CI, where nodes were interested in all objects. When a correct node  $q$  receives the LPVV from  $p$ , it responds with two types of summaries: (1) all precise summaries  $ps$  that are in  $p$ ’s interest set but are not included in  $p$ ’s LPVV and (2) for all such  $ps$ , all imprecise summaries  $is$  that  $ps$  depends upon such that  $is$  covers some update that  $p$  has not observed yet (i.e. such that  $end\_time_{is}$  is not included in  $p$ ’ LPVV).

**Accepting updates** ASTRO’s current policy for determining which of the summaries contained in  $q$ ’s response will be accepted by  $p$  is simple: to accept any of the summaries sent by  $q$ ,  $p$  must accept all of them. Although it is possible to implement a less Draconian approach, our evaluation of ASTRO using the mobility patterns of real users (see Section 6) suggest that this simple policy can perform well in realistic settings.

A key difference between ASTRO-CI and ASTRO that impacts which summaries are accepted by a correct node is that nodes in ASTRO may receive multiple overlapping summaries that claim to cover the same set of updates: a node may first receive an imprecise summary  $su_{2-7}$  claiming to cover updates  $b_2$  to  $b_7$ , and later receive precise updates for  $b_2$  and  $b_3$ , or a different imprecise update  $su_{6-9}$ . Partially overlapping summaries pose a threat to fork-causal consistency, as it is in general impossible to determine whether they cover incompatible updates. For example, suppose the creator  $b$  of  $su_{2-7}$  and  $su_{6-9}$  is Byzantine: the updates that  $b$  includes in positions 6 and 7 in the first summary may be incompatible with those that  $b$  uses for the same positions in the second summary—if a receiver accepted both, fork-causal consistency could be violated.

To avoid this situation,  $p$  applies the following composability check to every summary  $su$  received from  $q$  that overlaps with a summary  $p$  has already observed:

1. If  $su$  can be generated by appending a set of updates  $p$  has already observed, then  $p$  tentatively accepts  $su$ .

<sup>1</sup>Just as ASTRO-CI, ASTRO too supports ICHS, but for simplicity we omit the discussion of this optimization

2. If by appending  $su$  to other not-yet-rejected summaries received from  $q$  it is possible to generate a summary  $su'$  that  $p$  has already observed, then  $p$  tentatively accepts  $su$  and all the other summaries from  $q$  that were used to generate  $su'$ .
3. Otherwise,  $su$  is rejected.

The paragraph dedicated to Liveness later in this section discusses the implications of the composability check on the ability of correct nodes to share updates with each other.

For each summary  $su$  that does not overlap with any summary it has already  $p$ , performs the following sequence of checks:

**Validity Check:** The validity check ensures that  $su$  can be used as a valid input to the *append* operator.

**Inclusion Check:** The inclusion check enforces **FC1** by ensuring that  $q$  has accepted all the summaries and hence all the writes on which  $su$  depends. The inclusion check verifies that the hash of the summary of the update summaries observed by  $q$  that are included in the DVV matches the summary hash of  $su$ . The collision resistance property of the *append* operator ensures that it is improbable for the summaries that  $p$  and  $q$  compute to match unless these summaries were obtained by successive applications of the *append* operator over the same sequences of compatible writes.

Note that, although  $p$  may have summaries that overlap with all the write events in the DVV of  $su$ , the overlap may not be perfect:  $p$  may have accepted some summaries that straddle the cut in the observer graph defined by  $su$ ’s DVV. Once again, partial overlaps may limit what  $p$  can accept from  $q$

**Compatibility Check:** The compatibility check enforces **FC2** by ensuring that neither  $su$  nor any of the summaries that  $su$  depends upon includes writes that are incompatible with any that  $p$  has already observed. The check is virtually the same as in ASTRO-CI: if  $su$  has survived the inclusion check and does not overlap with any summary that  $p$  has already observed, then all that is required is to verify that (1) if  $su$  signed by  $b$ , then  $su$  has a higher *start\_tts* than the *end\_tts* of any update from  $p$  previously observed by  $b$ , and that (2)  $su$ ’s DVV includes the most recent update by  $b$  observed by  $p$ .

If a non-overlapping  $su$  passes the validity, inclusion, and compatibility check, then it is tentatively accepted.

If in the end all the summaries sent by  $q$  are tentatively accepted, then  $p$  uses the summaries to update the set  $W_{old}$  as described in the ASTRO-CI section to permanently accept the summaries. Hence ASTRO enforces **FC3** by having each read operation that  $q$  performs on *old* return the current content of  $q$ ’s  $W_{old}$ . The read to *old* blocks if  $W_{old}$  contains any imprecise summary. The following theorem then follows:

**Theorem:** *ASTRO-PI enforces fork-causal consistency.*

Figure 4 illustrates how the use of summaries ad-

addresses the attacks that a faulty node may mount. Suppose that the creator node  $a$  in the previous example is faulty. Node  $b$  has the updates  $a_0$  through  $a_8$  and node  $d$  has received a correct summary from node  $a$ . Hence, nodes  $b$  and  $d$  can exchange updates as indicated by the arrow connecting them. Node  $a$  now sends an different version of update  $u_7$  to node  $c$  which forks it with respect to both nodes  $b$  and  $d$ , who have received  $a_7$  instead. Forking in the figure is shown by the  $X$  sign over the communication channel between the nodes indicating that the nodes can't exchange updates without detecting the fork. Similarly,  $a$  omits object  $o_5$  from the update summary  $su'_{2-7}$  it sends to node  $e$ , thereby forking  $e$  with respect to nodes  $b$  and  $d$ . Note that even nodes  $c$  and  $e$  are mutually forked as they have both received inaccurate yet mutually incompatible update summaries.

**Liveness:** ASTRO creates summaries to avoid the cost of receiving, processing and storing updates that a node is not interested in accessing. Unfortunately, summarizing updates can sometimes prevent sharing of data in disconnected groups. For instance, if two correct nodes have received partially overlapping summaries (say  $su_{4-7}$  and  $su_{4-5}$ ), then, without additional information, ASTRO can not determine if the two summaries are compatible. Generally speaking, applications have two options to address this problem: (1) relaxing consistency to ensure complete availability, (2) giving up the ability to always exchange data. We sketch several mechanisms for each option and implement a few of these.

Relaxing consistency is an interesting alternative for applications that want to be always available despite partial information or don't require strong guarantees that fork-causal consistency provides. Our prototype supports relaxation of fork-causal consistency to allow nodes to accept incompatible summaries as long as they are incomparable but ensures that a node never observes comparable incompatible summaries. Nodes can eventually detect all forks, generate POMs, and reconcile while staying always live.

The second alternative is to maintain complete histories at some highly available servers and rely on these servers to provide the desired summaries when needed. This is a viable alternative for applications where autonomy is desired for performance and mistrust rather than limited connectivity. For instance, applications such as personal data sharing, where most interactions are expected to be within a trusted domain and for rare cross-domain interactions, a low bandwidth liveness server can be expected to be available, can rely on the server for providing the additional summaries to enable compatibility check. Similarly, in systems such as Microsoft Active Directory [12], nodes can expect the creator to maintain all the relevant history for summaries they create and obtain the relevant summaries from them on demand.

An alternative approach to coordinate communication

so that most summaries are generated are at the same granularity. For instance, applications spanning multiple administrative domains can ensure that nodes in one domain only accept updates created by nodes in other domain at certain pre-specified points in logical time. Collective enforcement of this policy in a domain guarantees that nodes within a domain can always mutually exchange information and be live.

**Checkpoints and Garbage Collection:** ASTRO uses summaries to construct checkpoints that only need to include the last update for each object: all other updates can be condensed as summaries. Thus, checkpoints store a collection of precise and imprecise summaries. Checkpoints can be used to garbage collect logs by replacing all but the last update to each object with summarized updates.

As we saw, independently garbage collecting the state may lead to generating incomparable summaries. Therefore, we need a mechanism for coordinating the garbage collection and ensuring that either (1) all the nodes in the system have reached the same state or (2) at least one node maintains enough precise summaries to identify the faulty node and reconcile the histories. Nodes in ASTRO rely on a server to maintain complete histories and garbage collect any prefix of their history that they know is present at the server. Such servers may be arranged hierarchically to ensure that only the root needs to store all the updates—intermediate nodes can simply maintain the histories relevant to their domain.

**Immediate detection, Reconciliation, and POM:** ASTRO uses the same approach for reconciling forked branches and POM generation as ASTRO-CI.

## 5.3 Implementation

In this section, we describe how ASTRO implements the *append* operator on summaries. We first introduce *associative hash summarization* (AHS), a new construct based on Merkle hash trees that supports associative summarization of a series of values and later we use it to implement *append*.

### 5.3.1 Associative Hash Summarization

Associative Hash Summary (AHS) is a substitute for simple hashing that supports the join operation (+) for associatively joining an AHS value to another. Let  $h_{i-j}$  denote the associative hash computed using series of values from  $v_i$  to  $v_j$ , where each value  $v_i$  has a unique label  $i$  that reflects its position in the list and the labels start at 0. Then AHS guarantees:  $h_{1-5} = h_1 + h_{2-5} = h_{1-2} + h_{3-5} = \dots = v_1 + v_2 + \dots + v_5$ .

We implement AHS using a simple construction based on Merkle hash trees [28]. A Merkle tree is a binary tree where the leaf nodes contain data values and the values at internal nodes are a hash of the concatenation of the values of the left and right child. The AHS of a list of values  $v_0, \dots, v_{n-1}$  is the ordered list of the roots of canon-

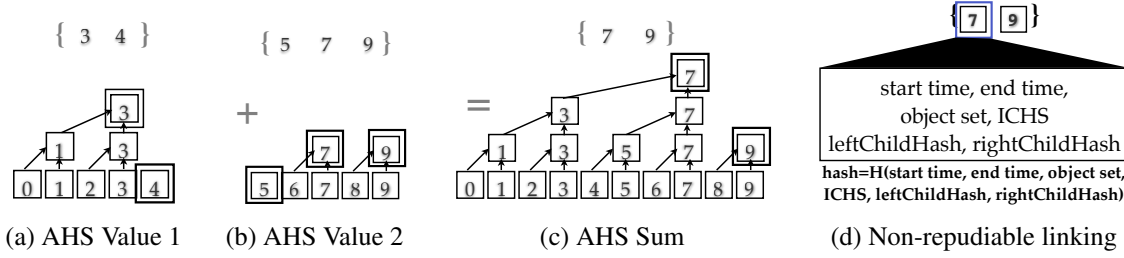


Fig. 5: The AHS values (a and b), their addition (c) and extending Merkle tree to support non-repudiable linking. The value indicated in parenthesis over the figure ( $h_{0-3}, v_4$  for Figure 5(a)) is the AHS value for the series of values shown in that figure.

ical Merkle trees built on  $v_0, \dots, v_{n-1}$  according to the rules described below. The join operation is supported by combining smaller consecutive trees to form larger trees. AHS inherits its collision resistance from the collision resistance of SHA1, which is used to construct Merkle trees in AHS. The associativity of AHS is guaranteed by adhering to the following two rules:

**AHS1:** An AHS contains only roots of balanced binary trees. This constraint implies that for a series of values, we may have to construct more than one Merkle trees on those values.

**AHS2:** The minimum label of a leaf of a tree spanning  $2^k$  elements is a multiple of  $2^k$ . Two adjacent tree roots covering  $2^k$  elements can be combined to form a tree containing  $2^{(k+1)}$  elements iff the first of the two roots has a starting label that is an even multiple of  $2^k$ .

These rules ensure that the size of AHS is logarithmic in the number of input values.

We argue that the maximum size of an AHS of  $n$  values is  $2\log(n) + 1$ . Note that in a given AHS, it is not possible to have a lower level value surround by a higher level value on both sides due to the structuring constraints described above. This observation implies that the AHS value can increase in level upto a maximum level and then must start decreasing. Hence, for  $n$  values we can't have more than  $2\log(n) + 1$  values.

Figure 5(a, b) illustrate AHS1 and AHS2. The leaf node with label  $i$  corresponds to value  $v_i$  and  $h_{0-1}$  is the internal node with label 1 formed by hashing together values  $v_0$  and  $v_1$ . AHS1 prevents the join of  $h_{0-3}$  and  $v_4$  and AHS2 sees that  $h_{6-7}$  and  $h_{8-9}$  are not combined to form  $h_{6-9}$ .

**Join:** The join operation on AHS first concatenates the roots of the trees in the input AHSs and then creates a larger tree by combining trees that have the same height and satisfy AHS2. The join process is repeated until no more trees satisfy AHS2. The roots of the remaining trees are the new AHS value produced by the join operation. Figure 5(c) illustrates the join of the two AHS values of Figure 5(a) and Figure 5(b). First the rootlist values are placed together to obtain  $\{h_{0-3}, v_4, v_5, h_{6-7}, h_{8-9}\}$ . Since  $\{v_4\}$  and  $\{v_5\}$  are both

level 0 trees, they are combined to form a level 1  $\{h_{4-5}\}$  tree and the list becomes  $\{h_{0-3}, h_{4-5}, h_{6-7}, h_{8-9}\}$ . The process is repeated to obtain the final result  $\{h_{0-7}, h_{8-9}\}$ . AHS2 prevents further merging between  $\{h_{0-7}\}$  and  $\{h_{8-9}\}$ . Note that this combination is associative as creating the AHS of the original sequence from  $v_0, \dots, v_9$  also produces the same AHS value,  $h_{0-7}, h_{8-9}$ .

### 5.3.2 Implementing Append

To support the *append* operator, we represent update summaries by using the tree nodes in the AHS. Leaf nodes represent precise summaries and internal nodes represent summaries for the precise updates present in the subtree rooted at that internal tree node. To attain this goal, we extend AHS by (1) replacing leaf nodes with precise summaries, (2) describing rules to deterministically construct the fields of a larger summary using the smaller summaries, and (3) securely associating the fields of an imprecise summary with the AHS of the subtree containing that summary.

The rules for constructing the fields of an internal tree node in an AHS tree are based on the requirements of the *append* operator specified in Section 5.1. Specifically, the *start\_lts* and the *end\_lts* of a tree node are computed by taking the min and max of the *start\_lts* and the *end\_lts* of its children. Similarly, the CHS of an internal leaf node is computed to include all updates that preceded the updates summarized in the CHS of its children. The objects are summarized using an application specific summarization strategy. Our prototype creates a sorted list of all modified objects. The data field is set to null for internal tree nodes.

The secure association between the AHS and fields of a summary ensures that any modification in any of the fields results in a different summary thereby guaranteeing the integrity and collision resistance of the summaries. ASTRO achieves this association as illustrated in fig 5(d). Typically, a hash for a node in a Merkle tree is calculated as the hash of the concatenation of its children hashes. We extend the hash construction to include the fields of a summary to establish the non-repudiable linking. Thus we have:  $H_{node} = H(H_{leftChild} || H_{rightChild} || start\_lt || end\_lt || old\_list || CHS || data)$ .

Whenever a summary  $su_{i-j}$  is exchanged, the sender sends the corresponding root hashes from the AHS  $h_{i-j}$ . Further, for each tree node  $h$  in AHS, the sender additionally sends  $start\_lts$ ,  $end\_lt$ ,  $old\_list$ ,  $CHS$ ,  $data$ ,  $h_l$  and  $h_r$  where  $h_l$  and  $h_r$  are the hash values corresponding to the left and right child of  $h$  (if any). Thus, a summary is implemented as a vector  $V$  of tuples  $(start\_lt, end\_lt, old\_list, CHS, data, h_l, h_r, h)$ .

**Operations:** Our implementation of *append* is designed to work for ASTRO. Therefore, it exploits the knowledge that summaries are first accepted tentatively and that if any summary  $su$  received during an exchange is rejected, then all the summaries received with  $su$  are rejected as well, even if they had been tentatively accepted. We leverage this knowledge to optimize the validity check: when determining whether to accept a set of summaries, we only invoke the validity check on the last summary sent by each creator. To be valid, every summary must submit to certain well-formedness constraint in terms of  $start\_lts$  and  $end\_lts$  of the summaries it spans, and it must be signed—either explicitly by its creator, or implicitly by appearing in the CHS of a later summary explicitly signed by the creator. Clearly, for any given node  $p$ , the last summary  $lsu_p$  created by  $p$  and received by  $q$  must be explicitly signed to be valid. If  $lsu_p$  is eventually accepted, it must also be well-formed, and the inclusion check that ASTRO performs guarantees that the CHS of  $lsu_p$  contains summaries for all of  $p$  write operations prior to  $lsu_p$ . Therefore, owing to the collision resistance of CHS, we implicitly possess a signature for the summaries of all prior updates performed by  $p$  too.

**Validity:** The receiver declares an imprecise summary, comprising of vector  $V$  of tuples  $(start\_lt, end\_lt, old\_list, CHS, data, h_l, h_r, h)$  valid if for each  $h \in v$ , it satisfies the three checks: (1)  $h = H(H_l || H_r || start\_lt || end\_lt || old\_list || CHS || data)$ , (2) each tree node  $t \in V$  satisfies the structuring constraints of AHS, and (3)  $v$  is signed. All precise summaries are valid if they are signed.

**Append:** The append operation on summaries  $su^1$  and  $su^2$  can now be implemented by joining the corresponding AHS after ensuring that the inputs are structurally well formed according to the constraints of AHS and that  $su^1 \prec_G su^2$  by performing the inclusion checks. To perform this check, summaries  $su^1$  and  $su^2$  are tentatively accepted and on failing this check, the update packet containing the summaries  $su^1$  and  $su^2$  is rejected. The signature for  $su^2$  serves as the signature for  $su^1 \oplus su^2$  because  $su^1 \prec su^2$  and hence  $su^1$  is present in the CHS of  $su^2$ . Now, either  $su^2$  is signed or it will be eventually rejected, and if  $su^2$  is signed, a proof for integrity and authenticity of  $su^1 \oplus su^2$  can also be structured. Therefore, we get that that  $su^1 \oplus su^2$  is also signed.

The associativity of append follows from the associativity of AHS join. The collision resistance property of summaries follows from the use of merkle hash trees. The result of append are valid because (1) outputs of AHS append satisfies the merge constraint and (2)  $su^1 \oplus su^2$  is signed.

## 6 Evaluation

We have constructed a prototype of ASTRO by extending the PRACTI framework [8] to support the autonomy features. ASTRO is built in Java and it Sun Java encryption library for performing cryptographic operations. Our evaluation shows that ASTRO has modest overhead compared to the approaches with integrity checks but no consistency checks. Since PRACTI is the system closest in spirit to ASTRO. Therefore, we compare performance of PRACTI and ASTRO. We also implemented a few other simple approaches discussed in literature to understand their performance characteristics and compare their overheads with ASTRO.

### 6.1 Micro-benchmarks

In this section, we compare the performance of PRACTI and ASTRO for some simple workloads. The key metrics we consider are *write latency*, *apply latency*, and *network bandwidth*. Write latency is the time to create a precise summary with appropriate fields for consistency, sign it, and store it. Apply latency is the time taken to verify an update summary received from the network and update local state. Network bandwidth gives the size of data sent on the wire. Our micro-benchmarks indicate that the our consistency checks impose insignificant overheads compared to the cost of maintaining integrity by signing updates. The Java encryption library we used takes 6.0 ms for signature generation and 11.9 ms for verification. We do 10000 writes to 1000 objects for experiments in this section.

**Write latency:** We compare the write latency of three systems with varying object sizes: PRACTI, PRACTI with signatures, and ASTRO. As shown in Figure 6(a), the most of the overhead occurs comes from signature operations. ASTRO incurs an insignificant additional overhead of 2% over the signed PRACTI approach. The write latency doesn't vary significantly with the size of writes because the only operation dependent on the size of the write is the hashing of data; signatures are created by encrypting the hash of the data.

**Apply latency:** Next we compare the apply latency of PRACTI, PRACTI+Signatures, ASTRO-100%, ASTRO-90%, ASTRO-50%, and ASTRO-10%. ASTRO-X% refers to a workload where the reader is interested in only X% of the all updates. Since write latencies did not show significant variation with object size, we limit ourselves to 1 byte objects. Figure 6(b) compares the average apply latencies. The variation of apply latency

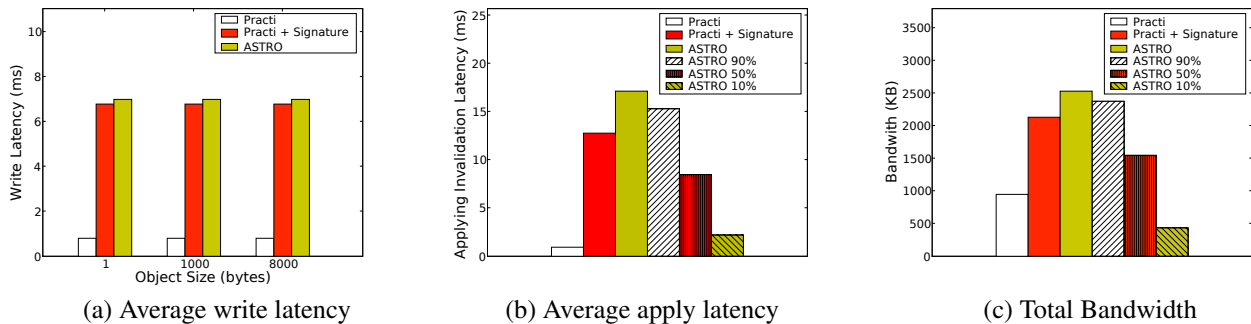


Fig. 6: Average write latency (a), apply latency (b) and total bandwidth (c) for synchronizing 10000 writes.  $x\%$  ASTRO is ASTRO in which reader is only interested in  $x\%$  of the written data. ASTRO, PRACTI and PRACTI+Sign correspond to 100% interest set.

shows similar behavior to write latency for PRACTI, PRACTI+Signatures, and ASTRO. The ASTRO- $X\%$  settings with varying interest sets illustrate the benefit of partial information. As the fraction of writes in which the reader is interested falls, the average apply latency per logical write goes down. However, the decrease is not linear because ASTRO sends summarized updates of logarithmic sizes to ensure consistency. Nevertheless, the reduction is significant when a node’s interest set falls from 100% (ASTRO) to 10% (ASTRO-10%).

**Bandwidth:** This experiment compares the bandwidth of different configurations in a setup similar to previous experiments. As indicated in Figure 6(c), ASTRO’s bandwidth consumption is about 19% more than the signed PRACTI approach for one-byte writes. Our protocol incurs the additional overhead because of the additional ICHS field that ASTRO sends. As before, the bandwidth consumption of our system falls significantly as the fraction of updates in which the receiver is interested decreases. Also note that the relative difference of the two approaches would shrink for objects larger than the 1-byte objects examined here.

## 6.2 Trace Experiments

We evaluate ASTRO using a connectivity trace indicating the mobility patterns of 100 users [14]. The trace contains the record of call log, cell tower accessibility, and device proximity of each cell phone user. The cell tower accessibility indicates which tower was accessible at any point in time, and the device proximity information indicates when two such cellphone devices came in reachable range to each other.

The relevant aspect of this trace is the connectivity information of different users, which reflects a possible communication pattern in a real setting. We generate the workload by canonically mapping calls in the trace to new writes in ASTRO and the currently reachable cell tower id to the *old* of writes by users. The cell-tower id gives a useful localized workload. We generated the sync operations based on the connectivity information in the trace. In a sync, one node connects to another node and fetches all the new updates sender has obtained since the

	Checkpoint	Log	Log w/o IDVV	SVV
BW (MB)	1.00	18.96	132.04	14.74
Latency (ms)	83	3,644	4,4778	4,050

Fig. 7: Comparison of ASTRO using log-exchange (with and without IDVV), ASTRO using checkpoint and SignedVV approach

last sync.

For this experiment, we use a subset of the trace spanning 31878 updates and 286 synchronizations sessions among 100 nodes. In each synchronization session, one user fetches all new updates from the other user.

**Comparison with Signed Version Vector:** In the first experiment, we use a Signed Version Vector (SVV) approach [35] as our baseline. In the SVV approach, the writer signs its component of version vector and uses the signed components obtained from the other writes it has observed. Receiver verifies that each component of a version vector is signed by its respective node, thereby preventing faulty nodes from reordering updates created by correct replicas. We apply the IDVV optimization to the SVV approach since directly using the SVV approach would be extremely inefficient due to prohibitive signing, verification, and bandwidth overhead; SVV requires separately signing, verifying and transporting all entries of the version vector. The IDVV optimization excludes the components of DVV that are covered by previous entries.

Table 7 shows the total bandwidth consumed in three settings: Signed Version Vector, ASTRO without the IDVV optimization, and ASTRO. This experiment illustrates two points; first, that the IDVV optimization in ASTRO enables order of magnitude improvement in network efficiency for realistic connectivity scenarios; second, ASTRO performs as well as the simple SVV approach while providing stronger properties. Table 7 shows similar results regarding the total synchronization time of these three systems.

**Benefit of Checkpoints:** This experiment illustrates the benefit of using checkpoints based on partial information in ASTRO, which is not possible in other systems that support autonomy. We use the bandwidth and synchrono-

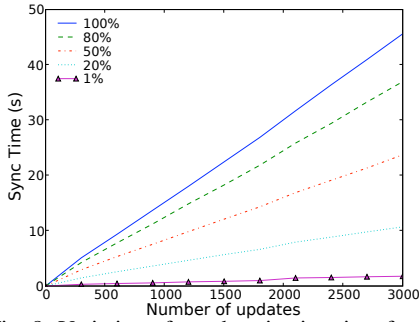


Fig. 8: Variation of synchronization time for different selective sharing settings.

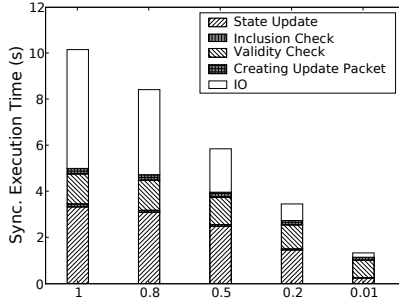


Fig. 9: Breakdown of synchronization cost excluding the signature verification cost

nization time as our performance metrics and compare the performance of ASTRO with log, where all updates are exchanged using the log exchange, with ASTRO with checkpoint, where a checkpoint summarizes all but last update to each object as summaries.

As indicated in Table 7, the checkpoint exchange protocol reduces the bandwidth consumption by 95% and the synchronization overhead by 97% compared to the systems that require receiving all updates.

### 6.3 Partial Information

In this section, we evaluate the benefits and limitations of partial information. We evaluate an artificial setting comprising of 10 nodes and compared the two settings: nodes are interested in all the data, nodes are interested in 10% of the data. We did random updates and periodically synced two random nodes. With an update/sync ratio of 10 updates per sync, we observed a reduction in overall bandwidth consumption from 12 MB in full information to 8.7 MB in partial information setting. In the same setting, we observed that even with partial information, only 11 synchronizations over a period of 10000 synchronizations had to rely on the external server to provide smaller summaries to avoid partial overlap.

### 6.4 Synchronization time

Figure 8 shows the variation of synchronization time as a function of number of updates for different selective sharing settings. As expected, the cost of synchronizing increases linearly with the number of writes. In addition,

selective sharing reduces the synchronization overheads. Note that a significant fraction of the synchronization latency comes from the encryption overhead.

Figure 9 shows the breakdown of the synchronization latency across different steps: 1) state update for updating the local state, LPVV and summary logs, 2) the cost of performing the inclusion check, 3) cost of validity check, 4) cost of constructing a packet containing all the summaries that the sender needs after receiving his LPVV, and 6) the IO cost. As expected, the costs reduce with the reduction in the fraction of interested writes. Note that we intentionally omitted the latency due to the encryption operations to better highlight overheads of our system. The breakdown indicates that the security component adds modest overheads in comparison to others.

## 7 Related Work

We have already discussed in the Introduction how Astro relates to systems that support opportunistic data sharing like Bayou [33], PRACTI [8], and Cimbiosys [34] as well as systems, such as SUNDR [27] and Efficient Fork Linearizable Shared Memory (EFL) [10] that provide data services through untrusted servers.

Zeno [37] recently introduced the concept of eventually consistent state machine replication that allows availability despite partitions. However, Zeno requires  $f + 1$  available servers, making it unsuitable for autonomous systems.

In his dissertation [19], Kang defines a causal graph in which each update contains a hash of the update’s causal predecessors and of the contents of the update to determine dominance relations between updates to a single object. Spreitzer et al. [38] provide a design for dealing with server corruption in Bayou’s server replication protocol that relies on audits and assumes full replication. These approaches provide guarantees weaker than fork-causal consistency and require complete information thereby limiting their usefulness in autonomous data sharing settings. Reiter and Gong [35] describe approaches to identify causal ordering of updates in distributed settings but their approaches don’t address omission or forking attacks.

TimeWeave [24] allows the secure history of events happening at one node to be entangled with the history at another node thereby ensuring that a tamper-evident ordering of two events from two different histories can be established. However, TimeWeave does not prevent a faulty node from exposing two different histories to two different nodes and does not provide fork-causal consistency.

PeerReview [16] provides a way to check that nodes in a distributed protocol follow the protocol’s specification by requiring them to include a signed statement about all messages they receive in all messages they send. PeerReview is not well-suited for autonomous settings as it

relies on (1) presence of a quorum of witnesses out of which at least one must be correct to ensure the correctness properties and (2) the availability of complete information at these witnesses.

## 8 Conclusion

ASTRO is the first system to support disconnected operation and opportunistic data sharing among potentially Byzantine nodes while continuing to provide precise and useful consistency guarantees to correct nodes. Specifically, ASTRO supports *fork-causal consistency*, the strongest consistency semantics that supports autonomous computations and guarantees that Byzantine nodes cannot alter the causal ordering of updates generated by correct nodes.

## References

- [1] Kazaa. <http://www.kazaa.com>, 2002.
- [2] Wizzy project. <http://www.wizzy.org.za>, 2005.
- [3] All things distributed. [http://www.allthingsdistributed.com/2007/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2007/12/eventually_consistent.html), 2007.
- [4] Live mesh. <https://www.mesh.com>, 2008.
- [5] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. OSDI*, Dec. 2002.
- [6] A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *PODC*, Aug. 1997.
- [7] M. Ahamad, P. Hutto, and R. John. Implementing and programming causal distributed shared memory. May 1991.
- [8] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI 06*.
- [9] K. Birman and R. Cooper. The ISIS Project: Real Experience with a Fault Tolerant Programming System. In *European Workshop on Fault-Tolerance in OS*, pages 103–107, 1990.
- [10] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *PODC*, 2007.
- [11] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI*, Feb. 1999.
- [12] M. Corporation. About active directory domain services. [http://msdn.microsoft.com/en-us/library/aa772142\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa772142(VS.85).aspx).
- [13] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed storage system for challenged networks in developing regions, Feb. 2008.
- [14] N. Eagle and A. S. Pentland. CRAWDAD data set mit/reality. <http://crawdad.cs.dartmouth.edu/mit/reality>, 2005.
- [15] S. Gilbert and N. Lynch. Brewer’s conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News*, 33(2), Jun 2002.
- [16] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.
- [17] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Sys.*, 12(3), 1990.
- [18] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *ICDCS*, pages 302–311, 1990.
- [19] B. Kang. *S2D2: A framework for scalable and secure optimistic replication*. PhD thesis, UC Berkeley, Oct. 2004.
- [20] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM TOCS*, 10(1):3–25, Feb. 1992.
- [21] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton, 1988.
- [22] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pages 203–213, 1998.
- [23] D. Malkhi and D. Terry. Concise version vectors in WinFS. In *Symp. on Distr. Comp. (DISC)*, 2005.
- [24] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford, 2003.
- [25] P. Marshall. The disruption tolerant networking program. [www.darpa.mil/sto/solicitations/DTN/briefs.htm](http://www.darpa.mil/sto/solicitations/DTN/briefs.htm), 2005.
- [26] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *In SOSP 09*.
- [27] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC*, 2002.
- [28] R. C. Merkle. Protocols for public key cryptosystems. *sp*, 00:122, 1980.
- [29] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. OSDI*, Dec. 2002.
- [30] L. Novik, I. Hudis, D. Terry, S. Anand, V. Jhaveri, A. Shah, and Y. Wu. Peer-to-peer replication in winfs.
- [31] D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *Proc. OSDI*, 2006.
- [32] A. S. Pentland, R. Fletcher, and A. Hasson. Daknet: Rethinking connectivity in developing nations. *Computer*, 2004.
- [33] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*, Oct. 1997.
- [34] V. Ramasubramanian, T. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI*, 2009.
- [35] M. Reiter and L. Gong. Securing causal relationships in distributed systems. *The Computer Journal*, 38(8):633–642, 1995.
- [36] M. Schroeder, A. Birrell, and R. Needham. Experience With Grapevine: The Growth of a Distributed System. *ACM TOCS*, 1984.
- [37] A. Singh, P. F. P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine fault tolerance. In *NSDI*, 2009.
- [38] M. Spreitzer, M. Theimer, K. Petersen, A. Demers, and D. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *Proc. MOBICOM*, 1997.
- [39] J. Taylor, S. Gillette, R. Hodgson, J. Downing, M. Burns, D. Chavez, and J. Hogan. Informing the network: Improving communication with interface communities during wildland fire. *Human Ecology Review*, 14(2):198–211, 2007.
- [40] B. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Internat. Conf. on Parallel and Distributed Information Systems*, pages 140–149, Sept. 1994.