PRACTI Replication (Extended version)

Mike Dahlin, Lei Gao, Amol Nayate,
Praveen Yalagandula, Jiandan Zheng
University of Texas at Austin

Arun Venkataramani
University of Massachusettes at Amherst

Draft – October 2005
See http://www.cs.utexas.edu/users/dahlin/papers.html for the current version.

Abstract

We present PRACTI, a new approach and architecture for large-scale replication. PRACTI systems can replicate or cache any data on any node (Partial Replication), provide a broad range of consistency and coherence guarantees (Arbitrary Consistency), and permit any node to share updates with any other node (Topology Independence). Our PRACTI architecture yields two significant advantages. First, it provides *better trade-offs* than existing mechanisms: our prototype dominates existing approaches by providing as much as an order of magnitude better bandwidth and storage efficiency than AC-TI replicated server systems, as much as an order of magnitude better synchronization delay compared to PR-AC hierarchical systems, and consistency guarantees not achievable by PR-TI per-object replication systems. Second, our architecture's *flexibility* simplifies the design of replication systems by allowing a single framework to subsume a broad range of existing systems and to reduce development costs for new ones. For example, we use our PRACTI prototype both to emulate existing server replication, client-server, and object replication systems and to implement novel policies that improve performance for mobile users, web edge servers, and grid computing.

1 Introduction

This paper describes PRACTI, a new data replication approach and architecture that can reduce replication costs by an order of magnitude for a range of large-scale systems and also simplify the design, development, and deployment of new systems.

Data replication is a building block for many large-scale distributed systems such as mobile file systems, web service replication systems, enterprise file systems, and grid replication systems. Because there is a fundamental trade-off between performance and consistency [26] as well as between availability and consistency [7, 36], systems make different trade-offs among these factors by implementing different placement policies, consistency policies, and topology policies for different environments. Informally, *placement policies* such as demand-caching, prefetching, push-caching, or replicate-all define which nodes store local copies of which data, *consistency policies* such as sequential [25] or causal [19] define which reads must see which writes, and *topology policies* such as client-server, hierarchy, or ad-hoc define the paths along which updates flow.

The goal of the approach is to simultaneously provide all three PRACTI properties:

- *Partial Replication* means that a system can place any subset of data on any node. In contrast, some systems require a node to maintain copies of all objects in all volumes they export [30, 47].
- *Arbitrary Consistency* means that a system provides flexible semantic guarantees, including the ability to selectively enforce both *consistency* and *coherence* guarantees.[1] In contrast, some systems can only enforce coherence guarantees but make no guarantees about consistency [16, 34].
- *Topology Independence* means that any node can exchange updates with any other node. In contrast, many systems restrict communication to client-server [18, 21, 29] or hierarchical [6, 45] patterns.

Although many existing systems can each provide two of the properties, we are aware of no system that provides all three. As a result, systems give up the ability to exploit locality, support a broad range of applications, or dynamically adapt to network topology.

Our PRACTI architecture provides all three properties by drawing on key ideas of existing protocols but recasting them to remove the deeply-embedded policy assumptions that prevent one or more PRACTI properties. In particular, our design begins with log exchange mechanisms that support a range of consistency guarantees and topology independence but that fundamentally assume full replication [30, 47]. It then adapts these mechanisms to support partial replication using three design principles.

[1] Although the operating systems and distributed systems literature often use the terms consistency and coherence interchangeably, the architecture literature is more precise [17]: consistency semantics constrain the order that updates across multiple objects become observable to readers. Coherence semantics constrain the order that updates to a single object become observable but do not additionally constrain the ordering of updates across different objects. We find this precision useful and follow that terminology in this paper.

1. In order to allow partial replication of data, our design *separates 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. Distinct invalidation messages are widely used in hierarchical caching systems, but we demonstrate how to use them in topology-independent systems: we develop explicit synchronization rules to enforce consistency constraints despite multiple streams of information, and we introduce general mechanisms for handling demand read misses.
2. In order to allow partial replication of update metadata, we use *explicit conservative encoding* via *imprecise invalidations*, which allow a single invalidation to summarize a set of invalidations.
3. In order to allow our system to serve as a flexible toolkit for constructing a broad range of systems, our implementation cleanly *separates mechanism from policy* by splitting the system into a core that defines a node's mechanisms for maintaining local state and a controller that embodies a system's policies for communication among nodes.

We have constructed and evaluated a prototype. Our primary conclusion is that by disentangling mechanism from policy and simultaneously supporting the three PRACTI properties, *PRACTI replication enables better trade-offs for system designers than possible with existing mechanisms.* For example, for some workloads in our mobile storage and grid computing case studies, our system dominates existing approaches by providing more than an order of magnitude better bandwidth and storage efficiency than AC-TI replicated server systems, by providing more than an order of magnitude better synchronization delay compared to PR-AC hierarchical systems, and by providing consistency guarantees not achievable by PR-TI per-object replication systems.

More broadly, by subsuming a large portion of the design space, the PRACTI architecture can simplify the design of replication systems. At present, because 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. In contrast, our system can be viewed as a "replication microkernel" that defines a common substrate of core mechanisms over which a broad range of systems can be constructed by selecting appropriate policies. For example, in this study we use our prototype both to emulate existing server replication, client-server, and object replication systems and to implement novel policies to support mobile users, web edge servers, and grid scientific computing.

In summary, this paper makes four contributions. First, it defines the PRACTI paradigm and shows how existing systems fail to provide all of the desired properties. Second, it describes the first replication architecture to simultaneously provide all three PRACTI properties. Third, it provides a prototype PRACTI replication toolkit that cleanly separates mechanism from policy and thereby allows nearly arbitrary replication, consistency, and topology policies. Fourth, it demonstrates that PRACTI replication offers decisive practical advantages compared to existing approaches.

Section 2 of this paper explores the limitations of existing approaches, Section 3 describes our PRACTI architecture, and Section 4 experimentally evaluates the prototype. Finally, Section 5 surveys related work and Section 6 highlights our conclusions.

2 Background

Although providing all three PRACTI properties has obvious potential benefits, we know of no system that does so. Most existing systems fall into three categories that each provide at most two of the PRACTI properties:

Server replication systems like Bayou [30] provide log-based peer-to-peer update exchange that allows any node to send updates to any other node (TI) and that consistently orders writes. TACT [47] uses this approach to provide a wide range of tunable consistency guarantees (AC). Unfortunately, these protocols fundamentally assume full replication: all nodes store all data from any volume they export and all nodes receive all updates. As a result, these systems are unable to exploit workload locality to efficiently use networks and storage, and they may be unsuitable for devices with limited resources.

Client server [18, 29] and *hierarchical* [6, 27] caching systems permit caching of arbitrary subsets of data (PR), and existing cache consistency protocols can provide a wide range of consistency guarantees (AC). However, these protocols fundamentally require communication to flow between a child and its parent. Even when client-server systems permit limited client-client communication for cooperative caching [12] they must still serialize control messages at a central server for consistency [8]. These restricted hierarchical communication patterns (1) hurt performance when network topologies do not match the fixed communication patterns or when network costs change over time (e.g., in environments with mobile nodes), (2) hurt availability when a network path or node failure disrupts a fixed communication topology, and (3) limit the ability to support sharing during disconnected operation when a set of nodes can communicate with one another but not with the rest of the system.

Object replication systems [16, 34] allow nodes to choose arbitrary subsets of data to store (PR) and arbitrary peers with whom to communicate (TI). But, these protocols enforce no ordering constraints on updates across multiple objects, so they can provide coherence

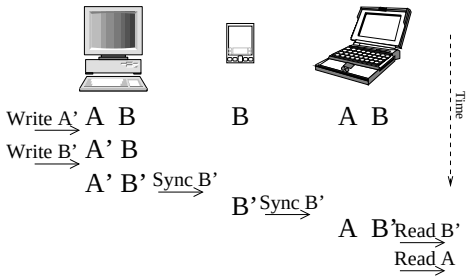


Fig. 1: Naive addition of PR to an AC-TI log exchange protocol fails to provide consistency.

but not consistency guarantees. Unfortunately, reasoning about the corner cases of consistency protocols is complex, so providing weak consistency or coherence guarantees can complicate constructing, debugging, and using applications. Furthermore, in some cases support for only weak consistency semantics may prevent deployment of applications with more stringent requirements.

It is surprising that despite the significant costs of omitting any of the PRACTI properties, no system has succeeded in providing all three. Our analysis of these protocols suggests that these limitations are fundamental to these protocol families: the assumption of full replication is deeply embedded in the core of Bayou and other server replication protocols; the assumption of hierarchical communication is fundamental to client-server consistency protocols; and the lack of consistency is a key factor in the flexibility of object replication systems.

Example. To understand challenges of providing PRACTI, consider the naive attempt to add PR to a AC-TI server replication protocol like Bayou illustrated in Figure 1. Suppose a user's desktop node stores all of the user's files, including files A and B, but the user's palmtop only stores a small subset that includes B but not A. Then, the desktop issues a series of writes, including a write to file A (making it A') followed by a write to file B (making it B'). When the desktop and palmtop synchronize, for PR, the desktop sends the write of B but not the write of A. At this point, everything is OK: the palmtop and desktop have exactly the data they want, and reads of local data provide a consistent view of the order that writes occurred. But for TI, we not only have to worry about local reads but also propagation of data to other nodes. For instance, suppose that the user's laptop, which also stores all of the user's files including both A and B, synchronizes with the palmtop: the palmtop can send the write of B but not the write of A. Unfortunately, the laptop now can present an inconsistent view of data to a user or application. In particular, a sequence of reads at the laptop can return the new version of B and then return the old version of A, which is inconsistent with the writes that occurred at the desktop under causal [19] or even the weaker FIFO consistency [26].

As this example illustrates, topology independence

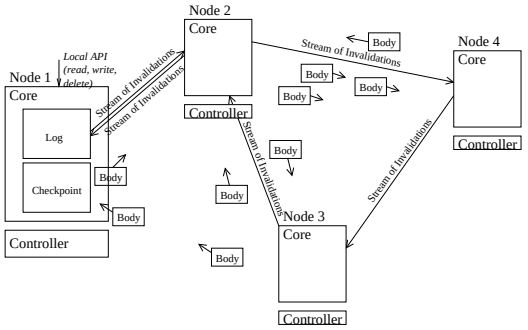


Fig. 2: High level PRACTI architecture.

makes combining partial replication and arbitrary consistency hard because when a node receives updates, it must not only consistently order updates to the data it cares about but also ensure that it has enough information to order updates *for the data of interest to all nodes with which it might communicate in the future.*

Existing systems resolve this dilemma in one of three ways. AC-TI server replication systems' full replication ensures that all nodes have enough information to order all updates. PR-AC client-server and hierarchical systems restrict communication so that the root of a subtree can track what information is cached by descendents and can safely omit sending invalidations or updates for data that no descendent is currently caching; if a descendent later tries to read such data, cache miss bubbles up the hierarchy to a node that has sufficient information to supply consistent data to the read. Finally, PR-TI object replication systems simply give up ability to consistently order writes to different objects and allow inconsistencies such as the one just described.

3 PRACTI replication

Figure 2 shows the high-level architecture of our PRACTI architecture. Each node exports a *Local API* for reading and writing data, and each node stores an arbitrary subset of data using a *Log* of updates and a *Checkpoint* for random access. Furthermore, any node can exchange information with any other node at any time.

The architecture is based on three key ideas described in more detail in the following subsections:

1. *Separation of invalidations and bodies.* As Figure 2 illustrates, nodes exchange two types of updates: ordered streams of invalidations and unordered body messages. This separation supports partial replication of bodies.
2. *Partial replication of invalidation metadata.* Although the invalidation streams of Figure 2 each logically contain a causally consistent record of all writes in the system, the implementation can omit sending groups of invalidations by sending *imprecise invalidation* summaries instead. Imprecise invalidations allow partial replication of metadata and flexible consistency.

3. *Separation of mechanism and policy.* As Figure 2 illustrates, our implementation of each node comprises a *core* and a *controller.* The *core* instantiates the basic PRACTI *mechanisms* by processing incoming messages and maintaining a local view of the system's state. The *controller* embodies a system's *policies* by initiating communication among nodes. Different deployments use different controllers to implement different replication, topology, and consistency policies.

The rest of this section details these three aspects of the design. It then discusses the crosscutting issue of how to provide flexible consistency that (a) supports strong consistency semantics for those applications that require them and (b) does not introduce unnecessary overhead for applications that do not. After that, Section 3.5 describes several novel features that enable our prototype to support the broadest range of policies.

3.1 Separation of invalidations and bodies

As Figure 2 illustrates, nodes exchange two types of updates: ordered streams of invalidations and unordered body messages. *Invalidations* are metadata that describe writes; each contains an object ID[2] and logical time of a write. A write's logical time is assigned at the local interface that first receives the write, and it contains the current value of the node's Lamport clock [24] and the node's ID. Like invalidations, *body messages* contain the write's object ID and logical time, but they also contain the actual contents of the write.

The protocol for exchanging updates is simple.

- As illustrated by node 1 in Figure 2, each node maintains a *log* of the invalidations it has received sorted by logical time. And, for random access, each node stores bodies in its *checkpoint* indexed by object ID.
- Invalidations from a log are sent via a causally-ordered stream that logically contains all invalidations known to the sender but not to the receiver. As in Bayou, nodes use version vectors to summarize the contents of their logs in order to efficiently identify which updates in a sender's log are needed by a receiver [30].
- A receiver of an invalidation inserts the invalidation into its sorted log and updates its checkpoint. Checkpoint update of the entry for object ID entails marking the entry *INVALID* and recording the logical time of the invalidation. Note that checkpoint update for an incoming invalidation is skipped if the checkpoint entry already stores a logical time that is at least as high as the incoming invalidation's.
- A node can send any body from its checkpoint to any other node at any time. When a node receives a body, it updates its checkpoint entry by first checking to see if the entry's logical time matches the body's logical time and, if so, storing the body in the entry and marking the entry *VALID*.

Rationale. Separating invalidations from bodies provides topology-independent protocol that supports both arbitrary consistency and partial replication.

Supporting arbitrary consistency requires a node to be able to consistently order all writes. Log-based invalidation exchange meets this need by ensuring three crucial properties [30]. First the *prefix property* ensures that a node's state always reflects a prefix of the sequence of invalidations by each node in the system. I.e., if a node's state reflects the ith invalidation by some node n in the system, then the node's state reflects all earlier invalidations by n. Second, each node's local state always reflects *causally consistent* [19] view of all invalidations 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 the invalidation for write w, all of its subsequent writes' logical timestamps will exceed w's. Third, the system ensures *eventual consistency*: all connected nodes eventually agree on the same total order of all invalidations. This combination of properties provides the basis for a broad range of tunable consistency semantics using standard techniques [47].

At the same time, this design supports partial replication by allowing bodies to be sent to or stored on any node at any time. It supports arbitrary body replication policies including demand caching, push-caching, prefetching, pre-positioning bodies according to a global placement policy, or push-all.

Design issues. The basic protocol adapts well-understood log exchange mechanisms [30]. But, the separation of invalidations and bodies raises two new issues: (1) coordinating disjoint streams of invalidations and bodies and (2) handling reads of invalid data.

The first issue is how to coordinate the separate body messages and invalidation streams to ensure that the arrival of out-of-order bodies does not break the consistency invariants established by the carefully ordered invalidation log exchange protocol. The solution is simple: when a node receives a body message, it does not apply that message to its checkpoint until the corresponding invalidation has been applied. A node therefore buffers body messages that arrive "early." As a result, the checkpoint is always consistent with the log, and the flexible consistency properties of the log [47] extend naturally to the checkpoint despite its partial replication.

The second issue is how to handle demand reads at nodes that replicate only a subset of the system's data. The core mechanism supports a wide range of policies: by default, the system blocks a local read request until

[2]For simplicity, we describe the protocol in terms of full-object writes. For efficiency, our implementation actually tracks checkpoint state, invalidations, and bodies on arbitrary byte ranges.

Fig. 3: Example imprecise invalidation.

the requested object's status is *VALID*[3]. Of course, to ensure liveness, when an *INVALID* object is read, an implementation should arrange for someone to send the body. As we describe in more detail in Section 3.3, when a local read blocks, the core notifies the controller. The controller can then implement any policy for locating and retrieving the missing data such as sending the request up a static hierarchy (i.e., ask your parent or a central server), querying a separate centralized [13] or DHT-based [39] directory, using a hint-based search strategy [35], or relying on a push-all strategy [30] (i.e., "just wait and the data will come.")

3.2 Partial replication of invalidations

Although separation of invalidations from bodies supports partial replication of bodies, for true partial replication that supports a broad range of policies, workloads, and devices the system must not require all nodes to see all invalidations or to store metadata for each object. For example, consider palmtops caching data from an enterprise file system with 10,000 users and 10,000 files per user: if each palmtop were required to store 100 bytes of per-object metadata, then 10GB of storage would be consumed on each device; and if the palmtops were required to receive every invalidation during log exchange and if an average user issued just 100 updates per day, then invalidations would consume 100MB/day of bandwidth to each device. Exploiting locality is fundamental to replication in large-scale systems, and requiring full replication of metadata would prevent deployment of a replication system for a wide range of environments, workloads, and devices.

To support true partial replication, invalidation streams *logically* contain all invalidations as described in Section 3.1, but in *reality* they omit some invalidations by replacing them with *imprecise invalidations.*

As Figure 3 illustrates, an imprecise invalidation is a conservative summary of several standard or *precise invalidations.* Each imprecise invalidation has a *targetSet* of objects, *start* logical time, and an *end* logical time, and it means "one or more objects in *targetSet* were updated between *start* and *end*." An imprecise invalidation must be *conservative*: each precise invalidation that it replaces must have its *objId* included in *targetSet* and must have its logical *time* included between *start* and *end*, but for efficient encoding *targetSet* may include additional objects. In our prototype, the *targetSet* is encoded as a list of subdirectories and the *start* and *end* times are partial version vectors with an entry for each node whose writes are summarized by the imprecise invalidation.

Imprecise invalidations act as "placeholders" in the log to ensure that nodes that try to access data updated by omitted writes can detect and correct the missing information. When a node receives a new imprecise invalidation, it logically marks all covered objects "INVALID." For efficiency, however, rather than iterating through all covered objects, the implementation uses some additional bookkeeping to efficiently track local state.

Design issues. Tracking the effects of imprecise invalidations actually encompasses four related problems:

1. We cannot require a node to store per-object state for all objects. As the example above illustrates, doing so would significantly restrict the range of replication policies and workloads that can be accommodated.
2. We need to efficiently apply imprecise invalidations covering many objects. In particular, an implementation should not have to iterate across all objects in *targetSet* to apply an imprecise invalidation.
3. We need to be able to determine when objects whose state was "made IMPRECISE" by one or more imprecise invalidation have been "made PRECISE" by later seeing all of the missing precise invalidations for those objects.
4. We need to handle demand reads to objects whose state is currently IMPRECISE.

Our solution is to maintain simple bookkeeping information about groups of objects. In particular, each node independently partitions the object ID space into one or more *interest sets* and decides whether to store per-object state on a per-interest set basis. A node tracks whether each interest set is PRECISE (has observed all invalidations) or IMPRECISE (has overlapped some imprecise invalidations and may have missed some precise invalidations) by maintaining two pieces of state.

- Each node maintains a global variable *currentVV*, which is a version vector encompassing the highest timestamp of any invalidation (precise or imprecise) applied to any interest set.
- Each node maintains for each interest set IS the variable $IS.lastPreciseVV$, which is the latest version vector for which IS is known to be PRECISE.

If $IS.lastPreciseVV = currentVV$, then interest set IS has not missed any invalidations and it is PRECISE.

In this arrangement, applying an imprecise invalidation I to an interest set IS merely involves updating two variables—the global $currentVV$ and the interest set's $IS.lastPreciseVV$. In particular, a node that receives imprecise invalidation I always advances $currentVV$ to

[3]The read interface also provides a flag that indicates that a read of an INVALID object should throw an exception rather than block.

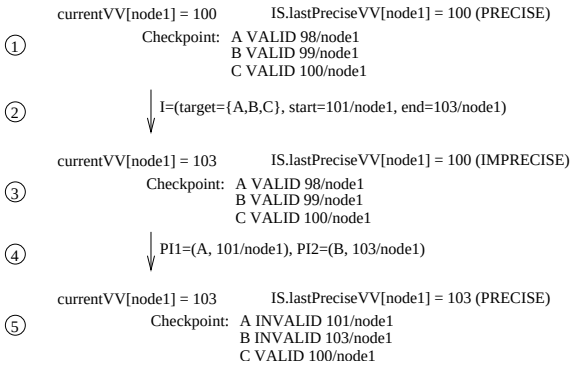


Fig. 4: Example of applying an imprecise invalidation I and then applying precise invalidations $PI1$ and $PI2$. For clarity, we only show node1's elements of $currentVV$ and $IS.lastPreciseVV$.

include I's end logical time because after applying I, the system's state may reflect events up to $I.end$. Conversely, the node only advances $IS.lastPreciseVV$ to the latest time for which IS has missed no invalidations.

This per-interest state addresses the four problems listed above. (1) Storage is limited: each node only needs to store per-object state for data currently of interest to that node, and the total metadata state at a node is proportional to the number of objects of interest plus the number of interest sets. Note that our implementation allows a node to dynamically repartition its data across interest sets as its locality patterns change. (2) Imprecise invalidations are efficient to apply, requiring work that is proportional to the number of interest sets rather than the number of summarized invalidations. (3) Recovery to precise is guaranteed under the following conditions: if an interest set IS is initially PRECISE at a node, the node then sees an imprecise invalidations I that make an interest set IS IMPRECISE, and later the node sees the a sequence of precise invalidations that includes all invalidations in I that target any object in IS, then the interest set IS is made PRECISE up to at least the end time of I. (4) A local read request includes a flag that indicates whether the read requires consistency guarantees. If not, then the read does not consult the per interest set status and it may return as soon as the object is VALID. Conversely, if the read does require consistency, then the read blocks until the interest set in which the object lies is PRECISE. This blocking ensures that "precise reads" only observe the checkpoint state they would have observed if all invalidations were precise, and therefore allows them to enforce the same consistency as protocols without imprecise invalidations. As with regular read misses, for liveness the core signals the controller when a read of an IMPRECISE interest set blocks, and the controller is responsible for arranging for the missing precise invalidations to be sent.

The following example illustrates the maintenance of the interest set status state in more detail.

Example. Suppose that initially as label (1) in Fig. 4 illustrates, A, B, and C were last written at node1's logical times $98/node1$, $99/node1$, and $100/node1$, that all are currently VALID, and that interest set IS containing A, B, and C is PRECISE with *IS.lastPreciseVV[node1] = currentVV[node1] = 100*.

Then, (2) an imprecise invalidation *I* with a *targetSet* that includes A, B, and C, a *start* time of *101/node1*, and an *end* time of *103/node1* arrives. The system must conservatively assume A, B, and C are all invalid up to time *103/node1*, so (3) it sets *currentVV[node1] = 103* but leaves *IS.lastPreciseVV[node1] = 100*, making *IS* IMPRECISE.

But now (4) suppose precise invalidations *PI1 = (A, 101/node1)* and *PI2 = (B, 103/node1)* arrive on a single invalidation channel from another node. (5) The first invalidation advances *IS.lastPreciseVV[node1]* to *101* and leaves *currentVV* unchanged. The second advances *IS.lastPreciseVV[node1]* to *103*, and the final state is *IS.lastPreciseVV[node1] = currentVV[node1] = 103*, *IS* is PRECISE, *A* and *B* are INVALID, and *C* is VALID.

Notice that although we never saw a precise invalidation with time *102/node1*, the fact that a single stream contains invalidations at times *101/node1* and *103/node1* allows us to infer by the prefix property that no invalidation at time *102/node1* occurred and therefore we were able to advance *IS.lastPreciseVV* to make IS PRECISE.

A technical report [11] provides pseudo-code and details how our implementation copes with (a) applying invalidations in causal order despite the multiple start and end times in imprecise invalidations and despite concurrency across streams and (b) maximizing the information extracted and stored from each invalidation in a stream to minimize the amount of IMPRECISE data in the system.

3.3 Separation of mechanism and policy

Our goal is to construct a toolkit that not only subsumes server replication, client/server, and object replication systems, but one that also makes it easy to construct new systems that explore new replication, topology, and consistency policies. Our system therefore seeks to serve as a "replication microkernel" that provides basic low level mechanisms over which higher-level services can be built. As Figure 2 illustrates, we achieve this goal by splitting each node into a *core* and a *controller*.

The core's mechanisms enforce their safety properties regardless of what incoming messages they see. Our cores use 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. Any machine can therefore send any legal protocol message to any other machine at any time.

The controller implements policies that focus on liveness (e.g., performance and availability.) The controller's basic job is to ensure that the right cores send useful data at the right times in order to do such things as satisfy a read miss, prefetch data to improve performance, or provision a node's local storage for disconnected operation. Controllers accomplish this work by sending requests to trigger communication between cores.

The controller is defined by its interface. Within this interface, different implementations provide different policies. Controllers use three sets of interfaces to accomplish their work: a core calls a controller's *inform* interface to inform the controller of important local events like message arrival or read miss, a controller calls a remote core's *remote request* interface to trigger sends of invalidation streams or bodies, and a controller calls its core's *management* interface for maintenance functions like log garbage collection and interest set split/join.

Enumerating the full API is outside the scope of this paper. To provide a high-level understanding of the workings of the system, the remainder of this subsection describes the typical control flow for two examples: a read miss and a subscription for an invalidation stream. Additionally, Section 4 briefly describes several example controllers we have built.

Read miss example. A local read blocks in the core until the specified object is VALID. If it is INVALID, the core calls *Controller.informDemandReadMiss(objId)*. The controller then typically selects a peer from which to fetch the data, transmits a demand read request to that peer's core, and sets a retry timer. When the body arrives, the local core applies it to the the checkpoint, unblocks the waiting read, and calls *Controller.informDemandReply(objId)*, and the controller cancels the retry timer.

Subscribe invalidations example. A controller chooses one or more peers from which to receive invalidations according to some policy. For each remote peer, the controller transmits a *subscribeInval* request, specifying for each the *preciseSet* of the object ID space for which it would prefer to receive precise invalidations from that peer and the *startVV* logical time at which the invalidation stream should begin. Typically, *startVV* is one of two things: (1) the local node's *currentVV* version vector of the highest invalidations seen or (2) the *IS.lastPreciseVV* version vector of some currently-IMPRECISE interest set IS that the local node is trying to make PRECISE. The controller also sets an internal retry timer for each such subscription request.

Upon receiving a connection from the remote peer, the local core calls *Controller.informInvalStreamInitiated(senderNode, preciseSet, startVV)* and the controller cancels its retry timeout. Also, as each invalidation arrives on such a stream, the core calls *Controller.informReceiveInval(...)*, which some controllers use to track which objects are VALID/INVALID in the local checkpoint.

3.4 Consistency: Costs and approach

Enforcing cache consistency entails fundamental tradeoffs. For example Siegel [36] proves what has come to be known as the CAP dilemma [7]: 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 [26].

A system that seeks to support arbitrary consistency must therefore do two things. First, it must allow a range of consistency guarantees to be enforced. Second, it must ensure that workloads only pay for the consistency guarantees they actually need.

Our system addresses these goals by distinguishing the availability and response time costs paid by read and write requests from the bandwidth overhead paid by invalidation propagation.

The read interface allows each read request to specify its consistency requirements. Therefore, a read does not block unless *that read* requires the local node to gather more recent invalidations and updates than it already has. Similarly, most writes complete locally, and a write only blocks to synchronize with other nodes if *that write* requires it. Therefore, the performance/availability versus consistency dilemmas are resolved on a per-read, per-write basis [47].

Conversely, all invalidations that propagate through the system must carry with them sufficient information that a later read can get whatever consistency level it requests. Therefore, the system may pay an extra cost: if a deployment never needs strong consistency, then our protocol will propagate some information that is never needed. We believe this cost is acceptable for two reasons: (1) other features of the PRACTI design—separation of invalidations from bodies and imprecise invalidations—minimize the amount of extra data transferred; and (2) we believe the bandwidth costs of consistency are less important than the availability and response time costs. Our experimental evaluation in Section 4 quantifies these bandwidth costs, and we argue that they are insignificant.

Implementation. Because our design uses a variation of peer-to-peer log exchange [30], adapting flexible consistency techniques from the literature is straightforward. We provide the TACT flexible consistency interface to bound order error and temporal error [47]; we have not yet implemented TACT numerical error, but we see no

fundamental barriers. Additionally, we include the option of a two phase write that first distributes invalidations and later distributes bodies [23, 47]; using this optional interface, one can ensure that once a write returns, no subsequent read can return the data's old value and that once a read returns the new value no read will return the old value. Additionally, as described above, an *imprecise read* skips consistency checks and provides causal coherence (ordering of updates for a single item) rather than causal consistency. Finally, we provide a general interface for detecting and resolving write-write conflicts according to application-specific semantics [21, 30].

3.5 Additional features

Three novel aspects of our implementation further our goal of constructing a flexible framework that can accommodate the broadest range of policies. First, our implementation allows systems to use any desired policy for limiting the size of their logs and to fall back on an efficient *incremental checkpoint transfer* to transmit updates that have been garbage collected from the log. This feature both limits storage overheads and improves support for synchronizing intermittently connected devices. Second, our implementation uses *self-tuning body propagation* to enable prefetching policies that are simultaneously aggressive and safe. Third, our implementation provides *incremental log exchange* to allow systems to minimize the window for conflicting updates. Due to space constraints, we will only briefly outline these aspects of the implementation.

Garbage collection and incremental checkpoint transfer. Imprecise invalidations yield an unexpected benefit: incremental checkpoint transfer.

Nodes can garbage collect any prefix of their logs, which allows each node to bound the amount local storage used for the log to any desired fraction of its total disk space. But, if a node *n1* garbage collects log entries older than *n1.omitVV* and another node *n2* requests a log exchange beginning before *n1.omitVV*, then *n1* cannot send a stream of invalidations. Instead, *n1* sends a checkpoint of its per-object state to bring *n2*'s state up to *n1.currentVV*.

In existing server replication protocols [30], in order to ensure consistency, such a checkpoint exchange must atomically update $n2$'s state for all objects in the system. Checkpoint exchange, therefore, may block interactive requests for a long period of time while the checkpoint is atomically assembled at $n1$ or applied at $n2$ and may waste system resources if a checkpoint transfer is started but fails to complete.

Rather than transferring information about all objects, our incremental checkpoints can update arbitrary interest sets. As Figure 5 illustrates, each incremental checkpoint

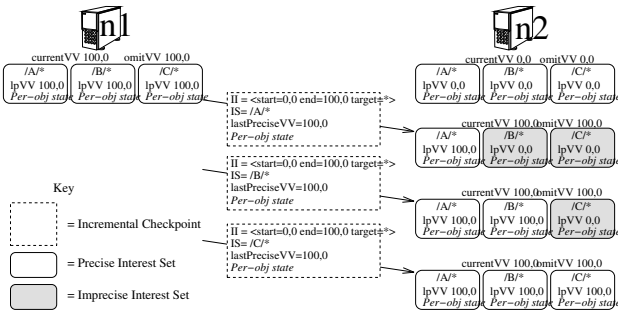


Fig. 5: Incremental checkpoints from $n1$ to $n2$.

includes (1) an imprecise invalidation that covers all objects from the receiver's *currentVV* up to the sender's *currentVV*, (2) interest set state for interest set *IS* (*IS.lastPreciseVV*), and (3) per-object logical timestamps for all objects in interest set *IS* that were invalidated later than the receiver's *IS.lastPreciseVV*. The receiver's *currentVV*, *IS.lastPreciseVV*, and per-object state are thus brought up to include the updates known to the sender.

Overall, this approach makes checkpoint transfer a much smoother process under PRACTI than under existing protocols: the receiver can receive an incremental checkpoint for a small portion of its ID space and then either background fetch checkpoints of other interest sets or fault them in "on demand" as Figure 5 illustrates.

Self-tuning body propagation. In addition to supporting demand-fetch of particular objects, our prototype provides a novel self-tuning prefetching mechanism. A node $n1$ subscribes to updates from a node $n2$ by sending a list L of directories of interest along with a $startVV$ version vector. $n2$ will then send $n1$ any bodies it sees that are in L and that are newer than $startVV$. To do this, $n2$ maintains a priority queue of pending sends: when a new eligible body arrives, $n2$ deletes any pending sends of older versions of the same object and then inserts a reference to the updated object. This priority queue drains to $n1$ via a low-priority network connection that ensures that prefetch traffic does not consume network resources that regular TCP connections could use [40]. When a lot of "spare bandwidth" is available, the queue drains quickly and nearly all bodies are sent as soon as they are inserted. But, when little "spare bandwidth" is available, the buffer sends only high priority updates and absorbs repeated writes to the same object.

Incremental log propagation. The PRACTI prototype implements a novel variation on existing batch log exchange protocols. In particular, in the batch log exchange used in Bayou, a node first receives a batch of updates comprising a start time $startVV$ and a series of writes, it then rolls back its checkpoint to before $startVV$ using an undo log, and finally it rolls forward, merging the newly received batch of writes with its existing redo log and applying updates to the checkpoint. In contrast, our

incremental log exchange applies each incoming write to the current checkpoint state without requiring roll-back and roll-forward of existing writes [11].

The advantages of the incremental approach are efficiency (each write is only applied to the checkpoint once), concurrency (a node can process information from multiple continuous streams), and consistency (connected nodes can stay continuously synchronized which reduces the window for conflicting writes.) The disadvantage is that it only supports simple conflict detection logic: for our incremental algorithm, a node detects a write/write conflict when an invalidation's *prevAccept* logical time (set by the original writer to equal the logical time of the overwritten value) differs from the logical time the invalidation overwrites in the node's checkpoint. Conversely, batch log exchange supports more flexible conflict detection: Bayou writes contain a *dependency_check* procedure that can read any object to determine if a conflict has occurred [38]; this works in a batch system because rollback takes all of the system's state to a specified moment in time at which these checks can be re-executed. Note that this variation is orthogonal to the PRACTI approach: a full replication system such as Bayou could be modified to use our incremental log propagation mechanism, and a PRACTI system could use batch log exchange with roll-back and roll-forward.

4 Evaluation

We have constructed a prototype PRACTI system written in Java and using BerkeleyDB [37] for per-node local storage. The prototype is fully functional but not performance tuned. All features described in this paper are implemented including local read/write/delete, flexible consistency, incremental log exchange, remote read and prefetch, garbage collection of the log, incremental checkpoint transfer between nodes, and crash recovery. We have also constructed several example controllers in order to emulate existing server replication, client-server, and object replication systems and to implement and evaluate novel policies to support mobile users, web edge servers, and grid scientific computing.

In this section we evaluate the properties of our prototype to answer two questions.

1. *Does a PRACTI architecture offer significant advantages over existing replication protocols?* We find that our PRACTI system can dominate existing approaches by providing more than an order of magnitude better bandwidth and storage efficiency than replicated server systems, as much as an order of magnitude better synchronization delay compared to hierarchical systems, and consistency guarantees not achievable by per-object replication systems. Furthermore, even in environments for which these existing policies suffice, our flexible architecture can subsume these existing approaches.
2. *What are the costs of PRACTI's generality?* In particular, is it significantly more expensive to implement a given system using PRACTI than to implement it using narrowly-focused specialized mechanisms? We find that the primary "extra" cost of PRACTI's generality is that our system might transmit more consistency information than a customized system might require. But, our implementation reduces this cost compared to past systems via separating invalidations and bodies and via imprecise invalidations, so these costs appear to be minor.

To provide a framework for exploring these issues, we first focus on partial replication in 4.1. We then examine topology independence in 4.2. Finally, we examine the costs and benefits of flexible consistency in 4.3.

4.1 Partial replication

In this section, we focus on partial replication. We find that PRACTI's support for partial replication dramatically improves performance compared to full replication protocols from which our system descends for three reasons:

1. *Locality of Reference:* partial replication of bodies and invalidations can *each* reduce storage and bandwidth costs by an order of magnitude for nodes that care about only a subset of the system's data.
2. *Bytes Die Young:* partial replication of bodies can significantly reduce bandwidth costs when "bytes die young" [4].
3. *Self-tuning Replication:* self-tuning replication minimizes response time for a given bandwidth budget.

It is not a surprise that partial replication can yield significant performance advantages over existing server replication systems. What is significant is that (1) these experiments provide evidence that despite the the good properties of server replication systems (e.g., support for disconnected operation, flexible consistency, and dynamic network topologies) these systems may be impractical for many environments and (2) they demonstrate that these trade-offs are not fundamental—a PRACTI system can support PR while retaining the good AC-TI properties of server replication systems.

Locality of reference. Different devices in a distributed system often access different subsets of the system's data because of locality and different hardware capabilities. In such environments, some nodes may access 10%, 1%, or less of the system's data, and partial replication may yield significant improvements in both bandwidth to distribute updates and space to store data.

Figure 6 examines the impact of locality on replication cost for three systems implemented on our PRACTI core

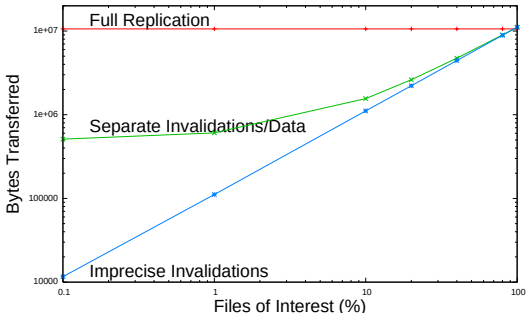


Fig. 6: Impact of locality on replication cost.

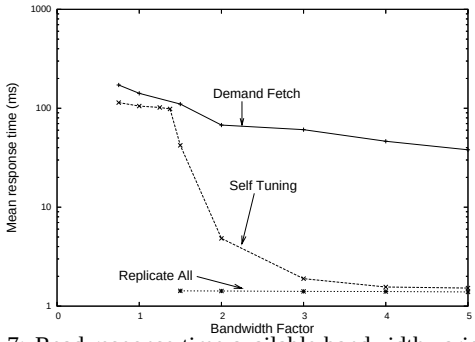


Fig. 7: Read response time available bandwidth varies for full replication, demand reads, and self-tuning replication.

	Storage	Dirty Data	Wireless	Internet
Office server	1000GB	100MB	10Mb/s	100Mb/s
Home desktop	10GB	10MB	10Mb/s	1Mb/s
Laptop	10GB	10MB	10Mb/s 1Mb/s	50Kb/s Hotel only
Palmtop	100MB	100KB	1Mb/s	N/A

Fig. 8: Configuration for mobile storage experiments.

using different controllers: a full replication system similar to Bayou, a partial-body replication system that sends all precise invalidations to all nodes but that only sends some bodies to a node, and a partial-replication system that sends some bodies and some precise invalidations to a node but that summarizes other invalidations using imprecise invalidations. In this benchmark, we overwrite a collection of 1000 files of 10KB each. A node subscribes to invalidations and body updates for the subset of the files that are "of interest" to that node. The x axis shows the fraction of files that belong to a node's subset, and the y axis shows the total bandwidth required to transmit these updates to the node as measured on the prototype.

The results show that partial replication of both bodies and invalidations is crucial when nodes exhibit locality. Partial replication of bodies yields up to an order of magnitude improvement, but it is then limited by full replication of metadata. Our true PRACTI system, however, can gain over another order of magnitude as locality increases via its use of imprecise invalidations.

Note that Figure 6 shows bandwidth costs. Partial replication provides similar improvements for space requirements (graph omitted for space.)

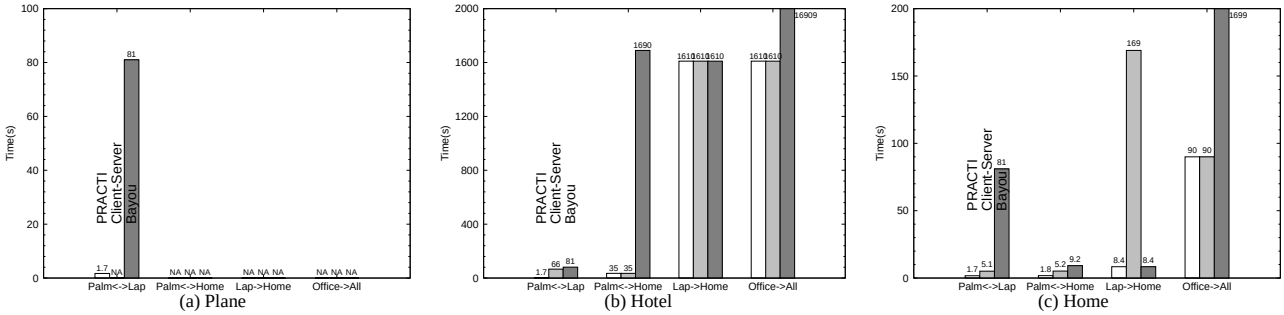
Bytes die young. Bytes are often overwritten or deleted soon after creation [4]. Full replication systems send all writes to all servers, even if some of the writes are quickly made obsolete. In contrast, PRACTI replication can send invalidations separately from bodies: if a file is written multiple times on one node before being read on another, overwritten bodies need never be sent.

To examine this effect, we randomly write a set of files on one node and randomly read the same files on another node. Due to space constraints, we defer the graph to the extended report [11]. To summarize: when the write to read ratio is 2, PRACTI uses 55% of the bandwidth of full replication, and when the ratio is 5, PRACTI uses 24%.

Self-tuning replication. Separation of invalidations from bodies enables a novel self-tuning data prefetching mechanism described in Section 3. As a result, systems can replicate bodies aggressively when network capacity is plentiful and replicate less aggressively when network capacity is scarce.

Figure 7 illustrates the benefits of this approach by evaluating three systems that replicate a web service from a single origin server to multiple edge servers. In the *dissemination services* [28] this system hosts, all updates occur at the origin server and all client reads are processed at edge servers, which serve both static and dynamic content. We compare the read response time observed by the edge server when accessing the database to service client requests for three replication policies: *Demand Fetch*, implemented as a client-server system, replicates precise invalidations to all nodes but sends new bodies only in response to demand requests, *Replicate All* follows a Bayou-like approach and replicates both precise invalidations and all bodies to all nodes, and *Self Tuning* exploits PRACTI to replicate precise invalidations to all nodes and to have all nodes subscribe for all new bodies via the self-tuning mechanism. We use a synthetic workload where the read:write ratio is 1:1, reads are Zipf distributed across files (α = 1.1), and writes are uniformly distributed across files. We use Dummynet to vary the available network bandwidth from 0.75 to 5.0 times the system's average write throughput.

As Figure 7 shows, when spare bandwidth is available, self-tuning replication improves response time by up to a factor of 20 compared to *Demand-Fetch*. A key challenge, however, is preventing prefetching from overloading the system. Whereas our self-tuning approach adapts bandwidth consumption to available resources, *Replicate All* sends all updates regardless of workload or environment. This makes *Replicate All* a "poor neighbor"—it consumes bandwidth corresponding to the current write rate for prefetching even if other applications could make better use of the network.



(a) Plane (b) Hotel (c) Home

Fig. 9: Synchronization time among devices for different network topologies and protocols.

4.2 Topology independence

In this section we examine topology independence by examining two environments, a mobile data access system that is distributed across multiple devices and a wide-area-network file system designed to make it easy for PlanetLab and Grid researchers to run experiments that rely on distributed state. In both cases, PRACTI's combined partial replication and topology independence allows our design to dominate topology-restricted hierarchical approaches by doing two things:

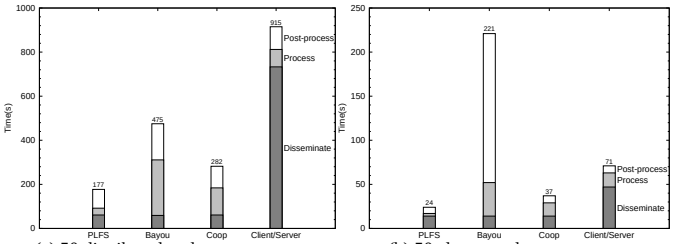
1. *Adapt to changing topologies*: a PRACTI system can make use of the best paths among nodes.
2. *Adapt to changing workloads*: a PRACTI system can optimize communication paths to, for example, use direct node-to-node transfers for some objects and distribution trees for others.

For completeness, our graphs also compare against topology-independent, full replication systems; the data indicate that topology independence without partial replication is not an attractive alternative. Due to space limits, we do not comment further on this subset of the results.

Mobile storage. We evaluate PRACTI in the context of a mobile storage system that distributes data across palmtop, laptop, home desktop, and office server machines. We compare PRACTI to a client-server Coda-like system that supports partial replication but that distributes updates via a central server [21] and to a full-replication Bayou-like system that can distribute updates directly between interested nodes but that requires full replication [30]. All three systems are realized by implementing different controller policies.

As summarized in Figure 8 our workload models a department file system that supports mobility: an office server stores data for 100 users, a user's home machine and laptop each store one user's data, and a user's palmtop stores 1% of a user's data. Note that due to resource limitations, we store only the "dirty data" on our test machines, and we use desktop-class machines for all nodes; we control the network bandwidth of each scenario using a library that throttles transmission.

Figure 9 shows the time to synchronize dirty data among machines in three scenarios: (a) *Plane*: the user is on a plane with no Internet connection, (b) *Hotel*: the user's laptop has a 50Kb/s modem connection to the Internet, and (c) *Home*: the user's home machine has a 1Mb/s connection to the Internet. The user carries her laptop and palmtop to each of these locations and co-located machines communicate via wireless network at speeds indicated in Figure 8. For each location, we measure time for machines to exchange updates: (1) P↔L: the palmtop and laptop exchange updates, (2) P↔L: the palmtop and home machine exchange updates, (3) L→H: the laptop sends updates to the home machine, (4) O→All: the office server sends updates to all nodes.



(a) 50 distributed nodes + remote server (b) 50 cluster nodes + remote server

Fig. 10: Execution time for the WAN-Experiment benchmark.

In comparing the PRACTI system to a client-server system, topology independence has significant gains when the machines that need to synchronize are near one another but far from the server: in the isolated *Plane* location, the palmtop and laptop can not synchronize at all in a client-server system; in the *Hotel* location, direct synchronization between these two co-located devices is an order of magnitude faster than synchronizing via the server (1.7s v. 66s); and in the home location directly synchronizing co-located devices is between 3 and 20 times faster than client-server synchronization.

WAN-FS for Researchers. Figure 10 evaluates a wide-area-network file system called PLFS designed for PlanetLab and Grid researchers. The controller for PLFS is simple: for invalidations, PLFS forms a multicast tree to distribute all precise invalidations to all nodes. And, when an *INVALID* file is read, PLFS uses a DHT-based system [43] to find the nearest copy of the file; not only does this approach minimize transfer latency, it effectively forms a multicast tree when multiple concurrent

reads of a file occur [2, 39]. Like Shark [2], PLFS is designed to be convenient for allowing a user to export data from her local file system to a collection of remotely running nodes. However, unlike the read-only Shark system, PLFS supports read/write data.

We examine a 3-phase benchmark that represents running an experiment: in phase 1 *Disseminate*, each node fetches 10MB of new executables and input data from the user's home node; in phase 2 *Process*, each node writes 10 files each of 100KB and then reads 10 files from randomly selected peers; in phase 3, *Post-process*, each node writes a 1MB output file and the home node reads all of these output files. We compare PLFS to three systems: a client-server system, client-server with cooperative caching of read-only data (e.g., a Shark-like system [2]), and server-replication (e.g., a Bayou-like system [30]). All 4 systems are implemented over PRACTI.

Figure 10 shows performance for an experiment running on (a) 50 distributed nodes each with a 5.6Mb/s connection to the Internet (we emulate this case by throttling bandwidth) and (b) 50 "cluster" nodes at the University of Texas with a switched 100Mbit/s network among them and a shared path via Internet2 to the origin server at the University of Utah.

The speedups range from 1.5 to 9.2, demonstrating the significant advantages enabled by the PRACTI architecture. Compared to client/server, it is faster in both the Dissemination and Process phases due to its multicast dissemination and direct peer-to-peer data transfer. Compared to full replication, it is faster in the Process and Post-process phases because it only sends the required data. And compared to cooperative caching of read only data, it is faster in the Process phase because data is transferred directly between nodes.

4.3 Arbitrary consistency

This subsection first examines the benefits and then examines the costs of supporting flexible consistency.

Improved consistency trade-offs. Gray [15] and Yu and Vahdat [46] show a trade-off: aggressive propagation of updates dramatically improves consistency and availability but can also increase system load. PRACTI has three features that improve these trade-offs: (1) separation of invalidations from bodies allows invalidations to propagate aggressively, (2) streaming log exchange (rather than batch) allows nodes to continuously update one another when they are connected, and (3) self-tuning body propagation maximizes the amount of *VALID* data at a node for a given consistency requirement and bandwidth budget [28].

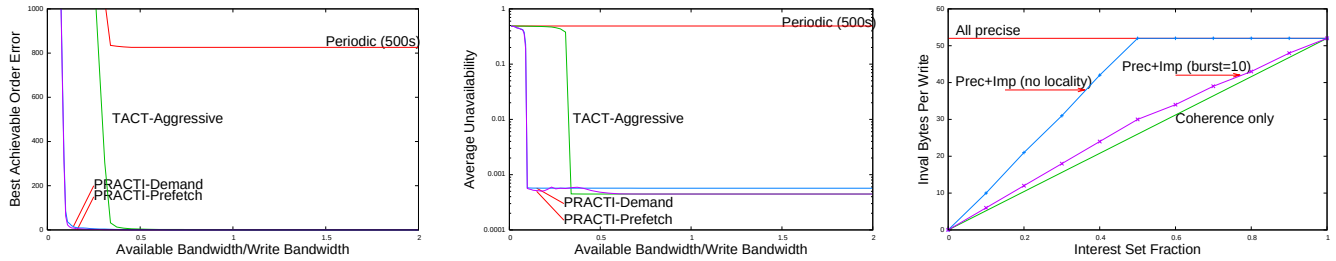
We examine a range of consistency requirements and network failure scenarios via simulation (all other experiments in this paper are prototype measurements.) We use the read/write workload described for Figure 7. We use an average network path unavailability of 0.1% with Pareto distributed repair time R(t) = $1 - 15t^{-0.8}$ [10].

In Figure 11-a we measure the best order error that can be maintained for a given bandwidth budget. Order error constrains the number of outstanding uncommitted writes [47]. We compare the *TACT Aggressive* policy [46] to a *PRACTI Prefetch* policy that aggressively distributes invalidations as in TACT's policy but that distributes bodies using the self-tuning approach. This technique reduces the bandwidth needed to maintain reasonable consistency by a factor of 3 compared to *TACT Aggressive* and improves the consistency bounds attainable for some bandwidth budgets by orders of magnitude.

Figure 11-b plots system unavailability for an order error of 100 as bandwidth varies. Following Yu and Vahdat's methodology [46], we say the system is *available* to a read or write request if the request can issue without blocking and the system is *unavailable* if the request must block to meet the consistency target. When bandwidth is limited, PRACTI dramatically improves system availability under consistency constraints compared to full replication.

Consistency overheads. As Section 3.4 describes, PRACTI ensures that individual requests pay only the latency and availability costs of consistency that they require. But, distributing sufficient bookkeeping information to support a wide range of per-request semantics does impose a modest bandwidth cost. In particular, object replication systems [16, 34] do not provide cross-object consistency guarantees. In the context of our system, if all applications in a system only care about coherence guarantees, the system could completely omit imprecise invalidations.

Figure 11-c quantifies the cost to distribute both precise and imprecise invalidations (in order to support consistency) versus the cost to distribute only precise invalidations for the subset of data of interest and omitting the imprecise invalidations (and thus only supporting coherence.) Note that the cost of imprecise invalidations depends on the workload: if there is no locality and writers tend to quickly alternate between writing objects of interest and objects not of interest, then the imprecise invalidations "between" the precise invalidations will cover relatively few updates and save relatively little overhead, but if writes to different interest sets arrive in bursts, then the system will generally be able to accumulate large numbers of updates into imprecise invalidations. We vary the fraction of data "of interest" to a node on the x axis and show the invalidation bytes received per write on the y axis. All objects are equally likely to be written by a set of remote nodes, but the locality of writes varies: the "No Locality" line shows the worst case scenario, with no locality across writes, and the "burst=10" line shows



(a) Best consistency (order error) achievable for a given bandwidth cost.

(b) Best unavailability achievable while meeting a required order error of 100.

(c) Bandwidth cost of distributing consistency information.

Fig. 11: Consistency trade-offs (a-b) and costs (c).

the case when a write is ten times more likely to hit the same interest set as the previous write than to hit a new interest set.

When there is significant locality for writes, the cost of distributing imprecise invalidations is small: imprecise invalidations to support consistency never add more than 20% to the bandwidth cost of supporting only coherence. When there is no locality, the cost is higher, but in the worst case imprecise invalidations add under 50 bytes per precise invalidation received. Overall, the difference in invalidation cost is likely to be small relative to the total bandwidth consumed by the system to distribute bodies.

5 Related work

Replication is fundamentally difficult. As noted in Section 3.4, the CAP dilemma [7, 36] and performance/-consistency dilemma [26] describe fundamental availability/performance/consistency trade-offs. 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 designer to retain full flexibility along at most two of the three dimensions of replication, consistency, or topology policy. Section 2 compares PRACTI with existing PR-AC [1, 6, 12, 18, 21, 29], AC-TI [14, 20, 23, 30, 47], and PR-TI [16, 34] approaches. These systems can be seen as special case "projections" of the more general PRACTI mechanisms [11].

Our PLFS prototype uses a DHT-based system [43] in its control plane to track where objects are stored. Other DHT-based systems store the data, itself, in the DHT [9, 31, 32, 33]. These systems implement a specific—if sophisticated—topology and replication policy: they can be viewed as generalizations of client-server systems where the server is split across a large number of nodes on a per-object or per-block basis for scalability and replicated to multiple nodes for availability and reliability. We believe these policies could be implemented over PRACTI mechanisms, but doing so is future work.

Like PRACTI, the Deceit file system [36] provides a flexible substrate that subsumes a range of replication systems. Deceit, however, focuses on replication across a handful of well-connected servers, and it therefore makes very different design decisions than PRACTI. For example, each Deceit server maintains a list of all files and of all nodes replicating each file, communication among servers is via an Isis [5] group for each distinct subset of servers, and all nodes replicating a file receive all bodies for all writes to the file.

Microsoft has announced that a new replication system, WinFS, will appear at some future date [41]. It will reportedly support synchronization across multiple nodes, however no detailed technical description of the protocol has been published. One report [42] suggests that the system transfers sets of updated items "rather than maintaining and synchronizing a log of each individual action," which may indicate that WinFS takes a PR-TI approach.

The web edge server system described in Section 4.1 is based on the TRIP system [28]. TRIP has been extensively evaluated via simulation [28], but ours is the first implementation of the approach.

6 Conclusion

In this paper, we introduce the PRACTI paradigm for replication in large scale systems and we describe the first system to simultaneously provide all three PRACTI properties. Evaluation of our prototype suggests that *by disentangling mechanism from policy, PRACTI replication enables significantly better trade-offs for system designers than possible with existing mechanisms.* By subsuming existing approaches and enabling new ones, 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] 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, Feb. 1996.

[2] S. Annapureddy, M. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *Proc NSDI*, May 2005.

[3] M. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, 1994.

[4] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proc. ASPLOS*, pages 10–22, Sept. 1992.

[5] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. SOSP*, Nov. 1987.
[6] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *ICDCS*, June 1992.
[7] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July/August 2001.
[8] S. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a Language for Writing Coherence Protocols. In *USENIX Conf. on Domain-Specific Lang.*, Oct. 1997.
[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct. 2001.
[10] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN service availability. *ACM/IEEE Transactions on Networking*, 11(2), Apr. 2003.
[11] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. "PRACTI replication for large-scale systems (extended technical report)". http://www.cs.utexas.edu/users/dahlin/papers/PRACTI-2005-10-extended.pdf, Oct. 2005.
[12] 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, Nov. 1994.
[13] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Wkshp. on Internet Svr. Perf.*, June 1998.
[14] R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, 1992.
[15] J. Gray, P.Helland, P. E. O'Neil, and D. Shasha. Dangers of replication and a solution. In *Proc. SIGMOD*, pages 173–182, 1996.
[16] R. Guy, J. Heidemann, W. Mak, T. Page, G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer Conf.*, pages 63–71, June 1990.
[17] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.
[18] 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, Feb. 1988.
[19] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *ICDCS*, pages 302–311, 1990.
[20] P. Keleher. Decentralized replicated-object protocols. In *PODC*, pages 143–151, 1999.
[21] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, Feb. 1992.
[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, Nov. 2000.
[23] 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.
[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.
[25] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.
[26] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton, 1988.
[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, Jan. 1992.
[28] A. Nayate, M. Dahlin, and A. Iyengar. Transparent information dissemination. In *Proc. Middleware*, Oct. 2004.
[29] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1), Feb. 1988.
[30] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. SOSP*, Oct. 1997.
[31] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. USENIX FAST*, Mar. 2003.
[32] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc SIGCOMM*, 2005.
[33] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, 2001.
[34] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. OSDI*, Dec. 2002.
[35] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proc. OSDI*, pages 35–46, Oct. 1996.
[36] A. Siegel. *Performance in Flexible Distributed File Systems*. PhD thesis, Cornell, 1992.
[37] Sleepycat Software. *Getting Started with BerkeleyDB for Java*, Sept. 2004.
[38] 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*, Dec. 1995.
[39] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *ICDCS*, May 1999.
[40] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In *Proc. OSDI*, Dec. 2002.
[41] http://msdn.microsoft.com/data/winfs/, Mar. 2005.
[42] http://longhorn.msdn.microsoft.com/lhsdk/winfs/consynchronizationoverview.aspx, Mar. 2005.
[43] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc SIGCOMM*, Aug. 2004.
[44] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering web cache consistency. *ACM Trans. on Internet Tech.*, 2(3), 2002.
[45] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proc USITS*, Oct. 1999.
[46] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proc. SOSP*, 2001.
[47] 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), Aug. 2002.

Acknowlegements

We thank Haifeng Yu, Emmett Witchell, Lily Qiu, and Jean-Phillip Martin for their helpful comments on earlier drafts of this paper.

A Algorithm details

The body of this paper describes the PRACTI protocol, but it omits some important low-level details. These details follow from the high-level PRACTI design provided in the paper, but our experience developing the PRACTI prototype was that getting these details precisely right was one of the most intellectually challenging parts of our effort.

Figure 12 details the incremental log exchange algorithm for processing a causally-consistent stream of incoming *generalized invalidations* gi_1, gi_2, ... starting from logical time $prevVV$. A generalized invalidation gi can be either a precise invalidation or an imprecise invalidation. $prevVV$ is the logical time of the causal stream that was current *just before* the next gi in the stream was applied.

There are three sets of state that are updated: (1) each incoming invalidation is inserted, sorted by logical time, into the local log (line 21), (2) precise invalidations update the local per-object state checkpoint (lines 29 to 32, and (3) imprecise invalidations update the interest set status (lines 22 to 24 and lines 37 to 40).

The implementations of the log update and the interest set status update are complicated by two factors. First, imprecise invalidations span an array of per-node logical start times to an array of per-node logical end times, and our system concurrently processes multiple incoming invalidation streams, yet we must apply invalidations to the local state in a causally-consistent order. Second, we wish to maximize the amount of information extracted from each invalidation so that the amount of IMPRECISE state is minimized. We have found that we must carefully define the precise low-level rules for updating interest set status and the log.

Interest set status. As Section 3.2 indicates, each node groups its objects into *interest sets* and applies imprecise invalidations to interest sets rather than individual objects to (a) improve performance and (b) ensure liveness. To accommodate different workloads across nodes, our prototype allows each node to independently group objects into interest sets and to dynamically split and join interest sets in response to workload changes. To ensure consistency, a node must mark an interest set *IMPRECISE* when a new imprecise invalidation intersects with it. To ensure liveness, when a node has later seen sufficient precise invalidations, it must mark interest set as PRECISE.

To explain how interest set status is tracked, we now detail a node's algorithm for processing an incoming stream of invalidations. As indicated in Figure 12 line 7, each incoming invalidation stream consists of a logical start time $startVV$ followed by a series of general invalidations gi_1, gi_2, ... such that any invalidation whose start time logically occurs after $startVV$ and on which gi_i causally depends appears before gi_i.

At the core of the algorithm is a simple idea: an interest set is *PRECISE* if it has missed no precise invalidations. Three variables are therefore central to processing an invalidation stream:

1. The **global** $currentVV$ version vector holds the highest logical time observed by the system across all invalidations processed from all streams.
2. The **per-interest-set** *last precise version vector* ($IS.lpVV$) indicates the highest logical time for which interest set IS is *PRECISE*. In particular, $IS.lpVV$ holds the highest logical time such that all objects in interest set IS reflect all writes up to $IS.lpVV$. An interest set IS is regarded as *PRECISE* if and only if $IS.lpVV = currentVV$. Otherwise, the interest set may have missed one or more precise invalidations, and we regard the interest set as *IMPRECISE*.
3. The **per-stream** $prevVV$ variable always holds the logical time just *before* the next invalidation in the stream is applied. Each invalidation gi is processed in the context of the logical time at which it was applied to determine if gi can advance $IS.lpVV$. $prevVV$ is initialized to the stream's $startVV$ and advanced to include $gi.end$ as each gi is processed.

The interest set status information is updated in four places as summarized in Figure 13. The first three updates occur when gi is first encountered in the stream, i.e., when it is known that there is no event that is causally after $prevVV$ and causally before gi. The fourth occurs at $gi.end$, i.e., when it is known that no remaining gi_i in the stream contains any event that causally occurs before $gi.end$.

When gi is first encountered in the stream, we always advance $currentVV$ to include the *end time* of gi because the system now reflects information in gi (update number 1 in the table, line **??** in the pseudo-code). Further, gi's presence in the causal invalidation stream means that any interest set that was *PRECISE* before gi is still *PRECISE* to $gi.start$. So, if interest set IS was *PRECISE* at time $stream.prevVV$ then we advance $IS.lpVV$. We advance $IS.lpVV$ differently depending on whether gi is a precise or imprecise invalidation. If gi is precise, then there have been no imprecise invalidations between $stream.prevVV$ and $gi.start$, and we advance $IS.lpVV$ to include $gi.end$ (note: $gi.start = gi.end$ if gi is precise.) That case is update number 2 in the table and line 26 in the pseudo-code. Conversely, if gi is imprecise, we can only advance $IS.lpVV$ to just before $gi.start$ (i.e., $\forall \alpha : IS.lpVV_\alpha = max(IS.lpVV_\alpha, gi.start_\alpha - 1)$). That case is update number 3 in the table and line 28 in the pseudo-code.

Two points should be emphasized:

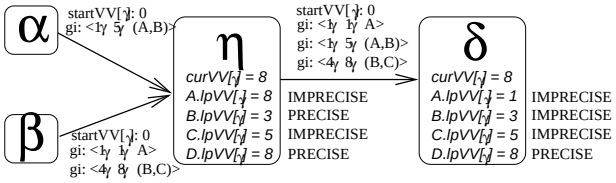
- Notice that when there is a gap in the logical time sequence for a given node, $gi.start$ may exceed

```
1: // Global state:
2: //   currentVV – node's current version vector
3: //   IS.lpVV – IS's last precise version vector
4: //   CP_obj – per-object checkpoint
5: //   log – replay log
6: // Per-stream state:
7: //   stream = startVV, gi_1, gi_2, ...
8: //   gi – next generalized invalidation to apply
9: //   prevVV – logical time before next gi applied
10: //   pending – set of gi's whose end time has not passed
11:
12: Procedure ProcessInvalStream(IS, stream)
13: prevVV = stream.readObj()
14: if ! includes(currentVV, prevVV)
15:     return; // Reject streams that do not preserve prefix property
16: pending = new Set()
17: gi = stream.readObj()
18: while (gi ≠ EOF) do
19:     nextStartVV =advanceToInclude(prevVV, gi.start)
20:     if !(∃bufferedInval ∈ pending |includes(nextStartVV,bufferedInval.end))
21:         log.insert(gi, prevVV)
22:         // Update interest set status
23:         currentVV =advanceToInclude(currentVV, gi.end) // update (1)—see text
24:         if includes(IS.lpVV, prevVV)   // If no gaps, update lpVV
25:             if gi.isPrecise()  // Advance to include precise inval
26:                 IS.lpVV =advanceToInclude(IS.lpVV, gi.start) // update (2)
27:             else // Advance to just before imprecise inval
28:                 IS.lpVV =advanceNoInclude(IS.lpVV, gi.start) // update (3)
29:         // Update per-object state
30:         if gi.isPrecise()
31:             CP_gi.objId.valid = INVALID
32:             CP_gi.objId.accept = gi.start
33:         pending.insert(gi) // Apply to non-overlapping later
34:         prevVV = nextStartVV // Update stream logical time
35:         gi = stream.readObj()
36:     else // Apply non-overlapping bufferedInval from pending at end time
37:         if !(bufferedInval.target intersects IS)
38:             if includes(lpVV, prevVV_α)
39:                 IS.lpVV =advanceToInclude(IS.lpVV,bufferedInval.endVV) // update (4)
40:         pending.remove(bufferedInval)
41:
42: Procedure advanceToInclude(VV1, VV2)
43: for all nodeId do
44:     retVV_nodeId = max(VV1_nodeId, VV2_nodeId)
45: return retVV
46:
47: Procedure advanceNoInclude(VV1, VV2)
48: for all nodeId do
49:     retVV_nodeId = max(VV1_nodeId, VV2_nodeId − 1)
50: return retVV
51:
52: Procedure includes(VV1, VV2)  // Does VV1 include VV2?
53: for all nodeId do
54:     if VV2_nodeId > VV1_nodeId
55:         return false
56: return true
```

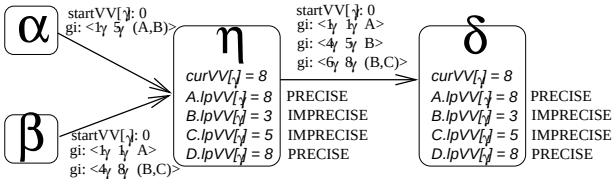
Fig. 12: **ProcessInvalStream** for interest set $IS\ stream$ = $\{prevVV, gi_1, gi_2, \dots\}$

Update Number	Code Line	When	IS state PRECISE/IMPRECISE	gi PRECISE/IMPRECISE	gi intersects IS	Action
(1)	23	$gi.start$	ANY	ANY	ANY	Advance cVV to include $gi.end$
(2)	26	$gi.start$ (= $gi.end$)	PRECISE	PRECISE	ANY	Advance $IS.lpVV$ to include $gi.end$ (= $gi.start$)
(3)	28	$gi.start$	PRECISE	IMPRECISE	ANY	Advance $IS.lpVV$ to just before $gi.start$
(4)	39	$gi.end$	PRECISE	IMPRECISE	NO	Advance $IS.lpVV$ to include $gi.end$

Fig. 13: Summary of cases for updating interest set PRECISE/IMPRECISE status.



(a) Naive log exchange.



(b) Log exchange with gap-filling and intersection.

Fig. 14: Example log exchange when node η first receives a log from α, then receives a log from β, and then sends the combined log to δ. Generalized invalidations have three fields: $< start, end, target >$. Note that all writes were issued by node γ and, for clarity, we show only γ's component for all version vectors.

$IS.lpVV$ even though no invalidations were skipped. This is why we maintain $prevVV$ for each stream and why line 24 compares $IS.lpVV$ against $prevVV$ rather than against $gi.start$ when deciding whether it is safe to advance $IS.lpVV$.

- Notice that an imprecise invalidation gi will always advance $currentVV$ to include gi's *end* time but can at most advance $IS.startVV$ to just before gi's *start* time. It is this difference that causes imprecise invalidations to make interest sets *IMPRECISE*.

If we stopped here, an imprecise invalidation would make both interest sets it overlaps and interest sets it does not overlap *IMPRECISE*. The algorithm addresses this issue by buffering each imprecise invalidation after it is first applied at its start time and applying a buffered invalidation *bufferedInval* again once $stream.prevVV$ includes bi's end time (i.e., once all gis whose start times preceed *bufferedInval*'s end time have been processed.) Buffered invalidation *bufferedInval* advances $IS.lpVV$ to include *bufferedInval*$.end$ for any interest set IS that (a) *bufferedInval*$.target$ does *not* intersect and that (b) is *PRECISE* as of logical time $stream.prevVV$. This case is update number 4 in the table and line 39 in the code. Notice that by waiting until bi's end time before advancing "nonoverlapping" invalidations to the end time, we avoid erroneously advancing $IS.lpVV$ for an interest set that becomes *IMPRECISE* between *bufferedInval*$.start$ and *bufferedInval*$.end$.

Finally notice that the algorithm above ensures that if an interest set IS becomes *IMPRECISE*, it can be made precise by receiving a stream that contains all precise invalidations that occurred between $IS.lpVV$ and $currentVV$ and that targets IS.

Log update. As indicated in line 21, a node stores each incoming invalidation in its log.

Imprecise invalidations complicate log updates. For example, a node η may receive different subsets of information from different peers α and β. η must ensure that imprecise invalidations received from α do not "mask" precise invalidations received from β and vice versa. Notice that the algorithm just described updates a node's local state by interpreting each gi relative to a per-stream $prevVV$, which allows the algorithm to infer that there are no missing invalidations between $prevVV$ and gi. But, if η were simply to store each gi in its log, some of this valuable "no missing invalidations" information that comes from gi's position in a stream could be lost. Then, as Figure 14-(a) illustrates, if η were to send its log to some other node δ, then even if δ receives the same gis as η, δ could end up *IMPRECISE* where η is *PRECISE* (e.g., for objects A).

In order to ensure that a node can transmit all information received including both the generalized invalidations and the information implicit in the incoming invalidation stream, we augment our logs in three ways.

First, each node maintains separate *per-writer logs*: when a node inserts gi into its log, it first decomposes gi into per-writer general invalidations and then inserts the per-writer pieces into separate logs. Decomposing gi into per-writer general invalidations gi_α is simple: for each server α in $gi.start$, generate gi_α with $start = gi.start_\alpha$, $end = gi.end_\alpha$, and $target = gi.target$.

Second, each per-writer log uses *gap filling* to explicitly encode the knowledge that each incoming stream is causally consistent and is therefore FIFO consistent for each writer. When a node inserts gi_α into its per-writer log for α, if gi_α is newer than the newest element in the log, it fills any gap between $gi_\alpha.start$ and the existing element by inserting a new gap-filling invalidation with a start stamp one larger than the highest existing end stamp, an end stamp one smaller than $gi_\alpha.start$, and an empty target.

Third, each per-writer log uses *intersection* to combine information received across multiple streams. In particular, 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 a and b 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 a and b's targets; and the third covers the time from the earlier end to the later end and covers the targets of the later end.

As Figure 14-(b) illustrates, when a node sends a stream of invalidations to another node, it discards gap-

filling invalidations and it combines per-writer invalidations into multi-writer invalidations.

Forming imprecise invalidations. When a controller asks node α to send a stream of invalidations from α's log to node β, the controller specifies two parameters that each filter the transmitted information: $startVV$ provides a filter on logical time, and $preciseFilter$ provides a filter on the ID space. α replies with a causally consistent stream of all invalidations it knows about that logically occurred after $startVV$. Invalidations whose target intersects $preciseFilter$ are sent as is (typically they are precise, but some may be imprecise), but α combines other invalidations into imprecise summaries as just described. PRACTI forms an imprecise invalidation I by combining generalized invalidations A and B. I has *start* and *end* arrays with entries for every node η in either A or B's *start*, and $I.start_\eta = min(A.start_\eta, B.start_\eta)$, and $I.end_\eta = max(A.end_\eta, B.end_\eta)$. Finally, $I.target$ encompasses all objects encompassed by A and B's *targets*. This process is incremental and continuous—as new invalidations arrive at α, α sends them on to β once all causally prior invalidations have been sent.

Split-join example. The following example is a bit involved, but we have found that working through it step by step sheds considerable light on the purpose of the rules for updating the interest set status and log *gap filling* and *intersection* just described.

Figure 15 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 invalidations (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.[4] 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}.

B General framework

PRACTI mechanisms represent a general framework for implementing a broad range of replication systems that specify their own policies for distributing bodies, handling read misses, sending invalidations, and enforcing consistency. For example, existing 2-of-3 protocols (AC-TI, PR-AC, and PR-TI) can be viewed as special cases or projections of the PRACTI protocol with certain features "optimized out" of the mechanisms by embedding restrictive policy assumptions. At the same time, the more general PRACTI mechanisms allow new trade-offs that existing protocols can not accommodate.

AC-TI. Server-replication systems such as Bayou [30], TACT [47], and lazy replication [23] allow arbitrary communication between nodes and can provide flexible consistency, but they fully replicate all objects in a volume and send all updates to all nodes that serve the volume. In the PRACTI framework, these AC-TI protocols can be viewed as using a replicate-all strategy for both precise invalidations and bodies, never sending or receiving imprecise invalidations, and not implementing any mechanism to handle read misses because objects are always *PRECISE* and *VALID*.

PR-AC. Client-server and hierarchical systems such as AFS [18], Sprite [29], and Coda [21] allow nodes to cache or prefetch arbitrary subsets of data and in principle could support a range of consistency policies [44] (though, in practice, such systems typically implement

[4] And, in this case, b. Our current log maintenence 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.

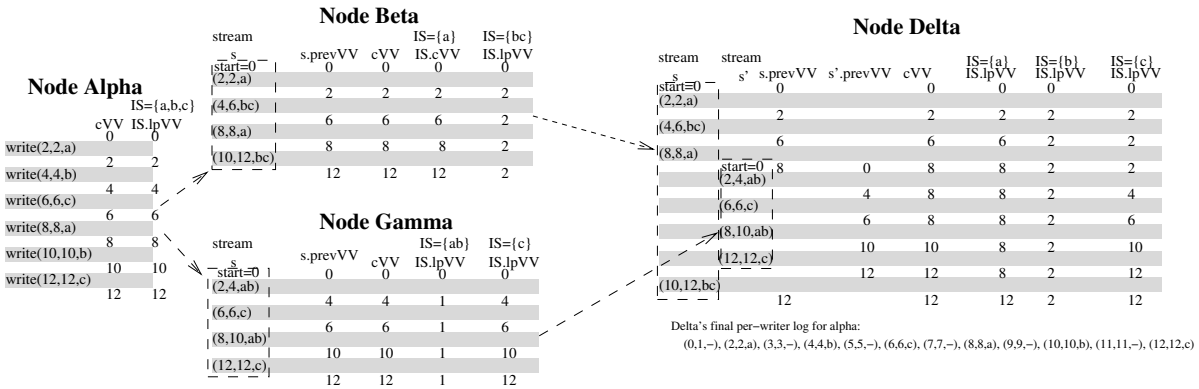


Fig. 15: Illustration of imprecise invalidation mechanisms in *split-join* scenario. Nodes α, β, γ, and δ share objects a, b, and c. At each node, we show the per-interest-set information (last precise version vector $lpVV$ and current version vector cVV), the per-invalidation-stream information ($startVV$ 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.

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 [12] serialization of control messages at the server is vital for reasoning about consistency [8]. In the PRACTI framework, these PR-AC protocols can be viewed as using separate invalidation and body messages, with invalidations sent by parents to children and bodies fetched by children from parents. Their callback protocols can be viewed as specialized instances of PRACTI's sendInval module that actively track which objects a child caches and that send precise invalidations only for those objects. Note that in PRACTI, the module would also send imprecise invalidations covering any omitted precise invalidations, but the hierarchical topology allows PR-AC protocols to omit these implicit imprecise invalidations. Interestingly, recovery when a server loses callback state [3] or when a topology changes [45] falls back on what are essentially explicit imprecise invalidations: the client receives a message (i.e., an imprecise invalidation covering all objects) indicating that it should treat all of its consistency state as suspect (i.e., *IMPRECISE*) and the client then revalidates all objects with its server (i.e., make the interest set precise).

PR-TI. Object replication systems such as Ficus [16] and Pangaea [34] maintain synchronization information separately for each object and support arbitrary topology policies and arbitrary placement of objects on nodes. 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. Furthermore, these systems cleanly separate invalidations and body messages: for any given object o and node η they either propagate o's update bodies to η or propagate no information at all about o's updates to η. In the PRACTI framework, these PR-TI protocols can be viewed as using a replication policy that sends invalidations and bodies for a given object to the same policy-specified subset of nodes and also as omitting all imprecise invalidations and thereby giving up the ability to consistently order writes across different objects.

PRACTI. In comparing PRACTI to these protocols, a key distinction is how consistent ordering of writes is achieved. Server-replication (AC-TI) and client-server (PR-AC) systems order invalidations across objects by enforcing an *inclusion property*—any node that receives and then transmits updates must see all updates for all objects about which it may speak. Server-replication mechanisms enforce this property by replicating all updates to all nodes, and client-server systems meet this obligation by assuming hierarchical inclusion. Because these policy assumptions are deeply embedded in these mechanisms it is difficult to, for example, "tweak" Bayou to support partial replication or to "tweak" Coda to support arbitrary topologies. Conversely, PRACTI introduces explicit imprecise invalidations to allow ordering of all updates without assuming full replication or hierarchical communication. Alternatively, object replication systems (PR-TI) dispense with this requirement by not providing cross-object ordering guarantees.

In addition to subsuming existing mechanisms, PRACTI exposes new regions of the design space and potentially offers better trade-offs than existing protocol families. For example, a designer who wants consistency is no longer forced to choose between using a desired topology but with full replication on one hand versus using a desired replication strategy but with restricted topology on the other. Section 4 examines several ex-

amples in detail and demonstrate that PRACTI can gain significant advantages compared to the alternatives.

As one example, consider a personal file system that distributes a user's data across a dozen information devices (e.g., a desktop machine, laptop machine, palmtop machine, home machine, mobile phone, media player, etc.). In such an environment, partial replication seems essential to cope with the dramatically varying capacities of different devices and to exploit locality of reference. For example, it seems undesirable to require a palmtop to store all the files available from a desktop. At the same time, node mobility makes it highly desirable for nodes to be able to optimize communications to changing network conditions. For example, if a user is in a hotel room or on a plane, she would like to be able to directly synchronize updates between her palmtop, phone, and laptop rather than use a hierarchical file system that requires her to send updates to and then retrieve updates from a server via a slow or expensive modem connection. Finally, causal and eventual consistency simplify reasoning about and resolving the inevitable inconsistencies introduced by disconnected operation. For example, causal consistency and eventual consistency are essential for ensuring that Bayou and TACT'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 such 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 states of different nodes arbitrarily diverge.

As a second example, a global file system for Grid or PlanetLab researchers might also benefit from the PRACTI properties. Requiring full replication appears untenable: partial replication allows data to be sent only where it is needed to, for example, send different subsets of input to different processing nodes. And topology independence also seems valuable to send datasets by the best available paths; for example, one could send a large data set from a repository to a distant cluster by sending the data once across the WAN to one node in the cluster and then flooding the data to other nodes in the cluster using the cluster's fast LAN links. Finally, flexible consistency guarantees allow the replication system to meet the application's consistency requirements without paying for stronger guarantees than required.

Finally, consider an enterprise file system spanning multiple departments in a university and supporting disconnected operation by portable devices. Partial replication avoids the need for computer science department servers to see all updates by faculty in the biology department except for a few subdirectories that contain joint projects. Similarly, a user's laptop does not need to see updates by all other users; instead it just sees updates relating to projects the user is working on. Topology independence allows peer-to-peer synchronization of mobile devices when a group on a retreat or at a conference hotel is collaborating on a document. And arbitrary consistency could provide strong guarantees for connected servers but flexible guarantees and reconciliation for mobile devices.

C Additional features

Due to space constraints, the main body of Section 3.5 omits discussion of several additional features of our implementation.

Write commitment. As in Bayou [30], PRACTI provides eventual consistency: for any write w, eventually all nodes will agree on a total order of all writes preceding w. A node considers a write w *committed* when the node knows w's final position in the global total order. For simplicity, we use Golding's algorithm [14]: each node η maintains a $currentVV$ version vector, and each entry $currentVV_\alpha$ stores the highest accept stamp of any invalidation by α that η has processed. Then, any write whose accept stamp is less than the lowest entry in $currentVV$ is *committed*. Supporting other write commitment protocols such as primary commit [30] or voting [20] would be straightforward, but we have not implemented these variations yet.

Bound writes. Separating invalidations from updates enables partial replication but also raises the issue of reliability: in Bayou, for example, all nodes have copies of all data, but a PRACTI system must 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 [34] to propagating all data to a well-provisioned central server [18] or replicated server "core" [21, 22] to replicating everything to everyone [30]. When an application issues a bound write, it creates a *bound invalidation* that includes the body of the write. Bound invalidations propagate through the system using log exchange and controllers manage this propagation to meet replication requirements. A controller can later issue messages to unbind a write, after which the invalidation can propagate without the body.

Embargoed writes. We provide an *embargoed write* low-level interface over which we provide a 2-phase write interface that ensures that once a write returns, no subsequent read can return the old value of the data. In particular, an embargoed write attaches an *EMBARGOED* flag to its invalidation record, and all reads of

EMBARGOED objects block. A controller can later insert a *RELEASE* record into the log to end the embargo of the record. Of course, the strong consistency of 2-phase commit comes at a price to availability [7], but our implementation provides this option so that applications can whatever point in the range of "arbitrary consistency" they require.

Crash recovery. The checkpoint stores per-object state and per-interest set state. The log acts as a replay log to recover events not yet reflected in the checkpoint.

Conflict detection and resolution. The protocol described in Section 3 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 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

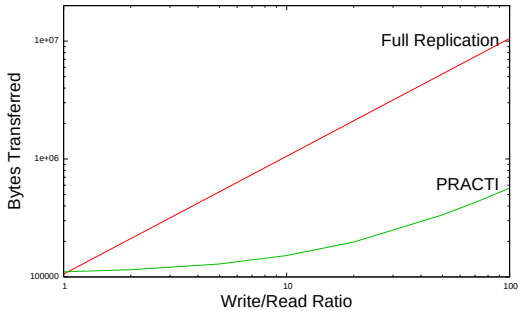


Fig. 16: Bandwidth cost of distributing updates for full replication and for partial replication as the number of successive writes to a file between reads varies.

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.

D Omitted graphs

The following graph was omitted from the main body of Section 4.1 due to space constraints.

Bytes die young. Bytes are often overwritten or deleted soon after their creation. For example, Baker et al. observed in a academic environment that between 50% and 70% of written data survive for more than 1 minute, and between 10% and 60% survive for more than 10 minutes [4]. Full replication systems send all writes to all servers, even if some of the writes are quickly made obsolete [30, 47]. In contrast, PRACTI replication can send invalidations separately from bodies, and overwritten bodies need never be sent.

Figure 16 illustrates this effect. For this experiment, we use a synthetic workload that randomly writes a set of files on one node and randomly reads the same set of files on another node on our prototype. On the x axis, we vary the ratio of writes to reads. We plot the bandwidth consumed on the y axis. PRACTI's gains are significant when bytes die young. For example, when the write to read ratio is 2, PRACTI uses 55% of the bandwidth of full replication, and when the ratio is 5, PRACTI uses 24%. At ratios exceeding 20, PRACTI's gains exceed an order of magnitude.

Mobile storage office topology Figure 17 was omitted from Section 4.2. It shows synchronization time among different devices when the palmtop and laptop are co-located with the Office Server.

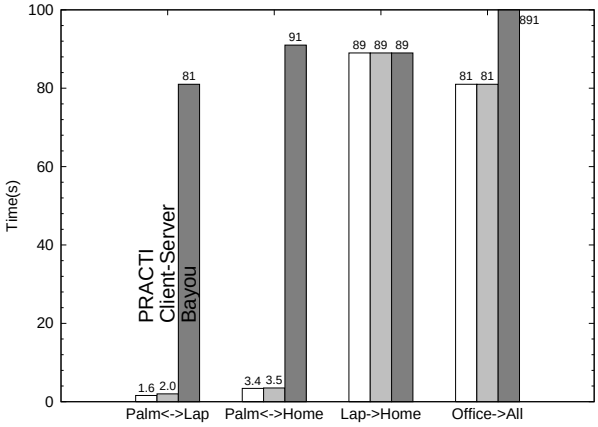


Fig. 17: Synchronization time among devices for "office" network topology and various protocols (see Section 4.2.