PRACTI Replication

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

Arun Venkataramani
University of Massachusettes at Amherst

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

Abstract

We present the first PRACTI (Partial Replication, Arbitrary Consistency, and Topology Independent) replication toolkit. PRACTI mechanisms can replicate or cache any data on any node (PR), provide a broad range of consistency and coherence guarantees (AC), and permit any node to communicate with any other node at any time (TI). Compared to existing mechanisms that force a system designer to compromise a system's replication, topology, or consistency policy, PRACTI yields two significant advantages: *flexibility* and *improved trade-offs*. PRACTI's flexibility simplifies the design of replication systems by allowing a single framework to subsume a broad range of existing approaches. At the same time, PRACTI provides better trade-offs than the point-solution policies embedded in existing mechanisms: for workloads of interest, our PRACTI design dominates existing approaches by providing an order of magnitude better bandwidth and storage efficiency than replicated server systems (AC-TI), an order of magnitude better synchronization delay compared to hierarchical systems (PR-AC), and consistency guarantees not achievable by per-object replication systems (PR-TI).

1 Introduction

Data replication is a building block for a broad range of large-scale distributed systems such as mobile file systems, web service replication systems, enterprise file systems, or grid replication systems. Because there is a fundamental trade-off between performance and consistency [27] as well as between availability and consistency [8, 32], systems make different trade-offs among these factors by implementing different consistency policies, placement policies, and topology policies for different environments. Informally, *consistency policies* such as sequential [26] or causal [20] define which reads must see which writes, *placement policies* such as demand-caching, prefetching, push-caching, or replicate-all define which nodes store local copies of which data, and *topology policies* such as client-server, hierarchy, or ad-hoc define the paths along which communication flows.

This paper describes a set of mechanisms that for the first time simultaneously provide all three PRACTI (Partial Replication, Arbitrary Consistency, and Topology Independence) properties. *Partial replication* means that a system can place any subset of data on any node. In contrast, some systems require a node to maintain copies of all objects in all volumes they export [30, 44]. *Arbitrary consistency* means that the 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 [17, 31]. *Topology independence* means that any node can communicate with any other node. In contrast, many systems restrict communication to client-server [19, 22, 29] or hierarchical [7, 42] patterns.

We base PRACTI on log exchange mechanisms that support a range of consistency guarantees and topology independence but that fundamentally assume full replication [30, 44]. We adapt these mechanisms to support partial replication using two principles.

- First, in order to allow partial replication of data, we *separate the control path from the data path* by separating invalidation messages that identify what has changed from body messages that encode the changes to the contents of files.
- Second, in order to allow partial replication of update metadata, we use *explicit conservative encoding* via *imprecise invalidations*, which allow a single invalidation to summarize a set of invalidations.

We have constructed and evaluated a prototype. Our primary conclusion is that by disentangling mechanism from policy and simultaneously supporting the three PRACTI properties, *PRACTI replication enables better trade-offs for system designers than possible with existing mechanisms.* For example, for some workloads in our mobile storage and grid computing case studies, PRACTI dominates existing approaches by providing more than

[1] Although the operating systems and distributed systems literature often use the terms consistency and coherence interchangeably, the architecture literature is more precise [18]: 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.

an order of magnitude better bandwidth and storage efficiency than replicated server systems (AC-TI), by providing more than an order of magnitude better synchronization delay compared to hierarchical systems (PR-AC), and by providing consistency guarantees not achievable by per-object replication systems (PR-TI).

More broadly, by subsuming a large portion of the design space, the PRACTI toolkit 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, PRACTI can be thought of 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.

This paper makes three contributions. First, it describes novel mechanisms that support efficient and scalable PRACTI replication. To our knowledge past systems have provided at most two of the PRACTI properties. Second, it provides a prototype replication toolkit based on PRACTI that cleanly separates mechanism from policy and thereby allows nearly arbitrary replication, consistency, and topology policies. Third, it demonstrates that PRACTI replication provides two significant advantages over existing replication mechanisms: *flexibility* to simplify the design and deployment of replication systems and *better trade-offs* among performance, availability, and consistency than supported by existing mechanism-defined point-solution policies.

The rest of this paper is organized as follows. Section 2 describes the PRACTI mechanisms, and Section 3 details our implementation. Section 4 experimentally evaluates the prototype. Finally, Section 5 surveys related work and Section 6 highlights our conclusions.

2 PRACTI design

This section provides an overview of the PRACTI mechanisms. It focuses on PRACTI's basic mechanisms for transmitting updates among nodes while supporting partial replication, arbitrary consistency, and topology independence. Section 3 details how our prototype implements these mechanisms and also discusses important additional features such as our support for tunable consistency, hooks for enforcing a minimum degree of replication, implementation of conflict detection and resolution, and garbage collection of the logs.

2.1 Design overview

The backbone of the PRACTI protocol is a log-exchange framework that is similar to that of Bayou [30]: nodes exchange portions of their logs in order to propagate writes

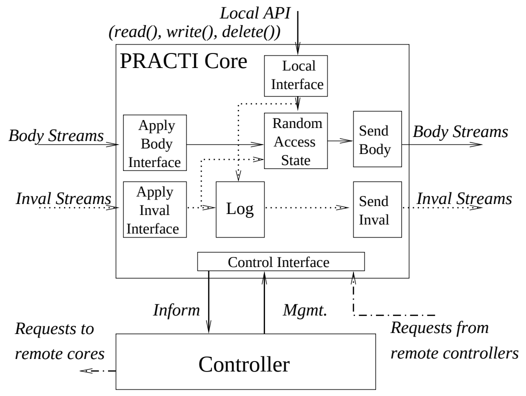


Fig. 1: High level PRACTI architecture for one node.

through the system. The use of log exchange supports arbitrary communication patterns among nodes, and it provides a basis for providing a range of consistency guarantees [44]. Unfortunately, existing log exchange mechanisms fundamentally assume full replication in order to maintain the invariant that each node's log represents a causally-consistent prefix of the system's writes.

The PRACTI mechanisms therefore differ from existing protocols in two key ways. First, in order to allow any node to efficiently receive updates about and store any subset of data, PRACTI mechanisms *separate the control path from the data path* by separating invalidation and body messages. Second, in order to allow any node to receive and store invalidations about any subset of data, PRACTI uses *explicit conservative encoding* via *imprecise invalidations*. The rest of this subsection first provides an overview of PRACTI's main data structures and interfaces and then describes these two mechanisms.

Data structures and interfaces. Figure 1 provides a high-level view of the PRACTI design. Each PRACTI node has two main components: the *core* and the *controller*. The *core* instantiates the basic PRACTI *mechanisms* by processing incoming messages and maintaining a local view of the system's state. The *controller* embodies a system's *policies* by initiating communication among nodes; different PRACTI deployments will use different controllers to implement different replication, topology, and consistency policies. This paper focuses on the core; Section 3 includes an overview of the controller, and the Evaluation section describes several example controllers.

The two main data structures in the core are the *random access state* and the *log*. The *random access state* (RAS) stores the current state of objects indexed by object ID.[2] A node uses its RAS for local reads and for sending data to other nodes. The *log* maintains a causally ordered list of updates that have been applied to the RAS.

[2]For simplicity, we describe the protocol in terms of full-object writes. In practice, we track RAS state, invalidations, and bodies on arbitrary byte ranges.

A node uses its log as a replay log for reliability and also for transmitting updates to other nodes. Additionally, as indicated in the figure and discussed in more detail below, a node maintains four interfaces—*local*, *applyInval*, *applyBody*, and *control*—for processing incoming messages. Finally, a node's *sendInval* and *sendBody* modules assemble and transmit outgoing messages.

Separate invalidations from bodies. The separation of invalidations from bodies allows partial replication of data: a controller can choose any policy for distributing bodies to nodes. To accomplish this separation, the PRACTI architecture splits *write messages* into two parts: (1) *invalidation messages* identify what objects were written and when the writes occurred and (2) *body messages* contain the bodies of the writes. Figure 2 in Section 3 defines the message types in more detail.

PRACTI distributes invalidations using a straightforward variation of Bayou's log exchange protocol that operates on invalidations rather than complete writes. When a node receives a new invalidation message, it applies the message to its log, and when it sends its log to another node, this log contains invalidations rather than complete writes. When a node receives an invalidation, then if the invalidation's logical timestamp exceeds the logical time for the object in the RAS, the node updates the object's RAS state by marking it *INVALID* and updating its logical timestamp. PRACTI thus retains three key invariants on invalidations [30]. First the *prefix property* requires that a node's state always reflects a prefix of the sequence of invalidations by each node in the system. Second, each node's local state always reflects *causally consistent* [20] view of all invalidations that have occurred. This property follows from the prefix property and from the use of Lamport clocks [25] 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. Given this basis, we can enforce a broad range of consistency semantics [44].

Although invalidations must be sent and applied in causal, logical timestamp order, PRACTI nodes can distribute bodies according to arbitrary policies, in arbitrary order, across arbitrary topologies. A PRACTI node must therefore synchronize arriving bodies with the invalidation streams before applying bodies to its local state. PRACTI maintains the invariant that update bodies are not applied to the RAS until after the corresponding invalidation message. To ensure this invariant, the core's applyBody interface buffers updates until they may be safely applied. When a node finally can apply a body message, if the logical time for the object in the RAS has not advanced past the body's logical time, the RAS marks the object *VALID* and stores the body.

The separation of bodies from invalidations affects local read requests. 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. A controller can implement any policy for doing this from a static hierarchy (i.e., ask your parent or a central server for the missing data) to a separate, centralized directory [13], to a DHT-based directory [36], to a hint-based search strategy, to a push-all strategy [30] (i.e., "just wait and the data will come.") In addition to distributing bodies in response to demand reads, controllers can also prefetch bodies, pre-push bodies, or pre-position bodies according to a global placement policy.

Imprecise invalidations and interest sets. Where separation of invalidations from bodies supports partial replication of data, imprecise invalidations allow partial replication of metadata: nodes receive precise invalidations for objects they plan to access but receive only summaries of invalidations for other objects. Although each invalidation is small, imprecise invalidations are crucial for large systems: our data in Sec. 4 show that imprecise invalidations can reduce replication costs by an order of magnitude compared to requiring every node to see every invalidation of every object.

An *imprecise invalidation* is a conservative summary of a group of ordinary invalidations, which we refer to as *precise invalidations.* We use the term *general invalidation* to refer to either a precise or imprecise invalidation. An imprecise invalidation includes a set of *targets* and a range of logical times defined by some *start* and *end* times, and it denotes that "One or more objects in *targets* was updated between *start* and *end*." An imprecise invalidation must *cover* all summarized invalidations—any invalidation summarized by an imprecise invalidation i must have its target ID included in $i.targets$ and its logical time included between $i.start$ and $i.end$. Notice that a general invalidation's start and end times are partial version vectors with as few as one element (to cover invalidations by one node) and as many as n elements (to cover invalidations by all nodes in the system). Also note that an imprecise invalidation i can be conservative: $i.targets$ can include objects that were not invalidated between $i.start$ and $i.end$. This rule supports concise encodings of large numbers of files (e.g., a list of subdirectories or a Bloom filter of object IDs).

When a node α receives an imprecise invalidation i, α applies i to both its log and its RAS. For the log, i serves as a "placeholder" so that if α sends its log to another node, i indicates which precise invalidations are omitted. Logs thus still maintain the prefix property, causal consistency, and eventual consistency invariants. The benefit of imprecision is efficiency: when a controller tells node α to send invalidations to node β, the request indicates subsets of the object ID space for which α should summarize invalidations using imprecise invalidations before sending them.

Imprecise invalidation i must also update the RAS. A naive strategy would mark every object covered by $i.target$ as *INVALID* to logical time $i.end$. Such an approach has two problems. The first is performance: the cost to process an imprecise invalidation would be proportional to its $target$ set size. The second problem is liveness: each invalidated object o would remain *INVALID* until the node receives a body for o with a logical time at least $i.end$. Note that typically, only one object in $i.targets$ actually was written as late as $i.end$ by the summarized invalidations; for any other object p in $i.target$, there may exist no write in the system that can make p *VALID*.

To address these performance and liveness problems, nodes use a more sophisticated approach: nodes allow portions of the RAS to include stale state after an imprecise invalidation, but they ensure consistency by preventing observation of stale RAS entries. To do this, a node partitions its RAS into one or more *interest sets*. An *interest set* is a portion of the object ID space that is either *PRECISE* or *IMPRECISE*. An interest set IS is *PRECISE* if and only if the RAS reflects all precise invalidations for all objects in IS up to the node's current logical time. For consistency, a local read of an object must block until the enclosing interest set is *PRECISE*; when a read blocks, the controller must initiate sufficient communication in order to make the interest set *PRECISE* and to allow the read to complete. We detail the algorithm for tracking interest set status in Section 3.2.

Note that enforcing stronger consistency than required can hurt availability [8, 43]. Therefore, as an optimization, PRACTI provides an additional local interface to issue an *imprecise read* that skips the *PRECISE* check just described and returns as soon as the requested object is *VALID*.

2.2 General framework

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

AC-TI. Server-replication systems such as Bayou [30], TACT [44], and lazy replication [24] allow arbitrary communication between nodes and can provide flexible consistency, but they fully replicate all objects in a volume and send all updates to all nodes that serve the volume. In the PRACTI framework, these AC-TI protocols can be viewed as using a replicate-all strategy for both precise invalidations and bodies, never sending or receiving imprecise invalidations, and not implementing any mechanism to handle read misses because objects are always *PRECISE* and *VALID*.

PR-AC. Client-server and hierarchical systems such as AFS [19], Sprite [29], and Coda [22] allow nodes to cache or prefetch arbitrary subsets of data and in principle could support a range of consistency policies [41] (though, in practice, such systems typically implement a specific consistency policy). But these protocols fundamentally assume a topology policy that restricts communications to hierarchical paths. Even when client-server systems permit limited client-client communication for cooperative caching [12] serialization of control messages at the server is vital for reasoning about consistency [9]. In the PRACTI framework, these PR-AC protocols can be viewed as using separate invalidation and body messages, with invalidations sent by parents to children and bodies fetched by children from parents. Their callback protocols can be viewed as specialized instances of PRACTI's sendInval module that actively track which objects a child caches and that send precise invalidations only for those objects. Note that in PRACTI, the module would also send imprecise invalidations covering any omitted precise invalidations, but the hierarchical topology allows PR-AC protocols to omit these implicit imprecise invalidations. Interestingly, recovery when a server loses callback state [4] or when a topology changes [42] falls back on what are essentially explicit imprecise invalidations: the client receives a message (i.e., an imprecise invalidation covering all objects) indicating that it should treat all of its consistency state as suspect (i.e., *IMPRECISE*) and the client then revalidates all objects with its server (i.e., make the interest set precise).

PR-TI. Object replication systems such as Ficus [17] and Pangaea [31] maintain synchronization information separately for each object and support arbitrary topology policies and arbitrary placement of objects on nodes. However, although these systems can provide some *coherence* guarantees on the order of reads and writes when an individual object is considered, they provide limited *consistency* guarantees regarding the ordering of reads and writes across objects. Furthermore, these systems cleanly separate invalidations and body messages: for any given object o and node η they either propagate o's

writeTarget	objId, offset, length
acceptStamp	logicalClock, nodeId
preciseInvalidation	writeTarget, acceptStamp, prevAccept, realTime
boundInvalidation	writeTarget, acceptStamp, prevAccept, realTime, body
impreciseInvalidation	targetSet, start[], end[], real[]
invalStream	startVV [generalInval]*
bodyMsg	writeTarget, acceptStamp, body

Fig. 2: Basic data types and messages.

update bodies to η or propagate no information at all about o's updates to η. In the PRACTI framework, these PR-TI protocols can be viewed as using a replication policy that sends invalidations and bodies for a given object to the same policy-specified subset of nodes and also as omitting all imprecise invalidations and thereby giving up the ability to consistently order writes across different objects.

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

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

3 Implementation

We have constructed a prototype of the PRACTI system written in Java and using BerkeleyDB [34] for per-node local storage. The prototype is fully functional but not performance tuned. All features described in this paper are implemented including local read/write/delete, flexible consistency, incremental log exchange, bound invalidations, remote read and prefetch, garbage collection of the log, checkpoint transfer between nodes, and crash recovery. For simplicity, we continue to describe the protocol in terms of whole-object reads and writes, but our prototype actually tracks object state on the granularity of arbitrary byte-ranges.

Section 3.1 first provides an overview of a number of basic features that are implemented using standard techniques from the literature as well as two novel but generally applicable enhancements on existing techniques: incremental log propagation and self-tuning body propagation. Then, Sections 3.2 and 3.3 dive into low-level details of the Core and Controller implementation.

3.1 Basic features

Because the system is built over a solid and well-explored framework of causally consistent log exchange, it is straightforward to include most of the following features by extending techniques described in the literature. Due to space constraints, we will only briefly outline these aspects of the implementation here and defer details to an extended technical report [11].

Incremental log propagation. The PRACTI prototype implements a novel incremental 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 RAS 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 RAS. In contrast, PRACTI's incremental log exchange protocol, which we detail in the next subsection, applies each incoming write to the current RAS state without requiring roll-back and roll-forward of existing writes.

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 PRACTI could be modified to use batch log exchange with roll-back and roll-forward. The advantages of the incremental approach are efficiency (each write is only applied to the RAS once) and concurrency (a node can process information from multiple continuous streams.) The advantage of the batch approach is flexible conflict detection: Bayou writes contain a *dependency_check* procedure that can read any object to determine if a conflict has occurred [35]; 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. For our incremental algorithm, we simply detect write/write conflicts when a write's *prevAccept* stamp (set by the original writer to equal the accept stamp of the overwritten value) differs from the accept stamp the invalidation overwrites in the RAS.

Self-tuning body propagation. In addition to supporting demand-fetch of particular objects, our prototype provides a novel self-tuning prefetching mechanism. A node α subscribes to updates from a node β by sending a list L of directories of interest along with a $startVV$ version vector. β will then send α any bodies it sees that are in L and that are newer than $startVV$. To do this, β maintains a priority queue of pending sends: when a new eligible body arrives, β 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 α via a low-priority network connection that ensures that prefetch traffic does not consume network resources that regular TCP connections could use [37]. 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.

Flexible consistency. We provide flexible consistency on a per-read/per-write basis by providing several read and write interfaces. We provide the TACT flexible consistency interface to bound order error and temporal error [44]; 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 [24, 44]; 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.

Write commitment. As in Bayou [30], PRACTI provides eventual consistency: for any write w, eventually all nodes will agree on a total order of all writes preceding w. A node considers a write w *committed* when the node knows w's final position in the global total order. For simplicity, we use Golding's algorithm [15]: each node η maintains a $currentVV$ version vector, and each entry $currentVV_\alpha$ stores the highest accept stamp of any invalidation by α that η has processed. Then, any write whose accept stamp is less than the lowest entry in $currentVV$ is *committed*. Supporting other write commitment protocols such as primary commit [30] or voting [21] would be straightforward, but we have not implemented these variations yet.

Bound writes. Separating invalidations from updates enables partial replication but also raises the issue of reliability: in Bayou, all nodes have copies of all data, but a PRACTI system must enforce an explicit policy decision about the minimum acceptable level of replication so that the loss of a node or a local cache replacement decision does not render some data unavailable or the storage system unreliable. We provide a simple, low-level mechanism that supports a broad range of high-level policies from maintaining a fixed number of "gold" copies of each object [31] to propagating all data to a well-provisioned central server [19] or replicated server "core" [22, 23] to replicating everything to everyone [30]. When an application issues a bound write, it creates a *bound invalidation* that includes the body of the write. Bound invalidations propagate through the system using log exchange and controllers manage this propagation to meet replication requirements. A controller can later issue messages to unbind a write, after which the invalidation can propagate without the body.

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

3.2 Core details

The PRACTI core draws heavily from existing log-exchange literature [30, 44] with two key changes: the separation of invalidations and update bodies and the use of imprecise invalidations.

Separating invalidations and bodies is straightforward. As Section 2.1 describes, the system transmits causally-consistent streams of invalidations using a streaming version of the Bayou protocol, maintains a *VALID/INVALID* flag for each stored object, transmits prefetched and demand fetched bodies in arbitrary orders, and delays applying bodies to the RAS until the corresponding invalidation has been applied.

Imprecise invalidations, however, raise four additional issues that we address in the rest of this subsection: We first define imprecise invalidations and describe how to form them. We next describe how nodes track which interest sets are *PRECISE* to enforce consistency. We then describe how systems manage their local logs using *per-writer logs*, *intersection*, and *gap filling* to properly merge data received on different invalidation streams. Finally, we describe how imprecise invalidations allow incremental checkpoint transfer among nodes.

Forming imprecise invalidations. PRACTI forms an imprecise invalidation I by combining generalized invalidations A and B. I has *start* and *end* arrays with entries for every node η in either A or B's *start*, and $I.start_\eta = min(A.start_\eta, B.start_\eta)$, and $I.end_\eta = max(A.end_\eta, B.end_\eta)$. Finally, $I.target$ encompasses all objects encompassed by A and B's *targets*.

When a controller asks node α to send a stream of invalidations to node β, the controller specifies two parameters that each filter the transmitted information: $startVV$ provides a filter on logical time, and $preciseFilter$ provides a filter on the ID space. α replies with a causally consistent stream of all invalidations it knows about that logically occurred after $startVV$. Invalidations whose target intersects $preciseFilter$ are sent as is (typically they are precise, but some may be imprecise), but α combines other invalidations into imprecise summaries as just described. This process is incremental and continuous—as new invalidations arrive at α, α sends them on to β once all causally prior invalidations have been sent.

Interest set status. As Section 2.1 indicates, each node groups its objects into *interest sets* and applies imprecise invalidations to interest sets rather than individual objects to (a) improve performance and (b) ensure liveness. To accommodate different workloads across nodes, our prototype allows each node to independently group objects into interest sets and to dynamically split and join interest sets in response to workload changes. To ensure consistency, a node must mark an interest set *IMPRECISE* when a new imprecise invalidation intersects with it. To ensure liveness, when a node has later seen sufficient precise invalidations, it must mark interest set as PRECISE.

To explain how interest set status is tracked, we now detail a node's algorithm for processing an incoming stream of invalidations.[3] As indicated in Figure 2, each incoming invalidation stream consists of a logical start time $startVV$ followed by a series of general invalidations gi_1, gi_2, ... such that any invalidation whose start time logically occurs after $startVV$ and on which gi_i causally depends appears before gi_i.

At the core of the algorithm is a simple idea: an interest set is *PRECISE* if it has missed no precise invalidations. Three variables are therefore central to processing an invalidation stream: (1) The **global** $currentVV$ version vector holds the highest logical time observed by the system across all invalidations processed from all streams. (2) The **per-interest-set** *last precise version vector* ($IS.lpVV$) indicates the highest logical time for which interest set IS is *PRECISE*. In particular, $IS.lpVV$ holds the highest logical time such that all objects in interest set IS reflect all writes up to $IS.lpVV$. An interest set IS is regarded as *PRECISE* if and only if $IS.lpVV = currentVV$. Otherwise, the interest set may have missed one or more precise invalidations, and we regard the interest set as *IMPRECISE*. (3) The **per-stream** $stream.prevVV$ variable always holds the logical time just *before* the next invalidation in the stream is applied. Each invalidation gi is processed in the context of the logical time at which it was applied to determine if gi can advance $IS.lpVV$. $stream.prevVV$ is initialized to the stream's $startVV$ and advanced to include $gi.end$ as each gi is processed.

[3]The extended technical report [11] includes detailed pseudo-code for this invalidation stream processing algorithm.

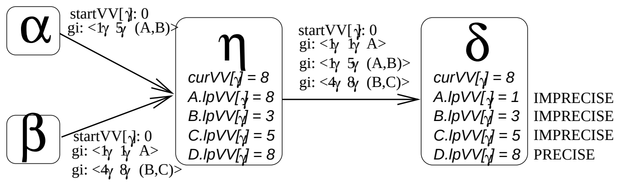
For each general invalidation gi, the log, the per-object state, and the interest set status must be updated. Updating the per-object state was described in Section 2.1, and we will discuss updating the log in a moment. The remaining issue is updating the per-interest set *PRECISE* state (i.e., updating $currentVV$ and one or more $lpVV$'s). This state is updated in two phases.

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

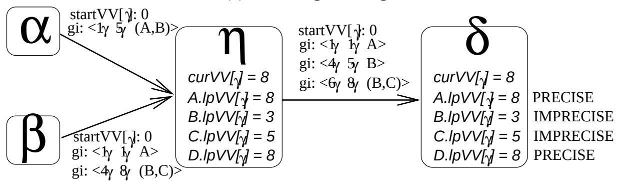
Notice that an imprecise invalidation gi will always advance $currentVV$ to include gi's *end* time but can at most advance $IS.startVV$ to just before gi's *start* time. It is this difference that allows imprecise invalidations to make interest sets *IMPRECISE*. If we stopped here, an imprecise invalidation would make both interest sets it overlaps and interest sets it does not overlap *IMPRECISE*. The algorithm addresses this issue by buffering each imprecise invalidation after it is first applied at its start time and applying a buffered invalidation bi again once $stream.prevVV$ includes bi's end time (i.e., once all gis whose start times preceed bi's end time have been processed.) Buffered invalidation bi advances $IS.lpVV$ to include $bi.end$ for any interest set IS that (a) $bi.target$ does *not* intersect and that (b) is *PRECISE* as of logical time $stream.prevVV$. Notice that by waiting until bi's end time before advancing "nonoverlapping" invalidations to the end time, we avoid erroneously advancing $IS.lpVV$ for an interest set that becomes *IMPRECISE* between $bi.start$ and $bi.end$.

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

Log update. Imprecise invalidations complicate log updates. For example, a node η may receive different subsets of information from different peers α and β. η



(a) Naive log exchange.



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

Fig. 3: Example log exchange when node η first receives a log from α, then receives a log from β, and then sends the combined log to δ. Generalized invalidations have three fields: $< startendtarget >$. Note that all writes were issued by node γ and, for clarity, we show only γ's component for all version vectors.

must ensure that imprecise invalidations received from α do not "mask" precise invalidations received from β and vice versa. Notice that the algorithm just described updates a node's local state by interpreting each gi relative to a per-stream $stream.prevVV$, which allows the algorithm to infer that there are no missing invalidations between $stream.prevVV$ and gi. But, if η were simply to store each gi in its log, some of this valuable "no missing invalidations" information could be lost. Then, as Figure 3-(a) illustrates, if η were to send its log to some other node δ, then even if δ receives the same gis as η, δ could end up *IMPRECISE* where η is *PRECISE* (e.g., for objects A).

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

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

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

Third, each per-writer log uses *intersection* to combine information received across multiple streams. In particular, we maintain the invariant that there is at most one invalidation that covers any moment in time in a per-writer log. We intersect two general invalidations a and b by replacing them with up to three general invalidations: the first covers the time from the earlier start to the later start and targets the objects targeted by the earlier start; the second covers the time from the later start to the earlier end and covers targets represented by the intersection of a and b's targets; and the third covers the time from the earlier end to the later end and covers the targets of the later end.

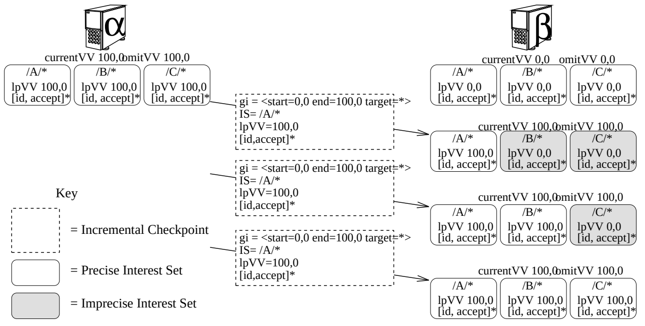


Fig. 4: Incremental checkpoints from α to β.

As Figure 3-(b) illustrates, when a node sends a stream of invalidations to another node, it discards gap-filling invalidations and it combines per-writer invalidations into multi-writer invalidations.

Incremental checkpoint transfer. Imprecise invalidations yield an unexpected benefit: incremental checkpoint transfer.

As in Bayou, PRACTI nodes can garbage collect any committed prefix of their logs [30]. Under Bayou, if a node α garbage collects log entries older than $\alpha.omitVV$ and another node β requests a log exchange beginning before $\alpha.omitVV$, then α must perform a full checkpoint transfer of its state for all objects; this transfer brings β's state up to $\alpha.currentVV$.

Rather than transferring information about all objects, PRACTI incremental checkpoints can include logical timestamp information for individual interest sets. As Figure 4 illustrates, each incremental checkpoint includes an imprecise invalidation that covers all objects from the receiver's $currentVV$ up to the sender's $currentVV$ and an interest set tranfer that includes the sender's $lpVV$ and per-object logical timestamps for some interest set IS. The receiver's $currentVV$ and $IS.lpVV$ are thus brought up to include the sender's $currentVV$ and $IS.lpVV$.

Overall, this approach makes checkpoint transfer a much smoother process under PRACTI than under Bayou: 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 4 illustrates.

3.3 Controller

Our PRACTI techniques cleanly separate mechanism from policy in order to support a broader range of replication, topology, and consistency policies than made available by current techniques. Our implementation therefore seeks to serve as a “replication microkernel” that provides basic low level mechanisms over which higher-level services can be built. As Figure 1 illustrates, the PRACTI prototype achieves this goal by splitting the design into a *core* and a *controller*.

The PRACTI core’s mechanisms enforce their safety properties regardless of what incoming messages they see. Our cores use an asynchronous style of communication in which incoming messages or streams are self-describing—the rules for processing each incoming message are completely defined, and interpreting a message does not require knowledge of what request triggered its transmission. Any machine can therefore send any legal protocol message to any other machine at any time.

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

The controller is defined by its interface. Within this interface, different implementations provide different policies. Controllers use three sets of interfaces to accomplish their work: a core calls a controller’s *inform* interface to inform the controller of important local events like message arrival or read miss, a controller calls a remote core’s *remote request* interface to trigger sends of invalidation streams or bodies, and a controller calls its core’s *management* interface for maintenance functions like garbage collection and interest set split/join. Additionally, a set of controllers implementing a specific distributed policy may communicate with one another using policy-specific interfaces. We provide several concrete example controllers in Section 4 and describe the interface in more detail in the extended report [11].

4 Evaluation

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

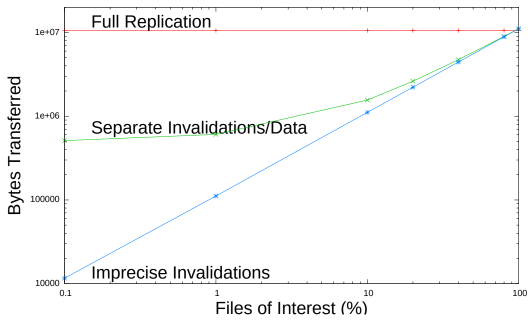


Fig. 5: Impact of locality on replication cost.

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

4.1 Partial replication

In this section, we focus on partial replication. We find that PRACTI’s support for partial replication dramatically improves its performance compared to the full replication protocols from which it 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” [5].
3. *Self-tuning Replication:* self-tuning replication minimizes response time for a given bandwidth budget.

Locality of reference. Different devices in a distributed system often access different subsets of the system’s data because (a) different users use different devices (e.g., in a corporation, user A’s laptop may access different files than user B’s laptop) and (b) different devices may have capacity or functionality constraints that influence the data that they access (e.g., a palmtop device may be useful for storing phone numbers and text notes but it may be less well suited for browsing a spreadsheet or editing a home video.) 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 5 examines the impact of locality on replication cost for three systems: 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

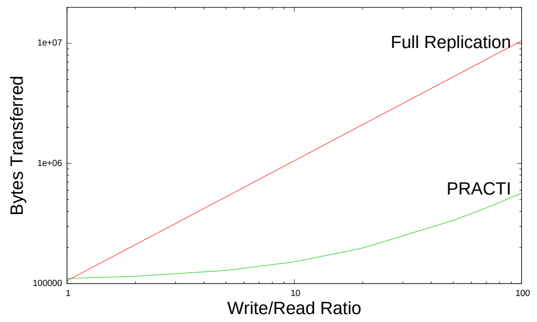


Fig. 6: Bandwidth cost of distributing updates as the number of writes to a file between reads varies.

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 updates for the subset of the files that are "of interest" to that node. The x axis shows the fraction of updates 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.

This experiment shows that partial replication of both bodies and invalidations is important when devices exhibit locality of interest—each of these factors can yield order of magnitude improvements. When a node subscribes to between 10% and 100% of data, partial replication of bodies allows the bandwidth cost of replication to fall nearly linearly with the size of the subscription set. But, for smaller subscription sets full replication of 30 to 50-byte precise invalidations limits gains. Conversely, PRACTI's imprecise invalidations allow replication bandwidth cost to fall nearly linearly with subscription set size.

Note that Figure 5 shows bandwidth costs of replication. Partial replication provides similar improvements for space requirements (graph omitted for space.) A PRACTI node need not store a body if an object lies in an *IMPRECISE* interest set or if the object is *INVALID*. Similarly, a node does not track per-object metadata for *IMPRECISE* interest sets that it does not plan to access.

Bytes die young. Bytes are often overwritten or deleted soon after creation. For example, in an academic environment, between 50% and 70% of written data survive for more than 1 minute, and between 10% and 60% survive for more than 10 minutes [5]. 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. As Figure 6 shows, PRACTI's gains are significant when bytes die young. For example, when the write to read ratio is 2, PRACTI uses 55% of the bandwidth of full replication, and when the ratio is 5, PRACTI uses 24%. At ratios exceeding 20, PRACTI's gains exceed an order of magnitude.

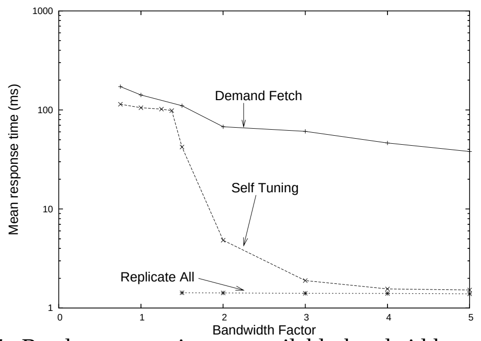
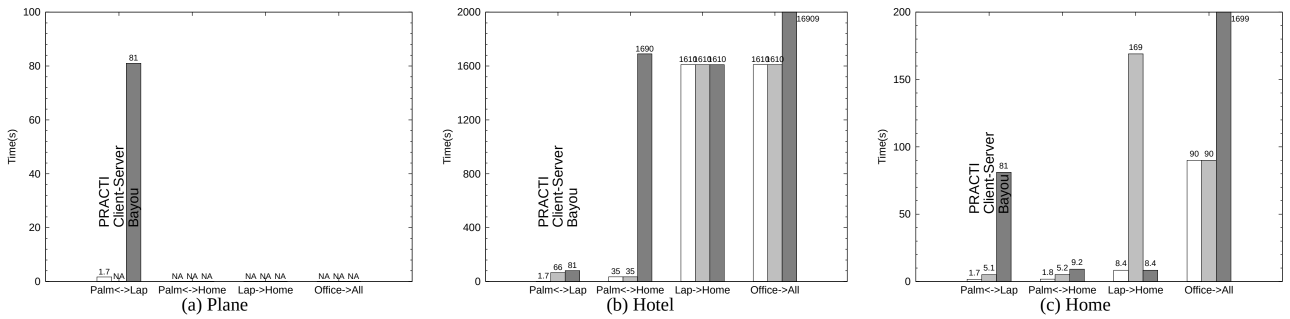


Fig. 7: Read response time as available bandwidth varies for full replication, demand reads, and self-tuning replication.

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

Figure 7 illustrates the benefits of this approach by comparing the read response time for three replication policies: *Demand Fetch* replicates precise invalidations to all nodes but sends new bodies only in response to demand requests, *Replicate All* replicates both precise invalidations and all bodies to all nodes by marking all invalidations as *bound*, and *Self Tuning* replicates precise invalidations to all nodes and has all nodes subscribe for all new bodies via the self-tuning mechanism. For this experiment, we model a producer/consumer access pattern where one node writes and another reads. 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 7 shows, when sufficient bandwidth is available, self-tuning replication can improve response time by up to a factor of 20 compared to *Demand-Fetch*. A key challenge, however is ensuring that prefetching does not overload the system. Whereas PRACTI's 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 attempts to consume bandwidth corresponding to the current write rate for prefetching even if other applications could make better use of the network. And even the replication system suffers: when bandwidth equals the average write rate, 37% of *Replicate All*'s requests see stale data (compared to less than



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

Fig. 8: Synchronization time among devices for different network topologies and protocols.

4% for *Self-Tuning*.)

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 it to dominate client-server and full replication approaches. In particular, PRACTI's support for topology independence yields advantages over hierarchical topologies for two reasons:

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

This section focuses on topology, and demonstrates PRACTI's advantages over topology-restricted hierarchical systems. 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 further discuss this subset of the results.

Mobile storage. Figure 8 evaluates 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) [22] and to a full-replication Bayou-like system (that can distribute updates directly between interested nodes but that requires full replication) [30]. All three systems are realized by implementing different controller policies over PRACTI.

As summarized in Figure 9 our synthetic 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 1% of that 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.

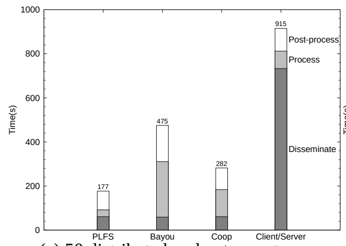
	Storage	Dirty Data	Wireless	Internet
Office server	1000GB	100MB	10Mb/s	100Mb/s
Home desktop	10GB	10MB	10Mb/s	1Mb/s
Laptop	10GB	10MB	10Mb/s 1Mb/s	50Kb/s Hotel only
Palmtop	100MB	100KB	1Mb/s	N/A

Fig. 9: Configuration for "mobile storage" experiments.

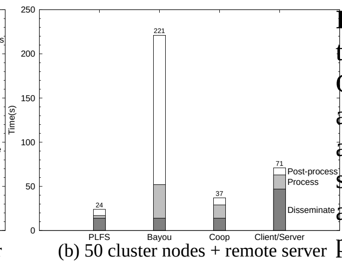
Figure 8 charts the time to synchronize dirty data among machines in four scenarios: (a) *Plane*—the user is on a plane with no Internet connection, (b) *Hotel*—the user's laptop has a 50Kb/s modem connection to the Internet, and (c) *Home*—the user's home machine has a 1Mb/s connection to the Internet. (Due to space constraints, we omit case (d) *Office*—the user's office has a 100Mb/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→*: the office server sends updates to all other machines.

In comparing the optimized 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 topology; in the *Hotel* location, direct synchronization between these two devices is an order of magnitude faster than synchronizing via the server (1.7s v. 66s); and in the home location directly synchronizing co-located devices is between 3 and 20 times faster than client-server synchronization.

WAN-FS for Researchers. Figure 10 evaluates a wide-area-network file system called PLFS designed for PlanetLab and Grid researchers. The controller for PLFS



(a) 50 distributed nodes + remote server



(b) 50 cluster nodes + remote server

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

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 [40] 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 [3]. Like Shark [3], 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 [3]), and server-replication (e.g., a Bayou-like system [30]). All 4 systems are implemented over PRACTI.

Figure 10 shows performance for an experiment running on (a) 50 distributed nodes each with a 5.6Mb/s connection to the Internet (we emulate this case by throttling bandwidth) and (b) 50 "cluster" nodes at university X with a switched 100Mbit/s network among them and a shared path via Internet2 to the origin server at university Y. PLFS's combination of partial replication and topology independence allows it to dominate the other designs. Compared to client/server, it is faster in both the Dissemination and Process phases due to its multicast transmission and direct 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 Processing phase because data can be transferred directly between nodes.

4.3 Arbitrary consistency

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

Improved consistency trade-offs. PRACTI improves the range of consistency available for replication. Gray [16] and Yu and Vahdat [43] show a trade-off: aggressive propagation of updates improves consistency and availability but can also increase system load. Yu's study finds an order of magnitude improvement in unavailability for some workloads when using aggressive propagation of updates compared to lazy propagation and Gray shows that the number of conflicts can rise with the square of propagation delay.

PRACTI has three features that improve the overhead versus consistency and availablity trade-offs: (1) separation of invalidations from bodies allows invalidations to propagate aggressively, (2) streaming log exchange (rather than batch [30]) 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.

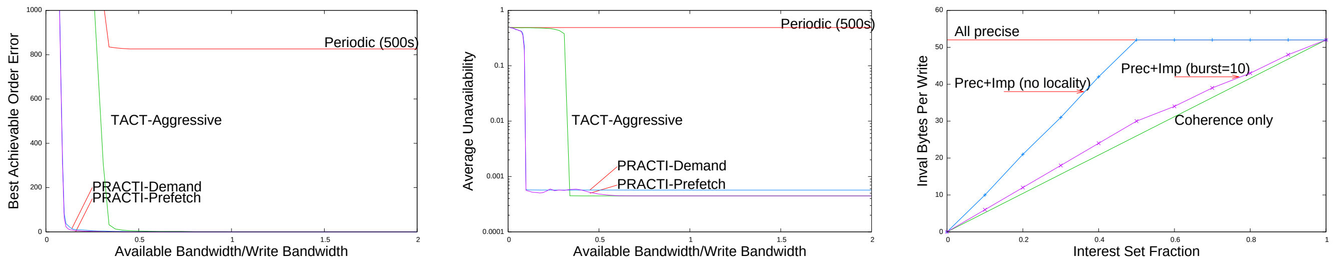
We examine a range of consistency requirements and network failure scenarios via simulation (all other experiments in this paper are prototype measurements.) We use a synthetic read/write workload with the same parameters as the workload used in Fig 7. We use an average network path unavailability of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

In Figure 11-a we measure the best order error that can be maintained for a given bandwidth budget. Order error constrains the number of oustanding uncommitted writes [44]. We compare the *TACT Aggressive* policy [43] to a *PRACTI Prefetch* policy that aggressively distributes invalidations as in TACT's policy but that distributes bodies using the self-tuning approach. PRACTI reduces the bandwidth needed to maintain reasonable consistency by a factor of 3 compared to *TACT Aggressive* and improves the consistency bounds attainable for some bandwidth budgets by orders of magnitude.

Figure 11-b plots system unavailability for an order error of 100 as bandwidth varies. Following Yu and Vahdat's methodology [43], we say the the system is *available* to a read or write request if the request can issue without blocking and the system is *unavailable* if the request must block in order to meet the consistency target. When bandwidth is limited, PRACTI dramatically improves system availability under consistency constraints compared to full replication.

Consistency overheads. Different applications require different consistency and coherence guarantees. Providing stronger guarantees than needed may hurt system availablity [8] or response time [27], and distributing more consistency information than needed can increase network overheads.

Our PRACTI prototype addresses the first two issues



(a) Best consistency (order error) achievable for a given bandwidth cost.
(b) Best unavailability achievable while meeting a required order error of 100.
(c) Bandwidth cost of distributing consistency information.

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

by shipping bookkeeping information around the system for all updates in the form of precise invalidations and imprecise invalidations but then only enforcing consistency constraints on a flexible, per-request basis by specifying whether a read should be precise (consistent) or imprecise (coherent) and by specifying TACT requirements on reads and writes [44].

Distributing sufficient bookkeeping information to support demanding requests does impose a modest cost whether requests require the information or not. In particular, object replication systems [17, 31] do not provide cross-object consistency guarantees. In the context of the PRACTI protocol, if all applications in a system only care about coherence guarantees, the system could completely omit imprecise invalidations.

Figure 11-c quantifies the prototypes 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 "Locality 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 in these experiments imprecise invalidations contribute an additional 20 bytes per average update. Overall, the difference in invalidation cost is likely to be small relative to the toal bandwidth consumed by the system to distribute bodies.

5 Related work

Replication is fundamentally difficult. For example Siegel [32] proves what has come to be known as the CAP dilemma [8]: a replication system that provides sequential **C**onsistency cannot simultaneously provide 100% **A**vailability in an environment that can be **P**artitioned. Similarly, Lipton and Sandberg describe fundamental performance limitations for distributed systems that provide sequential consistency [27]. 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 2.2 compares PRACTI with existing PR-AC [2, 7, 12, 19, 22, 29], AC-TI [15, 21, 24, 30, 44], and PR-TI [17, 31] approaches. As noted there, these systems can be seen as special case "projections" of the general PRACTI mechanisms, so ideas relating to PRACTI's mechanisms can be seen in these systems. For example, the separation of invalidations from bodies is standard in client-server systems [19, 29], and imprecise invalidations are closely related to messages sent by client-server systems during callback-state recovery [4, 41]. Several systems have noted the value of separating data and metadata paths [2, 31].

Like PRACTI, the Deceit file system [32, 33] provides a flexible substrate that subsumes a range of its contemporary replication systems. Deceit, however, focuses on replication across a handful of well-connected servers, and it therefore makes very different design decisions than PRACTI. For example, each Deceit server maintains a list of all files and of all nodes replicating each file, communication among servers is via an Isis [6] group for each distinct subset of servers, and all nodes replicating a file receive all bodies for all writes to the file.

Microsoft has announced that a new replication system, WinFS, will appear at some future date [38]. It will reportedly support synchronization across multiple nodes, however no detailed technical description of the protocol have been published. One report [39] suggests that the system transfers sets of updated items "rather than maintaining and synchronizing a log of each individual action," which may indicate that WinFS takes a PR-TI approach.

The current PRACTI mechanisms support a broad range of replication techniques such as client/server, server replication, object replication, 2-phase commit, and TACT, but it is not clear if the current protocols can easily support some other replication techniques such as quorum replication [14] or Byzantine storage [1, 28]. Broadening the PRACTI approach to include such techniques may be interesting future work.

6 Conclusion

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

References

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. OSDI*, Dec. 2002.

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

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

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

[5] 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.

[6] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. SOSP*, Nov. 1987.

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

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

[9] 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.

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

[11] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. "PRACTI replication for large-scale systems". Technical Report UTCS-04-28, University of Texas Department of Computer Sciences, June 2004.

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

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

[14] D. Gifford. Weighted voting for replicated data. In *Proc. SOSP*, pages 150–162, Apr. 1979.

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

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

[17] 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.

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

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

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

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

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

[23] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, Nov. 2000.

[24] 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.

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

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

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

[28] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pages 203–213, 1998.

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

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

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

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

[33] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. Technical Report 89-1042, Cornell, Nov. 1989.

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

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

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

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

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

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

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

[41] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering web cache consistency. *ACM Trans. on Internet Tech.*, 2(3), 2002.

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

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

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