PRACTI Replication

Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate,
Arun Venkataramani, Praveen Yalagandula, Jiandan Zheng
University of Texas at Austin[*]

Abstract

We present PRACTI, a new approach for large-scale replication. PRACTI systems can replicate or cache any subset of data on any node (Partial Replication), provide a broad range of consistency guarantees (Arbitrary Consistency), and permit any node to send information to any other node (Topology Independence). A PRACTI architecture yields two significant advantages. First, by providing all three PRACTI properties, it enables *better trade-offs* than existing mechanisms that support at most two of the three desirable properties. The PRACTI approach thus exposes new points in the design space for replication systems. Second, the *flexibility* of PRACTI protocols simplifies the design of replication systems by allowing a single architecture to subsume a broad range of existing systems and to reduce development costs for new ones. To illustrate both advantages, we use our PRACTI prototype 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 by as much as an order of magnitude.

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 [22] as well as between availability and consistency [9, 31], systems make different compromises among these factors by implementing different placement policies, consistency policies, and topology policies for different environments. Informally, *placement policies* such as demand-caching, prefetching, or replicate-all define which nodes store local copies of which data, *consistency policies* such as sequential [21] or causal [16] 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.

This paper argues that an ideal replication framework should provide all three PRACTI properties:

- *Partial Replication* (PR) means that a system can place any subset of data and metadata on any node. In contrast, some systems require a node to maintain copies of all objects in all volumes they export [26, 37, 39].
- *Arbitrary Consistency* (AC) means that a system can provide both strong and weak consistency guarantees and that only applications that require strong guarantees pay for them. In contrast, some systems can only enforce relatively weak coherence guarantees and can make no guarantees about stronger consistency properties [11, 29].
- *Topology Independence* (TI) means that any node can exchange updates with any other node. In contrast, many systems restrict communication to client-server [15, 18, 25] or hierarchical [4] patterns.

Although many existing systems can each provide two of these 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.

This paper presents the first replication architecture to provide all three PRACTI properties. The protocol draws on key ideas of existing protocols but recasts them to remove the deeply-embedded assumptions that prevent one or more of the 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 [26, 37, 39]. To support partial replication, we extend the mechanisms in two simple but fundamental ways.

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 despite multiple streams of information, and we introduce general

[*]This work was supported in part by the National Science Foundation (CNS-0411026, SCI-0438314), the Texas Advanced Technology Program, and an IBM University Partnership Award. Gao's present affiliation: Oracle, Inc. Venkataramani's present affiliation: University of Massachusettes, Amherst. Yalagandula's present affiliation: Hewlett Packard Labs.

mechanisms for handling demand read misses.

2. In order to allow partial replication of update metadata, we introduce *imprecise invalidations*, which allow a single invalidation to summarize a set of invalidations. Imprecise invalidations provide cross-object consistency in a scalable manner: each node incurs storage and bandwidth costs proportional to the size of the data sets in which it is interested. For example, a node that is interested in one set of objects A but not another set B, can receive precise invalidations for objects in A along with an imprecise invalidation that summarizes omitted invalidations to objects in B. The imprecise invalidation then serves as a placeholder for the omitted updates both in the node's local storage and in the logs of updates the node propagates to other nodes.

We construct and evaluate a prototype using a range of policies and workloads. Our primary conclusion is that by 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 full replication AC-TI replicated server systems, by providing more than an order of magnitude better synchronization delay compared to topology constrained PR-AC hierarchical systems, and by providing consistency guarantees not achievable by limited consistency PR-TI object replication systems.

More broadly, we argue that PRACTI protocols 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 computing.

In summary, this paper makes four contributions. First, it defines the PRACTI paradigm and provides a taxonomy for replication systems that explains why existing replication architectures fall short of ideal. Second, it describes the first replication protocol 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 revisits the design of existing systems in light of the PRACTI taxonomy. Section 3 describes our protocol for providing PRACTI replication, and Section 4 experimentally evaluates the prototype. Finally, Section 5 surveys related work, and Section 6 highlights our conclusions.

2 Taxonomy and challenges

In order to put the PRACTI approach in perspective, this section examines existing replication architectures and considers why years of research exploring many different replication protocols have failed to realize the goal of PRACTI replication.

Note that the requirements for supporting flexible consistency guarantees are subtle, and Section 3.3 discusses the full range of flexibility our protocol provides. PRACTI replication should support both the weak coherence-only guarantees acceptable to some applications and the stronger consistency guarantees required by others. Note that *consistency* semantics constrain the order that updates across *multiple objects* become observable to nodes in the system while *coherence* semantics are less restrictive in that they only constrain the order that updates to a *single object* become observable but do not additionally constrain the ordering of updates across multiple locations. (Hennessy and Patterson discusses the distinction between consistency and coherence in more detail [12].) For example, if a node $n1$ updates object A and then object B and another node $n2$ reads the new version of B, most consistency semantics would ensure that any subsequent reads by $n2$ see the new version of A, while most coherence semantics would permit a read of A to return either the new or old version.

PRACTI Taxonomy. The PRACTI paradigm defines a taxonomy for understanding the design space for replication systems as illustrated in Figure 1. As the figure indicates, many existing replication systems can be viewed as belonging to one of four protocol families, each of which provides at most two of the PRACTI properties.

Server replication systems like Replicated Dictionary [37] and Bayou [26] 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 across all objects. Lazy Replication [19] and TACT [39] use 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

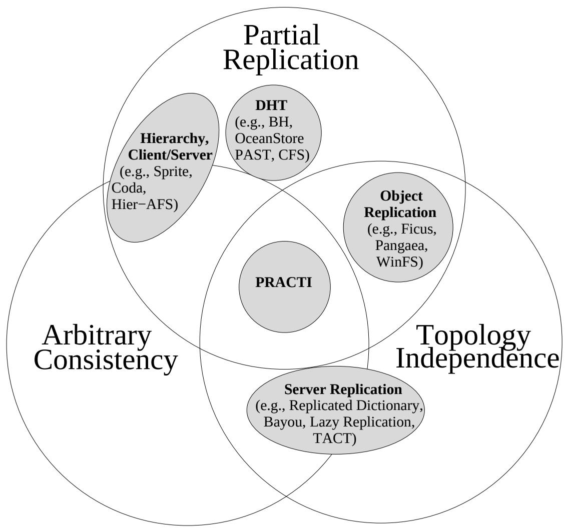

Partial Replication
DHT (e.g., BH, OceanStore PAST, CFS)
Hierarchy, Client/Server (e.g., Sprite, Coda, Hier−AFS)
Object Replication (e.g., Ficus, Pangaea, WinFS)
PRACTI
Arbitrary Consistency
Topology Independence
Server Replication (e.g., Replicated Dictionary, Bayou, Lazy Replication, TACT)

Fig. 1: The PRACTI taxonomy defines a design space for classifying families of replication systems.

and storage, and they may be unsuitable for devices with limited resources.

Client-server systems like Sprite [25] and Coda [18] and *hierarchical* caching systems like hierarchical AFS [24] permit nodes to cache arbitrary subsets of data (PR). Although specific systems generally enforce a set consistency policy, a broad range of consistency guarantees are provided by variations of the basic architecture (AC). However, these protocols fundamentally require communication to flow between a child and its parent. Even when systems permit limited client-client communication for cooperative caching, they must still serialize control messages at a central server for consistency [5]. These restricted communication patterns (1) hurt performance when network topologies do not match the fixed communication topology 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 sharing during disconnected operation when a set of nodes can communicate with one another but not with the rest of the system.

DHT-based storage systems such as BH [35], PAST [28], and CFS [6] 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. This division and replication, however, introduce new challenges for providing consistency. For example, the Pond OceanStore prototype assigns each object to a set of primary replicas that receive all updates for the object, uses an agreement protocol to coordinate these servers for per-object coherence, and does not attempt to provide cross-object consistency guarantees [27].

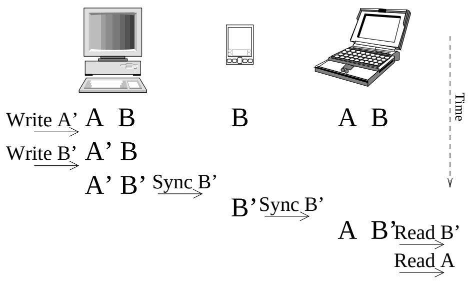


Fig. 2: Naive addition of PR to an AC-TI log exchange protocol fails to provide consistency.

Object replication systems such as Ficus [11], Pangaea [29], and WinFS [23] 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 but not consistency guarantees. Unfortunately, reasoning about the corner cases of consistency protocols is complex, so systems that provide only weak consistency or coherence guarantees can complicate constructing, debugging, and using the applications built over them. Furthermore, support for only weak consistency may prevent deployment of applications with more stringent requirements.

Why is PRACTI hard? It is surprising that despite the disadvantages of omitting any of the PRACTI properties, no system provides all three. Our analysis suggests that these limitations are fundamental to these existing protocol families: the assumption of full replication is deeply embedded in the core of server replication protocols; the assumption of hierarchical communication is fundamental to client-server consistency protocols; careful assignment of key ranges to nodes is central to the properties of DHTs; and the lack of consistency is a key factor in the flexibility of object replication systems.

To understand why it is difficult for existing architectures to provide all three PRACTI properties, consider Figure 2's illustration of a naive attempt to add PR to a AC-TI server replication protocol like Bayou. 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

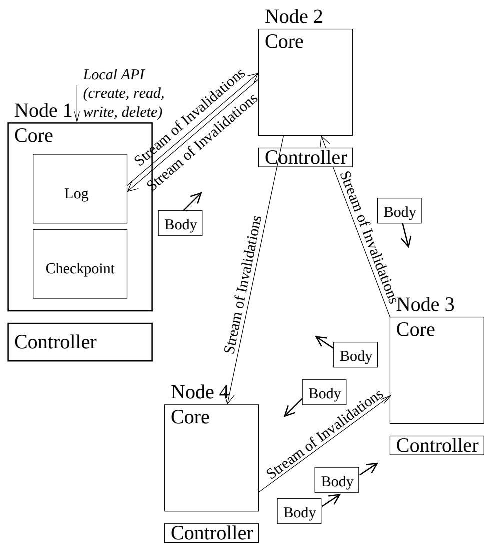


Fig. 3: High level PRACTI architecture.

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 [16] or even the weaker FIFO consistency [22].

This example illustrates the broader, fundamental challenge: supporting flexible consistency (AC) requires careful ordering of how updates propagate through the system, but consistent ordering becomes more difficult if nodes communicate in ad-hoc patterns (TI) or if some nodes know about updates to some objects but not other objects (PR).

Existing systems resolve this dilemma in one of three ways. The full replication of AC-TI replicated server systems ensures that all nodes have enough information to order all updates. Restricted communication in PR-AC client-server and hierarchical systems ensures that the root server of a subtree can track what information is cached by descendents; the server can then determine which invalidations it needs to propagate down and which it can safely omit. Finally, PR-TI object replication systems simply give up ability to consistently order writes to different objects and instead allow inconsistencies such as the one just illustrated.

3 PRACTI replication

Figure 3 shows the high-level architecture of our implementation of a PRACTI protocol.

Node 1 in the figure illustrates the main local data structures of each node. A node's *Core* embodies the protocol's mechanisms by maintaining a node's local state. Applications access data stored in the local core via the per-node *Local API* for creating, reading, writing, and deleting objects. These functions operate the local node's *Log* and *Checkpoint*: modifications are appended to the log and then update the checkpoint, and reads access the random-access checkpoint. To support partial replication policies, the mechanisms allow each node to select an arbitrary subset of the system's objects to store locally, and nodes are free to change this subset at any time (e.g., to implement caching, prefetching, hoarding, or replicate-all). This local state allows a node to satisfy requests to read valid locally-stored objects without needing to communicate with other nodes.

To handle read misses and to push updates between nodes, cores use two types of communication as illustrated in the figure—causally ordered *Streams of Invalidations* and unordered *Body* messages. The protocol for sending streams of invalidations is similar to Bayou's [26] log exchange protocol, and it ensures that each node's log and checkpoint always reflect a causally consistent view of the system's data. But it differs from existing log exchange protocols in two key ways:

1. *Separation of invalidations and bodies.* Invalidation streams notify a receiver that writes have occurred, but separate body messages contain the contents of the writes. A core coordinates these separate sources of information to maintain local consistency invariants. This separation supports partial replication of data—a node only needs to receive and store bodies of objects that interest it.

2. *Imprecise invalidations.* Although the invalidation streams each logically contain a causally consistent record of all writes known to the sender but not the receiver, nodes can omit sending groups of invalidations by instead sending *imprecise invalidations*. Whereas traditional *precise invalidations* describe the target and logical time of a single write, an imprecise invalidation can concisely summarize a set of writes over an interval of time across a set of target objects. Thus, a single imprecise invalidation can replace a large number of precise invalidations and thereby support partial replication of metadata—a node only needs to receive traditional precise invalidations and store per-object metadata for objects that interest it.

   Imprecise invalidations allow nodes to maintain consistency invariants despite partial replication of metadata and despite topology independence. In particular, they serve as placeholders in a receiver's log to ensure that there are no causal gaps in the log a node stores and transmits to other nodes. Similarly, just as a node tracks which objects are *INVALID* so it can block a read to an object that has been invalidated but for which the corresponding body message has not been received, a node tracks which sets of objects are *IMPRECISE* so it can block a read to an object that has been targeted by an imprecise invalidation and for which the node therefore may not know about the most recent write.

The mechanisms just outlined, embodied in a node's *Core*, allow a node to store data for any subsets of objects, to store per-object metadata for any subset of objects, to receive precise invalidations for any subset of objects from any node, and to receive body messages for any subset of objects from any node. Given these mechanisms, a node's *Controller* embodies a system's replication and topology policies by directing communication among nodes. A node's controller (1) selects which nodes should send it invalidations and, for each invalidation stream subscription, specifies subsets of objects for which invalidations should be precise, (2) selects which nodes to prefetch bodies from and which bodies to prefetch, and (3) selects which node should service each demand read miss.

These mechanisms also support flexible consistency via a variation of the TACT [39] interface, which allows individual read and write requests to specify the semantics they require. By using this interface, applications that require weak guarantees can minimize performance [22] and availability [9] overheads while applications that require strong guarantees can get them.

The rest of this section describes the design in more detail. It first explains how our system's log exchange protocol separates invalidation and body messages. It then describes how imprecise invalidations allow the log exchange protocol to partially replicate invalidations. Next, it discusses the crosscutting issue of how to provide flexible consistency. After that, it describes several novel features of our prototype that enable it to support the broadest range of policies.

3.1 Separation of invalidations and bodies

As just described, nodes maintain their local state by exchanging two types of updates: ordered streams of invalidations and unordered body messages. *Invalidations* are metadata that describe writes; each contains an object ID[1] 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 [20] 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 for node 1 in Figure 3, each node maintains a *log* of the invalidations it has received sorted by logical time. And, for random access, each node stores bodies in a *checkpoint* indexed by object ID.

[1]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.

- 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 [26].
- 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 logical time of the incoming invalidation.
- 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 [26]. 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 *i*th 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 a *causally consistent* [16] 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 local 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 [39].

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, hoarding, pre-positioning bodies according to a global placement policy, or push-all.

Design issues. The basic protocol adapts well-understood log exchange mechanisms [26, 37]. 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 [39] 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 the requested object's status is *VALID*. Of course, to ensure liveness, when an *INVALID* object is read, an implementation should arrange for someone to send the body. Therefore, 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 [8] or DHT-based [35] directory, using a hint-based search strategy, or relying on a push-all strategy [26, 37] (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 the system must not require all nodes to see all invalidations or to store metadata for each object. 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. 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. Similarly, 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.

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

As Figure 4 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

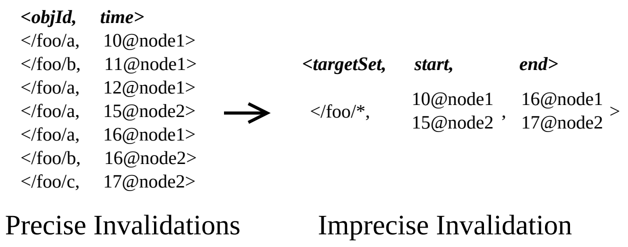


Fig. 4: Example imprecise invalidation.

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.

A node reduces its bandwidth requirements by subscribing to receive precise invalidations only for desired subsets of data and receiving imprecise invalidations for the rest. And a node saves storage by tracking per-object state only for desired subsets of data and tracking coarse-grained bookkeeping information for the rest.

Processing imprecise invalidations. When a node receives imprecise invalidation *I*, it inserts *I* into its log and updates its checkpoint. For the log, imprecise invalidations act as placeholders to ensure that the omitted precise invalidations do not introduce causal gaps in the log that a node stores locally or in the streams of invalidations that a node transmits to other nodes.

Tracking the effects of imprecise invalidations on a node's checkpoint must address four related problems:

1. For consistency, a node must *logically* mark all objects targeted by a new imprecise invalidation as *INVALID*. This action ensures that if a node tries to read data that may have been updated by an omitted write, the node can detect that information is missing and block the read until the missing information has been received.
2. For liveness, a node must be able to unblock reads for an object once the per-object state is brought up to date (e.g., when a node receives the precise invalidations that were summarized by an imprecise invalidation.)
3. For space efficiency, a node should not have to store per-object state for all objects. As the example at the start of this subsection illustrates, doing so would significantly restrict the range of replication policies, devices, and workloads that can be accommodated.
4. For processing efficiency, a node should not have to iterate across all objects encompassed by *targetSet* to apply an imprecise invalidation.

To meet these requirements, rather than track the effects of imprecise invalidations on individual objects, nodes keep bookkeeping information on groups of objects called *Interest Sets*. 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* (per-object state reflects all invalidations) or *IMPRECISE* (per-object state is not stored or may not reflect all precise invalidations) by maintaining two pieces of state.

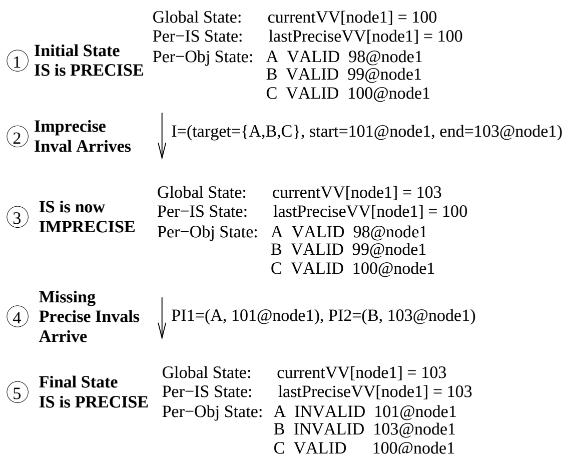
- Each node maintains a global variable *currentVV*, which is a version vector encompassing the highest timestamp of any invalidation (precise or imprecise) applied to any interest set.
- Each node maintains for each interest set *IS* the variable *IS.lastPreciseVV*, which is the latest version vector 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 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 set state meets the four requirements listed above.

1. By default, a read request blocks until the interest set in which the object lies is *PRECISE* and the object is *VALID*. This blocking ensures that reads only observe the checkpoint state they would have observed if all invalidations were precise and therefore allows nodes to enforce the same consistency guarantees as protocols without imprecise invalidations.
2. For liveness, the system must eventually unblock waiting reads. 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. When the missing invalidations arrive, they advance *IS.lastPreciseVV*. The algorithm for processing invalidations guarantees that any interest set *IS* can be made *PRECISE* by receiving a sequence *S* of invalidations from *IS.lastPreciseVV* to *currentVV* if *S* is causally sorted and includes all precise invalidations targeting *IS* in that interval.
3. Storage is limited: each node only needs to store per-object state for data currently of interest to that node. Thus, 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.
4. Imprecise invalidations are efficient to apply, requiring work that is proportional to the number of interest sets at the receiver rather than the number of summarized invalidations.


1 Initial State IS is PRECISE
Global State: currentVV[node1] = 100
Per−IS State: lastPreciseVV[node1] = 100
Per−Obj State: A VALID 98@node1
B VALID 99@node1
C VALID 100@node1
2 Imprecise Inval Arrives
I=(target={A,B,C}, start=101@node1, end=103@node1)
3 IS is now IMPRECISE
Global State: currentVV[node1] = 103
Per−IS State: lastPreciseVV[node1] = 100
Per−Obj State: A VALID 98@node1
B VALID 99@node1
C VALID 100@node1
4 Missing Precise Invals Arrive
PI1=(A, 101@node1), PI2=(B, 103@node1)
5 Final State IS is PRECISE
Global State: currentVV[node1] = 103
Per−IS State: lastPreciseVV[node1] = 103
Per−Obj State: A INVALID 101@node1
B INVALID 103@node1
C VALID 100@node1
Fig. 5: Example of maintaining interest set state. For clarity, we only show node1's elements of *currentVV* and *lastPreciseVV*.

Example. The example in Figure 5 illustrates the maintenance of interest set state. Initially, (1) interest set *IS* is *PRECISE* and objects *A*, *B*, and *C* are *VALID*. Then, (2) an imprecise invalidation *I* arrives. *I* (3) advances *currentVV* but not *IS.lastPreciseVV*, making *IS IMPRECISE*. But then (4) precise invalidations *PI1* and *PI2* arrive on a single invalidation channel from another node. (5) These advance *IS.lastPreciseVV*, and in the final state *IS* is *PRECISE*, *A* and *B* are *INVALID*, and *C* is *VALID*.

Notice that although the node never receives a precise invalidation with time *102@node1*, the fact that a single incoming stream contains invalidations with times *101@node1* and *103@node1* allows it to infer by the prefix property that no invalidation at time *102@node1* occurred, and therefore it is able to advance *IS.lastPreciseVV* to make *IS PRECISE*.

3.3 Consistency: Approach and costs

Enforcing cache consistency entails fundamental trade-offs. For example the CAP dilemma states that a replication system that provides sequential **C**onsistency cannot simultaneously provide 100% **A**vailability in an environment that can be **P**artitioned [9, 31]. Similarly, Lipton and Sandberg describe fundamental consistency v. performance trade-offs [22].

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.

Providing flexible guarantees. Discussing the semantic guarantees of large-scale replication systems requires careful distinctions along several dimensions. *Consistency* constrains the order that updates across multiple memory locations become observable to nodes in the system, while *coherence* constrains the order that updates to a single location become observable but does not additionally constrain the ordering of updates across multiple locations [12]. *Staleness* constrains the real-time delay from when a write completes until it becomes observable. Finally, *conflict resolution* [18, 34] provides ways to cope with cases where concurrent reads and writes at different nodes conflict.

Our protocol provides considerable flexibility along all four of these dimensions.

With respect to consistency and staleness, it provides a range of traditional guarantees such as the relatively weak constraints of causal consistency [16, 20] or delta coherence [32], to the stronger constraints of sequential consistency [21] or linearizability [13]. Further, it provides a continuous range of guarantees between causal consistency, sequential consistency, and linearizability by supporting TACT's order error for bounding inconsistency and temporal error for bounding staleness [39]. Because our design uses a variation of peer-to-peer log exchange [26, 37], adapting flexible consistency techniques from the literature is straightforward.

With respect to coherence, although our default read interface enforces causal consistency, the interface allows programs that do not demand cross-object consistency to issue *imprecise reads*. Imprecise reads may achieve higher availability and performance than precise reads because they can return without waiting for an interest set to become *PRECISE*. Imprecise reads thus observe causal coherence (causally coherent ordering of reads and writes for any individual item) rather than causal consistency (causally consistent ordering of reads and writes across all items.)

With respect to conflict resolution, our prototype provides an interface for detecting and resolving write-write conflicts according to application-specific semantics [18, 26]. In particular, nodes log conflicting concurrent writes that they detect in a way that guarantees that all nodes that are *PRECISE* for an interest set will eventually observe the same sequence of conflicting writes for that interest set. The nodes then provide an interface for programs or humans to read these conflicting writes and to issue new compensating transactions to resolve the conflicts.

Costs of consistency. PRACTI protocols should ensure that workloads only pay for the semantic guarantees they need. Our protocol does so 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 to specify its consistency and staleness 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, as in TACT [39], the performance/availability versus consistency dilemmas are resolved on a per-read, per-write basis.

Conversely, all invalidations that propagate through the system carry sufficient information that a later read can determine what missing updates must be fetched to ensure the consistency or staleness level the read demands. Therefore, the system may pay an extra cost: if a deployment never needs strong consistency, then our protocol may propagate some bookkeeping information that is never used. We believe this cost is acceptable for two reasons: (1) other features of the 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. Experiments in Section 4 quantify these bandwidth costs, and we argue that they are not significant.

3.4 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 briefly outline these aspects of the implementation and provide additional details in an extended technical report [3].

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

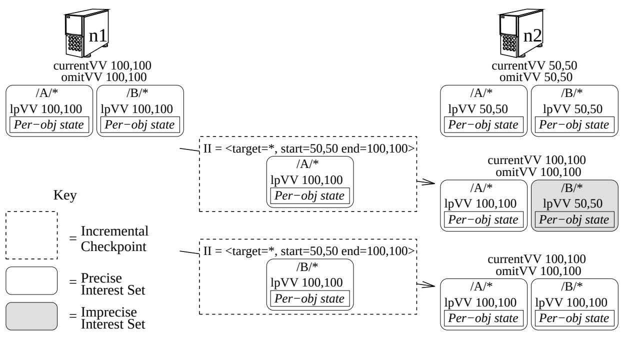


Fig. 6: Incremental checkpoints from *n1* to *n2*.

a log exchange beginning before *n1.omitVV*, then *n1* cannot send a stream of invalidations. Instead, *n1* must send a checkpoint of its per-object state.

In existing server replication protocols [26], in order to ensure consistency, such a checkpoint exchange must atomically update *n2*'s state for all objects in the system. Otherwise, the prefix property and causal consistency invariants could be violated. Traditional checkpoint exchanges, therefore, may block interactive requests while the checkpoint is atomically assembled at *n1* or applied at *n2*, and they may waste system resources if a checkpoint transfer is started but fails to complete.

Rather than transferring information about all objects, an incremental checkpoint updates an arbitrary interest set. As Figure 6 illustrates, an incremental checkpoint for interest set *IS* includes (1) an imprecise invalidation that covers all objects from the receiver's *currentVV* up to the sender's *currentVV*, (2) the logical time of the sender's per-object state for *IS* (*IS.lastPreciseVV*), and (3) per-object state: the logical timestamp for each object in *IS* whose timestamp exceeds the receiver's *IS.lastPreciseVV*. Thus, the receiver's state for *IS* is brought up to include the updates known to the sender, but other interest sets may become *IMPRECISE* to enforce consistency.

Overall, this approach makes checkpoint transfer a much smoother process than under existing protocols. As Figure 6 illustrates, 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.

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 [36]. 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 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.

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 [34]; this approach works in a batch system because rollback takes all of the system's state to a logical 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 [33] for per-node local storage. All features described in this paper are implemented including local create/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 use this prototype both (1) to evaluate the PRACTI architecture in several environments such as web service replication, data access for mobile users, and grid scientific computing and (2) to characterize PRACTI's properties across a range of key metrics.

Our experiments seek to answer two questions.

1. *Does a PRACTI architecture offer significant advantages over existing replication architectures?* We find that our system can dominate existing approaches by providing more than 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.

2. *What are the costs of PRACTI's generality?* Given that a flexible PRACTI protocol can subsume existing approaches, 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 can 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 of flexible consistency in 4.3.

4.1 Partial replication

When comparing to the full replication protocols from which our PRACTI system descends, we find that support for partial replication dramatically improves performance 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" [2].

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) our experiments provide evidence that despite 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.

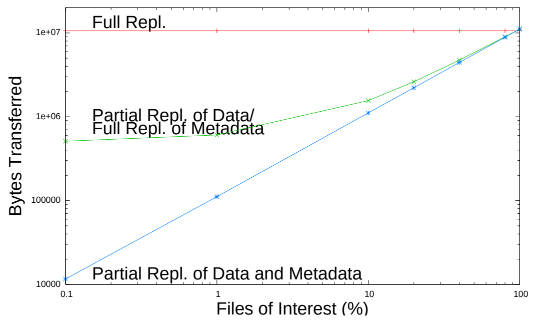


Fig. 7: Impact of locality on replication cost.

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 7 examines the impact of locality on replication cost for three systems implemented on our PRACTI core using different controllers: a full replication system similar to Bayou, a partial-body replication system that sends all precise invalidations to each node 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.

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. Using imprecise invalidations to provide true partial replication can gain over another order of magnitude as locality increases.

Note that Figure 7 shows bandwidth costs. Partial replication provides similar improvements for space requirements (graph omitted.)

Bytes die young. Bytes are often overwritten or deleted soon after creation [2]. 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

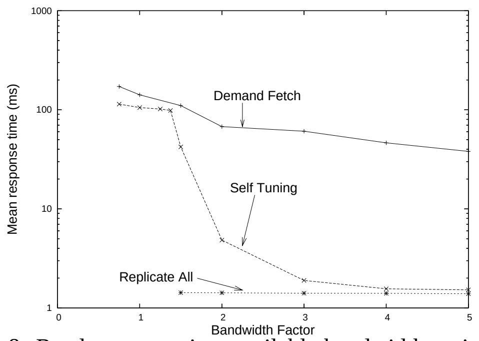


Fig. 8: Read response time available bandwidth varies for full replication, demand reads, and self-tuning replication.

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 files on another node. Due to space constraints, we defer the graph to the extended report [3]. 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 3.4. As a result, systems can replicate bodies aggressively when network capacity is plentiful and replicate less aggressively when network capacity is scarce.

Figure 8 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* we examine, 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* follows a standard client-server HTTP caching model by replicating precise invalidations to all nodes but sending 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 8 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 prefetching bandwidth corresponding to the current write rate even if other applications could make better use of the network.

	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. 9: Configuration for mobile storage experiments.

4.2 Topology independence

We examine topology independence by considering two environments: a mobile data access system 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:

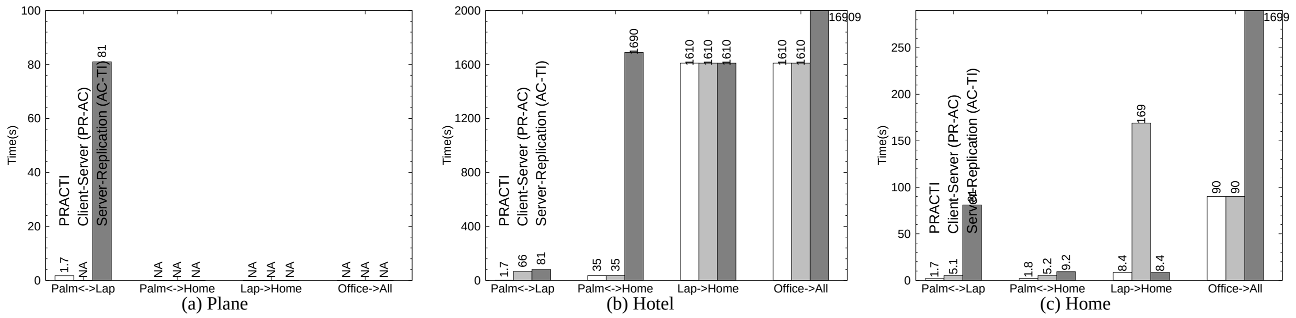
1. *Adapt to changing topologies*: a PRACTI system can make use of the best paths among nodes.
2. *Adapt to changing workloads*: a PRACTI system can optimize communication paths to, for example, use direct node-to-node transfers for some objects and distribution trees for others.

We primarily compare against standard restricted-topology client-server systems like Coda and IMAP. For completeness, our graphs also compare against topology-independent, full replication systems like Bayou.

Mobile storage. We first consider a mobile storage system that distributes data across palmtop, laptop, home desktop, and office server machines. We compare a PRACTI system to a client-server system that supports partial replication but that distributes updates via a central server and to a full-replication system that can distribute updates directly between any nodes but that requires full replication. All three systems are realized by implementing different controller policies.

As summarized in Figure 9 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 10 shows the time to synchronize dirty data among machines in three scenarios: (a) *Plane*: the user



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

Fig. 10: Synchronization time among devices for different network topologies and protocols.

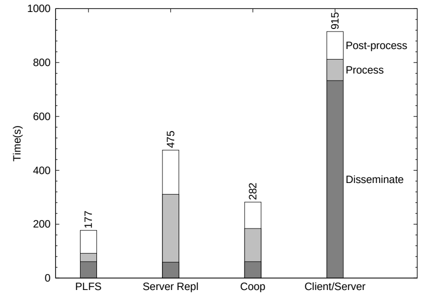


Fig. 11: Execution time for the WAN-Experiment benchmark on 50 distributed nodes with a remote server.

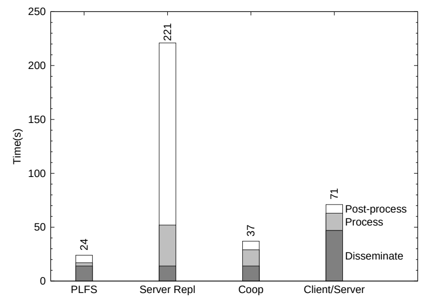


Fig. 12: Execution time for the WAN-Experiment benchmark on 50 cluster nodes plus a remote server.

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 9. For each location, we measure time for machines to exchange updates: (1) P↔L: the palmtop and laptop exchange updates, (2) P↔H: 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.

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 synchronization via the server.

WAN-FS for Researchers. Figures 11 and 12 evaluate 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 [38] 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 [1, 35].

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 [1], and server-replication [26]. All 4 systems are implemented via PRACTI using different controllers.

The figures show performance for an experiment running on 50 distributed nodes each with a 5.6Mb/s connection to the Internet (we emulate this case by throttling bandwidth) and 50 cluster nodes at the University of Texas with a switched 100Mb/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 trans-

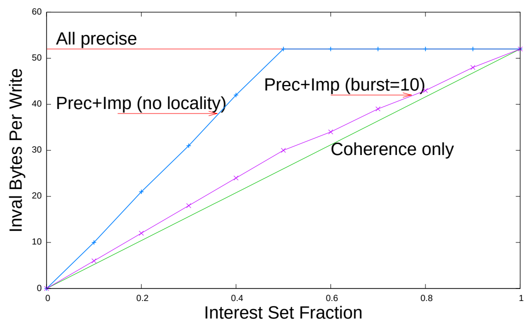


Fig. 13: Bandwidth cost of consistency information.

ferred directly between nodes.

4.3 Arbitrary consistency

This subsection examines the costs of PRACTI's generality. As Section 3.3 describes, our protocol ensures that requests pay only the latency and availability costs of the consistency they require. But, distributing sufficient bookkeeping information to support a wide range of per-request semantics does impose a bandwidth cost. If all applications in a system only care about coherence guarantees, a customized protocol for that system could omit imprecise invalidations and thereby reduce network overheads.

Three features of our protocol minimize this cost. First, transmitting invalidations separately from bodies allows nodes to maintain a consistent view of data without receiving all bodies. Second, transmitting imprecise invalidations in place of some precise invalidations allows nodes to maintain a consistent view of data without receiving all precise invalidations. Third, self-tuning prefetch of bodies allows a node to maximize the amount of local, valid data in a checkpoint for a given bandwidth budget. In an extended technical report [3], we quantify how these features can greatly reduce the cost of enforcing a given level of consistency compared to existing server replication protocols.

Figure 13 quantifies the remaining 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.) 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. Our workload is a series of writes by remote nodes in which all objects are equally likely to be written. Note that the cost of imprecise invalidations depends on the workload's locality: if there is no locality and writers tend to 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. Conversely, 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 show two cases: the *No Locality* line shows the worst case scenario, with no locality across writes, and the *burst=10* line shows 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 3.3, the CAP dilemma [9, 31] and performance/ consistency dilemma [22] describe fundamental 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 examines existing PR-AC [15, 18, 25], AC-TI [10, 17, 19, 26, 37, 39], and PR-TI [11, 29] approaches. These systems can be seen as special case "projections" of the more general PRACTI mechanisms.

Some recent work extends server replication systems towards supporting partial replication. Holliday et al.'s protocol allows nodes to store subsets of data but still requires all nodes to receive updates for all objects [14]. Published descriptions of Shapiro et al.'s consistency constraint framework focus on full replication, but the authors have sketched an approach for generalizing the algorithms to support partial replication [30].

Like PRACTI, the Deceit file system [31] 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, and all nodes replicating a file receive all bodies for all writes to the file.

A description of PRACTI was first published in an earlier technical report [7], and an extended technical report [3] describes the current system in more detail.

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.

Acknowlegements

We thank Lorenzo Alvisi, Eric Brewer, John Byers, Allen Clement, Ravi Kokku, Jean-Phillipe Martin, Jayaram Mudigonda, Lili Qiu, Marc Shapiro, Harrick Vin, Emmett Witchel, and Haifeng Yu for their feedback on earlier drafts of this paper.

References

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

[2] 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.

[3] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for large-scale systems (extended version). Technical Report UTCS-06-17, U. of Texas at Austin, Apr. 2006.

[4] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *ICDCS*, June 1992.

[5] 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.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, Oct. 2001.

[7] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for large-scale systems. Technical Report TR-04-28, U. of Texas at Austin, Mar. 2004.

[8] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Wkshp. on Internet Svr. Perf.*, June 1998.

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

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

[11] 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.

[12] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Inc., 2nd edition, 1996.

[13] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Sys.*, 12(3), 1990.

[14] J. Holliday, D. Agrawal, and A. E. Abbadi. Partial database replication using epidemic communication. In *ICDCS*, July 2002.

[15] 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.

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

[17] P. Keleher. Decentralized replicated-object protocols. In *PODC*, pages 143–151, 1999.

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

[19] 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.

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

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

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

[23] D. Malkhi and D. Terry. Concise version vectors in WinFS. In *Symp. on Distr. Comp. (DISC)*, 2005.

[24] 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.

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

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

[27] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. USENIX FAST*, Mar. 2003.

[28] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

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

[30] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. OPODIS*, Dec. 2004.

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

[32] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in Beehive. In *SPAA*, 1997.

[33] Sleepycat Software. *Getting Started with BerkeleyDB for Java*, Sept. 2004.

[34] 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 *SOSP*, Dec. 1995.

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

[36] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In *Proc. OSDI*, Dec. 2002.

[37] G. Wuu and A. Berstein. Efficient solutions to the replicated log and dictionary problem. In *PODC*, pages 233–242, 1984.

[38] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc SIGCOMM*, Aug. 2004.

[39] 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.