PRACTI Replication for Large-Scale Systems

Department of Computer Sciences Technical Report 04-28
Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng
University of Texas at Austin

Abstract

Many replication mechanisms for large scale distributed systems exist, but they require a designer to compromise a system's replication policy (e.g., by requiring full replication of all data to all nodes), consistency policy (e.g., by supporting per-object coherence but not multi-object consistency), or topology policy (e.g., by assuming a hierarchical organization of nodes.) In this paper, we present the first PRACTI (Partial Replication, Arbitrary Consistency, and Topology Independence) mechanisms for replication in large scale systems. These new mechanisms allow construction of systems that replicate or cache any data on any node, that provide a broad range of consistency and coherence guarantees, and that permit any node to communicate with any other node at any time. Our evaluation of a prototype suggests that by disentangling mechanism from policy, PRACTI replication enables better trade-offs for system designers than possible with existing mechanisms. For example, for one workload we study, PRACTI's partial replication reduces bandwidth requirements by over an order of magnitude compared to full replication for nodes that only care about a subset of the system's data.

1 Introduction

Data replication is a fundamental technique for improving the performance [3, 11, 14, 30, 33, 39], availability [7, 14, 22, 43], ubiquity [21, 31], persistence [26], and managability [1] of a broad range of large-scale distributed systems such as personal file systems [21, 31], web service replication systems [11, 14, 39], global-scale file systems [9, 43, 33], or enterprise data distribution systems [1]. Because no replication system can have perfect performance properties [25] or perfect availability and consistency [5], systems designed for different environments make different trade-offs among these factors by implementing different consistency policies, placement policies, and topology policies. Informally, *consistency policies* such as sequential [24] or causal [20] regulate how quickly newly written data are seen by reads, *placement policies* such as demand-caching [19, 30], prefetching [16], push-caching [18] or replicate-all [31] define which nodes store local copies of which data, and *topology policies* such as client-server [19, 30], hierarchy [4, 27], or ad-hoc [17, 22, 31] define the paths along which communication flows.

Unfortunately, existing replication mechanisms are entangled with specific policy assumptions. For example, Bayou [31] allows arbitrary topologies for communication among nodes but fundamentally assumes a policy of full replication where all nodes store all data from any volume they export. Conversely, Coda's [21] replication policy allows nodes to cache subsets of data, but Coda fundamentally assumes a restrictive client-server communication topology.

This paper describes a set of mechanisms that for the first time simultaneously provide all three PRACTI (Partial Replication, Arbitrary Consistency, and Topology Independence) properties. *Partial replication* means that a system can place any subset of data on any node. In contrast, some replication systems require a node to maintain full copies of all objects in all volumes they export [28, 31, 44]. *Arbitrary consistency* means that the system provides flexible semantic guarantees, including the ability to selectively enforce both consistency guarantees (which constrain the order that updates across multiple objects become observable to readers) and coherence guarantees (which constrain the order that updates to a single object become observable but do not additionally constrain the ordering of updates across different objects.) In contrast, some replication systems can only enforce coherence guarantees but make no guarantees about consistency [17, 33]. *Topology independence* means that any node can communicate with any other node. In contrast, many systems restrict communication to client-server [19, 21, 30] or hierarchical [4, 42] patterns.

We base the PRACTI protocols on Bayou's log exchange mechanisms [31], which support a range of consistency guarantees [44] and topology independence, but which fundamentally assume full replication in order to maintain the invariant that each node's log represents a causally-consistent prefix of the system's writes. We adapt this protocol to support partial replication using two principles. First, we *separate the control path from the data path* by separating invalidation messages that identify what has changed from body messages that encode the changes to the contents of files [2, 33]. In contrast with Bayou's protocol that assumes that invalidations and bodies go hand-in-hand, these modifications require us to introduce new synchronization rules to enforce ordering restrictions, mechanisms for handling demand read misses, and protocols for enforcing policies on the minimum safe degree of data replication [33]. Second, we use *imprecise invalidations*, which allow a single invalidation to conservatively summarize a set of omitted invalidations. We define a protocol that allows nodes to compose precise invalidations into imprecise ones, to incrementally exchange logs of mixed precise and imprecise invalidations, to allow precise reads (that see a consistent view of the data) or imprecise reads (that see only a *coherent* view of the data), and to recover precision for an interest set that has become imprecise.

Because PRACTI mechanisms support a broad range of replication, topology, and consistency policies, we design our prototype as a "replication microkernel" that carefully separates mechanism from policy. Replication *cores* communicate with one another using an asynchronous communication protocol, and each core uses the PRACTI mechanisms to enforce a node's safety properties regardless of what messages other nodes sent to it. A separate *controller* layer implements the system's policies and provides liveness by triggering communication between nodes. We implement several flavors of controller including a novel one that uses SDIMS (a DHT-based Scalable Distributed Information Management System) [40] for a number of purposes including locating data on read misses and forming per-interest-set spanning trees to propagate data to interested nodes.

We have constructed a prototype system and we evaluate it using microbenchmarks. Our primary conclusion is that *by disentangling mechanism from policy, PRACTI replication enables better trade-offs for system designers than possible with existing mechanisms.* For example, PRACTI makes it possible to build a system that provides causal consistency and that—like Bayou—allows any node to exchange updates with any other node and that—like Coda—allows each node to store and see updates for only the data about which it cares. For one workload we study, PRACTI's partial replication reduces bandwidth requirements by an order of magnitude compared to a full replication for nodes that only care about a subset of the system's data, and PRACTI's topology independence reduces synchronization latency by over a factor of three and enable synchronization in scenarios where it would otherwise be impossible compared to a restricted-topology, central server system for mobile clients that are weakly connected to main server. Finally, we find that imprecise invalidations are effective at limiting the additional cost of providing consistency over the cost of providing coherence.

More broadly, we envision PRACTI as a step towards a "Unified Replication Architecture" toolkit that will simplify the development and deployment of large-scale replication systems. Because current mechanisms and policies are entangled, when a replication system is built for a new environment, it must often be built from scratch or must modify existing mechanisms to accommodate new policy trade-offs. PRACTI may help define a common substrate over which a broad range of replication systems can be constructed. Note, however, that although the current system provides a great deal of flexibility, it does fall short of our eventual goal of providing a unified replication architecture in two significant ways. First, although our current system supports a wide range of consistency options—including causal coherence, eventual coherence, causal consistency, eventual consistency, and acknowledged writes—there are some limitations on this flexibility. As we discuss in Section 2.4, several enhancements appear to be relatively straightforward extensions given our current mechanisms; these extensions include application-specific conflict detection and resolution [35] and tunable quantitative limits on inconsistency [44]. Still, we have not precisely quantified the boundaries of what semantics can be conveniently accommodated within PRACTI's "arbitrary" consistency. Second, we do not yet accommodate some families of replication techniques, such as quorums for replication, callback state for coordinating communications among nodes [19, 30], and leases for limiting staleness [15], though we eventually hope integrate such techniques within a common framework.

This paper makes two contributions. First, it describes novel mechanisms that support efficient and scalable PRACTI replication. To our knowledge past systems have provided two, but never all three, of the PRACTI properties. Second, it provides a prototype replication toolkit based on PRACTI that cleanly separates mechanism from policy and that allows nearly arbitrary replication, consistency, and topology policies.

The rest of this paper is organized as follows. Section 2 describes the design of the PRACTI mechanisms, and Section 3 details our prototype of the core (mechanisms/safety) and controller (policies/liveness). Section 4 experimentally evaluates the design. Finally, Section 5 surveys related work and Section 6 highlights our conclusions.

2 PRACTI design

This section describes the key ideas required to provide scalable PRACTI replication. The basic idea is simple. As Section 2.1 describes, we begin with a basic log exchange protocol similar to that used in Bayou [31]. Then, we modify the protocol to separate the control path from the data path by separating invalidations from update bodies as described in Section 2.2; this separation allows us to avoid sending all body updates to all nodes and to avoid storing all bodies at all nodes. Third, we use *imprecise invalidations* to avoid full replication of consistency messages and state as described in Section 2.3. Fourth, we extend the interface over these basic mechanisms in order to support strengthening or weakening of the consistency semantics as described in Section 2.4.

2.1 Background: Log exchange

Our protocol extends Bayou's log exchange protocol [31]. In order to clarify our terminology and differences between our protocol and Bayou's, we review the basic protocol here.

When a node issues a write, it assigns the write an *accept stamp* comprising the node's ID and a logical clock value. The logical clock is a Lamport clock [23] that is advanced on each local operation and which, upon communication with another node, is advanced to exceed the maximum of the local and remote nodes' logical clocks. A node maintains a checkpoint representing all writes up to a time represented by a version vector $cpVV$, where $cpVV_\alpha$ holds the highest accept stamp from node α reflected in the checkpoint. Additionally, a node maintains a log of all writes it has seen since the checkpoint sorted by the writes' accept stamps (using the logical clock as the primary key and the node ID to break ties) as well as a version vector $currentVV$ that indicates the highest per-node accept stamps in the log.

At Bayou's core are three properties. First, the *prefix property* is the invariant that a node's state always reflects a prefix of the sequence of writes by all nodes in the system: if a node β has $currentVV_\alpha = t$, then β's state reflects all writes by α up to and including the write at logical time t. Second, each node's local state always reflects *causally consistent* [20] view of all writes that have occurred. This property follows from the prefix property and from the use of Lamport clocks to ensure that once a node has observed a write w, all of its subsequent writes' accept stamps will exceed w's. Third, the system ensures eventual consistency—all connected nodes will eventually agree on the same total order of all writes.

Bayou's log exchange protocol[1] enforces these properties. If β would like α to send it a stream of updates, β sends α its current version vector $currentVV$. Then, α connects to β and sends a sequence of messages: $\{startVV, w_1, w_2, \ldots\}$. When β receives such an incoming stream, it rejects the stream if any element of the stream's $startVV$ exceeds $currentVV$. Otherwise it processes each write w_i by inserting w_i into its sorted log, updating $currentVV$ and its local Lamport clock. In order to support fast local reads, each node also maintains a snapshot *store* of the per-object state at time $currentVV$. $Store_{objId}$ contains two fields: $accept$, the accept stamp of the latest write to $objId$, and $body$, the value of that write. When processing w_i, if ($w_i.accept > store_{w_i.objId}.accept$) then update $store_{w_i.objId}.body = w_i.body$.

Note that the simple protocol described here omits several features. Most notably, in Bayou, writes are more general queries that can affect multiple different objects and that carry with them references to application specific conflict detection and resolution routines [35]. Furthermore, Bayou implements a primary-commit protocol to establish a final order on a prefix of writes despite uncommunicative nodes. We discuss both of these issues when we address flexible consistency in Section 2.4.

Overall, the Bayou protocol provides several attractive features. It provides *topology independence* in that any node can exchange updates with any other node at any time. And, it provides the relatively strong consistency gurantees of causal consistency and eventual consistency which are stronger guarantees than just providing coherence. These stronger consistency guarantees are essential for ensuring that Bayou's application-specific detection and resolution procedures eventually agree on the same total order on all writes and therefore eventually converge on the same state: given the power of Bayou's conflict resolution mechanisms, even with coherence of each individual object any difference in the order that writes to different objects are observed could cause a "butterfly effect" where the state at different nodes arbitrarily diverge.

2.2 Separate invalidations from update bodies

In order to add partial replication to the log exchange protocol's topology independence and flexible consistency, we first separate the control path from the data path by separating invalidation messages from update messages. This separation allows update bodies to be sent to and stored at arbitrary subsets of the nodes according to the system's data replication policy.

Invalidation messages contain two fields: *objId*, which identifies the modified object, and *accept*, which is the accept stamp assigned by the writer when the write occurs.[2] A node's local state includes a log (sorted by accept stamp) and a per-object store representing the cur-

[1]We describe our extension of Bayou's log exchange protocol that supports either the *batch-mode* log exchange in the original Bayou, in which a batch of updates is atomically applied to a node's local state, or a *streaming* log exchange in which one node sends another a sequence of updates, each of which is individually applied.

[2]For simplicity, we describe the protocol in terms of full-object writes. In practice we track writes on the granularity of arbitrary byte ranges: Invalidation and body messages contain an *offset* and *length* field in addition to the fields discussed here, and our per-object state contains per-byte-range *accept*, *valid*, and *body* fields.

rent state of each object for reads. $Store_{objId}$ contains three fields: *accept*, *valid*, and *body*. Finally, each node maintains a *currentVV* version vector and a *currentAccept* Lamport clock.

Invalidation log exchange. When a node receives a stream of updates $\{startVV,\ w_1,\ w_2,\ \ldots\}$, it rejects the stream if $startVV_\alpha > currentVV_\alpha$ for any node α. Otherwise, it processes each w_i by inserting the write into its sorted log and updating the store as follows:

```
if $w_i.accept > store_{w_i.objId}.accept$ then
    $store_{w_i.objId}.valid = INVALID$
    $store_{w_i.objId}.accept = w_i.accept$
```

The node also updates its *currentVV* and *currentAccept.*

Applying bodies. Although invalidations must be sent in causal, sequence number order, PRACTI supports distribution of bodies according to arbitrary policies, in arbitrary order, across arbitrary topologies. A PRACTI node must therefore synchronize arriving bodies with the invalidation streams before applying them to its local state. For correctness, PRACTI maintains the invariant that update bodies are not applied until the corresponding invalidation message has been. To ensure this invarient, nodes maintain a *pendingUpdate* list of updates that have been received but not yet applied to the local state, and they sort this list by accept stamp to put the earliest-numbered update at the head of the queue. When a body message b is at the head of the pending update queue, the node waits until $store_{b.objId}.accept \geq b.accept$ and then (a) if $store_{b.objId}.accept == b.accept$, applies the update by setting the *body* field of that object's checkpoint state to $b.body$ and setting the *valid* field to *VALID* or (b) if $store_{b.objId}.accept > b.accept$, discards b.

Systems can use whatever replication policy they want for bodies from demand caching to client-driven prefetching [16] to replicate-all [31] to server-driven push [18, 39] to globally-optimized placement [38] to pushing updates of whatever objects have been fetched on demand [33]. A policy we advocate is having a sender enqueue update bodies in a local priority queue sorted by update priority which drains to the receiver's *pendingUpdate* list via TCP-Nice background network transfers [37].

Also notice that although causal consistency restricts the order in which a node applies incoming invalidations, nodes can use different policies to delay application of invalidations in order to improve read latency or availability [29]. In particular, when a node α receives an invalidation I, rather than immediately apply I to its local state, α can wait to apply I until either (a) the corresponding update body arrives or (b) a maximum delay expires. As we demonstrate in a recent study [29], waiting until an update arrives before applying an invalidation can increase the fraction of objects stored locally in the VALID state and thereby improves both read latency and system availability. The maximum delay parameter allows nodes to limit worst case staleness while pursuing these benefits.

Demand reads. The system ensures the safety property of providing a causally consistent view of data by having a local read request block until the requested object's status is *VALID*. To ensure liveness, when an *INVALID* object is read, an implementation should arrange for someone to send the body. PRACTI supports any policy for doing this from a static hierarchy (i.e., ask your parent [4] or a central server [19] for the missing data) to a separate, centralized location-metadata directory [13], to a DHT-based location-metadata directory [36], to a hint-based search strategy [34], to a push-all strategy [31] (i.e., "just wait and the data will come.")

Reliability. Separating invalidations from updates enables partial replication but also raises the issue of reliability: in Bayou, all nodes have copies of all data, but a PRACTI system will need to enforce an explicit policy decision about the minimum acceptable level of replication so that the loss of a node or a local cache replacement decision does not render some data unavailable or the storage system unreliable. We provide a simple, low-level mechanism that supports a broad range of high-level policies from maintaining a fixed number of "gold" copies of each object [9, 33] to propagating all data to a well-provisioned central server [19] or replicated server "core" [21] to Bayou's strategy of replicating everything to everyone: a PRACTI invalidation message can be of one of two types—an *unbound* invalidation as described above or a *bound* invalidation that contains, in addition to the fields listed above, a *body* field that contains the body of the write that created the bound invalidation. When a write is created, its invalidation is initially bound. An *unbind message* contains an accept stamp and is propagated through the system using a flooding strategy: when a node receives an unbind message, it checks to see if it has the corresponding bound invalidation in its local log; if so, it (1) converts that invalidation to be unbound and (2) propagates the unbind message to all neighbors with whom it is currently connected. If the node either has not seen the corresponding invalidation or already has it in the unbound state, it does nothing.

In our implementation unbind propagation is best-effort—if the connection topology changes between when a write occurs and when it is unbound, some nodes may not see the unbind and continue to propagate the invalidation in the bound state for longer than necessary. But, because this situation should be rare and hurts performance rather than correctness, we have elected not to include a more heavy-weight mechanism for reliably propagating unbind messages with the logs. Conversely, integrating the propagation of bound invalidations with the log is a conscious choice. By integrating our mechanism for ensuring reliability to the log exchange, we tie reliability to the causal order guarantees: any write in a node's log depends only on (a) explicitly unbound writes (judged safe by some higher level policy) or (b) bound writes in that node's log (which are as good as safe due to fate sharing).

To help the reliability algorithm decide when it is safe to unbind a write, each node provides an interface *sync(replyTo, acceptStamp)*, which is an asynchronous request that asks the node to send a message to *replyTo* after the node has stored the invalidation corresponding to *acceptStamp* in its persistent redo log. In addition, for convenience, when a node β receives a bound invalidation for a write originally issued by node α, β sends α a sync-reply message regardless of whether β has received a *sync* request for that write. A policy controller can implement, for example, a k-copy policy by issuing sync requests to various nodes when a write occurs and then, when it receives k replies, issuing an unbind request to the local node (which will flood the unbind to its neighbors).

Analysis. Separating invalidations from bodies retains the topology independence and causal consistency of log exchange protocols, but it allows arbitrary policies to control the replication of bodies. Note, however, that all nodes must still see all invalidations.

2.3 Imprecise invalidations

Imprecise invalidations allow a node to omit details from logs that it sends while still allowing receivers to enforce causal consistency. Imprecise invalidations work by (1) replacing invalidation messages with a *consevative summary* of them and (2) maintaining per-node data structures that track which objects are safe to access.

Invalidation log exchange. An imprecise invalidation contains three fields: *start* and *end*, which are arrays of accept stamps, and *target*, which describes the objects affected by the invalidation. For every node α that has one or more writes summarized by an imprecise invalidation, $start_\alpha$'s value is at most the earliest summarized accept stamp and end_α's value is at least the latest summarized accept stamp. *Target* may encode covered objects in any manner, as long as it is conservative and allows the receiver of the invalidation to identify all objects affected by the summarized writes. Our implementation encodes *target* as a list of directory paths where each path represents either an individual file or directory (e.g., /foo/bar) or a subtree (e.g., /flim/flam/*). Note that a precise invalidation is a special case of an imprecise invalidation with a single writer, *start* = *end*, and a single object as a *target*. We use the term *general invalidation* to refer to either a precise or imprecise invalidation. A system forms an imprecise invalidation using the union operation on two general invalidations: $gi_U = gi_1 \cup gi_2$ has *start* and *end* arrays with entries for every server in either gi_1 or gi_2's *start* and *end* with $gi_U.start_\alpha = min(gi_1.start_\alpha, gi_2.start_\alpha)$, $gi_U.end_\alpha = max(gi_1.end_\alpha, gi_2.end_\alpha)$, and $gi_U.target$ encompassing all objects encompassed by gi_1 and gi_2's *targets*.

A node groups system data into *interest sets* and tracks whether an interest set is *precise*—meaning that the node's local state reflects all invalidations overlapping the interest set—or *imprecise*—meaning that the node's local state is not causally consistent for that interest set due to one or more overlapping imprecise invalidations.

Algorithm 1 summarizes how a node processes a stream s of general invalidations $\{startVV$, gi_1, gi_2, $\ldots\}$ against one interest set of data. For each such interest set IS, the node maintains the interest set membership, the last precise version vector $lpVV$ that represents the highest version vector for which all precise invalidations have been applied to IS and the current version vector cVV that represents the highest version vector for which a general invalidation has been applied to IS.

Algorithm 1 ProcessInvalStream $s = \{startVV$, gi_1, $gi_2, \ldots\}$ for interest set IS

```
$startVV = s$.next()
if $\exists\alpha \mid startVV_\alpha > currentVV_\alpha$ then
  return; // Reject stream that does not preserve prefix property
$pending$ = new Set()
$gi = s$.next()
while $(gi \neq null)$ do
  $\forall\alpha : nextStartVV_\alpha = MAX(startVV_\alpha, gi.start_\alpha)$
  if !$(\exists p \in pending \mid (\forall\alpha : p.end_\alpha \leq nextStartVV_\alpha))$ then
    // Apply overlapping gi from s at start time
    $log$.insert($gi$, $startVV$)
    if $gi.target$ intersects $IS$ then
      if $gi.isPrecise()$ AND $\forall\alpha : lpVV_\alpha \geq startVV_\alpha$ then
        // If no gaps to this precise inval, update lpVV
        $\forall\alpha : lpVV_\alpha =$MAX$(lpVV_\alpha, gi.start_\alpha)$
      $\forall\alpha : currentVV_\alpha =$ MAX$(currentVV_\alpha, gi.end_\alpha)$
      $startVV = nextStartVV$
    if $gi.isPrecise()$ then
      $store_{gi.objId}$.update$(gi.start, INVALID)$
    $pending$.insert$(gi)$
    $gi = s$.next()
  else // Apply non-overlapping p from pending at end time
    if !$(p.target$ intersects $IS)$ then
      if $\forall\alpha : lpVV_\alpha \geq start_V V\alpha$ then
        $\forall\alpha : lpVV_\alpha =$MAX$(lpVV_\alpha, p.endVV_\alpha)$
      $\forall\alpha : currentVV_\alpha =$MAX$(currentVV_\alpha, p.endVV_\alpha)$
    $pending$.remove$(p)$
```

We rely on the prefix property for reasoning about messages in a stream. In particular, a stream that begins with *startVV* guarantees that the subsequent invalidations represent a causally consistent sequence with no omissions starting from *startVV*. To support incremental application in which multiple instances of Algorithm 1 execute concurrently and interleave their application of invalidations, our algorithm updates a per-stream, per-interest set *startVV* after processing each invalidation. For simplicity the pseudo-code shows a single interest set version of our protocol; see the appendix the full version.

A node applies general invalidations from a stream to an interest set in sorted order based on their timestamps, but they are handled differently depending on whether they overlap an interest set or not. If an invalidation overlaps an interest set, it is applied at its *start time* as it arrives from the stream, but if it does not overlap, it is buffered until its *end time* is guaranteed not to be causally dependent on any remaining start time in the stream, which happens when its end time is at most $nextStartVV$, the $startVV$ value that will hold after the next invalidation is processed at its start time.

At gi's start time, we first insert it into the sorted log of all invalidations and update the local random access store. Then, if the invalidation overlaps IS, we advance $currentVV$ to the end time of the invalidation (indicating that the data in IS must reflect invalidations up to *end* in order to be considered current). We advance *lpVV* for the interest only if (a) *startVV* is at most the current *lpVV* for IS (i.e., there is no missing precise invalidation before gi) and (b) this general invalidation is, in fact, a precise invalidation (i.e., gi does not introducing a missing precise invalidation.) Finally, if the invalidation overlaps the interest set, we advance *startVV* for the interest set; if the invalidation is precise, we update the per object state in the same way as described in Section 2.2.

Conversely, if an invalidation does not overlap an interest set, it could safely be ignored since it carries no invalidations that could make the interest set imprecise. But, the very fact that the invalidation does not intersect the interest set is useful—it shows that there was a period of time over which no invalidations (precise or imprecise) intersected the interest set; this information can help disambiguate other general invalidations that overlap the interest set and this one in time. Therefore, at gi's end time, if the invalidation target does not overlap IS, (1) advance IS's *currentVV* to $gi.end$ and (2) if $startVV$ is at most IS's *lpVV*, update *lpVV* so that all elements are at least as great as $gi.end$.

Log update. Our desire to support both partial replication and arbitrary topologies complicates log updates. Simply inserting each received invalidation gi into the local log in sorted order would not be sufficient because interpreting a general invalidation is done in the context of the stream in which it is received. In particular, gi is interpreted based on the per-stream $startVV$ which indicates that no causally required invalidations are missing between $startVV$ and $gi.start$. In order to avoid losing this valuable information, when a node inserts gi into its log, it first decomposes gi into per-writer general invalidations; it then uses *gap filling* and *intersection* operations to encode this "no missing invalidations" information.

Decomposing gi into per-writer general invalidations gi_α is simple: for each server α in $gi.start$, generate gi_α with $gi.start_\alpha$, $gi.end_\alpha$, and $gi.target$.

For the gap filling operation, each per-writer log maintains the invarient that there is no gap between the end time of an element and the start time of the next element. When a node inserts gi_α into its per-writer log for α at $startVV_\alpha$, if gi is newer than the newest element in the log, it fills any gap between gi and existing element by inserting a new gap-filling invalidation with a start stamp one larger than the highest existing end stamp, and end stamp one smaller than gi's start, and an empty target.

For the intersection operation, we maintain the invariant that there is at most one invalidation that covers any moment in time in a per-writer log. We intersect two general invalidations gi_1 and gi_2 by replacing them with up to three general invalidations: the first covers the time from the earlier start to the later start and targets the objects targeted by the earlier start; the second covers the time from the later start to the earlier end and covers targets represented by the intersection of gi_1 and gi_2's targets; and the third covers the time from the earlier end to the later end and covers the targets of the later end.

When we send a stream of invalidations to another node, we discard gap-filling invalidations and we combine per-writer invalidations into multi-writer invalidations using the policy described in Section 3.1.

Demand reads. When a demand read occurs, it blocks until the interest set it targets becomes precise. This blocking ensures the safety property that reads always observe a causally consistent view. In Section 2.4 we describe how a reader can relax these guarantees. As with reads of invalidated objects, a system can use any policy for selecting one or more nodes to which to connect in order to retrieve the precise invalidations needed to make an interest set precise.

Example. Figure 1 illustrates these mechanisms in action. Node α writes objects a, b, and c; node β cares about object a and receives from α precise invalidations about a and imprecise invalidations about b and c. Node γ cares about object c and receives from α precise invalidations about c and imprecise invalidations about a and b. Finally, node δ cares about a and c and receives from β precise invalidations about a (but imprecise invalidations about b and c due to β's imprecision) and from γ precise invalidations about c (but imprecise invalidations about a and b.) First, α sends a stream of invalida-

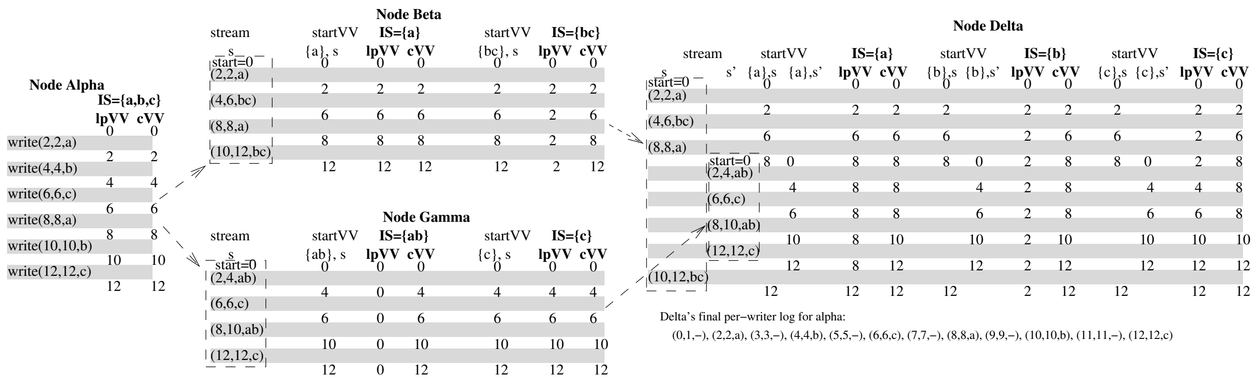
Node Alpha
IS={a,b,c}
lpVV cVV
Node Beta
IS={a}
IS={bc}
Node Gamma
IS={ab}
IS={c}
Node Delta
IS={a}
IS={b}
IS={c}
stream
startVV
Delta's final per−writer log for alpha:
(0,1,–), (2,2,a), (3,3,–), (4,4,b), (5,5,–), (6,6,c), (7,7,–), (8,8,a), (9,9,–), (10,10,b), (11,11,–), (12,12,c)

Fig. 1: Illustration of imprecise invalidation mechanisms in *split-join* scenario. Nodes α, β, γ, and δ share objects a, b, and c. At each node, we show the per-interest-set information (last precise version vector $lpVV$ and current version vector cVV), the per-invalidation-stream information ($startVV$ and a series of generalized invalidations), and the per-interest-set per-invalidation-stream information ($startVV$ as it is updated as each generalized invalidation is applied.) For clarity, we show only α's component for all version vectors and omit the node ID (α) in accept stamps.

tions (precise for a and imprecise for b and c) to β. As illustrated in the figure, each invalidation advances β's per-invalidation-stream, per-interest-set $startVV$ value as well as β's per-interest-set last precise version vector ($lpVV$) and current version vector (cVV) for interest set {a}. However, because the second invalidation $(4, 6, bc)$ intersects interest set {b,c}, that message causes that interest set to become imprecise and subsequent invalidations fail to advance that interest set's $lpVV$. After processing all four invalidations in that stream, β is precise for interest set {a}, but imprecise for interest set {b,c}. γ's behavior processing the stream of precise invalidations for c and imprecise invalidations for a and b is similar.

Then, when β and γ send their log contents to δ, we show the case where γ processes β's first three invalidations, then γ's four invalidations, and finally β's fourth invalidation. As the figure shows, after processing the first three invalidations from β, δ is precise for {a}, but imprecise for {b} and {c}. The next four messages (from γ) make δ precise for {c} but imprecise for {a} and {b}. Finally, the last message (from β) brings δ to the state one would desire: after seeing all precise invalidations for objects a and c, δ is precise for both interest set {a} and {c} despite the fact that these precise messages were mixed with some imprecise invalidations for objects a, b, and c. Finally, one may verify that because of the δ's gap filling and intersection operations, δ's log contains sufficient information so that a node ϵ that receives δ's log contents could get precise updates for objects a or c.[3] Conversely, note that if δ were simply to interleave the messages it received from α and β without gap filling and intersection and then send them to ϵ, information would be lost and ϵ would be left imprecise for interest sets {a}, {b}, and {c}.

Checkpoint recovery. The above protocol describes the common case of streaming, incremental log exchange. However, nodes can garbage collect their logs, so the system must handle the case when a node β requests data from α, but α's *currentVV* is newer than β's *lpVV* for a given interest set. The protocol handles this case by doing a full state transfer for the interest set: α sends β its *lpVV* and *cVV* for the interest set along with the *accept* stamp for each object in that interest set from α's per-object state. β updates its *lpVV* and *cVV* for the interest set and, if the *accept* time it receives for an object exceeds the locally stored *accept* time, it updates the local *accept* time for the object and marks the object *INVALID*. Note that checkpoint recovery can be done on a per-interest set basis, but for any interest sets not updated, *currentVV* must be advanced to at least the *currentVV* of the checkpoint.

Analysis. This algorithm retains topology independence and causal consistency, but it also allows partial replication of both bodies and invalidations. In particular, to maintain an interest set in the precise state requires O(number of writes to the interest set) precise invalidations plus one imprecise invalidation summarizing invalidations that do not intersect the interest set. In practice, systems may send more imprecise invalidations than this minimum in order to limit the delay in assembling and sending an invalidation stream as described in Section 3.1.

2.4 Tunable consistency

The basic mechanisms above provide a solid substrate over which it is straightforward to weaken the system's consistency guarantees (e.g., to improve performance [25] or availability in the face of partitions [5])

[3]And, in this case, b. Our current log maintanence algorithm actually extracts a bit more information from the stream of incoming requests than our interest set status algorithm; we are not sure if there is a clean way to extract this information during interest set maintenance as well.

or to strengthen the system's consistency guarantees to meet application semantic requirements.

Weakening consistency. By default, demand reads block until the interest set they reference is precise and they can ensure that the data they return represents a causally-consistent view of the system's state. We provide an interface that overrides this behavior by allowing *imprecise reads* that skip the *lpVV = cVV* check and return data as soon as the local store record for the requested object is valid regardless of whether the interest set in which it resides is precise or imprecise. Nodes that use this interface observe *causally coherent* data—if a node reads a version v_j of an object and then writes another version v_{j+1} of the object, than once any node reads version v_{j+1} of the object, any subsequent read will return version v_{j+1} or a later version—but they are no longer guaranteed to observe a causally consistent view—if a node reads version v_a of object a and then writes version v_b of object b, a node that reads version v_b of object b using an imprecise read may still observe a version of object a older than v_a.

The potential benefit of doing an imprecise read is that a node can read an object from a currently-imprecise interest set without communicating with other nodes to make that interest set precise. Imprecise reads can therefore reduce bandwidth consumption, improve response time, or improve availability. Note that even if a node α executes one or more imprecise reads and then issues some writes, the protocol ensures that α's log contains sufficient imprecise invalidations to put all of its invalidations into a causally consistent order: even if α sends its log to β, β can continue to provide causal consistency across all objects.

Strengthening consistency. A library interface built over the low-level mechanisms provided by the basic PRACTI interface can strengthen consistency guarantees. In particular, the *sync()* interface described above allows the construction of a write() that blocks until the update has propagated to a specified set of machines [22, 33]. Another option for strengthening consistency that we plan to explore is layering TACT over these basic mechanisms to provide tunable consistency guarantees [44].

Conflict detection and resolution. The simple protocol described above provides incremental log exchange and last-writer-wins conflict resolution with global eventual consistency in the case of concurrent writes. However, it is useful to not only resolve conflicts in a globally consistent way but also to flag them and provide information about conflicting writes to a more flexible manual or programmatic conflict resolution procedure. As we discuss in an extended technical report [8], we augment the protocol described above by including hooks to detect write-write conflicts (by adding a *prevAccept* field in all invalidation messages and per-object store records), storing "losing" writes in a local (unshared) per-object conflict file, and providing utility functions to read and delete losing writes from conflict files as part of a "compensating transaction" for application-specific conflict resolution. Causal consistency (as opposed to coherence) is useful for conflict detection and resolution: our protocol ensures that all nodes agree on the same set of conflicts and "losing" writes.

The extended report also describes how to use the PRACTI mechanisms with Bayou's more powerful strategy of associating application-specific conflict detection and resolution functions with writes [35]. Our reasons for a simpler approach are (1) to support incremental (rather than batch) log exchange for improved performance and (2) to avoid the need for a *commit* protocol that can ensure that late-arriving writes (which can include detection/resolution "programs" that can arbitrarily disrupt the current state) are placed after committed writes.

3 Implementation

Our PRACTI techniques cleanly separate mechanism from policy in order to support a broader range of replication policies than made available by current techniques that entangle policy choices with their mechanisms for replication, consistency, or topology. Our implementation therefore seeks to serve as a "replication microkernel" that provides basic low level mechanisms over which higher-level services can be built.

The PRACTI mechanisms ensure safety. Our prototype uses an asynchronous style of communication in which incoming messages or streams are self-describing—the rules for processing each incoming message are completely defined, and interpreting a message does not require knowledge of what request triggered its transmission. Because message handling rules are based on the PRACTI algorithms, they ensure safety regardless of the policy used for sending messages: any machine can send any legal protocol message to any other machine at any time, and the receiver's rules for processing incoming messages embed no assumptions about who communicates with whom, make no assumptions about what data is replicated to which machines as well as enforce rules that track each object or interest set's consistency state based on all messages received.

Because the low-level mechanisms enforce safety independent of policy, higher level policies can focus on liveness (including performance and availability concerns.) Essentially, the policy layer's job is to ensure that the right nodes send useful data at the right time in order to do such things as to satisfy a read miss, prefetch data to improve performance, or provision a node's local storage so that it can make its data available while discon-

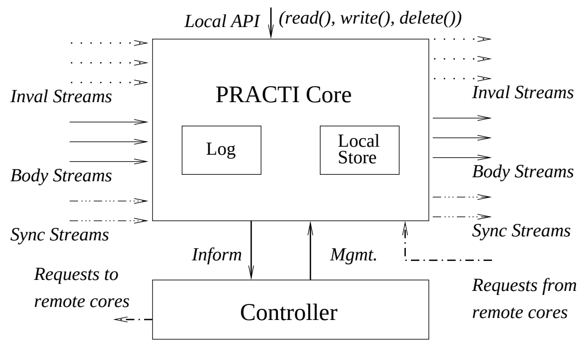

Fig. 2: High level architecture of PRACTI prototype.

nected. Each node provides an interface for requesting that the node send invalidations or bodies to other nodes, but these requests can be regarded as hints: the loss of messages or the introduction of extra messages can affect system performance but not the correctness of responses to application read and write requests.

Figure 2 illustrates the division of labor between mechanism and policy in our system. A PRACTI *core* maintains local state in a *log* for reliability and communication and a *local store* for random access. A core receives and generates streams of general invalidations (precise, imprecise, and bound invalidations), bodies (demand replies and prefetched/pushed data), and sync replies (to support unbind and consistency policies as described in Sections 2.2 and 2.4.) The core also provides a remote request interface that allows remotely-generated requests to trigger outgoing streams or individual messages.

A *controller*'s purpose is to send requests asking other nodes' cores to trigger streams or messages. To aid this task, the core informs its local controller of important events (e.g., connection initiation/termination, local requests, and message arrivals.) To customize a replication system to an environment, different controllers use different policies. For example, we implement a StaticTopologyController that creates a static topology among its nodes for propagating invalidations and bodies and for satisfying demand requests, a BayouController that permanently leaves all invalidations in the bound state, and a SDIMSController that uses the DHT-based Scalable Information Management System [40] to track the state of the distributed system.

3.1 Core implementation

The core implements procedures applying incoming request messages to its local state that ensure that the rules described in Section 2 are enforced. A core's local state has two main parts: a log and a data store.

Log. A core's log has two main purposes. First, it acts as a replay log for reliability. Second, it maintains causally-ordered lists of invalidations for communicaton with other nodes.

Our log implementation has two components. First, it has a single on-disk append-only replay log in which invalidation messages, local updates, and unbind messages are stored in the order they are received. Second, it maintains an in-memory per-writer log of invalidations and local updates sorted by accept stamp. Incoming messages are first appended to the on-disk replay log and then, as described in Section 2.3, they are decomposed into single-writer invalidations that are merged with the single writer logs using *gap filling* and *intersection* to enforce the invarient that each per-writer log contains a gap-free list of elements that do not overlap in time.

Data store. The data store maintains per-interest set *status*, which tracks the last precise version vector (*lpVV*) and current version vector (*cVV*) for each interest set as described in Section 2.3. In our implementation, an interest set is identified by a subdirectory name and includes the path from the root to that subdirectory as well as all enclosed subdirectories.

The data store also maintains per-object metadata and body information. For each object in the system, one file on the local disk holds the body of the object, with byte i of the file corresponding to byte i of the object. A second file holds the object's consistency state: a series of records with an *offset* and *length* identifying a byterange, *accept* identifying accept stamp of the most recent invalidation applied to the byterage, *prevAccept* identifying the accept stamp of the previous write to the byterage (for conflict detection as described in Section 2.4), and a *valid* flag indicating whether the body file's contents are VALID or INVALID for this range. For simplicity, we implement each object's consistency state as a Java object, manage an in-memory cache of these objects, and serialize dirty objects to per-object disk files for checkpoints.

Operation. Section 2 outlines how a core processes incoming invalidation and body messages as well as local read requests. Local write and delete requests are treated like incoming invalidation requests—they are first applied to the log and then to the local store. Incoming sync replies have no effect on the core's state.

Each core has an interface to trigger outgoing streams of invalidations. A request to start an invalidation stream includes the *destination* node ID to which to send the data, a *startVV* version vector indicating the desired starting point, and a precise set *P* listing subdirectories for which the receiver would prefer to receive precise invalidations if the sender has them available. A sender thread has two tasks: First, it must draw requests from the per-writer logs in a causally consistent order, and second it should reduce network overhead by combining some invalidations into imprecise invalidations and sending the resulting stream of general invalidations in a causally consistent order. It accomplishes the first task by initializing *sentVV* = *startVV*, drawing from the per-writer logs the element with the lowest accept stamp that exceeds *sentVV*, and updating *sentVV* to include the end time of the element. Key to accomplishing the second task is the following observation:

> Given a causally consistent sequence S of general invalidations $S = (g_0, g_1, \ldots, g_{n-1})$, select any two subsequences S_1 and S_2 such that g_0 appears in S_1, each element of S appears in either S_1 or S_2, and all elements in S_1 and S_2 appear in the same relative order as in S. Form an imprecise invalidation I that is the *union* of all invalidations in S_1 (as defined in Section 2.3.) Then, the sequence $S' = (I, S_2)$ represents a causally consistent sequence.

This property follows from the fact that if the imprecise invalidation I intersects a receiver's interest set IS, then when it arrives, the receiver advances $IS.cVV$ to $I.end$ but does not advance $IS.lpVV$; conversely, if I does not intersect IS, then when it arrives, the receiver waits until at least $I.end$ before advancing either $IS.cVV$ or $IS.lpVV$. In the either case, when processing each message g_i from S_2, $IS.lpVV$ is no higher than it would have been if g_i were processed as part of the original sequence S, and $IS.cVV$ is at least as high, so IS is only precise after processing message g_i under S_2 if it would have been precise after processing the message under S.

To save bandwidth while avoiding unnecessarily making interest sets imprecise, a sender therefore buffers outgoing invalidations and aggregates ones that do not intersect *P*. When an outgoing stream draws a sequence S of invalidations out of the log, it adds each g_i to I if g_i does not intersect the precise set P and it appends g_i to a sequence of pending invalidations S_2 otherwise. A node sends and clears I and then S_2 after one of two timeouts occurs: either $T_{precise}$ ms have elapsed since the first element was placed in this instance of S_2 or $T_{imprecise}$ ms have elapsed since I became non-empty. Typically $T_{imprecise} > T_{precise}$ since nodes may tolerate longer delays for updates about information they don't care about. Note that our current prototype implements a limited version of this logic that allows $T_{imprecise}$ to be set by the trigger request but that assumes $T_{precise} == 0$.

Generating outgoing body streams is similar but simpler because the safety of the system does not depend on the order of body messages or sync replies. When a node receives a request for a body, the node uses data in its local store to generate and send a body message with the object ID, byte range, the range's accept stamp, and as much data beginning at the requested offset as is valid. Note that if the local data is in the INVALID state, the node's reply would indicate a zero-length body, which has no semantic effect at the receiver, but which will generate an event the receiver's controller can use as a hint that it should retry (perhaps to a different node); if the data store does not have a record for that object/offset, the node generates an impossibly low-numbered accept stamp for its reply which has the same effect. For efficiency, our prototype maintains a pool of TCP connections for body messages to amortize TCP setup costs and to pipeline sends when multiple bodies are sent to a node.

A core also provides an interface to request that a node α push bodies newer than some version vector to another node β for some specified object or subtree in the object name space. In our implementation α allocates a bounded-size priority queue which drains update body messages to β over a low-priority network connection [37], and α inserts into this priority queue a reference to each new body matching the subtree using a per-object priority supplied by α's controller.

Recovery and garbage collection. In order to allow trimming of update logs, nodes checkpoint their local store state. A checkpoint comprises a *currentVV* version vector that indicates the on-disk state reflects at least the application of general invalidations up to *currentVV*, the list of interest sets, a per interest set *lpVV* version vector indicating the last time the interest set was precise, the per-object metadata (current to at least *lpVV* for each object's interest set), and the per-object body for at least any bound invalidations that are reflected in the checkpoint (a node's controller is always free to direct the node to discard any unbound body to limit space consumption.) Once such a checkpoint is stored, the prefix of the log before *currentVV* may be truncated, though in practice we keep a longer prefix in the log to facilitate incremental synchronization among nodes [31].

3.2 Controller implementation

Each core has a controller that initiates the communication that the core needs such as subscriptions to invalidation streams, subscriptions to prefetch body streams, and requests for bodies to satisfy demand read misses. Controllers also issue maintenance directives to the local core for issues like cache replacement and garbage collection.

The controller subsystem is defined by its interface. Within this interface, we anticipate a wide range of different implementations providing different policies.

Interface and operation. Controllers use three interfaces to accomplish their work: a core calls a controller's *inform* interface to inform the controller of important events, a controller calls a remote core's *remote request* interface to trigger sends, and a controller calls its core's *management* interface for maintenance functions like cache control. Additionally, a set of controllers implementing a specific distributed policy may communicate with one another using policy-specific interfaces.

A core uses its local controller's inform interface to inform the controller of events of interest. In our implementation, a core informs it local controller of (1) *stream connection* initiation or termination for invalidations or updates, (2) inval, sync, and body *message arrival* events, and (3) *local events* like read hit, read miss, read imprecise (a read that blocks accessing an imprecise interest set), and write.

Controllers can respond to inform events by sending request messages to a remote core's remote request interface. For example, when informed of a read miss, a controller uses some policy-specific strategy to identify a node that can supply the miss and sends a request to that node for the body. Then, one of three things will happen: (1) the body arrives at the core, unblocks the waiting read request, and causes the core to inform its controller of the body arrive event, (2) an empty body arrives at the core (signifying that the sender does not have the desired data), the controller receives a body arrive event for the empty body, and the controller sends another body read request, or (3) a timeout event occurs within the controller and the controller issues another body read request.

Finally, the core has a local management interface that allows the controller to query the core to learn about internal state (e.g., the intererest set status, per-object state, log status, and connection status) and to manage that local state (e.g., shut down a connection, mark an object as invalid and garbage collect its body storage, or begin tracking *lpVV* and *cVV* for a new interest set.)

SDIMS Controller To more concretely illustrate the interactions between the controller and the core, we describe one of the controllers we have built. The SDIMS controller uses the DHT-based SDIMS system [40] to coordinate a distributed collection of controllers. Note that the current SDIMS Controller is intended as a proof of concept for the PRACTI mechanisms rather than as a full-fledged replication system. Although we intend to build complete replication system using SDIMS and PRACTI, some desirable features are not yet implemented as we detail below.

Our prototype uses SDIMS to maintain per-interest-set spanning trees for both invalidation and update streams. As Figure 3 illustrates, for a given interest set IS, a node informs SDIMS of its interest in IS, and SDIMS aggregates this information across locality-aware and administrative-unit-aware trees, selecting an interested node from each subtree to function as the subtree's root. A node then finds its parent using SDIMS and creates invalidation (and optionally, update) streams to and from its parent for IS. Note that some updates to the in-

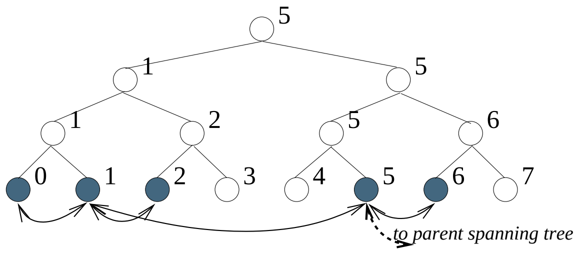

Fig. 3: Example invalidation/update spanning tree formed by SDIMSController for an interest set (e.g., /foo/bar/*). The circles represent the virtual tree formed by SDIMS for interest set /foo/bar/*, the solid nodes represent nodes interested in /foo/bar/*, the numbers denote the ID of the node selected by SDIMS as the spanning tree root for each SDIMS subtree, and the arrows show the node-to-node connections made based on SDIMS's guidance.

terest set */foo/bar/** are relevant to the interest set */foo/** (and vice versa), so the root node of the spanning tree for IS selects as its parent any node in the spanning tree for the shorter *path'* formed by deleting after the last "/" in IS's *path*. A controller maintains spanning tree connections by retrying on communication failures and when SDIMS notifies a node that its parent in in the spanning tree has changed.

We use a similar approach for maintaining a distributed directory for satisfying local read misses. Each node informs SDIMS of the valid byte ranges it caches and queries SDIMS on misses to find a nearby copy of data [32, 36].

Note that SDIMS ensures only eventual consistency, so a spanning tree parent or body supplier suggested by SDIMS may not be the correct parent, may not have the desired data, or may be unreachable. The first problem is handled by using SDIMS's *continuous probe* interface to notify a controller when its parent changes. A controller handles stale values and timeouts by retrying SDIMS queries with a flag to *reaggregate* stored values from children in the distributed tree [40].

A complete version of an SDIMS-based distributed file system would require several additional features. First, we plan to use SDIMS to allow a node to locate a nearby node whose interest set status for some interest set is precise up to a specified point in time. This information is useful for "filling holes" when a node receieves an imprecise invalidation for an interest set it wishes to maintain as precise. Providing this information will entail maintaining per-interest set, per-writer aggregation functions so that an SDIMS subtree will identify the node in the subtree with the highest accept stamp for a given interest set and writer. Second, we plan to use SDIMS to track the read and write rates to different objects. Prefetch algorithms use this information to prioritize replication [38, 39]. Third, a complete controller should implement policies for local cache replacement

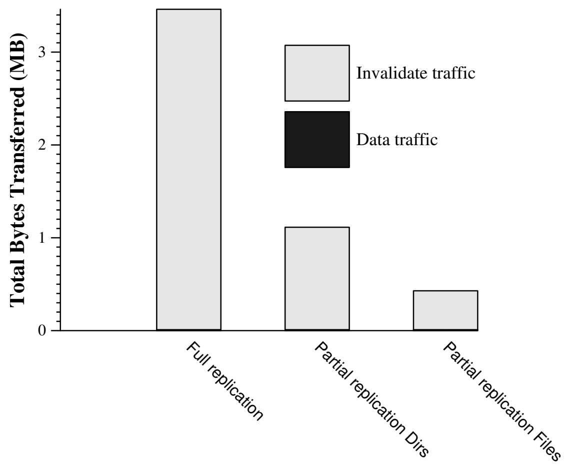

Fig. 4: Scalability of PRACTI

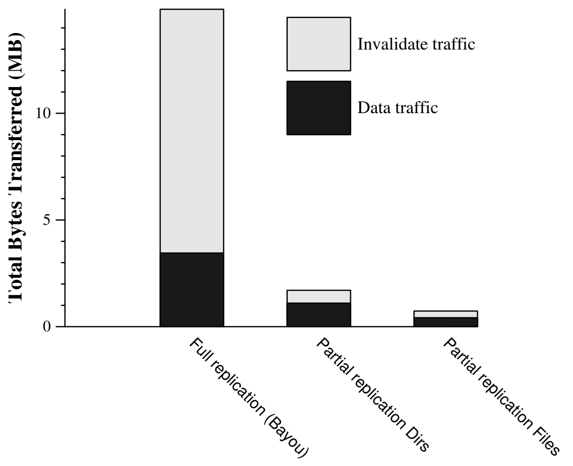

Fig. 5: Scalability of PRACTI

and garbage collection of the log.

4 Evaluation

In this section we evaluate the properties of our PRACTI prototype. The flexibility provided by the PRACTI mechanisms provides two significant advantages over past systems. First, by disentangling mechanism from policy, PRACTI represents a single flexible system that can match systems that have been optimized for specific topology, replication, or consistency environments. Second, by providing a clean general substrate, PRACTI enables better trade-offs than are available to any existing system for some important environments.

Based on our experiments, our primary conclusions are (1) the separation of invalidations from updates can reduce bandwidth consumption by an order of magnitude compared to full-replication systems when workloads have locality of interest, (2) the use of imprecise invalidations can provide a further significant reduction in synchronization overheads in systems with large numbers of files when some nodes only care about subsets of those files, (3) flexible topologies can significantly reduce synchronization delays, particularly in mobile or low-bandwidth environments, and (4) imprecise invalidations make the bandwidth cost of providing consistency guarantees approach the cost of providing weaker coherence guarantees.

We show in figure 4 and 5 the number of bytes transferred for each of our various replication strategies. We run our experiments on two machines - a sender, which writes to random files, and a receiver that reads random files. At the sender, we generate 1000 files with 10000 bytes each, and perform 10000 random writes. The receiver then reads 10 of those files. We assume that the receiver replicates 10% of the directories in the system, and for each directory, we assume that the receiver replicates 10% of the files in that directory.

We evaluate the bandwidth consumed by PRACTI under 3 configurations: (1) *Full replication*, where the sender sends precise invalidates for all modified files, (2) *Partial replication Dirs*, where the sender sends a precise invalidate message for each modified file *f* if *f* lies in a directory *d* that is replicated at the receiver (even if only a subset of *d* is replicated at the receiver) and (3) *Partial replication Files*, where the sender sends precise invalidate messages for exactly only those files that are replicated at the receiver.

In figure 4 we evaluate the bandwidth consumed by PRACTI under *conservative* algorithms, where the sender does not push any files to the receiver but instead forces the receiver to demand-fetch files as necessary. We note that by restricting the sender to sending precise invalidate messages for only those files that lie in replicated directories, we successfully reduce bandwidth consumption by a factor of 3.1. When we restrict the sender to sending invalidate messages only for exactly those files that lie in the receiver's interest set, we successfully reduce bandwidth consumption by a factor of 8.1. Due to the large difference between the number of files written at the sender and the number of reads at the receiver, most of the bandwidth ($> 97.6\%$) is spent sending invalidate messages; as a result, the *Demand Data* bar is not visible. If transferring file data consumes more bandwidth, the relative benefits yielded by sending imprecise invalidates would be reduced.

Figure 5 shows the bandwidth consumed by PRACTI when using *aggressive* algorithms, where for each modified file for which the sender sends an invalidate message it also sends the modified data. The first line in the figure represents the case where the sender pushes all updates to the receiver, as is done by *Bayou* [31]. However, we note that by restricting the sender to sending only those updates that occur in files that lie in replicated directories, the sender consumes a factor of 8.7 less bandwidth.

	Laptop/PDA/Phone BW	All BW
Replicate all	2521KB	35125KB
All-inval, interest-update	483KB	3612KB
Interest-update, interest-inval	443KB	2991KB
Hierarchy	*impossible*	1588KB

Table 1: Bandwidth consumption for synchronization.

Furthermore, by restricting the sender to sending only those updates that are to files replicated at the receiver, the sender uses a factor of 20 less bandwidth compared to the fully replicated configuration.

Table 4 shows the bandwidth costs of synchronizing a collection of machines using various mechanisms and policies. In this (emulated via NistNet) scenario, a user in a hotel room has a laptop, PDA, and phone that share a 1Mbit/s wireless connection, and the user also has an account on a fixed server that the laptop can access via a 50Kbit/s modem link (when it is available). We use a synthetic workload in which 100K files each of 10KB size exist at the server, with 10K of those files at the laptop, 1K at the PDA, and 100 at the phone. We assume that since the last synchronization event, 1% of the files at each location have changed. We compare synchronization costs under two scenarios: (1) no connection to the server is available and the laptop, PDA, and phone are only able to communicate with one another and (2) a connection to the server is available.

The table compares four protocols for synchronizing the devices. First, the replicate-all approach replicates all data and distributes all updates to all devices (similar to Bayou). The second strategy separates invalidations and updates, has the devices subscribe for all invalidations, but has them only subscribe to (i.e., hoard [21]) updates for the files in their interest sets. The third strategy restricts subscriptions to the interest sets for both metadata and data. And, the fourth strategy requires all communication to be between the server and a client as in traditional client-server systems; like the third approach, our client-server toplogy system restricts subscriptions to the interest set for both data and metadata.

As the table illustrates, separating invalidations from update bodies and providing nodes with the flexabity to only access the bodies they care about significantly reduces bandwidth requirements. In this example, the second strategy uses about an order of magnitude less bandwidth than the first. Also note that allowing nodes to observe only subsets of invalidations provides significant further reductions. In this example, where the laptop and server share 10% of their data, the third strategy reduces bandwidth by about 10%; if the universe of data were larger than the 1GB used here and as if devices shared smaller subsets of data, this number would increase.

Finally note the advantage of topology independence. The centralized synchronization of metadata required by

	Laptop/PDA/Phone Sync Time	All Sync Time
Replicate all	26s	>1200s(*)
All-inval, interest-update	7.5s	402.2s
Interest-update, interest-inval	7.4s	400.3s
Hierarchy	impossible	427.4s

Table 2: Synchronization delays. (*) Due to time limitations, we were unable to complete the replicate-all run over the slow network link for this submission, and we cut the run short after 1200 seconds. Given these bandwidth constraints, the full run must take at least 1873 seconds.

some replication systems would force the user in this scenario to dial in in order to synchronize her PDA and laptop, even if the two devices are in the same room, thousands of miles away from the server; clearly such restrictions are burdensome.

Table 4 further illustrates this scenario. This table shows the synchronization times for an unoptimized version of our system, using NistNet to restrict bandwidths to the values listed above. Compared to a replicate-all strategy, partial replication reduces synchronization delay by over a factor of five, and we would expect that gap to widen as we tune our system. The optimized peer-to-peer exchange of data also reduces time compared to a hierarchical system, even when the network to the server is available.

The following table illustrates the efficiencies that come from imprecise invalidations as well as the benefits of having the flexibility to choose which data to track in detail:

	Precise	Imprecise
Subscribe 10000	349723 bytes	1769 bytes
Subscribe 1000	4546 bytes	3122 bytes

In this experiment, a node that had been imprecise for a directory subtree containing 100,000 files references a file in that subtree. To do so, the node must become precise for at least the file in question, but since the process is likely to reference other data in that subtree, it may also make the directories that include that file precise for several levels of ancestors. We show two cases: where the node makes the nearest 10,000 files (10% of the data) precise and where the node makes the nearest 1,000 files (1% of the data) precise. Note that the first case requires about an order of magnitude more bandwidth than the second approach due to imprecise invalidations' ability to stand in for large numbers of precise invalidations. Note also that the additional overheads required to carry imprecise invalidations (and thereby provide consistency, not just coherence) are small compared to both the precise information and (not shown) compared to the body data of the files being accessed. And finally note that the imprecise invalidations reduce the metadata bandwidth cost of synchronizing a subset of this volume by orders of magnitude compared to synchronizing all items precisely.

The experiments described above demonstrate the key properties of the PRACTI approach. Our evaluation efforts are ongoing and we expect to complete additional experiments in the immediate future.

5 Related work

Replication is fundamentally difficult. For example Brewer describes the CAP dilemma [5]: a replication system that provides sequential *C*onsistency cannot simultaneously provide 100% *A*vailability in an environment that can be *P*artitioned. Similarly, Lipton and Sandberg describe fundamental performance limitations for distributed systems that provide sequential consistency [25]. As a result, systems *must* make compromises or optimize for specific workloads. Unfortunately, these workload-specific compromises are often reflected in system mechanisms, not just their policies.

In particular, state of the art mechanisms allow a system designer to retain full flexibility along at most two of the three dimensions of replication, consistency, or topology policy.

A first set of systems such as Sprite [30], AFS [19], and Coda [21] support arbitrary replication policies and in principle could support a range of consistency policies [41] (though, in practice, such systems typically implement a specific consistency policy), but these protocols fundamentally assume a topology policy that restricts communications to hierarchical paths. Even when client-server systems permit limited client-client communication for cooperative caching [2, 10, 12] serialization of control messages at the server is vital for reasoning about consistency [6].

A second set of systems such as Bayou [31], TACT [43], and Ivy [28] use a log-propagation mechanism that is capable of providing a range of consistency guarantees [43] and that supports arbitrary topologies. However, these mechanisms assume a replicate-all placement policy that maintains a copy of all objects in a volume on each node that participates in the volume's replication system.

A third set of systems such as Ficus [17] and Pangaea [33] maintain synchronization information separately for each object and support arbitrary topology policies and arbitrary replication policies. However, although these systems can provide some coherence guarantees on the order of reads and writes when an individual object is considered, they provide limited consistency guarantees regarding the ordering of reads and writes across objects.

6 Conclusion

In this paper, we present the first PRACTI (Partial Replication, Arbitrary Consistency, and Topology Independence) mechanism for replication in large scale systems. These new mechanisms allow construction of systems that replicate or cache any data on any node, that provide a broad range of consistency and coherence guarantees, and that allow any node to communicate with any other node at any time. Our evaluation of our prototype suggests that *by disentangling mechanism from policy, PRACTI replication enables better trade-offs for system designers than possible with existing mechanisms.* By cleanly separating mechanism from policy, we speculate that PRACTI may serve as the basis for a *unified replication architecture* that simplifies the design and deployment of large-scale replication systems.

References

[1] Tivoli data exchange data sheet. `http://www.tivoli.com/products/documents/datasheets/data_exchange_ds.pd%f`, 2002.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Trans. on Computer Systems*, 14(1):41–79, February 1996.

[3] Bent and Voelker. Whole page performance. In *Workshop on Web Caching and Content Distribution*, September 2002.

[4] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 521–528, June 1992.

[5] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July/August 2001.

[6] S. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a Language for Writing Coherence Protocols. In *USENIX Conference on Domain-Specific Languages*, October 1997.

[7] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN service availability. *ACM/IEEE Transactions on Networking*, 11(2), April 2003.

[8] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for large-scale systems. http://www.cs.utexas.edu/users/dahlin/papers/2004-PRACTI-osdi-submission-extended.pdf, May 2004.

[9] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proc. SIGMETRICS*, pages 150–160, May 1994.

[10] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proc. OSDI*, pages 267–280, November 1994.

[11] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proc USITS*, December 1997.

[12] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proc. SOSP*, pages 201–212, December 1995.

[13] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Workshop on Internet Server Performance*, June 1998.

[14] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *International World Wide Web Conference*, May 2003.

[15] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. SOSP*, pages 202–210, 1989.

[16] J. Griffioen and R. Appleton. Reducing File System Latency Using A Predictive Approach. In *USENIX Summer Conf.*, June 1994.

[17] R. Guy, J. Heidemann, W. Mak, T. Page, Gerald J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer Conf.*, pages 63–71, June 1990.

[18] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *HOTOS95*, pages 51–55, May 1995.

[19] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[20] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proc. ICDCS*, pages 302–311, 1990.

[21] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, February 1992.

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

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

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

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

[26] P. Maniatis, M. Roussopoulos, TJ Giuli, D. Rosenthal, M. Baker, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. SOSP*, October 2003.

[27] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash. In *USENIX Winter Conf.*, pages 305–313, January 1992.

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

[29] A. Nayate, M. Dahlin, and A. Iyengar. Transparent information dissemination. In *Proc. Middleware*, October 2004.

[30] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1), February 1988.

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

[32] G. Plaxton, R. Rajaram, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[33] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. OSDI*, December 2002.

[34] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proc. OSDI*, pages 35–46, October 1996.

[35] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. SOSP*, pages 172–183, December 1995.

[36] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proc. ICDCS*, May 1999.

[37] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In *OSDI02*, December 2002.

[38] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a wan. In *PODC*, August 2001.

[39] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. *Elsevier Computer Communications*, 25(4):367–375, March 2002.

[40] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *ACM SIGCOMM 2004 Conference*, August 2004.

[41] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering web cache consistency. *ACM Transactions on Internet Technologies*, 2(3):224–259, 2002.

[42] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proc USITS*, October 1999.

[43] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proc. SOSP*, 2001.

[44] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. on Computer Systems*, 20(3), August 2002.

A Additonal details

A.1 Design: Multi-interest set generalization

Section 2.3 shows a simplified version of the PRACTI algorithm for applying a stream s of general invalidations to a single interest set IS. Our actual implementation handles multiple interest sets by drawing each invalidation from the stream and applying the invalidation to the log and then to each interest set status record. To support this, we maintain per-stream, per-interest-set *startVV*s. Algorithm 2 shows this extension.

A.2 Design: Conflict detection and resolution

The simple protocol described in Section 2.4 provides incremental log exchange and last-writer-wins conflict resolution with global eventual consistency in the case of concurrent writes. However, it is useful to not only resolve conflicts in a globally consistent way but also to flag them and provide information about conflicting writes to a more flexible manual or programmatic conflict resolution procedure.

To support more flexible conflict detection and resolution, we augment the algorithm described above by adding a field, *prevAccept* to both invalidation messages and to per-object store state. When a node receives an invalidation inv and applies inv to the local store of an object obj (with $inv.accept \neq obj.accept$), there are three cases to consider. First, if $inv.prevAccept == obj.accept$, there is no write-write conflict. The second case, $inv.prevAccept > obj.accept$, is impossible by the prefix property. The third case, $inv.prevAccept < obj.accept$, represents a write-write conflict, which is resolved by updating obj with either inv or obj dependig on which has a higher accept stamp and by storing the losing entry to disk in a local (non-shared) per-object *conflict file*; bodies that match stored losing writes are also stored. PRACTI implementations can provide a local interface for reading and deleting these "losing" conflicting writes, which allows higher-level code to resolve conflicts using application-specific rules by generating compensating transactions.

Note that although different nodes can see different series of "losing" writes, all nodes that make an interest set precise are guaranteed to see the "final" write to each

Algorithm 2 ProcessInvalStream $s = \{startVV,\ gi_1,\ gi_2, \dots\}$

```
// First, process the startVV token.
$svv = s$.next()
for all $IS$ in $allISs$ do
    $startVVs_{IS} = svv$;
    if $\exists\alpha \mid svv_\alpha > IS.currentVV_\alpha$ then
        // Create and apply "gap filling" imprecise inval
        $\forall\alpha : gapInv.start_\alpha = MIN(svv_\alpha, IS.currentVV_\alpha)$
        $\forall\alpha : gapInv.end_\alpha = MAX(svv_\alpha, gapInv.start_\alpha)$
        $gapInv.target = IS.path$ // Any target that overlaps IS
        applyOverlappingAtStrtTm($IS$, $gapInv$, $IS.currentVV$)
// Now, process each general invalidation in stream.
$pending$ = new Set()
$gi = s$.next()
while $(gi \neq null)$ do
    $\forall\alpha : nextStartVV_\alpha = MAX(startVV_\alpha, gi.start_\alpha)$
    if $!(\exists p \in pending \mid (\forall\alpha : p.end_\alpha \leq nextStartVV_\alpha))$ then
        $log$.insert($gi$)
        for all $IS$ in $allISs$ do
            if $gi.target$ intersects $IS$ then
                applyOverlappingAtStrtTm($IS$, $gi$, $startVVs_{IS}$)
        $startVVs_{IS} = nextStartVV$
        if $gi.isPrecise()$ then
            $store_{gi.objId}$.update($gi.start, INVALID$)
        $pending$.insert($gi$)
        $gi = s$.next()
    else // Apply non-overlapping p from $pending$ at end time
        for all $IS$ in $allISs$ do
            if !($p.target$ intersects $IS$) then
                applyNonOverlappingAtEndTime($IS$, $p$, $startVVs_{IS}$)
        $pending$.remove($p$)
Procedure applyOverlappingAtStrtTm($IS$, $gi$, $startVV$)
    if $gi.isPrecise()$ AND $\forall\alpha : IS.lpVV_\alpha \geq startVV_\alpha$ then
        // If no gaps to this precise inval, update lpVV
        $\forall\alpha : IS.lpVV_\alpha$ =MAX($IS.lpVV_\alpha, gi.start_\alpha$)
    $\forall\alpha : IS.currentVV_\alpha$ = MAX($IS.currentVV_\alpha, gi.end_\alpha$)
Procedure applyNonOverlappingAtEndTime($IS$, $p$, $startVV$)
    if $\forall\alpha : lpVV_\alpha \geq startVV_\alpha$ then
        $\forall\alpha : lpVV_\alpha$ =MAX($lpVV_\alpha, p.endVV_\alpha$)
    $\forall\alpha : currentVV_\alpha$ =MAX($currentVV_\alpha, p.endVV_\alpha$)
```

causally–independent series. For example, consider the case of two causal chains of writes to one location by the nodes α, β, and γ: (1) $w0\alpha, w1\beta, w2\beta, w3\beta$ and (2) $w0\alpha, w4\gamma$. The protocol guarantees that eventually any precise node will agree that the final state of the write is the result of γ's write at time 4 and that there was a write-write conflict that $w3\beta$ lost, and but different nodes may see different subsets of $w1\beta, w2\beta, w3\beta$, which seems acceptable in that neither causal chain regards either $w1\beta$ or $w2\beta$ as important values for the final state of the system.

Alternative: Per-write conflict detection and resolution code. The PRACTI mechanisms are also compatible with Bayou's more powerful strategy of associating application-specific conflict detection and resolution code with each write w and re-executing this code each time the set of writes preceding w is changed by a log exchange operation. An advantage of this more flexible approach is that it can detect both write-write and read-write conflicts. We chose to use the simpler last-writer-wins and compensating transaction approach for two reasons.

First, our more restrictive approach allows efficient incremental application of interleaved streams of updates because it does not require "roll back" of the current random access state to process an arriving write: the determination of whether a conflict occurred and the decision about the final state of the object can be made by comparing the write's *acceptStamp* and *prevWrite* fields with the local object's *acceptStamp* and *prevWrite* fields. In contrast, Bayou's conflict detection and resolution code logically run at the point in time when the write occurs, so they must be able to read the state of the system at that time. As a result, to apply a newly-arriving write, the system first rolls back its state to the logical time of the write; it then applies the write and reapplies all subsequent writes. This cost is tenable in Bayou because Bayou was designed for batch application of updates, which amortizes the cost of rolling back and reapplying updates across a batch of newly arrived updates.

Second, our simpler approach allows us to avoid the need for a "commit protocol" that can establish a final write order that differs from the natural order on accept stamps. Bayou's "in line" execution of powerful conflict resolution code introduces the possibility of a "butterfly effect" in which the introduction of a single, previously unseen, low-timestamped write into a log can cause any or all newer writes in the log to execute a different conflict detection or resolution code path and to therefore write different values to different objects. In principle, whenever a previously unseen old write is applied, the resulting system state can look arbitrarily different from the previous system state. Bayou limits this problem by using a primary commit protocol so that connected nodes can establish an order on writes that causes "late-arriving" writes to be sorted after "on-time" writes. Conversely, a last-writer-wins approach is less vulnerable to late-arriving writes: either (a) despite the delay the late-arriving write is logically the newest write to the object and the object is updated or (b) the late-arriving write is logically older than other writes that have been applied and it has no effect other than being logged as a conflict.

Neither of these considerations is fundamental to PRACTI, and these trade-offs would apply to existing systems as well. One factor that may be more relevant to PRACTI is that the centralized commit protocol used in Bayou may limit scalability under PRACTI because it requires the primary node to see all invalidation messages; this issue does not limit Bayou because Bayou already requires all nodes to see all updates. An open question is whether there exists a suitable *scalable* commit protocol that can avoid the need for any node to see all of the invalidations.

A.3 Implementation: Checkpoint recovery

When a node boots, it first recovers the local store from the checkpoint and the in-memory per-writer log from the on-disk log. Then, the node simply issues a request to its own network request interface asking the log to send a stream of invalidations with a precise set comprising the union of the node's interest sets and starting at *currentVV*. The node then receives a causally consistent stream of all of the updates it has in the log but has not yet applied to its checkpoint, and it processes those requests normally, which adds them to the log (where they are ignored as redundant) and to the data store.

If a node receives a request for a stream of updates beginning earlier than the start of the node's log, the node's responds by sending its checkpoint of the requested interest set followed by the normal stream of invalidations. To apply a remote checkpoint to a node's local state for interest set IS, the node first treats the checkpoint as an imprecise invalidation that intersects all of its interest sets and that starts at node's current *currentVV* and that ends at max(the node's *currentVV*, the checkpoint's *currentVV*) and applies that inferred imprecise invalidation to its log and interest set status. The node then update's the state of IS by updating it's $IS.lpVV$ to max($IS.lpVV$, checkpoint $lpVV$) and by applying each invalidation stored in the checkpoint's per-object state as a precise invalidation to the local log and local state.