The Dissertation Committee for Jiandan Zheng

certifies that this is the approved version of the following dissertation:

URA: A Universal Data Replication Architecture

Committee:

Mike Dahlin, Supervisor

Lorenzo Alvisi

James C. Browne

Arun Iyengar

Lili Qiu

Emmett Witchel

URA: A Universal Data Replication Architecture

by

Jiandan Zheng, B.S.; M.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2008

Dedicated to my beloved husband,

my caring parents,

and my wonderful brothers.

Acknowledgments

Thanks to Mike Dahlin, my research adviser, who offered me the opportunity to work in the LASR lab with a group of talented and energetic people. Mike showed me the art and science in research in computer systems. I am really impressed by Mike's intuition, thoughtfulness, and quick comprehension capability. His valuable suggestions and the discussions on research ideas lead me towards better and clearer understanding of my research subject. I greatly cherished his advice on research, coding, writing, presentation, and time management and I drew a great inspiration and learned a lot from articles that he occasionally distributed to all his students on career and general advice, and on programming standards. Without Mike's encouragement and help during the difficult times in my PhD study, this dissertation would not have been possible.

I am also thankful to professor Lorenzo Alvisi for his encouragement and valuable advice. I really enjoyed several chats with him regarding career and research and really appreciate his help on my thesis, my presentations, and job hunting etc.. I learned a lot from Lorenzo's Distributed Systems class; that is the best class I have ever had.

I would like to thank J. C. Browne, Arun Iyengar, Lili Qiu and Emmett Witchel for serving on my PhD committee. I have a deeper understanding of the subject thanks to their insight and suggestions. I also appreciate their patience to reschedule my defense due to family reason.

I owe a great deal to all of the LASR group. In particular, I must extend my thanks to Sara D Strandtman for her wonderful administrative support, especially for her help in

arranging my final oral examination while I was in China. I would like also thank my colleagues Amol Nayate, Lei Gao, Ramakrishna Kotla, Yin Jian, J.P. Martin, Harry Li, Amitanand Aiyer, Edmund L. Wong, Prince Mahajan, Sangmin Li, Nalini Balaramani, Ravi Kokku, Taylor Riche, Allen Clement, Jeff Naper, Arun Venkataramani, Praveen Yalagandula, Upendra Shevade, and Navendu Jain, not only for their cooperative works and helping hands during these school years, but also for the friendly and relaxing working environment created by them.

Last, but not the least, I would like to extend my appreciation to my family and all my friends for their long-term caring and support. In particular, I want to thank my beloved mother who always taught me since I was a little girl to work hard, never give up, never be satisfied with past achievements, always look ahead and set bigger goals.

JIANDAN ZHENG

The University of Texas at Austin
August 2008

URA: A Universal Data Replication Architecture

Publication No. _____

Jiandan Zheng, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Mike Dahlin

Data replication is a key building block for large-scale distributed systems to improve availability, performance, and scalability. Because there is a fundamental trade-off between performance and consistency as well as between availability and consistency, systems must make trade-offs among these factors based on the demands and technologies of their target environments and workloads.

Unfortunately, existing replication protocols and mechanisms are intrinsically entangled with specific policy assumptions. Therefore, to accommodate new trade-offs for new policy requirements, developers have to either build a new replication system from scratch or modify existing mechanisms.

This dissertation presents a universal data replication architecture (URA) that cleanly separates mechanism and policy and supports Partial Replication (PR), Any Consistency (AC), and Topology Independence (TI) simultaneously. Our architecture yields two significant advantages. First, by providing a single set of mechanisms that capture the common underlying abstractions for data replication, URA can serve as a common substrate for building and deploying new replication systems. It therefore can significantly reduce the effort required to construct or modify a replication system. Second, by providing a set of

general and flexible mechanisms independent of any specific policy, URA enables better trade-offs than any current system can provide. In particular, URA can simultaneously provide the three PRACTI properties while any existing system can provide at most two of them. Our experimental results and case-study systems confirm that universal data replication architecture is a way to build *better* replication systems and a *better way* to build replication systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data replication is a fundamental technique for improving the performance [8, 10, 24, 27, 68, 73, 77, 94, 28], availability [18, 27, 50, 102, 21], ubiquity [48, 72, 23], persistence [61, 49], and managability [1, 69, 71] of a broad range of large-scale distributed systems such as mobile file systems [48, 69, 72, 23], web services [24, 27, 94], enterprise file systems [42, 77, 29], and grid replication systems [5, 73].

Despite decades of research on data replication, we lack a single "perfect replication system" for all environments. The fundamental technical reason behind this problem is that any replication system must make trade-offs between availability, consistency, performance, and partition-resilience based on the demands and technologies of their target environments and workloads. For example, the well-known CAP (Consistency/Availability/Partition-resiliance) dilemma [13, 31, 84] proves that systems cannot simultaneously achieve both sequential consistency and high availability if there are network partitions. Similarly, Lipton and Sandburg [54] prove that there is a fundamental trade-off between performance and consistency when data are shared by multiple nodes. As a result, most existing replication systems are built for specific environments or workloads.

Therefore, as technologies and workloads evolve, it is safe to presume that just as many replication systems have been built for different environments in the past [48, 69, 72,

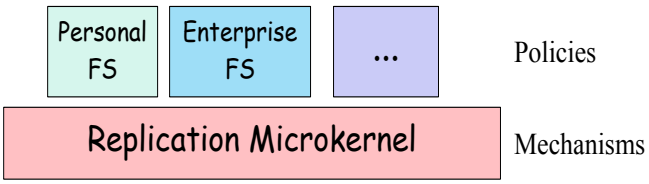


Figure 1.1: URA vision.

42, 77, 29, 23], we will continue to need to develop new replication systems to meet current and future application challenges.

Unfortunately, the state of the art for constructing new replication systems for new environments is lacking. New systems are typically built from the ground up and co-mingle new mechanisms and new policies in their design. This ad-hoc approach makes it hard to construct and deploy new systems because each system must be built from scratch, makes it hard to evolve systems because the existing implementation is based on specific policy assumptions, makes it hard to teach the principles of replication because we are forced to teach a series of case studies rather than a set of basic principles, and makes it hard to focus the community research agenda because it is difficult to identify the open questions or to isolate the contributions of a newly-proposed system.

We believe that, many existing systems are "special cases" of a more fundamental underlying protocol. This dissertation therefore investigates a *universal data replication architecture* (URA) over which a broad range of replication systems can be constructed with dramatically less effort than current approaches.

As indicated in Figure 1.1, the goal of the *universal data replication architecture* is to build a *replication microkernel* that cleanly separates mechanisms and policies so that different replication systems can be built on the same set of mechanisms, and they only differentiate from each other by policies.

URA yields two significant advantages compared to existing approaches. First, it is a *better way* to build replication systems. By providing a single set of mechanisms that

	U-Bayou[72] +Small Device	U-Chain Repl [91]	U-Coda [48] +Coop Cache	U-Pangaea [77]	U-Tier Store [23]+CC	U-TRIP [66]+Hier
Consistency	Causal	Lin.	Open/close	Coherence	Coherence → Causal	Seq.
Topology	Ad-Hoc	Chains	Client/ Server	Ad-Hoc	Tree	Client/Server → Tree
Partial Replication	√		√	√	√	
Prefetching/ Replication	√	√	√	√	√	√
Cooperative Caching			√		√	
Disconnected operation	√		√		√	
Callbacks			√	√		√
Leases			√			
All reads served locally	√	√			√	
Crash recovery	√	√	√	√	√	√
Object store interface	√	√	√	√	√	√
File system interface	√	√	√	√	√	√

Figure 1.2: Systems built on URA and the features they implemented.

capture the common underlying abstractions for data replication, URA can serve as a common substrate for building and deploying replication systems. It therefore can significantly reduce the effort required to construct or modify a replication system.

As illustrated in Figure 1.2, to test whether we succeed in building a universal "microkernel", we have built a broad range of replication systems that covers a large design space using URA. These systems demonstrate the power of URA: (1) it is *flexible* in that we are able to construct systems with a wide range of architectures and features; (2) URA is *efficient* in that we are able to build systems that are comparable to hand-built systems from literature with respect to the central properties of a replication architecture;and (3) URA *facilitates innovation* by exposing new design space and making it much easier to add new features in existing systems.

Although we do not argue that our URA implementations are *identical* to the original systems on which they are based, we do believe that they capture all important features relating to how these systems maximize performance, availability, and consistency.

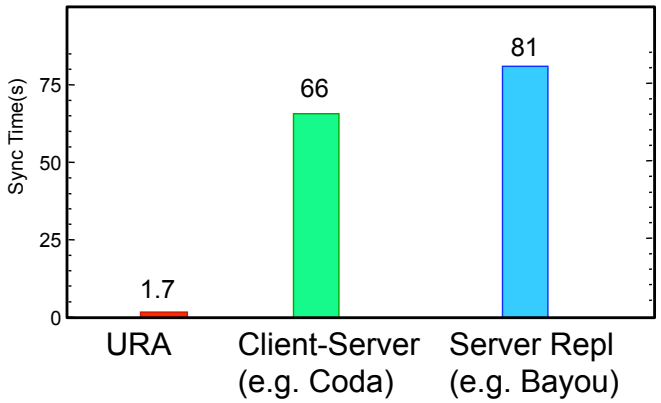


Figure 1.3: Synchronization time between palmtop and laptop.

Figure 1.2 enumerates many of the key features included in our implementations. In Section 7.1.1, we more formally define a notion of *replication equivalence* and argue that our URA implementations are *replication equivalent* to the systems they model.

Second, it is a way to build *better replication systems*. By providing a set of general and flexible mechanisms independent of any specific policy, URA enables better tradeoffs than any current system can provide. In particular, URA can simultaneously provide PRACTI properties, i.e., Partial Replication (PR), Any Consistency (AC), and Topology Independence (TI) while existing systems [96, 50, 103, 68, 48, 65, 20, 25, 4, 89, 76, 17, 75, 72, 36, 77, 60] can only simultaneously provide at most two of them. As a result, PRACTI allows us to construct systems whose performance is at least comparable to and sometimes much better than systems constructed using existing replication protocols.

Notably, by providing PRACTI simultaneously, URA replication can yield significant practical advantages over existing approaches. For example, Figure 1.3 summarizes an experiment that we will describe in more detail in Chapter 4. It shows the time required to synchronize updates between a palmtop and a laptop when the connection between these two devices is good, but when the connection to the rest of the Internet is limited. As indicated in the graph, the URA architecture is *more than an order of magnitude*

better than two of the three major classes of existing replication architectures—traditional hierarchical/client-server [65, 68, 48] and server-replication architectures [96, 72]. Because URA can adapt to the network conditions (TI), it dominates the restricted topology PR-AC client-server approach; because URA can exploit locality to send just the information of interest to the palmtop (PR), it dominates the full-replication (AC-TI) server replication approach. Although the third major class of existing replication architectures—PR-TI object replication architectures [36, 77, 60]—are capable of having performance comparable to URA, these systems give up URA's ability to provide cross-object consistency guarantees (AC), which can increase complexity and errors for applications and users. By providing all these PRACTI properties, URA dominates most of existing approaches.

1.1 Challenges

The challenges to constructing such a universal data replication architecture mainly come from the need to meet three requirements.

- First, the architecture must be *flexible*. Because replication systems cover a large design space, to serve as a common substrate for developing different systems, the architecture must be capable of implementing a wide range of systems including client-server systems [65, 68, 48], server-replication systems [96, 72], object replication systems [36, 77, 60], systems that use callbacks [65, 68, 48], systems that use polling [79, 11], systems that use leases [33, 101], systems that subscribe to groups of objects [23, 48], systems that fetch individual objects on demand [65], and so on.

- Second, the architecture must be *efficient*. It should not only be able to support a wide variety of systems but also ensure that the cost a specific system implementation pays is proportional to the demands it has, i.e., resulting in systems that are comparable to hand-built systems.

- Finally, the architecture interface exposed to system designers must be *simple*. On

one hand the architecture should support as many replication policy choices as possible to cover the large design space. On the other hand, it must make it easy for system designers to specify replication policies and reason about replication cost and consistency semantics.

Meeting those requirements in a single framework is challenging because the combination of some of the requirements make it difficult to support other requirements. In particular, as we will illustrate in more details in Chapter 2, supporting flexible consistency (to be flexible and simple) requires careful ordering of how updates propagate through the system, but consistent ordering becomes more difficult if nodes communicate in ad-hoc patterns (to be efficient) or if some nodes only know about updates to some objects but not those to other objects (to be efficient).

1.2 Approaches

To address these challenges, the URA design strictly follows one principle: *separation of mechanisms and policies*. The mechanisms define the abstractions for storage, communication, and consistency that automatically handle the bookkeeping needed to allow policies to distribute data however they want. Policies specify system-specific choices such as what to synchronize, who to synchronize with, and when to synchronize. The URA implementation defines a *core* that embodies these mechanisms and leaves policy definition to a separate *controller* so that different deployments may use different controllers to implement different policies. This thesis focuses on the core mechanisms.

1.2.1 CR-Repl: Coarse-grain PRACTI Replication

As a first step to meet the above three requirements, the URA *core* implements a *coarse-grain replication protocol (CR-Repl)* that supports three vital properties simultaneously: *Partial Replication* (PR)—for flexibility and efficiency any data can be replicated on any

device; *Any Consistency* (AC)—for flexibility and simplicity the architecture supports a range of consistency guarantees; and *Topology Independence* (TI)—for flexibility and simplicity information can flow between any pair of nodes.

To implement these three vital properties, CR-Repl draws on key ideas of existing protocols but recasts them to remove the deeply-embedded policy assumptions that prevent one or more properties. In particular, our design begins with log exchange mechanisms that support a range of consistency guarantees and ad-hoc communication topology but that fundamentally assume full replication [72, 96, 103]. To support partial replication, we extend the mechanisms in two simple but fundamental ways.

1. In order to allow partial replication of data, our design *separates the control path from the data path* by separating invalidation messages that identify what has changed from body messages that encode the changes to the contents of objects. Distinct invalidation messages are widely used in hierarchical caching systems [65, 48, 12], but we demonstrate how to use them in topology-independent systems: we develop explicit synchronization rules to enforce consistency despite multiple streams of information, and we introduce general mechanisms for handling demand read misses.

2. In order to allow partial replication of update metadata, we introduce *imprecise invalidations*, which allow a single invalidation to conservatively summarize a set of invalidations. Imprecise invalidations allow us to provide cross-object consistency in a scalable manner in which each node incurs storage and bandwidth costs proportional to the size of the requested data sets.

   To support imprecise invalidations, we define a protocol that allows nodes to compose precise invalidations into imprecise ones, to incrementally exchange logs of mixed precise and imprecise invalidations, to allow precise reads (that see a *consistent* view of the data) or imprecise reads (that see only a *coherent* view of the data), and to recover precision for objects that have become imprecise.

1.2.2 UR-Repl: Universal Replication Mechanisms

Although the basic CR-Repl protocol covers a large design space and is efficient for some scenarios, it incurs significant overhead when it comes to implementing single object callbacks [42, 68], which are fundamental to many caching protocols, or supporting workloads where checkpoint exchange is a more efficient way for synchronization. To generalize the PRACTI mechanisms to a universal architecture, we extend the CR-Repl protocol in three fundamental ways to address these limitations:

1. In order to efficiently support both coarse-grained and fine-grained subscriptions, we *multiplex invalidation subscriptions* over a single network stream. By maintaining a shared state for multiple subscriptions and allowing one imprecise invalidation to be used across all active subscriptions, UR-Repl allows a node to dynamically subscribe or unsubscribe for invalidations to a large set of objects or to individual objects with the cost proportional to the total number of updates to the subscribed objects.

2. In order to support fast resynchronization of different type of workloads, UR-Repl introduces a novel *incremental checkpoint exchange* and smoothly integrates it with CR-Repl's log exchange protocol. Instead of freezing the receiver and sender's states as required in most existing checkpoint exchange protocols [72], UR-Repl allows the receiver to receive an incremental checkpoint for a small portion of its ID space and then either prefetch checkpoints of other interest sets or fault them in on demand. Therefore, UR-Repl is more efficient to support bandwidth-limited network environment than existing approaches.

3. In order to support efficient conflict detection for both log exchange and checkpoint exchange, we use novel *dependency summary vectors (DSV)* to detect write-write conflicts. This new write-write conflict detection algorithm yields three advantages. First, by having multiple objects to share the same version vector to detect conflicts, UR-Repl avoids sending and storing per-object version vector, thus saving network

bandwidth and storage space. Second, this conflict detection mechanism not only works for the log exchange protocol but also works for the checkpoint exchange protocol. Finally, we show how to efficiently implement the DSV using the state already required by the consistency maintenance algorithm—one version vector per interest set plus one version vector per network connection.

1.2.3 Case-Study Systems

A core hypothesis of URA is that a single framework can cleanly support a broad range of systems. To test whether our mechanisms capture the right abstractions and simplify the understanding and construction of replication systems, we first explain how to map a broad range of existing technologies such as callbacks, leases, and quorums. onto our underlying mechanisms and then combine a subset building blocks into series of case-study systems inspired by systems from the literature spanning a significant portion of the design space. These case-study systems include:

- Two client-server systems modeled on Coda [48] and TRIP [66] which illustrate how to map existing techniques such as *callbacks*, *leases*, *hoarding list sequential consistency*, and *disconnected operation* to URA;
- Two server replication systems modeled on Bayou [72] and Chain Replication [91] which demonstrate how to map existing techniques such as *primary server commit* (CSN) [72], *anti-entropy*, and *checkpoint exchange* to URA;
- Two object replication systems modeled on Pangaea [77] and TierStore [23] which show how to map existing techniques or features such as *gold nodes* [77], *best-effort coherence*, and *DTN support* to URA.

We also demonstrate how URA facilitates rapid evolution by adding significant features to several of these systems. For example, we add cooperative caching to U-Coda, our version of Coda, so that a clique of devices can share data even when disconnected from

the server; we add support for small devices to U-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 U-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 U-TRIP to improve scalability.

1.3 Contributions

This dissertation makes the following contributions:

- It defines the PRACTI paradigm and provides a taxonomy for replication systems that explains why existing replication architectures fall short of ideal.
- It describes the first replication protocol architecture to simultaneously provide all three PRACTI properties.
- It defines common abstractions of data replication systems that cleanly separate mechanism from policy and thereby simplify the understanding and construction of replication systems.
- It demonstrates that URA replication offers decisive practical advantages compared to existing approaches.
- It demonstrates the usefulness of URA by building several key case study applications and mapping existing techniques on URA.
- It proposes novel incremental checkpoint exchange, flexible commit primitive, and efficient conflict detection algorithms.

The rest of this dissertation is organized as follows: Chapter 2 defines the PRACTI properties and taxonomy to understand data replication design space, and it illustrates why it

is challenging to provide PRACTI properties in one single framework. This chapter closes with a discussion of the scope of URA and some issues that URA does not attempt to address. Chapter 3 gives an overview of the architecture and describes the simple abstractions URA exposes to system designers. Chapter 4 shows how to implement CR-Repl mechanisms and demonstrates the experiment results. Chapter 5 presents how to make CR-Repl mechanisms efficient by *multiplexing subscriptions*, adding *incremental checkpoint exchange*, and using *dependency summary vectors* for conflict detection. Chapter 6 describes how to map existing techniques to URA, and then Chapter 7 illustrates how to combine some of them to build a broad range of case-study systems. Chapter 8 summarizes related work. Finally, Chapter 9 concludes the dissertation.

Chapter 2

PRACTI Taxonomy, Challenges and Scope

In order to put the URA approach in perspective, this section defines the PRACTI properties, examines existing replication architectures, and considers why years of research exploring many different replication protocols have failed to realize PRACTI properties simultaneously. Finally we define the scope of the PRACTI paradigm and identify aspects of replication system design that are not within the scope of the PRACTI taxonomy on the URA protocol.

2.1 PRACTI Properties

Replication systems cover a large design space. Some cache objects on demand [42, 68], while others replicate all data to all nodes [72, 96]; some guarantee strong consistency [48, 91, 66] while others sacrifice consistency for higher availability [77, 23, 36]; some invalidate stale objects [42, 68], while others push updates [37]; some disseminate updates among nodes via a tree structure [23], while others synchronize data in an ad-hoc fashion [72]; and so on.

Informally, we can categorize replication policies into three families: *placement policies* such as demand-caching [42, 68], prefetching [35], push-caching [37], or replicate-all [72, 96] define which nodes store local copies of which data; *consistency policies* such as sequential [52] or causal [43] define which reads must see which writes; *topology policies* such as client-server [42, 68], hierarchy [12, 65], or ad-hoc [36, 50, 72] define the paths along which updates flow.

To cover the large design space of replication systems, any universal data replication architecture should provide all three PRACTI properties:

- *Partial replication (PR)* which allows any node to replicate any subset of data so that replication systems that make use of locality to improve performance [48] or that control placement to improve reliability [77] can be supported.

- *Any consistency (AC)* which allows a variety of consistency guarantees to be implemented so that systems that require weaker consistency like best-effort coherence and those which require stronger consistency like linearizability [40] can be supported.

- *Topology independence (TI)* which allows any node to synchronize with any other node so that systems with fixed update propagation topologies as well as those with dynamic, ad-hoc update topologies can be supported.

Although many existing systems can each provide two of these properties, we are aware of no system that provides all three. As a result, systems give up the ability to exploit locality, support a broad range of applications, or dynamically adapt to network topology.

Note that the requirements for supporting flexible consistency guarantees are subtle, and Chapter 6 discusses the full range of flexibility our protocol provides. URA should support both the weak coherence-only guarantees acceptable to some applications and the stronger consistency guarantees required by others. Note that *consistency* semantics constrain the order that updates across *multiple objects* become observable to nodes in the

system while *coherence* semantics are less restrictive in that they only constrain the order that updates to a *single object* become observable but do not additionally constrain the ordering of updates across multiple locations. (Hennessy and Patterson discusses the distinction between consistency and coherence in more detail [39].) For example, if a node $n1$ updates object A and then object B and another node $n2$ reads the new version of B, most consistency semantics would ensure that any subsequent reads by $n2$ see the new version of A, while most coherence semantics would permit a read of A to return either the new or old version.

2.2 PRACTI Taxonomy

The PRACTI paradigm defines a taxonomy for understanding the design space for replication systems as illustrated in Figure 2.1. As the figure indicates, many existing replication systems can be viewed as belonging to one of four protocol families, each of which provides at most two of the PRACTI properties.

Server replication systems like Replicated Dictionary [96] and Bayou [72] provide log-based peer-to-peer update exchange that allows any node to send updates to any other node (TI) and that consistently orders writes across all objects. Lazy Replication [50] and TACT [103] use this approach to provide a wide range of tunable consistency guarantees (AC). Unfortunately, these protocols fundamentally assume full replication: all nodes store all data from any volume they export and all nodes receive all updates. As a result, these systems are unable to exploit workload locality to efficiently use networks and storage, and they may be unsuitable for devices with limited resources.

Client-server systems like Sprite [68] and Coda [48] and *hierarchical* caching systems like hierarchical AFS [65] permit nodes to cache arbitrary subsets of data (PR). Although specific systems generally enforce a set consistency policy, a broad range of consistency guarantees are provided by variations of the basic architecture (AC) [99]. However, these protocols fundamentally require communication to flow between a child and

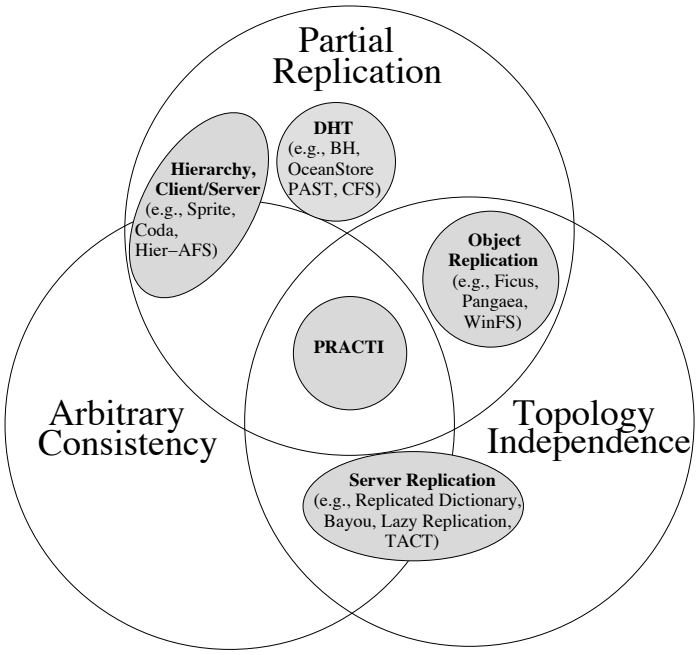


Figure 2.1: Replication system classification in the PRACTI taxonomy.

its parent. Even when systems permit limited client-client communication for cooperative caching [20, 25, 4], they must still serialize control messages at a central server for consistency [14]. These restricted communication patterns (1) hurt performance when network topologies do not match the fixed communication topology or when network costs change over time (e.g., in environments with mobile nodes), (2) hurt availability when a network path or node failure disrupts a fixed communication topology, and (3) limit sharing during disconnected operation when a set of nodes can communicate with one another but not with the rest of the system.

DHT-based storage systems such as BH [89], PAST [76], and CFS [17] implement a specific—if sophisticated—topology and replication policy: they can be viewed as generalizations of client-server systems where the server is split across a large number of nodes on a per-object or per-block basis for scalability and replicated to multiple nodes for availability and reliability. This division and replication, however, introduce new challenges for

providing consistency. For example, the Pond OceanStore prototype assigns each object to a set of primary replicas that receive all updates for the object, uses an agreement protocol to coordinate these servers for per-object coherence, and does not attempt to provide cross-object consistency guarantees [75].

Object replication systems such as Ficus [36], Pangaea [77], and WinFS [60] allow nodes to choose arbitrary subsets of data to store (PR) and arbitrary peers with whom to communicate (TI). But, these protocols enforce no ordering constraints on updates across multiple objects, so they can provide coherence but not consistency guarantees. Unfortunately, reasoning about the corner cases of consistency protocols is complex, so systems that provide only weak consistency or coherence guarantees can complicate constructing, debugging, and using the applications built over them. Furthermore, support for only weak consistency may prevent deployment of applications with more stringent requirements.

2.3 Why Is PRACTI Hard?

It is surprising that despite the disadvantages of omitting any of the PRACTI properties, no system provides all three. Our analysis suggests that these limitations are fundamental to these existing protocol families: the assumption of full replication is deeply embedded in the core of server replication protocols; the assumption of hierarchical communication is fundamental to client-server consistency protocols; careful assignment of key ranges to nodes is central to the properties of DHTs; and the lack of consistency is a key factor in the flexibility of object replication systems.

To understand why it is difficult for existing architectures to provide all three PRACTI properties, consider Figure 2.2's illustration of a naive attempt to add PR to a AC-TI server replication protocol like Bayou. Suppose a user's desktop node stores all of the user's files, including files A and B, but the user's palmtop only stores a small subset that includes B but not A. Then, the desktop issues a series of writes, including a write to file A (making it A') followed by a write to file B (making it B'). When the desktop and palmtop syn-

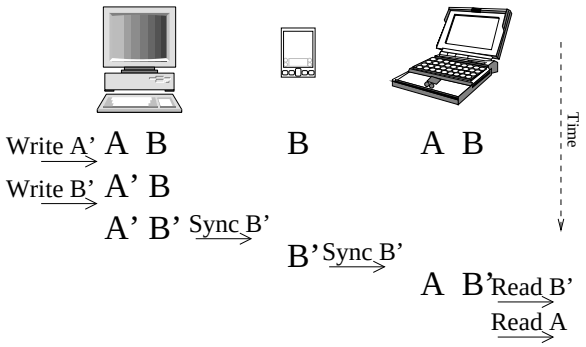


Figure 2.2: Naive addition of PR to AC-TI.

chronize, for PR, the desktop sends the write of B but not the write of A. At this point, everything is OK: the palmtop and desktop have exactly the data they want, and reads of local data provide a consistent view of the order that writes occurred. But for TI, we not only have to worry about local reads but also propagation of data to other nodes. For instance, suppose that the user's laptop, which also stores all of the user's files including both A and B, synchronizes with the palmtop: the palmtop can send the write of B but not the write of A. Unfortunately, the laptop now can present an inconsistent view of data to a user or application. In particular, a sequence of reads at the laptop can return the new version of B and then return the old version of A, which is inconsistent with the writes that occurred at the desktop under causal [43] or even the weaker FIFO consistency [54].

This example illustrates the broader, fundamental challenge: supporting flexible consistency (AC) requires careful ordering of how updates propagate through the system, but consistent ordering becomes more difficult if nodes communicate in ad-hoc patterns (TI) or if some nodes know about updates to some objects but not other objects (PR).

Existing systems resolve this dilemma in one of three ways. The full replication of AC-TI replicated server systems ensures that all nodes have enough information to order all updates. Restricted communication in PR-AC client-server and hierarchical systems ensures that the root server of a subtree can track what information is cached by descendants; the server can then determine which invalidations it needs to propagate down and which it

can safely omit. Finally, PR-TI object replication systems simply give up ability to consistently order writes to different objects and instead allow inconsistencies such as the one just illustrated.

2.4 Scope and Excluded Properties

A deeper challenge to designing a replication architecture is how to identify the essential characteristics of replication systems the architecture should encompass. Research papers discuss many features of their systems, and prototype implementations often include even more features not mentioned in papers. However, not all features are crucial to the architecture being advocated. Spending the time to create "bug compatible" implementations would detract from the central thrust of our research effort and would also likely confuse rather than clarify our results.

URA therefore focuses on supporting key features that relating to how replication systems maximize the performance, availability, and consistency to address the CAP [31] and PC [54] trade-offs. In particular, in Section 7.1.1, we formally define a *replication equivalence* notion based on *overhead*, *consistency* and *available local data*.

2.4.1 Excluded properties

These definitions restrict the scope of our architecture. 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 [9] to address the CAP [31] and PC [54] trade-offs. Mahajan et al. explore security issues for a simple protocol elsewhere [56].

Second, we do not systematically address the local interface a system exposes (e.g., file system [48, 77, 66], object store [22], tuple store [72], etc.) because we do not regard these differences as fundamental. URA 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 [48, 88, 47, 83, 22]. URA 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 [88].

Finally, we do not attempt to duplicate how systems are configured (e.g., specifying lists of peers or replication policy with configuration files [48] or symbolic links [62, 22]). 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.

Chapter 3

Architecture Overview

Given the PRACTI properties, URA's mechanisms can expose very simple abstractions for system designers to implement a broad range of systems. Before we describe how to address the challenge of providing the PRACTI properties, this chapter first gives an overview of the URA architecture by defining the basic replication abstractions URA expose to system designers and then describes the requirements to implement such abstractions. Chapter 4 and Chapter 5 describe how to implement these abstractions efficiently and correctly. Chapter 6 discusses how to map existing features to these abstractions and Chapter 7 describes how to build more sophisticated case-study systems.

3.1 Replication Abstractions

Figure 3.1 provides a high-level view of the URA architecture. To separate the mechanism and policy, the architecture is divided into two separate layers. The mechanism layer *core* is composed of a set of basic mechanisms that support PRACTI features. The policy layer *controller* implements specific system policies using the underlying common mechanisms through a set of replication interfaces. System designers can build different replication systems in the policy layer by constructing different controller instances.

The controller's major task is to allow system designers to specify replication policies to trigger the right communication between the right cores at the right time to do such things as satisfying a read miss, prefetching data to improve performance. In particular, to build a replication system, a system designer must specify two sets of replication policies.

- Policies for storage. This set of policies specify the consistency requirements and the data placement strategies of a system. For example, they answer questions such as "what consistency semantics" to enforce and "what objects to store locally on each node".
- Policies for communication. This set of policies specify the update distribution strategies including where to send invalidations, when to send bodies, and where to go for body etc.

As Figure 3.1 illustrates, to implement those replication policies, the URA core exposes two basic abstractions: storage and communication. A controller instance works with the core to construct a replication system through the *local API* and *communication API* to implement the storage and communication policies: a core informs the controller of important local events like message arrival or read miss, a controller calls a remote core's *communication API* to trigger transmissions of invalidation streams or bodies and calls a local core's *Local API* to enforce consistency guarantees. Additionally, a set of controllers implementing a specific distributed policy may communicate with one another using policy-specific interfaces.

To illustrate how the system works, let's look at how to deal with a read miss. When a core can not satisfy a local read request because the data is INVALID, it informs the controller and blocks the read thread until the data is VALID. The controller responses to the *inform* call by invoking a core demand read request to fetch the data from the core of a remote node it selects. When the body arrives, the local core applies it to make it VALID, unblocks the waiting read, and informs the controller so that the controller does not have to retry. Chapter 7 has more concrete illustration of the interactions between the controller

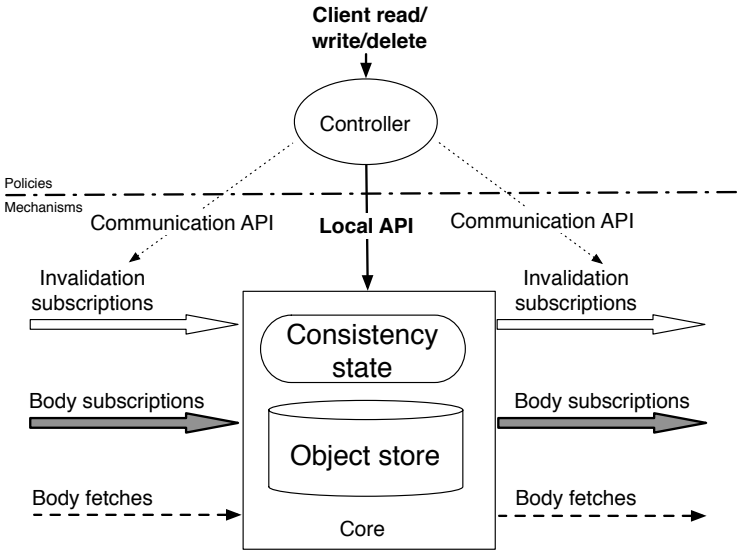


Figure 3.1: One node replication abstractions.

and the core.

3.1.1 Storage

The storage abstraction is simple and has two main parts: (1) an *object store* to store a subset of the system's data locally selected by the policy for serving local read/write requests and (2) *consistency states* to track and expose the dependencies among updates.

Object store. For simplicity, URA exposes a bare object-store interface for local reads and writes. It is straightforward to build other sophisticated higher-level interfaces such as a file system on top of it. For example, Chapter 6 will describe an NFS interface we implement using the basic object-store interface.

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* (IS) 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 consistency state information related to the object 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 sufficient information to derive the consistency state for all objects covered by an interest set.

Consistency state. To be universal, URA's consistency state must meet two requirements. First, it must be flexible to support a wide range of constraints. Second, it must be simple. The underlying system should handle details to maintain sufficient consistency bookkeeping information to make it easy for policy writers to specify high-level requirements.

URA defines two sets of primitives to implement the consistency state. First, it maintains a per-node logical vector clock and a per-node real-time vector clock to track the overall state of each node. These version vectors are updated when invalidations or heartbeats are received from other nodes. The logical vector clock is used to track the updates a node has learned of and is useful for supporting TACT's [103] *order error* (OE); the real-time vector clock is used to track objects' staleness and is useful for enforcing to support TACT's tunable *temporal error* (TE) and leases. We will discuss how to support those mechanisms in more detail in Chapter 6.

Second, by maintaining the per-object state and per-IS state, a node tracks 3 basic consistency states for each object that is included in its interest sets:

1. *VALID* tracks whether the node has data corresponding to the highest received invalidation for the target object. It is useful for enforcing coherence by blocking each read when the target object is invalid and for maximizing availability by ensuring that invalidations received from other nodes are not applied until they can be applied with their corresponding data [23, 66].

2. *PRECISE* tracks whether the object's local state reflects all updates before the node's current logical time. It is useful for enforcing cross object consistency. For example, to enforce causal consistency, a node blocks a read until the targeted object is *VALID* and *PRECISE*. This combination ensures that each read returns the newest update of its target object up to the node's current logical time, therefore they provide a consistent view across different objects.

   When an object is imprecise, it means that the node may not receive all updates targeting this object, which might lead to inconsistency. For example, suppose there are four updates $W0$ and $W2$ on $o1$ and $W1$ and $W3$ on $o2$ that are causally ordered as $W1 < W2 < W3 < W4$. Suppose a node α only receives $W0$ and $W3$, then α sees the old value of $o1$ and the new value of $o2$, which violates the causal consistency.

3. *SEQUENCED* tracks whether the current local version of the target object has been placed in a total order. It is useful for enforcing strong consistency semantics such as sequential consistency and linearizability that require commit protocols. For example, to support a primary commit protocol [72], the primary server can commit updates by setting the *SEQUENCED* status in the order invalidations arrive, and then each client blocks each read until the target object is *SEQUENCED*.

Local API. Given the range of consistency policies enabled by the basic object store and consistency state mechanisms, URA allows system designers to specify a consistency policy via a node's *local API*. As Figure 3.1 indicates, the local API operates on the local object store and consistency state and allows clients to read/write objects. A consistency policy defines the circumstances under which it is safe to process a read/write 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.

URAś local API therefore allows *blocking predicates* to block a read request, a write

isValid	Block until node has body corresponding to highest received invalidation for the target object, i.e., the target per-object state VALID is $true$.
isComplete	Block until object's consistency state reflects all updates before the node's current logical time, i.e., the target object state PRECISE is $true$.
isSequenced	Block until object's total order is established
propagated *nodes, count, p*	Block until $count$ nodes in $nodes$ have received my pth 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 message matching *tuple-spec*

Figure 3.2: Conditions available for defining consistency policies.

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 [103] and thereby implement a broad range of consistency models from best effort coherence to delta coherence [85] to causal consistency [51] to sequential consistency [52] to linearizability [103].

URA 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 3.2 lists the conditions available to consistency predicates. *isValid* if set $true$ blocks a read/write request until the per-object state is $VALID$. It is useful for enforcing coherence. *isComplete* and *isSequenced* are useful for enforcing consistency semantics like causal, sequential, or linearizable. *Propagated* and *maxStaleness* are based on the status

Local Operations	
Read	obj, off, len, blockUntil_VALID, blockUntil_PRECISE, blockUntil_SEQUENCED
Write	obj[], off[], len[], time, blockUntil_PRECISE, blockUntil_SEQUENCED
Delete	obj, time
AssignSequence	obj, version, time

Figure 3.3: Local storage access interface for policy writers.

of the per-node logical vector clock and real-time vector clock; they are useful for enforcing TACT order error and temporal error tunable consistency guarantees [103]. *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 policies produce a tuple matching a specified pattern.

For maximum flexibility, each read/write operation includes parameters to specify the consistency state semantics. Figure 3.3 lists the basic local read/write interfaces URA core exposes. The $blockUntil\_X$ parameters if specified will block the operations until the consistency state of the target object is X. Note that the $Write$ interface allows a write operation to atomically update one or more objects which is useful for implementing a file system interface and $AssignSequence$ interface is useful for implementing commit protocols such as Bayou's primary commit (CSN) protocol [72]. System designers 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 URA. For example, to enforce causal consistency, a system designer can add a wrapper that calls the basic read interface by setting the $blockUntil\_VALID$ and $blockUntil\_PRECISE$ to be $true$, i.e., put the *isValid* and *isPrecise* predicate to the read interface. Chapter 6 explains how to use these basic API and predicates to write wrappers for different consistency semantics.

3.1.2 Communication

Each update is distributed by two types of messages: an *invalidation* message that summarizes which object [1] the update targets and when the update occurs and a *body* message that contains the actual contents of the update. These messages are distributed in the system by setting up uni-directional invalidation subscriptions and body subscriptions/fetches between nodes.

Invalidation subscriptions. An invalidation subscription has two parameters: a start time *startVV* (a version vector) and a subscription set *SS* (a collection of object IDs) that together define the request: "send all invalidations for objects in SS that have occurred since startVV". Besides sending the invalidations for subscribed objects, the invalidation stream also delivers sufficient metadata information for objects outside the subscription set to allow the receiver to track dependencies among all updates.

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 byte range 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 and incremental log transfer.

Invalidation subscriptions can be used for coarse-grained replication of directory trees or volumes. For example, in a departmental file system, a user, Alice, can subscribe the subdirectory of */users/Alice* so as to get only the information about her files. Invalidation

[1] Our prototype API allows atomic multi-object write; for simplicity, we only describe the single-object invalidations.

Updates Distribution Interface	
Add Inval Sub	srcId, destId, objs, LOG|CP|CP+Body
Remove Inval Sub	srcId, destId, objects
Add Body Sub	srcId, destId, objs
Remove Body Sub	srcId, destId, objs
Send Body	srcId, destId, objId, off, len, time

Figure 3.4: Communication interface for policy writers.

subscriptions can also be used to support fine-grained *callbacks* mechanism commonly used in caching systems [48, 42, 68, 12, 65, 100]. For example, by creating an invalidation subscription with a subscription set composed of only o, a client can create a callback for object o on the server so that the server will notify the client whenever an update to o occurs.

Body subscriptions and body fetches. Whereas invalidation subscriptions make nodes aware of the remote updates that have occurred, body messages deliver the contents of these updates. Each body message identifies the invalidation with which it is associated and carries the contents of the corresponding update. Each node can request an individual body message or specify a subscription set *SS* and a start time *startVV* to subscribe for receiving the body messages for *SS* that have occurred since *startVV* from any other node.

Discussion. These simple communication abstractions are sufficient to cover the large design space summarized in [78]. In particular, they can be used to support arbitrary *distribution topology* including client-server structures like Coda, chain structures like Chain Replication, and Bayou style ad hoc topologies by simply setting up invalidation subscriptions and body subscriptions between nodes to form any desired structure. Arbitrary *synchronization schedules* can be enforced simply by orchestrating the start and stop of invalidation or body subscriptions. Regarding the *synchronization contents*, although the subscription abstraction separates the distribution of invalidation and body, distribution of *entire update* can be supported simply by setting up the invalidation stream the same way as the body

Connection Events	
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 Operation Events	
Read blocks	obj, off, len, EXIST|VALID|PRECISE|SEQUENCED
Write	obj, off, len, time
Delete	obj, time
Message Arrival events	
Inval arrives	sender, obj, off, len, time
Fetch success	sender, obj, off, len, time
Fetch failed	sender, receiver, obj, offset, length, time

Figure 3.5: Events exposed to policy writers.

stream and delaying applying an invalidation until the corresponding body arrives. Arbitrary *data placement* can be enforced by setting the subscription sets of invalidation streams and body streams.

Communication API. Figure 3.4 lists the communication APIs URA exposes for policy writers to set up subscriptions for any subset of updates between any pair of nodes at any time.

In order to provide sufficient information for policy writers to build sophisticated policies, besides the storage API and communication API, URA also exposes a set of event notification interface to the controller. As listed in Figure 3.5, these events include local operation events such as local read blocked and local write issued, connection events such as body subscription start and invalidation subscription failed, and message arrival events such as invalidation arrived or body fetch failed.

3.2 Requirements

To be a universal substrate for building different replication systems for different workloads or environments, the protocol should not only be able to support a wide variety of systems but also ensure that the cost a specific system implementation pays is proportional to the demands it has, i.e., it should limit the cost for generality. In particular, the implementation of the above abstractions needs to meet the following correctness and cost requirements:

- To support flexible consistency guarantees, the local storage and invalidation streams should at least preserve the causality of updates. As demonstrated in Chapter 6, maintaining causality is almost as cheap as coherence, and causality can be used as a basic building block for stronger consistency guarantees such as sequential consistency [51] and linearizability [40] and also for tunable consistency guarantees like TACT [103].

- To support small devices with limited storage capacity, a node's storage requirement should be proportional to the number of and size of the objects that the node is interested in.

- To support bandwidth-limited network connections, the overall bandwidth consumption for an invalidation stream should be proportional to the total number of updates to the objects subscribed.

Although the basic abstractions are simple, the challenge is how to implement them to meet the correctness and cost requirements. For example, to support traditional per-object callbacks, the network bandwidth cost should be proportional to the number of updates to these objects that have callbacks, therefore the protocol must omit the information about other updates. However to support causal consistency, we have to send sufficient metadata information about updates to other objects as well. In the next two chapters, we present protocols that implement these abstractions and meet these goals.

Chapter 4

CR-Repl: Coarse-grain PRACTI Replication

We implement the replication abstractions described in the previous chapter in two steps.

First, we implement a novel peer-to-peer replication protocol CR-Repl that realizes the invalidation/body subscription abstractions and the storage abstractions. In particular, this protocol provides the following key properties:

- It provides all the three PRACTI properties described in Chapter 2: (1) *Partial Replication*—it allows any node to replicate any subset of data and metadata; (2) *Any Consistency*—it provides both strong and weak consistency guarantees so that only applications that require strong guarantees pay for them; and (3) *Topology Independence*—it allows any node to exchange updates with any other node.
- It implements a novel log maintenance algorithm to efficiently store and form imprecise invalidations.
- It enables *self-tuning body propagation* for efficient prefetching.
- It provides *incremental log exchange* to allow systems to minimize the window for conflicting updates.

- It provides a conflict detection mechanism based on a per-update *previous stamp*.

The CR-Repl protocol has three key limitations. First, although the use of imprecise invalidations enables efficient coarse-grain invalidation subscriptions, it falls short to support dynamic fine-grained invalidation subscriptions. Second, the incremental log exchange protocol is inefficient for some workloads where checkpoint exchange is more efficient. Finally, although it supports a simple write-write conflict detection mechanism, doing so adds an extra per-update stamp overhead for both storage and network bandwidth, and the conflict detection protocol may not work properly when the log is truncated.

Therefore, at the second step, we implement a universal replication protocol UR-Repl to fix these limitations. This protocol supports four key features: (1) *incremental log exchange and checkpoint exchange* which allows a node to receive an incremental prefix of updates or an incremental checkpoint to support fast resynchronization of subsets of data and incremental progress despite network disruptions; (2) *efficient support for both coarse-grained and fine-grained invalidation subscriptions* which allows a node to dynamically subscribe or unsubscribe for invalidations to a large set of objects or to individual objects from any other node with the cost proportional to the total number of updates to the subscribed objects; (3) *efficient conflict detection* which allows nodes to accurately detect conflicting writes during either log exchange or checkpoint exchange with reasonable overheads so that it can support large-scale data replication systems. ; and (4) *flexible commit mechanisms* which implement the *SEQUENCED* consistency state without requiring any rollback as is required in Bayou [72] and enable a range of commit protocols by allowing system designers to control when and under what circumstances to commit a write.

This chapter focuses on the description of the CR-Repl protocol, and we leave the description of the UR-Repl protocol in the next Chapter. The rest of this chapter first gives an overview of the basic data structures and key ideas of CR-Repl, and then describes the key ideas in more details in Section 4.2 and Section 4.3. After that, Section 4.4 presents the special log maintenance algorithm and Section 4.5 describes other novel features of

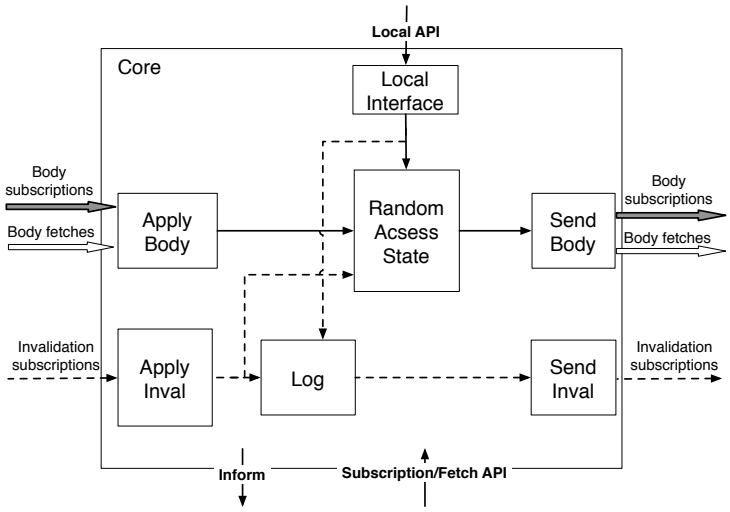


Figure 4.1: Core data structures.

our prototype that enable it to support the broadest range of policies. Finally, Section 4.6 experimentally evaluates the prototype.

4.1 Implementation Overview

Figure 4.1 illustrates the main local data structures of each node's core to implement the abstractions described in Chapter 3. Each core maintains a *Log* and a *Random Access State* (RAS). The log is used to store updates for future potential update exchange. The RAS implements the storage abstractions including the *object store* and the *consistency state*. Applications access data stored in the local core via the per-node *Local API* for creating, reading, writing, deleting, and sequencing objects. These functions operate the local node's log and RAS: modifications are appended to the log and then update the RAS, and reads access the RAS. To support partial replication policies, the mechanisms allow each node to select an arbitrary subset of the system's objects to store locally, and nodes are free to change this subset at any time (e.g., to implement caching, prefetching, hoarding, or replicate-all).

As described in Chapter 3, to handle read misses and to exchange updates between nodes, URA cores use two types of communication—causally ordered *Streams of Invalidations* and unordered *Body* messages. To implement these communication abstractions, the URA core includes four communication modules as illustrated in the figure: *ApplyBody* and *ApplyInval* modules for processing incoming messages, *SendBody* and *SendInval* modules for assembling and transmitting outgoing messages.

The protocol for sending streams of invalidations is similar to Bayou's [72] log exchange protocol, and it ensures that each node's log and RAS always reflect a causally consistent view of the system's data. But it differs from existing log exchange protocols in two key ways:

1. *Separation of invalidations and bodies.* Invalidation streams notify a receiver that writes have occurred, but separate body messages contain the contents of the writes. A core coordinates these separate sources of information to maintain local consistency invariants. This separation supports partial replication of data—a node only needs to receive and store bodies of objects that interest it.

2. *Imprecise invalidations.* Although the invalidation streams each logically contain a causally consistent record of all writes known to the sender but not the receiver, nodes can omit sending groups of invalidations by instead sending *imprecise invalidations*. Whereas traditional *precise invalidations* describe the target and logical time of a single write, an imprecise invalidation can concisely summarize a set of writes over an interval of time across a set of target objects. Thus, a single imprecise invalidation can replace a large number of precise invalidations and thereby support partial replication of metadata—a node only needs to receive traditional precise invalidations and store per-object metadata for objects that interest it.

   Imprecise invalidations allow nodes to maintain consistency invariants despite partial replication of metadata and despite topology independence. In particular, they serve as placeholders in a receiver's log to ensure that there are no causal gaps in the log

a node stores and transmits to other nodes. Similarly, just as a node tracks which objects are *INVALID* so it can block a read to an object that has been invalidated but for which the corresponding body message has not been received, a node tracks which sets of objects are *IMPRECISE* so it can block a read to an object that has been targeted by an imprecise invalidation and for which the node therefore may not know about the most recent write.

In the next two subsections we first detail how the protocol separates invalidations and bodies and then describe how it implements imprecise invalidations.

4.2 Separation of Invalidations and Bodies

As just described, nodes maintain their local state by exchanging two types of updates: ordered streams of invalidations and unordered body messages. *Invalidations* are metadata that describe writes; each contains an object ID[1] and logical time of a write (a.k.a. accept stamp). A write's logical time *acceptStamp* is assigned at the local interface that first receives the write, and it contains the current value of the node's Lamport clock [51] and the node's ID. Like invalidations, *body messages* contain the write's object ID and logical time, but they also contain the actual contents of the write.

The protocol for exchanging updates is simple.

- As illustrated in Figure 4.1, each node maintains a *log* of the invalidations it has received sorted by logical time. And, for random access, each node stores bodies in the RAS indexed by object ID.

- Invalidations from a log are sent via a causally-ordered stream that logically contains all invalidations known to the sender but not to the receiver. As in Bayou, nodes use

[1] For simplicity, we describe the protocol in terms of full-object writes. For efficiency, our implementation actually tracks per-object state, invalidations, and bodies on arbitrary byte ranges.

version vectors to summarize the contents of their logs in order to efficiently identify which updates in a sender's log are needed by a receiver [72].

- A receiver of an invalidation inserts the invalidation into its sorted log and updates its RAS. RAS update of the entry for object ID entails marking the entry *INVALID* and recording the logical time of the invalidation. Note that RAS update for an incoming invalidation is skipped if the RAS entry already stores a logical time that is at least as high as the logical time of the incoming invalidation.

- A node can send any body from its RAS to any other node at any time. When a node receives a body, it updates its RAS entry by first checking to see if the entry's logical time matches the body's logical time and, if so, storing the body in the entry and marking the entry *VALID*.

4.2.1 Rationale

Separating invalidations from bodies provides topology-independent protocol that supports both arbitrary consistency and partial replication.

Supporting arbitrary consistency requires a node to be able to consistently order all writes. Log-based invalidation exchange meets this need by ensuring three crucial properties [72]. First the *prefix property* ensures that a node's state always reflects a prefix of the sequence of invalidations by each node in the system, i.e., if a node's state reflects the *i*th invalidation by some node *n* in the system, then the node's state reflects all earlier invalidations by *n*. Second, each node's local state always reflects a *causally consistent* [43] view of all invalidations that have occurred. This property follows from the prefix property and from the use of Lamport clocks to ensure that once a node has observed the invalidation for write *w*, all of its subsequent local writes' logical timestamps will exceed *w*'s. Third, the system ensures *eventual consistency*: all connected nodes eventually agree on the same total order of all invalidations. This combination of properties provides the basis for a broad range of tunable consistency semantics using standard techniques [103].

At the same time, this design supports partial replication by allowing bodies to be sent to or stored on any node at any time. It supports arbitrary body replication policies including demand caching, push-caching [37, 77, 94], prefetching [35], hoarding, prepositioning bodies according to a global placement policy [93], or push-all [72].

4.2.2 Design Issues

The basic protocol adapts well-understood log exchange mechanisms [72, 96]. But, the separation of invalidations and bodies raises two new issues: (1) coordinating disjoint streams of invalidations and bodies and (2) handling reads of invalid objects.

The first issue is how to coordinate the separate body messages and invalidation streams to ensure that the arrival of out-of-order bodies does not break the consistency invariants established by the carefully ordered invalidation log exchange protocol. The solution is simple: when a node receives a body message, it does not apply that message to its RAS until the corresponding invalidation has been applied. A node therefore buffers body messages that arrive "early." As a result, the RAS is always consistent with the log, and the flexible consistency properties of the log [103] extend naturally to the RAS despite its partial replication.

The second issue is how to handle demand reads at nodes that replicate only a subset of the system's data. The core mechanism supports a wide range of policies: by default, the system blocks a local read request until the requested object's status is *VALID*. Of course, to ensure liveness, when an *INVALID* object is read, an implementation should arrange for someone to send the body. Therefore, when a local read blocks, the core notifies the controller. The controller can then implement any policy for locating and retrieving the missing data such as sending the request up a static hierarchy (i.e., ask your parent [12] or a central server [42]), querying a separate centralized [26] or DHT-based [89] directory, using a hint-based search strategy [81], or relying on a push-all strategy [72, 96] (i.e., just wait and the data will come.)

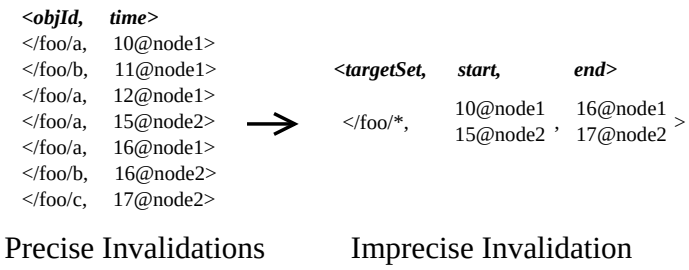


Figure 4.2: Imprecise invalidation example.

4.3 Partial Replication of Invalidations

Although separation of invalidations from bodies supports partial replication of bodies, for true partial replication the system must not require all nodes to see all invalidations or to store metadata for each object. Exploiting locality is fundamental to replication in large-scale systems, and requiring full replication of metadata would prevent deployment of a replication system for a wide range of environments, workloads, and devices. For example, consider palmtops caching data from an enterprise file system with 10,000 users and 10,000 files per user: if each palmtop were required to store 100 bytes of per-object metadata, then 10GB of storage would be consumed on each device. Similarly, if the palmtops were required to receive every invalidation during log exchange and if an average user issued just 100 updates per day, then invalidations would consume 100MB/day of bandwidth to each device.

The implementation of invalidation subscriptions is complicated by the following two restrictions. On one hand, an invalidation stream $(SS, startVV)$ is supposed to only send updates for objects in SS. Therefore the cost needs to be proportional to the number of invalidations to those objects. On the other hand, to maintain cross-object causal consistency, invalidation streams need to send information for updates to objects outside of SS. Even if a node never looks at objects outside of SS and can tolerate not seeing those updates, it could relay to another node updates about SS but not updates about objects outside

of SS on which those updates depend and thereby lead to violations of consistency.

To support true partial replication and meet the efficiency and consistency requirements, CR-Repl invalidation streams *logically* contain all invalidations as described in Section 4.2, but in *reality* they omit some by replacing them with *imprecise invalidations.*

Imprecise invalidations. As Figure 4.2 illustrates, an imprecise invalidation is a conservative summary of several standard or *precise invalidations.* Each imprecise invalidation has a *targetSet* of objects, *start* logical time, and an *end* logical time, and it means "one or more objects in *targetSet* were updated between *start* and *end*." An imprecise invalidation must be *conservative*: each precise invalidation that it replaces must have its *objId* included in *targetSet* and must have its logical *time* included between *start* and *end*, but for efficient encoding *targetSet* may include additional objects. In our prototype, the *targetSet* is encoded as a list of subdirectories and the *start* and *end* times are partial version vectors with an entry for each node whose writes are summarized by the imprecise invalidation. Finally, note that a standard precise invalidation is simply a special case of an imprecise invalidation with a single-object *targetSet*, single-entry *start* and *end* times, and $start = end$.

A node reduces its bandwidth requirements by subscribing to receive precise invalidations only for desired subsets of data and receiving imprecise invalidations for the rest. And a node saves storage by tracking per-object state only for desired subsets of data and tracking coarse-grained bookkeeping information for the rest.

4.3.1 Forming Imprecise Invalidations

URA forms an imprecise invalidation I by combining generalized invalidations GI_1 and GI_2. I has *start* and *end* arrays with entries for every node η in either GI_1 or GI_2's *start*, and $I.start_\eta = min(GI_1.start_\eta, GI_2.start_\eta)$, and $I.end_\eta = max(GI_1.end_\eta, GI_2.end_\eta)$. Finally, $I.targetSet$ encompasses all objects encompassed by GI_1 and GI_2's *targetSet*s.

When a controller asks node α to send a stream of invalidations to node β, the controller specifies two parameters that each filter the transmitted information: a start time

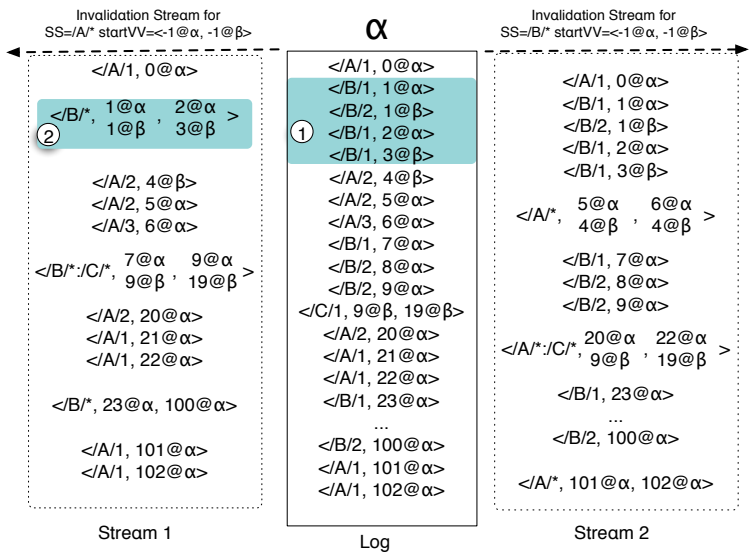


Figure 4.3: Invalidation streams with imprecise invalidations.

$startVV$ to provide a filter on logical time and a subscription set SS to provide a filter on the ID space. α replies with a causally consistent stream of all invalidations it knows about that logically occurred after $startVV$. Invalidations whose target intersects SS are sent as is (typically they are precise, but some may be imprecise), but α combines other invalidations into imprecise summaries as just described. This process is incremental and continuous—as new invalidations arrive at α, α sends them on to β once all causally prior invalidations have been sent.

Examples. Figure 4.3 gives two examples of invalidation streams. Node α has received a list of causally ordered invalidations as indicated in the second box. When a node subscribes for $/A/*$ from $startVV = \langle -1@\alpha, -1@\beta \rangle$, the invalidation stream consists of precise invalidations for $/A/*$ and imprecise invalidations for $/B/*$ and $/C/*$ and they are sent in causal order as indicated in *Stream 1*. For example, the list of precise invalidations for $/B/*$ in ①are replaced by one imprecise invalidation ②. Similarly, when the node subscribes for $/B/*$, α sends all invalidations of $/B/*$ as they are and combines invalidations for $/A/*$

and $/C/*$ into imprecise invalidations as indicated in *Stream 2*.

4.3.2 Applying Imprecise Invalidations

In the previous section, we define how a node creates and sends an invalidation stream; the rest of this section details how our implementation copes with maintaining local state as nodes receive invalidation streams by (a) applying invalidations in causal order despite the multiple start and end times in imprecise invalidations and despite concurrency across streams and (b) maximizing the information extracted and stored from each invalidation in a stream to minimize the amount of *IMPRECISE* data a node stores locally and to minimize the scope of imprecise invalidations propagated to other nodes.

When a node receives an imprecise invalidation *I*, it inserts *I* into its log and updates its RAS. For the log, imprecise invalidations act as placeholders to ensure that the omitted precise invalidations do not introduce causal gaps in the log that a node stores locally or in the streams of invalidations that a node transmits to other nodes.

Design issues. Tracking the effects of imprecise invalidations on a node's RAS must address four related problems:

1. For consistency, a node must *logically* mark all objects targeted by a new imprecise invalidation as *INVALID*. This action ensures that if a node tries to read data that may have been updated by an omitted write, the node can detect that information is missing and block the read until the missing information has been received.

2. For liveness, a node must be able to unblock reads for an object once the per-object state is brought up to date (e.g., when a node receives the precise invalidations that were summarized by an imprecise invalidation.)

3. For space efficiency, a node should not have to store per-object state for all objects. As the example at the start of this subsection illustrates, doing so would significantly

restrict the range of replication policies, devices, and workloads that can be accommodated.

4. For processing efficiency, a node should not have to iterate across all objects encompassed by *targetSet* to apply an imprecise invalidation.

Interest set status. To meet these requirements, rather than track the effects of imprecise invalidations on individual objects, nodes keep bookkeeping information on groups of objects called *Interest Sets*. In particular, each node independently partitions the object ID space into one or more interest sets and decides whether to store per-object state on a per-interest set basis. A node tracks whether each interest set is *PRECISE* (per-object state reflects all invalidations) or *IMPRECISE* (per-object state is not stored or may not reflect all precise invalidations) by maintaining two pieces of state.

- Each node maintains a global variable *currentVV*, which is a version vector encompassing the highest timestamp of any invalidation (precise or imprecise) applied to any interest set.

- Each node maintains for each interest set *IS* the variable *IS.lpVV*, which is the latest version vector for which *IS* is known to be *PRECISE*.

If *IS.lpVV* = *currentVV*, then interest set *IS* has not missed any invalidations and it is *PRECISE*. Otherwise, the interest set may have missed on or more precise invalidations, and we regard the interest set as *IMPRECISE*.

In this arrangement, applying an imprecise invalidation *I* to an interest set *IS* merely involves updating two variables—the global *currentVV* and the interest set's *lpVV*. In particular, a node that receives imprecise invalidation *I* always advances *currentVV* to include *I*'s *end* logical time because after applying *I*, the system's state may reflect events up to *I.end*. Conversely, the node only advances *IS.lpVV* to the latest time for which *IS* has missed no invalidations.

```
1: // Global state:
2: //   currentVV – node's current version vector
3: //   IS.lpVV – IS's last precise version vector
4: //   RAS_obj – per-object state
5: //   log – replay log
6: // Per-stream state:
7: //   stream = startVV, GI_1, GI_2, ...
8: //   GI – next generalized invalidation to apply
9: //   prevVV – logical time before next GI applied
10: //   pending – set of GI's whose end time has not passed
11:
12: Procedure ProcessInvalStream(IS, stream)
13: prevVV = stream.readObj()
14: if ! includes(currentVV, prevVV)
15:    return; // Reject streams that do not preserve prefix property
16: pending = new Set()
17: GI = stream.readObj()
18: while (GI ≠ EOF) do
19:    nextStartVV =advanceToInclude(prevVV, GI.start)
20:    if !(∃bufferedInval ∈ pending |includes(nextStartVV,bufferedInval.end))
21:       log.insert(GI, prevVV)
22:       // Update interest set status
23:       currentVV =advanceToInclude(currentVV, GI.end) // update (1)—see text
24:       if includes(IS.lpVV, prevVV)   // If no gaps, update lpVV
25:          if GI.isPrecise()  // Advance to include precise inval
26:             IS.lpVV =advanceToInclude(IS.lpVV, GI.start) // update (2)
27:          else // Advance to just before imprecise inval
28:             IS.lpVV =advanceNoInclude(IS.lpVV, GI.start) // update (3)
29:       // Update per-object state
30:       if GI.isPrecise()
31:          RAS_GI.objId.valid = INVALID
32:          RAS_GI.objId.accept = GI.start
33:       pending.insert(GI) // Apply to non-overlapping later
34:       prevVV = nextStartVV // Update stream logical time
35:       GI = stream.readObj()
36:    else // Apply non-overlapping bufferedInval from pending at end time
37:       if !(bufferedInval.target intersects IS)
38:          if includes(lpVV, prevVV)
39:             IS.lpVV =advanceToInclude(IS.lpVV,bufferedInval.endVV) // update (4)
40:       pending.remove(bufferedInval)
```

Figure 4.4: Stream processing algorithm for interest set IS with $stream = \{startVV, GI_1, GI_2, \ldots\}$

```
 1: Procedure advanceToInclude(VV1, VV2)
 2: for all nodeId do
 3:    retVV_nodeId = max(VV1_nodeId, VV2_nodeId)
 4: return retVV
 5:
 6: Procedure advanceNoInclude(VV1, VV2)
 7: for all nodeId do
 8:    retVV_nodeId = max(VV1_nodeId, VV2_nodeId − 1)
 9: return retVV
10:
11: Procedure includes(VV1, VV2)  // Does VV1 include VV2?
12: for all nodeId do
13:    if VV2_nodeId > VV1_nodeId
14:       return false
15: return true
```

Figure 4.5: Utility functions for **ProcessInvalStream**.

Algorithm details. One of the most intellectually challenging parts of our effort in developing the CR-Repl prototype was to get the imprecise invalidation processing precisely right. Figure 4.4 and Figure 4.5 detail the algorithm for processing an incoming stream of invalidations. As indicated in Figure 4.4 line 7, each incoming invalidation stream consists of a logical start time $startVV$ followed by a series of general invalidations GI_1, GI_2, ... such that any invalidation whose start time logically occurs after $startVV$ and on which GI_i causally depends appears before GI_i. A generalized invalidation GI_i can be either a precise invalidation or an imprecise invalidation.

For each general invalidation GI, the log, the per-object state, and the interest set status must be updated. Updating the per-object state (lines 29 to 32) was described in Section 4.2, and we will discuss updating the log (line 21) in Section 4.4. The remaining issue is updating the per-interest set *PRECISE* state (lines 22 to 24 and lines 37 to 40), i.e., updating $currentVV$ and one or more $lpVV$s.

At the core of the algorithm is a simple idea: an interest set is *PRECISE* if it has missed no precise invalidations. To track an interest set IS's state, besides the two variables $currentVV$ and $IS.lpVV$ as described above, the receiver also tracks a **per-stream** version vector $prevVV$ that always holds the logical time just *before* the next invalidation

Update Number	Code Line	When	IS state	GI type	GI intersects IS	Action
(1)	23	$GI.start$	ANY	ANY	ANY	Advance cVV to include $GI.end$
(2)	26	$GI.start$ (= $GI.end$)	PRECISE	PRECISE	ANY	Advance $IS.lpVV$ to include $GI.end$ (= $GI.start$)
(3)	28	$GI.start$	PRECISE	IMPRECISE	ANY	Advance $IS.lpVV$ to just before $GI.start$
(4)	39	$GI.end$	PRECISE	IMPRECISE	NO	Advance $IS.lpVV$ to include $GI.end$

Figure 4.6: Summary of cases for updating interest set PRECISE/IMPRECISE status.

in the stream is applied. Each invalidation GI is processed in the context of the logical time $stream.prevVV$ at which it was applied to determine if GI can advance $IS.lpVV$. $stream.prevVV$ is initialized to the stream's $startVV$ (line 13) and advanced to include $gi.end$ as each GI is processed (line 19 and line 34).

The interest set status information is updated in four places as summarized in Figure 4.6. The first three updates occur when GI is first encountered in the stream, i.e., when it is known that there is no event that is causally after $stream.prevVV$ and causally before GI. The fourth occurs at $GI.end$, i.e., when it is known that no remaining GI_i in the stream contains any event that causally occurs before $GI.end$.

When GI is first encountered in the stream, we always advance $currentVV$ to include the *end time* of GI because the system now reflects information in GI (update number 1 in the table, line 23 in the pseudo-code). Further, due to the prefix property, GI's presence in the causal invalidation stream means that any interest set that was *PRECISE* before GI is still *PRECISE* to $GI.start$. So, if interest set IS was *PRECISE* at time $stream.prevVV$, then we advance $IS.lpVV$.

We advance $IS.lpVV$ differently depending on whether GI is a precise or imprecise invalidation. If GI is precise, then there have been no imprecise invalidations between $stream.prevVV$ and $GI.start$, and we advance $IS.lpVV$ to include $GI.end$ (note: $GI.start = GI.end$ if GI is precise.) That case is update number 2 in the table and line 26 in the pseudo-code. Conversely, if GI is imprecise, we can only advance $IS.lpVV$ to just before $GI.start$ (i.e., $\forall \alpha : IS.lpVV_\alpha = max(IS.lpVV_\alpha, GI.start_\alpha - 1)$). That case

is update number 3 in the table and line 28 in the pseudo-code.

Two points should be emphasized:

- Notice that when there is a gap in the logical time sequence for a given node, $GI.start$ may exceed $IS.lpVV$ even though no invalidations were skipped. This is why we maintain $prevVV$ for each stream and why line 24 compares $IS.lpVV$ against $prevVV$ rather than against $GI.start$ when deciding whether it is safe to advance $IS.lpVV$.

- Notice that an imprecise invalidation GI will always advance $currentVV$ to include GI's *end* time but can at most advance $IS.startVV$ to just before GI's *start* time. It is this difference that causes imprecise invalidations to make interest sets *IMPRECISE*.

If we stopped here, an imprecise invalidation would make both interest sets it overlaps and interest sets it does not overlap *IMPRECISE*. The algorithm addresses this issue by buffering each imprecise invalidation after it is first applied at its start time and applying a buffered invalidation *bufferedInval* again once $stream.prevVV$ includes *bufferedInval*'s end time (i.e., once all GIs whose start times precede *bufferedInval*'s end time have been processed.) Applying *bufferedInval* advances $IS.lpVV$ to include *bufferedInval*$.end$ for any interest set IS that (a) *bufferedInval*$.targetSet$ does *not* intersect and that (b) is *PRECISE* as of logical time $stream.prevVV$. This case is update number 4 in the table and line 39 in the code. Notice that by waiting until *bufferedInval*'s end time before advancing "nonoverlapping" invalidations to the end time, we avoid erroneously advancing $lpVV$ for an interest set that becomes *IMPRECISE* between *bufferedInval*$.start$ and *bufferedInval*$.end$.

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

Summary. This algorithm meets the four requirements listed above.

1. By default, a read request blocks until the interest set in which the object lies is *PRECISE* and the object is *VALID*. This blocking ensures that reads only observe the RAS state they would have observed if all invalidations were precise and therefore allows nodes to enforce the same consistency guarantees as protocols without imprecise invalidations.

2. For liveness, the system must eventually unblock waiting reads. The core signals the controller when a read of an *IMPRECISE* interest set blocks, and the controller is responsible for arranging for the missing precise invalidations to be sent. When the missing invalidations arrive, they advance *IS.lpVV*. The algorithm for processing invalidations as described above guarantees that any interest set *IS* can be made *PRECISE* by receiving a sequence *S* of invalidations from *IS.lpVV* to *currentVV* if *S* is causally sorted and includes all precise invalidations targeting *IS* in that interval.

3. Storage is limited: each node only needs to store per-object state for data currently of interest to that node. Thus, the total metadata state at a node is proportional to the number of objects of interest plus the number of interest sets. Note that our implementation allows a node to dynamically repartition its data across interest sets as its locality patterns change.

4. Imprecise invalidations are efficient to apply, requiring work that is proportional to the number of interest sets at the receiver rather than the number of summarized invalidations.

Example. The example in Figure 4.7 illustrates the maintenance of interest set state. Initially, (1) interest set *IS* is *PRECISE* and objects *A*, *B*, and *C* are *VALID*. Then, (2) an imprecise invalidation *I* arrives. *I* (3) advances *currentVV* but not *IS.lpVV*, making *IS IMPRECISE*. But then (4) precise invalidations *PI1* and *PI2* arrive on a single invalidation

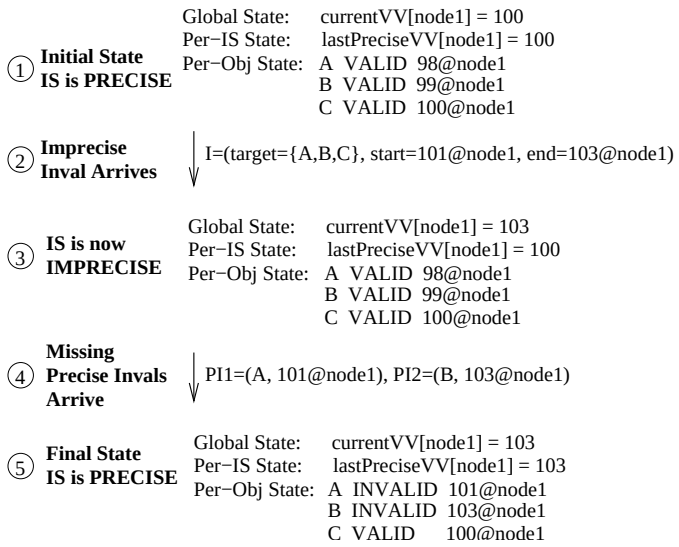


Figure 4.7: Example of maintaining interest set state. For clarity, we only show $node1$'s elements of *currentVV* and *lpVV*.

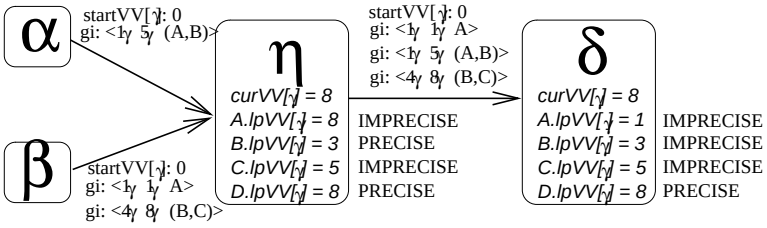
channel from another node. (5) These advance *IS.lpVV*, and in the final state *IS* is *PRECISE*, *A* and *B* are *INVALID*, and *C* is *VALID*.

Notice that although the node never receives a precise invalidation with time *102@node1*, the fact that a single incoming stream contains invalidations with times *101@node1* and *103@node1* allows it to infer by the prefix property that no invalidation at time *102@node1* occurred, and therefore it is able to advance *IS.lpVV* to make *IS PRECISE*.

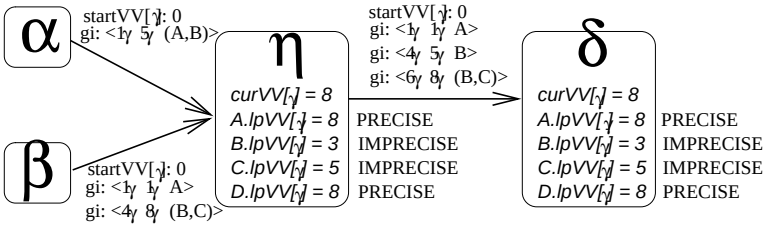
4.4 Log Maintenance

Each node stores the invalidations it generates or receives from other nodes in a log for future potential invalidation subscriptions with other nodes. One of the main goals of log maintenance is to maximize the information extracted and stored from each invalidation in a stream to minimize the amount of *IMPRECISE* data a node stores locally and to minimize the scope of imprecise invalidations propagated to other nodes.

Imprecise invalidations complicate log updates. For example, a node η may receive



(a) Naive log exchange.



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

Figure 4.8: Log exchange example. Node η first receives a log from α, then receives a log from β, and then sends the combined log to δ. Imprecise invalidations have three fields: $\langle start, end, targetSet \rangle$. Note that all writes were issued by node γ and, for clarity, we show only γ's component for all version vectors.

different subsets of information from different peers α and β. η must ensure that imprecise invalidations received from α do not "mask" precise invalidations received from β and vice versa. Notice that the algorithm just described updates a node's local state by interpreting each invalidation relative to the per-stream $prevVV$, which allows the algorithm to infer that there are no missing invalidations between $stream.prevVV$ and the invalidation. But, if η were simply to store each invalidation in its log, some of this valuable "no missing invalidations" information could be lost. Then, as Figure 4.8-(a) illustrates, if η were to send its log to some other node δ, then even if δ receives the same invalidations as η, δ could end up *IMPRECISE* where η is *PRECISE* (e.g., for object A) as indicated in the colored boxes in Figure 4.8-(a). Another problem with the naive approach is that it sends redundant invalidations which overlap some time intervals.

In order to ensure that a node can transmit all information received including both the generalized invalidations and the information implicit in the incoming invalidation

stream and only sends one invalidation for each time point, we augment our logs in three ways.

First, each node maintains a single on-disk append-only replay log in which invalidations are stored in the order they are received. Additionally each node maintains separate *per-writer logs*: when a node inserts an imprecise invalidation II into its log, it first appends II to the on-disk log and decomposes II into per-writer general invalidations and then inserts the per-writer pieces into separate logs. Decomposing II into per-writer general invalidations II_α is simple: for each server α in $II.start$, generate II_α with $start = II.start_\alpha$, $end = II.end_\alpha$, and $target = II.target$. Note that precise invalidation PI can be treated as an imprecise invalidation with $start = PI.acceptStamp$ and $end = PI.acceptStamp$, and it is already a per-writer invalidation.

Second, each per-writer log uses *gap filling* to explicitly encode the knowledge that each incoming stream is causally consistent and is therefore FIFO consistent for each writer. In particular, each per-writer log maintains the invariant that there is no gap between the end time of an element and the start time of the next element. When a node inserts II_α into its per-writer log for α, if II_α is newer than the newest element in the log, it fills any gap between $II_\alpha.start$ and the existing element by inserting a new gap-filling invalidation with a start stamp one larger than the highest existing end stamp, an end stamp one smaller than $II_\alpha.start$, and an empty target.

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

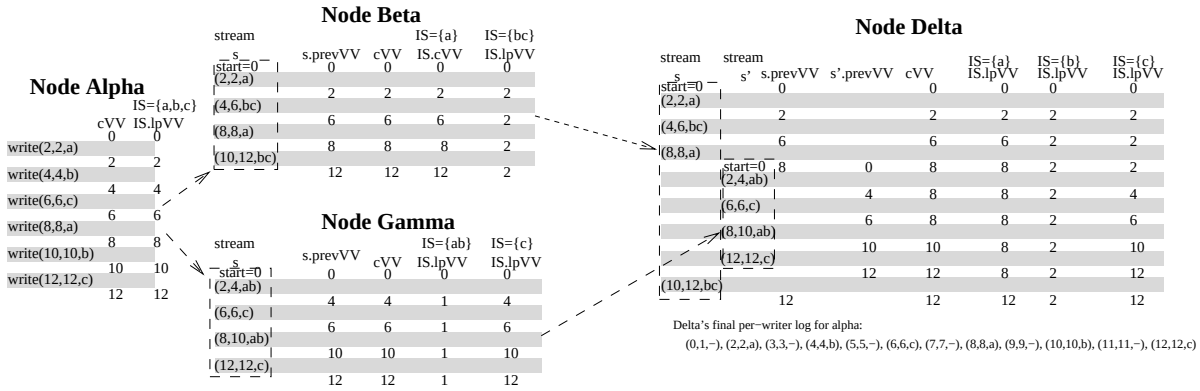


Figure 4.9: Illustration of imprecise invalidation mechanisms in *split-join* scenario. Nodes α, β, γ, and δ share objects a, b, and c. At each node, we show the per-interest-set information (last precise version vector $lpVV$ and current version vector cVV), the per-invalidation-stream information ($startVV$ and a series of generalized invalidations), and the per-interest-set per-stream information ($prevVV$ as it is updated as each generalized invalidation is applied.) For clarity, we show only α's component for all version vectors and omit the node ID (α) in accept stamps.

As Figure 4.8-(b) illustrates, when a node sends a stream of invalidations to another node, it discards gap-filling invalidations and it combines per-writer invalidations into multi-writer invalidations. Notice that now A is precise on δ.

Nodes can garbage collect any prefix of their logs, which allows each node to bound the amount local storage used for the log to any desired fraction of its total disk space. The truncated prefix can be summarized by a version vector *omitVV* [72].

Split-join example. The following example is a bit involved, but we have found that working through it step by step sheds considerable light on the purpose of the rules for updating the interest set status and log *gap filling* and *intersection* just described.

Figure 4.9 illustrates these mechanisms in action. Node α writes objects a, b, and c; node β cares about object a and receives from α precise invalidations about a and imprecise invalidations about b and c. Node γ cares about object c and receives from α precise invalidations about c and imprecise invalidations about a and b. Finally, node δ cares about

a and c and receives from β precise invalidations about a (but imprecise invalidations about b and c due to β's imprecision) and from γ precise invalidations about c (but imprecise invalidations about a and b.) First, α sends a stream of invalidations (precise for a and imprecise for b and c) to β. As illustrated in the figure, each invalidation advances β's per-invalidation-stream, per-interest-set $prevVV$ value as well as β's per-interest-set last precise version vector ($lpVV$) and current version vector (cVV) for interest set $\{$a$\}$. However, because the second invalidation $(4, 6, bc)$ intersects interest set $\{$b,c$\}$, that message causes that interest set to become imprecise and subsequent invalidations fail to advance that interest set's $lpVV$. After processing all four invalidations in that stream, β is precise for interest set $\{$a$\}$, but imprecise for interest set $\{$b,c$\}$. γ's behavior processing the stream of precise invalidations for c and imprecise invalidations for a and b is similar.

Then, when β and γ send their log contents to δ, we show the case where γ processes β's first three invalidations, then γ's four invalidations, and finally β's fourth invalidation. As the figure shows, after processing the first three invalidations from β, δ is precise for $\{$a$\}$, but imprecise for $\{$b$\}$ and $\{$c$\}$. The next four messages (from γ) make δ precise for $\{$c$\}$ but imprecise for $\{$a$\}$ and $\{$b$\}$. Finally, the last message (from β) brings δ to the state one would desire: after seeing all precise invalidations for objects a and c, δ is precise for both interest set $\{$a$\}$ and $\{$c$\}$ despite the fact that these precise messages were mixed with some imprecise invalidations for objects a, b, and c. Finally, one may verify that because of the δ's gap filling and intersection operations, δ's log contains sufficient information so that a node ϵ that receives δ's log contents could get precise updates for objects a or c. Conversely, note that if δ were simply to interleave the messages it received from α and β without gap filling and intersection and then send them to ϵ, information would be lost and ϵ would be left imprecise for interest sets $\{$a$\}$, $\{$b$\}$, and $\{$c$\}$.

4.5 Additional Features

Three novel aspects of our implementation further our goal of constructing a flexible framework that can accommodate the broadest range of policies. First, our implementation uses *self-tuning body propagation* to enable prefetching policies that are simultaneously aggressive and safe. Second, our CR-Repl implementation adds a *prevAccept* state to support a simple and flexible conflict detection and resolution mechanism. Third, our implementation provides *incremental log exchange* to allow systems to minimize the window for conflicting updates. Finally, we use Golding's algorithm [32] to implement the *isSequenced* predicate.

4.5.1 Self-tuning Body Propagation

In addition to supporting demand-fetch of particular objects, our prototype provides a novel self-tuning prefetching mechanism. A node *n1* subscribes to updates from a node *n2* by sending a list *L* of directories of interest along with a *startVV* version vector. *n2* will then send *n1* any bodies it sees that are in *L* and that are newer than *startVV*. To do this, *n2* maintains a priority queue of pending sends: when a new eligible body arrives, *n2* deletes any pending sends of older versions of the same object and then inserts a reference to the updated object. This priority queue drains to *n1* via a low-priority network connection that ensures that prefetch traffic does not consume network resources that regular TCP connections could use [92]. When a lot of spare bandwidth is available, the queue drains quickly and nearly all bodies are sent as soon as they are inserted. But, when little spare bandwidth is available, the buffer sends only high priority updates and absorbs repeated writes to the same object.

4.5.2 Conflict Detection and Resolution

The log exchange protocol just described last-writer-wins conflict resolution with global eventual consistency in the case of concurrent writes. However, it is useful to not only resolve conflicts in a globally consistent way but also to flag them and provide informa-

tion about conflicting writes to a more flexible manual or programmatic conflict resolution procedure.

To support more flexible conflict detection and resolution, we augment the algorithm described above by adding a field, *prevAccept* to both invalidation messages and to per-object state. When a node receives an invalidation GI and applies GI to the local store of an object obj (with $GI.acceptStamp \neq obj.acceptStamp$), there are three cases to consider. First, if $GI.prevAccept == obj.acceptStamp$, there is no write-write conflict. The second case, $GI.prevAccept > obj.acceptStamp$, is impossible by the prefix property. The third case, $GI.prevAccept < obj.acceptStamp$ and $Gi.acceptStamp$ is not included in $obj.lpVV$, represents a write-write conflict, which is resolved by updating obj with either GI or obj depending on which has a higher accept stamp and by storing the losing entry to disk in a local (non-shared) per-object *conflict file*; bodies that match stored losing writes are also stored. CR-Repl implementations can provide a local interface for reading and deleting these "losing" conflicting writes, which allows higher-level code to resolve conflicts using application-specific rules by generating compensating transactions.

Note that although different nodes can see different series of "losing" writes, all nodes that make an interest set precise are guaranteed to see the "final" write to each causally–independent series. For example, consider the case of two causal chains of writes to one object by the nodes α, β, and γ: (1) $0@\alpha, 1@\beta, 2@\beta, 3@\beta$ and (2) $0@\alpha, 4@\gamma$. The protocol guarantees that eventually any precise node will agree that the final state of the write is the result of γ's write at time 4 and that there was a write-write conflict that $3@\beta$ lost, and but different nodes may see different subsets of $1@\beta, 2@\beta, 3@\beta$, which seems acceptable in that neither causal chain regards either $1@\beta$ or $2@\beta$ as important values for the final state of the system.

4.5.3 Incremental Log Propagation

The prototype implements a novel variation on existing batch log exchange protocols. In particular, in the batch log exchange used in Bayou, a node first receives a batch of updates comprising a start time *startVV* and a series of writes, it then rolls back its checkpoint to before *startVV* using an undo log, and finally it rolls forward, merging the newly received batch of writes with its existing redo log and applying updates to the checkpoint. In contrast, our incremental log exchange applies each incoming write to the current RAS state without requiring roll-back and roll-forward of existing writes.

The advantages of the incremental approach are efficiency (each write is only applied to the RAS once), concurrency (a node can process information from multiple continuous streams), and consistency (connected nodes can stay continuously synchronized which reduces the window for conflicting writes.)

The disadvantage is that it only supports simple conflict detection logic: for our incremental algorithm, a node detects a write/write conflict when an invalidation's *prevAccept* logical time (set by the original writer to equal the logical time of the overwritten value) differs from the logical time the invalidation overwrites in the node's RAS. Conversely, batch log exchange supports more flexible conflict detection: Bayou writes contain a *dependency_check* procedure that can read any object to determine if a conflict has occurred [88]; this approach works in a batch system because rollback takes all of the system's state to a logical moment in time at which these checks can be re-executed. Note that this variation is orthogonal to the CR-Repl approach: a full replication system such as Bayou could be modified to use our incremental log propagation mechanism, and a PRACTI system could use batch log exchange with roll-back and roll-forward.

4.5.4 Simple Commit Implementation

As described in Chapter 3, URA exposes an *isSequenced* predicate that blocks requests according to write commit status. It is useful for implementing the stronger consistency

semantics such as *sequential consistency* and *linearizability* that require a *commit* protocol to establish a total order. A write is *committed locally* if a node sees all preceding writes in the final total order. The state of the art has different commit protocols including Golding's algorithm [32], primary commit [72], and quorum-based commit protocol [90]. For simplicity, here we use Golding's algorithm[2] to implement the *isSequenced* predicate.

It is straightforward to implement Golding's commit algorithm in CR-Repl. Each node uses its *currentVV* to determine whether or not it has seen all writes with logical time less than or equal to t and thus makes sure that all writes with logical stamp less than t are committed. For example, if $\alpha.currentVV$ equals $(3@\alpha, 5@\beta)$ and the system only has two nodes α and β, then we can derive that any write with an accept stamp less than 3 are committed. Note that for liveness, we need to put heartbeats at some nodes to bring the last write ($5@\beta$ in the above example) committed if there are no other updates at the other nodes.

4.6 Evaluation

In this section we evaluate the properties of CR-Repl protocol. We use the prototype both (1) to evaluate the PRACTI mechanisms in several environments such as web service replication, data access for mobile users, and grid scientific computing and (2) to characterize PRACTI's properties across a range of key metrics.

Our experiments seek to answer three questions.

1. *Can protocols implemented on CR-Repl match/approximate wide range of existing protocols?* We find that our system performance can match most of existing systems such as the PR-AC and AC-TI systems and approximate the performance of PR-TI object replication systems that gives up cross-object consistency.

2. *Do CR-Repl offer significant advantages over existing replication architectures for at*

[2]We have also implemented a sequential consistency library based on the primary commit protocol

least some environments? We find that our system can dominate existing approaches by providing more than an order of magnitude better bandwidth and storage efficiency than AC-TI replicated server systems, as much as an order of magnitude better synchronization delay compared to PR-AC hierarchical systems, and consistency guarantees not achievable by PR-TI per-object replication systems for some environments.

3. *What are the costs of CR-Repl's generality?* Given that a flexible CR-Repl protocol can subsume existing approaches, is it significantly more expensive to implement a given system using URA than to implement it using narrowly-focused specialized mechanisms? We find that the primary "extra" cost of CR-Repl's generality is that our system can transmit more consistency information than a customized system might require. But, our implementation reduces this cost compared to past systems via separating invalidations and bodies and via imprecise invalidations, so these costs appear to be minor.

To provide a framework for exploring these issues, we compare our system with the three major classes of replication architectures defined by the PRACTI taxonomy as we described in Chapter 2. In particular, we first focus on partial replication by comparing our protocol with AC-TI systems in 4.6.1. We then compare our protocol with PR-AC and AC-TI systems in 4.6.2. Finally, we examine the costs of flexible consistency by comparing our protocol with PR-TI in 4.6.3.

4.6.1 Partial Replication

When comparing to the AC-TI full replication protocols from which our CR-Repl system descends, we find that support for partial replication can dramatically improve performance for three reasons:

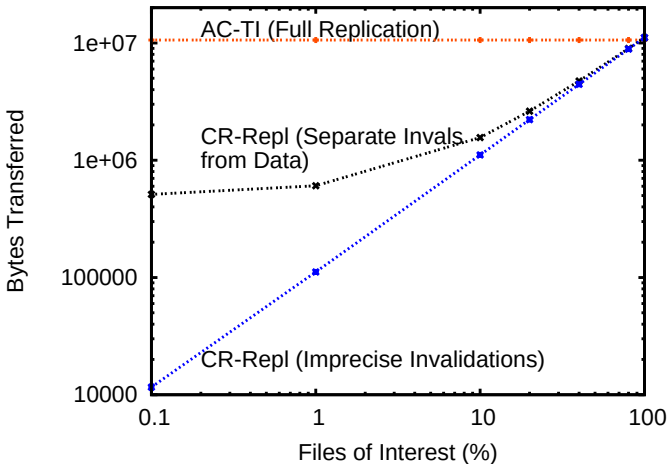


Figure 4.10: Impact of locality on replication cost.

1. *Locality of Reference:* partial replication of bodies and invalidations can *each* reduce storage and bandwidth costs by an order of magnitude for nodes that care about only a subset of the system's data.

2. *Bytes Die Young:* partial replication of bodies can significantly reduce bandwidth costs when "bytes die young" [7].

3. *Self-tuning prefetching:* self-tuning prefetching minimizes response time for a given bandwidth budget.

It is not a surprise that partial replication can yield significant performance advantages over existing server replication systems. What is significant is that (1) our experiments provide evidence that despite the good properties of server replication systems (e.g., support for disconnected operation, flexible consistency, and dynamic network topologies) these systems may be impractical for many environments; and (2) they demonstrate that these trade-offs are not fundamental—a CR-Repl system can support PR while retaining the good AC-TI properties of server replication systems.

Locality of reference. Different devices in a distributed system often access different subsets of the system's data because of locality and different hardware capabilities. In such environments, some nodes may access 10%, 1%, or less of the system's data, and partial replication may yield significant improvements in both bandwidth to distribute updates and space to store data.

Figure 4.10 examines the impact of locality on replication cost for three systems implemented on our CR-Repl core using different controllers: a full replication system similar to Bayou, a partial-body replication system that sends all precise invalidations to each node but that only sends some bodies to a node, and a partial-replication system that sends some bodies and some precise invalidations to a node but that summarizes other invalidations using imprecise invalidations. In this benchmark, we overwrite a collection of 1000 files of 10KB each. A node subscribes to invalidations and body updates for subset of the files that are of interest to that node. The X axis shows the fraction of files that belong to a node's subset, and the y axis shows the total bandwidth required to transmit these updates to the node.

The results show that partial replication of both bodies and invalidations is crucial when nodes exhibit locality. Partial replication of bodies yields up to an order of magnitude improvement, but it is then limited by full replication of metadata. Using imprecise invalidations to provide true partial replication can gain over another order of magnitude as locality increases.

Note that Figure 4.10 shows bandwidth costs. Partial replication provides similar improvements for space requirements (graph omitted).

Bytes die young. Bytes are often overwritten or deleted soon after creation [7]. ACTI Full replication systems send all writes to all servers, even if some of the writes are quickly made obsolete. In contrast, CR-Repl replication can send invalidations separately from bodies: if a file is written multiple times on one node before being read on another, overwritten bodies need never be sent.

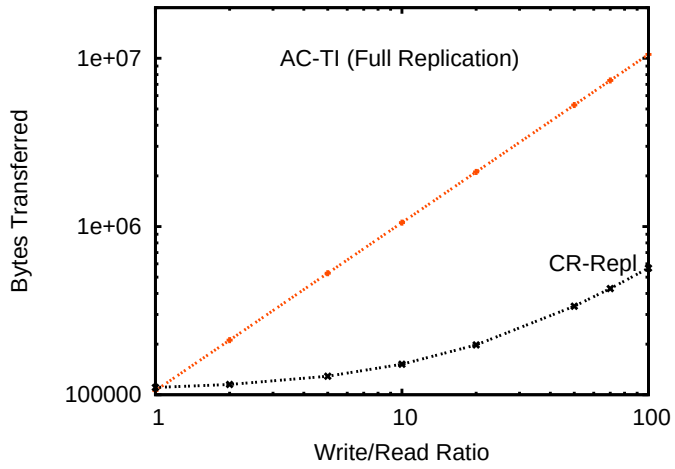


Figure 4.11: Bandwidth cost of distributing updates as the number of writes to a file between reads varies.

To examine this effect, we randomly write a set of files on one node and randomly read the files on another node. As Figure 4.11 shows, CR-Repl's gains are significant when bytes die young. For example, when the write to read ratio is 2, CR-Repl uses 55% of the bandwidth of full replication, and when the ratio is 5, CR-Repl uses 24%. At ratios exceeding 20, CR-Repl's gains exceed an order of magnitude.

Self-tuning prefetching. Separation of invalidations from bodies enables a novel self-tuning data prefetching mechanism described in Section 4.5. As a result, systems can replicate bodies aggressively when network capacity is plentiful and replicate less aggressively when network capacity is scarce.

Figure 4.12 illustrates the benefits of this approach by evaluating three systems that replicate a web service from a single origin server to multiple edge servers. In the *dissemination services* [66] we examine, all updates occur at the origin server and all client reads are processed at edge servers, which serve both static and dynamic content. We compare the read response time observed by the edge server when accessing the database to service client requests for three replication policies: *Demand Fetch* follows a standard client-server

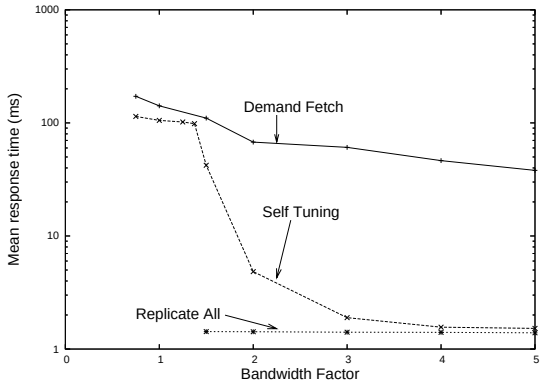


Figure 4.12: Read response time when available bandwidth varies for full replication, demand fetching, and self-tuning prefetching.

HTTP caching model by replicating precise invalidations to all nodes but sending new bodies only in response to *demand read*requests, *Replicate All* follows a Bayou-like approach and replicates both precise invalidations and all bodies to all nodes, and *Self Tuning* exploits CR-Repl to replicate precise invalidations to all nodes and to have all nodes subscribe for all new bodies via the self-tuning prefetching mechanism. We use a synthetic workload where the read:write ratio is 1:1, reads are Zipf distributed across files (α = 1.1), and writes are uniformly distributed across files. We use Dummynet to vary the available network bandwidth from 0.75 to 5.0 times the system's average write throughput. In addition to prototype benchmark experiment reported here, we also simulate performance under a range of other parameters, which yields expected results: increasing α improves the read hit rate when not all bodies are prefetched, decreasing the read to write ratio for a given write rate hurts the read response time for *Demand Fetch*, and increasing the write rate shifts the curves to the right.

As Figure 4.12 shows, when spare bandwidth is available, self-tuning prefetching improves response time by up to a factor of 20 compared to *Demand-Fetch*. A key challenge, however, is preventing prefetching from overloading the system. Whereas our self-tuning approach adapts bandwidth consumption to available resources, *Replicate All* sends all updates regardless of workload or environment. This makes *Replicate All* a poor

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

Figure 4.13: Configuration for mobile storage experiments.

neighbor—it consumes prefetching bandwidth corresponding to the current write rate even if other applications could make better use of the network.

4.6.2 Topology Independence

We examine topology independence by considering two environments: a mobile data access system distributed across multiple devices and a wide-area-network file system designed to make it easy for PlanetLab and Grid researchers to run experiments that rely on distributed state. In both cases, CR-Repl's combined partial replication and topology independence allows our design to dominate PR-AC topology-restricted hierarchical approaches by doing two things:

1. *Adapt to changing topologies*: a CR-Repl system can make use of the best paths among nodes.

2. *Adapt to changing workloads*: a CR-Repl system can optimize communication paths to, for example, use direct node-to-node transfers for some objects and distribution trees for others.

We primarily compare against standard PR-AC restricted-topology client-server systems like Coda and IMAP. For completeness, our graphs also compare against AC-TI topology-independent, full replication systems like Bayou.

Mobile storage. We first consider a mobile storage system that distributes data across palmtop, laptop, home desktop, and office server machines. We compare a CR-Repl sys-

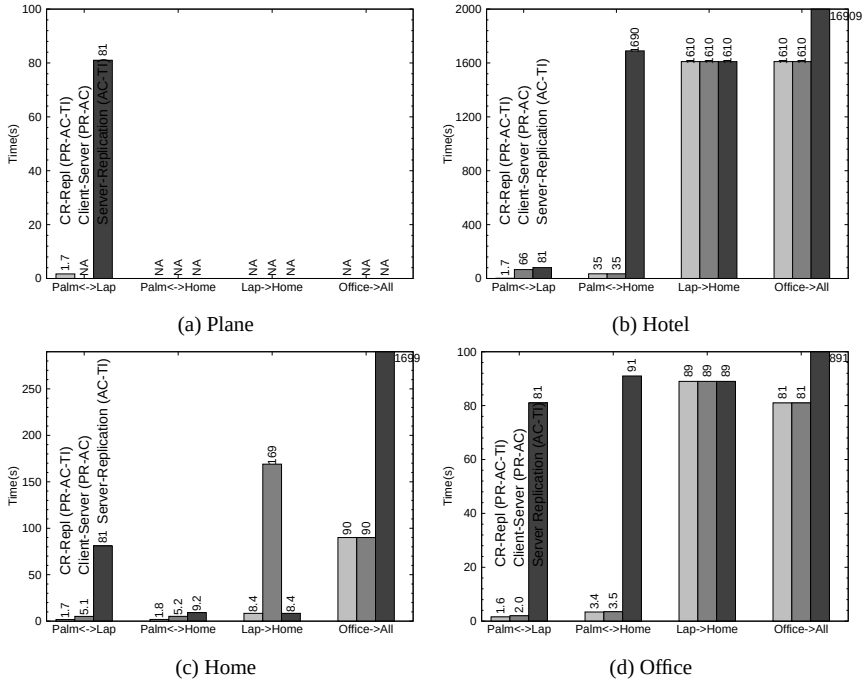


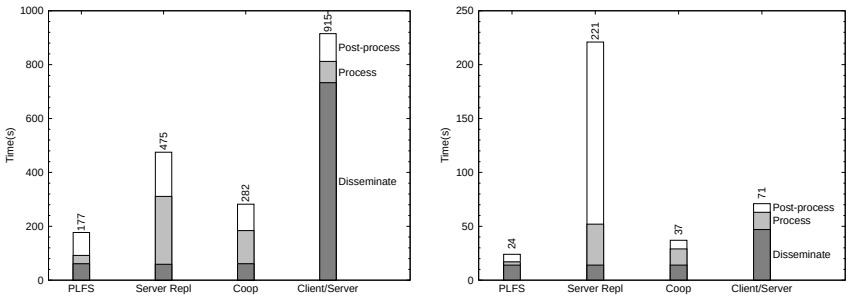
Figure 4.14: Synchronization time among devices for different network topologies and protocols.

tem to a client-server Coda- or IMAP-like system that supports partial replication but that distributes updates via a central server and to a full-replication Bayou-like system that can distribute updates directly between any nodes but that requires full replication. All three systems are realized by implementing different controllers with different policies.

As summarized in Figure 4.13, our workload models a department file system that supports mobility: an office server stores data for 100 users, a user's home machine and laptop each store one user's data, and a user's palmtop stores 1% of a user's data. Note that due to resource limitations, we store only the "dirty data" on our test machines, and we use desktop-class machines for all nodes. We control the network bandwidth of each scenario using a library that throttles transmission.

Figure 4.14 shows the time to synchronize dirty data among machines in four scenarios: (a) *Plane*: the user is on a plane with no Internet connection, (b) *Hotel*: the user's laptop has a 50Kb/s modem connection to the Internet, (c) *Home*: the user's home machine has a 1Mb/s connection to the Internet, and (d) *Office*: the office desktop has a 100Mb/s connection to the Internet. The user carries her laptop and palmtop to each of these locations and co-located machines communicate via wireless network at speeds indicated in Figure 4.13. For each location, we measure time for machines to exchange updates: (1) Palm↔Lap: the palmtop and laptop exchange updates, (2) Palm↔Home: the palmtop and home machine exchange updates, (3) Lap→Home: the laptop sends updates to the home machine, (4) Office→All: the office server sends updates to all nodes.

In comparing the CR-Repl system to a client-server system, topology independence has significant gains when the machines that need to synchronize are near one another but far from the server: in the isolated *Plane* location, the palmtop and laptop can not synchronize at all in a client-server system; in the *Hotel* location, direct synchronization between these two co-located devices is an order of magnitude faster than synchronizing via the server (1.7s v. 66s); and in the *Home* location, directly synchronizing co-located devices is between 3 and 20 times faster than synchronization via the server.



(a) 50 distributed nodes + remote server (b) 50 cluster nodes + remote server

Figure 4.15: Execution time for the WAN-Experiment benchmark.

Conversely, as the "Full Replication" lines show, although existing server-replication systems provide topology independence, full-replication limits their effectiveness. For example, even if a palmtop is only interested in 100MB of the system's data and in 100KB of the laptops' updates, full replication would require it to store 100GB of data and receive 10MB of updates when synchronizing with the laptop. Such a configuration appears infeasible, so if such an existing "full replication" system were used, it would likely manually partition data into volumes and configure the system so that different devices store different subsets of volumes. In principle, careful volume configuration could approximate the performance of CR-Repl in this experiment, but it is not clear how to configure or manage such a system. Also note that partitioning data into separate replication volumes would sacrifice causal consistency across volumes and would likely prevent conflict detection and reconciliation rules [88] whose inputs or outputs span volumes.

WAN-FS for Researchers. Figure 4.15 evaluates a wide-area-network file system called PLFS designed for PlanetLab and Grid researchers. The controller for PLFS is simple: for invalidations, PLFS forms a multicast tree to distribute all precise invalidations to all nodes. And, when an *INVALID* file is read, PLFS uses a DHT-based system [97] to find the nearest copy of the file; not only does this approach minimize transfer latency, it effectively forms

a multicast tree when multiple concurrent reads of a file occur [5, 89].

We examine a 3-phase benchmark that represents running an experiment: in phase 1 *Disseminate*, each node fetches 10MB of new executables and input data from the user's home node; in phase 2 *Process*, each node writes 10 files each of 100KB and then reads 10 files from randomly selected peers; in phase 3, *Post-process*, each node writes a 1MB output file and the home node reads all of these output files. We compare PLFS to three systems: a client-server system, client-server with cooperative caching of read-only data [5], and server-replication [72]. All 4 systems are implemented via CR-Repl using different controllers.

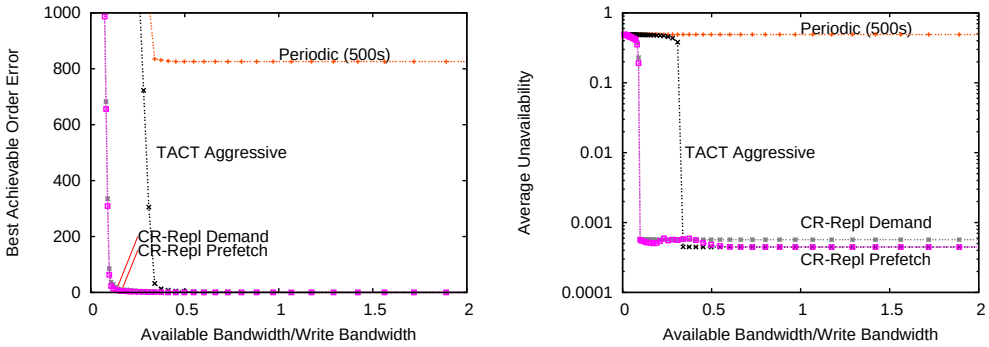
The figures show performance for an experiment running on 50 distributed nodes each with a 5.6Mb/s connection to the Internet (we emulate this case by throttling bandwidth) and 50 cluster nodes at the University of Texas with a switched 100Mb/s network among them and a shared path via Internet2 to the origin server at the University of Utah.

The speedups range from 1.5 to 9.2, demonstrating the significant advantages enabled by the CR-Repl architecture. Compared to client/server, it is faster in both the Dissemination and Process phases due to its multicast dissemination and direct peer-to-peer data transfer. Compared to full replication, it is faster in the Process and Post-process phases because it only sends the required data. And compared to cooperative caching of read only data, it is faster in the Process phase because data is transferred directly between nodes.

4.6.3 Any Consistency

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

Improved consistency trade-offs. CR-Repl improves the range of consistency trade-offs available for replication. Gray [34] and Yu and Vahdat [102] show a trade-off: aggressive propagation of updates improves consistency and availability but can also increase system



(a) Best consistency (order error) achievable for a given bandwidth cost.

(b) Best unavailability achievable while meeting a required order error of 100.

Figure 4.16: Consistency trade-offs.

load. Yu's study finds an order of magnitude improvement in reducing unavailability for some workloads when using aggressive propagation of updates compared to lazy propagation and Gray shows that the number of conflicts can rise with the square of propagation delay for some workloads [34].

We examine a range of consistency requirements and network failure scenarios via simulation (all other experiments in this paper are prototype measurements.) We use a synthetic read/write workload with the same parameters as the workload used in Fig 4.12. We use an average network path unavailability of 0.1% with Pareto distributed repair time R(t) = $1 - 15t^{-0.8}$ [19].

In Figure 4.16-a we measure the best order error that can be maintained for a given bandwidth budget. Order error constrains the number of outstanding uncommitted writes [103]. We compare the *TACT Aggressive* policy [102] to a *PRACTI Prefetch* policy that aggressively distributes invalidations as in TACT's policy but that distributes bodies using the self-tuning approach. CR-Repl reduces the bandwidth needed to maintain reasonable consistency by a factor of 3 compared to *TACT Aggressive* and improves the

consistency bounds attainable for some bandwidth budgets by orders of magnitude.

Figure 4.16-b plots system unavailability for an order error bound of 100 as bandwidth varies. Following Yu and Vahdat's methodology [102], we say that the system is *available* to a read or write request if the request can issue without blocking and the system is *unavailable* if the request must block in order to meet the consistency target(i.e, the current total order error is less than 100). When bandwidth is limited, CR-Repl dramatically improves system availability under consistency constraints compared to full replication.

Consistency overheads. Our protocol ensures that requests pay only the latency and availability costs of the consistency they require. But, distributing sufficient bookkeeping information to support a wide range of per-request semantics does impose a bandwidth cost. If all applications in a system only care about coherence guarantees, a customized protocol for that system could omit imprecise invalidations and thereby reduce network overheads.

Three features of our protocol minimize this cost. First, transmitting invalidations separately from bodies allows nodes to maintain a consistent view of data without receiving all bodies. Second, transmitting imprecise invalidations in place of some precise invalidations allows nodes to maintain a consistent view of data without receiving all precise invalidations. Third, self-tuning prefetch of bodies allows a node to maximize the amount of local, valid data in a checkpoint for a given bandwidth budget.

Figure 4.17 quantifies the remaining cost to distribute both precise and imprecise invalidations (in order to support consistency) versus the cost to distribute only precise invalidations for the subset of data of interest and omitting the imprecise invalidations (and thus only supporting coherence.) We vary the fraction of data of interest to a node on the x axis and show the invalidation bytes received per write on the y axis. Our workload is a series of writes by remote nodes in which all objects are equally likely to be written. Note that the cost of imprecise invalidations depends on the workload's locality: if there is no locality and writers tend to alternate between writing objects of interest and objects not of interest, then the imprecise invalidations between the precise invalidations will cover

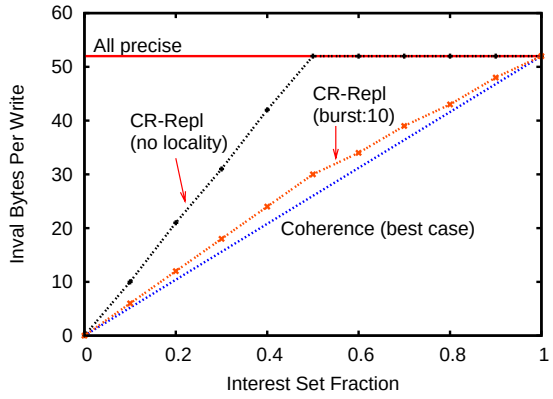


Figure 4.17: Bandwidth cost of consistency information.

relatively few updates and save relatively little overhead. Conversely, if writes to different interest sets arrive in bursts, then the system will generally be able to accumulate large numbers of updates into imprecise invalidations. We show two cases: the *No Locality* line shows the worst case scenario, with no locality across writes, and the *burst=10* line shows the case when a write is ten times more likely to hit the same interest set as the previous write than to hit a new interest set.

When there is significant locality for writes, the cost of distributing imprecise invalidations is small: imprecise invalidations to support consistency never add more than 20% to the bandwidth cost of supporting only coherence. When there is no locality, the cost is higher, but in the worst case imprecise invalidations add under 50 bytes per precise invalidation received. Overall, the difference in invalidation cost is likely to be small relative to the total bandwidth consumed by the system to distribute bodies.

Chapter 5

UR-Repl: Universal Replication Mechanisms

The CR-Repl protocol just described in previous chapter is only a first step towards a universal data replication architecture. Although it implements the replication abstractions by supporting PRACTI properties simultaneously, it does fall short of our eventual goal of providing a unified architecture in four significant ways. First, it does not efficiently support dynamic fine-grained invalidation subscriptions that are needed for many caching protocols that support single object callbacks [42, 68]. In particular, the processing cost is proportional to the number of interest sets, which is reasonable for coarse-grain subscriptions, but not for per-object fine-grain subscriptions. Second, the invalidation subscription is inefficient for some workloads where checkpoint exchange is more efficient. Third, although it supports a simple commit, it lacks a general commit mechanism to efficiently support commit protocols such as the primary commit CSN [72]. Finally, although the CR-Repl protocol supports a simple write-write conflict detection mechanism, it adds extra per-update stamp overhead for both storage and network bandwidth, and it could not work properly when the log is truncated.

To complete the CR-Repl mechanisms to serve as a replication “microkernel”, this

chapter presents a novel replication protocol UR-Repl that addresses these limitations via four key ideas:

1. In order to efficiently support both coarse-grained and fine-grained subscriptions, we *multiplex invalidation subscriptions* over a single stream. By maintaining a shared state for multiple subscriptions and allowing one imprecise invalidation to be used across all active subscriptions, the processing overhead to handle each invalidation is reduced from $O(number\ of\ activeinterestsets)$ to update each interest set's state to $O(1)$ to update the per-stream's state.

2. In order to support fast resynchronization of different type of workloads, besides incremental log exchange, UR-Repl introduces a novel *incremental checkpoint exchange* and smoothly integrates it with CR-Repl log exchange protocol. Instead of freezing the receiver and sender's states as required in most of existing checkpoint exchange protocols such as that in [72], UR-Repl allows the receiver to receive an incremental checkpoint for a small portion of its ID space and then either prefetch checkpoints of other interest sets or fault them in "on demand".

3. In order to support flexible commit protocols, we introduce a novel mechanism that makes use of a special primitive and a special message *commit invalidation* to allow system designers to explicitly specify when and under what circumstances to commit a write.

4. In order to support efficient conflict detection for both log exchange and checkpoint exchange, we use novel *dependency summary vectors* to detect write-write conflicts. UR-Repl allows multiple objects to share the same version vector to detect conflict so as not to incur additional costs for conflict detection other than that already being paid for consistency maintenance.

In the rest of this chapter, we first describe the enhanced invalidation subscription protocol by multiplexing subscriptions in one stream in Section 5.1. Then Section 5.2 de-

	UR-Repl	CR-Repl
Network bandwidth	N_{Sub}* (S_{Setup}+N_{Pre}*S_{Pre}+N_{Imp}*S_{Imp})	N_{Sub} * (S_{Setup}+N_{Pre}*S_{Pre}) + N_{Imp}*S_{Imp}
Inv Processing overhead	$O(number\ of\ interest\ sets)$	$O(1)$
Read Overhead	$O(1)$	$O(number\ of\ connections)$

Figure 5.1: Invalidation subscription cost. Here, for simplicity, we assume each subscription request although subscribes a different subscription set, it requires the same number of precise invalidations and imprecise invalidations. N_{Sub} is the number of subscription requests; $N_{Precise}$ and $S_{Precise}$ are the number of updates targeting objects in the subscription set from the subscription start time to the current logical time and the size of one precise invalidation; Similarly, $N_{Imprecise}$ and $S_{Imprecise}$ are the number of imprecise invalidations in one subscription and the size of an imprecise invalidation; S_{Setup} is the size of setup a subscription.

scribes the novel incremental checkpoint exchange protocol, Section 5.3 details the general commit mechanism. and Section 5.4 presents the *dependency summary vectors* conflict detection algorithm. Finally, Section 5.5 evaluates the UR-Repl prototype.

5.1 UR-Repl Invalidation Subscription

As we described in Section 3.1.2, the invalidation subscription abstraction is a natural way to implement callbacks. To create a callback for a single object o on the server so that it will be notified of any new updates to o by the server, a client simply sets up an invalidation subscription with a subscription set composed of only o and a start time of $o.acceptStamp$. The *callback break* can be implemented simply by stopping the corresponding subscription.

Unfortunately, the invalidation subscription protocol described in the previous chapter makes it expensive to implement such fine-grained dynamic callbacks for two reasons.

First, although *imprecise invalidations* alone significantly reduce the bandwidth cost of a single invalidation stream [9], the total bandwidth cost could still be significant for serving multiple dynamic fine-grained subscription requests. For multiple subscription re-

quests, the same set of updates are sent multiple times for consistency although in different compact formats. For example, comparing *Stream 1* and *Stream 2* in Figure 4.3, to serve two subscription requests from the same node, α needs to send two different streams each of which must include precise or imprecise invalidations to cover all the updates issued after *startVV*. When the number of objects included in one subscription set is small and the number of subscription requests is large, the total bandwidth cost will be much more than the cost that is actually demanded for the subscribed workload.

Similarly, as the processing overhead to handle each invalidation is proportional to the number of interest sets, the total processing overhead of multiple subscriptions might be huge. Upon receiving an invalidation, a node must iterate across all interest sets twice to update the consistency state as indicated in Figure 4.4.

UR-Repl addresses these issues by *multiplexing subscriptions*. Recall that the key subscription invariant is that the sender sends all updates as precise invalidations or imprecise invalidations from the subscription start time to its current time. Instead of sending a stream of invalidations for one subscription, UR-Repl multiplexes all invalidation subscriptions from one node to another onto a single underlying invalidation stream. It thus reduces the network bandwidth overhead for adding a new subscription by allowing one imprecise invalidation to be used across all active subscriptions.

In addition, by allowing multiple interest sets to share a single *stream state*, UR-Repl reduces the processing overhead to handle each invalidation to update each interest set's state from $O(number\ of\ interest\ sets)$ to $O(1)$. The disadvantage is that the read performance may increase from $O(1)$ to $O(number\ of\ connections)$.

Figure 5.1 summarizes the invalidation subscription cost of CR-Repl and UR-Repl in terms of network bandwidth and processing cost. UR-Repl thus makes the protocol efficient for both coarse-grained callbacks and dynamically-created, fine-grained callbacks.

In the rest of this section, we first explain how to form an invalidation stream when multiplexing subscriptions and describe the steps to form such a stream in Section 5.1.1. We

$$InvalStream = StreamStart + [\ [\ LogCatchup | CPCatchup\ ]^* + [preciseInv | impreciseInv]^*\ ]^*$$
$$LogCatchup = CatchupStart + [preciseInv | impreciseInv]^* + CatchupEnd$$
$$CPCatchup = CatchupStart + \langle /*,\ r.cvv,\ s.cvv \rangle + SS.lpVV + [per object\ states\ of\ SS] + CatchupEnd$$

Figure 5.2: UR-Repl invalidation stream.

then explain how to process the stream to maintain the consistency state in Section 5.1.2.

5.1.1 Forming Invalidation Streams

Given the CR-Repl protocol, the basic idea to implement UR-Repl is simple. Similar to the CR-Repl replication, UR-Repl separates the distribution of updates into invalidations and bodies for partial replication of data and uses *imprecise invalidations* to reduce the bandwidth overhead of a single invalidation subscription to be proportional to the updates of subscribed objects. Then to reduce the overall invalidation stream bandwidth between two nodes, it *multiplexes* subscriptions to allow a node to dynamically add objects or remove objects from the subscription set on a single stream.

Figure 5.2 defines a UR-Repl invalidation stream. Similar to a CR-Repl invalidation stream, it has a $StreamStart$ to indicate the start point of the stream followed by a sequence of causally ordered precise or imprecise invalidations. The only difference from a CR-Repl invalidation stream is that a UR-Repl invalidation stream may include some *catchup streams* that are needed for adding new invalidation subscriptions to an existing stream.

As the separation of invalidations and bodies and the forming of *imprecise invalidation* are exactly the same as described in Section 4.2 and Section 4.3 respectively, here we focus on explaining how to add/remove subscriptions to/from an existing stream.

Multiplexing subscriptions. In order to multiplex invalidation subscriptions, each node α maintains a per-receiver outgoing stream shared by all active subscriptions between the receiver and α. A stream tracks a *subscription set SS* to identify which objects are currently

subscribed and a *version vector streamCVV* to summarize all the previous updates sent by the stream.

To implement the subscription abstraction, a stream promises that it will send all invalidations after *streamCVV* in causal order. In particular, it will send all invalidations to objects in *SS* as they are and combine invalidations about objects outside of *SS* into imprecise invalidations.

As indicated in Figure 5.2, to add a new subscription with *(newSS, startVV)* to a stream, a node must first send a *catchup stream* between *startVV* and *streamCVV* before adding *newSS* to the stream's current subscription set *SS*. The *catchup* stream is needed to implement the semantics of the invalidation subscription abstraction as described in Chapter 3: "send all precise invalidations to objects in newSS after startVV". Because *newSS* is not included in *SS*, the stream might have sent invalidations to objects in *newSS* as imprecise invalidations. Therefore, we need the catchup stream to send the omitted precise information about *newSS* from *startVV* to *streamCVV*.

Note that the receiver must already have received imprecise or precise invalidations for all objects up to *streamCVV*, so the sender is free to send this information about past updates to *newSS* without violating consistency.

Note that instead of sending an ordered sequence of invalidations between *startVV* and *streamCVV* to catchup *newSS*, a node can alternatively send the checkpoint of all the objects in *newSS* that were updated between *startVV* and *newSS*. As a result, as indicated in Figure 5.2 an invalidation stream is composed of a version vector to indicate where it starts, a series of precise invalidations and imprecise invalidations sent in causal order, and a set of catchup invalidations or checkpoints for adding new objects to the stream. We focus on describing the invalidation catchup here and defer the description of the checkpoint catchup to Section 5.2.

Sending an invalidation stream. Figure 5.3 illustrates the sender's invalidation subscription protocol in action. The protocol is simple and includes 4 parts: initialization, adding

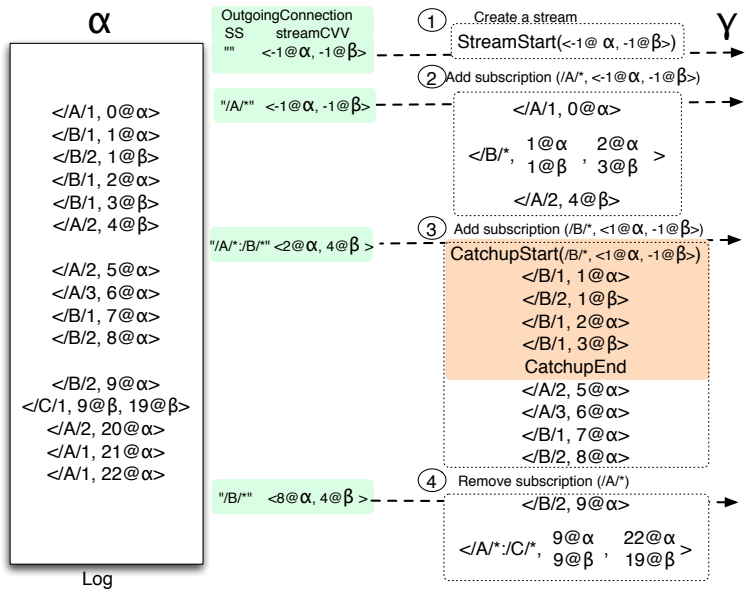


Figure 5.3: Multiplexing invalidation subscriptions.

a catchup stream, removing a set of objects from a stream, sending regular invalidation streams. As illustrated in ① of Figure 5.3, initially the sender sets its *SS* to empty and *streamCVV* to its *currentVV* and sends a *StreamStart* message that includes its *currentVV* so that the receiver knows where the stream starts.

When there is no pending subscription requests, the sender sends precise invalidations and imprecise invalidations accumulated according to the stream's *subscription set* as described in Section 4.3 and updates the *streamCVV* accordingly. For example, in Figure 5.3 step ③, α sends all the precise invalidations targeting directory $/A$ and sends an imprecise invalidation to summarize all the updates targeting any object in directory $/B$ because the stream's *subscription set* is $/A/*$ during this period. Whereas in step ⑤, as the *subscription set* is $/B/*$, the stream sends precise invalidations targeting any object in directory $/B$ and summarizes all the other invalidations in imprecise invalidations.

As indicated in step ④, whenever a new subscription request comes, the sender sends an *invalidation catchup stream* that includes all the precise invalidations targeting

any object in *newSS* between *startVV* and *streamCVV* as shown in the grey box. A *catchup stream* starts with a *CatchupStart* message that includes *newSS* and *startVV* and ends with a *CatchupEnd* message so that the receiver knows how to process the catchup stream. Note that the catchup stream can safely omit all invalidations not targeting objects in *newSS* because the stream has already logically sent information about every updates from stream start time up to *streamCVV*.

To remove a subscription set *removeSS* from a stream, the sender only needs to remove the *removeSS* from *SS* so that the sender will replace future invalidations targeting *removeSS* by imprecise invalidations. As indicated in Figure 5.3 step ⑤, after the *SS* is updated to $/B/*$, the invalidation stream combines invalidations to $/A/*$ thereafter.

5.1.2 Applying Invalidation Streams

In the previous subsection, we define how a sender creates an invalidation stream; in this subsection, we explain how to efficiently maintain the local state when receiving an invalidation stream.

Note that the processing of *imprecise invalidations* is different here due to the *multiplexing of subscriptions*. In particular, where CR-Repl applies each invalidation to each interest set for which it tracks consistency state for, UR-Repl only applies the invalidation to a stream shared by multiple interest sets and therefore reduces the processing cost from $\emptyset(number\,of\,invalidations \times number\,of\,interest\,sets)$ to $O(number\,of\,invalidations)$.

Consistency state. Like the CR-Repl replication, UR-Repl needs to track the per-object state and per-interest set state. For completeness, here we give a brief overview of maintaining these states.

First, because updates are distributed separately by invalidation subscriptions and body subscriptions, each node needs to track if the object it is interested in has received an invalidation with or without the corresponding body. In particular, it maintains a per-object state that includes an object ID, the *acceptStamp* of the last known invalidation to the object,

a flag *VALID* that if true indicates that the corresponding data associated to *acceptStamp* is available. When applying a newer invalidation, the per-object state *acceptStamp* is updated and the *VALID* is marked $false$. When receiving a body message, if the *acceptStamp* matches with the one in per-object state, *VALID* is marked to $true$. Sometimes we also say the object is marked *INVALID*.

Second, due to imprecise invalidations, a node also needs to track the *preciseness* of each object, i.e., whether an object has missed any invalidations. To save the storage space, we use *an interest set* to track the *preciseness* of a group of objects. In particular, each interest set *IS* maintains a version vector *lpVV* to identify the latest time at which a node is known to have seen all invalidations that could affect any object in *IS*. A node tracks whether each interest set *IS* is PRECISE (per-object state reflects all precise invalidations) or IMPRECISE (per-object state is not stored or may not reflect all precise invalidations) by comparing *IS.lpVV* and the node's *currentVV*. If *IS.lpVV* equals *currentVV*, then *IS* has not missed any invalidations and it is precise.

Stream state. To avoid applying each invalidation to each interest set for maintaining the per-interest set *lpVV*, a receiver maintains a per-stream state *streamCVV* shared by multiple interest sets. A *streamCVV* summarizes all the invalidations sent in the stream and is advanced whenever the node receives a new invalidation from the stream. UR-Repl leverages the per-stream *streamCVV* to track the *lpVV* efficiently by *attaching* multiple interest sets to an active stream.

In order to derive *lpVV* by *streamCVV*, UR-Repl enforces an invariant when *attaching* any interest set IS to any stream S: IS can be *attached* to S if and only if IS has received all precise invalidations to objects in IS up to $streamCVV$. Therefore, a node must *detach* an interest set IS from a stream S whenever the node misses any precise invalidation to IS up to $S.streamCVV$. Note that a stream S might omit some precise invalidations to IS, but a node might have learned those precise invalidations from other streams, therefore, IS might still be able to be attached to S. A node can attach an interest

set to multiple streams from different senders.

In this arrangement, to track the *lpVV*, each interest set *IS* stores a list of references *attachedStreams* to the active streams to which *IS* is currently attached and a *lastKnownLPVV* to record the last known *lpVV* when it is detached from a stream. When the node needs to calculate the *IS.lpVV*, it only needs to take the maximum of *IS.lastKnownLPVV* and the *streamCVV* of every stream included in *IS.attachedStreams*.

Processing an invalidation stream. A node must ensure two things when applying an invalidation stream to guarantee the *attach* invariant,

- It cannot attach an interest set *IS* to a stream *S* until *IS* is precise up to *S.streamCVV*.
- It must *detach* the interest set *IS* from a stream if it receives an imprecise invalidation *II* overlapping with *IS* and *IS* is not made precise up to *II.end* because after applying *II*, *S.streamCVV* will be advanced to include *II.end*.

Besides *streamCVV*, a node also maintains an *attachedSS* for each active incoming stream to track which interest sets are attached so that it knows whom to detach when receiving an imprecise invalidation.

Processing an invalidation stream merely involves updating these data structures to implement the *attach* semantics described above. In the rest of this section, we describe how to apply messages received from a stream *S* on a node α.

Figure 5.4 details the algorithm for processing an incoming invalidation stream.

- As indicated in Figure 5.4 from Line 42 to Line 46, upon receiving a precise invalidation *PI*, α advances *S.streamCVV* to include *PI* and updates the corresponding per-object state if *PI.targetSet* is one of the objects that it is interested in. Because any invalidation stream preserves the prefix property [72], the arrival of *PI* implies that there are no updates between *S.streamCVV* and *PI.acceptStamp*. Therefore any attached *IS* can remain attached when *S* advances its *streamCVV*.

```
1: // Global state:
2: //   IS.lastKnownLPVV – IS's last known precise version vector
3: //   IS.attachedStreams – streams to which IS is currently attached
4: // Per-stream state:
5: //   streamCVV – stream current version vector
6: //   attachedSS – attached subscription set
7: //   pendingCVV – current version vector of the catchup stream
8: //   pendingSS – pending subscription set to be attached
9:
10: Procedure ProcessURInvalStream(stream)
11: // Initialization
12: attachedSS = empty
13: streamCVV = stream.readObj()
14: if ! includes(currentVV, streamCVV)
15:    applyInv(new ImpreciseInv(currentVV, streamCVV, "/*")) // Cover the gap
16: NextMSG = stream.readObj()
17: while (NextMSG ≠ EOF) do
18:    if NextMSG instanceof GeneralInv  // Normal invalidation stream
19:       applyInval(NextMSG, streamCVV, attachedSS)
20:    else // Catchup stream starts
21:       // Apply catchup stream
22:       pendingSS = NextMSG.catchupSS, pendingCVV = NextMSG.catchupStartVV
23:       IgnoreCatchup =! includes(pendingCVV, pendingSS.getLPVV())
24:       while (NextMSG ≠ CatchupEnd) do
25:          NextMSG = stream.readObj()
26:          if ! IgnoreCatchup
27:             applyInval(NextMSG, pendingCVV, pendingSS)
28:       if ! IgnoreCatchup  // Attach pendingSS
29:          attachedSS.add(pendingSS)
30:          pendingSS.attachedStreams.add(S)
31:    NextMSG = stream.readObj()
32:
33: Procedure applyInval(GI, CVV, SS)
34: currentVV =advanceToInclude(currentVV, GI.end)
35: log.insert(GI)
36: if GI.isImprecise() and GI.targetSetoverlapsSS  // If imprecise, remove overlapped subscription set
    and advance lpvv to just before imprecise inval
37:    KickedSet = GI.targetSet.getIntersection(SS)
38:    KickedSet.lastKnownLPVV =advanceNoInclude(CVV, GI.start))
39:    KickedSet.attachedStreams.remove(S)
40:    SS.remove(KickedSet)
41: // Advance CVV to include GI
42: CVV = advanceToInclude(CVV, GI.end)
43: // Update per-object state
44: if GI.isPrecise() and GI.targetSet ∈ attachedSS
45:    RAS_GI.objId.valid = INVALID
46:    RAS_GI.objId.accept = GI.start
```

Figure 5.4: Stream processing algorithm for $stream$ = {$startVV$, [[$CatchupStart$, [$preciseInv|impreciseInv$]*, $CatchupEnd$]*, [$preciseInv|impreciseInv$]*]*}

- As indicated in Figure 5.4 from Line 36 to Line 40, upon receiving an imprecise invalidation *II*, to ensure that every attached interest set is still precise after applying *II*, α needs to detach any interest set that is made imprecise by *II*. Suppose the overlapping interest set of *II.target* and *S.attachedSS* is *KickedSet*, if *KickedSet* is not precise up to *II.end*, α must detach *KickedSet* from the stream *S*.

  In particular, the stream needs to do three things before it advances its *streamCVV* to *II.end*: (1) updates *S.attachedSS* to exclude *KickedSet* (Line 40); (2) removes its reference from *KickedSet.attachedStreams* (Line 39); and (3) advances *KickedSet.lastKnownLPVV* to include *streamCVV* and $(II.start - 1)$[1] (Line 38.)

  Note because of the stream's prefix property, the arrival of *II* implies that there are no updates between *S.streamCVV* and *II*'s *start*, therefore it is safe to update *KickedSet.lastKnownLPVV* to include $II.start - 1$.

- Upon receiving a catchup stream for interest set *catchupSS* starting from *catchupStartVV*, if *catchupStartVV* has any component larger than *catchupSS.lpVV*, α ignores the catchup stream because the gap might hide some precise invalidations (Line 23). Otherwise, α needs to track the status of the catchup stream and decide at the end of the catchup stream if it can attach *catchupSS* to *S*.

  A node maintains the status for a catchup stream for two reasons. First, the catchup stream might still contain imprecise invalidations for *catchupSS* because the sender might do not have all of the precise invalidations either. Therefore it might not be able to attach the *catchupSS* at the end of the catchup stream. Second, even if a catchup stream fails to attach *catchupSS* at the end, it might still make some progress for the *catchupSS.lpVV* before receiving the imprecise invalidation.

  To track the status for a catchup stream, *S* maintains a *pendingSS* to identify the objects expecting to join *S* and a *pendingCVV* to summarize the catchup progress.

[1] For simplicity, we use $VV - 1$ to represent the version vector of whom each entry's timestamp is one less than VV's.

As described in Section 5.1.1, a catchup stream starts with a *CatchupStart* message, which includes a subscription set *catchupSS* to identify which objects to catchup and a *catchupStartVV* to indicate the start time of the catchup stream.

The processing of a catchup stream is similar to the processing of a normal invalidation stream. It mainly involves updating the *pendingSS*, *pendingCVV*, and *catchupSS.lastKnownLPVV*. In particular, it handles four cases: (1) as indicated in Figure 5.4 line 22, when α receives a *CatchupStart* message, it updates the *pendingSS* and *pendingCVV* to *catchupSS* and *catchupStartVV* respectively; (2) similar to applying invalidations from the normal stream as described above, α advances the *pendingCVV* and updates per-object state when receiving precise invalidations in the catchup stream (from Line 42 to Line 46); (3) when α receives an imprecise invalidation *II* in the catchup stream, it removes *II.targetSet* from *pendingSS* and advances the removed set's per-interest set state *II.targetSet.lastKnownLPVV* to include both $(II.start - 1)$ and *pendingCVV* before advancing the *pendingCVV* to include *II.end* (from Line 36 to Line 40); (4)finally, when α receives the *CatchupEnd* message, α attaches the remaining *pendingSS* to *S* because it must have received all precise invalidations to *S.streamCVV* (Line 29 and Line 30).

5.2 Checkpoint Catchup

As indicated in Figure 5.2, to share the same stream, a new subscription request *(SS, startVV)* must catch up with the current status of the stream first. One way to catch up is to send the log from *startVV* to *streamCVV* as described in Section 5.1. An alternative is to send checkpoints—the status of objects in *SS* that were updated after *startVV* as of *streamCVV*.

Checkpoint catchup is needed for two reasons. First, it is needed for log garbage collection. Nodes can garbage collect any prefix of their logs, which allows each node to bound the amount local storage used for the log to any desired fraction of its total disk

space. If a node α garbage collects all log entries older than $\alpha.omitVV$ and another node β requests a subscription catchup with *startVV* older than $\alpha.omitVV$, then α can not bring β up by sending a catchup invalidation stream. Instead, α must send a checkpoint of its per-object state and interest set *lpVV*.

Second, checkpoint catchup is more efficient than log catchup in some cases. 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.

In existing server replication protocols [72], in order to ensure consistency, such a checkpoint exchange must atomically update receiver's state for all objects in the system. Otherwise, the prefix property and causal consistency invariants could be violated. Traditional checkpoint exchanges, therefore, may block interactive requests while the checkpoint is atomically assembled at the sender or applied at receiver, and they may waste system resources if a checkpoint transfer is started but fails to complete.

Imprecise invalidations yield an unexpected benefit: incremental checkpoint transfer. Rather than transferring information about all objects, an incremental checkpoint updates a subset of checkpoint.

5.2.1 Incremental Checkpoint Transfer Protocol

As indicated in Figure 5.2, a checkpoint catchup for *(SS, startVV)* includes (1) an imprecise invalidation that covers all objects in the system from the receiver's *currentVV* up to the sender's *currentVV*, (2) *SS.lastPreciseVV* if newer than *startVV*, and (3) the per-object state for any object in *SS* whose *acceptStamp* exceeds *startVV*. The purpose of (1) is to ensure the consistency of *SS* and other interest sets, which is the key to enable updating partial

checkpoint consistently. The receiver first applies (1) as a normal imprecise invalidation as described in Section 5.1.2. Then the receiver applies (2) to its corresponding per-interest set states and applies (3) to the corresponding per-object states. Thus, the receiver's state for *SS* is brought up to include the updates known to the sender, but other interest sets may become *IMPRECISE* to enforce consistency.

This checkpoint catchup algorithm yields three advantages over traditional checkpoint exchange algorithms [72]. First, it is incremental. Whereas existing checkpoint exchange must either give up causal consistency across objects like in WinFS [70] or atomically update the receiver's state for all objects in the system (otherwise, the prefix property and causal consistency invariants could be violated), our protocol can incrementally send subset of the checkpoint with an imprecise invalidation for the rest of the objects to ensure consistency when full replication of checkpoint is less likely succeed. Second, as illustrated in Figure 5.2, by simply adding the *CatchupStart* and *CatchupEnd* messages, this approach enables UR-Repl subscription protocol to smoothly integrate the checkpoint catchup option to implement the subscription abstraction as the checkpoint catchup is semantically equivalent to log catchup. Finally, the processing of checkpoints has less impact on local access performance. The receiver does not need to freeze the local read/write while it is uploading the checkpoint.

5.2.2 Discussion

Informally, most existing peer-to-peer consistent replica synchronization protocols fall into two families: log-based [72, 103] and state-based [70]. However, neither is perfect. Although the log-based approach can continuously synchronize updates and thereby provide stronger consistency and fewer conflicts, it adds additional storage overhead and needs a careful garbage-collection protocol [3, 32, 80]. State-based protocols require no extra storage, but there is no easy way to summarize the local state if the synchronization is interrupted, and they have to either give up consistency across objects [70] or require full

replication and blocking for checkpoint exchange [72].

UR-Repl seamlessly combines both worlds by using imprecise invalidations and multiplexing subscriptions. Policies can make the tradeoffs between the local extra log storage overhead and extra checkpoint network bandwidth. When both catchup options are available, policies can optimize the cost of an invalidation subscription by selecting either of the two semantically-equivalent catchup options.

5.3 Flexible Commit Mechanism

As a universal data replication architecture, simply implementing a single commit protocol is not sufficient since any of the state of the art commit protocol has limitations. For example, the Golding's algorithm [32] as described in Chapter 4 requires periodical heartbeat messages for liveness which might hurt availability when some nodes are disconnected and which may incur additional network bandwidth overhead. Although Bayou's primary server commit protocol [72] mitigates the availability issue and does not require heartbeat messages for liveness, it requires the primary server to issue a new sequence number to each update and thereby leads to write reordering, which complicates the protocol. In particular, it requires each node to rollback all uncommitted updates and remove the corresponding uncommitted update to insert each newly committed update.

To facilitate the implementation of different commit protocols, UR-Repl implements a flexible commit primitive via an *AssignSequence* interface to allow system designer to explicitly control when to commit an update and then leave the propagation of the commit operation to a special invalidation *sequence invalidation*. This *AssignSequence* interface has two parameters: a *targetSet* that specifies the target object(s) and a *targetAS* that identifies the accept stamp of the write to be sequenced. When this operation is called, similar to processing a write operation, the node generates an accept stamp called *sequenceStamp* and a special invalidation called *sequence invalidation*. Like a general invalidation, a *sequence invalidation* has a *targetSet* and an accept stamp which is the *sequenceStamp*. The only

difference between a *sequence invalidation* and a precise invalidation is that the sequence invalidation includes another accept stamp *targetAS* to identify the sequenced write.

Once generated, a *sequence invalidation* is propagated and processed in the system exactly the same way as a precise invalidation. For example, it can also be accumulated in an imprecise invalidation. The post condition of applying a *sequence invalidation* is that if the *targetSet*'s per-object state is not marked as $SEQUENCED$ and the accept stamp of the per-object state is *targetAS*, then the per-object state will be marked as $SEQUENCED$. Note that initially all per-object states are marked as $UNSEQUENCED$.

This commit primitive is flexible to implement different commit protocols for the *isSequenced* predicate. To demonstrate the flexibility and elegance of this mechanism, in the rest of this section, we explain how to implement the primary server commit protocol using this primitive.

Primary server commit. Besides Golding's algorithm [32] as described in Chapter 4, Bayou's primary commit protocol [72] is another useful commit protocol in server replication systems. By forcing all writes committed by one primary server, this protocol does not slow down the commit process due to lengthy disconnections of some replicas. In this protocol, one node is designated as the "primary" server that assigns a monotonically increasing commit sequence number (CSN) to each write. The CSN defines a total commit order for all writes. The other nodes send their updates in the partial causal order to the primary, and then the committed writes are propagated back among nodes in the committed total order after getting the CSNs from the "primary" server.

With the *AssignSequence* interface, implementing Bayou's CSN commit protocol is straightforward. First, a server commits writes when it sees them using *AssignSequence*. Second, all clients only read committed writes by setting the *ReadNowBlock* predicate to *isValid*, *isComplete* and *isSequenced*. Because all *sequence invalidations* are generated by one server and all writes are sequenced in the order they arrive, the *sequence invalidations* preserves the partial causal order among writes. By blocking a read until an object is pre-

cise, valid, and sequenced, each client sees the same view that is consistent to the sequenced order.

Note that our implementation of the primary server commit protocol does not reorder any updates because the invalidation subscription protocol and the *isSequenced* predicate for read naturally guarantee that every node's reads reflect the same view as if all updates are ordered by the commit sequences issued by the primary server.

5.4 Conflict Detection

Integrating the *incremental checkpoint catchup* into the invalidation log exchange protocol raises an issue for the *PrevAccept* conflict detection mechanism described in Chapter 4. This detection algorithm requires a node to receive all the precise invalidations of the object it is interested in. When falling back to checkpoint catchup, it could have false positives due to some missing old updates. This section first gives a brief survey of existing design choices of conflict detection mechanisms and then introduces a novel efficient conflict detection mechanisms *dependency summary vectors* that works efficiently for both log catchup and checkpoint catchup.

5.4.1 Design Choices

UR-Repl focuses on syntactic conflict detection based on the causal relationship [51] rather than relying on any application-specific semantics. In particular, any two updates to the same object that do not have any causality relationship are considered *conflicting*.

The state of art to detect conflicting writes defined by causality includes three main families:

1. *Previous stamps*. This approach [34] includes in each write the version just overwritten *previous stamp* and stores the *previous stamp* in the local per-object state. When receiving a write, a node compares the write's *previous stamp* to the per-object state

current time. If they mismatch, then a conflict is detected.

This approach can accurately detect all conflicts in any log exchange protocol that ensures the prefix property as we described in section 4.5, but it adds extra per-update stamp overhead for both storage and network bandwidth. More importantly, in the case when the log is truncated and a node falls back to checkpoint exchange, it could have false positives due to some missing old updates.

2. *Hash histories*. Kang et. al. [45] use *hash histories* to detect conflicts. Whenever the local state changes, a node creates a new hash summarizing the current entire state. Each node keeps a list of hashes ordered by generating times. Whenever a node α synchronizes its state with another node β, it looks up β's last hash in its own hash history. If the hash exists, then α's version is a newer version. Similarly, if β finds α's last hash in β's hash history, then β's version is a newer version. If neither of the last hashes exists in the other's history, then it is a conflict.

   Although the size of a hash history is independent of the number of replicas, it grows proportionally to the total number of updates. More importantly, because the hashes summarize the entire local state, it could have false negatives due to concurrent updates to different objects. Therefore, it requires complicated commit protocol to carefully garbage collect stable updates' hashes. More importantly, in the case when there are concurrent updates on two different nodes to two different objects, the last hash of either node will not be found in the other node's history, then it has a false negative.

3. *Version vectors*. Using *version vectors* is the most well-known and popular alternative [48, 74]. A version vector [44] accurately captures the causality relationship between updates. Two writes are conflicting if and only if neither of their version vectors dominates the other.

   Although this approach can accurately detect conflicts, it is expensive to maintain the per-object version vectors, especially in large-scale systems. In order to reduce the

version vector overhead, WinFS's *predecessor vectors with exceptions* (PVE) [60] uses one global *version vector with a list of exception stamps* to replace the per-object version vectors. But when a synchronization process is interrupted frequently, the exception list might grow indefinitely. To address this issue, a later approach [57] *vector sets* bounds the worst case cost to per-object version vector by grouping multiple objects and represents their state in a single version vector. Note that neither of these two approaches can provide causal consistency when a synchronization process is interrupted prematurely because a node might only receive subset of all changed objects.

5.4.2 Dependency Summary Vectors

UR-Repl conflict detection algorithm extends WinFS's *vector sets* algorithm to detect conflicts dynamically while processing invalidation streams.

Definition. A write W's *dependency summary vector (DSV)* is a version vector that summarizes all the updates on which W depends on. In particular, any version vector that satisfies the following two conditions is called W's DSV: (1) includes all the writes on $W.target$ that precedes W and (2) excludes any writes on $W.target$ that are causally ordered after W. For example, suppose all the causally ordered updates on object o are $1@\alpha$, $3@\alpha$, $10@\beta$. Regarding a write W with $acceptStamp = 3@\alpha$, $\langle 1@\alpha, 9@\beta \rangle$ is W's *DSV*, while both $\langle 0@\alpha, 9@\beta \rangle$ and $\langle 3@\alpha, 10@\beta \rangle$ are not because the first one does not include the causally preceding write $1@\alpha$ and the second one does not exclude the causally newer write $10@\beta$.

An object version can have multiple DSVs. For the same example described above, both $\langle 1@\alpha, 9@\beta \rangle$ and $\langle 2@\alpha, 6@\beta \rangle$ can be W's DSVs. Therefore, *dependency summary vectors (DSV)* can make conflict detection efficient by finding a common DSV for multiple objects to save the storage space and bandwidth cost.

If we know the *DSV*s of any two writes $w1$ and $w2$ that update the same object, we can detect conflicts according to the following three rules:

Message in an invalidation stream S	DSV
Normal precise invalidation inv	$S.streamCVV$
Precise invalidation inv in log catchup	$S.pendingCVV$
Checkpoint catchup $o \in IS$	$IS.LastPreciseVV$

Figure 5.5: Invalidation stream messages and their DSVs.

r1 If *w1.acceptStamp* is included by *w2.DSV*, then *w1* causally precedes *w2*.

r2 Similarly, if *w2.acceptStamp* is included by *w1.DSV*, then *w2* causally precedes *w1*..

r3 If neither *w2.DSV* includes *w1.acceptStamp* nor *w1.DSV* includes *w2.acceptStamp*, and *w1* and *w2* are not equal, then $w1$ and $w2$ conflict.

The key challenge is how to maintain the *DSV* in the local state efficiently and deliver this information in the invalidation stream efficiently.

Conflict detection in applying invalidation stream. UR-Repl leverages the cost already paid for maintaining causal consistency, i.e. *per-interest set lpVV*, to augment the protocol to detect conflicts without any additional cost. To ensure that the *LpVV* of an object can be used as the current object version's *DSV*, UR-Repl enforces two rules: (1) only apply a precise invalidation to the per-object state when the targeted object is precise and (2) block a local write until the object is precise.

By ensuring (1) and (2), the local stored newest version is guaranteed to be the newest version up to the object's current *LpVV*. If any of these two rules is violated, the node might miss some updates between *LpVV* and the version just applied.

Figure 5.5 summarizes the *DSV* of any update version received from an invalidation stream. For a checkpoint catchup stream CP, because the sender's $IS.LpVV$ is part of the checkpoint, by comparing the sender's *(CP[IS].LpVV, CP[o].acceptStamp)* with the receiver's *(IS.LpVV, o.acceptStamp)* according to the rules [r1] [r2] [r3], we can accurately detect any conflicts.

By taking advantage of the prefix property of invalidation streams, a node can derive the *DSV* for any arriving invalidation without the sender explicitly sending any vectors except the initial *StreamStart*. In particular, a stream's *streamCVV* summarizes all previous writes of the current sending invalidation *inv*, and any newer invalidations to the same object should not be sent before *inv* by the prefix property. Therefore, *streamCVV* can serve as the *DSV* of *inv*. When receiving a normal precise invalidation *inv* from the stream, the receiver simply compares *(streamCVV, inv.acceptStamp)* and *((inv.targetSet).LpVV, per-object state[inv.targetSet].acceptStamp)* to detect conflicts. Similarly, when receiving a catchup precise invalidation *inv*, the receiver uses the *pendingCVV* as the *DSV* of *inv*.

5.5 Evaluation

As a unified replication protocol, UR-Repl should be *flexible* enough to construct a broad range of systems and *efficient* so that the costs associated with building a system with UR-Repl is proportional to the demands of the system, i.e. the extra cost for generality is minimal. We will demonstrate the flexibility of UR-Repl in the next two chapters. In this section, we focus on the efficiency of UR-Repl. In particular, we quantify the efficiency of subscriptions and conflict detection.

5.5.1 Cost for Subscriptions

Experimental environment. The prototype implementation is written in Java. 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. We use Fedora Core 6, BEA JRocket JVM, and Berkeley DB Java Edition 3.2.23.

Our primary performance goal is to minimize network overheads. We focus on network costs for two reasons. First, we want UR-Repl to be useful for network-limited environments. Second, if network costs are close to the ideal, it would be evidence that UR-Repl captures the right abstractions for constructing replication systems.

	Best Case	UR-Repl 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}	$\hat{S}_{body}$

Figure 5.6: Network overheads breakdown. 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.

Network Efficiency

Figure 5.5.1 shows the cost model of our implementation of UR-Repl's communication abstractions and compares these costs to the costs of hypothetical 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, UR-Repl invalidation 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 [105]. Note that in our prototype this cost is amor-

tized 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* [72] 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. 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 as we demonstrated in Section 4.6. 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.

UR-Repl vs. CR-Repl. We first compare the cost of establishing callbacks on UR-Repl and on the implementation of CR-Repl described in previous chapter. We approximate ideal callbacks by establishing single-object subscriptions over both protocols. Figure 5.7 shows the cost for synchronizing the updates to 1000 object in a 100-node system (1) using single-object interest sets and (2) using a single interest set spanning all 1000 objects. We consider the ideal cost of a callback in a client-server system as sending an object Id and receiving the object and the timestamp.

As Figure 5.7 shows, when fine-grained callbacks are established, UR-Repl approximates callbacks and is an order of magnitude cheaper than the CR-Repl replication.

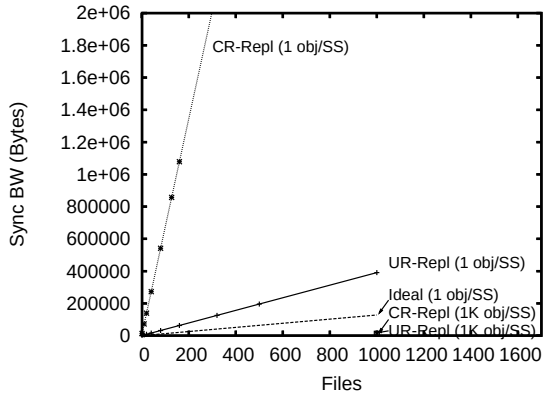


Figure 5.7: Bandwidth for subscribing to varying number of 1-object interest sets for UR-Repl and CR-Repl .

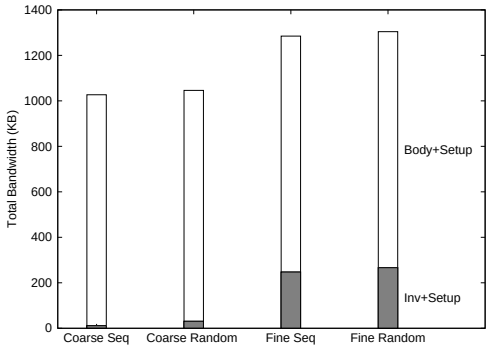


Figure 5.8: Network bandwidth cost to synchronize 1000 10KB files, 100 of which are modified.

The CR-Repl replication pays a higher cost because every subscription establishes a new, independent connection which involves sending a version vector summarizing the current state of IS and then receiving an imprecise invalidation describing all invalidations to objects not in IS. In addition, coarse-grained subscriptions have much less overhead than fine-grained subscriptions because version vectors are only transmitted once when establishing the coarse-grained subscription instead of each time a fine-grained subscription is established.

Figure 5.8 and Figure 5.10 shows the impact of the size of a subscription set on

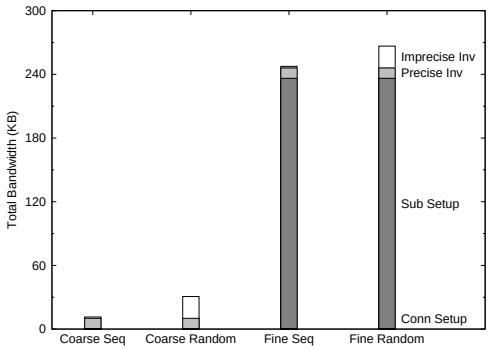


Figure 5.9: Invalidation overheads breakdown to synchronize 1000 10KB files, 100 of which are modified.

the subscription cost. By varying the size of a subscription set, i.e. the number of objects included in one subscription, invalidation subscriptions provide trade-offs between the cost of setting up subscriptions and the cost of sending invalidations for objects in a large subscription set that are not of immediate interest. Given the same total number of objects that need to create callbacks, the size of one subscription set affects the overall bandwidth overhead because the larger the size, the less information needed to send for objects outside of the subscription set on one subscription.

Subscription cost breakdown. Figure 5.8 illustrates the overall synchronization cost and Figure 5.9 illustrates the breakdown of invalidation 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.

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 UR-Repl over the best-case due to connection setup and imprecise invalidations 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.

Log catchup vs. checkpoint catchup. More generally, our implementation efficiently implements both fine-grained and coarse-grained subscriptions by taking advantage of both log catchup and checkpoint catchup. As discussed in Section 5.1 an ideal implementation of an invalidation subscription will send *CatchupStart* message when it is established, and a *CatchupEnd* message once the past invalidations or the checkpoint has been sent. Each of these messages can be as small as a single byte. In UR-Repl, since multiple subscriptions are multiplexed on a single stream, the *CatchupStart* and *CatchupEnd* messages contain the encoding of the associated subscription set.

Figure 5.10 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 [42].

The figure demonstrates that the cost of establishing subscriptions for the log catchup

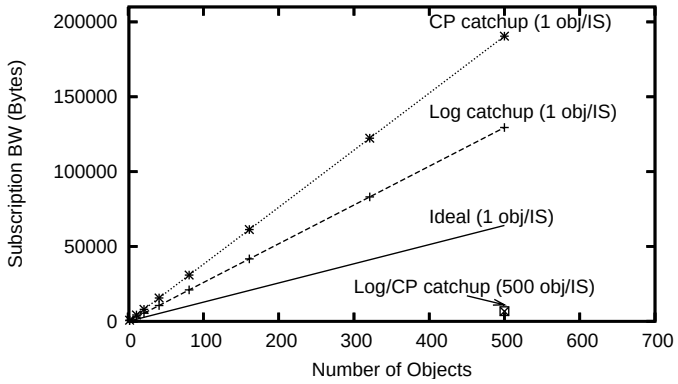


Figure 5.10: Bandwidth for establishing invalidation subscriptions.

and the checkpoint catchup is within a factor of the ideal implementation. The overhead can be attributable to the size of the *CatchupStart* message. The checkpoint catchup does worse than the log catchup because the size of the invalidation meta-data for each object is bigger than an actual invalidation sent during the log catchup.

We also quantify the cost of establishing a single coarse-grained subscription for all objects. The cost of a coarse-grained log and checkpoint catchup is almost the same. Both fairly better than the ideal because the *CatchupStart* and *CatchupEnd* messages are only sent once instead of 500 times.

Consistency overhead. Invalidation subscriptions also have the additional overhead of *imprecise invalidations* sent to maintain consistency information. Figure 5.11 quantifies this overhead when compared to a system that does not send any consistency ordering information. We compared the overheads under three workloads. As we can see from Figure 5.11, even in the worst case, the overhead for maintaining consistency is at most 2x the number of invalidations sent over the subscription.

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

Figure 5.11: 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
UR-Repl object store	8.47	1.27	0.25	0.16

Figure 5.12: Read/write performance for 1KB objects/files in ms.

Performance Overheads

This section examines the performance of the UR-Repl 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, UR-Repl is a relatively untuned prototype rather than well-tuned production code. Second,our implementation emphasizes portability and simplicity, so UR-Repl is written in Java and stores data using BerkeleyDB rather than running on bare metal. Third, UR-Repl provides additional functionality such as tracking consistency metadata not required by a local file system.

Figures 5.12 and 5.13 summarize the performance for reading or writing 1KB or 100KB objects stored locally in UR-Repl 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.

	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
UR-Repl object store	52.43	43.08	0.90	0.35

Figure 5.13: Read/write performance for 100KB objects/files in ms.

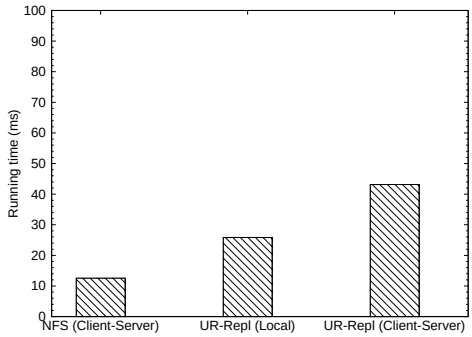


Figure 5.14: Performance for Andrew benchmark.

Figure 5.14 depicts the time required to run the Andrew benchmark [42] over the UR-Repl prototype via the Java NFS wrapper that we will describe in Chapter 6. UR-Repl successfully runs the benchmark, but it is slower than a well-tuned local file system.

	Previous stamps	Version vectors	PVE	Vector sets	DSV
Storage overhead lower bound	$O(N+q+R)$	$O(N \times R)$	$O(N+R)$	$O(N+R)$	$O(N+R)$
Storage overhead upper bound	$O(N+Q+R)$	$O(N \times R)$	unbounded	$O(N \times R)$	$O(N \times R)$
Communication overhead lower bound	$O(q+R)$	$O(q \times R)$	$O(q+R)$	$O(q \times R)$	$O(q+R)$
Communication overhead upper bound	$O(q+R)$	$O(N \times R)$	unbounded	$O(N \times R)$	$O(q \times R)$

Figure 5.15: Storage and communication overhead lower and upper bounds comparison of existing approaches.

5.5.2 Cost for Conflict Detection

Figure 5.15 compares the storage and communication overhead of the various conflict detection approaches in terms of N objects, R replicas, and q recent updates. It compares CR-Repl's previous stamp approach, traditional version vectors, WinFS's predecessor vectors with exceptions (PVE), WinFS's vector sets and UR-Repl's *dependency summary vectors* (DSV).

Under normal low-fault situations, in which synchronization occurs in one go, DSV, like the vector sets and concise version vectors, maintains and communicates as little as a single version vector to summarize all previous updates for all objects. Therefore, the storage cost lower bound of these approaches is one version vector plus one acceptStamp per object, i.e. $O(N + R)$, and the communication cost lower bound is per update stamp plus one version vector, i.e. $O(R + q)$.

Under severe communication disruptions, there is extra network and storage costs for some approaches because of the increase in size of book-keeping information. The worst case storage cost for both DSV and vector sets is a per-object version vector which is no worse than traditional version vectors. However, DSV's network cost is better than the vector sets and traditional version vectors because it takes advantage of the prefix properties of invalidation subscriptions and only needs to send the per update version vector instead of per-object version vector. The upper bounds of both storage and communication overhead for the concise version vector approach are unbounded because of the size of a concise version vector can be unbounded due to exceptions. Note that although, analytically, the previous stamp approach could outperform DSV, practically, it is unable to detect conflicts during checkpoint catchup whereas all other approaches can.

Chapter 6

Flexibility

A core hypothesis of our work is that most of existing systems are "special cases" of a set of underlying mechanisms and that UR-Repl is the set of mechanisms for such a replication microkernel. We study this question in two steps. First we study this question "in the small" and demonstrate the flexibility of URA by mapping a range of existing protocols in terms of our lower-level mechanism abstractions. Second, we study this question "in the large" by explaining how to put those existing protocols together to build a few case-study systems on top of URA.

We defer the description of case-study systems to the next chapter. This chapter focuses on explaining how to mapping existing techniques to URA abstractions. In particular, Section 6.1 discusses how to implement a range of consistency semantics using URA local API. Then Section 6.2 explains how to use the subscription abstractions to implement callbacks leases. Finally Section 6.3 demonstrates how to support quorums over URA.

6.1 Flexible Consistency

This section discusses the crosscutting issue of how to provide flexible consistency that (a) supports strong consistency semantics for those applications that require them and (b) does

not introduce unnecessary overhead for applications that do not.

Enforcing cache consistency entails fundamental trade-offs. For example the CAP dilemma states that a replication system that provides sequential **C**onsistency cannot simultaneously provide 100% **A**vailability in an environment that can be **P**artitioned [31, 84]. Similarly, Lipton and Sandberg describe fundamental consistency v. performance trade-offs [54].

A system that seeks to support flexible consistency must therefore do two things. First, it must allow a range of consistency guarantees to be enforced. Second, it must ensure that workloads only pay for the consistency guarantees they actually need.

This section first describes the range of consistency guarantees that URA provides and proves the correctness. It then discusses how URA provides these guarantees without introducing unnecessary overhead.

6.1.1 Providing Flexible Guarantees.

Discussing the semantic guarantees of large-scale replication systems requires careful distinctions along several dimensions. *Consistency* constrains the order that updates across multiple memory locations become observable to nodes in the system, while *coherence* constrains the order that updates to a single location become observable but does not additionally constrain the ordering of updates across multiple locations [39]. *Staleness* constrains the real-time delay from when a write completes until it becomes observable.

Our protocol provides considerable flexibility along all three of these dimensions.

Causal Coherence

With respect to coherence, although our default read interface enforces causal consistency, the interface allows programs that do not demand cross-object consistency to issue *imprecise reads* by setting the *ReadNowBlock* predicate to *isValid*. Imprecise reads may achieve higher availability and performance than precise reads because they can return without wait-

ing for an interest set to become *PRECISE*. Imprecise reads thus observe causal coherence (causally coherent ordering of reads and writes for any individual item) rather than causal consistency (causally consistent ordering of reads and writes across all items.)

With respect to consistency and staleness, URA provides a range of traditional guarantees such as the relatively weak constraints of causal consistency [43, 51] or delta coherence [85], to the stronger constraints of sequential consistency [52] or linearizability [40]. Further, it provides a continuous range of guarantees between causal consistency, sequential consistency, and linearizability by supporting TACT's order error for bounding inconsistency and temporal error for bounding staleness [103].

The read/write interface provides best-effort coherence and causal consistency by setting the read interface parameters. The stronger guarantees of sequential consistency and linearizability make use of "wrapper" interfaces over the default interface.

Causal Consistency

We enforce the causal consistency defined by Hutto and Ahamad [43]: "causal memory that requires all processors to agree on the order of causally related effects (writes) but allows events not related by potential causality to be observed in differing orders."

URA provides causal consistency by simply setting the *ReadNowBlock* predicate to *isValid* and *isComplete*. By blocking the read until the target object is $PRECISE$ and $VALID$, URA provides causal consistency by enforcing three constraints.

- (C1) *Use of Lamport logical clock.* UR-Repl uses Lamport logical clock to capture the causal relationship between writes.
- (C2) *Prefix property of invalidation propagation.* The invalidation propagation protocol maintains an invariant: if a node's state reflects the i'th invalidation by some node n, then the node's state reflects all earlier invalidations by n precisely or imprecisely.
- (C3) *Body apply rule.* A body can not be applied until the corresponding invalidation

has arrived. Any older body is overwritten by a newer body for the same object. The "newer" and "older" are defined by the corresponding lamport logical clock and breaking ties by the original writer's node ID.

We now prove that the default interface provides causal consistency.

Lemma 1. *An object o on node α is* precise *only if α's local state of o has reflected all the precise invalidations related to o covered by α's current version vector $currentVV$.*

Proof. We prove lemma 1 in two steps. First, we prove the simple case where there is no checkpoint catchup. Suppose o is precise, and there is a missing precise invalidation pi s.t. pi's acceptStamp ($as@\beta$) is included in $currentVV$. It's obvious that $\beta \neq \alpha$. By $C2$, α must have received at least an imprecise invalidation ii such that ii covers pi. Since pi is missing, every invalidation received by α that overlaps pi's logical time region is imprecise and its *target* overlaps o. According to the per-interest set state update rules as described in Section 4.3 or Section 5.1.2, $o.lpVV$ will never advance to include ($as@\beta$). Therefore, $currentVV > o.lpVV$. e.g. o is IMPRECISE which contradicts the assumption.

Now consider the case where a checkpoint related to o is sent for the first time, because the checkpoint of o reflects all precise invalidations to o up to the checkpoint's $lpVV$, the receiver's state also reflects the same set of invalidations after applying the checkpoint. After the catchup, the normal invalidation exchange protocol follows. Therefore, it is not hard to derive that Lemma 1 still holds in the presence of checkpoint catchup. □

Theorem 1. *Blocking reads of INVALID and IMPRECISE ensures causal view of writes.*

Proof. We can prove this theorem by contradiction. In particular, we prove that the following scenario would never happen if we block reads of INVALID and IMPRECISE: suppose for some write $W1$ ($as1@n1$, $o1$) causally precedes write $W2$ ($as2@n2$, $o2$), i.e. $W1 \rightarrow W2$, there exists a node $n4$ such that its first read $R1$ of $o2$ returns $W2$ and its following read $R2$ of $o1$ returns $W3$ ($as3@n3$, $o1$) where $as3@n3 < as1@n1$.

By C1, we have $as1@n1 < as2@n2$. Now we prove that there is no such node in both of the following two cases.

1. $o1 = o2$.

   By C3, $R2$ always returns a value which is at least as new as the previous read $R1$ value, i.e. $as3@n3 \geq as2@n2$, therefore $as3@n3 > as1@alpha$ which contradicts with the assumption.

2. $o1 \neq o2$.

   Since $W1$ causally precedes $W2$, either (1) $W1$ happens before $W2$ at the same node; or (2) $W1$ has been propagated to the node where $W2$ is first issued; or (3) there exists Wi s.t. $W1 \rightarrow Wi$ and $Wi \rightarrow W2$.

   In the first two cases, $W1$ is already in $n2$'s log when $W2$ is inserted to the same log. By C2, we can easily derive that this property still holds in the third case. Hence $W1$ will always be the prefix of $W2$ in any stream, i.e. whenever a node learns of $W2$, the node must have also learned of $W1$. Therefore, when the first read returns $W2$, the $currentVV$ should have already included $W1$. By lemma 1, when the second read returns, the node should have reflected all the precise invalidations related to $o1$ up to $currentVV' \geq currentVV$ which includes $W1$. Therefore, $as3@n3 \geq as1@n1$ which contradicts with the assumption.

Therefore blocking reads of INVALID and IMPRECISE reflects the causal order of writes.

□

Sequential Consistency

Lamport [52] defines sequential consistency as follows: "the result of any execution is the same as if the [read and write] operations by all processes were executed in some sequential

order and the operations of each processor appear in this sequence in the order specified by its program."

In this section, we describe how to implement *sequential consistency* on URA.

For simplicity, here we use Golding's algorithm as described in Chapter 4 to implement consistency semantics discussed in this Chapter. Recall that the accept stamps assigned to each write and the commit protocol define a total order on all writes that is consistent with the program order of writes. Further recall the commit protocol ensures that eventually all nodes agree on this total order for a prefix of all writes [88]. We can thus enforce sequential consistency by ensuring (1) that reads only observe committed writes and (2) that any reads by a program appears after all preceding writes by that program in a total order.

Implementation. The *SequentialLocalInterface* wrapper may delay completion of reads and writes in order to meet the constraints of sequential consistency. It ensures the above two conditions as follows.

After the wrapper issues a write to the URA default local interface, the wrapper stores the *acceptStamp* assigned to that write in the variable *lastLocalWrite*.

Then, a read via the wrapper proceeds as follows:

- (*ReadBlock1*) The wrapper blocks until the *lastLocalWrite* is committed.
- The wrapper issues a read using URA's default causal consistency read interface which returns a *body* and the accept stamp *observedAcceptStamp* of the write that was just observed.
- (*ReadBlock2*) The wrapper blocks until *observedAcceptStamp* is committed.

Note: the first block makes sure that the read is ordered after any preceding local write in the global serialization order. As illustrated in Figure 6.1, if it does not block until all the previous local writes (e.g., W1 $1@n2$) commit, it is possible that some writes (e.g.,

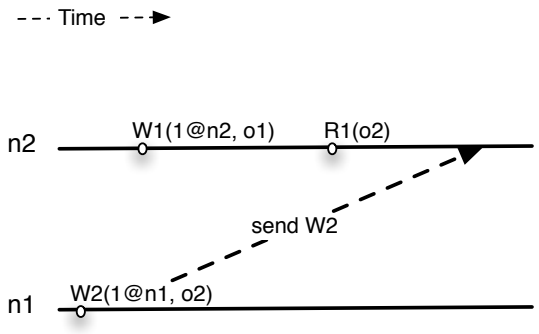


Figure 6.1: Sequential consistency violation example 1. The committed order is $W2 < W1$. However, $R1 < W2$ because $R1$ returns older value than W2. Therefore $R1 < W1$ which contradicts the program order.

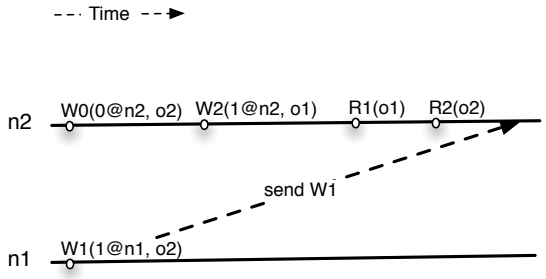


Figure 6.2: Sequential consistency violation example 2. The committed order is $W1 < W2 < R1$. However, $R2 < W1$ as $R2$ reads an older value than $W1$. Therefore $R2 < R1$ which contradicts with their program order.

W2 $1@n1$) ordered before the last local write (e.g., W1) in the global order defined by the accept stamps have not arrived yet. Then the read R1 returns an value older than W2 since W2 arrives after the read. Therefore in the corresponding sequential order, the read should be ordered before the writes which precedes the last local write, i.e, $R1 < W2$. Then because $W2 < W1$, by transitiveness we have $R1 < W1$, i.e., the read is ordered before the last local write which is inconsistent with the program order.

Similarly, the third block makes sure that the order between reads across multiple objects are preserved as defined in the program. Otherwise, as illustrated in in Figure 6.2,

the sequential order of reads (ordered by the read values) might violate the program order. In this example, the first read $R1$ on object $o1$ observes $W2$ while $W2$ is not committed locally. Because $W2$ is not committed, $W1$ which precedes $W2$ in the committed order might have not arrived to node $n3$ yet. If $R1$ returns $W2$ without waiting for $W2$ to be committed, the following read $R2$ on $o2$ will return an older value $W0$ instead of $W1$. Therefore, we have $R2 < W1$. Since $W1 < W2$ and $W2 < R1$, we have $R2 < R1$ which is inconsistent with the program order.

Correctness. In order to prove that the *SequentialLocalInterface* wrapper ensures sequential consistency, we first define a four-member tuple for each operation.

- Mark a WRITE with an acceptStamp $(time, nodeId)$ as $(time, nodeId, -\infty, -\infty)$.
- Mark a READ that is issued on node *readNode* at real time t, and that returns at the moment when the maximum committed timestamp at *readNode* is *(maxStamp, maxNodeId)* as *(maxStamp, maxNodeId, readNode, t)*.

We define the order between the four-member tuples as:

$(as1, nid1, rnd1, rno1) < (as2, nid2, rnd2, rno2)$

iff ($as1 < as2$)

or (($as1$ = $as2$) and ($nid1 < nid2$))

or (($as1$ = $as2$) and ($nid1$ = $nid2$) and ($rnd1 < rnd2$))

or (($as1$ = $as2$) and ($nid1$ = $nid2$) and ($rnd1$ = $rnd2$)) and ($rno1 < rno2$))

These tuples give a global serialization order S for all reads and writes. Note that we use ($readNode$, t) to distinguish reads with the same last committed write. The order of those reads does not affect the read results. We use $(-\infty, -\infty)$ for writes so that writes can always order before the corresponding reads in the sequential order.

Lemma 2. *The reads and writes issued through the* SequentialLocalInterface *are consistent with the global sequential order of* S.

Proof. First, it is obvious that this serialization order preserves the total order defined by the write accept stamps and the write commit protocol, therefore the writes are consistent with this sequential order.

Now let us prove that the read through *SequentialLocalInterface* wrapper returns the same value as executing it at one node according to S.

Suppose at node α, a *SequentialLocalInterface* read on object o returns when the write with the maximum commit accept stamp on node α is $maxCommittedWrite$. By *ReadBlock1* and *ReadBlock2*, the value returned to the read is the last write W issued to o that precedes or equals to $maxCommittedWrite$.

Now let us simulate the sequential order according to S, i.e., execute all the operations at one node in the order of S. As this read is ordered right after $maxCommittedWrite$ (there might be some other reads in between, but they will not affect the value this read returns), it is executed right after $maxCommittedWrite$. Since the writes in S are consistent with the order defined by their accept stamps, the write W is the last executed after all the other writes that update o before the read. Therefore the read returns the same value as the read through the *SequentialLocalInterface* wrapper. □

Lemma 3. *This global serialization order S is consistent with each node's program order P.*

Proof. For any two writes $W1(as1, n1 - \infty, -\infty)$, $W2(as2, n1, -\infty, -\infty)$, and any two reads $R1(as3, n3, n1rno3)$, $R2(as4, n4, n1rno4)$ in a program P running at node $n1$, we consider the following four cases:

1. $W1$ precedes $W2$ in P
   $\Rightarrow as1 < as2$, by lamport logical clock
   $\Rightarrow (as1, n1) < (as2, n1)$
   $\Rightarrow (as1, n1 - \infty, -\infty) < (as2, n1 - \infty, -\infty)$
   $\Rightarrow W1$ precedes $W2$ in S

2. $W1$ precedes $R1$ in P

   $\Rightarrow (as1, n1) \leq (as3, n3)$, by *ReadBlock1*, $W1$ must have committed before $R1$ returns

   $\Rightarrow (as1, n1 - \infty, -\infty) < (as3, n3, n1rno3)$

   $\Rightarrow W1$ precedes $R1$ in S

3. $R1$ precedes $W1$ in P

   $\Rightarrow (as1, n1) > (as3, n3)$, By *ReadBlock2*, write $(as3, n3)$ must have been committed at $n1$ before issuing $W1$

   $\Rightarrow (as1, n1 - \infty, -\infty) > (as3, n3, n1rno3)$

   $\Rightarrow R1$ precedes $W1$ in S

4. $R1$ precedes $R2$ in P

   $\Rightarrow rno3 < rno4$ and $(as3, n3) \leq (as4, n4)$, because the maximum committed accept stamp at one node is monotonically increasing.

   $\Rightarrow (as3, n3, n1rno3) < (as4, n4, n1rno4)$

   $\Rightarrow R1$ precedes $R2$ in S

Therefore the order of each individual program's operations in S preserves the order specified by the program. □

Theorem 2. *The reads through the* SequentialLocalInterface *wrapper observe sequential consistency of all writes issued.*

Proof. From Lemma 2 and Lemma 3, by the definition of sequential consistency in [52], we can conclude that the above algorithm can support sequential consistency. □

Linearizability

Herlihy and Wing define Linearizability as follows [40]: an concurrent execution is linearizable if it satisfies: (1) "processes act as if they were interleaved at the granularity of

complete operations" and (2) " this apparent sequential interleaving respects the real-time precedence ordering of operations".

The above *SequentialLocalInterface* ensures the first condition. In order to ensure the second condition, we must make sure the write/read order preserves the real-time precedence. Especially, we must make sure that (1) any write $W1$ that finishes before another write $W2$ is started in real time appears before $W2$ in the committed order; (2) any read appears in the total order after all writes that finish before the read is started and (3) any read $R2$ that is requested after another read $R1$ returns in another node must observe the same or newer view.

Implementation. The *LinearizableLocalInterface* wrapper may delay completion of reads and writes in order to ensure these conditions as follows.

- (*WriteBlock1*) each write blocks until the write is visible at all nodes. A node $n1$ determines that its write w is *visible* at another node $n2$ by issuing a *synchronization request* to $n2$ and waiting for the requested acknowledgement to arrive. $n2$ will send back an acknowledgement to $n1$ as soon as its *currentVV* includes $w.acceptStamp$.

  A read via the wrapper proceeds as follows:

- (*ReadBlock1*) the wrapper blocks until the newest write a node learned *maxKnownWrite* is committed locally;

- The wrapper issues a read using URA's default causal consistency interface that returns a *body* and the accept stamp *observedAcceptStamp* of the write that was just observed;

- (*ReadBlock2*) blocks until *observedAcceptStamp* is visible at all nodes.

Note that the *WriteBlock1* and *ReadBlock1* is necessary to ensure that read happens after all real-time preceding writes W. As illustrated in Figure 6.3, if $W2$ does not wait for

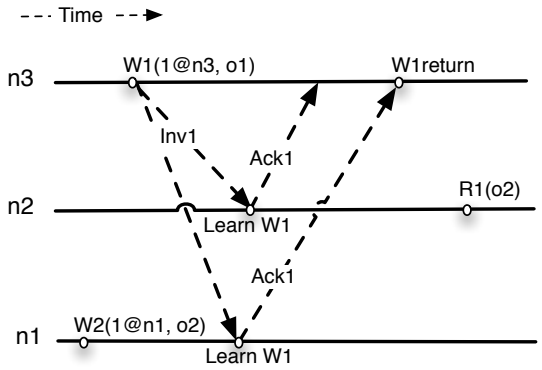


Figure 6.3: Linearizability violation example. The committed order is $W2 < W1$, but $R1 < W2$ because $R1$ returns an value older than $W2$. Therefore $R1 < W1$ which violates the real-time order.

$n2$ to get the invalidations, $R1$ will return a value of $o2$ older than $W2$. Since $W1$ returns earlier than $R1$, $R1$ should not return a value older than $W2$. Therefore the scenario in Figure 6.3 violates the semantics of *linearizability*.

Note that *ReadBlock2* is necessary to make sure that reads at different nodes returns values consistent to their real-time order. When a read returns a value, the following read should returns a value which relates to the same or newer committed point.

Correctness. Similar to the proof of *sequential consistency*, we also assign each read/write operation a four-member tuple and prove the theorem in two steps. First, we prove that the effects of any read/write operation through the *LinearizableLocalInterface* wrapper is consistent with the global serialization order "S" defined by the four-member tuple. Second, we prove that the global serialization order "S" is consistent with the real-time precedence order.

Lemma 4. *The* LinearizableLocalInterface *wrapper is consistent with the global serialization order defined by "S".*

We skip the proof which is similar to the proof of lemma 2.

Lemma 5. *The global serialization order "S" is consistent with the real-time precedence order.*

Proof. For any two writes $W1 : (as1, n1, -\infty, -\infty)$, $W2 : (as2, n2, -\infty, -\infty)$, and any two reads $R1 : (as3, n3, rn3, rno3)$, $R2 : (as4, n4, rn4, rno4)$, we consider the following four cases:

1. $W1$ precedes $W2$ in real-time

   $\Rightarrow as1 < as2$, by *WriteBlock1*, $W1$ returns before $W2$, therefore $n2$ knows $W1$ before issuing the accept stamp to $W2$

   $\Rightarrow (as1, n1 - \infty, -\infty) < (as2, n1 - \infty, -\infty)$

   $\Rightarrow W1$ precedes $W2$ in S

2. $W1$ precedes $R1$ in real-time

   $\Rightarrow (as1, n1) \leq (as3, n3)$, by *WriteBlock1*, $W1$ should have been known at $rn3$ when $W1$ returns, therefore by *ReadBlock1* $W1$ must have committed before $R1$ returns

   $\Rightarrow (as1, n1, -\infty, -\infty) < (as3, n3, nr3, rno3)$

   $\Rightarrow W1$ precedes $R1$ in S

3. $R1$ precedes $W1$ in real-time

   $\Rightarrow as1 > as3$, by *ReadBlock2*, write $(as3, n3)$ is known at $n1$ when $W1$ is issued

   $\Rightarrow (as1, n1, -\infty, -\infty) > (as3, n3, rn3, rno3)$

   $\Rightarrow R1$ precedes $W1$ in S

4. $R1$ precedes $R2$ in real-time

   $\Rightarrow (as3, n3)$ is known at $n4$ when $R2$ is issued by *ReadBlock2*

   $\Rightarrow (as4, n4) \geq (as3, n3)$

   $\Rightarrow (as3, n3, rn3, rno3) < (as4, n4, rn4, rno4)$

   $\Rightarrow R1$ precedes $R2$ in S

□

Theorem 3. *Each read/write operation through the* LinearizableLocalInterface *wrapper is linearizable with respect to all other reads and writes through the same interface.*

Proof. From Lemma 4 and Lemma 5, we can conclude that the above algorithm can support linearizability. □

Order Error Bound and Temporal Error Bound

Besides the above commonly-used consistency wrappers, URA also support TACT's [103] tunable temporal error (TE) bound and order error (OE) bound. The read interface has a *TE parameter* to restrict staleness by specifying the maximum real time delay between a remote update and a read. To support TE, every node maintains a real-time version vector $currentTime$ that is updated when an invalidation is received. A read is blocked if any component of the real-time version vector is older than $currentTime - TE$. A node sends periodic heartbeats to its peers when there are no updates to prevent reads from blocking unnecessarily.

Similarly, UR-Repl's write interface has an *OE* parameter that specifies the maximum number of outstanding, uncommitted local writes. We employ Golding's algorithm [32] for write commit. As described in Chapter 4, every node uses its *currentVV* to determine how many writes have not been committed, providing a means to block a write if the number is greater than that specified by the *OE* parameter.

We generalize the TE interface to support the *maxStale(nodes, count, t)* predicate and generalize the OE interface to support the *propagated(nodes, count, p)* predicate as listed in Figure 3.2.

6.1.2 Costs of Consistency

URA protocols should ensure that workloads only pay for the semantic guarantees they need. Our protocol does so by distinguishing the availability and response time costs paid

by read and write requests from the bandwidth overhead paid by invalidation propagation.

The read interface allows each read to specify its consistency and staleness requirements. Therefore, a read does not block unless *that read* requires the local node to gather more recent invalidations and updates than it already has. Similarly, most writes complete locally, and a write only blocks to synchronize with other nodes if *that write* requires it. Therefore, as in TACT [103], the performance/availability versus consistency dilemmas are resolved on a per-read, per-write basis.

Conversely, all invalidations that propagate through the system carry sufficient information that a later read can determine what missing updates must be fetched to ensure the consistency or staleness level the read demands. Therefore, the system may pay an extra cost: if a deployment never needs strong consistency, then our protocol may propagate some bookkeeping information that is never used. We believe this cost is acceptable for two reasons: (1) other features of the design—separation of invalidations from bodies and imprecise invalidations—minimize the amount of extra data transferred; and (2) we believe the bandwidth costs of consistency are less important than the availability and response time costs. Experiments in Section 4.6 quantify these bandwidth costs, and we argue that they are not significant.

6.2 Callbacks And Leases

Callbacks and leases are important techniques widely used in a range of client-server caching systems [48, 42, 68, 12, 65, 100]. In this section, we discuss how to map these techniques to URA.

6.2.1 Callbacks

Callbacks have long been studied for distributed file systems [48, 42, 68, 12]. In traditional callback algorithms, servers (or parents in hierarchical systems) keep track of which clients (or children) are caching what objects and promise to notify the clients whenever

an update to the cached object occurs. Callbacks save network bandwidth by only sending invalidations of the objects that a client currently caches.

URA generalizes "callbacks" to ad-hoc communication topologies. But without the client-server or hierarchy topology assumption, it is a challenge to save network bandwidth. For example, in the client-server systems, the server (or parent) does not need to send invalidations of updates to other objects that a client does not currently cache while still maintaining cross-object consistency because it has all the newest objects and the client only renews the invalidated object from the server. As another example, in the client-server systems, once the server sends an invalidation for an object to the client, it does not have to send other invalidations about that object because it knows that the client no longer has the data, i.e., a "callback break". However, in an arbitrary topology, a node can not decide if the other node's cache is still marked as "INVALID" because the destination may have gotten the new version from someone else between the last invalidation and "now".

Implementation on URA

URA generalizes the notion of establishing a callback with a server to adding (attaching) an interest set to a stream and the "callback break" as removing (detaching) an interest set from a stream.

For example, on a read miss for object o, a client takes two actions. First, it issues a single-object fetch for the current body of o. Second, it creates a "callback" on the server by setting up an invalidation subscription for object o starting from $o.lpVV$ from the server so that the server will notify the client if o is updated. As a result of these actions, the client will receive the current version of o, receive previous updates from $o.lpVV$ up to server's current time, and receive an invalidation when o is next modified; the server will block any new update until the client with callbacks receives the corresponding invalidation.

The invalidation subscription invariant—sending precise invalidations for subscribed objects and imprecise invalidations for other objects simplifies the implementation of "call-

back break". In particular, once a client α removes o from the stream, the server will send imprecise invalidations when o is updated later. Once α receives an imprecise invalidation for o, the next read of o with $isValid$ and $isComplete$ predicates on α will trigger a read miss because o is imprecise. As a result, a callback for o as described above will be established. Therefore, to break a callback, the server only needs to remove o from the outgoing stream to the client.

As illustrated in Chapter 5, URA minimizes the *callbacks* cost by using two techniques. First, it employs imprecise invalidations to concisely summarize missing information. Second, it multiplexes multiple subscriptions over the same network connection so as to only send information relating to any update once per connection rather than repeatedly sending this bookkeeping information once per subscription. As demonstrated in Section 5.5, it is an order of magnitude more efficient than the CR-Repl replication, and performs close to traditional callbacks or even outperforms traditional callbacks when the workload exhibits high predictability and locality.

Callback recovery. Using UR-Repl's invalidation subscriptions to implement *callbacks* yields a significant advantage: it makes "recovery" a natural implication of UR-Repl mechanisms, especially when compared to traditional protocols where "recovery" is treated as a special case. Recovery in most systems simply entails restoring lost connections and the re-establishment of the invalidation subscription and the *isComplete* predicate on reads automatically ensures that local state reflects any updates that were missed.

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, URA's invalidation subscription protocol advances the client's consistency state only for objects whose subscriptions have been added to the new connec-

tion, i.e. remains *PRECISE*. Other objects are then treated as *IMPRECISE* as soon as the first invalidation arrives on the new connection. As a result, no special actions are needed to resynchronize a client's state during recovery.

6.2.2 Leases

Leases [33, 98] are another important mechanism for maintaining strong consistency for client-server systems. An object lease represents permission to access an object until specified time [33]. It introduces a tradeoff between the availability of write and the availability of read. With the *callbacks* mechanisms, when a client with a valid callback is disconnected from the server, the server can not make progress for updates. This problem is addressed by using leases which specify a length of time during which servers notify clients of modifications to cached objects. After the lease expires, a client must renew it before it access any cached object. Therefore, the server could still modify the object even if any client with callbacks is disconnected.

URA supports Yin et al.'s volume lease [101] to expire callbacks from unreachable clients. The volume leases protocol is a variation of leases protocol that generalize leases to work efficiently with WAN workloads. A volume lease is a lease on a group of objects (volume). Under the volume leases protocol, a client may access a cached object if it holds valid leases on both the object and the object's volume, and a server can modify data as soon as either lease expires.

Implementation. Implementing volume leases mainly involves how to expire/renew the corresponding callbacks. To implement leases, URA needs to extend the basic callback implementation in three ways. First, clients maintain incoming subscriptions to a specific volume lease object from the server. Second, in order to renew the lease automatically if connected, the server keeps the client's view up-to-date by sending periodic heartbeats via a volume lease object, i.e., periodically updates the volume lease object. As a result of the subscription, the clients will receive the precise invalidations of the volume lease object

and all other precise invalidations of any subscribed object. Note that the *realTime* accept stamp in each precise invalidation updates the clients' notion of server's real time so that the clients can check for expired leases.

Finally, to realize the lease expiration effect, it need to do two things for handling reads and writes.

- We need an additional condition to the *ReadNowBlock* predicate besides the *isValid* and *isComplete* condition variables if the client's view of the server's state is too stale. It turns out that the TACT's TE parameter is the right interface to implement this block. If we set TE to be the lease period, the read will be blocked if the client is disconnected from the server for more than TE or the object has not renewed the lease from the server.

- The server must maintain the lease status of all clients who have callbacks so that a write can proceed once all callbacks either break or the corresponding leases expire. We can do this by splitting the callback list into two lists *callback_1* and *callback_2*. When a callback for a client C is established, the server adds C to *callback_1*. When a callback for a client C is broken, the server removes C from either list where it exists. Suppose the lease period is L seconds, in order to expire the lease, the server periodically empties *callback_2*, removes the corresponding subscriptions, and then moves nodes in *callback_1* to *callback_2* in every $L/2$ seconds. The server can return a write whenever both of the list are empty.

Note that by transporting heartbeats via a URA object, we ensure that a client observes a heartbeat only after it has observed all causally preceding events, which greatly simplifies reasoning about consistency.

6.3 Quorums

Another important class of replication protocols is the quorum-based (a.k.a. voting) approaches [90, 30]. In the quorum-based protocols, any write/read operation only requires a subset of nodes (write/read quorum) to be available to process it. The intersection of read/write quorums invariant guarantees regular semantics [53] in the presence of network partitions and node failures. There are a number of variations of quorum-based protocols that address different workloads and failure models. For example, Grid quorums [15], tree quorums [2] etc. reduce the quorum size by imposing a logical structure on the set of replicas. Byzantine quorum systems [58] use quorum systems to tolerate arbitrary failures.

In addition, the quorum method is also a powerful framework that can address a broad range of other replication systems. For example, server replication systems such as Bayou are fundamentally Read-One-Write-All-Asynchronous (ROWAA) systems. A ROWAA system can be viewed as a special quorum system with read quorum of 1 and write quorum of all. In this "quorum" system, instead of waiting for voting from all nodes, the write operation returns immediately after one node applies it and propagates it to the other nodes asynchronously.

Client-server replication systems are also related to quorum systems. In particular, dual-quorum [28] generalizes client-server systems by separating the read/write quorums into two independent quorum systems. The output quorum system caches/prefetches data from the input quorum system. The input quorum system acts as servers and grants volume leases [101] to the output quorum system, and the output quorum system renews the leases from the input quorum system.

6.3.1 Implementation on URA

To facilitate implementing a range of quorum systems on URA, we provide a *QuorumLocalInterface* wrapper over the basic local operation interface. The *QuorumLocalInterface* wrapper provides primitives similar to Q-RPC [59] so that the controller layer can freely

implement different styles of quorum systems ranging from those that number a simple majority, to explicit enumeration of the membership of each possible quorum. The complete description of policy implementation of those quorum systems is beyond the scope of this dissertation. In the following paragraphs, we discuss how to extend URA's local interface to implement the building blocks, i.e., the *QuorumLocalInterface* wrapper that includes two basic operations: *quorum read* and *quorum write*.

Quorum read. In order to provide *regular semantics*, the quorum read must ensure that the read value is the newest value among all local values stored at all of the nodes in a read quorum. Therefore a quorum read issued at a node α with a read quorum of Q_{read} includes two steps.

- First, the wrapper blocks until α has an active incoming invalidation stream from each node in Q_{read}. This block makes sure that α's state reflects all updates processed at any node in Q_{read} before this read is issued.
- Then the wrapper issues a URA local read at α with the *ReadNowBlock* predicate set to *isValid* and *isComplete* and finally returns the read value to the quorum read client. These two blocks ensure that the read returns the newest value among all values stored in all of the nodes in Q_{read}.

Note that because of separation of invalidation and body, in the second step, if the read is blocked because of $INVALID$, the controller can specify any policy to bring the matching or newer body to unblock the read. The sender does not have to be one member of the Q_{read}. Therefore the wrapper can offer a range of policies to optimize the quorum read latency and reduce network bandwidth overhead.

Quorum write. In order to provide *regular semantics*, the quorum write W must ensure that any following quroum read after W returns a value at least as new as W. Therefore a quorum write issued at a node α with a write quorum of Q_{write} includes three steps.

- First, the wrapper blocks until α has an incoming invalidation subscription from each node in a read quorum. These subscriptions bring $alpha.currentVV$ up to include each node's $currentVV$ in a read quorum. Since any read quorum intersects any write quorum, this block and the lamport clock ensures that the current pending quorum write is ordered after any previous completed writes.

- Then the wrapper issues a local write W through the URA local write interface.

- Finally, the wrapper sends an *synchronization request* to each node in Q_{write} and returns the write after it receives an acknowledgement from each node in Q_{write}. This block ensures that at least all nodes in Q_{write} is aware of the new update so that any following read will return a value as new as W.

The *QuorumLocalInterface* wrapper enforces *regular semantics* if the Q_{read} and Q_{write} has intersections. Note that in the third step, the controller can specify any policy to distribute the invalidation of the quorum write to Q_{write} and specify any reliability policy to distribute the body to any node while the wrapper can still enforce the *regular semantics*.

We can implement different controllers for different quorum systems using the basic *QuorumLocalInterface* wrapper.

6.4 Other Features

There are additional features that are useful for implementing replication systems. Here we briefly list some of them.

Atomic writes. Besides the traditional strong consistency semantics described above, we also provide *atomic multi-object writes*. This write interface allows an update targeting to multiple objects and assigns the updates to multiple objects with one logical stamp and thus propagates them atomically as one invalidate in the system, i.e., either all the objects are updated/invalidated or non of the objects are updated/invalidated. Note that the only

difference between the multi-object invalidation and the normal precise invalidation is the *targetSet*. We can treat a multi-object invalidation as a normal precise invalidation to be inserted in the log or be accumulated in an imprecise invalidation or to be applied to local storage state as described in chapter 5.

This interface is very useful in implementing file system interface on top of our raw object store as described in the next paragraph.

NFS interface. To demonstrate the feasibility of supporting useful applications built on URA, we implement an NFS interface wrapper over URA object interface. The NFS interface wrapper serves as a bi-directional translation between the NFS client calls and the URA local interface calls. The wrapper implements a user-level loop-back NFS server that listens on the NFS port, parses RPC requests, translates calls to URA local interface calls, and finally puts up corresponding responses to the requests.

The major complexity comes from translating the file and directory concepts into the raw object access interface. We implement each file as two URA objects, one for the content and the other for the metadata information like file size etc. We also implement a directory as a normal file except that the corresponding metadata object is marked as a directory type, which enables the read to parse the object correctly.

Although the NFS specification provides only loose requirements on the consistency of the files that are exposed to users, NFS requires that each operation be *atomic*, i.e, the system should never be visible in a state where a particular operation has only partly completed. Because each file or directory is composed of two URA objects, any translated URA object update operations must atomically change both objects. We simply use the atomic multi-object write described above to implement this semantics.

Detailed description of the implementation of our NFS interface can be found in [67].

Chapter 7

Case-study Systems

To examine the claim that URA is a *better way* to build replication systems by providing a common substrate, this chapter evaluates URA with a series of case-study systems that span a significant portion of the design space.

Figure 7.1 summarizes the features covered by 6 case-study systems and 4 additional variations we implemented using URA. 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.

In the rest of this chapter, we first describe the evaluation criteria to compare our prototype systems to systems from the literature. Then Section 7.2 describes how to implement *client-server systems* like Coda [48] and TRIP [66] and then compares these URA

	U-Bayou[72] +Small Device	U-Chain Repl [91]	U-Coda [48] +Coop Cache	U-Pangaea [77]	U-Tier Store [23]+CC	U-TRIP [66]+Hier
Consistency	Causal	Lin.	Open/close	Coherence	Coherence → Causal	Seq.
Topology	Ad-Hoc	Chains	Client/ Server	Ad-Hoc	Tree	Client/Server → Tree
Partial Replication	√		√	√	√	
Prefetching/ Replication	√	√	√	√	√	√
Cooperative Caching			√		√	
Disconnected operation	√		√		√	
Callbacks			√	√		√
Leases			√			
All reads served locally	√	√			√	
Crash recovery	√	√	√	√	√	√
Object store interface*	√	√	√	√	√	√
File system interface*	√	√	√	√	√	√

Figure 7.1: 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.

implementations with existing systems, Section 7.3 details the implementation of *server-replication systems* like Bayou [72] and Chain Replication [91], and finally Section 7.4 explain how to implement the *object replication systems* like Pangaea [77] and TierStore [23].

7.1 Evaluation Criteria

The reader should be a bit concerned at this point. We claim that URA 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 URA 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 URA.

7.1.1 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 small constant factor of system B's. "Small constant factor" sounds weak but appropriate because we believe that although there exists no meaningful way to check what "small" means in all possible workloads, we can and will show empirically that our system does perform well in many real-life scenarios.

E2. *Equivalent consistency.* System A provides consistency, coherence, 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 for any workload.

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) [31] and PC (Performance

v. Consistency) [54] 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 [66]. Similarly, one can improve the availability a system offers for a given level of consistency by using more network bandwidth to synchronize more often [103], or one can reduce the resources consumed by replication by delaying propagation of updates and weakening consistency [7] 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.

More broadly, systems may target specific workloads, and their behavior for other workloads may not be of interest.

Definition. System A is *W-equivalent* (workload equivalent) to system B if properties E1 through E3 hold at specified times for a specified subset of workloads. Obviously, one must judge whether a W-equivalent system is interesting based on the specific subset of workloads included.

All of the six systems we build and discuss in this paper are designed to be Q-equivalent with their original systems.

7.2 Client-server Systems

As described in Chapter 2, *Client-server* systems like Sprite [68] and Coda [48] and *hierarchical* caching systems like hierarchical AFS [65] permit nodes to cache arbitrary subsets of data (PR) and ensure certain consistency semantics. However, these protocols fundamentally require communication to flow between a child and its parent, which hurts performance and availability under certain environments.

In this section, we discuss how to implement such systems and how URA enables better performance by removing the topology restriction of those systems by detailing two case-study systems: U-Coda [48] and U-Chain Replication [66].

7.2.1 U-Coda

Coda [48] is a typical client-server system. All data is located at the server whereas the client caches some files locally. The server maintains a list of clients who cache valid copies of files and notifies the clients once it learns a new update (callbacks). 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.

We discuss in detail U-Coda, a system inspired by the version of Coda. U-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 [82], trickle reintegration [63], and variable granularity cache coherence [64]. We see no fundamental barriers to adding them in U-Coda. We also illustrate the ease with which co-operative caching can be added to U-Coda.

Implementing consistency policies. 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.

We employ an open/close wrapper library that buffers writes in the library until close, at which point it will make the writes to URA's storage. A file "open" is implemented as a read of an object. The causal consistency read interface is used, i.e. all reads will block until the local object is $VALID$ and $PRECISE$ before the object is accessed. All writes to file are buffered until the file is closed at which point, the object is written. If the client is connected to the server, an object write will be blocked until the server gathers invalidation acknowledgements from all clients that have callbacks and reports "done." Otherwise, the write is simply stored in log and will be transferred to the server when connected later. This standard open/close library is usable by different systems.

Implementing topology policies. U-coda topology policies can be divided into 7 main groups: configuration, connectivity, demand read, write propagation, recovery, hoarding, and safety enforcement. Note the safety enforcement are part of the consistency library, we include them here for completeness.

Note that because the configuration and connectivity are similar for most of case-study systems, we discuss these implementation here and skip them in the other case-study system description.

Configuration and connectivity. A configuration file stores the server's identity and another configuration file provides the hoard list. Each client C periodically pings the server S to check whether the server S is currently reachable. We base on an implementation of Narada [55] to track connectivity information and generate *newLiveNeighbor* and *declareDeadNode* events, which invoke different subscription actions respectively as specified by the topology policies.

Handling read blocks. When a read of object o is blocked at a connected client C. C subscribes from S for o's invalidations using a checkpoint catchup for efficiency, and fetches the body from S. Eventually, o is no longer inconsistent, and C unblocks the read. The invalidation subscription ensures that if another node updates o, it will become invalid.

Write propagation and callbacks. To propagate client writes to the server, an invalidation and a body subscription from the client to the server are triggered when the client connects to the server.

The invalidation subscriptions created by clients when they read objects from the server ensure that our underlying mechanisms transmit invalidations. To avoid sending repeated invalidations to a client, U-coda removes an object from the invalidation subscription when a client receives an invalidation.

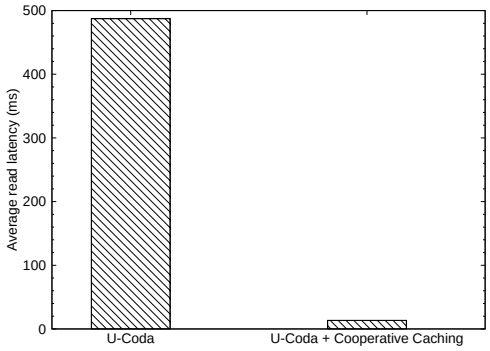


Figure 7.2: Average read latency of U-Coda and U-Coda with co-operative caching.

Recovery. When a client reconnects to a server, it establish an invalidation subscription for an empty subscription set from the server. This action makes all the locally cached objects $IMPRECISE$ if server has any update during the disconnected period. Therefore, it breaks all callbacks and hence enforce the consistency.

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

Safety enforcement. The bulk of the safety enforcement is done at the server. The client sends an acknowledgement to the server whenever it receives an invalidation. Once the server has collected the required acknowledgements from other clients, it sends "done" acknowledgement to the client. The arrival of this acknowledgement unblocks the blocked write.

Adding cooperative caching We add co-operative caching to U-Coda so that disconnected clients can fetch valid files from their peers. This allows a disconnected client to

access files it previously couldn't.

To 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 a client reads the configuration file, it generates a list of peers and keeps track of connectivity to peers. A client triggers a fetch attempt from reachable peers if the server is not reachable.

Cooperative caching allows a clique of connected devices to share data without relying on the server. Figure 7.2 demonstrates the significant improvement by adding cooperative caching to U-Coda. 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.

Equivalence. U-Coda is Q-equivalent to Coda if the number of writes at each node between disconnections exceeds the number of nodes and 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. 5.10). 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 U-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, on 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.

7.2.2 U-TRIP

TRIP [66] seeks to provide transparent replication for web edge servers of dynamic content. All nodes enforce sequential consistency and a limit on staleness.

With the detailed description of Coda, here we give a brief overview of the U-TRIP implementation.

Implementing consistency and topology policy. U-TRIP uses the standard causal consistency library with the maxStaleness predicate set. Note that a causal consistency library also enforces sequential consistency if, as in TRIP, there is a single writer. The topology policies are simple: clients subscribe to receive all invalidations and bodies from the server.

Extending U-TRIP. What is perhaps most interesting about this example is the extent to which URA facilitates evolution. For example, the TRIP implementation assumes a single server and a star topology. By implementing on URA, 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 [55], a few additional rules to subscribe/unsubscribe produce a dynamic-tree version of TRIP that still enforces sequential consistency.

7.3 Server Replication Systems

As described in Chapter 2, server replication systems like Dictionary [96], Bayou [72] provide log-based peer-to-peer update exchange that allows any node to send updates to any other node (TI) and that consistently orders writes across all objects. However, these systems are unable to exploit workload locality to efficiently use networks and storage, and they may be unsuitable for devices with limited resources because their protocols fundamentally requires all nodes store all data from and all nodes receive all updates.

In this section, we discuss how to implement such systems and how URA enables to add new features to those systems which would be difficult in the original implementations by detailing two case-study systems: U-Bayou [72] and U-Chain Replication [91].

7.3.1 U-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 URA modeled on the version of Bayou described by Petersen et. al. [72]. 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 consistency semantics. Implementing the consistency policy of Bayou on URA is simple. Since Bayou provides causal consistency and eventual consistency, as we described in Chapter 6, U-Bayou simply uses the default causal consistency interface provided by URA, i.e, sets the *ReadNowBlock* predicate to *isValid* and *isComplete*. To provide 100% availability as the original Bayou does, U-Bayou delays applying an invalidation to a node until the node has received the corresponding body, i.e., sets the *ApplyUpdateBlock* predicate to *isValid*.

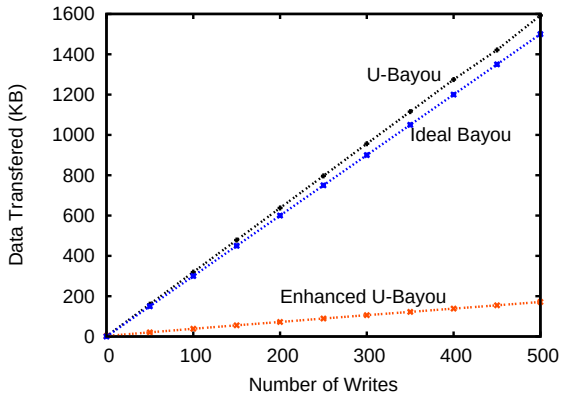


Figure 7.3: Anti-Entropy bandwidth on U-Bayou.

Implementing update distribution policies. The topology policy of Bayou is mainly about carrying out anti-entropy sessions. Anti-entropy sessions can be easily implemented by establishing invalidation and body subscriptions between nodes for "/*"(means all objects) and removing the subscriptions once all updates have been transferred. Note, as in Bayou, by default the catchup sending invalidations instead of checkpoint. If the log at the sender is truncated to a point after subscription's start time, the invalidation subscription will automatically send a checkpoint.

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 URA, 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.

As Figure 7.3 indicates, the overhead for anti-entropy in U-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 U-Bayou greatly reduces the bandwidth required for anti-entropy.

Equivalence. U-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 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.

7.3.2 U-Chain Replication

Chain Replication [91] is another server replication protocol in which the nodes are arranged as a chain to provide high availability and linearizability. To implement 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.

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

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

Implementing consistency and topology policies Although we could use our standard linearizability consistency wrapper as described in Chapter 6, to gain performance and availability comparable to the original Chain Replication system we implement a customized consistency wrapper that exploits the chain topology and simply blocks a write

until it receives an acknowledgement from the tail.

U-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.

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. URA 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.

7.4 Object Replication Systems

As described in Chapter 2, *Object replication* systems such as Ficus [36], Pangaea [77], and WinFS [60] allow nodes to choose arbitrary subsets of data to store (PR) and arbitrary peers with whom to communicate (TI). But, these protocols enforce no ordering constraints on updates across multiple objects, so they can provide coherence but not consistency guarantees and therefore their applications are restricted to those with less stringent semantics.

In this section, we discuss how to implement such systems by detailing two case-study systems: U-TierStore [23] and U-Pangaea [77].

7.4.1 U-TierStore

TierStore [23] 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 consistency and topology policies U-TierStore uses the standard best-effort coherence interface by setting the *ReadNowBlock* predicate to *isValid*. For 100%

availability, it sets the *ApplyUpdateBlock* predicate to *isValid* to delay applying an invalidation to a node until the node has received the corresponding body.

To implement TierStore's tree-based topology, we make use 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, that it is interested in. On initialization, as these files are read, the corresponding subscriptions between parents and children are established as follows: (1) from a parent to child, invalidation and body subscriptions for each of the publications a child is interested in are established and (2) from a child to parent, invalidation and body subscriptions for "/*" are established.

Extending U-TierStore. Just like U-Coda, cooperative caching is easily added to U-TierStore by adding a few lines of code. 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.

7.4.2 U-Pangaea

Pangaea [77] 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.

U-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 consistency and topology policies. Similar to U-TierStore, U-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 topology policy for Pangaea is pretty complex. 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. We do not go into details due to space constraints, but provide overview of the different aspects.

U-Pangaea considers harbingers as invalidations, and hence each edge of a Pangaea graph is an invalidation subscription. U-Pangaea's liveness policy sets up and maintains the m-connected graph for each object among the nodes. In particular, (1) it maintains m-connectedness for each replica; (2) it makes sure that a replica is connected to at least one gold node; and (3) it makes sure that there are at least g gold nodes which are connected in a clique.

In case of a read miss, U-Pangaea will try to identify if the object exists in the network or if it is a new object. If the object exists, it creates a new local replica by finding the closest replica and adding an invalidation subscription from it for the object with catchup policy set to *checkpoint* so that it can quickly retrieve the object. Then it integrates itself into the replica graph by setting up subscriptions with $m - 1$ other neighbors. In case of a new object, it will create the object, locate $g - 1$ other nodes to be gold nodes, create replicas on them and establish subscriptions among each other.

In case of an update, invalidations are automatically flooded along the subscriptions. When a node receives an invalidation, it will demand read for the body from the sender it receives the first invalidation. If it does not receive the body within a certain timeout, it will find another neighbor to retrieve the body from.

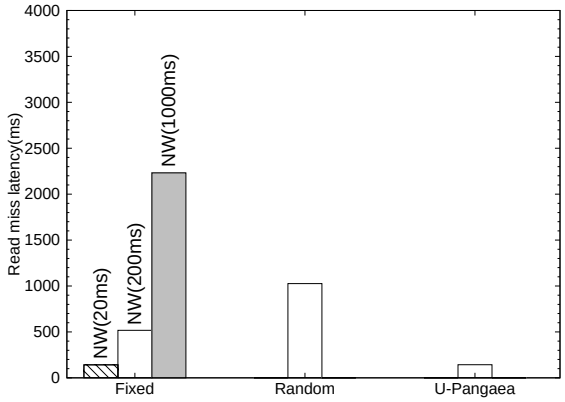


Figure 7.4: Read miss latency for U-Pangaea and alternatives.

Figure 7.4 illustrates one aspect of U-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 [77], 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.

Equivalence. URA propagates sufficient information for any node to enforce causal consistency using local information. U-Pangaea does not use this information, so it retains the high availability, partition-resilience, and performance [54, 31] available to weak-consistency systems. Furthermore, the overhead of propagating this information is modest for several reasons. First, as Figure 5.11 indicates, once a connection between a pair of nodes is established, URA sends at most two invalidation messages for every one sent by a coherence-only algorithm—at worst an URA 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.

Chapter 8

Related work

Decades of research on data replication yield a number of replication techniques and frameworks for different environments or workloads in different perspectives. This chapter surveys the related work.

8.1 Replication Mechanisms

State of the art mechanisms allow a designer to retain full flexibility along at most two of the three dimensions of replication, consistency, or topology policy. Chapter 2 examines existing PR-AC [42, 48, 68, 4, 12, 20], AC-TI [32, 46, 50, 72, 96, 103], and PR-TI [36, 77] approaches. These systems can be seen as special case "projections" of the more general URA mechanisms. Ideas relating to URA's mechanisms can be seen in these systems. For example, the separation of invalidations from bodies is standard in client-server systems [42][68], and imprecise invalidations are closely related to messages sent by client-server systems during callback-state recovery [6][99]. Several systems have noted the value of separating data and metadata paths [4][77][89].

Some recent work extends server replication systems towards supporting partial replication. Holliday et al.'s protocol allows nodes to store subsets of data but still requires

all nodes to receive updates for all objects [41]. Published descriptions of Shapiro et al.'s consistency constraint framework focus on full replication, but the authors have sketched an approach for generalizing the algorithms to support partial replication [83].

8.2 Replication Framework

A number of other efforts have defined frameworks for constructing replication systems for different environments. Like URA, the Deceit file system [84] provides a flexible substrate that subsumes a range of replication systems. Deceit, however, focuses on replication across a handful of well-connected servers, and it therefore makes very different design decisions than PRACTI. For example, each Deceit server maintains a list of all files and of all nodes replicating each file, and all nodes replicating a file receive all bodies for all writes to the file. Zhang et. al. [104] define an object storage system with flexible consistency and replication policies in a cluster environment. As opposed to these efforts for cluster file systems, URA 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 [38] seek to provide a way to add features and compose file systems, but it focuses on adding features to local file systems.

WinFS [70] employs a peer-to-peer state-based exchange algorithm to synchronize nodes but exchanges stored state rather than logs. Although it provides *topology independence*, *partial replication*, *eventual consistency*, and efficient conflict detection using *vector sets*, it can not provide causal consistency when synchronization is interrupted.

URA incorporates the order error and staleness abstractions of TACT tunable consistency [103]; we do not currently support numeric error. Like URA, Swarm [87] 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, sequential, or linearizability. IceCube [47] and actions/constraints [83] provide frameworks for specifying

general consistency constraints and scheduling reconciliation to minimize conflicts. Fluid replication [16] provides a menu of consistency policies, but it is restricted to hierarchical caching.

8.3 Conflict Detection

As described in Section 5.4, there are three main approaches for conflict detection: *previous stamps* [34, 9], *hash histories* [45], and *version vectors* [48, 74, 95]. Both *previous stamps* and *hash histories* imposes per-update storage overhead and might have false negatives under certain scenarios. *Version vectors* can accurately detect conflicts but it imposes one vector per object overhead which is prohibitive when the number of replicas is large. *predecessor vectors with exceptions (PVE)* [60] and *vector sets* [57] are variations of the *version vectors* approach employed by WinFS [70] to reduce the total number of version vectors each node maintain.

Although some existing systems provide conflict resolution mechanisms that take into account the semantics of the data being resolved - for e.g. Bayou [72], TACT [102], and Coda [48] - URA provides a simpler interface that allows users to view a *conflict log* and take appropriate action (e.g. perform a compensating write). Although we do not provide additional facilities to help conflict resolution such as log roll back [72] or conflict-handling procedures [88], we speculate that such features can be added to our system without difficulty.

Chapter 9

Conclusions

The central thesis of this dissertation is that there is a set of flexible common replication mechanisms that capture the right abstractions for replication and therefore can serve as a replication "micro-kernel" for building and deploying replication systems for different environments and workloads by simply defining the right policies on top of the mechanism layer.

To answer this question, this dissertation presented a *universal data replication architecture* (URA) that cleanly separates mechanism from policy to simplify the development of new replication systems, subsumes most of existing replication protocols, and enables better trade-offs for some environments than are currently available.

As we have shown in the dissertation, we presented the architecture in two steps. First, we presented the PRACTI replication mechanisms that for the first time provide three vital features simultaneously: **P**artial **R**eplication, **A**ny **C**onsistency, and **T**opology **I**ndependence. Through prototype experiments, our conclusion is that by providing all three PRACTI properties, PRACTI replication enables better trade-offs than existing mechanisms that support at most two of the three properties: it dominates existing families of architectures by providing order of magnitude performance improvements for some key environments and workloads and matching their performance for most other environments

and workloads.

Second, we described the UR-Repl protocol that extends the PRACTI mechanisms to support efficient callbacks, incremental checkpoint exchange and efficient conflict detection. Our micro-benchmark experiments demonstrated that the UR-Repl protocol is an order of magnitude more efficient than the PRACTI replication protocol for supporting callbacks, and its conflict detection algorithm is more efficient than existing approaches in most cases.

To demonstrate the flexibility of URA, we described how to map most of existing techniques and consistency semantics over URA and presented the results of a set of case-study systems that cover a significant portion of the design space. Overall, our experience suggests that (1) URA is *flexible* in that we are able to implement a broad range of systems; (2) URA is *efficient* in that we are able to build systems that are comparable to hand-built systems from literature with respect to the central properties of a replication architecture; and (3) URA *facilitates innovation* by exposing new design space and making it much easier to add new features in existing systems.

In summary, this dissertation makes the following key contributions:

- It defines the PRACTI paradigm and provides a taxonomy for replication systems that explains why existing replication architectures fall short of ideal.
- It describes the first replication protocol architecture to simultaneously provide all three PRACTI properties.
- It defines common abstractions of data replication systems that cleanly separate mechanism from policy and thereby simplify the understanding and construction of replication systems.
- It demonstrates that URA replication offers decisive practical advantages compared to existing approaches.

- It demonstrates the usefulness of URA by building several key case study applications and mapping existing techniques on URA.
- It proposes novel incremental checkpoint exchange, flexible commit primitive, and efficient conflict detection algorithms.

Limitations and future work. The current URA prototype focuses on getting the architecture right and has the following limitations and potential future work:

- Scalability. The version vector limits the architecture scaling to a large number of nodes. A possible future work is to add/remove ids from an interest set's version vector on the fly (a la Bayou's adding and removing servers on the fly). Essentially, one could insert a "remove X from VV for interest set I" into the log that instructs anyone receiving the write that it is OK to stop tracking node X for interest set I; this "write" would get serialized in the log according to the normal rules and if X later does a write to I, it gets added back into the version vector starting at the time that it does the write.

  If this dynamic version vector algorithm works, then the version vector length for an interest set is proportional to the number of "active writers" in that interest set rather than proportional to the number of nodes in the system.
- Performance. As indicated in Section 4.6, we have a lot of room for performance-tuning as our prototype is implemented in Java and the local storage implementation is based on the BerkeleyDB JE [86]. For example, we have not spent time parameters of the BerkeleyDB JE for our workload.
- Local storage management. Currently, we leave all of the local storage management work to BerkeleyDB. In the future, it will be interesting to use RAID and play with disk layout and data replacement etc..
- Policies for splitting/joining interest sets. As indicated in Chapter 4, the size of an interest set affects the URA's storage and performance for tracking per-IS consistency

state. Although currently, URA makes static configuration of the structure of interest sets, URA allows system designers to dynamically split an interest set or join different interest sets so as to make different tradeoffs for different workloads. In the future, we may be able to build applications that take advantage of this mechanism.

Bibliography

[1] Tivoli data exchange data sheet. `http://www.tivoli.com/products/documents/datasheets/data_exchange_ds.pd%f`, 2002.

[2] D. Agrawal and A. Abbadi. The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. 1990.

[3] D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *The Computer Journal*, 34(6):534–541, 1991.

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

[5] S. Annapureddy, M. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the Second USENIX Symposium on Networked Systems Design and Implementation*, May 2005.

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

[7] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 10–22, September 1992.

[8] M. Beck and B. Dempsey. I2-DSI overview. In *4th Intl. Web Caching Workshop*, March 1999.

[9] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proceedings of the Third USENIX Symposium on Networked Systems Design and Implementation*, May 2006.

[10] Bent and Voelker. Whole page performance. In *Workshop on Web Caching and Content Distribution*, September 2002.

[11] T. Berners-Lee, R. Fielding, and H. Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0. Internet Draft draft-ietf-http-v10-spec-00, Internet Engineering Task Force, March 1995.

[12] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

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

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

[15] S. Cheung, M. Ahamad, and M. Ammar. The grid protocol: a high performance scheme for maintaining replicated data. pages 438–445, 1990.

[16] L. Cox and B. Noble. Fast reconciliations in fluid replication. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001.

[17] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative

storage with CFS. In *Proceedings of the Eightteenth ACM Symposium on Operating Systems Principles*, October 2001.

[18] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 2003.

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

[20] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[21] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles*, 2007.

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

[23] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed storage system for challenged networks. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, February 2008.

[24] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[25] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fif-*

teenth ACMSymposium on Operating Systems Principles, pages 201–212, December 1995.

[26] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Workshop on Internet Server Performance*, June 1998.

[27] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *International World Wide Web Conference*, May 2003.

[28] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar. Dual-quorum replication for edge services. In *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference*, November 2005.

[29] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[30] D. Gifford. Weighted voting for replicated data. In *7th ACM Symposium on Operating Systems Principles*, pages 150–162, April 1979.

[31] 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.

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

[33] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

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

[35] J. Griffioen and R. Appleton. Reducing File System Latency Using A Predictive Approach. In *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

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

[37] J. Gwertzman and M. Seltzer. The case for geographical pushcaching. In *HOTOS95*, pages 51–55, May 1995.

[38] J. Heidemann and G. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

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

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

[41] J. Holliday, D. Agrawal, and A. El Abbadi. Partial database replication using epidemic communication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, July 2002.

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

[43] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *10th International Conference on Distributed Computing Systems*, pages 302–311, 1990.

[44] D. S. Parker (Jr.), G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, S. Kiser, D. Edwards, and C. Kline. Detection of Mutual Inconsistency

in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[45] B. Kang, R. Wilensky, and J. Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[46] P. Keleher. Decentralized replicated-object protocols. In *Proceedings of the 18th Symposium on the Principles of Distributed Computing*, pages 143–151, 1999.

[47] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube aproach to the reconciliation of divergent replicas. In *Proceedings of the 20th Symposium on the Principles of Distributed Computing*, 2001.

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

[49] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007.

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

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

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

[53] L. Lamport. On interprocess communications. *Distributed Computing*, pages 77–101, 1986.

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

[55] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[56] P. Mahajan, S. Lee, J. Zheng, and M. Dahlin. SADDR: Secure autonomous distributed data replication, May 2008. `http://www.cs.utexas.edu/users/zjiandan/papers/saddr08.pdf`.

[57] D. Malkhi, L. Novik, and C. Purcell. P2P Replica Synchronization with Vector Sets. *ACM SIGOPS Operating Systems Review*, 41(2):68–74, 2007.

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

[59] D. Malkhi and M. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, pages 187–202, March 2000.

[60] D. Malkhi and D. Terry. Concise version vectors in WinFS. In *20th Symposium on Distributed Computing (DISC)*, 2005.

[61] P. Maniatis, M. Roussopoulos, TJ Giuli, D. Rosenthal, M. Baker, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[62] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, December 1999.

[63] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 143–155, December 1995.

[64] L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[65] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX Conference*, pages 305–313, January 1992.

[66] A. Nayate, M. Dahlin, and A. Iyengar. Transparent information dissemination. In *Proceedings of the ACM/IFIP/USENIX 5th International Middleware Conference*, October 2004.

[67] Amol Nayate. *Transparent Replication*. UT Austin, Austin, Texas, 2006.

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

[69] E. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, December 2004.

[70] L. Novik, I. Hudis, D. Terry, S. Anand, V. Jhaveri, A. Shah, and Y. Wu. Peer-to-peer replication in winfs. Technical Report MSR-TR-2006-78, Microsoft Research, June 2006.

[71] D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, 2006.

[72] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[73] A. Rajasekar, M. Wan, and R. Moore. MySRB and SRB - components of a data grid. In *The 11th International Symposium on High Performance Distributed Computing*, July 2002.

[74] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the Summer 1994 USENIX Conference*, 1994.

[75] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, March 2003.

[76] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eightteenth ACM Symposium on Operating Systems Principles*, 2001.

[77] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[78] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

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

[80] S. Sarin and N. A. Lynch. Discarding Obsolete Information in a Replicated Database

System. *IEEE Transactions on Software Engineering*, SE-13(1):39–47, January 1987.

[81] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 35–46, October 1996.

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

[83] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proceedings of the 8th International Conference on the Principles of Distributed Systems*, December 2004.

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

[85] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in Beehive. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[86] Sleepycat Software. *Getting Started with BerkeleyDB for Java*, September 2004.

[87] S. Susarla and J. Carter. Flexible consistency for wide area peer replication. In *Proceedings of the 25th International Conference on Distributed Computing Systems*, June 2005.

[88] 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 *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, December 1995.

[89] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems*, May 1999.

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

[91] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, December 2004.

[92] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A mechanism for background transfers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[93] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a wan. In *Proceedings of the 20th Symposium on the Principles of Distributed Computing*, August 2001.

[94] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. *Elsevier Computer Communications*, 25(4):367–375, March 2002.

[95] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–69, October 1983.

[96] G. Wuu and A. Berstein. Efficient solutions to the replicated log and dictionary problem. In *Proceedings of the Third Symposium on the Principles of Distributed Computing*, pages 233–242, 1984.

[97] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *ACM SIGCOMM 2004 Conference*, August 2004.

[98] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering server-driven consistency for large scale dynamic web services. In *Proceedings of the 2001 International World Wide Web Conference*, May 2001.

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

[100] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[101] 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*, February 1999.

[102] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proceedings of the Eightteenth ACM Symposium on Operating Systems Principles*, 2001.

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

[104] 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.

[105] 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.

Vita

Jiandan Zheng was born in Yueqing, Zhejiang China in 1977. After completing her high school at Yueqing High School in 1994, she enrolled in Beijing University. She received the degree of Bachelor of Science from Beijing University in July 1998 and the degree of Master of Science from Institute of Software, Chinese Academy of Sciences in 2001. She received the degree of Master of Art in Computer Science from the University of Texas at Austin in May 2005.

Permanent Address: Xiangyang Town Liunan Village
Yueqing, Zhejiang 325619
P.R. China

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.