Making Replication Simple with Ursa

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

Abstract: Our goal is to qualitatively simplify developing large scale data replication systems and improving existing ones. To realize this goal, we address a fundamental question: What are the right abstractions for defining replication systems? Our new architecture, Ursa, defines (1) mechanisms that define abstractions for storage, communication, and consistency that automatically handle the bookkeeping needed to allow policies to distribute data however they want, (2) a model of policy that cleanly separates safety and liveness concerns, (3) a way to define safety policies leveraging standard consistency algorithms, and (4) a way to define liveness policies using a declarative language. By capturing the right abstractions, Ursa dramatically reduces the effort needed to construct a new system, and it facilitates innovation by making it easy to evolve existing systems. For example, we build systems that closely approximate Coda, Bayou, Pangaea, Chain Replication, TRIP, and TierStore using fewer than 200 lines of system-specific policy code for each, and we add significant new features like cooperative caching to Coda and TierStore, small-device support for Bayou, or hierarchy to TRIP using fewer than 10 additional lines.

1 Introduction

As technologies and workloads evolve, new replication systems will continue to be needed. In particular, managing distributed state is a fundamental challenge for a broad range of systems such as personal multimedia [29], services for developing regions [8], web edge services [9], and mobile applications [16, 18]. However, fundamental trade-offs between consistency, availability, and performance [10, 22] mean that new environments often demand new systems that make new trade-offs among these factors.

To address these needs, this paper presents Ursa—a universal replication service architecture over which a broad range of replication systems can be constructed with dramatically less effort than current approaches.

To realize this goal, Ursa addresses a fundamental question: What are the right abstractions for defining replication systems? Ursa's architecture flows from four key ideas.

- First, Ursa provides mechanisms that define abstractions for storage, communication, and consistency that automatically handle the low-level bookkeeping needed to allow policies to distribute data however they want.
- Second, Ursa models policy in a way that cleanly splits the decisions that distinguish one system from another into *safety constraints* and *liveness strategies*. In essence, a safety policy defines when a read or write must block to enforce the system's semantics, while a liveness policy defines how data must flow through the system to minimize blocking of reads and writes.
- Third, we observe that safety policy maps to the problem of enforcing consistency constraints. Systems can therefore choose from a set of standard consistency libraries to select a safety policy.
- Fourth, we observe that liveness policy in a replication system defines a system's topology and replication policy, and it can be cast as an overlay routing problem: to which nodes should updates to each object flow and from which nodes should requests for each object be fetched? Given this insight, Ursa employs a concise declarative language to express sophisticated topology and replication policies.

By capturing the right abstractions, Ursa simplifies development of replication systems. To test this claim, we first build a series of replication systems inspired by systems from the literature spanning a significant portion of the design space. These case studies include two client-server systems modeled on Coda [18] and TRIP [28], two server replication systems modeled on Bayou [30] and Chain Replication [38], and two object replication systems modeled on Pangaea [32] and TierStore [8]. We demonstrate that each of these systems approximates the corresponding published system in that its behavior is within a modest constant factor for a well-defined set of key properties including network bandwidth, storage, consistency, latency, and availability. Each system was remarkably easy to build, requiring fewer than 200 lines of system-specific policy code.

We also demonstrate how Ursa facilitates rapid evolution by adding significant features to several of these systems. For example, we add cooperative caching to Coda so that a clique of devices can share data even when disconnected from the server; we add support for small devices to Bayou so that a limited-storage device can participate in Bayou replication without storing all of the system's data; and we add cooperative caching to TierStore so that once one user in a developing region downloads data across an expensive modem link, nearby users can retrieve that data using their local wireless network; and we add hierarchy to TRIP to improve scalability. Each of these features yields significant advantages,

yet each enhancement required less than 10 lines of additional system-specific policy code.

Overall, our experience suggests that Ursa can dramatically reduce the effort needed to construct a replication system. More specifically,

1. *Ursa is flexible* in that we are able to implement a broad range of systems.
2. *Ursa is pithy* in that the Ursa implementation of each of these systems comprised fewer than 200 lines of system-specific policy code.
3. *Ursa is efficient* in that we are able to build systems that are comparable to hand-built systems from the literature with respect to the central properties of a replication architecture; and
4. *Ursa facilitates innovation* by making it easy to add or change features in systems.

These results suggest that Ursa captures key abstractions at the core of a significant range of replication architectures.

In the rest of this paper, we first discuss thorny issue of comparing Ursa-based systems to systems from the literature (§2.) Then, §3 and §4 present the abstractions and mechanisms at the core of Ursa, and §5 describes how replication systems can be defined by specifying their policy. We discuss implementation decisions in §6. Our evaluation of Ursa is based on a series of case studies presented in §7. §8 contrasts Ursa with related work. Finally, §9 concludes by discussing our experience with Ursa.

2 Equivalence and scope

The reader should be a bit concerned at this point. We claim that Ursa simplifies the task of developing replication systems, but how can such a claim be judged? Our evaluation compares prototype systems to systems from the literature, but constructing perfect, "bug compatible" duplicates of such systems on Ursa is probably not a realistic (or useful) goal. On the other hand, if we are free to pick and choose arbitrary subsets of features to exclude, then the bar for evaluating our framework is too low: we can claim to have built any system by simply excluding any features our architecture has difficulty supporting.

This issue reflects a deeper challenge to designing a replication architecture: we must identify the essential characteristics of replication systems the architecture should encompass. Published replication systems have many features; some are fundamental to their design and some are peripheral. However, to be useful, an architecture must *restrict* the choices of a designer: an architecture that allows every possible variation of every possible design decision is not an architecture at all.

In this section, we therefore define a working *equivalence* relationship between replication systems that defines both the scope of and requirements on Ursa. This section closes with a discussion of several issues that Ursa does not attempt to address.

2.1 Equivalence

We define equivalence in terms of three properties:

E1. *Equivalent overhead.* System *A*'s cumulative network bandwidth between any pair of nodes and local storage at any node are within a constant factor of system *B*'s.

E2. *Equivalent consistency.* System *A* provides consistency and staleness properties that are at least as strong as system *B*'s.

E3. *Equivalent local data.* The set of data that may be accessed from system *A*'s local state without network communication is a super-set of the set of data that may be accessed from system *B*'s local state.

Notice that property E3 encompasses several factors including latency, availability, and durability.

There is a principled reason for believing that these properties capture something about the essence of a replication system: they highlight how a system resolves the fundamental CAP (Consistency v. Availability v. Partition-resilience) [10] and PC (Performance v. Consistency) [22] trade-offs that any replication system must make. More specifically, omitting any of these properties could allow a system to significantly cut corners. For example, one can improve read performance by increasing network and storage resource consumption to speculatively replicate more data to each node and weakening consistency by delaying invalidations until the corresponding body has been prefetched [28]. Similarly, one can improve the availability a system offers for a given level of consistency by using more network bandwidth to synchronize more often [39], or one can reduce the resources consumed by replication by delaying propagation of updates and weakening consistency [2] or by reducing the amount of data cached at a node.

We define different levels of equivalence specifying when E1–E3 must hold.

Definition. System *A* is *S-equivalent* (strongly equivalent) to system *B* if at any time for any workload E1, E2, and E3 hold.

Unfortunately, though appealing, the S-equivalence relation is too strong in practice—it can exclude systems that are "equivalent enough." For example, if two systems (or even two runs of the same system) make different non-deterministic choices about the order of two concurrent writes to an object, different nodes could end up with a copy of the object, making the system fail the third test. We therefore define a useful, weaker form of equivalence.

Definition. System *A* is *Q-equivalent* (quiescent equivalent) to system *B* if for a Q-workload consisting of a series of requests with a quiescent period after request i completes execution before request $i+1$ begins execution, properties E1 and E3 hold at the start of each quiescent period and property E2 holds for all requests.

Although the workload defined in Q-equivalence is unrealistic, it makes comparison of systems tractable by removing the non-determinism concurrency can introduce. Furthermore, we believe that if a system can meet the Q-equivalent requirements, it is likely the system will be "equivalent enough" for most realistic workloads.

2.2 Excluded properties

These definitions restrict the scope of our architecture in a well-defined way. Several excluded properties warrant discussion: security, interface, conflict resolution, and configuration.

First, we do not address security. We believe that ultimately our replication architecture should also define flexible security mechanisms and make specifying a system's security a policy choice. Providing this ability 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, consistency, and topology policies [3] to address the CAP [10] and PC [22] trade-offs.

Second, we do not systematically address the local interface a system exposes (e.g., file system [18, 28, 32], object store [8], tuple store [30], etc.) because we do not regard these differences as fundamental. Ursa currently implements a object store and we have constructed several file systems over it; future work is needed to extend it to support tuple-stores.

Third, we do not attempt to support all possible conflict resolution algorithms [8, 17, 18, 34, 37]. Ursa logs all write-write conflicts in a way that is data-preserving and consistent across nodes to support a broad range of application-level resolvers. We believe it is possible to extend our mechanisms to support Bayou's more flexible application-specified conflict detection and reconciliation programs, but supporting this additional flexibility would increase the cost of applying updates to a node's storage because it would requires a node's state to be rolled back to the logical time of an update in order to run the conflict detection and resolution programs in an appropriate context [37].

Finally, we do not attempt to duplicate how systems are configured (e.g., specifying lists of peers or replication policy with configuration files [18] or symbolic links [8, 25]). We rely on some configuration files and provide hooks for our liveness policies to access the object store, but we do not claim that our arrangement is optimal.

3 Architecture

Ursa's architecture defines abstractions that represent simple building blocks with which a system designer can construct a wide range of replication systems. As Figure 1 illustrates, a system designer views a replication system as having three parts. First, a set of core

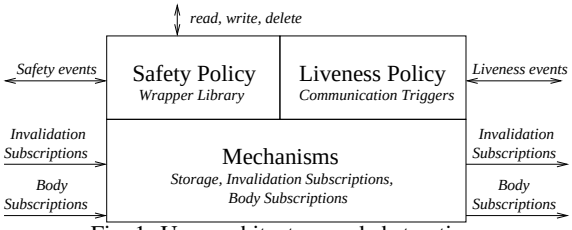

Fig. 1: Ursa architecture and abstractions.

mechanisms that capture the fundamental abstractions for state management, consistency, and communication of updates. Second, a *safety policy* embodied in a wrapper library that enforces the system's consistency constraints. Third, a *liveness policy* embodied in a set of communication triggers embody the system's replication strategy and node topology for distributing updates; it is such liveness policies that largely define a replication system's architecture, distinguishing server replication systems like Bayou [30] and Chain Replication [38] from client-server caching systems like Coda [18] from cooperative caching systems like Shark [1] from coarse-grained push-subscription systems like TierStore [8].

For example, to implement an AFS-like client-server system [15], a designer would instantiate a wrapper library that enforces open/close consistency on reads and writes, and she would construct a liveness policy in which (1) each client sends all updates to the server, (2) each client fetches demand read misses from the server, and (3) the server sends invalidations to all clients caching a file when the file is updated. The mechanisms would provide the necessary storage, would handle the bookeeping to ensure that updates flowing into a node are applied in a well-defined order, and would expose the resulting consistency state to the consistency policy.

§4 and §5 describe the abstractions provided by the mechanisms and policies.

4 Mechanisms

In this section, we describe the basic abstractions for managing storage, communication of invalidations, and communication of update bodies. Although these mechanisms are simple, the specific features they expose make it easy to construct a broad range of systems by instantiating appropriate policies over them. The goal of this section is to define the view of these mechanisms Ursa exposes to policy writers; we defer discussion of how to implement these abstractions efficiently to §6.

4.1 Storage

The storage abstraction presented to the policy layer is simple and has two main parts: (1) per-object state for a subset of the system's data selected by the policy and (2) basic consistency state that tracks and exposes the dependencies among updates.

Per-object and interest set state. In order to support replication and caching, each node has an object store that contains a subset of the system's data. Every object has an object ID and byte-addressable data. An *interest set* identifies an individual object or a group of objects (e.g., /a/b/*) and a policy can specify and update a list of interest sets for which a node should store per-object state. This per-object state includes a logical timestamp and real timestamp for the last known update for each byte-range and may include the data stored in each byte-range (if that data has been received by the node.) A node need not store per-object state for objects outside of the interest sets specified by the policy. A node also tracks per-IS-state that contains the logical timestamp of the latest known update for any object in the interest set, for poicy- specified interest sets.

Consistency state. A node tracks basic consistency state for the system's objects. For each interest set, the node consultes the per-IS-state to determine whether objects in the interest set are *presumed inconsistent*—the node has missed one or more invalidations that may affect the state of objects in the interest set. Additionally, for objects in interest sets for which the node is tracking per-object state, the node tracks whether each byte-range is *consistent* (the node has a version of the byte-range that is causally consistent [20] with all local reads and writes it has processed and all updates it has received from other nodes), *inconsistent* (the node has processed an invalidation that is newer than the data it is storing), or *not present* (the node does not have a copy of the byte-range.) For interest sets that are *presumed inconsistent*, a node tracks which ranges of update events it missed so that it can identify what information is needed to make the per-object consistency state current for that interest set.

Additionally, to support TACT's [39] tunable temporal error (TE) and order error (OE), each node maintains a real-time vector clock that is updated when invalidations or heartbeats are received from other nodes, and each node tracks the commit state of its updates using Golding's algorithm [11].

4.2 Invalidation subscriptions

Invalidation subscriptions allow nodes to share information about subsets of data while maintaining consistency. Each invalidation subscription delivers a causally consistent sequence of the updates for a subset of the system's data and also includes explicit records summarizing which updates have been omitted from the complete causally consistent stream. These explicit omissions allow a receiver to track which subsets of data it has received complete consistency information for and which subsets of data must be *presumed inconsistent*.

More specifically, the invalidations to be sent in a subscription are characterized by two parameters: a start time *startVV* (a vector clock) and a subscription set *SS* (a collection of objects) that together define a request: "Send all invalidations to *SS* that have occurred since *startVV*."

Log and checkpoint transfers. Policies can optimize the cost of an invalidation subscription by selecting either of two semantically-equivalent encodings: *log-iteration* and *checkpoint-and-log*.

In the *log-iteration* realization of a subscription, the sender iterates across a causally-sorted list of all invalidations it has received after time *startVV*, sending any invalidations that target objects in *SS* and combining other invalidations into concise *imprecise invalidations* [3] that specify a target set of objects, a start time, and an end time: "One or more objects in *target set* were updated between *start time* and *end time*."

Alternatively, the *checkpoint* variation of an invalidation subscription sends three things: the most recent invalidation the sender has for every element in *SS* that has been updated since *startVV*; the sender's *presumed inconsistent* state for interest sets that overlap *SS* to reveal any events the checkpoint may not reflect for objects in *SS*; and an imprecise invalidation covering *startVV* until the sender's current logical time to signify that the checkpoint may omit invalidations to objects outside of *SS*. After such a checkpoint is sent, the sender continues to stream each new invalidation it receives using the log streaming mechanism until the subscription is closed.

4.3 Body subscriptions and demand fetch

Whereas invalidation subscriptions make nodes aware of the updates that have occurred, body messages deliver the contents of these updates. Each body message identifies an invalidation with which it is associated and carries the contents of the corresponding update. Body messages can be delivered to a node in any order. A node can request an individual body message (i.e., for a demand fetch of an object) or it can subscribe to receive any new updates that appear for a specified subscription set of objects at another node.

To allow a system to enforce consistency despite having separate streams of (carefully-ordered) invalidations and (unordered) bodies, an *incoming message scheduler* coordinates how received invalidations and updates are applied to a node's state. The scheduler (1) discards any body that is older than currently stored data, (2) delays applying a body until the corresponding invalidation has been applied, and (3) delays applying an invalidation until either a configurable *timeout* has expired or a body matching this or some later invalidation received for the same byte-range has arrived.

This timeout allows a system to balance the performance and availability benefits of maximizing the amount of valid data it stores versus the reduced temporal imprecision [39] it achieves by applying invalidations

quickly [28]. For example, if the timeout is set to infinity, then a node will never apply an invalidation unless it can also apply a body to keep the data valid. Conversely, a timeout of zero ensures that a node becomes aware of writes quickly, though it may then have to block a subsequent read to receive the most current version of the data.

5 Specifying policy

Surprisingly, the simple storage and communication abstractions above are sufficient to describe a broad range of replication systems by specifying (a) a safety policy embedded in a wrapper library that determines a system's consistency guarantees by ensuring that read and write requests block until they may safely complete and (b) a liveness policy embedded in a system's communication triggers that determines a system's topology (which nodes communicate with which nodes) and replication strategy (which nodes receive and store which updates) in order to avoid or minimize blocking of requests.

5.1 Safety policy

In order to provide flexible consistency and higher level interfaces for applications (e.g., a file system over the object store), applications access storage via a wrapper library. Each wrapper library enforces a specific consistency policy by blocking read and write requests until some desired consistency semantic is ensured. Libraries use two basic approaches to determine when it is safe for a request to complete.

First, a library can use the basic consistency information maintained locally by each node's storage to enforce many levels of consistency including best effort coherence, causal consistency [20], temporal error [39], and order error [39]. These libraries simply set parameters on requests to the underlying object store. For example, if the *block-inconsistent* flag is set, reads will block until they can return data that are *causally consistent* as defined above. Otherwise, reads can return *inconsistent* or *presumed inconsistent* data.

Second, a library on one node can communicate with libraries on other nodes to enforce more stringent consistency guarantees like sequential consistency [21] 1-copy serializability [4], open/close consistency [15], or linearizability [14]. Such consistency algorithms are not always simple to implement and we don't advance the state of the art on this front: we implement most of our consistency libraries in Java. Fortunately, libraries are reusable across systems, so a designer who wants to provide standard consistency semantics can simply select from a set of a standard libraries. Future work includes investigating domain specific languages [**?**] or other flexible consistency frameworks [17, 34, 36, 39] for defining consistency protocols.

Local read/write events	
Read needs data	objId, offset, length, logicTime
Read of *presumed inconsistent*	objId, offset, length
Write	objId, offset, length, logicTime
Delete	objId
Connection events	
Inval subscription start	SS, senderId
Inval subscription attached	SS, senderId
Inval subscription end	SS, senderId
Body subscription start	SS, senderId
Body subscription end	SS, senderId
Outgoing inval subscription start	SS, receiverId
Outgoing inval subscription end	SS, receiverId
Outgoing body subscription start	SS, receiverId
Outgoing body subscription end	SS, receiverId
Message arrival events	
Inval arrives	sender, objId, off, len, logicTime
Requested body arrives	sender, objId, off, len, logicTime
Body request failed	senderId, objId, offset, length, logicTime

Fig. 2: Events for liveness policy generated by mechanisms.

5.2 Liveness policy

As just described, to enforce safety (consistency), reads and writes may block until data has propagated to some desired set of nodes in the system. Therefore, a liveness policy works to avoid or end blocking by causing nodes to subscribe to invalidation and body streams to ensure that desired information propagates between nodes. It is such liveness policies that largely define a replication system's architecture.

5.2.1 Events and actions

The liveness policy receives notification of important events that affect the local state (e.g., local read miss, local write, open incoming invalidation subscription, close incoming invalidation stream, receive invalidation, receive body, etc.). Fig. 2 lists all of the events generated by our implementation.

Reacting to such events, the policy then triggers communication of updates between nodes by adding a subscription for an interest set from a start time, removing an interest set from an invalidation-stream subscription, transmitting a body for a specified byte-range, or creating a subscription for a stream of new bodies. Fig. 3 lists policy actions that can be generated.

In addition to these events and communication actions, we provide a way for liveness policies to read and write tuples to persistent objects in our object store. This interface allows a liveness policy to base its action on per-object state. For example, each object stored in Pangaea [32] identifies three nodes that store the "gold replicas" of the object, so a liveness policy for Pangaea must be able to read and update a list of gold replicas for any object in the system. Liveness policies also use this feature to store configuration information (e.g., a list of nodes to peer with or list of files to prefetch [8, 18]) in

Actions generated by liveness policy	
Add pull inval subscription	SS, sourceId [, CP|LOG, start]
Add pull body subscription	SS, sourceId [, start]
Add push inval subscription	SS, destId [, CP|LOG, start]
Add push body subscription	SS, destId [, start]
Remove pull inval sub	SS, sourceId
Remove pull inval sub	SS, sourceId
Remove push body sub	SS, destId
Remove push body sub	SS, destId
Request body	srcId, objId, off, len, logicTime
Push body	destId, objId, off, len

Fig. 3: Actions for mechanisms generated by liveness policy. If the optional *start* parameter is omitted from an add subscription action, a default is used: for pull, the default is the last logical time *SS* was not *presumed inconsistent* and for push the default is the logical time the last connection to *destId* was closed. If the optional *CP|LOG* parameter is omitted, a log is sent unless *start* is earlier than the truncation point in the log, in which case a checkpoint is sent.

the object store.

The persistent-tuple interface simply adds a set of actions to write tuples to objects and a set of events triggered when tuples are read. For example, the action *writeTuple(objId, tupleName, field1, . . . , fieldN)* appends the tuple *tupleName(field1, . . . , fieldN)* to the object *objId*, and the action *readAndWatchTuple(objId)* generates an event for each tuple stored in *objId* and then generates a new event any time a new tuple is appended to that object.

5.2.2 Examples

The liveness policies are best understood by considering several examples. In order to convey the overall approach, these examples are high level and omit some details; we provide the *actual full implementation* of the liveness policies for several examples in Section 7.

Example: Server replication. In a server replication system like Bayou [30], all nodes maintain copies of all data, and any node can exchange updates with any other node. The liveness policy at each server periodically pings other servers to determine when they are reachable. When a server becomes reachable, the liveness policy creates an invalidation subscription and a body subscription for "*" (an interest set that includes all objects) with the node's current version vector [30] as a start time. To match Bayou's 100% availability and causal consistency, a node sets its scheduler to have an infinite timeout (do not apply an invalidation until the corresponding body has arrived) and uses the standard *causalWrapper* consistency library, which blocks reads of not-present, inconsistent, or presumed inconsistent data. ◊

Example: Client-server caching with callbacks. In a client-server system like AFS [15], clients cache recently referenced objects and send updates to a server. Whenever a client reads an object *o* that is *inconsistent*, *presumed inconsistent*, or *not present*, the liveness policy adds *o* to the invalidation subscription from the server *S* to the client *C* using as a subscription start $\perp$ (the beginning of time for the system) and specifying that the server should send a checkpoint containing the most recent invalidation of *o* rather than the full sequence of invalidations to *o*. The server adds *o* to the subscription set for the invalidation stream from *S* to *C* and sends that checkpoint of *o*'s most recent invalidation. Additionally, when such a read occurs, the liveness policy at *C* requests that *S* send a body for *o* if that body is newer than the version of the body stored at *C* or whatever version *S* has if *C* does not have a copy of *o*. Eventually, the invalidation and matching body arrive at *C* and the read may return. Subsequent reads of *o* will be local operations at *C* until the invalidation subscription delivers a newer invalidation of *o*. When such an invalidation arrives, we remove *o* from this invalidation subscription.

To ensure that writes propagate to the server, the liveness policy creates a subscription for "*" from *C* to *S* using *S*'s current version vector as a start time. ◊

Example: Client-server callback crash recovery. Often, a tricky issue in callback based systems is recovering callback state after a server crash or after a client becomes disconnected from the server and reconnects after perhaps missing some invalidations to objects it is caching. The abstractions presented here allow these cases to be handled quite naturally.

Suppose a client becomes disconnected from a server because one or both crash or because the network path between them is lost. If strong consistency is desired, the consistency wrapper at the client will block reads when the client is disconnected; alternatively, systems like Coda weaken consistency guarantees when the client is disconnected [18]. Later, when connectivity is restored, the liveness policy creates an invalidation subscription from the server to the client with an empty interest set ("") and a start time corresponding to the client's current version vector. The server sends a checkpoint for the invalidation state of "", which will, to ensure the stream carries a causal sequence of events, carry an imprecise invalidation covering all invalidations the client missed. This invalidation typically makes all of the client's state *presumed inconsistent.* As a result, subsequent reads block until they establish new callbacks by adding the objects being read to the subscription set as described in the previous example. ◊

5.2.3 Declaring liveness policy

We use the declarative OverLog language [23] and the P2 runtime [23] to specify and execute liveness policy. OverLog statements specify how to create a new tuple (event or table entry) when a set of existing tuples meet some constraint. For example

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

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

When the events listed above occur, Ursa generates the corresponding P2 tuple. Similarly, when a P2 tuple for an action appears, P2 generates an Ursa action.

To give a concrete example of how one could specify an action in a replication system's liveness policy, consider the following rule from our Ursa implementation of TierStore [8]:

```
addPullInvalSubscription@node(node, volume, parent) :-
    newLiveNeighbor@node(node, parent),
    isParent@node(node, parent),
    wantSubscribe@node(node, volume)
```

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

In the above examples, our English translations of simple OverLog statements are admittedly a bit awkward. Also, we acknowledge that this declarative style of programming may be unfamiliar to many system builders. After an initial learning curve where we learned to "think in OverLog," our experience using OverLog to specify liveness policy for replication systems echos Loo et al.'s experience for overlays: "The combination of a declarative language and a dataflow runtime forms a powerful and surprisingly natural environment for overlay specification and runtime." [23] Like Loo et al., we find the effort to learn an unusual syntax offset by the advantages of having descriptions of our systems that are as concise as pseudocode, that specify the system precisely, and that can be executed.

6 Implementation

The previous sections defined the abstractions Ursa relies on to construct replication systems. This section focuses Ursa's novel implementation of the invalidation subscription abstraction. The abstraction requires mechanisms that (1) support flexible liveness policies by allowing any nodes to exchange any subsets of data and (2) support consistency policies by tracking the causal relationships of updates. Supporting either of these aspects alone is straightforward, but efficiently supporting both is challenging. Our implementation realizes the invalidation subscription abstraction by multiplexing invalidation subscriptions on a single stream, allowing Ursa to support large numbers of interest sets an order of magnitude more efficiently than previous systems that provide similar flexibility [3].

6.1 Log and per-object state

Each node stores a log of invalidations as well as per-object state for a policy-specified subset of the system's data. These are similar to past systems [3, 30]; we include a brief description here for concreteness.

Each node maintains a log of invalidations it has processed, causally sorted by each invalidation's logical time. Logical times are based on a Lamport clock [20] so that each update's time exceeds the time of all updates on which it depends. Each node maintains a version vector *currentVV* summarizing which updates it has processed. A node can be configured to limit the log to any specified size by garbage collecting a prefix of the log.

Each node maintains per-object state for a policy-specified set of objects. For each byte-range of each object in that set, the node keeps the logical- and real-time of the last invalidation and, if a body has been processed, the logical time and contents of that body.

6.2 Interest sets and subscriptions

Although the abstraction specifies independent invalidation subscriptions, our implementation multiplexes all invalidation subscriptions from one node to another onto a single underlying invalidation stream. This multiplexing yields two significant advantages for dynamically-created, fine-grained subscription sets such as those needed by callbacks [15, 18, **?**].

First, the network overhead to add a new subscription is reduced because imprecise invalidations covering objects not in the subscription need only be sent once per stream. In particular, an invalidation subscription for subscription set *SS* must also include imprecise invalidations covering updates outside of *SS* so that the receiver can maintain its *presumed inconsistent* state. Multiplexing subscriptions allows one imprecise invalidation to be used across all active subscriptions.

Second, the processing overhead to handle each invalidation is reduced from *O(number of active interest sets)* to update each interest set's state to *O(1)* to update the stream's state. To accomplish this goal, we generalize the notion of establishing a callback with a server to attaching an interest set to a stream.

Implementation. An interest set's effective time *IS.effVV* is the latest time at which the interest set is known to have has seen all invalidations that could affect it. So, if $IS.effVV < currentVV$, the interest set is *presumed inconsistent*. Our implementation sets the the effective VV for an interest set *IS* to be the maximum across a stored VV for the interest set and the causal time for any connection to which *IS* is attached. For the former, to limit space overheads a policy specifies which interest sets maintain persistent state; if *IS* does not maintain such state, the stored VV is taken from $IS_{enclosing}$, the smallest enclosing interest set that maintains per-IS state. To track the latter, for each incoming invalidation stream

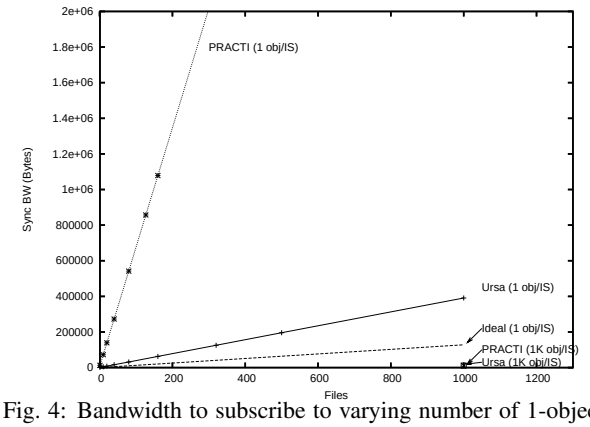

Fig. 4: Bandwidth to subscribe to varying number of 1-object interest sets for Ursa and PRACTI [3]

a node maintains $stream.VV$ that encompasses the latest invalidations provided by the stream. Thus, $IS.effVV = max(IS_{enclosing}.VV, \forall_{stream \in attached} stream.VV)$.

Attaching an interest set to a stream requires receiving a checkpoint or log that brings *IS* up to date with the sender, but once *IS* is attached to *stream*, each invalidation on *stream* advances *IS.effVV* with a single update to the receiver's $stream.VV$ state. When an imprecise invalidation that overlaps *IS* arrives on a stream, the receiver's state is no longer guaranteed to be causally consistent to $stream.VV$, so we detach *IS* from the stream; if the the system is maintaining per-interest set state for *IS*, we update the stored $IS.VV = stream.VV$ at this time.

Example. This approach provides an efficient way to track per-interest state for fine-grained, dynamic interest sets. Consider, for example, emulating per-object callbacks. To ensure that object *o* is not *presumed inconsistent*, a client attaches *o* to the stream of invalidations from its server by sending a request *addSubscription(o,* $\perp$*)* where $\perp$ is a special token concisely representing an all-zero version vector—when an interest set is small, it is cheaper to send $\perp$ as a start time and receive the current logical time for all objects in the interest set than to send a full version vector and receive the logical time only for more recent updates. The server sends the logical time of *o*'s last update and the token *attach:o*. The cost is thus two small messages. Note that in this case a policy would typically configure "/" to be the only persistent interest set to avoid keeping a version vector for each object. $\Diamond$

Example. The approach is also efficient for coarse-grained interest sets. Consider attaching an interest set with 1000 objects, 100 of which have been updated since *IS.effVV*. A node sends *IS.effVV* and receives an object ID and logical time for each of the 100 modified objects followed by the attach token. $\Diamond$

Evaluation. Fig. 4 shows the cost for synchronizing the current state of and requesting future updates to 1000 objects, 100 of which have been modified, in a 100-node system (1) using single-object interest sets to approximate object-by-object callbacks [15, 18, **?**] and (2) using a single interest set spanning all 1000 objects [3, 30]. We measure the cost of callbacks on our implementation of Ursa, an implementation of PRACTI provided by the authors [3], and estimate the ideal cost in a client-server system assuming a callback is established by sending an object id and receiving the object ID and a timestamp.

As Fig. 4 shows, when access patterns have sufficient predictability and locality to use coarse-grained subscriptions, both PRACTI and Ursa outperform even ideal callbacks. However, when fine-grained callbacks are established dynamically, Ursa approximates callbacks and is an order of magnitude cheaper than PRACTI. PRACTI pays a high cost because each subscription establishes a new, independent connection by sending a version vector summarizing the current state of IS and then receiving an imprecise invalidation describing all invalidations to objects not in IS. Additionally, PRACTI pays a cost proportional to the number of interest sets to process each invalidation because each invalidation must update the state of each interest set.

6.3 Other implementation details

Our prototype is written in Java and uses BerkeleyDB for storage.

We use an NFS server that implements a file system over our object store. Note that systems that provide file service typically run an NFS server on each node, mount that file system locally, and do not use NFS from a remote client

7 Case studies and evaluation

This section examines the value of Ursa's approach to constructing replication systems by examining a series of case studies. One should view this evaluation as both (1) assessing how useful our artifact is for constructing prototype replication systems and (2) testing whether the ideas provide useful principles for thinking about and understanding replication systems.

To evaluate our architecture, we consider three criteria:

- *Generality/Flexibility*. Is the architecture capable of describing a wide range of systems including client-server systems [15, 18], server-replication systems [30, 38], object replication systems [12, 32], and so on?
- *Pithiness*. Is the specification of systems concise and elegant? Pithiness is evidence that abstractions capture the essence of the underlying concepts.
- *Efficiency*. Are the resulting systems comparable to hand-built systems? Does the architecture facilitate improvements in a design?

	Safety Policy		Liveness Policy
	Standard	Customized	
Bayou	12	-	29
Chain Replication	-	88	76
Coda	157	-	31
Pangaea	12	-	67
TierStore	12	-	28
Trip	12	-	22

Fig. 5: Lines of code required to implement each system Standard safety policy means that a standard library wrapper was used. Customized means that a customized library wrapper was used to implement the system.

To that end, this evaluation focuses on a series of case studies spanning diverse classic and new systems from the literature. To illustrate pithiness, we provide a detailed implementation for one system (Ursa-Coda), and highlight key pieces of several others. To illustrate flexibility, we discuss in less detail a broad range of systems covering a large part of design space: Bayou [30], Pangaea [32], Tier-Store [8], Chain Replication [38], and TRIP [28]. To illustrate efficiency, we analytically and experimentally assess the extent to which our implementations are *equivalent* to prior systems according to the definition in §2. Fig. 5 summarizes the number of lines of code which were used to implement each system.

Finally, as an illustration of all three properties, we show how constructing a system on Ursa's flexible framework makes it easy to extend a design by adding new features. We add cooperative caching to Coda in 2 lines; this addition allows a set of disconnected devices to share updates while retaining consistency. We add small-device support to Bayou in 1 line; this addition allows devices that 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 Tier-Store in 2 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 an order of magnitude improvement for some scenarios of interest.

All experiments are run on Dell Dimension 4100 machines with 800MHz Pentium-III CPUs, 256MB of memory, and 100Mb/s Ethernet. We use Fedora Core 6, Sun JVM 1.5, and Berkeley DB Java Edition 1.7.1. In some experiments, we artificially introduce network latency or throttle network bandwidth using NIST Net.

7.1 Micro-benchmarks

Although Ursa has not been extensively tuned, its performance is adequate for prototyping systems. For example, Fig. 6 compares accessing local Ursa storage via our NFS wrapper with accessing the local Linux ext3 file system via the Linux NFS server. For the Andrew benchmark [15] Ursa suffers a factor of two slowdown. When we configure Ursa to implement a client-server system and run the benchmark with an empty client cache and a network with a 10ms latency, performance falls by about another factor of two.

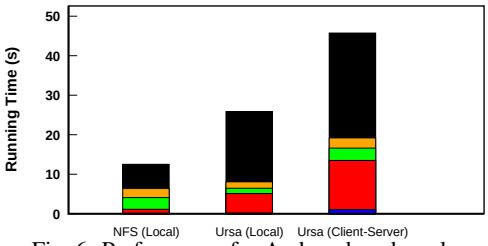

Fig. 6: Performance for Andrew benchmark.

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

Fig. 7: P2 Performance

Fig. 7.1 summarizes the round trip ping latency and maximum throughput per second for a null P2 event inserted by Ursa on a local machine or remote machine.

7.2 Ursa-Coda

To illustrate how to construct a replication system with Ursa, this section discusses in detail Ursa-Coda, a system designed to be *equivalent* to the version of Coda described by Kistler et al [18]. In particular, we support 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 particular, in Coda, some files and directories can be given a lower priority and will be discarded from cache before others. Thus, Ursa-Coda is only equivalent when the hoard set fits on client disk. 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 Ursa-Coda.

7.2.1 Implementing safety policy

Coda provides open/close semantics when connected and enforces causal consistency when disconnected. We employ an open/close wrapper library that buffers writes in the library until close, at which point it will make the writes to Ursa's storage. If the node is connected to the server, the library will block until the server gathers invalidation acknowledgements and reports "done." For a read, the library first looks in its buffer. If the object is not there, it issue a read to Ursa's storage which will block until the local data is *consistent* This standard library is usable by different systems. It has 124 semicolons of Java and 22 P2 rules.

```
// Get server and list of peers from config file
  cf1    readAndWatchTuple@X(X, nodeFile) :-
             init, nodeFile:=/coda/nodeList
  cf2    server@X(X, S) :-
             configServer@X(X, S)
// Server connection status
  c1     isConnected@X(X, V) :-
             newLiveNeighbor@X(X, S), server@X(X, S), V:=1
  c2     isConnected@X(X, V) :-
             declareDeadNode@X(X, S), server@X(X, S), V:=0
// Local read miss: Add an inval subscription
  sc1    addPullInvalSubscription@X(X, S, Catchup, Obj) :-
             localReadImprecise@X(X, Obj, _, _), server@X(X, S),
             isConnected@X(X, V), V==1, Catchup:=CP,
             X≠S //I am client
// ... and get a body
  sc2    demandRead@X(X, S, Obj, Off, Len) :-
             localReadInvalid@X(X, Obj, Off, Len), server@X(X, S),
             isConnected@X(X, V), V==1, X≠S //I am client
// Server is detected: add subscriptions to send updates to server
  cs1    addPullInvalSubscription@S(S, X, Catchup, SS) :-
             isConnected@X(X, V), V==1, server@X(X, S), SS:=/*,
             Catchup:=LOG, X≠S // I am client
  cs2    addPullBodySubscription@S(S, X, Catchup, SS) :-
             isConnected@X(X, V), V==1, server@X(X, S), SS:=/*,
             Catchup:=LOG, X≠S // I am client
// Client receives an inval: Remove subscription
  cbr    removePullInvalSubscription@S(S, X, Obj) :-
             invalArrives@X(X, S, Obj, _, _, _), server@X(X, S),
             X≠S // I am client
// Server is detected: Add subscription to "" to see if I missed anything
  re     addPullInvalSubscription@X(X, S, Catchup, SS) :-
           isConnected@X(X, V), V==1, server@X(X, S),
           SS:=EMPTY, Catchup := LOG, X≠S // I am client
// Hoarding: Add hoard subscriptions when server reachable
  h1     readAndWatchTuple@X(X, hoardFile) :-
             isConnected@X(X, V), V==1, hoardFile:=/coda/hoardList
  h2     addPullInvalSubscription@X(X, S, Catchup, SS) :-
             doHoard@X(X, SS), Catchup:=CP, server@X(X, S),
             X≠S // I am client
  h3     addPullBodySubscription@X(X, S, Catchup, SS) :-
             doHoard@X(X, SS), Catchup:=CP, server@X(X, S),
             X≠S // I am client
// Cooperative caching: Check reachable peers if server unreachable
  cc1    peer@X(X, P) :-
             configPeer@X(X, P)
  cc2    pConnected@X(X, P, V) :-
             newLiveNeighbor@X(X, P), peer@X(X, P), V:=1
  cc3    pConnected@X(X, P, V) :-
             declareDeadNode@X(X, P), peer@X(X, P), V:=0
  cc4    demandRead@X(X, P, Obj, Off, Len) :-
             localReadInvalid@X(X, Obj, Off, Len),
             isConnected@X(X, V), V==0, pConnected@X(X, P, W),
             W==1, server@X(X, S), X != S // I am client
```

Algorithm 1: Liveness policy for a Coda-like system. 17 rules for monitoring connectivity are excluded.

7.2.2 Implementing liveness policy

Ursa-Coda's 31 liveness rules can be divided into 6 main groups: configuration, connectivity, demand read, write propagation, recovery, and hoarding. Algorithm 1 defines Ursa-Coda's liveness policy.

Configuration and connectivity. A configuration file stores the server's identity in a *configServer* tuple and another configuration file provides the hoard list in a series of *doHoard* tuples. At each client C, the table entry *isConnected@C(C, S)* indicates whether the server S is currently reachable. We use 17 rules (not shown) based on the published P2 implementation of Narada [23] to track connectivity information and generate *newLiveNeighbor* and *declareDeadNode* tuples, which invoke rules c1 or c2 respectively.

Demand read. Two rules are triggered when a demand read of object *o* occurs at a connected client. sc1 subscribes for *o*'s invalidations using a checkpoint for efficiency, and sc2 demand-fetches the body. Eventually, *o* is no longer *presumed inconsistent* or *inconsistent*, and the safety policy can unblock the read. The invalidation subscription ensures that if another node updates *o*, it will become invalid so that writes can complete.

Write propagation and callbacks. To propagate client writes to the server, rules cs1 and cs2 are triggered when the client connects to the server, and they create an invalidation and a body subscription from the client to the server. Note that the server sets the scheduler timeout to infinity so that invalidations are not applied until the server receives the corresponding bodies. (Conversely, the clients set the scheduler timeout to 0 to process invalidations immediately and demand fetch bodies.)

The invalidation subscriptions created by clients when they read objects from the server ensures that our underlying mechanisms transmit invalidations. The safety policy is responsible for unblocking a write once the invalidations have been delivered by these liveness-triggered rules. To avoid sending repeated callbacks to a client, we include a rule cbr to remove an object from the invalidation subscription when a client receives a callback.

Recovery. When a client reconnects to a server, it triggers re, to establish an invalidation subscription for an empty subscription set from the server. This action causes the server to send an imprecise invalidation for all updates the client has missed, typically making all data *presumed inconsistent* at the client.

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

7.2.3 Adding cooperative caching

Four rules provide a way for disconnected clients to fetch data from their peers. We augment the node list configuration file to include a list of peers; when cf1 reads the config file, it generates *configPeer* tuples that populate the peer table via cc1. cc2 and cc3 keep track of connectivity to peers. cc3 triggers a demand read attempt from reachable peers if the server is not reachable.

7.2.4 Evaluation

Ursa-Coda is Q-equivalent to Coda if the number of writes at each node between disconnections exceeds the number of nodes and if if the initial state, the number of clients, the hoard set at each client, and the workload is the same.

E1. *Overhead.* Both systems issue and process the same writes, invalidations, and local/remote reads. Establishing or breaking each callback has a constant cost that is near the cost of ideal callbacks (see Fig. 4). Similarly, demand read requests are of constant size and demand read replies have the data plus a constant overhead. Establishing one body update subscription to propagate updates to the server and establishing the first invalidation connection to/from the server each entail sending a version vector, so Ursa-Coda is only Q-equivalent to Coda if the number of invalidation and body messages sent to/from the server exceeds the number of entries in a version vector so that network bandwidth is within a constant factor.

E2. *Consistency.* Both systems enforce open/close semantics when connected and causal when disconnected.

E3. *Available local data.* For both systems,at each client, only the objects that have an established callback are available during connected operation and only the objects that are *consistent* are available during disconnected operation.

7.3 Bayou

We use Ursa to implement a server replication system modelled on the version of Bayou described by Petersen et al. [30].

Bayou enforces causal consistency, so we instantiate a causal consistency wrapper library that blocks reads to *inconsistent* or *presumed inconsistent* data as our safety policy.

For liveness, Bayou uses peer-to-peer anti-entropy sessions to disseminate all updates to all nodes. Anti-entropy is easily implemented on Ursa. When node A wants to carry out anti-entropy with node B, it establishes an invalidation and a body subscription with subscription set "*". Once all updates have been applied and the two nodes have synchronized, A removes the subscriptions. Note that if the log at B is truncated to a point beyond A's knowledge, the invalidation subscription will automatically send a checkpoint rather than the log; Bayou's approach is similar.

The complete implementation of Ursa-Bayou's liveness policy requires 29 rules: 4 for anti-entropy liveness (Alg. 2), 8 for random neighbor selection (omitted) and 17 for connection management (omitted.)

Equivalence. Ursa-Bayou is Q-Equivalent to the original Bayou implementation assuming that nodes execute anti-entropy with the same peers during the same quiescent periods in the workload. The network overhead is

```
// Add Subscriptions when a random neighbor is selected
 bc01   addPullInvalSubcription@X(X, Y, SS) :-
            selectedNeighbor@X(X, Y), ss@X(X, SS)
 bc02   addPullBodySubcription@X(X, Y, SS) :-
            selectedNeighbor@X(X, Y), ss@X(X, SS)
// Remove Subscripiton when we have received all updates
 bc03   removePullInvalSubcription@X(X, Y, SS) :-
            informInvalSubscriptionAttached@X(X, Y, SS)
 bc04   removePullBodySubcription@X(X, Y, SS) :-
            informInvalSubscriptionAttached@X(X, Y, SS)
```

Algorithm 2: Rules for liveness in a Bayou-like system

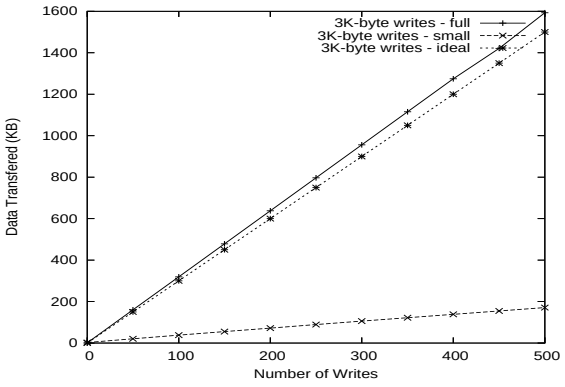

Fig. 8: Anti-Entropy bandwidth on Ursa-Bayou

the information transferred during anti-entropy. For both systems, the number of bytes transferred during anti-entropy is proportional to the number of updates which the sender has but the receiver doesn't and the size of the updates. Or if checkpoints are sent, the size of checkpoint is directory proportional to the size of changed objects. Both systems store all objects locally. As for the log, both garbage collect old log entries to keep the log to a specified maximum size. Additionally, both systems enforces casual and eventual consistency. Finally, for both systems, 100% of the data is always locally available at every node.

Small-device support. We extend this system to support small devices. Instead of storing the whole database, a node can specify an interest set containing the set of objects or directories it cares about. During anti-entropy, it simply needs to add subscriptions for its interest set rather than "*". After anti-entropy, all data in its interest set is *inconsistent*, and the rest is *presumed inconsistent*. This requires a change of 1 liveness rule.

Experimental Evaluation. As Figure 8 indicates, the overhead for anti-entropy in Ursa-Bayou is relatively small compared to "ideal" anti-entropy. In addition, if a node requires only 10% of the data, the small device enhancement in Ursa-Bayou greatly reduces the bandwidth required for anti-entropy.

	Coherence-only		Ursa	
	Messages	Bytes	Messages	Bytes
Worst Case	1	26	2	52
Bursty workload (10)	1	26	1.1	30

Fig. 9: Messages and bytes per invalidation sent by Ursa and coherence-only system.

7.4 Pangaea

Pangaea [32] 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 (i.e. invalidations) 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.

Ursa-Pangaea uses a standard coherence-only wrapper, which does not block any reads or writes. Additionally, if an individual node wishes to enforce stronger consistency, that node may instantiate the causal or TE wrapper to block reads and thereby enforce causal consistency or a bound on staleness (temporal error.)

The liveness policy comprises 67 rules to create a system Q-equivalent to Pangaea. Due to space constraints, we omit a detailed discussion. Most of the complexity stems from (1) constructing the required per-file 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.

Equivalence. Ursa propagates sufficient information for any node to enforce causal consistency using local information. Ursa-Pangaea does not use this information, so it retains the high availability, partition-resiliance, and performance [10, 22] available to weak-consistency systems. Furthermore, the overhead of propagating this information is modest for several reasons. First, as Fig. 9 indicates, once a connection between a pair of nodes is established, Ursasends at most two invalidation messages for every one sent by a coherence-only algorithm—at worst an Ursa node alternates sending an invalidation requested by the receiver and an imprecise invalidation summarizing updates the receiver has not requested. Furthermore, as long as there is locality workload's updates in the object ID space, imprecise invalidations are comparable in size to regular invalidations. Finally, as the figure indicates, if there are bursts of load to objects of interest, the ratio of invalidations to imprecise invalidations improves. As a result, network bandwidth remains within a constant factor of a coherence-only protocol if the workload has sufficient locality in the object ID space for imprecise invalidations to achieve good compression.

Experimental Evaluation. Fig. 10 illustrates one aspect of Pangaea's performance: its ability to dynamically choose the best replica from which to fetch data. In this experiment, a simplified version of the experiment presented in Saito et al.'s Figure 11 [32], we measure the time to satisfy a cache miss from one of three replicas. We compare three policies: (1) Pangaea (locality-based), (2) Random, and (3) Static. For the Static policy, we show results for each of the three possible choices. Our results are consistent with Saito et al.'s experiment: not surprisingly, fetching data from a nearby node is a good policy.

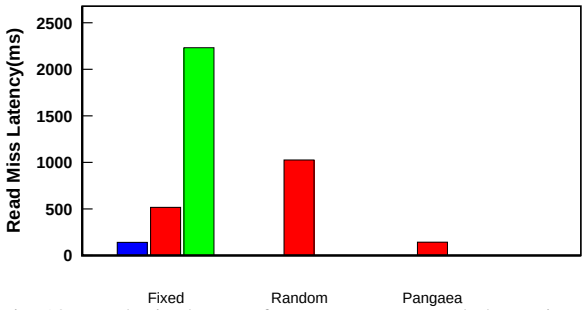

Fig. 10: Read miss latency for Ursa-Pangaea and alternatives.

7.5 Chain Replication

Chain Replication [38] is a server replication protocol in which the nodes are arranged as a chain to provide high availability and sequential consistency. 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. Chain management is carried out by a master node. Data are divided into volumes, and each volume is assigned to its own chain.

Ursa-CR implements this protocol with support for volumes, addition of a new node, and node failure and recovery. The master node is implemented in OverLog. Although we could use our standard library for enforcing sequential consistency we implement a customized local interface wrapper that exploits the chain topology and simply blocks an update until it receives an ack from the tail. This custom safety library required 85 semi-colons of Java and 3 OverLog rules.

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 and synchronizing a recovering node. Ursa's mechanisms handle these housekeeping details. In particular, if node A in the middle of the chain dies, the successor of A will establish subscriptions from node A's predecessor. Because of the semantics guaranteed by invalidation subscriptions, the successor will receive all updates it hasn't seen, including those that were sent to node A and were lost. The liveness policy totals 76 rules: 3 for update propagation, 9 for chain management at the servers, 35 for chain management at the master, 20 for connection management, and 9 for initialization and miscellany.

Our implementation is Q-equivalent to the published system. We omit detailed discussion of this unsurprising result due to space constraints.

7.6 TierStore and TRIP

We also implement TierStore [8], a hierarchical replication system for developing regions, and TRIPP [28], a system that seeks to provide transparent replication of dynamic content for web edge servers. We summarize the relevant statistics in Fig.5. Due to space constraints, we omit detailed discussion.

What is perhaps most interesting about these examples is the extent to which Ursa facilitates evolution. For example, the TRIP implementation assumes a single server and a star topology. By implementing on Ursa, 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—Ursa's mechanisms are general enough so that this "just works"—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 [23], a few additional rules to subscribe/unsubscribe produce a dynamic-tree version of TRIP that still enforces the required consistency. Note that we have implemented the static tree policy but not the dynamic tree policy.

8 Related work

PRACTI [3] defines a set of mechanisms for replication systems. Like PRACTI, Ursa supports Partial Replication, Arbitrary Consistency, and Topology Independence, and like PRACTI our mechanisms separate invalidations from bodies and use imprecise invalidations to avoid sending all invalidations to all nodes. Our mechanisms are an order of magnitude more efficient than PRACTI for the fine-grained, dynamic replication schemes used by a number of systems [**?**, 15, 32]. More broadly, where PRACTI defines a set of mechanisms, Ursa defines new mechanisms, defines an architecture for separately specifying safety and liveness policy, shows how to implement liveness policy using a declarative language, and demonstrates how this approach facilitates construction of a wide range of systems that are equivalent to or improvements on systems from the literature.

A number of other efforts have defined general frameworks for constructing replication systems for different environments. Deceit [35] focuses on replication across a well-connected cluster of servers. Zhang et al. [40] define an object storage system with flexible consistency and replication policies in a cluster environment. Stackable file systems [13] seek to provide a way to add features and compose file systems.

Ursa incorporates the order error and temporal error abstractions of TACT tunable consistency [39]; we do not currently support arithmetic error. Although we base Ursa on TACT, there are other efforts to support flexible consistency that could support the vision of providing a set of standard libraries in a replication toolkit. IceCube [17] and actions/constraints [34] provide frameworks for specifying general consistency constraints and scheduling reconciliation to minimize conflicts. Like Ursa, Swarm [36] 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 [20], sequential [21], 1SR [4], or linearizability [14]. Fluid replication [6] provides a menu of consistency policies, but it is restricted to hierarchical caching.

Ursa uses P2 [23] to execute our declarative liveness policies. More broadly, we follow in the footsteps of efforts to define runtime systems or domain-specific languages to ease the construction of routing [23], overlay networks [31], cache consistency protocols [**?**], and routers [19].

The specific optimizations we add to replication systems—cooperative caching [7], bulk resynchronization [27], and small device support [18]—have all been done before. Our contribution is to provide an abstractions that supports such optimizations in a general way and that make it simple to evolve an existing system by adding new features.

9 Experience and conclusion

We started this project with the goal of building an infrastructure that would allow a couple of graduate students to rapidly build a dozen or so classic and cutting edge replication systems. Looking back, four "aha" moments stand out. First, was our realization that much of what distinguishes different replication systems can be regarded as routing policy. This realization allowed us to exploit the OverLog declarative language for pithy description of that aspect of policy.

Routing could be used to define much of the policy of a replication system but there were still some aspects that didn't quite fit into routing. Was there a clean way to think about the rest? Our second realization was that "the rest" are essentially the safety guarantees and that they can be cleanly implemented at the read- and write- interface by casting these safety guarantees as consistency constraints.

Although it is appealing to be able to send any data to any node while enforcing consistency, doing so in a scalable way is daunting. The third insight was that we could adapt PRACTI-like mechanisms to support not just coarse-grained subscriptions but fine-grained callbacks by multiplexing logical invalidation streams onto a single physical connection.

Finally, we were left with the question of how to convince ourselves and others that our implementation captured the essence of the systems we were aiming to build. The fourth "aha" was when we realized that we needed to precisely define the equivalence between two systems in a way that captured the key trade-offs any system makes in addressing the CAP [10] and PC [22] dilemmas.

The Ursa architecture uses these observations to dramatically reduce the effort needed to construct a new replication system or enhance an existing one. Given Ursa we can prototype many interesting replication systems in a day or two using a handful of lines of system-specific policy code.

With Ursa building replication systems is easy to bear.

References

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

[2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proc. ASPLOS*, pages 10–22, Sept. 1992.

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

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

[5] S. Chandra, B. Richards, and J. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proc. PLDI*, May 1996.

[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. `http://tier.cs.berkeley.edu/docs/projects/tierstore.pdf`, Dec. 2006.

[9] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proc. WWW*, May 2003.

[10] 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.

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

[12] R. Guy, J. Heidemann, W. Mak, T. Page, G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer Conf.*, pages 63–71, June 1990.

[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] 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.

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

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

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

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

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

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

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

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

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

[25] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, Dec. 1999.

[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] D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *Proc. OSDI*, 2006.

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

[31] 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.

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

[33] 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. Technical Report 89-1042, Cornell, Nov. 1989.

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

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

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

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

[40] 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.