PADRE: A Policy Architecture for building Data REplication systems

N. Belaramani, M. Dahlin, A. Nayate, J. Zheng
Department of Computer Sciences
The University of Texas at Austin

Abstract: This paper presents PADRE, a *policy architecture* that qualitatively simplifies the development of data replication systems. PADRE achieves this by embodying the right abstractions for replication in large-scale systems. In particular, PADRE cleanly separates the problem of building a replication system into the subproblems of specifying safety policy and specifying liveness policy and identifies a surprisingly small set of primitives that are easy to use and sufficient to specify sophisticated safety constraints and liveness protocols. As a result, building a replication system is reduced to picking a consistency library and then writing a few rules in a declarative language to set up a communication topology. We demonstrate the *flexibility* and *simplicity* of PADRE by constructing 6 substantial and diverse systems using just a few dozen lines of system-specific policy code each. We demonstrate the *agility* of systems built on PADRE by adding new features to four systems yield significant performance improvements; each addition required fewer than 10 additional lines of code and took less than a day.

1 Introduction

This paper presents PADRE—a *policy architecture* that qualitatively simplifies the development of data replication systems by providing a conceptual framework that divides system design into two aspects: safety policy, embodying consistency and durability requirements, and liveness policy, describing how to route information among nodes.

Although it is common to *analyze* a protocol's safety and liveness properties separately, taking this idea a step further and separately specifying safety and liveness to *implement* replication systems is the foundation of PADRE's effectiveness in simplifying development. Given this clean division, a surprisingly small set of simple policy primitives is sufficient to implement sophisticated replication protocols.

- For safety, the key observation is that consistency and durability invariants can by ensured by blocking requests until they will not violate the invariants. PADRE therefore requires policies to specify predicates that must be satisfied before a read or update completes. For convenience, PADRE provides 5 standard predicates sufficient to specify most consistency semantics including best effort consistency, PRAM consistency, causal consistency, sequential consistency, delta coherence, and linearizability [30].
- For liveness, the key insight is that the policy choices that distinguish replication systems from each other can largely be regarded as routing decisions: Where should a node go to satisfy a read miss? Where should a node send updates it receives? Where should a node send invalidations when it learns of a new version of an object? PADRE therefore provides a *routing policy engine* that executes policy-specific declarative routing rules [15]. PADRE exposes a small number of carefully defined policy primitives, only 23 actions and 12 events, from which a broad range of replication systems can be built.

Even if an architecture is conceptually appealing, to be useful it must help system builders. We believe our experience is compelling: using PADRE a small team was able to quickly construct 6 systems representing diverse and significant research work and then extend 4 of them with significant new features. We do not think this feat would have been possible without PADRE.

More specifically, PADRE exhibits four properties vital for a replication policy architecture: (1) it is flexible enough to support systems that span the bulk of the design space, (2) it is easy to use and significantly reduces development time, (3) it facilitates innovation and evolution by making it straightforward to implement new systems or improve existing ones, and (4) it has an implementation with reasonable performance.

PADRE is flexible. We built 6 systems inspired by systems in the literature including two client-server systems modeled on Coda [10] and TRIP [18], two server replication systems modeled on Bayou [20] and Chain Replication [29], and two object replication systems modeled on Pangaea [22] and TierStore [5].

PADRE greatly simplifies system building. For example, it took 3 graduate students working part time less than 2 months to build our chosen systems. Each system only required between 22 and 164 lines of system-specific policy code.

PADRE facilitates the evolution of existing systems and the development of new ones. For example, we add cooperative caching to P-Coda so that a clique of devices can share data even when disconnected from the server; we add support for small devices to P-Bayou so that a limited-storage device can participate in Bayou replication without storing all of the system's data; we add cooperative caching to P-TierStore so that once one user in a developing region downloads data across an expensive modem link, nearby users can retrieve that data using their local wireless network; and we improved scalability of P-TRIP by changing the update propagation strategy from flat to hierarchical. Each of these features yields significant performance advantages—an order or magnitude in some cases—yet each enhancement requires fewer than 10 lines of additional system-specific policy code.

We find that the systems constructed with PADRE provide reasonable preformance for prototyping and moderately demanding environments. Most overheads can be attributed to inefficiencies in the current declarative language implementation rather than being fundamental to the architecture. However, the network overheads of systems built with PADRE come close to that of ideal hand-crafted implementations letting us conclude that PADRE captures the right abstractions for building replication systems.

2 PADRE Design

Replication systems cover a large design space. Some guarantee strong consistency while others sacrifice consistency for higher availability; some invalidate stale objects, while others push updates; some cache objects on demand, while others replicate all data to all nodes; and so on.

PADRE aims to be a general policy architecture for specifying large-scale replication systems.[1] PADRE divides the problem of specifying a replication system into two smaller problems—specifying safety policy and liveness policy. Surprisingly, given this formulation, a small set of carefully-defined policy primitives are sufficient to construct an extensive range of systems. These primitives are naturally derived from first principles, and as shown in Section 6, the predicates can be implemented with reasonable cost.

For safety, the key observation of PADRE is that a replication system's consistency and durability invariants can be ensured by blocking requests until they may proceed without violating the invariants. Therefore, as Fig. 1 illustrates, PADRE provides a *read/write intercepting layer and a update intercepting layer* that can block a local read, local write, or update received from the network until a policy-supplied predicate is satisfied. For example, for both consistency and durability's sake, a client-server system policy's predicate might specify that a write does not complete until its update is stored at the server.

[1]We informally define a large-scale replication system as one in which nodes can be partitioned from one another, and PADRE focuses on providing a framework for policies that must manage availability v. consistency v. resource consumption v. performance trade-offs in such environments.

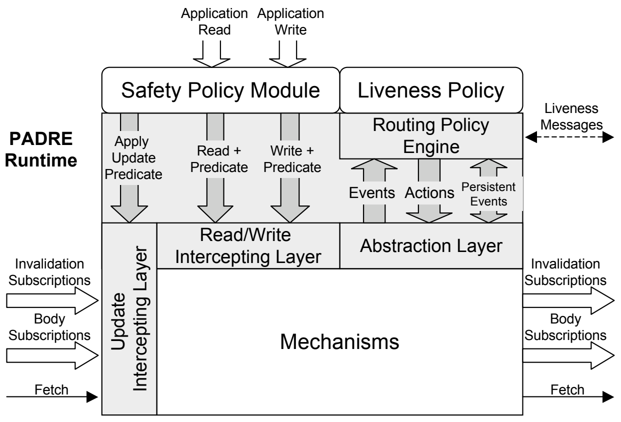

Fig. 1: Overview of PADRE.

Similarly, for liveness, the key insight is that the policy choices that distinguish replication systems can largely be regarded as routing decisions: Where should a node go to satisfy a read miss? Where should a node send updates it receives? Where should a node send invalidations when it learns of a new version of an object? A liveness policy simply supplies a set of rules to ensure propagation of the data and metadata needed for liveness—including not just eventual progress but also good performance and high availability. PADRE therefore provides a *routing policy engine* that executes policy-specific rules that are triggered by a set of 23 PADRE-specified *events*, that invoke one of 12 PADRE-specified *actions*, and that can store and later retrieve *persistent events* that allow declarative routing policy rules to operate on persistent replication system metadata.

As Fig. 1 illustrates, PADRE relies on a set of underlying mechanisms to store objects locally, receive and transmit updates, and maintain bookkeeping state to track the consistency of the local state with respect to updates by other nodes. We precisely define PADRE's requirements on the underlying mechanisms in Sections 2.1 and 2.2 where we define the primitives that PADRE must expose. As discussed in Section 3, our prototype system makes use of PRACTI [1] to provide suitable mechanisms.

2.1 Safety Policy

Every replication system guarantees some level of consistency and durability. Consistency and durability are cast as safety policy because each defines the circumstances under which it is safe to process a request or to return a response. In particular, enforcing consistency semantics generally requires blocking reads until a sufficient set of updates are reflected in the locally accessible state or blocking writes until the resulting updates have made it to some or all of the system's nodes, or both [12, 30]. Similarly, durability policies often require writes to propagate to some subset of nodes (e.g., a central server [8], a quorum [28], or an object's "gold" nodes [22]) before the write is considered complete or before the updates are read by another node.

Intercepting layers. As Fig. 1 indicates, PADRE defines a *read/write intercepting layer* that lies between the application's read/write interface and the interface provided by the underlying mechanisms. Similarly, it defines a *update intercepting layer* that intercepts updates received from remote nodes before they are applied to the underlying local state.

The *intercepting layers* define a total of 5 points for which a policy can supply a predicate and a timeout value that blocks a request until the predicate is satisfied or the timeout is reached. *ReadNowBlock* blocks a read until it will return data from a moment that satisfies the predicate and *WriteBeforeBlock* blocks a write before it accesses the underlying local store. *ReadEndBlock* and *WriteEndBlock* block read and write requests respectively after they have accessed the local store but before they return. *ApplyUpdateBlock* blocks an update received from the network before it is applied to the local store.

Safety policy module. The safety policy of a specific replication system defines the predicates for each of the 5 points to capture the safety conditions that must be satisfied before a request can continue. If no predicate is supplied, then requests go straight through without blocking.

A replication system's safety policy is implemented as a *safety policy module* that is installed between PADRE's intercepting layer and the application read/write interface. The module first installs its desired *ApplyUpdateBlock* predicate, and it then transforms incoming read/write requests by adding predicates before issuing them to the underlying system via PADRE's intercepting layer.

Note that some consistency and durability policies such as AFS's open/close consistency [8] operate on groups or sessions of operations rather than individual reads and writes. In such cases, a replication system's safety policy module buffers writes and/or implements a read cache. Also, some safety policies require a set of updates to be applied atomically, so PADRE's read/write intercepting layer provides a means to issue atomic multi-object writes. To use this facility, a replication system's safety policy module adds a *transaction ID* to writes and issues a commit directive at appropriate times.

Standard predicates and libraries. Specifying consistency semantics be tricky. In order to make things easier for policy designers, PADRE defines a set of built-in predicates, listed in Fig. 2, which are sufficient to specify any order-error or staleness constraint in the TACT model [30]. Yu and Vahdat describe how these constraints can be used to implement a broad range of consistency models [30].

Standard Blocking Predicates	
isValid(o)	Block until object o is valid
isCausal(o)	Block until the object o is causally consistent with previously completed local reads and writes
haveBody(o)	Block the application of an invalidation received for object o until the corresponding body arrives.
hasArrived (Nodes, p)	Block until each node in $Nodes$ has received my pth most recent write
maxStaleness(Nodes, operationStart, t)	Block until I have received all writes up to $(operationStart - t)$ from every node in $Nodes$.

Fig. 2: Standard, built-in predicates available for defining safety policies.

We also provide pre-built consistency libraries for a range of standard consistency semantics: best-effort coherence, causal consistency [13], 1-copy serializability [2], open/close consistency [8], sequential consistency [13], and linearizability [7]. These libraries specify safety predicates and also include the liveness rules for generating any messages required to eventually satisfy the safety conditions. Section 2.2 provides more details about the liveness rules.

Defining customized predicates. System designers may want to define their own predicates if their systems require consistency semantics not covered by the standard libraries or if they want to take advantage of their knowledge of the system's topology to implement consistency more efficiently. For example, our standard library for sequential consistency initiates communication between all nodes. However, to implement the same consistency semantics in a client-server system, it may be more appropriate to restrict communication patterns to only client-server communication [19].

Most commonly, a custom consistency predicate simply waits for a specific message or event generated by the liveness policy. Section 4, for example details how our implementation of Coda can block completion of a client's write until the server confirms that it has completed invalidating all other connected clients. Additionally, designers can define predicates that depend on the information about local state exposed by the underlying mechanisms. For completeness, we enumerate the local information on which predicate can operate on in Appendix A.

2.2 Liveness Policy

In PADRE, a liveness policy must set up data placement and the communication among nodes so that updates propagate and safety conditions can eventually be met.

Update propagation is perhaps the most important aspect of data replication systems. Every system implements protocols that answer the questions: "When and to whom should I propagate an update to?" and "Who should I contact if data is not locally available?" Each

Subscription Actions	
Add pull inval subscription	sourceId, SS [, startTime, CP|LOG]
Add pull body subscription	sourceId, SS [, startTime]
Add push inval subscription	destId, SS [, startTime, CP|LOG]
Add push body subscription	destId, SS [, startTime]
Remove pull inval sub	sourceId, SS
Remove pull inval sub	sourceId, SS
Remove push body sub	destId, SS
Remove push body sub	destId, SS
One-off Actions	
Fetch body	sourceId, objId, off, len [, logicTime]
Push body	destId, objId, off, len
Status Actions	
GetCurrLogicalTime	
GetCurrRealTime	

Fig. 3: Liveness actions provided by PADRE

system answers these questions differently according to what is best for the environment it targets. For example, in a client-server system, updates on a client are always sent to the server but in a peer-to-peer systems, an update on one node may flood the network of nodes.

Further more, a liveness policy must ensure that safety conditions are satisfied and requests eventually unblock. These safety conditions often rely on information about the propagation of an update to other nodes. A liveness policy therefore must also generate and route status information.

PADRE provides a *routing policy engine* and exposes a set of 12 actions and 23 events that can be used by policy rules to set up update and meta-data routing. It also exposes a mechanism to store events persistently and trigger them later. The liveness actions, liveness events and persistent events, listed in Figs. 3, 4 and 5, represent the complete interface necessary for specifying the liveness policies of a broad range of systems. Each primitive is carefully crafted to capture the right abstractions. In the rest of this section, we detail this interface.

2.2.1 Liveness Actions

A policy instructs PADRE to propagate data through *liveness actions*. The liveness actions, listed in Fig. 3, fall under two categories: actions that set up update propagation between two nodes and actions that collect local status information.

Update propagation actions. PADRE provides 6 basic actions, listed in Fig. 3 for the different variations of update propagation:- (pull or push) of a (one-off transfer or subscription) of (bodies or invalidations). PADRE provides four more actions to turn off subscriptions.

Nodes can communicate one of two types of information about an update, either an *invalidation* or a *body*. An invalidation captures the fact that an update occurred at a particular instant in logical time. A body contains the data of the update. The communication between nodes can be a continuous stream of updates that occurred at the sender or a one-off transfer of an update.

PADRE defines a uni-directional continuous stream as a *subscription* and supports both invalidation subscriptions and body subscriptions. Every subscription action is associated with a *subscription set* and a *start-time*. The *subscription set* specifies the group of objects (e.g. /a/b/*:/x/*) for which the receiver is interested in receiving invalidations or bodies of updates that occurred after the logical *start-time*.

An invalidation subscription contains a causally consistent stream of invalidations. The causal order is important to support stronger consistency semantics at a higher level. However, in order to maintain consistency across all objects despite subscription sets that may span arbitrary subsets of objects, invalidations that a node receives on a subscription must be causally ordered with respect to other invalidations. Therefore, invalidation subscriptions must contain causal ordering information of objects outside the subscription set as well. PRACTI, for example, uses *imprecise invalidations* to provide such ordering information [1].

Invalidations subscriptions also expose a third parameter, the *catchup* option. If the *catchup* option is *CP*, the subscription begins with a checkpoint that invalidates all objects in the *subscription set* that were updated between the *start-time* and the current time. If the *catchup* option is *LOG*, the subscriptions begins with the log of invalidations to objects in the *subscription set* from the *start-time*. Note that *CP* and *LOG* catchup options have identical semantics and post conditions, but as we detail in Section 3, their costs can differ.

A body subscription, on the other hand, is an unordered stream of bodies of updates.

PADRE also supports one-off transfer of bodies from one node to the other. The "fetch" or "push", is associated with an *objId* and an optional *logical time*. The sender sends the latest body of *objId* it has. If *logical time* is specified, the body is only sent if it is at least as new as *logical time*, otherwise, a *NACK* is sent otherwise. Note, PADRE does not support one-off transfer of an invalidation because invalidations must always be ordered with respect to other invalidations for consistency.

Local status actions. Some safety predicates may depend on the propagation of information through the system. PADRE, therefore provides actions that enable liveness policy to collect information about a node's local status. These actions include: *GetCurrLogicalTime* to retrieve the current logical time of the node and *GetCurrRealTime* to retrieve the current real time of the node.

2.2.2 Liveness Events

Policy writers need events about local state in order to decide when to invoke what actions. For example, a policy needs to know if a read was blocked so that it can decide to *fetch* a body from another node or to establish

Local read/write events	
Read blocked non-existent object	objId, offset, length, logicTime
Read blocked invalid object	objId, offset, length, logicTime
Read blocked inconsistent object	objId, offset, length, logicTime
Write	objId, offset, length, logicTime
Delete	objId

Message arrival events	
Inval arrives	sender, objId, off, len, logicTime
Fetch success	sender, objId, off, len, logicTime
Fetch failed	senderId, objId, offset, length, logicTime

Connection events	
Inval subscription start	SS, senderId
Inval subscription caught-up	SS, senderId
Inval subscription end	SS, senderId
Inval subscription failed	SS, senderId
Body subscription start	SS, senderId
Body subscription end	SS, senderId
Body subscription failed	SS, senderId
Outgoing inval subscription start	SS, receiverId
Outgoing inval subscription end	SS, receiverId
Outgoing inval subscription failed	SS, receiverId
Outgoing body subscription start	SS, receiverId
Outgoing body subscription end	SS, receiverId
Outgoing body subscription failed	SS, receiverId

Status events	
Current logical time	logicTime
Current real time	realTime

Fig. 4: Local events generated for liveness policy.

subscriptions. Fig. 4 lists the events provided by PADRE for this purpose. These events inform the policy layer about

- *Local read, write, delete events* for objects and whether the request was blocked.
- *Messages received.* Whether an invalidation arrived or fetches were successful.
- *Connection events.* Whether subscriptions were successfully established or whether they were removed.
- *Status events* about the requested information such as the logical time, or real time.

2.2.3 Persistent Events

Some policies need mechanisms to store routing or configuration information persistently. For example, in Pangaea [22], each object also identifies the list of gold replicas for that object. Other systems store configuration lists such as lists of nodes to peer with or lists of files to prefetch [5, 10].

Given that liveness policies in PADRE are described using rules that invoke actions when triggered by events, we convert persistent information into an event abstraction. PADRE provides a set of actions, listed in Fig. 5, that store of events as tuples persistently in an object and later use that object to generate the events.

For example, to append an item to a hoard list [10], a rule can invoke the action *WriteTuple: hoardListObjId, HOARD_ITEM, objIdToBeHoarded.* Later when the system wishes to walk the hoard list, a rule invokes the action *RadTuples: hoardListObjId* which causes the runtime system to read the specified object and generate a *HOARD_ITEM* event for each tuple stored in the object.

Persistent Events	
Write tuple	objId, tupleName, field1, ..., fieldN
Read tuples	objId
Read and watch tuples	objId
Stop watch	objId
Delete tuples	objId

Fig. 5: Interface to store events persistently

2.3 Excluded Properties

There are several properties that PADRE does not address or for which provides limited choice to designers. These include security, interface, and conflict resolution.

First, PADRE does not support security specification. We believe that ultimately our policy architecture should also define flexible security primitives. Providing this capability is important future work, but it is outside the scope of this paper, which can be regarded as focusing on the architectural problem of allowing systems to define their replication policy in terms of consistency, durability and topology.

Second, PADRE only exposes an object-store interface for local reads and writes. It does not expose other interfaces such as a file system interface or a tuple store. We believe that these interfaces are not difficult to incorporate. In fact, we have implemented a file system interface over our prototype of PADRE.

Third, PADRE only assumes a simple conflict resolution mechanism. Write-write conflicts are detected and logged in a way that is data-preserving and consistent across nodes to support a broad range application-level resolvers. We do not attempt to support all possible conflict resolution algorithms [5, 9, 10, 24, 27]. We believe it is straight forward to extend the PADRE model to support Bayou's more flexible application-specified conflict detection and reconciliation programs [20].

3 Implementation

We developed a prototype in order to explore the feasibility and practicality of building systems with PADRE. The prototype implementation takes advantage of PRACTI [1] because the mechanisms PRACTI provides match those needed by PADRE, and it uses OverLog/P2 [15] because OverLog/P2 provides a language and a runtime with which liveness rules can be conveniently written and executed.

PRACTI. PRACTI is a data replication substrate, implemented with Java and Berkeley DB. PRACTI provides all mechanisms required by PADRE including an object store, consistency tracking, invalidation and body subscriptions, and fetches. The semantics of these mechanisms match PADRE's expectations.

Primitive	Ideal Implementation	PRACTI Implementation
Inval subscription with LOG catchup	*(pastUpdates + newUpdates) * sizeInval*	*(pastUpdates + newUpdates) * sizeInval + numImpInv * sizeImpInv*
Inval subscription with CP catchup	*(pastUpdatedObjs + newUpdates) * sizeInval*	*pastUpdatedObjs * sizeMeta + newUpdates * sizeInval + numImpInv * sizeImpInv*
Body subscription	*(pastUpdatedObjs + newUpdates) * sizeBody*	*(pastUpdateObjs + newUpdates) * sizeBody*
Fetch/Push Body	*sizeBody*	*sizeBody*

Fig. 6: Ideal network overhead and PRACTI implementation of PADRE primitives where *sizeInval* is the size of an invalidation, *sizeImpInv* is the size of an imprecise invalidation, *sizeMeta* is size of meta data of object for checkpoint, *sizeBody* is average size of a body, *pastUpdates* is number of past updates – updates to the subscription set that occurred after start-time but before the subscription was established, *newUpdates* is number of current updates – updates to the subscription set after the subscription was established, *pastUpdateObjs* is number of objects in the subscription set modified by past updates, *numImpInv* is the number of imprecise invalidations sent on a invalidation subscription.

PADRE defines 4 modes for update propagation. Fig. 6 list the network overheads for an ideal implementation and PRACTI implementation of these primitives. The overheads for the ideal implementation are derived directly from the description of the primitive in Section 2.2. For example, when a body subscription is established, all bodies of objects in the subscription set that were updated from *start-time* to the current time are sent, after which the sender forwards any new body it receives. Hence the idealized network cost of a body subscription is: *(pastUpdatedObjs + newUpdates) * sizeBody*.

As Fig. 6 illustrates, the network overheads of PRACTI are within a constant factor of an ideal implementation of PADRE. Invalidation subscriptions in PRACTI also transfer *imprecise invalidations* that aggregate invalidations of updates outside the subscription set for consistency reasons. However, the overhead is at most 50% of the ideal implementation. The number and size of imprecise invalidations is greatly dependent on the locality of the workload. In the worst case scenario, for workloads with poor locality, at most one imprecise invalidation is sent per actual invalidation. The size of the imprecise invalidations is dependent on how concisely the list of affected objects can be encoded. Thus, for workloads with good locality, few imprecise invalidations are sent, and and the size of an imprecise invalidation is comparable to the size of a precise invalidation [1].

Another thing to note is that the log of invalidations that PRACTI maintains is truncated from time to time to maintain storage bounds [20]. If an invalidation subscription is established with LOG catch up but its *start-time* is set to a point before the log truncation point, a checkpoint up to the log truncation point, followed by invalidations are sent instead.

We made several changes to PRACTI during the implementation of PADRE. These changes include adding an interface that converts PRACTI commands and inform methods to PADRE's actions and events; making the underlying implementation of PRACTI subscriptions more efficient by multiplexing multiple subscriptions on a single stream; adding a file system interface support by implementing an NFS interface in Java; and adding support for atomic multi-object writes needed for the file system interface implementation. Details of these changes require a knowledge of PRACTI's mechanisms, which is outside the scope of this paper.

OverLog/P2. OverLog and P2 [15] are respectively a declarative language and a runtime layer for implementing overlays. Since PADRE considers liveness policy as specifying topology, OverLog/P2 are good for writing and executing liveness policy.

A program in OverLog is a set of table declarations for storing tuples and a set of rules that specify how to create a new tuple when a set of existing tuples meet some constraint. For example,

out(@Y, A, C, D) :-
in1(@X, A, B, C), in2(@X, A, B, D), in3(@X, A, _)

indicates that whenever there exist at node *X* tuples *in1*, *in2*, and *in3* such that all have identical second fields (*A*), and *in1* and *in2* have identical third fields (*B*), create a new tuple (*out*) at node *Y* using the second and fourth fields from *in1* (*A* and *C*) and the fourth field from *in2* (*D*). Note that for *in3*, the _ wildcard matches anything for field three.

PADRE actions and events map naturally to OverLog's tuples. OverLog rules can be used to specify liveness rules that trigger actions based on events received.

To give a concrete example of how one could specify an action in the liveness policy, consider the following rule from our implementation of TierStore [5]:

addPullInvalSubscription(@Node, Parent, Volume) :-
newLiveNeighbor(@Node, Parent),
isParent(@Node, Parent),
wantSubscribe(@Node, Volume)

This rule causes a node to subscribe for one or more volumes of interest when a node it regards as its parent becomes reachable.

PADRE prototype. We added support to allow the P2 runtime to communicate with PRACTI so that liveness the rules can invoke the appropriate mechanisms. In particular, we implemented an abstraction layer that converts the large collection of low-level PRACTI commands and inform methods into a smaller set of higher level PADRE actions and events. We also added a bridge between the abstraction layer and the P2 routing engine.

4 Building a System on PADRE

This section can be seen as a quick tutorial on how systems can be constructed on PADRE. To build a system, a designer specifies safety and liveness policies by typically picking a pre-built standard safety library and writing OverLog rules to set up the topology. Alternatively the designer can build a customized safety library by specifying predicates at the safety module in Java, and writing rules so that safety conditions can eventually be met.

We picked Coda [10] as our example system because it is a well-studied system that has a lot of demanding features and can demonstrate the ease with which each feature can be built on PADRE. Our implementation, P-Coda, is inspired by the version of Coda described by Kistler et. al. [10]. P-Coda supports disconnected operation, reintegration, crash recovery, whole-file caching, open/close consistency (when connected), causal consistency (when disconnected), and hoarding. We also illustrate the ease with which significant new features can be added to an existing system by adding co-operative caching to P-Coda.

We know of one feature from the system Kistler et. al. describes that we are missing: we do not support cache replacement prioritization. In Coda, some files and directories can be given a lower priority and will be discarded from the cache before others. Additionally, Coda is a long-running project with many papers worth of ideas. We omit features discussed in other papers such as server replication [23], trickle reintegration [16], and variable granularity cache coherence [17]. We see no fundamental barriers to adding them in P-Coda.

We provide a brief system description and then present the client-side safety and liveness policies respectively.

4.1 System Description

P-Coda is a client-server system. The server stores all files and each client caches some files. The server maintains a list of clients who cache valid copies of each file.

P-Coda provides open/close semantics, which means that when a file is opened at a client, the client will return the local valid copy or retrieve the newest version from the server. A file close operation on a connected client will block until all updates have been propagated to the server and the server has made sure that all copies cached on other connected clients have been invalidated. When a client is disconnected from the server, open/close consistency is relaxed and a client can access locally cached files that are causally consistent.

Every client has a list of files, the "hoard set", that it will periodically refetch from the server and store locally.

4.2 Implementing safety policy

P-Coda uses the standard open/close consistency library. A file "open" is implemented as a read of an object. The *ReadNowBlock* predicate is set to *isCausal*; i.e., all reads will block until the local object is consistent before the object is accessed. All writes to a file are buffered until the file is closed, at which point the object is written. If a client is connected to the server, the *WriteEndBlock* predicate is *recvAckFromServer*; otherwise it is the null predicate. The open/close safety module has 124 semicolons of Java code.

4.3 Implementing liveness policy

P-Coda's 33 client-side liveness rules can be divided into 7 main groups: configuration, connectivity, demand read, write propagation, recovery, hoarding, and safety metadata. Algorithm 1 defines P-Coda's client side liveness policy.

Configuration and connectivity. A configuration file stores the server's identity in a *configServer* tuple, and another configuration file provides the hoard list in a series of *doHoard* tuples. At each client C, the table entry *isConnected(@C, S)* indicates whether the server S is currently reachable. We use 17 rules (not shown) based on the published P2 implementation of Narada [15] to track connectivity information and generate *newLiveNeighbor* and *declareDeadNode* tuples, which invoke rules (c1) or (c2) respectively.

Handling blocked reads. Two rules are triggered when a read of object *o* is blocked at a connected client. (sc1) subscribes for *o*'s invalidations using a checkpoint for efficiency, and (sc2) fetches the body. Eventually, *o* is no longer inconsistent, and the safety policy unblocks the read. The invalidation subscription ensures that if another node updates *o*, it will become invalid.

Write propagation and callbacks. To propagate client writes to the server, rules (cs1) and (cs2) are triggered when the client connects to the server. They create an invalidation subscription and a body subscription for all updates from the client to the server. Note that because of open/close semantics, the writes within one open/close session only generate one invalidate and body that go through the subscriptions to the server.

The invalidation subscriptions created by clients when they cache objects from the server ensure that our underlying mechanisms transmit invalidations of new updates to maintain causal consistency. To avoid sending repeated invalidations to a client, we include a rule (cbr) to remove an object from the invalidation subscription when a client's cache is already invalidated by an invalidation.

Recovery. When a client reconnects to a server, it triggers (re) to establish an invalidation subscription for an

```
// Get server and list of peers from config file
  cf1    readAndWatchTuple(@X, nodeFile) :-
             atInit, nodeFile:= "/coda/nodeList"
  cf2    server(@X, S) :-
             configServer(@X, S)
// Server connection status
  c1   isConnected(@X, V) :-
           newLiveNeighbor(@X, S), server(@X, S), V:=1
  c2   isConnected(@X, V) :-
           declareDeadNode(@X, S), server(@X, S), V:=0
// Local read miss: Add an inval subscription
  sc1    addPullInvalSubscription(@X, S,Obj, Catchup) :-
             localReadBlocked(@X, Obj, _, _), server(@X, S),
             isConnected(@X, V), V==1, Catchup:="CP",
             X≠S //I am client
// ... and get a body
  sc2    fetch(@X, S, Obj, Off, Len) :-
             localReadBlocked(@X, Obj, Off, Len), server(@X, S),
             isConnected(@X, V), V==1, X≠S //I am client
// Server is detected: add subscriptions to send updates to server
  cs1    addPullInvalSubscription(@S, X, SS, Catchup) :-
             isConnected(@X, V), V==1, server(@X, S), SS:=/*,
             Catchup:="LOG", X≠S // I am client
  cs2    addPullBodySubscription(@S, X, SS, Catchup) :-
             isConnected(@X, V), V==1, server(@X, S), SS:=/*,
             Catchup:="LOG", X≠S // I am client
// Client receives an inval: Remove subscription
  cbr    removePullInvalSubscription(@S, X, Obj) :-
             invalArrives(@X, S, Obj, _, _, _), server(@X, S),
             X≠S // I am client
// Server is detected: Add subscription to ""
  re    addPullInvalSubscription(@X, S, SS, Catchup) :-
           isConnected@X(X, V), V==1, server@X(X, S),
           SS:=EMPTY, Catchup := "LOG", X≠S // I am client
// Hoarding: Add hoard subscriptions when server reachable
  h1    readAndWatchTuple(@X, hoardFile) :-
           isConnected(@X, V), V==1, hoardFile:="/coda/hoardList"
  h2    addPullInvalSubscription(@X, S, SS, Catchup) :-
           doHoard(@X, SS), Catchup:="CP", server(@X, S),
           X≠S // I am client
  h3    addPullBodySubscription(@X, S, SS, Catchup) :-
           doHoard(@X, SS), Catchup:="CP", server(@X, S),
           X≠S // I am client
// Safety metadata: Send Ack to server when I receive an inval
  sf1    ackServer(@S, X) :-
             invalArrives(@X, S, Obj, _, _, _), server(@X, S),
             X≠S // I am client
// Safety metadata: Inform safety policy when received ack from server
  sf2    recvAckFromServer(@X) :-
             ackFromServer(@X, S), server(@X, S), X≠S // I am client
// Safety metadata: Inform safety policy if not connected to server
  sf3    notConnected(@X) :-
             writeBlocked(@X), isConnected(@X, V), V≠1,
             X≠S // I am client
// Cooperative caching: Check reachable peers if server unreachable
  cc1    peer(@X, P) :-
             configPeer(@X, P)
  cc2    pConnected(@X, P, V) :-
             newLiveNeighbor(@X, P), peer(@X, P), V:=1
  cc3    pConnected(@X, P, V) :-
             declareDeadNode(@X, P), peer(@X, P), V:=0
  cc4    fetch(@X, P, Obj, Off, Len) :-
             localReadBlockedInconsistent(@X, Obj, Off, Len),
             isConnected(@X, V), V==0, pConnected(@X, P, W),
             W==1, server(@X, S), X != S // I am client
```

Algorithm 1: Client-side liveness rules in P-Coda. Connection monitoring rules are excluded

empty subscription set from the server. This action makes all locally cached objects inconsistent until new callbacks in the form of invalidation subscriptions are established.

Hoarding. As in Coda, we prefetch objects in a user-defined *hoard set*. The hoard set is stored as tuples in a local configuration file which is read when the server becomes connected (h1). The client then subscribes to receive invalidations and bodies for subscription sets listed in the hoard file (h2, h3).

Safety metadata. Several events propagate safety metadata to ensure that the open/close consistency library eventually allows progress. A client sends an acknowledgement to the server whenever it receives an invalidation (sf1). Once the server has collected the required acknowledgements from other clients, it sends *ackFromServer(@X,S)* to the client. The liveness policy generates *recvAckFromServer(@X)* so that the safety policy can unblock. If the client is disconnected from the server, it simply generates a *notConnected(@X)* message so that the write can unblock (sf3). These functions require 22 rules on the server.

4.4 Adding Cooperative Caching

We add cooperative caching to P-Coda so that disconnected clients can fetch valid files from their peers. This feature allows a disconnected client to access files it previously couldn't.

Four rules enable disconnected clients to fetch data from their peers while maintaining causal consistency. Algorithm 1 shows how we augment the node list configuration file to include a list of peers. When (cf1) reads the configuration file, it generates *configPeer* tuples that populate the peer table via (cc1). (cc2) and (cc3) keep track of connectivity to peers, and (cc4) triggers a fetch attempt from reachable peers if the server is not reachable. Note that the arrival of data unblocks a waiting read regardless of where it comes from and the predicates ensure that consistency is still enforced despite this significant change to how updates flow through the system. In section 6, we show the performance improvement achieved by these four rules.

5 Case Studies

This section examines a series of case-studies to explore whether PADRE's approach to constructing replication system does indeed benefit system designers.

We demonstrate the flexibility of PADRE by constructing 6 systems covering a large part of the design space including client-server systems like Coda [10] and TRIP [18], server-replication systems like Bayou [20]and Chain Replication [29], and object replication systems like Pangaea [22] and Tier-Store [5]. Figure 7 provides an overview of the scope of this effort.

	Bayou	Chain Replication	Coda	Pangaea	TierStore	TRIP
Consistency	Causal	Linearizability	Open/close		Causal*	Sequential
Coherence				√	√	
Structured Topology		√	√		√	√
Ad-hoc Topology	√			√		
Callbacks			√			
Co-operative Cashing			√*		√*	
Hoarding			√			
Anti-Entropy	√					
Flooding				√		
Never invalidate object before body available	√				√	√

Fig. 7: Features covered by case-study systems. Features with * are additional features added to our implementation.

	Safety Policy		Liveness Policy
	Standard	Customized	
Bayou	12	-	29
Chain Replication	-	85	79
Coda	124	-	55
Pangaea	12	-	67
TierStore	12	-	28
TRIP	12	-	22

Fig. 8: Lines of code required to implement each system. Standard safety policy means that a standard consistency library was used. Customized means that a customized library was used to implement the system.

```
// Define subscription set for anit-entropy
  init    ss(@X, SS) :-
              atInit, SS:="/*"
// Add subscriptions when a random neighbor is selected
  bc01    addPullInvalSubcription(@X, Y, SS, Catchup) :-
              doAntiEntropy(@X, Y), ss(@X, SS), Catchup:="LOG"
  bc02    addPullBodySubcription(@X, Y, SS) :-
              doAntiEntropy(@X, Y), ss(@X, SS)
// Remove subscriptions when we have received all updates
  bc03    removePullInvalSubcription(@X, Y, SS) :-
              informInvalSubscriptionCaughtup(@X, Y, SS)
  bc04    removePullBodySubcription(@X, Y, SS) :-
              informBodySubscriptionCaughtup(@X, Y, SS)
```

Algorithm 2: Anti-entropy rules in PADRE-Bayou

Constructing these systems also demonstrates the simplicity of using PADRE. Each system required a few dozen lines of policy code, see Fig. 8. All systems were built by 3 graduate students in less than 2 months of part-time effort. In particular, most of them took 1-2 weeks to build.

Finally, we demonstrate PADRE's support for rapid evolution by extending systems by adding new features. We add cooperative caching to P-Coda in four lines; this addition allows a set of disconnected devices to share updates while retaining consistency. We add small-device support to P-Bayou in one line; this addition allows devices with limited capacity or that do not care about some of the data to participate in a server replication system. We add cooperative caching to P-TierStore in four lines; this addition allows data to be downloaded across an expensive modem link once and then shared via a cheap wireless network. Each of these simple optimizations provides significant performance improvements or needed capabilities as illustrated in section 6.

We describe each system we developed in turn. Due to space constraints we only highlight how essential features of their designs were implemented on PADRE. We do not repeat the discussion of P-Coda.

5.1 P-Bayou

We implement a server replication system over PADRE modeled on the version of Bayou described by Petersen et. al. [20]. In particular, we implement the log-based peer-to-peer anti-entropy protocol, checkpoint exchange in case of log truncation, and causal consistency. We then add small-device support to P-Bayou by simply changing one rule.

Implementing safety and liveness policies. For safety, P-Bayou simply uses the standard causal consistency library provided by PADRE. To provide 100% availability as the original Bayou does, P-Bayou uses the standard *haveBody* predicate to delay applying an invalidation to a node until the node has received the corresponding body.

The liveness policy contains rules for carrying out anti-entropy sessions. Anti-entropy sessions can be easily implemented by establishing invalidation and body subscriptions between nodes for "/*" and removing the subscriptions once all updates have been transferred. Note, as in Bayou, if the log at the sender is truncated to a point after subscription's start time, the invalidation subscription will automatically send a checkpoint.

The complete implementation of P-Bayou's liveness policy requires 29 rules: 5 for anti-entropy (Algorithm 2), 8 for random neighbor selection (not shown), and 16 for connection management (not shown). The *doAntiEntropy* tuple is generated periodically by the random neighbor selection rules which in turn invokes the anti-entropy rules (bc01, bc02).

Small-device support. In standard Bayou, each node must store all objects in all volumes it exports and must receive all updates in these volumes. It is difficult for a small device (e.g, a phone) to share some objects with a large device (e.g, a server hosting my home directory). By building on PADRE, we can easily support small devices. Instead of storing the whole database, a node can specify the set of objects or directories it cares about by changing the subscription set for anti-entropy. This addition requires changing only the (init) rule. This change might introduce some read misses due to incomplete invalidate information propagations from "small" devices to "large" devices. We add two standard predicates *haveBody* and *isConsistent* to *ApplyUpdateBlock* to prevent this situation.

5.2 P-Pangaea

Pangaea [22] is a wide-area file system that supports high degrees of replication and high availability. Replicas of a file are arranged in an *m*-connected graph, with a clique of *g* gold nodes. The location of the gold nodes for each file is stored in the file's directory entry. Updates flood harbingers in the graph. On receipt of a harbinger, a node requests the body from the sender of the harbinger with the fastest link. Pangaea enforces weak, best-effort coherence.

P-Pangaea implements object creation, replica creation, update propagation, gold nodes and m-connected graph maintenance, temporary failure rerouting and permanent failure recovery. We do not implement the "red button" feature, which provides applications confirmation of update delivery or a list of unavailable replicas, but do not see any difficulty in integrating it.

Implementing safety and liveness policies. PADRE-Pangaea uses the standard best-effort coherence library which block reads until objects are valid.

PADRE-Pangaea considers harbingers as invalidations, and hence each edge of a Pangaea graph is an invalidation subscription. PADRE-Pangaea's liveness policy sets up and maintains the m-connected graph for each object among the nodes. If a read is blocked because an object is not locally available, the liveness policy has rules to add the node to the object's graph by finding a nearby replica, fetching the object from the replica, and adding an invalidation subscription from it. When a node receives an invalidation for an object, the object is immediately fetched from the replica with the fastest link.

The liveness policy comprises of 67 rules. Most of the complexity stems from (1) constructing the required per-object invalidation graph across gold and bronze replicas, (2) updating the invalidation graph when nodes become unreachable, and (3) creating new gold replicas for objects when an existing gold replica fails.

5.3 P-Chain Replication

Chain Replication [29] is a server replication protocol in which the nodes are arranged as a chain to provide high availability and linearizability. All updates are introduced at the head of the chain and queries are handled by the tail. An update does not complete until all live nodes in the chain have received it.

P-Chain Replication implements this protocol with support for volumes, node failure and recovery, and the addition of new nodes to the chain.

Implementing safety and liveness policies Although we could use our standard sequential consistency library, to gain performance and availability comparable to the original Chain Replication system we implement a customized consistency library that exploits the chain topology and simply blocks a write until it receives an acknowledgement from the tail. This customized safety library required 85 semi-colons of Java code and 3 OverLog rules for liveness to generate the necessary events.

P-Chain-Replication implements each link in the chain as a invalidation and a body subscription. When an update occurs at the head, the update flows down the chain via subscriptions. Chain management is carried out by a master, as in the original system. We implement the master in OverLog.

Note that most of the complexity in the original chain replication algorithm stems from the need to track which updates have been received by a node's successors so as to handle node failure and recovery. PADRE makes recovery simple because of the semantics guaranteed by subscriptions. When subscriptions are established, all updates that the successor is missing are automatically sent during catchup, making it unnecessary for predecessors to track the flow of updates.

The liveness policy totals 79 rules: 3 for update propagation, 9 for chain management at the servers, 35 for chain management at the master, 20 for connection management, 9 for initialization, and 3 for safety metadata.

5.4 P-TierStore and P-TRIP

We also implement TierStore [5], a hierarchical replication system for developing regions, and TRIP [18], a system that seeks to provide transparent replication of dynamic content for web edge servers. We summarize the relevant statistics in Fig.8. Due to space constraints, we omit detailed discussion.

What is perhaps most interesting about these examples is the extent to which PADRE facilitates evolution. For example, the TRIP implementation assumes a single server and a star topology. By implementing on PADRE, we can improve scalability by changing the topology from a star to a static tree simply by changing a node's configuration file to list a different node as its parent—invalidations and bodies flow as intended and sequential consistency is still maintained. Better still, if one writes a topology policy that dynamically reconfigures a tree when nodes become available or unavailable [15], a few additional rules to subscribe/unsubscribe produce a dynamic-tree version of TRIP that still enforces sequential consistency.

6 Evaluation

Section 5 demonstrated the benefits of PADRE's approach. In this section, we evaluate the practicality of PADRE by examining the performance of the prototype.

First, we show that PADRE faithfully captures the abstractions needed by replication systems in that systems specified at a high level still send the expected messages. Then, we show that latency and throughput of systems built on PADRE are acceptable for prototyping or for moderately demanding system deployments. Finally, we demonstrate the benefits of agility by quantifying the performance improvements achieved by the simple enhancements we made to the case-study systems.

We find that the performance is acceptable for prototyping and for moderately demanding deployments. Most of the overheads can be attributed to inefficiencies in the declarative language interpretation rather than being fundamental to the architecture, as demonstrated in Section 6.2. We believe that the performance should improve as P2 becomes a mature technology. Alternatively, falling back to an imperative implementation of critical rules can yield significant performance improvement.

All experiments are carried out on Dell Dimension 4100 machines with 800MHz Pentium-III CPUs, 256MB of memory, and 100Mb/s Ethernet. We use Fedora Core 6, Sun JVM 1.5, and Berkeley DB Java Edition 3.2.23.

6.1 Network Overheads

Section 3 discusses the cost model PADRE exposes to policy writers. This section shows experimentally that the constants abstracted by the model are modest even compared to the ideal implementation of the primitives.

As discussed in Section 3 an ideal implementation of an invalidation subscription will send *subscriptionStart* message when it is established, and a *subscriptionCaughtUp* message once the past invalidations or the checkpoint has been sent. Each of these messages can be as small as a single byte. In PADRE, since multiple subscriptions are multiplexed on a single stream, the *subscriptionStart* and *subscriptionCaughtUp* messages contain the encoding of the associated subscription set.

Fig. 9 quantifies the network bandwidth required to establish an invalidation subscription. 500 objects were updated and the x-axis corresponds to the number of objects for which subscriptions were established. The three lines correspond to the cost when a separate subscription for each object was established, like traditional callbacks [8].

The figure demonstrates that the cost of establishing subscriptions for *LOG* catchup and *CP* catchup is within a factor of the ideal implementation. The overhead can be attributable to the size of the *subscriptionStart* message. *CP* catchup does worse than *LOG* catchup because the size of the invalidation meta-data for each object is bigger than an actual invalidation sent during *LOG* catchup.

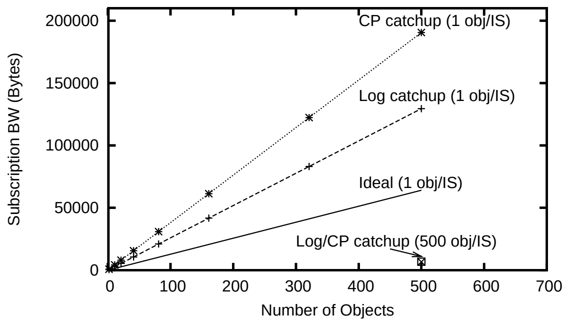

Fig. 9: Bandwidth for establishing invalidation subscriptions.

	Coherence-only	PADRE
1-in-1 update	26	26
1-in-10 updates	26	30
1-in-2 updates	26	52

Fig. 10: Number of bytes per relevant update sent over an invalidation stream for different workloads. 1-in-10 represents a workload in which every 1 out of 10 updates happen to objects in the subscription set.

We also quantify the cost of establishing a single coarse-grained submission for all objects. The cost of a coarse-grained *LOG* and *CP* catchup is almost the same. Both fairly better than the ideal because the *subscriptionStart* and *subscriptionCaughtUp* messages are only sent once instead of 500 times.

Invalidation subscriptions also have the additional overhead of *imprecise invalidations* sent to maintain consistency information. Fig. 10 quantifies this overhead when compared to a system that does not send any consistency ordering information. We compared the overheads under three workloads. As we can see from Fig. 10, even in the worst case, the overhead for maintaining consistency is at most 2x the number of invalidations sent over the subscription.

6.2 System Performance

This section evaluates the performance of systems built on PADRE prototype. We find that the performance is acceptable for prototyping and for moderately demanding deployments, and is greatly affected by the performance of the P2 runtime.

Fig. 11 depicts the read/write throughput at the interception layer for local objects with no predicates. Fig. 12 depicts the time required to run the Andrew benchmark [8] over PADRE prototype via the implemented NFS wrapper. As you can see from Fig. 12, PADRE suffers from a factor of two slowdown when compared to

	Throughput (MB/s)
Read	4.06
Write	2.67

Fig. 11: Local interface performance.

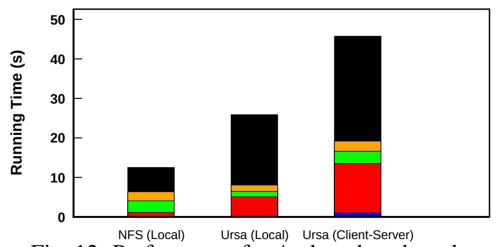

Fig. 12: Performance for Andrew benchmark.

	Latency	Maximum Throughput
Local Ping	4ms	224 req/s
Remote Ping	13ms	95 req/s

Fig. 13: Performance numbers for inserting null tuples into P2.

	P-Coda with liveness in P2	P-Coda with liveness in Java
File open on cold cache	96.92 ms	10.44 ms
File open on hot cache	2.59 ms	0.79 ms
File close	112.1 ms	19.24 ms

Fig. 14: Coda open/close performance with server at 10 ms ping latency.

the benchmark accessing the local Linux ext2 filesystem via a local NFS server. We configured the prototype to emulate a client-server system, and ran the benchmark on the client (with a cold cache) and the server at a ping latency of 10 ms. The performance dropped by another factor of two.

The prototype performance is greatly affected by the performance of P2. Fig. 13 demonstrates the round trip ping latencies and maximum throughput per second for a null P2 event inserted by PADRE on a local machine and remote machine. Infact, the performance of P-Coda greatly improves, as depicted by Fig. 14, when we switched from a P2/OverLog implementation of the liveness rules to a Java implementation of liveness policy. "File close" demonstrates significant difference in performance because its completion is blocked by a liveness predicate, which in turn is affected by P2's performance.

6.3 Benefits of Agility

This section illustrates some of the performance improvements resulting from the the additional features added to our case-study systems.

Fig. 15 demonstrates the significant improvement by adding the 4 rules for cooperative caching to P-Coda. Cooperative caching alllows a clique of connected devices to share data without relying on the server. For the experiment, the results of which are depicted in Fig. 15, the latency between two clients is 10ms, whereas the latency between a client and server is 500ms. Without cooperative caching, a client is restricted to retrieving data from

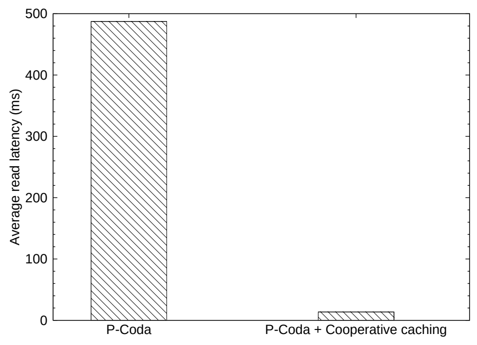

Fig. 15: Average read latency of P-Coda and P-Coda with cooperative caching.

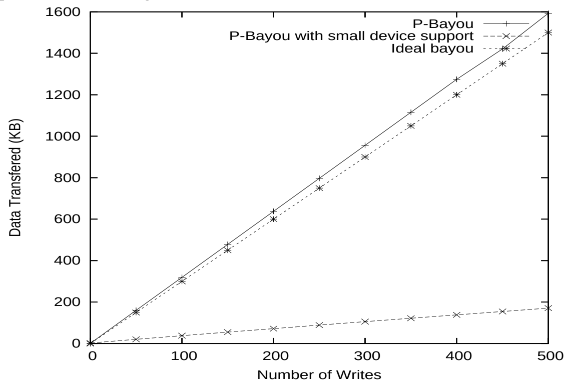

Fig. 16: Anti-Entropy bandwidth on PADRE-Bayou

the server. However, with cooperative caching, the client can retrieve data from a nearby client, thus greatly improving read performance. More importantly, with this new capability, clients can share data even when disconnected from the server. P-TierStore with cooperative caching also demonstrates similar improvements.

Fig. 16 serves two purposes. First, it demonstrates that the overhead for anti-entropy in P-Bayou is relatively small compared to an "ideal" Bayou implementation. More importantly, it demonstrates, if a node requires only 10% of the data, the small device enhancement in P-Bayou greatly reduces the bandwidth required for anti-entropy.

7 Related work

PRACTI [1] defines a set of *mechanisms* designed to span the design space of replication systems, and the PRACTI paper argues that the mechanisms can reduce replication costs by simultaneously supporting Partial Replication, Arbitrary Consistency, and Topology Independence. However PRACTI provides no guidance on how to specify policies that define a replication system. Although we had conjectured that it would be easy to construct a broad range of systems over the PRACTI mechanisms, when we then sat down to use PRACTI to implement a collection of representative systems, we realized that policy specification was a non-trivial task and that a policy architecture was needed to complement PRACTI's mechanisms. PADRE transforms PRACTI's "black box" for policies into an architecture and runtime system that cleanly separates safety and liveness concerns, that provides blocking predicates for specifying consistency and durability constraints, that defines a concise set of events and actions upon which liveness rules operate, and that introduces a persistence model for allowing declarative rules to operate on persistent replication metadata. This paper demonstrates how this approach facilitates construction of a wide range of systems that approximate or improve upon systems from literature.

A number of other efforts have defined general frameworks for constructing replication systems for different environments. Deceit [25] focuses on replication across a well-connected cluster of servers. Zhang et. al. [31] define an object storage system with flexible consistency and replication policies in a cluster environment. As opposed to these efforts for cluster file systems, PADRE focuses on systems in which nodes can be partitioned from one another, which changes the set of mechanisms and policies it must support. Stackable file systems [6] seek to provide a way to add features and compose file systems, but it focuses on adding features to local file systems.

PADRE incorporates the order error and staleness abstractions of TACT tunable consistency [30]; we do not currently support numeric error. Like PADRE, Swarm [26] provides a set of mechanisms that seek to make it easy to implement a range of TACT guarantees; Swarm, however, implements its coherence algorithm independently for each file, so it does not attempt to enforce cross-object consistency guarantees like causal [13], sequential [14], 1SR [2], or linearizability [7]. IceCube [9] and actions/constraints [24] provide frameworks for specifying general consistency constraints and scheduling reconciliation to minimize conflicts. Fluid replication [4] provides a menu of consistency policies, but it is restricted to hierarchical caching.

PADRE uses P2 [15] to execute liveness policies. More broadly, PADRE follows in the footsteps of efforts to define runtime systems or domain-specific languages to ease the construction of routing [15], overlay [21], cache consistency protocols [3], and routers [11].

The specific optimizations we add to replication systems have all been done before. Our contribution is to provide an abstraction that supports such optimizations in a general way and that makes it simple to evolve an existing system by adding new features.

8 Experience and Conclusion

We started this project with the goal of trying to build an architecture that would allow a couple of graduate students to rapidly build replication systems. We studied a dozen or so classic and cutting edge replication systems and we came to three realizations that influenced the design of PADRE:

First, the key design aspect that distinguishes different replication systems is how they define update routing. Every system defines a different routing policy.

Second, if the right mechanisms for update propagation are provided, it is much easier to write routing policy in a declarative language as rules. This realization enabled us to take advantage of PRACTI as the mechanism layer and OverLog as the language for writing routing rules.

There were still some aspects of replication systems that didn't quite fit into routing. Our third realization was that "the rest" was essentially consistency and durability constraints that define "safe-to-return" conditions for requests.

In order to confirm the effectiveness of PADRE, we decided to construct several existing systems over PADRE. Initial difficulties stemmed from trying to understand the protocol of the system we wanted to build and mapping it to the PADRE world and switching to a declarative way of thinking to specify liveness polices. Once we overcame these difficulties, systems became progressively easier to build.

In this paper, we describe PADRE a policy architecture which allows replication systems to be implemented by simply specifying policies. In particular, we show that replication policies can be cleanly separated into *safety* policies and *liveness* policies both of which can be implemented with a small number of primitives

Our experience building 6 systems on PADRE confirmed the benefits of PADRE. PADRE is flexible and can be used to construct a broad range of systems. All 6 systems were built using a few lines of code and in a short amount of time. We don't think this feat would have been possible without PADRE. PADRE also makes it easy to extend a system. We added significant features to 4 systems in less than a day. Finally, despite not being aggressively tuned, the performance of systems built on PADRE is acceptable for prototyping and deploying in moderately demanding environments.

References

[1] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc NSDI*, May 2006.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Replicated Database Systems*. Addison-Wesley, 1987.

[3] 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.

[4] L. Cox and B. Noble. Fast reconciliations in fluid replication. In *ICDCS*, 2001.

[5] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed storage system for challenged networks. `http://tier.cs.berkeley.edu/docs/projects/tierstore.pdf`, Dec. 2006.

[6] J. Heidemann and G. Popek. File-system Development with Stackable Layers. *ACM TOCS*, 12(1):58–89, Feb. 1994.

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

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

[9] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube aproach to the reconciliation of divergent replicas. In *PODC*, 2001.

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

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM TOCS*, 18(3):263–297, Aug. 2000.

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

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

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

[15] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, Oct. 2005.

[16] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *SOSP*, pages 143–155, Dec. 1995.

[17] L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *USENIX Summer Conf.*, June 1994.

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

[19] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM TOCS*, 6(1), Feb. 1988.

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

[21] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc NSDI*, 2004.

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

[23] M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, 23(5):9–21, May 1990.

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

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

[26] S. Susarla and J. Carter. Flexible consistency for wide area peer replication. In *ICDCS*, June 2005.

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

[28] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Database Systems*, 4(2):180–209, 1979.

[29] R. van Renesse. Chain replication for supporting high throughput and availability. In *Proc. OSDI*, Dec. 2004.

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

[31] Y. Zhang, J. Hu, and W. Zheng. The flexible replication method in an object-oriented data storage system. In *Proc. IFIP Network and Parallel Computing*, 2004.

A State Exposed to Predicates

As Section 2.1 noted, most safety policies can make use of predefined consistency libraries or blocking predicates. Policies may define customized predicates, and for completeness we list here the information on which they operate.

Predicates can operate on two classes of information: local built-in bookkeeping information and policy-defined events.

The underlying replication mechanisms expose four pieces of built-in bookkeeping information about the updates they have processed: (1) the local current logical version vector cVV (i.e., for each node n in the system, $cVV[n]$ is the highest logical timestamp of any event by n that has been processed by the local node); (2) the local current real time vector rVV (i.e., for each node in the system, a real time value such that the local state reflects the most recent event before that time but no events after that time); (3) whether any specified object o is *valid* (i.e., the local system stores for o an update body whose logical timestamp matches that of the most recent invalidation seen for o); and (4) whether any specified object o is *causally consistent* (i.e., if the most recent invalidation locally stored for o has logical time l_o and the local current version vector is cVV, then any update of o with logical time later than l_o must also be later than cVV: ($l_o < l'_o) \rightarrow (\forall n : cVV[n]) < l'_o$.)

In addition to using built-in bookeeping information, predicates can make use of policy-specific events generated by the policy's liveness rules. Predicates typically make use of such events to track the propagation of information to or from other nodes. For example, in a client-server system, a write might block until a node receives from the server an acknowledgement representing that the server has stored the update and invalidated all other caches.