PRACTI Replication

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

Arun Venkataramani

University of Massachusettes at Amherst

Abstract

We present PRACTI, a new approach and architecture 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 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, by providing all three PRACTI properties, it enables *better trade-offs* than existing mechanisms that support at most two of the three desirable properties. PRACTI thus exposes new points in the design space for replication systems. Second, our architecture's *flexibility* 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 [25] as well as between availability and consistency [12, 35], 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 [24] 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.

This paper introduces the PRACTI taxonomy and 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 [29, 45].
- *Arbitrary Consistency* (AC) 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 [15, 32].
- *Topology Independence* (TI) means that any node can exchange updates with any other node. In contrast, many systems restrict communication to client-server [18, 21, 28] or hierarchical [4, 43] 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.

This paper presents PRACTI, the first replication architecture to provide all three properties. It does this 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 [29, 45]. 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

---

[1] Although the operating systems and distributed systems literature often use the terms consistency and coherence interchangeably, the architecture literature is more precise [16]: 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.

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 introduce *imprecise invalidations*, which allow a single invalidation to conservatively summarize a set of invalidations. Imprecise invalidations allow us to provide cross-object consistency in a scalable manner in which each node incurs storage and bandwidth costs proportional to the size of the data set in which it is interested. Using imprecise invalidations, a node $n1$ 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 any omitted invalidations to objects in B. The imprecise invalidation then serves as a placeholder for the omitted updates so that if $n1$ forwards information about the updates to A to another node $n2$ that is interested in both A and B, $n2$ can know which omitted updates to B it must fetch from another node.

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 AC-TI full replication replicated server systems, by providing more than an order of magnitude better synchronization delay compared to PR-AC topology constrained hierarchical systems, and by providing consistency guarantees not achievable by PR-TI limited consistency per-object replication systems.

More broadly, we argue that 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
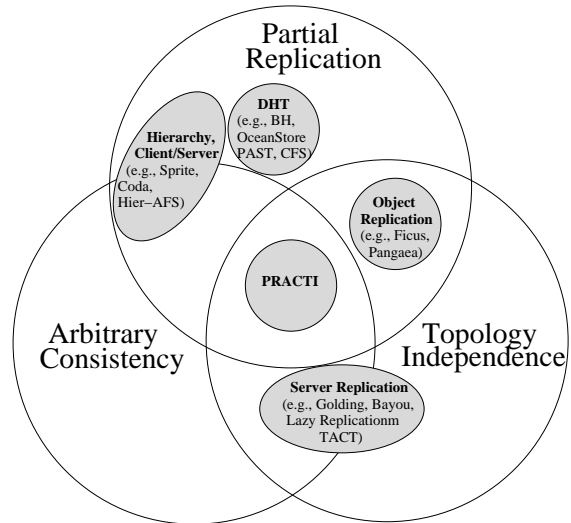

Fig. 1: The PRACTI taxonomy defines a design space for classifying families of replication systems.

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 provides a taxonomy for replication systems that explains why existing replication architectures fall short of ideal. 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 revisits the design of existing systems in light of the PRACTI taxonomy. 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 Taxonomy and challenges

As illustrated in Figure 1, the PRACTI paradigm defines a taxonomy for understanding the design space for replication systems. Although providing all three PRACTI properties has obvious potential benefits, we know of no existing system that does so. Most systems fall into categories that each provide at most two of the PRACTI properties:

Server replication systems like Golding's timestamped anti-entropy [13] and Bayou [29] 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. Lazy Replication [22] and TACT [45] 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 and storage, and they may be unsuitable for devices with limited resources.

Client server systems like Sprite [28] and Coda [21] and *hierarchical* caching systems like hierarchical AFS [26] permit caching of 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 client-server systems permit limited client-client communication for cooperative caching [10] they must still serialize control messages at a central server for consistency [5]. 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 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 [38], PAST [31], CFS [6], and OceanStore [30] 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, introduces 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 [30].

Object replication systems such as Ficus [15] and Pangaea [32] 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 semantics may prevent deployment of
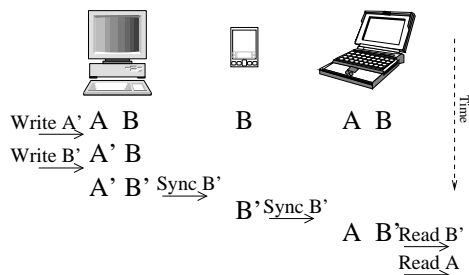

Fig. 2: Naive addition of PR to an AC-TI log exchange protocol fails to provide consistency.

applications with more stringent requirements.

Why is PRACTI hard? 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 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; 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 the naive attempt to add PR to a AC-TI server replication protocol like Bayou illustrated in Figure 2. 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 [25].

This example illustrates the broader, fundamental challenge: topology independence makes combining partial replication and arbitrary consistency hard because when a node receives updates, it must not only consis-
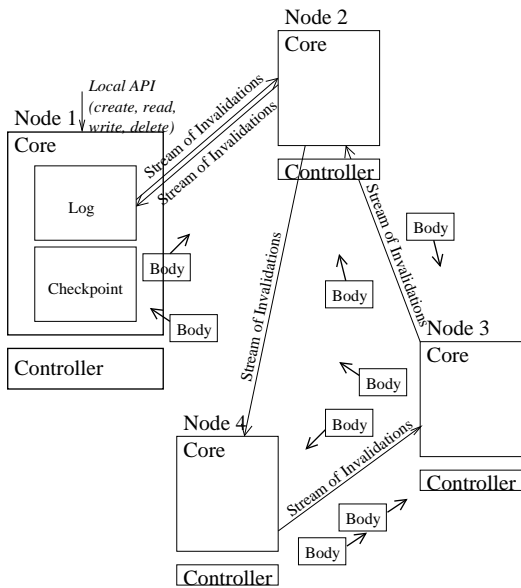
Fig. 3: High level PRACTI architecture.

tently 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 3 shows the high-level architecture of our PRACTI implementation.

Node 1 in the figure illustrates the main local data structures of each PRACTI node. Applications access data stored in PRACTI via the per-node *Local API* for creating, reading, writing, and deleting objects. These functions operate on local state stored in each node's *Log* of updates and random-access *Checkpoint*: modifications are appended to the log and then update the checkpoint, and reads access the checkpoint. To support partial replication policies, the PRACTI mechanisms allow each node to select an arbitrary subset of the system's objects to replicate, and nodes are free to change this subset at any time. The PRACTI mechanisms track

local state so that nodes can satisfy requests to read local, valid objects without needing to communicate with other nodes.

To handle read misses and to push new information between nodes, PRACTI makes use of 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 [29] 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 PRACTI's protocol differs from existing log exchange protocols two key ways:

1. *Separation of invalidations and bodies.* PRACTI invalidation streams notify a receiver that writes have occurred, but separate body messages contain the contents of the writes. 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, PRACTI nodes can omit sending groups of invalidations by instead sending *imprecise invalidations* that concisely and conservatively summarize the omitted invalidations. Imprecise invalidations serve as placeholders in the receiver's log (to prevent the invalidation streams that it transmits from containing causal gaps) and in the receiver's checkpoint (to allow the receiver to block reads of objects for which some invalidations may be missing.) Imprecise invalidations allow partial replication of metadata—a node only needs to receive traditional *precise invalidations* and store per-object metadata for objects that interest it.

Nodes select subsets of objects about which they want to store per-object metadata, and they select subsets of objects for which they want to prefetch body updates. They then receive streams of invalidations and body messages to maintain this local state. A node requests a stream of invalidations by sending a request that identifies the subset of objects for which the receiver desires to see precise invalidations. Body messages are initiated in two ways. First, if a local read blocks because an object is invalid, a node sends a demand-read request for that object to another node. Second, a node can set up a prefetch subscription with any other node in order to automatically receive body update messages for a specified subset of the system's objects.

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 PRACTI *Controller* embodies a system's replication and topology policies by directing communication among nodes. A node's controller implements a replication and consistency policy by (1) selecting which nodes should send it invalidations and, for each invalidation subscription, specifying subsets of objects for which invalidations should be full precise invalidations, (2) selecting which nodes to prefetch bodies from and which bodies to prefetch, and (3) selecting which node should service each demand read miss.

The rest of this section describes the design in more detail. It first explains how PRACTI'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 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, 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 Figure 3 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 [23] 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 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 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 [29].
- 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 [29]. 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 a *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 [45].

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 [29]. 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

---

[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.
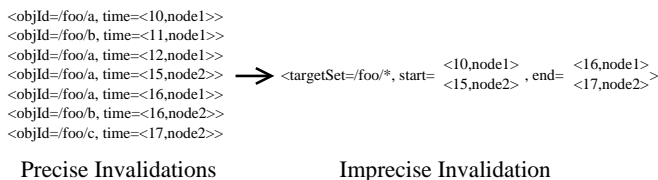
<objId=/foo/a, time=<10,node1>>
<objId=/foo/b, time=<11,node1>>
<objId=/foo/a, time=<12,node1>>
<objId=/foo/a, time=<15,node2>> → <targetSet=/foo/*, start= <10,node1> <15,node2> , end= <16,node1> <17,node2> >
<objId=/foo/a, time=<16,node1>>
<objId=/foo/b, time=<16,node2>>
<objId=/foo/c, time=<17,node2>>

Precise Invalidations   Imprecise Invalidation

Fig. 4: Example imprecise invalidation.

consistency properties of the log [45] 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*[3]. 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 [11] or DHT-based [38] directory, using a hint-based search strategy [33], or relying on a push-all strategy [29] (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 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 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 (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.

---

[3]To broaden the range of consistency semantics PRACTI can support, the read interface also provides a flag that indicates that a read of an INVALID object should return an exception rather than block.

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 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. 5 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 con-
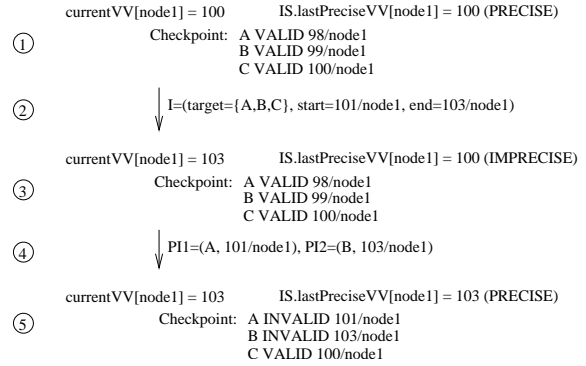

Fig. 5: Example of applying an imprecise invalidation I and then applying precise invalidations $PI1$ and $PI2$. For clarity, we only show node1's elements of $currentVV$ and $IS.lastPreciseVV$.

servatively 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 [9] 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 Consistency: Costs and approach

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 [12, 35]. Similarly, Lipton and Sandberg describe fundamental performance limitations for distributed systems that provide sequential consistency [25].

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 [45].

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 [29], adapting flexible consistency techniques from the literature is straightforward. We provide the TACT flexible consistency interface to bound order error and temporal error [45]; 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 [22, 45]; 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, 29].

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
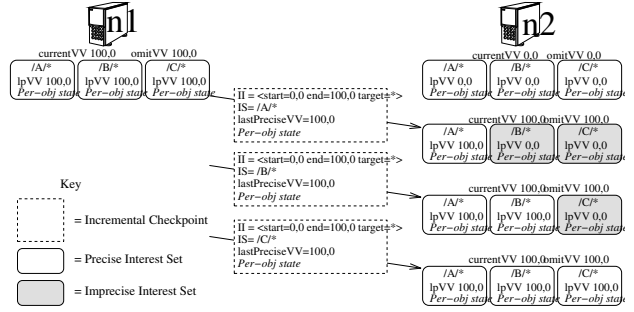

Fig. 6: Incremental checkpoints from $n1$ to $n2$.

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 [29], 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 6 illustrates, each incremental checkpoint 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 6 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 [39]. 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 [9].

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 [37]; 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 [36] for per-node local storage. The prototype is fully functional but not performance tuned. 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 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.
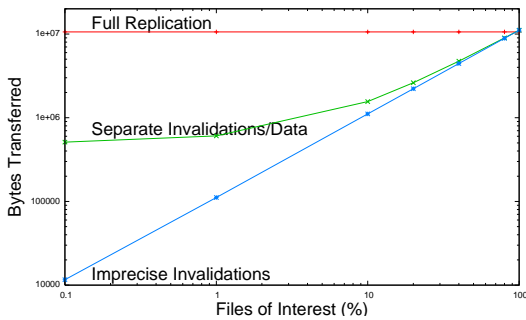
Fig. 7: Impact of locality on replication cost.

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

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" [3].
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 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 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
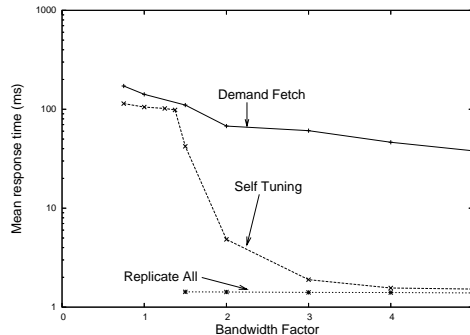
collection of 1000 files of 10KB each. A node subscribes to invalidations and body updates for the subset of the files that are "of interest" to that node. The x axis shows the fraction of files that belong to a node's subset, and the y axis shows the total bandwidth required to transmit these updates 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 7 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 [3]. 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 [9]. 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 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* [27] this system hosts, all up-

	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	50Kb/s
			1Mb/s	Hotel only
Palmtop	100MB	100KB	1Mb/s	N/A

Fig. 9: Configuration for mobile storage experiments.

dates 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 ($\alpha = 1.1$), 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 bandwidth corresponding to the current write rate for prefetching even if other applications could make better use of the network.

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 [29]. 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 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↔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.

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 11 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
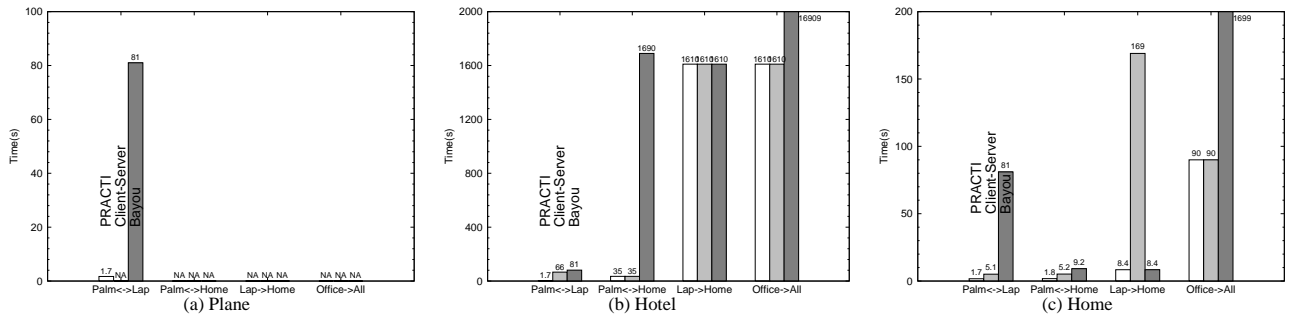
Fig. 10: Synchronization time among devices for different network topologies and protocols.

system [42] 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, 38]. 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 [29]). All 4 systems are implemented over PRACTI.

Figure 11 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

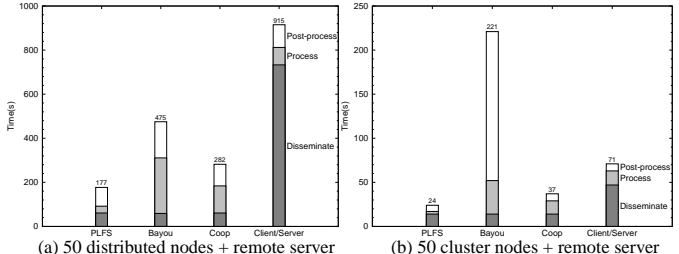This subsection first examines the benefits and then examines the costs of supporting flexible consistency.

Fig. 11: Execution time for the WAN-Experiment benchmark.

Improved consistency trade-offs. Gray [14] and Yu and Vahdat [44] 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 [27].

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 8. We use an average network path unavailability of 0.1% with Pareto distributed repair time R(t) = $1 - 15t^{-0.8}$ [7].

In Figure 12-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 [45]. We compare the *TACT Aggressive* policy [44] 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 12-b plots system unavailability for an order error of 100 as bandwidth varies. Following Yu and Vahdat's methodology [44], we say the system is *available* to a read or write request if the request can issue with-
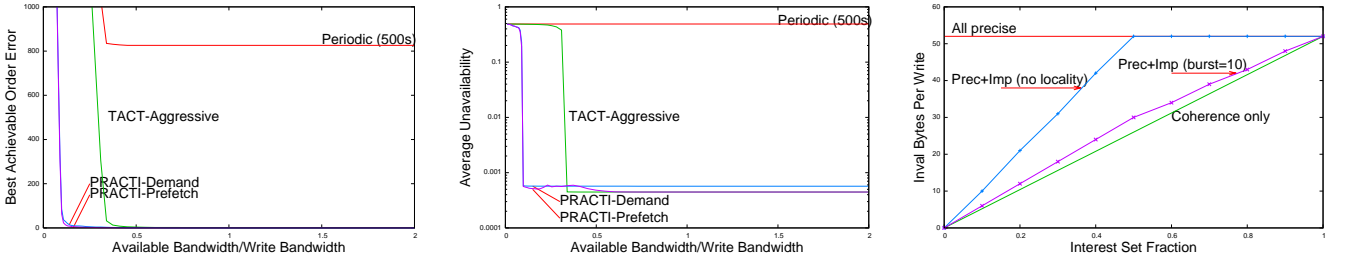
(a) Best consistency (order error) achievable for a given bandwidth cost.
(b) Best unavailability achievable while meeting a required order error of 100.
(c) Bandwidth cost of distributing consistency information.

Fig. 12: Consistency trade-offs (a-b) and costs (c).

out 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 3.3 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 [15, 32] 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 12-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 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 [12, 35] and performance/consistency dilemma [25] 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, 4, 10, 18, 21, 28], AC-TI [13, 20, 22, 29, 45], and PR-TI [15, 32] approaches. These systems can be seen as special case "projections" of the more general PRACTI mechanisms [8, 9].

Some recent work has focused on extending AC-TI 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 [17]. Published descriptions of Shapiro et al.'s consistency constraint framework focus on algorithms for full replication, but the authors have sketched an approach for generalizing the algorithms to support partial replication [34].

Like PRACTI, the Deceit file system [35] 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 group multicast 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 [40]. It will reportedly support synchronization across multiple nodes, however no detailed technical description of the protocol has been published. One report [41] 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.

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

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, 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.

[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 *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct. 2001.

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

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

[9] 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.

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

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

[12] 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.

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

[14] J. Gray, P.Helland, P. E. O'Neil, and D. Shasha. Dangers of replication and a solution. In *Proc. SIGMOD*, pages 173–182, 1996.

[15] 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.

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

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

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

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

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

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

[26] 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.

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

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

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

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

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

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

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

[34] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist. Sys.*, Dec. 2004.

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

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

[37] 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.

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

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

[40] http://msdn.microsoft.com/data/winfs/, Mar. 2005.

[41] http://longhorn.msdn.microsoft.com/lhsdk/winfs/consynchronizationoverview.aspx, Mar. 2005.

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

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

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

[45] 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.