PADRE: A Policy Architecture for building Data REplication systems

Nalini Belaramani[*], Jiandan Zheng[*], Amol Nayate[†], Robert Soulé[‡], Mike Dahlin[*], Robert Grimm[‡]
[*]University of Texas at Austin [†]IBM TJ Watson Research [‡]New York University

Abstract: This paper presents Padre, a new *policy architecture* for developing data replication systems. Padre simplifies design and implementation by embodying the right abstractions for replication in widely distributed systems. In particular, Padre cleanly separates the problem of building a replication system into the subproblems of specifying liveness policy and specifying safety policy, and it identifies a small set of primitives that are sufficient to specify sophisticated systems. As a result, building a replication system is reduced to writing a few liveness rules in a domain-specific language to trigger communication among nodes and specifying safety predicates that define when the system must block requests. We demonstrate the *flexibility* and *simplicity* of Padre by constructing a dozen substantial and diverse systems, each using just a few dozen system-specific policy rules. We demonstrate the *agility* that Padre enables by adding new features to several systems, yielding significant performance improvements; each addition required fewer than ten additional rules and took less than a day.

1 Introduction

A central task for a replication system designer is to balance the trade-offs among consistency, availability, partition-resilience, performance, reliability, and resource consumption. Because there are fundamental tensions among these properties [9, 23], no single best solution exists. As a result, when designers are faced with new or challenging workloads or environments such as geographically distributed nodes [28], mobile nodes [17, 19], or environmentally-challenged nodes [8], they often construct new replication systems or modify existing ones. Our goal is to reduce the effort required to construct or modify a replication system by providing the right abstractions to manage these fundamental trade-offs in such environments.

This paper therefore presents Padre, a new *policy architecture* that qualitatively simplifies the development of data replication systems, for environments with mobile or widely distributed nodes where data placement, request routing, or consistency constraints affect performance or availablity.

The Padre architecture divides replication system design into two aspects: liveness policy, defining how to route information among nodes, and safety policy, embodying consistency and durability requirements. 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 liveness policy, the 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? When and where should a node send updates it receives? Where should a node send invalidations when it learns of a new version of an object? What data should be prefetched or pushed to what nodes in anticipation of future requests?
- For safety policy, the observation is that consistency and durability invariants can by ensured by *blocking requests* until they do not violate those invariants. E.g., block until a write reaches at least 3 nodes, block until a server acknowledges a write, or block until local storage reflects all updates that occurred before the start of the current read.

Given these insights, the challenge to implementing Padre is to define the right set of primitives for concisely and precisely describing replication systems. We present a set of *triggers* (upcalls) exposing the flow of replication state among nodes, a set of *actions* (downcalls) to direct communication of specific subsets of replication state among nodes, and a set of *predicates* for blocking requests and state transfers. To simplify the definition of liveness (routing) policies in terms of these primitives, we define R/OverLog[1], an extension of the OverLog [24] routing language.

Even if an architecture is conceptually appealing, to be useful it must help system builders. We demonstrate Padre's power by constructing a dozen systems spanning a large portion of the design space; we do not believe this feat would have been possible without Padre. In particular, in contrast with the ten thousand or more lines of code it typically takes to construct such a system using standard practice, it requires just 6-75 policy rules and a handful of safety predicates to define each system over Padre. We believe this two to three orders of magnitude reduction in code volume is illustrative of the qualitative

[1] Pronounced "R over OverLog" (for Replication over OverLog) or "Roverlog."

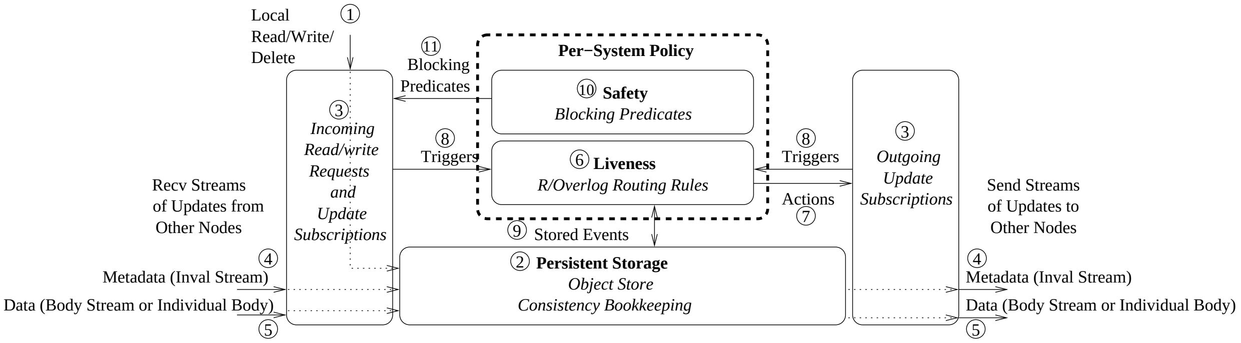

Fig. 1: Overview of the Padre architecture for building replication systems by expressing policy.

simplification Padre represents for system implementers. This simplification stems from two sources: (1) Padre captures the right abstractions for policy, which reduces the conceptual effort needed to design a system and (2) Padre primitives embody the right building blocks, so a developer does not have to reinvent them.

The rest of this paper describes Padre, demonstrates how to build systems with Padre, and evaluates the approach. The paper's contributions are (1) defining a set of abstractions that are useful for building and reasoning about replication systems and (2) providing a system that realizes these abstractions to facilitate system building.

2 Architecture

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. Our design choices for Padre are driven by the need to accommodate a broad design space while allowing policies to be simple and efficient.

Figure 1 provides an overview of the Padre architecture. To write a policy in Padre, a designer writes liveness rules to direct communication between nodes and safety predicates to block communication and requests until system invariants are met. To give intuition for how to build a system by writing such rules, this section provides an overview of the mechanisms on which these rules depend, of the abstractions for liveness rules, and of the abstractions for safety predicates.

Interface and mechanisms. As Figure 1 illustrates, Padre exposes a *local read/write/delete* object store interface to applications (①in the figure). These functions operate on local *persistent storage* mechanisms that handle object storage and consistency bookkeeping ②.

To propagate updates among machines, Padre requires a mechanism to transfer streams of updates from one node to another. For efficiency and flexibility, our Padre prototype utilizes the PRACTI [2, 46] protocol, which allows a node to set up a *subscription* to receive updates to a desired subset of objects ③.

Three properties of the PRACTI protocol are relevant to understanding Padre:

1. *Partial Replication:* For efficiency, a subscription can carry updates for any subset of the system's objects, and the protocol uses separate subscriptions for update metadata and update data. The update metadata are represented by *invalidation streams* that provide information about the logical times and ordering of updates ④. The update data are represented by *body streams* or *individually fetched* update bodies ⑤.

2. *Any Consistency:* Invalidation streams include sufficient information for the system to enforce a broad range of consistency guarantees, and the system automatically tracks objects' consistency status.

3. *Topology Independence:* A subscription can flow between any pair of nodes, so any topology for propagating updates can be constructed. Subscriptions can change dynamically, modifying a topology over time.

As a result of these properties, policies can set up subscriptions for any subset of updates between any pair of nodes at any time, and the underlying mechanisms handle the details of transferring this data and metadata and tracking consistency state.

Implementation details about the PRACTI protocol are available elsewhere [2, 46].

Two additional features of the local read/write interface must be mentioned.

First, for flexibility, this interface allows a write operation to atomically update one or more objects. It also allows writes to be made to specific byte-ranges.

Second, in order to support algorithms that define a total order on operations [29], the interface supports a variation on write called *writeCSN* (write commit sequence number) that assigns a CSN, in the form of a logical time, to a previous update.

Given these storage, consistency bookkeeping, and update communication mechanisms, which are common across all systems built on Padre, the focus of this paper is defining policy: what are the right primitives for making it easy to express the policies that define replication systems?

Liveness. The first half of defining a system in Padre is to define a set of *liveness rules* ⑥ that orchestrate communication of updates between nodes. For example, a client in a client-server system should send updates to the server on writes and fetch data from the server on read misses while a node in TierStore [8] should transmit any update it receives for a volume to all children subscribed for that volume.

The liveness rules describe how to generate communication *actions* (downcalls) ⑦ in response to *triggers* (upcalls) ⑧ and *stored events* ⑨.

Actions route information between nodes' persistent storage by setting up subscriptions for data or metadata streams or by initiating fetches of individual objects.

Triggers provide information about the state and needs of the underlying replication system. They include local events (e.g., local read blocked, local write issued), connection events (e.g., body subscription start, invalidation subscription failed), and message arrival events (e.g., invalidation arrived or body arrived). These three classes of triggers provide sufficient information to build sophisticated policies.

Finally, *stored events* ⑨ provide a means to store and retrieve the hard state many systems need make to their routing decisions. For example, a node in a client-server system needs to know who the server is and a distribution node in a dissemination system [8] needs to know who has subscribed to what publications.

Designers define liveness policy rules that initiate actions in response to triggers and stored events using a rule-based language we call R/OverLog, which extends OverLog [24] to the needs of replication policy.

Safety. Every replication system guarantees some level of consistency and durability. Padre casts consistency and durability as *safety policy* ⑩ 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, blocking writes until the resulting updates make it to some or all of the system's nodes, or both. Similarly, durability policies often require writes to propagate to some subset of nodes (e.g., a central server, a quorum, or an object's "gold" nodes [31]) before the write is considered complete or before the updates are read by another node.

Padre therefore allows *blocking predicates* ⑪ to block a read request, a write request, or application of received updates until a predicate is satisfied. The predicates specify conditions based on the consistency bookkeeping information maintained by the persistent storage or they can wait for the arrival of a specific message generated by the liveness policy. Basing the predicates on these inputs suffices to specify any order-error or staleness error constraint in Yu and Vahdat's TACT model [44] and thereby implement a broad range of consistency models from best effort coherence to delta coherence [37] to causal consistency [21] to sequential consistency [22] to linearizability [44].

Mechanism	
Persistent storage	Store objects and maintain consistency metadata
Subscriptions	Register interest in receiving updates to some subset of objects
Liveness Policy	
Actions	Route information from local persistent storage to remote persistent storage
Triggers	Notify liveness policy of local operations, messages, and connections
Stored Events	Store/retrieve persistent state that affects routing
Safety Policy	
Predicates	Block read, writes, and node-to-node updates to ensure safety

Fig. 2: Padre abstractions.

Summary. Figure 2 summarizes the main abstractions provided by Padre to build replication systems.

2.1 Example

To illustrate the approach, we describe the design and implementation of a simple client-server system as a running example. This simple system includes support for a client-server architecture, invalidation callbacks [16], sequential consistency [22], correctness in the face of crash/recovery of any nodes, and configuration; for simplicity, it assumes that a write overwrites an entire file.

We choose this example not because it is inherently interesting but because it is simple yet sufficient to illustrate the main aspects of Padre, including support for coarse- and fine-grained synchronization, consistency, durability, and configuration. In Sec. 4.1, we extend the example with features that are relevant for practical deployments, including leases [11], cooperative caching [7], and an NFS [32] interface with partial-file writes.

Ideally, a policy architecture should let a designer define such a system by describing its high level properties and letting the runtime handle the details. For example, a designer might describe a simple client-server system using the following statements:

L1. On a read miss, a client should fetch the missing object from the server.

L2. On a read miss, a client should register to receive callbacks for the fetched object.

L3. On a write, a client should send the update to the server.

L4. Upon receiving an update of an object, a server should send invalidations to all clients registered to receive callbacks for that object.

L5. Upon receiving an invalidation of an object, a client should send an acknowledgment to the server and cancel the callback.

L6. Upon receiving all acknowledgments for an update, the server should inform the writer.

Additionally, to configure the system the clients and server need to know each other.

L7. At startup, a node should read the server ID from a configuration file.

Similarly, a designer can make simple statements about the desired durability and consistency properties of the system. One such safety property relates to durability:

S1. Do not allow a write by a client to be seen by any other client until the server has stored the update.

Other safety properties relate to ensuring sequential consistency. One way to ensure sequential consistency is to enforce three properties S2, S3, and S4. The first two are straightforward:

S2. A write of an object must block until all earlier versions have been invalidated.

S3. A read of an object must block until the reader holds a valid, consistent copy of the object.

The last safety property for this formulation of sequential consistency is more subtle. Since multiple clients can issue concurrent writes to multiple objects, we must define some global sequential order on those writes and ensure that they are observed in that order. In a client-server system it is natural to have the server set that total order:

S4. A read or write of an object must block until the client is guaranteed to observe the effects of all earlier updates in the sequence of updates defined by the server.

This rule requires us to add one more liveness rule:

L8. Upon receiving acknowledgements for all of an update's invalidations, the server should assign the update a position in the global total order.

The 12 statements above seem to be about as simple a description as one could hope to have of our example system. If we can devise an architecture that allows a designer to build such a system with something close to this simple description while the architecture and runtime hide or handle the mechanical details, we will regard Padre as a success.

2.2 Excluded properties

There are at least three properties that Padre does not address or for which it provides limited choice to designers: 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 safety and liveness to address the CAP [9], performance [23], reliability, and resource consumption trade-offs.

Second, Padre exposes an object-store interface for local reads and writes. It does not expose other interfaces such as a file system or a tuple store. We believe that these interfaces are not difficult to incorporate. Indeed, we have implemented an NFS interface over our prototype [3].

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 [8, 18, 19, 34, 39]. We believe it is straightforward to extend Padre to support other models such as Bayou's application-specified conflict detection and reconciliation programs [39].

3 Detailed design

It is well and good to say that designers should build replication systems by specifying liveness with actions, triggers, and stored events and by specifying safety with blocking predicates, but designers can only take this approach if Padre provides the right set of primitives from which to build. These primitives must be simple, expressive, and efficient. Given the high-level Padre architecture, precisely defining these primitives is the central intellectual challenge of Padre's detailed design.

We first detail Padre's abstractions for defining liveness and safety policy. We then discuss two crosscutting design issues: fault tolerance and correctness.

3.1 Liveness policy

In Padre, a liveness policy must set up invalidation and body subscriptions so that updates propagate among nodes to meet a designer's goals. For example, if a designer wants to implement hierarchical caching, the liveness policy would set up subscriptions among nodes to send updates up and to fetch data down. If a designer wants nodes to randomly gossip updates, the liveness policy would set up subscriptions between random nodes. If a designer wants mobile nodes to exchange updates when they are in communications range, the liveness policy would probe for available neighbors and set up exchanges at opportune times. If a designer wants a laptop to hoard files in anticipation of disconnected operation [19], the liveness policy would periodically fetch files from the hoard list. Etc.

Liveness policies do such things by defining how *actions* are taken in response to *triggers* and *stored events*.

3.1.1 Actions

The basic abstraction provided by a Padre action is simple: *an action sets up a subscription to route updates from one node to another.*

The details of this primitive boil down to making subscriptions efficient by letting designers control what information is sent. To that end, the subscription actions API gives the designer 5 choices:

1. *Select invalidations or bodies.* Each update comprises an invalidation and a body. An invalidation indicates that an update of a particular object occurred at a particular instant in logical time; invalidations help enforce consistency by notifying nodes of updates and by ordering the system's events. Conversely, a body contains the data for a specific update.

2. *Select objects of interest.* A subscription specifies which objects are of interest to the receiver, and the sender only includes updates for those objects. Padre exports a hierarchical namespace so a group of related objects can be concisely specified (e.g., /a/b/*).

3. *For a body subscription, select streaming or single-item mode.* A subscription for a stream of bodies sends updated bodies for the objects of interest until the subscription terminates; such a stream is useful for coarse-grained replication or for prefetching. Alternatively, a policy can send a single body by having the sender push it or the receiver fetch it. For reasons discussed below, invalidations are always sent in streams.

4. *Select the start time for a subscription.* A subscription specifies a logical start time, and the stream sends all updates that have occurred since that time.

5. *Specify a catchup mode for a subscription.* If the start time for a subscription is earlier than the sender's current logical time, then the sender can transmit either a *log* of the events that occurred between the start time and the current time or a *checkpoint* that includes just the most recent update to each byterange since the start time. Sending a log is more efficient when the number of recent changes is small compared to the number of objects covered by the subscription. Conversely, a checkpoint is more efficient if (a) the start time is in the distant past (so the log of events is long) or (b) the subscription is for only a few objects (so the size of the checkpoint is small). Note that once a subscription catches up with the sender's current logical time, updates are sent as they arrive, effectively putting all active subscriptions into a mode of continuous, incremental log transfer.

Figure 18 in the Appendix lists the full *actions* API.

Example. Consider the operation of the simple client-server system. The actions required to route bodies and invalidations are simple, entailing four Padre actions to handle statements L1-L4 in Section 2.1.

In particular, on a read miss for object *o*, a client takes two Padre actions. First, it issues a single-object fetch for the current body of *o* (L1). Second, it sets up an invalidation subscription for object *o* so that the server will notify the client if *o* is updated (L2 and L4). As a result of these actions, the client will receive the current version of *o*, receive consistency bookkeeping information for *o*, and receive an invalidation when *o* is next modified.

To send a client's writes to the server (L3), rather than set up fine-grained, dynamic, per-object subscriptions as we do for reads, at startup a client's liveness policy simply creates two coarse-grained subscriptions: one to send data (bodies) for all objects starting from the server's current logical time and another to do the same for metadata (invalidations.)

3.1.2 Triggers

Liveness policies invoke Padre actions when Padre *triggers* signal important events.

- *Local read, write, delete operation* triggers inform the liveness policy when a read blocks because it needs additional information to complete or when a local update occurs.
- *Messages receipt* triggers inform the liveness policy when an invalidation arrives, a body arrives, a fetch succeeds, or a fetch fails.
- *Connection event* triggers inform the liveness policy when subscriptions are successfully established, when a subscription has allowed a receiver's state to catch up with a sender's state, or when a subscription is removed or fails.

Figure 18 in the Appendix lists the full triggers API.

Example. In the simple client-server example, to issue a demand read request and set up callbacks, statements L1 and L2 are triggered when a read blocks; to have a client acknowledge an invalidation, statement L5 is triggered when a client receives an invalidation message; and to notify a writer when its write has completed, statements L6 and L8 are triggered when a server receives acknowledgements from clients.

Statement L3 requires a client to send all updates to the server. Rather than setting up a subscription to send *o* when *o* is written, our implementation maintains a coarse-grained subscription for all objects at all times. Establishment of this subscription is triggered by system startup and by connection failure.

Statement L4 requires a server to send invalidations to all clients registered to receive callbacks for an updated object. These callback subscriptions are set up when a client reads data (L2), and because these subscriptions are already established, the liveness policy need not take any additional action when an update arrives.

3.1.3 Stored events

Systems often need to maintain hard state to make routing decisions. Supporting this need is challenging both because we want an abstraction that meshes well with our event-driven, rule-based policy language and because the techniques must handle a wide range of scales. In particular, the abstraction must handle not only simple, global configuration information (e.g., the server identity in a client-server system like Coda [19]), but it must also scale up to per-volume or per-file information (e.g., which children have subscribed to which volumes in a hierarchical dissemination system [8, 28] or which nodes store the gold copies of each object in Pangaea [31].)

To provide a uniform abstraction to address this range of concerns, Padre provides *stored events.* To use stored events, policy rules produce one or more tuples that are stored into a data object in the underlying persistent object store. Rules also define when the tuples in an object should be retrieved, and the tuples thus produced can then trigger other policy rules. Figure 18 in the Appendix shows the full API for stored events.

To illustrate the flexibility of stored events, we first illustrate their use for simple configuration information in the running example. We then illustrate several more dynamic, fine-grained applications of the primitive.

Example: Simple client-server. Clients must route requests to the server, so the liveness policy needs to know who that is. At configuration time, the installer writes the tuple *ADD_SERVER* [*serverID*] to the object `/config/server`. At startup, the liveness policy produces the stored events from this object, which causes the client's liveness policy to update its internal state with the identity of the server.

Example: Hoard list. To append an item to a hoard list [19], a rule can create the tuple *HOARD_ITEM, objId* and use the stored event abstraction to add that tuple to the persistent object `/config/[nodeID]/hoardlist`. Later when the policy wishes to walk the hoard list to prefetch objects, it can use the stored event abstraction to produce all of the tuples stored in `hoardlist`, which causes the runtime system to generate all of the *HOARD_ITEM* tuples stored in the object. The production of these tuples, in turn, activates rules that cause actions such as fetching an item from the server.

Example: Per-volume subscriptions. In a hierarchical dissemination system, to set up a persistent subscription for volume *v* from a parent *p* to a child *c*, a rule at the parent stores the tuple *SUBSCRIPTION c v* to an object `/subs/p/v`. Later, when a trigger indicates that an update for *v* has arrived at *p*, a policy rule uses the stored events abstraction to produce the events stored in `/subs/p/v`, which, in turn, activates rules that cause actions such as transmission of recent updates of *v* to each of the children with subscriptions.

Example: Per-file location information. In Pangaea [31], each file's directory entry includes a list of *gold nodes* that store copies of that file. To implement such fine-grained, per-file routing information, a Padre liveness policy creates a `goldList` object for each file, stores several *GOLD_NODE objId nodeId* tuples in that object, and updates a file's `goldList` whenever the file's set of gold nodes changes (e.g., due to a long-lasting failure.) When a read miss occurs, the liveness policy produces the stored *GOLD_NODE* tuples from file's `goldList`, and these tuples activate rules that route a read request to one of the file's gold nodes.

3.1.4 Liveness policies in R/OverLog

To write a liveness policy, a designer writes rules in R/OverLog. As in OverLog [24] a program in R/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) :-   in1(@X, A, B, C), t1(@X, A, B, D),
                   t2(@X, A, _), C < D
```

indicates that whenever there exist at node *X* a tuple *in1*, any entry in table *t1*, and any entry in table *t2* such that all have identical second fields (*A*), *in1* and the tuple from *t1* have identical third fields (*B*), and the fourth field (*C*) of *in1* is smaller than the fourth field (*D*) in the tuple from *t1*, create a new tuple (*out*) at node *Y* using the second and fourth fields from *in1* (*A* and *C*). Note that for the tuple in *t2*, the _ wildcard matches anything for field three.

R/OverLog extends OverLog by adding type information to tuples and by efficiently implementing the interface for inserting and receiving tuples from a running OverLog program. This interface is important for Padre to inject *triggers* to and receive *actions* from the policy.

R/OverLog implements *fixed point semantics* by which all rules triggered by the appearance of the same tuple are executed atomically in isolation from one another. Once all such tuples are executed, their table updates are applied, their actions are invoked, and the tuples they produce are enqueued for future execution. Then another new tuple is selected and all of the rules it triggers are executed.

Note that if learning a domain specific language is not one's cup of tea, one can define a (less succinct) policy by writing Java handlers for Padre triggers and stored events to generate Padre actions and stored events.

Example. Statement L1 of our simple client server example allows a client to fetch a missing object from the server when it suffers a read miss. We can write two R/OverLog rules to express this complete statement:

Blocking Conditions	
isValid	Block until node has body corresponding to highest received invalidation for the target object
isComplete	Block until object's consistency state reflects all updates before the node's current logical time
isSequenced *VV|CSN*	Block until object's total order is established via Golding's algorithm (VV) [10] or an explicit commit (CSN) [29].
propagated *nodes, count, p*	Block until *count* nodes in *nodes* have received my *p*th most recent write
maxStale *nodes, count, t*	Block until I have received all writes up to $(operationStart - t)$ from *count* nodes in *nodes*.
tuple *tuple-spec*	Block until receiving a tuple matching *tuple-spec*

Fig. 3: Conditions available for defining safety policies.

```
L1a: clientRead(@S, C, Obj, Off, Len) :-
     TRIG_informReadBlock(@C, Obj, Offset, Len, _),
     TBL_serverId(@C, S),  C ≠ S.

L1b: ACT_sendBody(@S, S, C, Obj, Off, Len) :-
     clientRead(@S, C, Obj, Off, Len).
```

The first rule is triggered when a read blocks at a client. It generates a *clientRead* tuple at the server. The appearance of this tuple at the server generates a *sendBody* action.

This approach allows us to define a liveness policy using rules that track our original statements L1-L8. See the appendix for a full listing of the 21-rule R/OverLog liveness policy for this example. Most of our original statements map to one or two rules. Tracking which nodes require or have acknowledged invalidations is a bit more involved, so L6 maps to eight rules that maintain lists of clients that must receive callbacks.

Although this example is simple, our experience for a broad range of systems is that Padre generally provides a natural, precise, and concise way to express a designer's intent and that it meets our goal of allowing a designer to construct a system by writing high-level statements describing the system's operation.

3.2 Safety policy

In Padre, a system's safety policy is defined by a set of blocking predicates that prevent state observation or updates until consistency or durability constraints are met.

Padre defines 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 modifies the underlying local store. *ReadEndBlock* and *WriteEndBlock* block read and write requests 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.

Figure 3 lists the conditions available to safety predicates. *isValid* is useful for enforcing coherence on reads and for maximizing availability by ensuring that invalidations received from other nodes are not applied until they can be applied with their corresponding bodies [8, 28]. *isComplete* and *isSequenced* are useful for enforcing consistency semantics like causal, sequential, or linearizable. *propagated* and *maxStaleness* are useful for enforcing TACT order error and temporal error tunable consistency guarantees [44]. *propagated* is also useful for enforcing some durability invariants. Cases not handled by these predicates are handled by *tuple*. *Tuple* becomes true when the liveness rules produce a tuple matching a specified pattern.

For maximum flexibility, each read/write operation includes parameters to specify the safety predicates. Replication system developers typically insulate applications and users from the full interface by adding a simple wrapper that exposes a standard read/write API and that adds the appropriate parameters before passing the requests through to Padre.

The *ApplyUpdateBlock* predicate is set by a function call that supplies the predicate to be enforced on incoming updates.

Example. Part of the reason for focusing on the client server example is that the example illustrates both some simple aspects of safety policy and some that are relatively complex.

Statement S1 requires the server to block application of invalidations until the corresponding body can be simultaneously applied. This restriction is easily enforced by setting *isValid* for the *ApplyUpdateBlock* predicate.

Statement S2 requires us to prevent a write from completing until all earlier versions of the updated object have been invalidated. So, we define a *writeComplete objId logicalTime* tuple that the server generates once it has gathered acknowledgements from all nodes that had been caching the object (L6), and we set the *writeEndBlock* predicate to block until this tuple is produced.

Statement S3 and S4 require a read to return only sequentially consistent data, so the *ReadNowBlock* predicate sets three flags: *isValid* ensures that the read returns only when the body is as fresh as the consistency state; *isComplete* ensures that the read returns only when the consistency metadata for the object is current; and *isSequenced CSN* ensures that the read of an object returns only once the reader has observed the server's commit of the most recent write of that object. Similarly, S4 requires us to prevent a write from completing until the local state reflects all previously sequenced updates, so the *writeEndBlock* predicate requires the *isSequenced CSN* condition.

Theorem 1. *The simple client server implementation enforces sequential consistency.*

Proof. We first define a total order on all updates and reads. We then show this total order is consistent with every node's program order.

Total order on updates. An update is assigned two logical timestamps called accept stamps: the writer's logical time when the writer issued the write and the server's logical time when it commits the write. An accept stamp has two parts, a node ID and a value. We define a total order on accept stamps:

$as1 < as2$ iff $as1.value < as2.value$
OR
$(as1.value = as2.value$
AND $as1.node < as2.node)$

The value of a timestamp is a Lamport clock [21]: a node n maintains the invariant that its $n.value$ exceeds the $value$ in any event it has previously observed. It does so by setting $n.value = \max(event.value + 1,\ n.value)$ whenever it receives an event and by setting $n.value = n.value + 1$ whenever it generates an event.

Note that if an update has a logical write time of w, then if it has a commit time of c, $c > w$: since a server issues a commit after seeing a write, the commit's record must have an accept stamp that exceeds the accept stamp of the write's record.

When two updates modify the same byte, the underlying Padre data storage and transfer mechanisms implement a last writer wins policy. So, if update $u2$'s write time exceeds update $u1$'s write time ($u2.w > u1.w$) and both updates modify the same byte, then update $u2$ must appear after update $u1$ in our total order.

We therefore assign an *effective commit time* to an update u that was committed by the server with commit c to be the following tuple:

$u.et =$ $c'.as$ where c' is a commit record s.t.
$c'.as > u.as$
AND
$c'.target = u.target$
AND
$(\nexists c''$ s.t. $c''.target = u.target$ AND
$u.as < c''.as < c'.as\ )$

I.e., u is effectively committed by the *next* commit record that commits any write that updates the same data as u.

Notice that for update $u1$ and $u2$ both updating the same data, if update $u1$ has an earlier logical timestamp than update $u2$ but commit $c2$ has an earlier logical timestamp than commit $c1$, then both $u1$ and $u2$ have an effective commit time of $c2.as$.

Write polistion. We can now define an update u's *position* based on u's effective commit time and accept stamp.:

u.pos = [u.et, u.as]

and we order the positions to define a total order on all updates:

$u1.pos < u2.pos$ iff $u1.pos.et < u2.pos.et$
OR
$(u1.pos.et = u2.pos.et$
AND $u1.pos.as < u2.pos.as)$

Read position. A read's effective time is the largest commit stamp for any write issued by the reader or received by the reader before the read accesses the object store. Since multiple reads can have the same effective time, we break ties using the nodes accept stamp at the time of the read. Note that each read operation advances the node's accept stamp.

We can now define a read's *position* based on its effective time and accept stamp:

r.pos = [r.et, r.as]

and we order the positions to define a total order:

$r1.pos < r2.pos$ iff $r1.pos.et < r2.pos.et$
OR
$(r1.pos.et = r2.pos.et$
AND $r1.p.as < r2.p.as)$

Total order on all reads and writes. A total order on all reads and writes is defined by the write position and the read position. Specifically all reads with an effective time et come after all writes with effective commit time et.

$o1.pos < o2.pos$ iff $o1.pos.et < o2.pos.et$

OR

$o1.pos.et = o2.pos.et$

AND

$[(o1.op = o2.op$ AND
$o1.as < o2.as)$

OR

$(op1.op \neq o2.op$ AND
$o2.op = READ)]$

Order of local operations is consistent with the total order. Recall that a write blocks until the server assigns a commit record for the write and an acknowledgement is received. Hence for two consecutive local writes, the effective time of the first write is always smaller than the effective time of the second write. For two writes, $w1$ and $w2$, if $w1$ happened before $w2$, we can conclude that ($w1.as < w2.as$ AND $w1.et < w2.et$). Hence, $w1.pos < w2.pos$

Since reads are issued with the *ReadNowBlock* predicate having the *isSequenced CSN* condition set, a read of an object blocks until the node receives the commit record for that write. Thus, the effective time of a read of byte is at least as large as the last committed write of that byte. If a write, $w1$, happened before a read, $r1$, we can conclude that ($w1.as < r1.as$ AND $w1.et <= r1.et$)

Since for the same effective times, reads are ordered after writes, $w1.pos < r1.pos$.

On the other hand, if a read, $r1$, locally happened before another read, $r2$, $r2$'s effective time is at least as large as $r1$. Since $(r1.as < r2.as$ AND $r1.et <= r2.et)$, we can conclude that $r1.pos < r2.pos$.

Finally, if a write, $w1$, locally happened before a read, $r1$, $r1$'s effective time is at least as large as that of $w1$. Since, $(w1.as < r1.as$ AND $w1.et <= r1.et)$. we can conclude that $w1.pos < r1.pos$.

Putting all the above cases together, we have shown that if an operation, $op1$, happened before another operation, $op2$, on a node, then $op1.pos$ is smaller than $op2.pos$.

The results of local operations are consistent with the total order. A write does not get a CSN until all nodes caching the target object have been invalidated. Since reads are issued with the *ReadNowBlock* predicate having the *isSequenced CSN* condition set, once data to be read is invalidated, a read of that object blocks until the node receives the commit record. Thus, for all writes that can affect the results of the read, either (a) the read completes before a write of the same object is assigned a CSN, gets assigned an effective time smaller than that of the write, comes before the write in the total order and does not reflect or (b) the read completes after the reader receives the CSN for the write, gets assigned an effective time equal to the commit time of the write and reflects the update.

Hence, any read reflects the result of all writes that precede the read in the total order and none that succeed it in the total order.

Given that there is total order on all operations and results of local operations are consistent with that total order, we conclude that the client server implementation enforces sequential consistency. □

Overall, we find that Padre's separation of safety and liveness and the simple view of consistency state exported by Padre not only simplifies implementing safety semantics but also simplifies reasoning about their correctness.

3.3 Crosscutting issues

Much of the simplicity of Padre policies comes because the Padre primitives automatically handle many low-level details that otherwise complicate a system designer's life. Some of these aspects of Padre can be illustrated by describing how a designer approaches two cross-cutting issues in Padre: tolerating faults and reasoning about the correctness of a policy.

3.3.1 Fault tolerance

At design time, Padre's role is to help a designer express design decisions that affect fault tolerance. For example, by setting up subscriptions to distribute data and metadata, the liveness policy determines where data are stored, which affects both durability and availability. E.g., a designer can store all data at all nodes [29], store data at a group of redundant, well-maintained servers and hoard data at clients for disconnected operation [19], store data at any k nodes [31], etc. Similarly, by defining a static or dynamic topology for distributing updates or fetching data, the liveness policy can affect availability by determining when nodes are able to communicate. E.g., a designer can specify an adaptive, gossip-based topology [29], a hierarchy that makes use of delay tolerant links [8] or that reconfigures around faults [42], a static hierarchy or client-server topology, etc. Finally, by specifying what consistency constraints to enforce, the safety policy affects availability, with stronger consistency generally making it more difficult or expensive to maintain a given level of system availability.

Then, during system operation, Padre liveness rules define how to detect and react to faults. Often, policies simply detect failures when failed network connections invoke a *subscription failed* trigger, but Padre's use of a variation of OverLog for defining liveness policies also allows more sophisticated systems to include rules to actively monitor nodes' connectivity [24], or even implement a group membership or agreement algorithm [36]. Example reactions to faults include retrying or aborting requests, rerouting requests, or making additional copies of data whose replication factor falls below a low water mark.

Finally, when a node recovers, Padre's role is to insulate the designer from the low-level details. Upon recovery, local mechanisms first reconstruct local state from persistent logs. Then, Padre's subscription primitives abstract away many of challenging details resynchronizing node state, allowing per-system policies to focus on reestablishing the system's high-level update flows.

In particular, an invalidation stream for object set O not only transfers detailed information about the updates of objects in O but also carries additional information identifying sets of objects to which causally-preceding updates may have been omitted. As a result, a Padre node's consistency bookkeeping logic (②in Figure 1) automatically tracks the subset of objects for which complete information is present and another subset that may be affected by omitted updates [2, 46], and safety policies can block access to the latter if they desire to enforce FIFO or stronger consistency.

Notably, these mechanisms track this consistency state even across crashes that could introduce gaps in the sequences of invalidations sent between nodes. As a result, crash recovery in most systems simply entails restoring lost connections and letting the underlying mechanisms ensure that local state reflects any updates that were missed.

Note that the protocols for transferring invalidation streams use two techniques to minimize the cost of sending the extra bookkeeping information needed to flag potentially inconsistent subsets of data. First, they use *imprecise invalidations* to concisely summarize missing information [2]. Second, they multiplex multiple subscriptions over the same network connection so that they only need to send information relating to any update once per connection rather than repeatedly sending this bookkeeping information once per subscription [46]. Implementation details for these low-level mechanisms appear elsewhere [2, 46].

Example. In the simple client-server system, significant design-time decisions include requiring all data to be stored at and fetched from a central server and enforcing sequential consistency.

Because of the semantics embedded in the subscription primitives, simple techniques then suffice to ensure correct operation even if the client crashes and recovers, the server crashes and recovers, or network connections fail and are restored.

In particular, after a subscription carrying updates from a client to the server breaks, the server periodically attempts to reestablish the connection. Because the server always restarts a subscription from where it left off, once a local write is applied to a client's local state, it eventually must be applied to the server's state

Additionally, after a connection carrying invalidations from the server to the client breaks and is reestablished, Padre's low-level consistency bookkeeping mechanisms advance the client's consistency state only for objects whose subscriptions have been added to the new connection. Other objects are then treated as potentially inconsistent as soon as the first invalidation arrives on the new connection. As a result, no special actions are needed resynchronize a client's state during recovery. Note that to ensure that writes can complete and to avoid having to reestablish all subscriptions, the part of the server's liveness policy that waits for clients to acknowledge invalidations treats a client's acknowledgement for an invalidation as also acknowledging all earlier invalidations from the server to the client.

3.3.2 Correctness

Three aspects of Padre's core architecture simplify reasoning about the correctness of Padre systems. First, the primitives over which policies are built handle the low-level bookkeeping details needed to track consistency state. Second, the separation of policy into safety and liveness reduces the risk of safety violations: safety constraints are expressed as simple invariants and errors in the (more complex) liveness policies tend to manifest as liveness bugs rather than safety violations. Third, the conciseness of Padre specifications greatly facilitates analysis, peer review, and refinement of designs.

Example In the client-server system, the same abstractions that simplify reasoning about consistency synchronization across failures also make systems robust to design errors. For example, if a policy starts a subscription "too late," fails to include needed objects in a subscription, or fails to set up a subscription from a node that has needed updates, the bookkeeping logic will identify any affected items as potentially inconsistent. Additionally, in such a situation, safety constraints will block reads or writes or both, but they will not allow applications to observe inconsistent data. Finally, the conciseness of the specification facilitates analysis: the system is defined by 21 liveness rules and 5 safety predicates (see the Appendix for the entire specification), and most of the 21 rules are trivial; the difficult parts of the design come down to 9 rules (L6a-L6i).

3.4 Local interface libraries.

The object store interface provided by Padre is intended as a simple low-level API over which a developer would typically add a wrapper through which applications send requests. To facilitate deployment of useful Padre systems, we provide several pre-built, standard wrapper libraries.

The simplest of these libraries provide high-level consistency or durability properties by wrapping the read and write calls to set the appropriate blocking parameters. We provide simple wrappers for linearizability, sequential consistency, causal consistency [21], bounded staleness [37, 44], best effort coherence, PRAM/FIFO consistency [23], and order error [44]. We also provide a wrapper library that does read and write buffering to provide open/close consistency [19].

Another use of wrapper libraries is to provide alternative interfaces to our object store. For example, we provide a wrapper that provides a user-level NFS server (similar to SFS [25] but written in Java) so that a user can run a Padre node with this wrapper on their local machine and then mount the file system; note that in this configuration the NFS protocol is used between the local in-kernel NFS client and the local user-level NFS server, and the Padre runtime handles communication between machines.

4 Evaluation

A policy architecture for replication systems should be flexible, should simplify system building, should facilitate the evolution of systems, and should have good performance. To examine the first three factors, our evaluation centers on a series of case studies. We then examine the performance of the prototype.

Experimental environment. The prototype implementation uses PRACTI [2, 46] to provide the mechanisms over which policy is built. We implement a R/OverLog to Java compiler using the xtc toolkit [12, 15]. Except where noted, all experiments are carried out on machines with single-core 3GHz Intel Pentium-IV Xeon processors, 1GB of memory, and 1Gb/s Ethernet. Machines are connected via an Emulab [41], which allows us to vary network latency. We use Fedora Core 6, BEA JRocket JVM Version 27.4.0, and Berkeley DB Java Edition 3.2.23.

4.1 Full example

In previous sections, we discuss implementation of a simple client-server system. This system requires just 21 Padre liveness rules and five Padre safety predicates, and it implements a client-server architecture, callbacks, sequential consistency, crash recovery, and configuration.

We can easily add additional features to make the system more practical.

First, to ensure liveness for all clients that can communicate with the server, we use volume leases [43] to expire callbacks from unreachable clients. Adding volume leases requires an additional safety predicate to block client reads if the client's view of the server's state is too stale. The liveness implementation keeps the client's view up-to-date by sending periodic heartbeats via a volume lease object. It requires 3 liveness rules to have clients maintain subscriptions to the volume lease object and have the server put heartbeats into that object, and 4 more to check for expired leases and to allow a write to proceed once all leases expire. Note that by transporting heartbeats via a Padre object, we ensure that a client observes a heartbeat only after it has observed all causally preceding events, which greatly simplifies reasoning about consistency.

Second, we add cooperative caching [7] by replacing the rule that sends a body from the server with 6 rules: 3 rules to find a helper and get data from the helper, and 3 rules to fall back on the server if there no helper is found or when the helper fails to satisfy the request. Note that reasoning about cache consistency remains easy because invalidation metadata still follow the client-server paths, and the safety predicates ensure that a body is not read until the corresponding invalidation has been processed. In contrast, some previous implementations of cooperative caching found it challenging to reason about consistency [5].

Third, we add support for partial-file writes by adding seven rules to track which blocks each client is caching and to cancel a callback subscription for a file only when all blocks have been invalidated.

Fourth, we add three rules to the server that check for blind writes when no callback is held and to establish callbacks for them.

Figure 4 illustrates the functionality of the enhanced system. To highlight the interactions, we add a 50ms delay on the network links between the clients and the server. We configure the system with a 2 second lease heartbeat and a 5 second lease timeout. In this experiment, one client repeatedly reads an object and then sleeps for 500ms and another client repeatedly writes the object and sleeps for 2000ms. We plot the start time, finish time, and value of each operation.

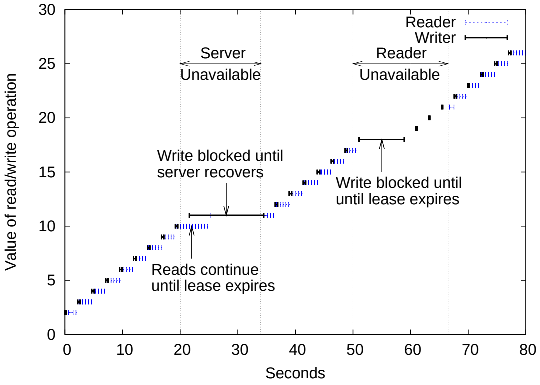

Fig. 4: Demonstration of full client-server system. The x axis shows time and the y axis shows the value of each read or write operation.

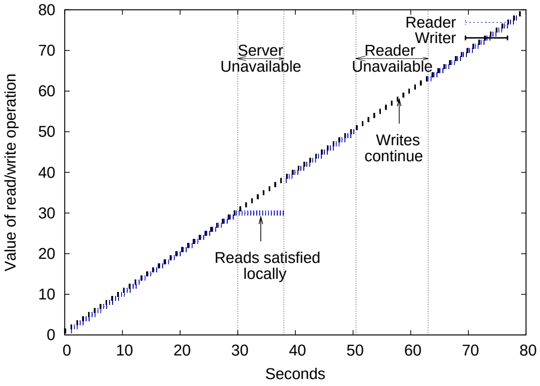

Fig. 5: Demonstration of TierStore under a workload similar to that in Figure 4.

During the first 20 seconds of the experiment, as the figure indicates and as promised by Theorem 1, sequential consistency is enforced.[2]

We kill the server process 20 seconds into the experiment and restart it 10 seconds later. While the server is down, writes block immediately and reads continue until the lease expires. Both resume shortly after the server restarts, and the mechanics of subscription reestablishment ensure that consistency is maintained.

We kill the reader at 50 seconds and restart it 10 seconds later. Initially, writes block, but as soon as the lease expires, writes proceed. When the reader restarts, reads resume as well.

4.2 Example: Weaker consistency

Many distributed data systems weaken consistency to improve performance or availability. For example, Figure 5 illustrates a similar scenario as Figure 4 but using the Padre implementation of TierStore [8], which enforces best effort coherence rather than sequential consistency and which propagates updates according to volume subscriptions rather than via demand reads. As a result, all reads and writes complete locally and without blocking, so both performance and availability are improved.

4.3 Additional case studies

This section discusses our experience constructing 7 base systems and 5 additional variations detailed in Figure 6. The case study systems cover a large part of the design space including client-server systems like Coda [19] and TRIP [28], server-replication systems like Bayou [29] and Chain Replication [40], and object replication systems like Pangaea [31] and TierStore [8].

The systems include a wide range of approaches for balancing consistency, availability, partition resilience, performance, reliability, and resource consumption, including demand caching and prefetching; coarse- and fine-grained invalidation subscriptions; structured and unstructured topologies; client-server, cooperative caching, and peer-to-peer replication; full and partial replication; and weak and strong consistency. The figure details the range of features we implement from the papers describing the original systems.

Except where noted in the figure all of the systems implement important features like well-defined consistency semantics, crash recovery, and support for both the object store interface and an NFS wrapper.

Rather than discussing each of these dozen systems individually [3], we highlight our overall conclusions:

1. *Padre is flexible.*

As Figure 6 indicates, we are able to construct systems with a wide range of architectures and features. Padre is aimed at environments where nodes are geographically distributed or mobile and where data placement affects performance or availability. Other environments such as machine rooms may prioritize different types of trade-offs and benefit from different approaches [1].

2. *Padre simplifies system building.*

As Figure 6 details, each system is described with 6 to 75 liveness rules and a few blocking predicates. As a result, once a designer knows how she wants a system to work (i.e., could describe the system in high-level terms like L1-L8 and S1-S4), implementing it is straightforward. Furthermore, the compactness of the code facilitates code review, and the separation of safety and liveness facilitates reasoning about correctness.

Many systems are considerably simpler than the client server example primarily because they require less stringent consistency semantics. Our Chain Replication and Pangaea implementations are more complex, at 75 rules each, due to the implementation of a membership service for Chain Replication and the richness of features in Pangaea.

3. *Padre facilitates the evolution of existing systems and the development of new ones.*

We illustrate Padre's support for rapid evolution by by adding new features to several systems. We add cooperative caching to the Padre version of Coda (P-Coda) in 4 lines; this addition allows a set of disconnected devices to share updates while retaining consistency. We add small-device support to P-Bayou in 1 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 4 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 4.5.

Overall, our experience supports our thesis that Padre facilitates the design of replication systems by capturing the right core abstractions for describing such systems.

We briefly describe our experience with these systems below.

4.3.1 P-Bayou

Bayou is a server replication system that uses anti-entropy to exchange updates between any pair of servers at any time. We implement a server replication system over Padre modeled on the version of Bayou described by Petersen et. al. [29]. In particular, we implement log-based peer-to-peer anti-entropy protocol, log truncation to limit state, checkpoint exchange in case of log truncation, primary commit, causal consistency, and eventual consistency.

Implementing safety and liveness policies. For safety, P-Bayou simply sets the *ReadNowBlock* predicate to *isValid* and *isComplete*. To provide 100% availability as the original Bayou does, P-Bayou sets the *ApplyUpdateBlock* predicate to *isValid* so as 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

[2] In this simple case, sequential consistency requires each read to return the value of the last write to complete before the start of the read or the value of any write to begin after the start but before the end of the read.

	Simple Client Server	Full Client Server	Bayou [29]	Bayou + Small Device	Chain Repl [40]	Coda [19]	Coda + Coop Cache	Pangaea [31]	Tier Store [8]	Tier Store +CC	TRIP [28]	TRIP +Hier
Liveness rules	21	43	9	9	75	31	35	75	16	20	6	6
Safety predicates	5	6	3	3	4	5	5	1	1	1	3	3
Consistency Consistency	Seq.	Seq.	Causal	Causal	Linear.	Open/ close	Open/ close	Coher.	Coher.	Causal	Seq.	Seq.
Topology	Client/ Server	Client/ Server	Ad- Hoc	Ad- Hoc	Chains	Client/ Server	Client/ Server	Ad- Hoc	Tree	Tree	Client/ Server	Tree
Partial replication	√	√		√		√	√	√	√	√		
Demand-only Caching	√	√										
Prefetching/Replication			√	√	√	√	√	√	√	√	√	√
Cooperative caching		√					√			√		
Disconnected operation			√	√		√	√		√	√		
Callbacks	√	√				√	√	√			√	
Leases		√				√	√					
Reads always satisfied locally			√	√	√				√	√		
Crash recovery	√	√	√	√	√	√	√	√	√	√	√	√
Object store interface*	√	√	√	√	√	√	√	√	√	√	√	√
File system interface*		√	√	√	√	√	√	√	√	√	√	√

Fig. 6: Features covered by case-study systems. *Note that the original implementations of some of these systems provide interfaces that differ from the object store or file system interfaces we provide in our prototypes.

```
// Add subscriptions when a random neighbor is selected
bc01 addInvalSubscription(@X, Y, X, SS, Catchup) :-
        doAntiEntropy(@X, Y), SS := "/*", Catchup:="LOG"

bc02 addBodySubscription(@X, Y, X, SS) :-
        doAntiEntropy(@X, Y), SS:= "/*"

// Remove subscriptions when we have received all updates
bc03 removeInvalSubscription(@X, Y, SS) :-
        informInvalSubscriptionCaughtup(@X, Y, SS)

bc04 removeBodySubscription(@X, Y, SS) :-
        informBodySubscriptionCaughtup(@X, Y, SS)
```

Algorithm 1: Anti-entropy rules in P-Bayou

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 9 rules: 4 for anti-entropy (Algorithm 1), 3 for random neighbor selection (not shown), and 2 to read list of servers from a configuration file. The *doAntiEntropy* tuple is generated periodically by the random neighbor selection rules which in turn invokes the anti-entropy rules (bc01, bc02).

P-Bayou + 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 the subscription set in (bc01) and (bc02) rules.

4.3.2 P-Coda

We implement P-Coda, a system inspired by the version of Coda described by Kistler et. al. [19]. P-Coda supports disconnected operation, reintegration, crash recovery, whole-file caching, open/close consistency (when connected), causal consistency (when disconnected), and hoarding. We know of one feature from this version 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 cache before others. Coda is long-running project with many papers worth of ideas. We omit features discussed in other papers like server replication [33], trickle reintegration [26], and variable granularity cache coherence [27]. We see no fundamental barriers to adding them in P-Coda. We also illustrate the ease with which co-operative caching can be added to P-Coda.

We provide a brief system description and discuss the safety and liveness policy of the client.

System Description P-Coda is a client-server system, similar to the example discussed earlier. The main differences between P-Coda and the client-server system are detailed below.

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 close on a client will block until all updates have been propagated to the server and the server has made sure that all copies cached on other clients have been invalidated. When a client is disconnected from the server, the client only accesses locally cached files that are valid.

Every client has a list of files, the "hoard set", that it will prefetch from the server and store it in its local cache whenever it is connected to the server.

P-Coda supports disconnected operation by weakening sequential consistency to causal consistency when a client is disconnected from the server: allowing writes to continue uncommitted and reads to access uncommitted but valid objects.

Implementing safety policy P-Coda uses the standard open/close consistency library. A file "open" is implemented as a read of an object. All writes to a file are buffered until the file is closed at which point, the object is written. The safety predicates

The *ReadNowBlock* predicate is set to *isValid* AND *isComplete* AND (*isCommited* OR *tuple notConnected*). Reads will read committed data when connected to the server, and casually-consistent objects when disconnected from the server. The *WriteEndBlock* predicate is set to *tuple notConnected* OR *tuple recvAckFromServer*. The safety policy will block a write until it receives a *recvAckFromServer* or a *notConnected* message from the liveness policy.

Implementing liveness policy We extend the liveness rules of the simple-client server model to implement P-Coda in 31 rules. First, like the full client-server example, we add 3 rules to check for blind writes when no callback is held and establish callbacks for them. Second, we add 4 rules to keep track of the server status, inform safety policy of the server status and unblock writes when disconnected from the server. Third, we implement hoarding in 3 rules by storing the hoard set as tuples in a configuration file and establishing invalidation and body subscriptions for each of them whenever the client connects to the server.

Adding cooperative caching We add co-operative caching to P-Coda so that disconnected clients can fetch valid files from their peer. This allows a disconnected client to access files it previously couldn't. We augment the node list configuration file to include a list of peers and add rules to fetch objects from nearby peers if the server is not reachable. In Figure 16, we show the performance improvement achieved by these four rules.

4.3.3 P-Pangaea

Pangaea [31] 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. P-Pangaea setst the *ReadNowBlock* predicate to *isValid* so that reads only access valid objects.

P-Pangaea considers harbingers as invalidations, and hence each edge of a Pangaea graph is an invalidation subscription. P-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 75 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.

4.3.4 P-Chain Replication

Chain Replication [40] 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 For safety, we set the *ReadNowBlock* predicate to *isValid* and *isComplete*. The *ApplyUpdatePredicate* is set to *isValid*. The *WriteAfterBlock* is set to *tuple ackFromTail* so that a write is blocked until an acknowledgement from the tail is received.

P-Chain-Replication implements each link in the chain as an 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 75 rules: 2 for update propagation, 2 for redirecting read requests to volume tail and write requests to volume head, 3 for generation of acknowlegements, 29 for chain management, 15 for failure recovery, 7 for initialization, and 17 for connection management at master.

4.3.5 P-TierStore

TierStore [8] is an object based hierarchical replication system for developing regions that provides eventual consistency and per-object coherence in the face of intermittent connectivity. It employs a "pub/sub" approach to distribute updates among nodes.

Implementing safety and liveness policies. P-TierStore sets the *ReadNowBlock* predicate to *isValid* to implement best-effort coherence and sets the *ApplyUpdateBlock* predicate to *isValid* to delay applying an invalidation to a node until the node has received the corresponding body.

The liveness policy maintains the tree-based topology with the help of configuration files, as in the original protocol. Every node has configuration files which specify its parent node, its children, and the "publication", i.e. data subtree, it is interested in. On initialization, these files are read as stored events and subscriptions are established. From a parent to child, invalidation and body subscriptions for each of the publications a child is interested in are established. From a child to parent, invalidation and body subscriptions for "/*" are established. The total number of liveness rules for reading configuration files and establishing connections is 16.

DTN support. Our current P-Tierstore implementation supports delay tolerant network (DTN) environments by allowing one or more mobile Padre nodes to relay information between a parent and a child in a distribution tree. In this configuration, whenever the relay node arrives, a node subscribes to receive any new updates the relay node brings and pushes all new local updates for the parent (or child) subscription to the relay node. The relay node can simply be a USB drive, in which case, a physical node runs two Padre instances, one that uses the local fixed storage and serves local requests and the other which uses the removable USB storage that acts as the relay.

An alternative approach would be to make use of existing DTN network protocols. This approach is straightforward to implement if the DTN layer informs the policy layer when it has an opportunity to send to another node and when that opportunity ends. An opportunity could be that a TCP connection opens up or a USB drive was inserted. The liveness policy would establish subscriptions to send updates within that connection opportunity as a DTN bundle.

Extending P-TierStore. Just like P-Coda, cooperative caching is easily added to P-TierStore by adding 4 rules. This addition enables users in a developing region to retrieve data using local wireless links from nearby peers who have already downloaded data across an expensive modem link.

Primitive	Best Case	Padre Prototype
Start conn.	0	$N_{nodes} * (\hat{S}_{id} + \hat{S}_t)$
Inval sub w/ LOG catchup	$(N_{prev} + N_{new}) * S_{inval}$ $+S_{sub}$	$(N_{prev} + N_{new}) * \hat{S}_{inval}$ $+N_{impr} * \hat{S}_{impr} + \hat{S}_{sub}$
Inval sub w/ CP catchup	$(N_{modO} + N_{new}) * S_{inval}$ $+S_{sub}$	$(N_{modO} + N_{new}) * \hat{S}_{inval}$ $+N_{impr} * \hat{S}_{impr} + \hat{S}_{sub}$
Body sub	$(N_{modO} + N_{new}) * S_{body}$	$(N_{modO} + N_{new}) * \hat{S}_{body}$
Single body	S_{body}	$\hat{S}_{body}$

Fig. 7: Network overheads of primitives. Here, N_{nodes} is the number of nodes; N_{prev} and N_{modO} are the number of updates and the number of updated objects from a subscription start time to the current logical time; N_{new} is the number of updates sent on a subscription after it has caught up to the sender's logical time until it ends; and N_{impr} is the number of imprecise invalidations sent on a subscription. S_{id}, S_t, S_{inval}, S_{impr}, S_{sub} and S_{body} are the sizes to encode a node ID, logical timestamp, invalidation, imprecise invalidation, subscription setup, or body message; S_x are the sizes of ideal encodings and $\hat{S}_x$ are the sizes realized in the prototype.

4.3.6 P-TRIP

TRIP [28] seeks to provide transparent replication for web edge servers of dynamic content. All nodes enforce sequential consistency and a limit on staleness.

Implementing safety and liveness policy. P-TRIP set the *readNowBlock* predicate to *isValid* and *isComplete* and set a limit for *maxStaleness*. The *ApplyUpdateBlock* predicate is to *isValid*. Note that since there is a single writer, these predicates are sufficient to enforce sequential consistency. The liveness rules are simple: clients subscribe to receive all invalidations and bodies from the server. The complete implementation of P-TRIP's liveness policy requires 6 rules.

Extending P-TRIP. What is perhaps most interesting about this example 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 [24], a few additional rules to subscribe/unsubscribe produce a dynamic-tree version of TRIP that still enforces sequential consistency.

4.4 Performance

Our primary performance goal is to minimize network overheads. We focus on network costs for two reasons.

First, we want Padre systems to be useful for network-limited environments. Second, if network costs are close to the ideal, it would be evidence that Padre captures the right abstractions for constructing replication systems.

4.4.1 Network efficiency

Figure 4.3.6 shows the cost model of our implementation of Padre's primitives and compares these costs to the costs of best-case implementations. Note that these best-case implementation costs are optimistic and may not always be achievable.

Two things should be noted. First, the best case costs of the primitives are proportional to the useful information sent, so they capture the idea that a designer should be able to send just the right data to just the right place. Second, the overhead of our implementation over the ideal is generally small.

In particular, there are three ways in which our prototype may send more information than a hand-crafted implementation of some systems.

First, Padre metadata subscriptions are multiplexed onto a single network connection per pair of communicating nodes, and establishment of such a connection requires transmission of a version vector [46]. Note that in our prototype this cost is amortized across all of the subscriptions and invalidations multiplexed on a network connection. A best-case implementation might avoid or reduce this communication, so we assume a best-case cost of 0.

Our use of connections allows us to avoid sending per-update version vectors or storing per-object version vectors. Instead, each invalidation and stored object includes an *acceptStamp* [29] comprising a 64-bit nodeID and a 64-bit Lamport clock.

Second, invalidation subscriptions carry both *precise invalidations* that indicate the logical time of each update of an object targeted by a subscription and *imprecise invalidations* that summarize updates to other objects [2]. The number of imprecise invalidations sent is never more than the number of precise invalidations sent (at worst, the system alternates between the two), and it can be much less if writes arrive in bursts with locality [2]. The size of an imprecise invalidation depends on the locality of the workload, which determines the extent to which the target set for imprecise invalidations can be compactly encoded. A best-case implementation might avoid sending imprecise invalidations in some systems, so we assume a best-case invalidation subscription cost of only sending precise invalidations.

Third, our Java-serialization of specific messages may fall short of the ideal encodings.

Figure 8 illustrates the synchronization cost for a simple scenario. In this experiment, there are 10,000 objects in the system organized into 10 groups of 1,000 objects each, and each object's size is 10KB. The reader registers to receive invalidations for one of these groups. Then, the writer updates 100 of the objects in each group. Finally, the reader reads all of the objects.

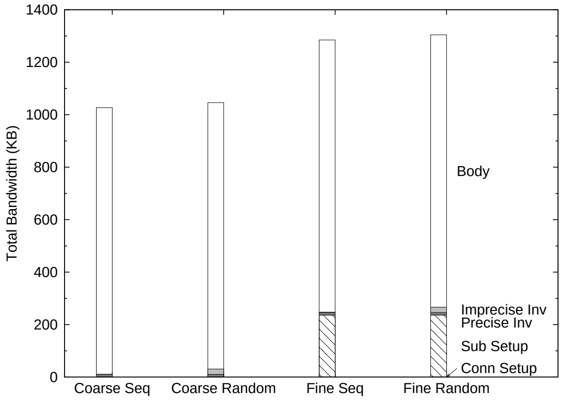

Fig. 8: Network bandwidth cost to synchronize 1000 10KB files, 100 of which are modified.

We look at four scenarios representing combinations of coarse-grained vs. fine-grained synchronization and of writes with locality vs. random writes. For coarse-grained synchronization, the reader creates a single invalidation subscription and a single body subscription spanning all 1000 objects in the group of interest and receives 100 updated objects. For fine-grained synchronization, the reader creates 1000 invalidation subscriptions, each for one object, and fetches each of the 100 updated bodies. For writes with locality, the writer updates 100 objects in the ith group before updating any in the $i+1$st group. For random writes, the writer intermixes writes to different groups in random order.

Four things should be noted. First, the synchronization overheads are small compared to the body data transferred. Second, the "extra" overhead of Padre over the best-case is a small fraction of the total overhead in all cases. Third, when writes have locality, the overhead of imprecise invalidations falls further because larger numbers of precise invalidations are combined into each imprecise invalidation. Fourth, coarse-grained synchronization has lower overhead than fine-grained synchronization because they avoid per-object setup costs. In particular, for this example, setting up a single-object callback requires transmission of 333 bytes, so setting up 1000 callbacks costs 333,000 bytes, and setting up a single subscription of all 1000 objects in a group costs 97,236 bytes.

More generally, our implementation efficiently implements both fine-grained and coarse-grained subscriptions. 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

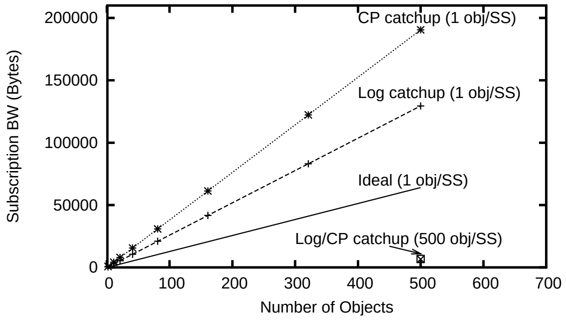

Fig. 9: Bandwidth for establishing invalidation subscriptions.

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 [16].

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.

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 compare the overheads under three workloads. Under the first workload (1-in-1), all updates are made to objects within the subscription set of the invalidation stream. Under the second workload (1-in-10), for every 10 updates made, one of them is made to an object within the subscription set. Under the (1-in-2) workload, for every update made to an object that lies in the subscription set, another is made an object that lies out of the subscription set. Fig. 10, even in the worst case, the overhead for maintaining consistency is at most 2x the number of invalidations sent over the subscription.

	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.

	Write (sync)	Write (async)	Read (cold)	Read (warm)
ext3	6.64	0.02	0.04	0.02
BerkeleyDB	8.01	0.06	0.06	0.01
Local NFS	8.61	0.14	0.10	0.05
Padre object store	8.47	1.27	0.25	0.16

Fig. 11: Read/write performance for 1KB objects/files in ms.

	Write (sync)	Write (async)	Read (cold)	Read (warm)
ext3	19.08	0.13	0.20	0.19
BerkeleyDB	14.43	4.08	0.77	0.18
Local NFS	21.21	1.37	0.26	0.22
Padre object store	52.43	43.08	0.90	0.35

Fig. 12: Read/write performance for 100KB objects/files in ms.

	P2 Runtime	R/OverLog runtime
Local Ping Latency	3.8ms	0.322ms
Local Ping Throughput	232 req/s	9,390 req/s
Remote Ping Latency	4.8ms	1.616ms
Remote Ping Throughput	32 req/s	2,079 req/s

Fig. 13: Performance numbers for processing NULL trigger to produce NULL event.

4.4.2 Performance overheads

This section examines the performance of the Padre prototype. Our goal is to provide sufficient performance for the system to be useful, but we expect to pay some overheads relative to a local file system for three reasons. First, Padre is a relatively untuned prototype rather than well-tuned production code. Second,our implementation emphasizes portability and simplicity, so Padre is written in Java and stores data using BerkeleyDB rather than running on bare metal. Third, Padre provides additional functionality such as tracking consistency metadata not required by a local file system.

Figures 11 and 12 summarize the performance for reading or writing 1KB or 100KB objects stored locally in Padre compared to the performance to read or write a file on the local ext3 file system. In each run, we read/write 100 randomly selected objects/files from a collection of 10,000 objects/files. The values reported are averages of 5 runs. Overheads are significant, but the prototype still provides sufficient performance for a wide range of systems.

Padre performance is signficantly affected by the runtime system for executing liveness rules. Our compiler converts R/OverLog programs to Java, giving us a signif-

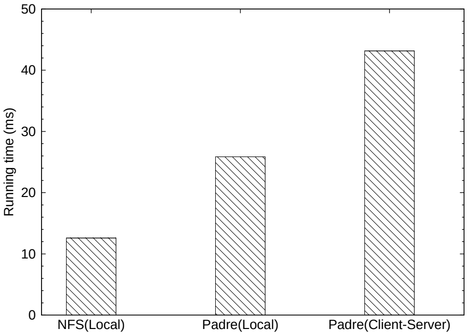

Fig. 14: Performance for Andrew benchmark.

	Reads			Writes		
	Min	Max	Avg	Min	Max	Avg
Full CS	0.18	9.34	1.07	19.69	42.03	26.28
P-TierStore	0.16	0.74	0.19	6.65	22.40	7.68

Fig. 15: Read/Write performance in milliseconds for reading 1KB objects under the same workload as Figure 4.

icant performance boost compared to an earlier version of our system, which used P2 [24] to execute OverLog programs. Figure 13 quantifies these overheads.

Fig. 14 depicts the time required to run the Andrew benchmark [16] over the Padre prototype via the Java NFS wrapper. Padre successfully runs the benchmark, but it is slower than a well-tuned local file system.

Figure 15 depicts read and write performance for the full client-server system and P-TierStore under the same workload as in Figure 4 but without any additional network latency. For the client-server system, read hits take under 0.2ms. Read misses average 4ms, yielding an average read time of 1ms. Writes need to wait until the write is stored by the server, the reader is invalidated, and a server ack is received and average 26ms. For P-TierStore, because of weaker consistency semantics, all reads and writes are locally satisfied.

4.5 Benefits of agility

As discussed in Section 1, replication system designs make fundamental trade-offs among consistency, availability, partition-resilience, performance, reliability, and resource consumption, and new environments and workloads can demand new trade-offs. As a result, being able to architect a replication system to send the right data along the right paths can pay big dividends.

This section measures improvements resulting from adding features to two of our case-study systems; due to space constraints, we omit discussion of similar experiences with two others (see Figure 6.) In both cases, we adapt the system to a new environment and gain order-of-magnitude improvements by making what are—because of Padre—trivial additions to existing designs.

Figure 16 demonstrates the significant improvement by adding 4 rules for cooperative caching to P-Coda. Cooperative caching allows a clique of connected devices to share data without relying on the server. For the experiment, 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 the server. However, with cooperative caching, the client can retrieve data from a nearby client, greatly improving read performance. More importantly, with this new capability, clients can share data even when disconnected from the server.

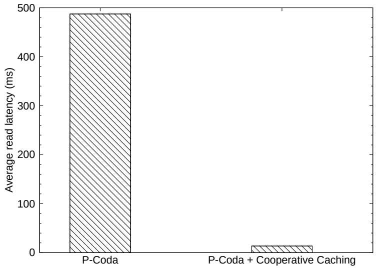

Fig. 16: Average read latency of P-Coda and P-Coda with cooperative caching.

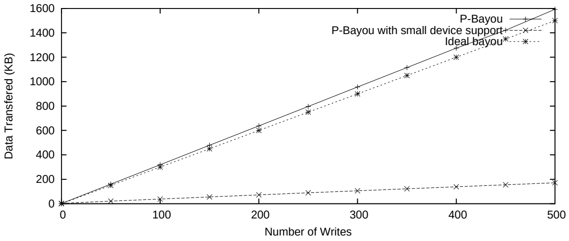

Fig. 17: Anti-Entropy bandwidth on P-Bayou

Figure 17 examines the bandwidth consumed to synchronize 3KB files in P-Bayou and serves two purposes. First, it demonstrates that the overhead for anti-entropy in P-Bayou is relatively small even for small files compared to an “ideal” Bayou implementation (plotted by counting the bytes of data that must be sent ignoring all overheads.) More importantly, it demonstrates that if a node requires only a fraction (e.g., 10%) of the data, the *small device enhancement*, which allows a node to synchronize a subset of data [3], greatly reduces the bandwidth required for anti-entropy.

5 Related work

PRACTI [2, 46] defines a set of *mechanisms* that can reduce replication costs by simultaneously supporting Partial Replication, Any 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 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. 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 actions, triggers, and stored events upon which liveness rules operate. This paper demonstrates how this approach facilitates construction of a wide range of systems.

A number of other efforts have defined frameworks for constructing replication systems for different environments. Deceit [35] focuses on replication across a well-connected cluster of servers. Zhang et. al. [45] 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 [13] 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 [44]; we do not currently support numeric error. Like Padre, Swarm [38] 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 [21], sequential [22], 1SR [4], or linearizability [14]. IceCube [18] and actions/constraints [34] provide frameworks for specifying general consistency constraints and scheduling reconciliation to minimize conflicts. Fluid replication [6] provides a menu of consistency policies, but it is restricted to hierarchical caching.

Padre follows in the footsteps of efforts to define runtime systems or domain-specific languages to ease the construction of routing [24], overlay [30], cache consistency protocols [5], and routers [20].

6 Conclusion

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 systems on Padre confirmed the benefits of Padre: Padre is flexible and can be used to construct a broad range of systems. All of our 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, Padre has near ideal network traffic and acceptable absolute performance for a wide range of workloads.

References

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamonlis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

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

[3] N. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. PADRE: A Policy Architecture for building Data REplication systems (extended). Technical Report TR-08-25, U. of Texas Dept. of Computer Sciences, May 2008.

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

[5] S. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a Language for Writing Coherence Protocols. In *USENIX Conf. on Domain-Specific Lang.*, Oct. 1997.

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

[7] 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.

[8] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed storage system for challenged networks. In *Proc. FAST*, Feb. 2008.

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

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

[11] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *SOSP*, pages 202–210, 1989.

[12] R. Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pages 38–51, June 2006.

[13] J. Heidemann and G. Popek. File-system development with stackable layers. *ACM TOCS*, 12(1):58–89, Feb. 1994.

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

[15] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proc. OOPSLA*, Oct. 2007.

[16] 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.

[17] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *SOSP*, Dec. 1995.

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

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

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

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

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

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

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

[25] D. Mazières. A toolkit for user-level file systems. In *USENIX Technical Conf.*, 2001.

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

[27] L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *USENIX Summer*

Conf., June 1994.

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

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

[30] 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.

[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] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *USENIX Summer Conf.*, pages 119–130, June 1985.

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

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

[35] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. Corenell TR 89-1042, 1989.

[36] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Proc NSDI*, 2008.

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

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

[39] 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.

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

[41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, Dec. 2002.

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

[43] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, Feb. 1999.

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

[45] 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.

[46] J. Zheng, N. Belaramani, M. Dahlin, and A. Nayate. A universal protocol for efficient synchronization. `http://www.cs.utexas.edu/users/zjiandan/papers/upes08.pdf`, Jan 2008.

A Liveness API and full example

Figure 18 lists all of the actions, triggers, and stored events that Padre designers use to construct liveness policies that route updates among nodes.

The following 21 liveness rules describe the full liveness policy for the simple client-server example.

```
// Read miss: client fetch from server
L1a: clientRead(@S, C, Obj, Off, Len) :-
       TRIG_informReadBlock(@C, Obj, Off, Len, _),
       TBL_serverId(@C, S), C ≠ S.
L1b: ACT_sendBody(@S, S, C, Obj, Off, Len) :-
       clientRead(@S, C, Obj, Off, Len).
// ReadMiss: establish callback
L2a: ACT_addInvalSubscription(@S, S, C, Obj, Catchup) :-
       clientRead(@S, C, Obj, Off, Len), Catchup := “CP”.
```

Liveness Actions	
Add Inval Sub	srcId, destId, objs, [time], LOG|CP|CP+Body
Remove Inval Sub	srcId, destId, objs
Add Body Sub	srcId, destId, objs
Remove Body Sub	srcId, destId, objs
Send Body	srcId, destId, objId, off, len, writerId, logTime
Assign Seq	objId, off, len , writerId, logTime

Connection Triggers	
Inval subscription start	srcId, destId, objs
Inval subscription caught-up	srcId, destId, objs
Inval subscription end	srcId, destId, objs, reason
Body subscription start	srcId, objs, destId
Body subscription end	srcId, destId, objs, reason

Local read/write Triggers	
Read block	obj, off, len, EXIST|VALID|COMPLETE|COMMIT
Write	obj, off, len, writerId, logTime
Delete	obj, writerId, logTime

Message arrival Triggers	
Inval arrives	sender, obj, off, len, writerId, logTime
Fetch success	sender, obj, off, len, writerId, logTime
Fetch failed	sender, receiver, obj, offset, len, writerId, logTime

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

Fig. 18: Padre interfaces for liveness policies.

```
L2b: TBL_hasCallback(@S, Obj, C) :-
       clientRead(@S, C, Obj, Off, Len).
//Maintain c-to-s subscriptions for updates.
L3a: ACT_addInvalSubscription(@S, C, S, SS, Catchup) :-
       clientCFGTuple(@S, C), SS:=“/*”, Catchup := “CPwithBody”,
       TBL_serverId(@S, S).
L3b: ACT_addBodySubscription(@S, C, SS) :-
       clientCFGTuple(@S, C), SS:=“/*”, TBL_serverId(@S, S).
L3c: ACT_addInvalSubscription(@S, C, SS, Catchup) :-
       TRIG_informInvalSubscriptionEnd(@S, C, S, SS, _),
       Catchup := “CPwithBody”.
L3d: ACT_addBodySubscription(@S, C, SS) :-
       TRIG_informBodySubscriptionEnd(@S, C, S, SS, _).
// No rules needed for L4 (see L2)
// When client receives an invalidation: ACK server, cancel callback
L5a: ackServer(@S, C, Obj, Off, Len, Writer, Stamp) :-
       TRIG_informInvalArrives(@C, S, Obj, Off, Len, Stamp, Writer),
       TBL_serverId(@C, S), S≠C.
L5b: removeInvalSubscription(@C, S, C, Obj) :-
       TRIG_informInvalArrives(@C, S, Obj, Off, Len, Stamp, Writer),
       TBL_serverId(@C, S), S≠C.
// Server receives inval: gather acks from all who have callback
// Acks are cumulative. Ack of timestamp i acks all earlier
L6a: TBL_needAck(@S, Ob, Off, Ln, C2, Wrtr, Stmp, Need) :-
       TRIG_informInvalArrives(@S, C, Ob, Off, Ln, Stmp, Wrtr),
       TBL_hasCallback(@S, Ob, C2), C2 ≠ Wrtr, Need := 1,
       TBL_serverId(@S, S).
L6b: TBL_needAck(@S, Ob, Off, Ln, C2, Wrtr, Stmp, Need) :-
       TRIG_informInvalArrives(@S, C, Ob, Off, Ln, Stmp, Wrtr),
       C2 == Wrtr, Need := 0, TBL_serverId(@S, S).
L6c: TBL_needAck(@S, Ob, Off, Ln, C, Wrtr, Stmp, Need) :-
       ackServer(@S, C, _, _, _, _, RStmp),
       TBL_needAck(@S, Ob, Off, Ln, C, Wrtr, Stmp, _),
       Stmp < RStmp, Need:= 0, TBL_serverId(@S, S).
```

```
L6d:  TBL_needAck(@S, Obj, C, Wrtr, Stmp, Need)  :-
        ackServer(@S, C, _, _, _, RecvWrtr, RStmp),
        TBL_needAck(@S, Obj, C, Wrtr, Stmp, _),  Stmp == RStmp,
        Wrtr ≤ RecvWrtr,  Need:= 0,  TBL_serverId(@S, S).
L6e:  delete TBL_hasCallback(@S, Obj, C)  :-
        TBL_needAck@S, Obj, _, _, C, _, _, Need),  Need == 0,
        TBL_serverId(@S, S).
L6f:  acksNeeded(@S, Ob, Off, Ln, Wrtr, RStmp, <count>)  :-
        TBL_needAck(@S, Ob, Off, Ln, C, Wrtr, RStmp, NeedTrig),
        TBL_needAck(@S, Ob, Off, Ln, C, Wrtr, RStmp, NeedCount),
        NeedTrig == 0,  NeedCount == 1.
L6g:  writeComplete(@Wrtr, Obj)  :-
        acksNeeded(@S, Obj, Off, Len, Wrtr, RStmp, Count),
        Count == 0.
L6h:  delete TBL_needAck(@S, Obj, C, WrtrId, RStmp, _)  :-
        acksNeeded(@S, Obj, Off, Len, Wrtr, RStmp, Count),
        Count == 0.
// Startup: produce configuration stored event tuples
L7a:  SE_readTuples(@X, Obj)  :-
        init(@X),  Obj := "clientCFG".
L7b:  SE_readTuples(@X, Obj)  :-
        init(@X),  Obj := "serverCFG".
// When all acks received: Assign a CSN to the update
L8:  ACT_assignSeq(Obj, Off, Len, Stmp, Wrtr)  :-
        acksNeeded(@S, Obj, Off, Len, Wrtr, Stmp, Count),  Count == 0.
```

For safety, this system sets the *ApplyUpdateBlock* predicate at the server to *isValid* and sets the *readNowBlock* predicate to *isValid* AND *isComplete* AND *isSequenced CSN*. Additionally, it sets the *writeEndBlock* predicate to *tuple writeComplete objId* with a timeout of 20 seconds. Clients retry the write if it times out; the retry ensures completion if there is a server failure and recovery.

B Planned Extension

In order to provide greater flexibility for defining liveness policies, we plan to expose the persistent store's consistency bookkeeping information as *live tables* in R/OverLog. This information would allow specification of policies which rely on state information. For example, a policy can establish subscriptions only if the sender has new updates by comparing the sender's logical version vector (*LVV*) to the local *LVV*. In principle, a R/OverLog policy could maintain this state itself, but exposing consistency state already maintained by the runtime system may simplify some policies. Figure 19 summarizes the information we plan to make available.

Live Tables Export	
LVV nodeId logTime	Logical version vector: the logical time of the most recent event received from each node
RVV nodeId time	Real time version vector: the real timestamp attached to the newest event received from each node
VALID objId TRUE|FALSE	True if the local object store stores for *objId* an update body whose logical timestamp matches that of the most recent invalidation seen for *objId*
CAUSAL objId TRUE|FALSE	True if the local object store has for *objId* the most recent invalidation on which any other update in the object store depends
COMMITTED objId TRUE|FALSE	True if the local object store has for *objId* a commit record that references the most recent invalidation received for *objId*

Fig. 19: Live tables export bookkeeping state to liveness policies.