The Dissertation Committee for Nalini Moti Belaramani
certifies that this is the approved version of the following dissertation:

Policy Architecture for Distributed Storage Systems

Committee:

Mike Dahlin, Supervisor

Lorenzo Alvisi

Mohamed Gouda

Lili Qiu

Petros Maniatis

Policy Architecture for Distributed Storage Systems

by

Nalini Moti Belaramani, B.Eng.; M.Phil.

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 2009

To Mom, Dad, and Kiran.

asato ma satgamaya,

tamaso ma jyotirgamaya,

mrityor ma amritamgamaya.

From untruth to truth,

From darkness to light,

From death to immortality.

Acknowledgments

So many people who have helped make my dream a reality. This dissertation would not have been possible were not for following people.

My adviser, Mike Dahlin. I have the utmost respect and gratitude for my adviser. Mike is probably the best adviser a Ph.D. student can have. What I admire most about Mike is his quest for perfection in everything he does—be it coding, writing, managing time, or managing students. I am grateful for his gentle prodding that has helped me push my limits and achieve what I have. Thank you Mike for imparting your knowledge to me.

My graduate committee members,who were great to work with. Their questions and comments have helped me to understand the value and the pitfalls of my work better. And of course, Petros Maniatis for the technical discussions that have helped shape my work and for always encouraging me to brag about my work.

Jiandan Zheng, Robert Soulé, and Robert Grimm with whom collaboration was easy and rewarding.

The LASR lab, which is full of a cool bunch of people who sincerely want every one to succeed. I thank everyone for sitting through my numerous practice talks, reading my drafts, and just lending me ears when I wanted to talk things out. The support staff, which is is phenomenal. If it weren't for Gloria Ramirez, Sara Strandtman, and Lydia Griffith, I would be lost in the bureaucratic maze of the UT-system. The technical support staff is also wonderful to work with. I would have been handicapped without their help in setting up and managing machines for experiments.

I have been lucky to meet a great set of friends who really care about me and have, in one way or the other, made sure I do not lock myself in a hole. They include: Edmund Wong for introducing me to gastronomic delights that make stress disappear into thin air, for feeding me, for sharing my frustrations, for answering my questions, and most importantly, for brightening up my day with his energy. Amol Nayate, Joseph Modayil, Prem Melville, Shilpa Chhada, and Misha Bilenko for taking me under their wings, for helping me adapt to the new culture I found myself in, and for being my family away from home. Richa Gargh and Yannis Korkolis for always being there when I need to fill my people quota, when I want to celebrate, or when I just want to bum around. The Brown Trash family for making me feel that I belong, for hosting awesome pot-lucks, and for being such great dance party crashers! My Hong Kong friends for being my personal cheerleaders, for not letting distance come in the way of friendships, and for always believing that I can achieve great things.

In the final stretch, when all I did was write my dissertation, I thank the following persons for helping me maintain my sanity: Alexandre Dumas and Stephenie Meyer, who gave me my daily dose of drama through their works, the Count of Monte Cristo and the Twilight series. DJ Tiesto, whose weekly podcast helped me drown out the rest of the world in techno beats and just focus on writing.

Lastly, I thank the four most important persons in my life:

Jason Brandt, for being there for me, sharing my happiness and frustrations, making sure I am well-fed and well-slept, for bearing with my silly and foul moods with a laugh. Thank you Jason for being in my life and for being you!

My mom, my dad, and my sister Kiran, for staying by my side through thick and thin, cheering me on, and for always believing in me. They have a big part in shaping who I am today and what I have achieved. Despite being far away, you are close to my heart. I love you.

Policy Architecture for Distributed Storage Systems

Nalini Moti Belaramani, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Mike Dahlin

Distributed data storage is a building block for many distributed systems such as mobile file systems, web service replication systems, enterprise file systems, etc. New distributed data storage systems are frequently built as new environment, requirements or workloads emerge. The goal of this dissertation is to develop the science of distributed storage systems by making it easier to build new systems. In order to achieve this goal, it proposes a new policy architecture, PADS, that is based on two key ideas: first, by providing a set of *common mechanisms* in an underlying layer, new systems can be implemented by defining *policies* that orchestrate these mechanisms; second, policy can be separated into *routing* and *blocking* policy, each addresses different parts of the system design. Routing policy specifies how data flow among nodes in order to meet performance, availability, and resource usage goals, whereas blocking policy specifies when it is safe to access data in order to meet consistency and durability goals.

This dissertation presents a PADS prototype that defines a set of distributed storage mechanisms that are sufficiently flexible and general to support a large range of systems, a small policy API that is easy to use and captures the right abstractions for distributed storage, and a declarative language for specifying policy that enables quick, concise implementations of complex systems.

We demonstrate that PADS is able to significantly reduce development effort by constructing a dozen significant distributed storage systems spanning a large portion of the design space over the prototype. We find that each system required only a couple of weeks of implementation effort and required a few dozen lines of policy code.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed data storage is a building block for many distributed systems such as mobile file systems, web service replication systems, enterprise file systems, etc. Often, these systems take advantage of data replication (i.e. storing copies of data on multiple nodes) to provide availability, durability, and performance guarantees.

However, there are tradeoffs inherent in any replication-based system design. For example, the CAP dilemma [30] states that replication systems cannot provide strong **C**onsistency and high **A**vailability in a network prone to **P**artitions. Also, there is a fundamental tradeoff between **P**erformance and **C**onsistency (known as the PC dilemma [55]). Because of these tradeoffs, it is difficult to design a single replication system that satisfies the needs of all environments and workloads. Every replication system needs to make tradeoffs between availability, consistency, and performance to suit the environment it is targeting. As new workloads, environments, and technologies emerge, new tradeoffs need to be made and new systems need to be developed.

For example, consider the current technology trends: increasing popularity of devices such as smartphones, set-top boxes and cameras, with adequate storage and networking capabilities, and the availablity of cheap always-available cloud storage. Users would want a distributed storage system that allows them to share data (such as photos, music, and files) among their personal devices and cloud, so that they can access the latest version of their data from any device. Due to the mobility and the resource limitations of the devices, the system needs to support arbitrary synchronization topologies, allow different devices to store different subsets of data, and take advantage of the various networking technologies for greater energy-efficiency.

Unfortunately, building a new storage system is a huge task. It can take months or even years to build such a system from scratch. Modifying existing systems to accommodate new requirements may still require a lot of effort. On the other hand, using current state-of-the-art replication frameworks may not be a viable option because they are limited in the range of systems they support (refer to Chapter 2).

The goal of this thesis is to make it easier to design and build distributed storage systems so that new systems can be quickly developed to address the needs of new environments. In order to achieve this goal, this thesis proposes a new policy architecture, PADS, with a new set of abstractions for building distributed storage systems. This work provides both practical and scientific benefits. Its practical benefits include significant reduction in effort required to build new distributed storage systems and increased flexibility allowing constructed systems to quickly adapt to new workloads. As for scientific benefits, this work enhances the understanding of distributed storage systems by identifying the right set of building blocks.

PADS is based on two key ideas: first, *the separation of mechanisms and policy*, and second, *the separation of policy into routing and blocking concerns*.

The first idea is based on the observation that there is large overlap in the low-level details of various replication systems. Every system stores data, transfers updates, and maintains bookkeeping information. The differences between the systems are the choices they make to accommodate their design tradeoffs, i.e. where data items are stored, how and when information is propagated, and what consistency and durability semantics to guarantee. If all the necessary primitives were provided by a common mechanisms layer, a system can be built by specifying policies that orchestrate these primitives.

The second idea separates the design of a system into routing policy and blocking policy.

- *Routing policy*: Many of the design choices of distributed storage systems are simply *routing decisions* about data flows between nodes. These decisions provide answers to questions such as: "When and where to send updates?" or "Which node to contact on a read miss?". They largely determine how a system meets its performance, availability, and resource consumption goals.

- *Blocking policy*: Blocking policy specifies predicates for when nodes must block incoming updates or local read/write requests to maintain system invariants. Blocking is important for meeting consistency and durability goals. For example, a policy might block the completion of a write until the update reaches at least 3 other nodes.

Using the PADS approach has its advantages: First, designers focus on high-level design rather than on low-level implementation. As a result, development time is greatly reduced. Designers can quickly build systems with new innovative techniques for new environments as they emerge. Second, because the underlying framework provides mechanims to take care of difficult details such as failure recovery and consistency enforcement, designers can handle corner-cases easily. Third, because the underlying framework is general, it allows developers to quickly adapt existing systems if requirements change. Fourth, since it is easier to informally review and formally verify higher-level specification, this approach facilitates the development of correct and deadlock-free systems.

However, the main challenge in designing PADS is to ensure that (1) it is sufficiently general to support a wide range of systems including client-server systems, object-replication systems and server-replication systems, (2) it has an API that is easy to learn and allows easy specification of common replication techniques such as demand caching, leases, and callbacks, (3) it makes it easy to reason about corner-cases such as failure detection and recovery, and (4) it is sufficiently efficient so that systems do not have pay a high cost for PADS's generality.

	Simple Client Server	**Full Client Server**	**Coda [49]**	**Coda +Coop Cache**	**TRIP [64]**	**TRIP +Hier**	**Tier Store [26]**	**Tier Store +CC**	**Chain Repl [88]**	**Bayou [69]**	**Bayou +Small Dev**	**Pang-aea [77]**
Routing Rules	24	43	37	45	6	6	14	22	45	10	10	59
Blocking Conditions	5	6	7	7	4	4	1	1	4	3	2	1
Topology	Client/ Server	Client/ Server	Client/ Server	Client/ Server	Client/ Server	Tree	Tree	Tree	Chains	Ad-Hoc	Ad-Hoc	Ad-Hoc
Replication	Partial	Partial	Partial	Partial	Full	Full	Partial	Partial	Full	Full	Partial	Partial
Demand caching	√	√	√	√	√	√						√
Prefetching			√	√	√	√	√	√	√	√	√	√
Cooperative caching		√		√				√			√	√
Consistency	Sequen-tial	Sequen-tial	Open/ Close	Open/ Close	Sequen-tial	Sequen-tial	Mono. Reads	Mono. Reads	Linear-izablity	Causal	Mono. Reads	Mono. Reads
Callbacks	√	√	√	√								
Leases		√	√	√								
Inval vs. whole update propagation	Invali-dation	Invali-dation	Invali-dation	Invali-dation	Invali-dation	Invali-dation	Update	Update	Update	Update	Update	Update
Disconnected operation			√	√	√	√	√	√		√	√	√
Crash recovery	√	√	√	√	√	√	√	√	√	√	√	√
Object store interface*	√	√	√	√	√	√	√	√	√	√	√	√
File system interface*		√	√	√	√	√	√	√	√	√	√	√

Table 1.1: Features covered by case-study systems.
Each column corresponds to a system implemented on PADS, and the rows list the set of features covered by the implementation.
*Note that the original implementations of some systems provide interfaces that differ from the object store or file system interfaces we provide in our prototypes.

To test the usefulness of PADS, we built a PADS prototype as an instantiation of the architecture. The prototype consists of three parts: First, it defines a *common set of mechanisms* that are general enough to build a wide range of systems. Second, it *separates system policy* into two parts: routing policy that defines how data flows between nodes and blocking policy that defines when access to data should block. This dichotomy provides a separation of concerns for designers allowing them to deal with one part of the design at a time. Third, it distills the main abstractions of distributed storage systems into a *small API* and supports *declarative* policy specification making it easy for designers to write complex policy concisely.

Ultimately, the evidence of PADS's benefits is simple: we were able to build 12 diverse systems (summarized in Table 1.1) in 4 months. These systems cover a large part of the design space suggesting that PADS is sufficiently general to support new systems. In addition, PADS's ease of use is demonstrated by the fact that each system was constructed in less than a hundred lines of policy code. Despite the conciseness, every constructed system is concrete and deployable—the implementations handle challenging real-world issues such as configuration and crash recovery. In addition, we find it easy to add new features to PADS-constructed systems. For example, we add cooperative caching to P-Coda, our implementation of Coda [49], in only one week.

Our experience provides evidence of PADS's generality, ease of use, concreteness, and ease of adaptability. This flexibility comes only at a modest cost to absolute performance. Microbenchmark performance of an implementation of one system (P-Coda) built on our user-level Java PADS prototype is within ten to fifty percent of the original system (Coda [49]) in the majority of cases and 3.3 times worse in the worst case we measure.

1.0.1 Contributions

By defining a new policy architecture, this dissertation makes the following contributions:

- It provides a new way to build distributed storage systems that significantly reduces development effort.
- It defines the set of mechanisms required for a common underlying layer over which a wide range of systems can be constructed.
- It proposes an intuitive and structured dichotomy for policy specification.
- It defines a policy API that is small yet sufficient for easy specification of complex systems.
- It provides a prototype with which designers can build new storage systems quickly with modest performance overheads.
- It provides ready-to-use prototypes of 12 diverse systems that cover a large part of the design space.

This dissertation is organized as follows: Chapter 2 makes a case for why a new framework is needed for building storage systems. It looks at the design space covered by distributed storage systems, current approaches for building a system and where they fall short, and defines the requirements for a new framework. Chapter 3 provides a high-level discussion of the PADS approach and the prototype. Chapters 4 and 5 discuss the routing policy and blocking policy respectively including their requirements, specification and implementation. Chapter 6 details the implementation of the mechanisms layer and how it meets the requirements of the policy layer. Chapter 7 details the interface glue between the mechanisms and the policy layers. Chapters 8 and 9 evaluate the benefits of PADS for system development. Chapter 8 evaluates the performance of PADS with various microbenchmarks and chapter 9 evaluates the benefits of PADS as a development platform by describing systems constructed with PADS. Chapter 10 presents some related work. Chapter 11 discusses the limitiations of the architecture and lays out directions for future work. We summarize the main contributions and the lessons learned from this dissertation in Chapter 12.

Chapter 2

Need for a General Framework

This section establishes the need for a general framework to build distributed storage systems. First, it explores the design range which replication-based storage systems cover and introduces a taxonomy to classify them. Next, it provides reasons for why new distributed storage systems will continue to be needed. Then, it describes the tremendous amount of effort required to build a new storage system and how current frameworks fall short in terms of generality. Finally, it defines the requirements for an ideal framework.

2.1 Large Design Space

Time Frame	**Systems**
Pre-1985	NFS [68], Replicated Dictionary [90]
1985-1994	AFS [42], Coda [49], Deceit [80], Ficus [36], Sprite [65] XFS [89], Zebra [38]
1995-2004	Bayou [86], BlueFS [66], Chain Replication [88], CFS [21], Farsite [12], Google File System [29], Ivy [63], OceanStore [50], Pangaea [77], PAST [76], Segank [82], TACT [43]
2005-present	Cimbiosys [70], Dynamo [25], Omnistore [46],Tier-Store [26], WinFS [59], WheelFS [84]

Table 2.1: Partial list of distributed storage systems published in the past 25 years.

Data storage the basic building block for many distributed systems such as mobile file systems, web service replication systems, enterprise file systems, and grid replication systems. Distributed storage systems often use data replication in order to provide durability, availability, and performance guarantees. In fact, over the years, a significant number of replication-based distributed storage systems have been built. Figure 2.1 depicts a subset of systems that have been been published in a few major conferences over the last 25 years.

Because distributed storage systems need to deal with a wide range of heterogeneity in terms of devices with diverse capabilities (e.g., phones, set-top-boxes, laptops, servers), workloads (e.g., streaming media, interactive web services, private storage, widespread sharing, demand caching, preloading), connectivity (e.g., wired, wireless, disruption tolerant), and environments (e.g., mobile networks, wide area networks, developing regions), they cover a wide design space—systems employ different techniques in order to accommodate the requirements, workloads, or environment they target. For example, some systems may replicate all data on every node whereas others may replicate specific subsets on each node. Also, systems propagate updates via different paths (i.e. update propagation topology) and implement different propagation techniques such demand caching, prefetching, co-operative caching. In addition, different systems may guarantee different levels of availability and consistency that are enforced by different means. For example, one system may guarantee data availability at all times at the expense of consistency whereas another may use callbacks and leases to implement stronger consis-

Design Choices		Options	Example Systems
Replication		Full Replication	Bayou [69], Chain Replication [88], TRIP [64], WinFS [67]
		Partial Replication	BlueFS [66], Coda [49], Sprite [65], Pangaea [77], TierStore [26]
Update Propagation	Topology	Client-Server	BlueFS [66], Coda [49], TRIP [64]
		Structured	Chain Replication [88], Sprite [65], TierStore [26], WheelFS [84]
		Ad-hoc	Bayou [69], IceCube [47], Pangaea [77]
	Type	Operation	Bayou [69], IceCube [47]
		State (Update only)	Ficus [36], TACT [43], TierStore [26], WheelFS [84]
		State (Update and Invalidation)	BlueFS [66], Coda [49] Fluid Replication [20]
Consistency	Guarantees	Strong	Chain Replication [88], TRIP [64]
		Causal	Bayou [69], TACT [43]
		Eventual	Ficus [36], Pangaea [77], WinFS [67]
	Commit Policy	Implicit	Pangaea [77], TierStore [26]
		Primary-commit	Bayou [69], Coda [49]
Availability		Always Available	Bayou [69], TierStore [26], Pangaea [77]
		Only When Connected	AFS [42], Coda [49], Chain Replication [88]
Durability		Local Sufficient	Bayou [69], TierStore [26], Pangaea [77]
		Multi-node	Coda [49], Chain Replication [88], WheelFS [84]

Table 2.2: Storage systems design space and examples

tency semantics and be unavailable in cases of network partitions. Another design difference between systems is when a system considers an update to be durable. For some systems, an update is sufficiently durable if it is persistent to local disk, whereas other systems may require an update to be persistent on multiple nodes to be considered durable. Table 2.2 provides a partial list of the design space covered by distributed storage systems.

2.1.1 PRACTI Taxonomy

In order to aid the understanding of the design space covered by replication-based systems, we define the *PRACTI* taxonomy. The taxonomy is based on three properties:

- *Partial Replication* (PR): a system can place any subset of data and metadata on any node. In contrast, some systems require a node to maintain copies of all data in the system [69, 90, 94].
- *Arbitrary Consistency* (AC): a system can provide both strong and weak consistency guarantees and only applications that require strong guarantees pay for them. In contrast, some systems can only enforce relatively weak coherence guarantees but make no guarantees about stronger consistency properties [36, 77].
- *Topology Independence* (TI): any node can exchange updates with any other node. In contrast, many systems restrict communication to client-server [42, 49, 65] or hierarchical [17, 92] patterns.

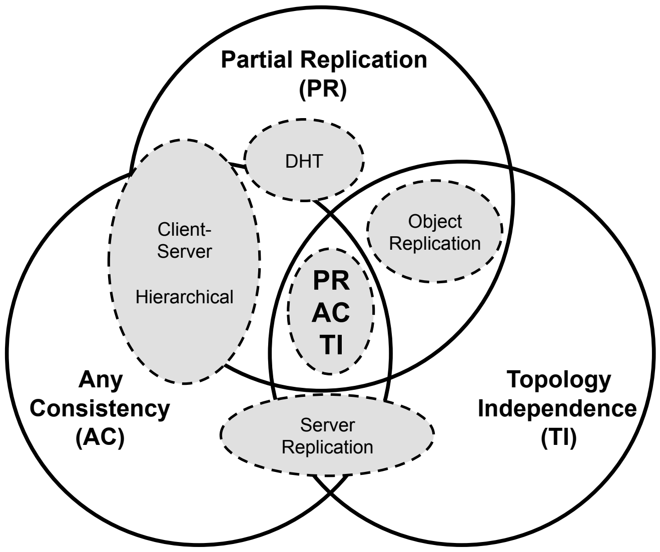

Figure 2.1: PR-AC-TI taxonomy for classifying families of storage systems.

Every distributed storage system can be categorized according to which of the PR-AC-TI properties it provides, as illustrated in Figure 2.1. As the figure indicates, many existing storage systems can be viewed as belonging to one of four protocol families, each providing at most two of the PR-AC-TI properties.

Server replication systems like Replicated Dictionary [90] and Bayou [69] 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 [51] and TACT [94] use this approach to provide a wide range of tunable consistency guarantees (AC). Some server replication systems like Chain Replication [88] do not assume topology independence and allow updates to propagate via fixed paths. Unfortunately, sever replication 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 [65] and Coda [49] and *hierarchical* caching systems like hierarchical AFS [62] 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). However, these protocols fundamentally require communication to flow between a child and its parent. Even when systems permit limited client-client communication for cooperative caching [14, 23, 28], they must still serialize control messages at a central server for consistency [18]. 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 [87], PAST [76], and CFS [21] 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 [73].

Object replication systems such as Ficus [36], Pangaea [77], and WinFS [59] 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.

Our analysis indicate that there is a fundamental challenge to provide all three PR-AC-TI properties: 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.

2.2 Need for New Storage Systems

Despite the plethora of existing systems, new systems are constantly developed to meet the needs of new environments and workloads. The main reason is that there are fundamental tradeoffs inherent in every storage system design. For example, the CAP dilemma [30] states that replication systems cannot provide strong **C**onsistency and high **A**vailability in a network prone to **P**artitions. Also, designs often need to make tradeoffs between **P**erformance and **C**onsistency (PC dilemma [55]). Because of these tradeoffs, every system makes design choices that balance performance, resource usage, consistency and availability to best meet the needs of the workload or environment it is targeting. As new environments and new workloads emerge, they often require different tradeoffs that existing systems cannot accommodate creating a need for new systems.

System	Lines of Code
Coda [49]	167,564
TierStore [26]	22,566
Pangaea [77]	30,000
Bayou [69]	20,425
TRIP [64]	16,909

Table 2.3: Approximate lines of code of several existing distributed data storage systems.

2.3 Difficulty in Building a New Storage System

Unfortunately, building a new storage system is not easy. A common approach to build a storage system is to implement it from scratch. The advantage of this approach is that it allows designers to hand-craft optimal tradeoffs for the target workload or environment. Unfortunately, this approach can take months or years of development effort: designers need to implement all the low-level details, often re-implementing common techniques. Also, getting the consistency guarantees right can be tricky. Table 2.3 provides an indication of the amount of effort required to implement several systems by listing their lines of code.

Another approach involves modifying existing systems to accommodate new tradeoffs. Although this approach would enable a lot of code-reuse, it is not straight-forward. For many systems, their design tradeoffs are embedded in their low-level implementation and there is a limit to how much they can be modified.

The third approach involves using an existing distributed storage framework. Unfortunately, current frameworks fall short in terms of generality: they support only full replication or make assumptions regarding update propagation topology. Systems built with these frameworks can only target specific environments and may have limited capabilities to adapt to new requirements.

For example, the TACT toolkit [43] allows designers to specify their consistency requirements in terms of *order error*, *staleness*, and *numerical order*. The toolkit automatically sets up synchronization streams to meet these requirements. Unfortunately, it assumes full replication of data among nodes. Swarm [85] is another wide-area peer-replication middleware that allows designers to pick consistency-related options for replica synchronization, failures, and concurrency. Again, it only supports full replication and carries out update propagation only in a hierarchical fashion.

WheelFS [84], a wide-area distributed file system, supports partial replication and allows applications to provide semantic cues to specify consistency, replication level, and placement. However, WheelFS only supports two levels of consistency on a per-object basis: open-to-close consistency or eventual. Also, it assumes a single-primary-multiple-backup scheme for update propagation. Deceit [80], a flexible distributed file system that also supports partial replication and allows users to set parameters for a file to achieve different levels of availability, performance, and consistency. Unfortunately, it is only applicable to specific environments: replication among well-connected servers and client-side caching. Fluid replication [20] also provides a menu of options for consistency and automatically creates replicas to meet those requirements. Unfortunately, it is restricted to hierarchical caching.

Stackable layers [39], allows designers to create flexible storage systems with layers that provide different services. Unfortunately, it supports only local storage not distributed storage.

2.4 Need for a General Framework

Because current approaches for building new distributed storage systems require substantial effort or are suited for very specific environments, there is a need for a general framework that can make it easier and faster to develop a wide range of new distributed storage systems. An ideal framework must support all PR-AC-TI properties. In particular, it must

- allow any node to store any subset of data,
- not restrict update propagation to a specific topology,
- allow designers to specify a wide range of consistency semantics, and
- facilitate development and evolution of systems to accommodate new requirements.

This dissertation addresses this need by introducing the PADS architecture for constructing distributed storage systems. Later chapters provide details of PADS, how it provides three PR-AC-TI properties, and how it simplifies system development.

Chapter 3

The PADS Approach

The previous chapter argues that a new general framework would make it easier to build distributed storage systems. In order to meet that need, this thesis proposes the PADS approach to simplify the design and implementation of storage systems. The PADS approach is based on two key ideas: First, by separating policy from mechanisms, system developers can focus their energies on high-level design rather than implementing low-level details. Second, by separating policy into routing and blocking concerns, the task of building a system is divided into two smaller sub-problems. This chapter discusses the intuition behind these approaches and briefly describes how these ideas are realized in the PADS prototype.

3.1 Separation of Mechanisms and Policy

During the development of a distributed replication system, a lot of effort is spent on the implementation and the debugging of low-level details, such as data storage, transmission of data, bookkeeping, conflict detection, etc. Also, there is a considerable overlap in the low-level techniques that different systems require, leading to wasted re-engineering effort.

PADS argues that by constructing a *flexible mechanisms layer* that provides low-level primitives, system development is reduced to writing *policies* that orchestrate these primitives, thereby reducing the implementation effort required.

PADS casts the mechanisms as part of a *data plane* and policies as part of a *control plane*. The data plane encapsulates a set of common mechanisms that handle details of storing and transmitting data and maintaining consistency information. System designers then build storage systems by specifying a control plane policy that call upon these mechanisms to implement system design.

The separation of policy from mechanisms significantly reduces development effort for several reasons: First, since the mechanisms layer takes care of the grunt work, system designers can focus their energies on high-level design issues. Second, debugging effort is reduced because developers only need to focus on policy debugging rather than debugging the underlying mechanisms layer. The effort spent on debugging the mechanisms layer is amortized over all the systems built over it. Third, if system requirements change, only policy needs to be updated rather than the low-level primitives.

3.2 Separation of Policy into Routing and Blocking

Since distributed storage systems span a wide design space, the next question is how should policies be specified so that the designs of different systems can be easily constructed over the mechanisms layer.

Our observation is that a lot of the design choices that distributed systems make can be seen as how data is *routed* among nodes and when it is "safe" to access data. PADS is built around this intuition and separates policy specification along this dichotomy: *routing policy* and *blocking policy.*

Routing policy sets up data flows between nodes. It provides answers to questions such as: "When and where to send updates?" or "Which node to contact on a read miss?", and largely determines how a system meets its performance, availability, and resource consumption goals. For example, in Bayou [69], a server replication system, a node periodically sets up an update flow to a random peer to exchange updates. In chain replication [88], another server replication protocol, updates occur at the head of each chain and flow down to the tail. In Coda [49], a client-server protocol, a client will fetch data from the server on a miss and send updates to the server when a file is closed or when an unreachable server becomes reachable.

Blocking policy can be seen as specifying the *system invariants* that must hold before or after an operation can continue. If the system invariants do not hold, then the operation is blocked. For example, for durability requirements, the completion of a write might block until the update reaches three other nodes. For consistency requirements, the read of a local data object might block until the local version matches the latest version in the system.

The advantage of this separation is that development is split into two smaller sub-problems presenting a division of concerns to the designer. Also, because routing and blocking concerns are very different, their separation enables PADS to provide specialized policy API designed to make the specification of each concern simpler. On another level, blocking policy and routing policy can be seen as working together to enforce the safety constraints and the liveness goals of a system. Blocking policy enforces safety conditions by ensuring that an operation blocks until system invariants are met, whereas routing policy guarantees liveness by ensuring that an operation will eventually unblock—by setting up data flows to ensure the conditions are eventually satisfied.

3.3 PADS Prototype

To test effectiveness of PADS, we built a Java-based prototype and constructed a dozen systems with it. The prototype consist of 3 parts: a flexible and efficient *mechanisms layer*, a *routing policy API*, and a *blocking policy API*. The mechanisms layer implements all the basic primitives for storing data objects, sending and receiving updates, and maintaining consistency information in a way that is able to support all PR-AC-TI properties. The policy API allows designers to invoke the mechanisms but does so in a way that is general enough to support a wide range of systems.

The key features of the prototype that enable it to simplify development include

- its *concise policy API* that is intuitive, easy to learn, and captures the right abstractions for distributed storage, and

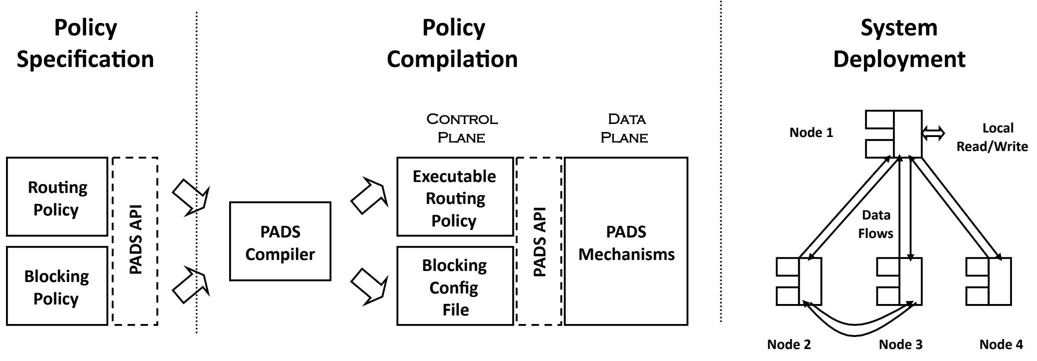

Figure 3.1: PADS approach to system development.

- its support for a *declarative policy specification* that enables quick and concise implementation of complex systems. Routing policy is written as a set of rules in a declarative domain specific language called R/OverLog. Blocking policy is written as a set of predicates in a configuration file.

The following chapters (chapters 4, 5, 6 and 7), detail each part of the prototype. Chapter 9 provides evidence of PADS's effectiveness by describing the range of systems built with PADS.

3.3.1 Using PADS

As Figure 3.1 illustrates, in order to build a distributed storage system on PADS, a system designer writes a routing policy as a set of rules and a blocking policy as a set of predicates. She uses a PADS compiler to translate her routing rules to Java byte-code and places the blocking predicates in a configuration file. Finally, she distributes a Java jar file containing PADS's standard data plane mechanisms and her system's control policy to the system's nodes. Once the system is running at each node, users can access locally stored data, and the system synchronizes data among nodes according to the policy.

A PADS node can be seen as an instantiation of a routing and a blocking policy over the mechanisms layer on a single system node.

3.4 Scope and Limitations

A PADS policy is a specific set of directives rather than a statement of a system's high-level goals. Distributed storage design is a creative process and PADS does not attempt to automate it: a designer must still devise a strategy to resolve trade-offs among factors like performance, availability, resource consumption, consistency, and durability. For example, a policy designer might decide on a client-server architecture and specify "When an update occurs at a client, the client should send the update to the server within 30 seconds" rather than stating "Machine X has highly durable storage" and "Data should be durable within 30 seconds of its creation" and then relying on the system to derive a client-server architecture with a 30 second write buffer.

PADS targets distributed storage environments with mobile devices, nodes connected by WAN networks, or nodes in developing regions with limited or intermittent connectivity. In these environments, factors like limited bandwidth, heterogeneous device capabilities, network partitions, or workload properties force interesting tradeoffs among data placement, update propagation, and consistency. Conversely, we do not target environments like well-connected clusters.

Within this scope, there are four design issues for which the current PADS prototype restricts a designer's choices:

First, the prototype does not support security specification. Ultimately, our policy architecture should also define flexible security primitives, and providing such primitives is important future work [57].

Second, the prototype exposes an object-store interface for local reads and writes. It does not expose a file system or a tuple store interface. We believe that these interfaces are not difficult to incorporate. Indeed, we have implemented an NFS [68] interface over our prototype.

Third, the prototype provides a single mechanism for conflict resolution. Write-write conflicts are detected and logged in a way that is data-preserving and consistent across nodes to support a broad range of application-level resolvers. We implement a simple last-writer-wins resolution scheme and believe that it is straightforward to extend PADS to support other schemes [26, 47, 49, 79, 86].

Fourth, the prototype only supports state-transfer (i.e. updates and meta-data about updates) and not operation-transfer [47, 69].

3.5 Summary

In order to simplify the development of distributed storage systems, the PADS approach incorporates two ideas: First, systems are constructed by specifying high level-policy over a common mechanisms layer. Second, policy specification is separated into routing policy and blocking policy, each of which addresses different design concerns. Later chapters provide details of the policy layer, the mechanisms layer, and the advantages of PADS for system development.

Chapter 4

Routing Policy

In PADS, routing policy specifies how data and meta-data propagate among nodes. This chapter discusses what exactly constitutes routing policy and how it can be specified. The challenge is to define an API that is sufficiently expressive to construct a wide range of systems and yet sufficiently simple for a designer to comprehend. We discuss how PADS meets this challenge.

4.1 What is Routing?

A large part of a distributed storage system's design defines how updates and information propagate among nodes to ensure that system meets its performance and availability goals. The design provides answers to questions such as: "When and where to send updates?" or "Which node to contact on a read miss?". In effect, the design sets up a overlay among the nodes for information propagation. For example, in a traditional client-server system, a client fetches data from the server and sends updates to it. Its design can be seen as setting up a star-shaped overlay among the nodes with the server in the center and clients at the edge. To implement hierarchical caching, a system defines a tree overlay among nodes so that updates or notifications propagate in a hierarchical manner.

What is needed from a policy layer are primitives to set up flows among nodes, ways to retrieve information about important events (e.g. read miss), and a means to invoke the flow primitives based on the information received. In order to address these needs, PADS provides the following tools:

- *Subscription*: a flexible primitive to set up data flow between two nodes.
- *Event-driven API*: that informs policy about important events and allows routing policy to set up data flows at appropriate times.
- *R/OverLog*: a domain specific declarative language for easy policy specification.

In short, in PADS, routing policy is written as an event-driven program comprising of a set of declarative rules that set up or remove subscriptions when particular events occur. We detail them in later sections of this chapter. Note that we do not claim that these primitives are minimal or that they are the only way to realize this approach. However, they have worked well for us in practice.

4.2 Subscriptions

Systems set up data flows differently. In client-server systems, clients only send updates to the server, whereas in peer-to-peer systems, any node can send updates to any other node. Nodes in a server replication system often exchange updates to the whole data set, whereas clients in a client-server system only retrieve updates for the data they cache. Some systems propagate notifications rather than updates to quickly propagate update information.

In order to support different types data flows, the primitives provided by PADS must be efficient, support all three PR-AC-TI properties, support separate data and meta-data propagation, expose options for different performance tradeoffs, and allow incremental progress in cases of disruptions.

Instead of having multiple data flow primitives, PADS is able to encapsulate sufficient flexibility into a single primitive—a subscription. A subscription is an unidirectional stream of updates between a pair of nodes. For each subscription, the policy writer can choose

- *The source and the destination nodes*: Updates are sent from the source node to the destination node. Since subscriptions can be set up between any nodes, they support topology independence (TI).
- *The subset of data that is of interest to the subscription*: Only updates to objects that are in the interest set will be sent on the subscription. This allows subscriptions to support partial replication (PR).
- *What type of information should be sent*: There are two choices: invalidations or bodies. An invalidation indicates that an update to a particular object occurred at a particular instant in time, whereas a body contains

the data for a specific update. The advantage of separating invalidation and body streams is that it enables meta-data and data to propagate via different paths. A node can quickly and cheaply inform other nodes about an update, via an invalidation, without having to send the entire update. Also, nodes can choose to only receive bodies of the updates they care about, leading to better bandwidth efficiency.

- *The logical start time*: Only updates that have occurred to the objects of interest from the start time will be sent on the subscription. Therefore, a new subscription can continue where the previous one left off supporting incremental progress.

- *The catchup method*: If the start time of an invalidation subscription is earlier than the sender's current time, then the sender has two options, each of which has different performance tradeoffs: the sender can transmit either a *log* of the updates that have occurred since the start time or a *checkpoint* of the latest versions of the objects. Body subscription, however, only send checkpoints and do not support the log option.

Hence, setting up data flows is reduced to setting up subscriptions among nodes. For example, if a designer wants to implement hierarchical caching, the routing policy would set up subscriptions among nodes to send updates up and to fetch data down the hierarchy. If a designer wants nodes to randomly gossip updates, the routing policy would set up subscriptions between random nodes. If a designer wants mobile nodes to exchange updates when they are in communication range, the routing policy would probe for available neighbors and set up subscriptions at opportune times.

The advantage of having a single primitive is that it makes make system design cleaner—all routing is set up in terms of subscriptions.

4.3 Event-driven API

In order to make it easier to write routing policies, the interface should allow designers to do four things: to set up subscriptions, to receive information about events, to store and look up data objects in the underlying layer, and to communicate with blocking policy . For example, a client's routing policy needs to know when a local read miss occurs so that it can establish a subscription to retrieve the required data from the server. In a publish/subscribe system, a node may store and read from a configuration object the sets of data it is interested in. For consistency reasons, a blocking policy may block the completion of a write until the routing policy indicates (by sending an message to the blocking policy) that the update has propagated to the required nodes.

PADS provides three sets of primitives for specifying routing policies: (1) a set of 9 *triggers* that expose events pertaining to the state of the data plane (2) a set of 7 *actions* for establishing and removing subscriptions (3) a set of 5 *stored events* that allow a policy to persistently store and access data objects for configuration options and information affecting routing decisions. Consequently, writing routing policy entails invoking the appropriate actions or storing events based on the triggers and events received.

Local Read/Write Triggers	
Operation block	obj, off, len, blocking_point, failed_condition
Write	obj, off, len, writerId, time
Delete	obj, writerId, time
Message Arrival Triggers	
Inval arrives	srcId, obj, off, len, writerId, time
Send body success	srcId, obj, off, len, writerId, time
Send body failed	srcId, destId, obj, off, len, writerId, time
Connection Triggers	
Subscription start	srcId, destId, objS, inval|body
Subscription caught-up	srcId, destId, objS, inval
Subscription end	srcId, destId, objS, reason, inval|body

Table 4.1: Routing triggers provided by PADS.
objId, *off*, and *len* indicate the object identifier, offset, and length of the update to be sent. *blocking_point* and *failed_condition* indicate at which point an operation blocked and what condition failed (refer to Chapter 5). *writerId and time* indicate the logical time of a particular update. *srcId* and *destId* are the identifiers of the source node and the destination node respectively. *objS* indicates the interest set of the subscription. *inval* | *body* indicate the type of subscription. *reason* indicates whether the subscription ended due to failure or termination.

In fact, because the number of primitives provided by PADS is small, designers are not overwhelmed by too much choice. Most importantly, since we were able to express a wide range of systems with these primitives (refer Chapter 9), we believe that these primitives capture the fundamental abstractions for distributed storage systems.

The following subsections detail these primitives.

4.3.1 Triggers

PADS *triggers* expose to the routing policy events that occur in the data plane. As Table 4.3.1 details, these events fall into three categories.

Subscription Actions	
Add Inval Sub	srcId, destId, objS, [startTime], LOG|CP|CP+Body
Add Body Sub	srcId, destId, objS, [startTime]
Send Body	srcId, destId, objId, off, len, writerId, time
Remove Inval Sub	srcId, destId, objS
Remove Body Sub	srcId, destId, objS
Consistency-related Actions	
Assign Seq	objId, off, len, writerId, time
B_Action	<policy defined>

Table 4.2: Routing actions provided by PADS.
objId, *off*, and *len* indicate the object identifier, offset, and length of the update to be sent. *startTime* specifies the logical start time of the subscription. *writerId and time* indicate the logical time of a particular update. The fields for the *B_Action* are policy defined.

- *Local operation* triggers inform the routing policy when an operation blocks because it needs additional information to complete or when a local write or delete occurs.

- *Message receipt* triggers inform the routing policy when an invalidation arrives or whether a send body succeeds or fails.

- *Connection* triggers inform the routing policy when a subscription is successfully established, when a subscription has caused a receiver's state to be caught up with a sender's state (i.e., the subscription has transmitted all updates to the subscription set up to the sender's current time), or when a subscription is removed or fails.

4.3.2 Actions

PADS *actions* allow the routing policy to set up data flows (i.e. subscriptions) among nodes. As Table 4.3.2 details, these actions fall into two categories.

- *Add subscription* actions that establish streams of invalidations or bodies for specific subsets of objects between two nodes. The *send body* action is a special case of a subscription in which only the body of a single object is transferred.
- *Remove subscription* actions that terminate invalidation or body streams between two nodes.

PADS also defines a third type of actions that aid the enforcement of consistency: the *Assign Seq* action and the *B_Action*. *Assign Seq* action marks a previous update with a commit sequence number allowing policies to implement various consistency and commit protocols. For example, in a client-server system, in order to enforce serializability, when a server receives an update to an object, it uses the *Assign Seq* action to commit the update once it is sure that all other versions of the object (located on other clients) have been invalidated. Serializability can be guaranteed at clients by reading only valid committed versions (refer to Chapter 9). *B_Action* allows routing policy to send messages to blocking policy. Blocking policy may define conditions based on information available to routing policy. For example, whether the

Stored Events	
Write event	objId, eventName, field1, ..., fieldN
Read event	objId
Read and watch event	objId
Stop watch	objId
Delete events	objId

Table 4.3: PADS's stored events interface.
objId specifies the object in which the events should be stored or read from. *eventName* defines the name of the event to be written and *field** specify the values of fields associated with it.

node is disconnected from the server, or whether an update has reached 3 other nodes. Once the conditions are satisfied, the routing policy uses the *B_Action* to inform the blocking policy so that the operation can continue.

4.3.3 Stored Events

Many systems look up or maintain persistent state to make routing decisions. For example, in a client-server system, routing policy needs to look up the identity of the server in order to determine who to establish subscriptions to. In a publish/subscribe system, the routing policy may need to look up or maintain the sets of objects for which to establish subscriptions.

Supporting this need is challenging both because we want an abstraction that meshes well with our event-driven programming model and because the techniques must handle a wide range of scales. In particular, the abstraction must not only handle simple, global configuration information (e.g., the server identity in a client-server system like Coda [49]), but it must also scale up to per-file information (e.g., which nodes store the gold copies of each object in Pangaea [77].)

PADS defines a new abstraction, the *stored events* primitives, in order to store and retrieve events from a data object in the underlying persistent object store. Table 4.3 details the full API for stored events. A *Write Event* stores an event into an object and a *Read Event* causes all events stored in an object to be fed as input to the routing program. The API also includes *Read and Watch* to produce new events whenever they are added to an object, *Stop Watch* to stop producing new events from an object, and *Delete Events* to delete all events in an object.

The stored events primitives enable routing policy to implement various scenarios intuitively in the event-driven fashion.

For example, in a client-server system, clients need to know the identity of the server to contact. At configuration time, the installer writes the event <`add_server, serverID`> to the object `/config/server`. At startup, the routing policy generates a <`read_event, /config/server`> action that looks up the `/config/server` object. The `add_server` event is retrieved from the object which, in turn,triggers actions in the routing policy to establish subscriptions to the server.

In a hierarchical information dissemination system, a parent p keeps track of what volumes a child subscribes to so that the appropriate subscriptions can be set up. When a child c subscribes to a new volume `v`, p stores the information in a configuration object `/subInfo` by generating a

<`write_event, /subInfo, child_sub, p, c, v`> action. When this information is needed, for example on startup or recovery, the parent generates a <`read_event, /subInfo`> action that causes a <`child_sub, p, c, v`> event to be generated for each item stored in the object. The `child_sub` events, in turn, trigger actions in the routing policy that re-establish subscriptions.

In some client-server systems [49], a client maintains a list of objects to prefetch from the server (i.e. a hoard list). The routing policy can store an item on the hoard list by using the `write_event` action to store the <`hoard_item, objId`> event into the persistent object `/config/[nodeID]/hoardlist`. Later when the policy wishes to walk the hoard list to prefetch objects, it can use the `read_event` action to produce all `hoard_item` events stored in the hoard list. The production of these events, in turn, activates policy actions to fetch an item from the server.

In Pangaea [77], each file's directory entry includes a list of *gold nodes* that store copies of that file. To implement such fine-grained, per-file routing information, a PADS routing policy creates a `.meta` object for each directory, stores the gold node information for each of its children as <`file_entry, objId, goldNodeList`>, and updates a file's `file_entry` whenever the file's set of gold nodes changes (e.g., due to a long-lasting failure). When a read miss occurs, the routing policy produces the stored `file_entry` event from the parent `.meta` object, and this event activates rules that route a read request to one of the file's gold nodes.

4.4 R/OverLog

In order to make it easier to write routing policies, PADS provides designers with a domain specific language, R/OverLog—a declarative language based on OverLog [56] network routing language.

The advantage of using R/OverLog for policy specification is three-fold: First, R/OverLog is event-driven so it fits well with the PADS paradigm. Second, since R/OverLog is based on a network routing language, it allows policies to use the same interface to do network management and invoke PADS actions based on network events. For example, a policy can easily specify "add a subscription when a peer is detected". Third and most importantly, because PADS uses the routing abstraction for replication, for which OverLog and R/OverLog have been specially designed, complex policies can be easily and concisely specified. Statements in an R/OverLog program closely follow pseudocode, making programs easy to write. Also, most R/OverLog programs fit in a couple of pages making it easier to do design reviews. Granted that there is a learning curve to learn R/OverLog, but the compactness of policies and the ease with which they can be written make the effort worth-while.

Since PADS mechanisms are written in Java, a R/OverLog routing policy is translated into Java and is compiled before being instantiated over the mechanisms. This section provide details of the R/OverLog language semantics. Details of how R/OverLog is translated into Java are provided in Chapter 7.

4.4.1 R/OverLog Language Semantics

A R/OverLog program defines a set of *tables* and a set of *rules*. Tables store *tuples* that represent internal state of the routing program, whereas rules define the logic of the program and are fired when *events* occur. We discuss each part of the language semantics in turn.

4.4.1.1 Tuples, Events and Data Types

Tuples represent the state of an R/OverLog program and are stored in tables. Every tuple has a name and consists of a series of fields. An event can be considered as a special case of a tuple that is not stored in a table.

Since R/OverLog is a typed-language, it defines 5 data types for fields:

- `int`: an integer (For example, `1`)
- `float`: an float (For example, `3.56`)
- `boolean`: a boolean value (For example, `true`)
- `string`: a string of characters enclosed in double quotes (For example, `"orange"`)
- `location`: network address of a node, including the hostname or IP address and port number. (For example, `myhost.mydomain.com:5000`)

In order to define field types, type declaration statements need to be used as follows:

```
tuple <name>(location [, type*]).
```

where `name` is string that starts with a lower-case letter. The first field of a tuple or an event must be of the `location` type. The location represents the node in the tuple is stored or the node to which the event should be sent. For example, the following type declaration:

```
tuple neighbor(location, location, float).
```

declares a `neighbor` tuple with 3 fields: 2 locations and 1 float.

In practice, it is not necessary to explicitly write tuple declarations for all tuples in the program. The number and types of fields are inferred from their use in program rules. However, if type information cannot be inferred, an error is reported and a tuple declaration is needed.

4.4.1.2 Tables and Facts

Tables store the state of the program as tuples. They must be explicitly defined via materialization statements that specify the name of the table, how long the tuples are retained, the maximum number of tuples stored in the table, and the fields making up the primary key. For example, the following table declarations

```
materialize (neighbor, 10, infinity, keys(1,2)).
materialize (sequence, infinity, 1, keys(1)).
```

define two tables: first, a `neighbor` table whose tuples are retained for 10 seconds, has unbounded size, and the primary key of the table is made up of the first two fields; second, a `sequence` table that stores only 1 tuple (i.e. the latest one added to the table) indefinitely and primary key of the table is the first key of the `sequence` tuple. Note that table names, like tuple and event names, always being with a lower-case letter.

Tuples are added to tables when they are generated due to an execution of a rule or are hard-corded in the program as *facts*. For example, the following statements

```
neighbor(host1.mydomain.com:5000, host2.mydomain.com:5000, 3).
neighbor(host1.mydomain.com:5000, host3.mydomain.com:5000, 0.5).
```

add two tuples to the neighbor table.

4.4.1.3 Timing Events

R/OverLog consists of a built-in event generator `periodic` that is generates events at periodic intervals. For example, the following rule:

```
r0 refreshEvent(@X):- periodic(@X, 5, -1).
```

generates a *refreshEvent* event every 5 seconds. The second field in the `periodic` tuple specifies the length of the period (in seconds), and the third field specifies the number of events that should be generated. The value `-1` specifies that events should be generated as long as the program is running

4.4.1.4 Rules

The logic of the program is written as a set of rules. Rules have the form:

```
<ruleId> <head>:- <body>.
```

The `ruleId` is a string identifier for the rule. The `head` of the rule specifies the events or the tuples generated when the rule is fired (i.e. executed) Generated events fire other rules. All generated tuples are added to their corresponding tables. The `body` of the rule specifies the constraints that must hold in order for the rule to fire. It consists of at most one event, table lookups, variable assignments, and constraints separated by commas. The commas are interpreted as an *AND* logical construct. Variables in the body begin with upper-case letters. Consider the follow rules:

```
r1  refreshSeq(@X, NewSeq):-
         refreshEvent(@X), sequence(@X, Seq), NewSeq=Seq+1.

r2  sequence(@X, NewSeq):-
         refreshSeq(@X, NewSeq).
```

The variable `X` and the at sign (`@`) represent the node at which the program is running. The rule `r1` specifies that whenever a `refreshEvent` is received, the sequence number is looked up from the `sequence` table and is incremented. A new `refreshSeq` event is generated with the new sequence number. The rule `r2` stores the new sequence number back in the sequence table.

Note the at sign(`@`) is also used to send an event to other nodes. Consider the following rule.

```
p1  ping(@Y, X):-
         doPing(@X), neighbor(@X, Y, _).
```

When a doPing event is received, the rule p1 looks up the neighbor table and sends a ping event to every neighbor, represented by @Y. The underscore sign (_) is a wild-card and matches any value.

The body of a rule must conform to the following two constraints: First, all table lookups and events in the body of a single rule must be localized to a single node. Second, the body consists of at most one event. In the case that the body as no event, the rule is fired whenever a new tuple is added to the tables specified in the body.

In order to delete a tuple from a table, the head of rule contains the delete keyword. For example, the following rule

```
d1  delete neighbor(@X, Y, L):-
         removeNeighbor(@X), neighbor(@X, Y, L),
         Y==host2.mydomain.com:5000.
```

removes from the tuple for host2 from neighbor table.

The R/OverLog program exits if a rule with the tuple exit(@X) is generated due to the firing of a rule.

4.4.1.5 Aggregates

R/OverLog allows a rule to carry out aggregation on the tuples that match the constraints in the body. For example, the following rule is fired by the checkStatus event, looks up the neighbor table and finds the minimum latency to generate the minLatency event.

```
minLatency(@X, a_MIN<L>):-
    checkStatus(@X), neighbor(@X, _, L).
```

Other supported aggregates include finding the maximum or average value of a field (`a_MAX` /`a_AVG`), counting the number of matched tuples (`a_COUNT`), and picking a random value (`a_RANDOM`). Note that aggregates are specified in the head of the rule and not the body.

4.4.1.6 Functions

R/OverLog defines two built-in functions that can be used for specifying assignments or conditions in the body: `f_now()` returns the current clock time as an integer and `f_rand()` returns a random float between 0 and 1. For example, the following rule generates a `currTime` event that contains the current clock time every minute.

```
t1 currTime(@X, T) :- periodic(@X, 60, -1), T= f_now().
```

For ease of specification, programmers can extend R/OverLog to define their own functions. They must declare the function in the R/OverLog program and define it in the R/OverLog runtime (refer to Chapter 7). For example, the following function defines a `f_getParent` function that takes in a string argument and returns a string.

```
fun string f_getParent(string).
```

4.4.1.7 Watch Statements

Watch statements are used to print local events or table updates. and are useful debugging tools. For example, the following statement prints every ping event received or generated.

```
watch(ping).
```

4.4.2 R/OverLog Policies

The routing policy is written as an R/OverLog program. In order to support the routing policy API, the underlying layer inserts triggers as events into the R/OverLog program and whenever an event that matches an action is generated by a rule, it invokes the appropriate mechanism primitive. For example, the follow rule:

```
sub01  addInvalSub(@X, S, X, Obj, CTP):-
            operationBlock(@X, Obj, Off, Len, BPoint,_),
            serverId(@X, S), CTP = "CP",
            BPoint == "readNowBlock".
```

specifies that whenever node `X` receives a `operationBlock` trigger informing it of an operation that is blocked at the `readNowBlock` blocking point, it should look up the server id from the `serverId` table, and produce an event `addInvalSub` event that will invoke the action to establish a subscription from the server for the object with check point catchup.

Hence, for a rule in the routing policy, the input event a trigger injected from the data plane, a stored event injected from the data plane's persistent state, or an internal event produced by another rule on a local machine or a remote machine. Every rule generates a single event that invokes an action in the data plane, fires another local or remote rule, or is stored in a table as a tuple.

4.4.3 R/OverLog vs. OverLog

The first version of our prototype employed OverLog as the language for writing routing policies. Unfortunately, the runtime to execute OverLog was slow, unstable, and did not provide a consistent rule execution model. We realized that even though the language was good for policy specification, the runtime made it impossible to use. Hence, we developed our own language and execution runtime for PADS. The main differences between R/OverLog and OverLog are as follows:

R/OverLog restricts a rule from doing remote table lookups. In other words, for a single rule, all the right side predicates (i.e. the triggering event and table lookups) reside on the same node. Since all communication between nodes are carried out with explicit message transfers, it was easier to implement an efficient runtime for R/OverLog.

R/OverLog implements *fixed point semantics* making it easier to reason about rule execution. The semantics guarantees that all rules triggered by the appearance of the same event are executed atomically in isolation from one another. Once all such rules are executed, their table updates are applied, their actions are invoked, and the events they produce are enqueued for future execution. Then, another new event is selected and all of the rules it triggers are executed. The original OverLog runtime did not support fixed-point semantics, but subsequent implementation have added support for it.

R/OverLog adds an interface to insert and receive events from a running R/OverLog program. This interface is important for PADS to inject *triggers* to and receive *actions* from the policy. On the other hand, the OverLog runtime was not designed to receive external events. R/OverLog also adds type information to events making the language safer.

Finally, in order to execute a R/OverLog program, it needs to be translated to Java and compiled before it can run on the provided runtime. OverLog, on the other hand, executes directly on its runtime. The design and the implementation of the runtime is detailed in Section 7.2.2.

4.5 Summary

Routing policy specifies how data and metadata flow in a distributed storage system. This chapter defines the primitives by PADS to support information routing as well as the policy API provided to designers for easy specification. Routing policy is written as an event-driven program as a set of rules in *R/OverLog*, a declarative language designed for this purpose, over an API comprising of a set of *actions* that set up data flows, a set of *triggers* that expose local node information, and a set of *stored events* to store and retrieve persistent state.

Chapter 5

Blocking Policy

The second part in the development of storage systems consists of writing blocking policies to specify the consistency and durability requirements. In order to to be useful, PADS needs to make it easy to implement a wide range of semantics. This chapter defines what constitutes blocking policy and specifies the primitives PADS provides for writing blocking policy.

5.1 What is Blocking?

The consistency and durability guarantees required by different systems fall on a broad spectrum. Some require weaker guarantees such as eventual consistency [26], whereas others may implement stronger guarantees such as linearizability [94] or sequential consistency [53]. Some systems may differentiate between tentative and committed updates [86] and use different policies for update commitment. For some systems, persistence of an update on a single node is sufficient [69], whereas for others, an update needs to be persistent on all nodes [88].

One approach is to pre-implement standard consistency and durability semantics as libraries and allow designers to pick the one they need. However, since the libraries are pre-defined, this approach does not allow designers to implement customized semantics or take advantage of the knowledge of the system to construct a more efficient implementation.

PADS takes a different approach: it provides designers with sufficient information about the consistency state of local objects so that they can implement their own consistency and durability guarantees. Since the state is automatically maintained by the mechanisms layer, designers do not need to write tricky bookkeeping code.

PADS casts consistency and durability as *blocking policy* because each defines the circumstances under which the processing of a request or the return of a response should block or continue. In particular, enforcing consistency semantics generally requires blocking reads until a sufficient set of updates is reflected in the locally accessible state, blocking writes until the resulting updates make it to some or all of the system's nodes, or both. Similarly, durability policies often require writes to propagate to some subset of nodes (e.g., a central server, a quorum, or an object's "gold" nodes [77]) before the write is considered complete or before the updates are read by another node.

PADS considers these durability and consistency constraints as invariants that must hold when an object is accessed. The system designer specifies these invariants as a set of predicates that block a read request, a write request, or the application of a received update to local state until the predicates are satisfied. To that end, PADS (1) defines 5 *blocking points* for which a system designer specifies predicates, (2) provides 4 *built-in conditions* that a designer can use to specify predicates, and (3) exposes a *B_Action interface* that allows a designer to specify custom conditions based on routing information. The set of conditions for each blocking point makes up the blocking policy of the system.

This small set of primitives is sufficient to specify any order-error or staleness error constraint in Yu and Vahdat's TACT model [94] and implement a broad range of consistency models from best effort coherence to delta coherence [81] to causal consistency [52] to sequential consistency [53] to linearizability [94].

5.2 Blocking Points

PADS defines five points for which a policy can supply a predicate and a timeout value to block a request until the predicate is satisfied or the timeout is reached. The first three are the most important:

- *readNowBlock* blocks a read until it will return data from a moment that satisfies the predicate. Blocking here is useful for ensuring consistency (e.g., *block until a read is guaranteed to return the latest sequenced write.*)
- *writeAfterlock* blocks a write request after it has updated the local object but before it returns. Blocking here is useful for ensuring consistency (e.g., *block until all previous versions of this data are invalidated*) and durability (e.g., *block here until the update is stored at the server.*)

- *applyUpdateBlock* blocks an invalidation received from the network before it is applied to the local data object. Blocking here is useful to increase data availability by allowing a node to continue serving local data, which it might not have been able to if the data had been invalidated. (e.g., *block applying a received invalidation until the corresponding body is received.*)

PADS also defines *writeBeforeBlock* to block a write before it modifies the underlying data object and *readEndBlock* to block a read after it has retrieved data from the data plane but before it returns.

5.3 Blocking Conditions

PADS defines a set of predefined conditions, listed in Table 5.3, to specify predicates at each blocking point. A blocking predicate can use any combination of these conditions. The first four conditions provide an interface to the consistency bookkeeping information maintained in the data plane on each node.

- *isValid* requires that the last body received for an object is as new as the last invalidation received for that object. *isValid* is useful for returning the latest known version of the object and for maximizing availability by ensuring that invalidations received from other nodes are not applied until they can be applied with their corresponding bodies [26, 64].

Predefined Conditions on Local Consistency State	
isValid	Block until node has received the body corresponding to the highest received invalidation for the target object.
isComplete	Block until target object's consistency state reflects all updates before the node's current logical time.
isSequenced *VV|CSN*	Block until object's total order is established via Golding's algorithm (VV) [31] or an explicit commit (CSN) [69].
maxStaleness *nodes, count, t*	Block until all writes up to *(operationStartTime-t)* from *count* nodes in *nodes* have been received.
User Defined Conditions on Local or Distributed State	
B_Action *event-spec*	Block until an event with fields matching *event-spec* is received from routing policy.

Table 5.1: Conditions available for defining blocking predicates.
Note: for the *maxStaleness* condition, setting *nodes* and *count* to -1 represents all nodes in the system.

- *isComplete* requires that a node has received all invalidations for the target object up to the node's current logical time. Since routing policies can direct arbitrary subsets of invalidations to a node, a node may have gaps in its consistency state for some objects. *isComplete* specifies that no such gap exists. If the predicate for *readNowBlock* is set to *isValid* and *isComplete*, reads are guaranteed to see causal consistency.

- *isSequenced* requires that the most recent write to the target object has been assigned a position in a total order. Policies that ensure sequential or stronger consistency can use the *AssignSeq* routing action (see Table 4.3.2) to allow a node to sequence other nodes' writes and specify the *isSequenced* condition as a *readNowBlock* predicate to block reads of unsequenced data. Alternatively, a variation of *isSequenced* can regard a write as sequenced when its logical time is earlier than the latest update a node has received from all other nodes in the system; at that point, no earlier updates can arrive, and a total order is well defined [31].

- *maxStaleness* is useful for bounding real time staleness.

The fifth condition on which a blocking predicate can be based on is *B_Action*. A *B_Action* condition provides an interface with which a routing policy can signal an arbitrary condition to a blocking predicate. An operation waiting for *event-spec* unblocks when the routing rules produce an event whose fields match the specified *event-spec*.

Rationale. The first four, built-in consistency bookkeeping primitives exposed by this API were developed because they are simple and inexpensive to maintain within the data plane [16, 97] but would be complex or expensive to maintain in the control plane. Note that they are primitives, not solutions. For example, to enforce linearizability, one must not only ensure that one reads only sequenced updates (e.g., via blocking at *readNowBlock* on *isSequenced*) but also that a write operation blocks until all prior versions of the object have been invalidated (e.g., via blocking at *writeEndBlock* on, say, the *B_Action allInvalidated* which the routing policy produces by tracking data propagation through the system).

Beyond the four pre-defined conditions, a policy-defined *B_Action* condition is needed for two reasons. The most obvious need is to avoid having to predefine all possible interesting conditions. The other reason for allowing conditions to be met by actions from the event-driven routing policy is that when conditions reflect distributed state, policy designers can exploit knowledge of their systems to produce better solutions than a generic implementation of the same condition.

For example, in the client-server system we describe in Chapter 9, a client blocks a write until it is sure that all other clients caching the object have been invalidated. A generic implementation of the condition might have required the client that issued the write to contact all other clients. However, a policy-defined event can take advantage of the client-server topology for a more efficient implementation. The client sets the *writeEndBlock* predicate to a policy-defined *writeComplete* event. When an object is written, an invalidation is sent to other clients via the server. When other clients receive an invalidation, they send acknowledgements to the server. The server gathers acknowledgements from all other clients and generates a *writeComplete* action for the client that issued the write. In this custom scheme, the client that issued the write only needs to contact the server rather than all other clients.

Conversely, in an earlier version of the interface, we included a built-in predicate *hasPropagated(nodes, count, p)*, which allowed a node to block until its pth most recent write had propagated to at least *count* nodes out of the list of nodes in *nodes*. This condition is potentially useful because it maps directly to Yu and Vahdat's *order error* condition for implementing tunable consistency [94]—it provides a way to wait until a write is durably stored at a quorum of servers. However, our built-in implementation of this primitive had a node poll other nodes directly to determine when a write had reached them, which makes sense for some systems but not others. By constructing such a condition using the *B_Action* primitive instead, a designer can gather the same information in a natural way for a particular system. For example, if the nodes are set up in a star topology, instead of having every node keep track of where its writes have propagated, it is more natural to implement routing rules so that the center node keeps track of update propagation and informs the original writer when the update has propagated to sufficient nodes. The routing policy then generates an appropriate message for the blocking policy, that was waiting on the *B_Action* primitive.

5.4 Examples

This section describes how PADS blocking primitives can be used to implement a wide range of consistency semantics from (weak) monotonic coherence to (strong) linearizability. To implement stronger semantics, the blocking policy makes extensive use of the *B_Action* condition. It relies on the routing policy to ensure that the necessary invariants are satisfied before generating a message for the blocking policy to unblock the operation. We describe below how various semantics can be implemented and the requirements from the routing policy when the *B_Action* condition is specified.

5.4.1 Monotonic-read coherence

Monotonic-read coherence is one of the weakest reasonable consistency semantics for distributed storage. It specifies that if a node reads the value of an object, any successive read operation by that node will always return the same value or a more recent version.

PADS guarantees monotonic-read coherence by default because the mechanisms layer replaces the body of an object only if a newer version is received. By setting the blocking points to the default *true* predicate, read and write operations continue without blocking and coherence is guaranteed.

5.4.2 Delta Coherence

Delta coherence [81] specifies that a read of a data object returns the last value that was produced at most *delta* time units preceding that read operation.

In order to implement delta coherence in PADS, the *readNowBlock* is set to the predicate *maxStaleness(-1,-1, delta) AND isValid*. The *maxStaleness* condition ensures that only updates that have occurred in the past *delta* time units are read. The *isValid* condition ensures that the body read matches the update. Note that if the *isValid* condition is not specified, then there is no guarantee that the body matches the latest invalidation received, and therefore a read may return a much older body, violating delta coherence.

An alternate implementation is to set the *readNowBlock* to *maxStaleness(-*

1, -1, delta) and the *applyUpdateBlock* to *isValid*. By setting the *applyUpdateBlock* to *isValid*, we ensure that all locally stored bodies match the latest invalidations received (i.e. all local objects are valid). However, this implementation may reduce the amount of data that can be read. Invalidations are received in causal order but bodies can be received in any order. If the body of an invalidation is delayed, then the its application and all subsequently received invalidations will be delayed (even though their bodies may have already arrived). By the time the required body arrives and all received updates are applied, it may be too late. Therefore, by setting the *isValid* condition for the *readNowBlock* instead, we prevent application of updates from being delayed and thus increasing the chances of meeting the temporal deadline.

5.4.3 Causal Consistency

Causal consistency defines that writes that potentially are causally related are seen by every node of the system in the same order. Concurrent writes (i.e. ones that are not causally related) may be seen in different order by different nodes [13].

In order to guarantee causal consistency in PADS, the *readNowBlock* is set to *isComplete AND isValid*. The *isComplete* condition guarantees that, for the target object, the locally stored meta-data reflects the latest update in casual order up to the node's current logical time and the *isValid* condition guarantees that the body returned matches that update. If the *isComplete* condition is not used, then it is possible that the local version is causally much older than the versions of other objects stored and reading it would lead to a violation of causal order. Chapter 6 details how causality is maintained in the underlying data layer.

5.4.4 Sequential Consistency

Sequential consistency [53] specifies that the results of any execution is the same as if all operations (reads and writes) were executed in some sequential order and operations of each individual processor appear in this sequence in program order.

Sequential consistency can be implemented in PADS as follows: First, when a write occurs at a node, the write is blocked until all other all other copies of the object have been invalidated. This ensures that future reads cannot return old values that existed before the write. The blocking policy sets the *writeAfterBlock* to *B_Action invalidatedAll* and relies on the routing policy to ensure that all other copies have been invalidated. In a fully-connected and full-replication scenario, the routing policy implements rules that send an ACK to the original writer whenever a node receives an invalidation. When sufficient ACKS have been collected, the routing policy generates an *invalidatedAll* message for the blocking policy. Second, after all the ACKS have been received, the routing policy implements a commit protocol to assign the write a position in the total order. The protocol can use a primary-commit scheme or Golding's algorithm to determine the order. The routing policy uses the *AssignSeq* action to indicate that the write has been assigned a fixed position in the total order. Finally, a read only returns the version of an object if the write that wrote is the last write for that object has been *sequenced*. Note that because invalidation subscriptions transmit the commit information along with invalidations in causal order, reading *valid, complete,* and *sequenced* objects is consistent with the total order determined by the commit protocol. Therefore, the *readNowBlock* is set to *isSequenced AND isValid AND isComplete*. This scheme is satisfies the two requirements (i.e. program order and write atomicity) defined by Adve et. al. [11] for enforcing sequential consistency.

5.4.5 Linearizability

Linearizability guarantees that each operation (read or write) takes effect instantaneously at some point between its invocation and its response [40]. Linearizability is similar to sequential consistency in the sense that all operations appear to have executed in some sequential order. However, linearizability has the additional requirement that the sequential order is consistent with the global order (i.e. real-time order).

In order to implement linearizability, we add an additional condition to the sequential consistency implementation to ensure that the total order is consistent with the real-time write order: the system ensures that only one write occurs at a time and that the invalidation for that write propagates to all other nodes before another write is allowed. This scheme can be implemented by defining a write token that a node needs to posses. The management of the token is carried out by routing rules. Before a write, a node checks if it has the token (i.e. the *writeBeforeBlock* is set to *B_Action getToken*). If it gets the token, it continues with the write. If not, it blocks until receives the token. Like the sequential consistency implementation, the *writeAfterBlock* is set to *B_Action invalidatedAll* to ensure that the write completes only after all nodes have acknowledged that they have received the update. Once the node receives ACKS from all other nodes, it commits the update by using the *AssignSeq* action and gives up the token so that another blocked write can continue. The *readNowBlock* is set to *isSequenced AND isValid AND isComplete* so that only *valid, complete,* and *sequenced* objects are read.

5.4.6 TACT

TACT [94] defines a continuous consistency model based on 3 parameters: *numerical error*, *order error*, and *staleness*. Numerical error defines the maximum number of writes not seen by a replica (i.e. unseen writes). Order error defines the maximum number of writes that have not established their commit order at the local replica (i.e. uncommitted writes). Staleness defines the maximum amount of time before a replica is guaranteed to observe a write accepted by a remote replica (i.e. the maximum staleness of the data observed).

Default conditions. The weakest semantics that TACT guarantees is casual consistency. The default blocking predicates are set follows: the *readNowBlock* is set to *isValid AND isComplete* and the *applyUpdateBlock* is set to *isValid* so that local data is always valid (for availability).

Numerical error. One way to support the numerical error constraint is to block reads if the number unseen writes exceeds the specified bound. The routing policy is defined such that every node maintains information about how many writes have occurred at other nodes that it has not yet received. Whenever a node writes to an object, it informs other nodes of its write. The *writeAfterBlock* predicate is set to *B_Action informedAll* to ensure that the write completes only when other nodes are aware of the write. Also, before reading an object, the node checks with its routing policy to ensure that the number of writes it has not seen does not exceed the numerical error bound. This can be implemented by adding the condition *B_Action okToRead* to the *readNowBlock*. When the routing policy is informed of the read block due to the *B_Action* predicate, it checks to see if the numerical error bound is satisfied. If so, it generates the *okToRead* message. If not, it will establish data flows to ensure ensure that the bound is satisfied before unblocking the read.

Depending on the topology of the system, it might be more efficient for a single primary node to keep track of the number of unseen writes in the system. In that case, whenever a local write occurs or when a remote write is received, the node informs the primary and before a read, the node checks with the primary node to ensure that the read can continue. For this scheme, the blocking conditions still remain the same, but the routing policy is implemented differently.

An alternate way to implement the numerical error constraint is to not accept a write unless it is known for certain that the write will not violate the numerical error bound. In this scheme, the routing policy of every node (e.g., A) maintains an *unseen writes vector* that keeps track of the number of local writes that have not been seen by another node (e.g., B), and a *bound vector* that specifies the upper bound on number of writes that A can accept without B seeing those writes. The blocking policy defines the *writeBeforeBlock* to *B_Action okToWrite* and the *writeAfterBlock* to *B_Action informedAll.* Note, this scheme is equivalent to the implementation mentioned in [43].

Order error. In order to implement the order error bound, writes are blocked if the number of local writes that are uncommitted violate the bound. For this scheme, the routing policy at each node keeps track of how many local writes have not been committed yet. The system can use any commit policy (e.g., a primary-commit scheme or Golding's algorithm [31]) as long as the routing policy is informed of the commit. The blocking policy adds to *readNowBlock* the condition *B_Action okToRead.* It is expected that the routing policy generates the *okToRead* message only if the number of uncommitted writes are within the order error bound. If not, the routing policy initiates data flows so that the bound can satisfied.

Staleness. In order to the staleness bound, the *maxStaleness(-1, -1, bound)* condition is added to the *readNowBlock*. This ensures that reads will be blocked if the current replica is considered to be stale. Note this scheme is similar to the one described in [43].

5.4.7 Two-phase Commit

The two-phase-commit protocol is often employed to commit transactions among a set of nodes. One node is designated as the co-ordinator which initiates the commitment protocol and the other nodes are considered as participants. The protocol is executed in the following steps:

- *Step 1:* At the end of the transaction, the co-ordinator sends a `vote` message to participants indicating that the transaction is ready for commit.
- *Step 2*: On receiving the `vote` message, every participant replies with either a `yes` message or a `no` message depending on whether it can commit the transaction.
- *Step 3:* If the co-ordinator receives a `yes` message from *all* participants, it commits the transaction and sends a `commit` message to the participants. Otherwise, the co-ordinator aborts the transaction and sends an `abort` message instead.
- *Step 4:* On receiving a `commit` or an `abort` message, the participant either commits or aborts the transaction accordingly and sends an `ack`

message to the co-ordinator.

- *Step 5:* After receiving the `ack` messages, the protocol is considered complete (i.e. the transaction is either committed or aborted on all nodes).

In PADS, we implement the two-phase commit protocol for a single write rather than for a transaction. The writer acts as the co-ordinator of the protocol. We assume that invalidation and body subscriptions are established between the writer and all other nodes. The blocking policy is set as follows: the *applyUpdateBlock* is set to *isValid*, the *writeAfterBlock* is set to *B_Action 2PC-Complete* and the *readNowBlock* is set to *isValid and isCommitted*. The routing policy implements the protocol as follows:

- *Step 1:* When a write occurs, the update is sent to other nodes via subscriptions. This update is considered as the `vote` message of the protocol.

- *Step 2:* When the participant receives the update (i.e. the precise invalidation), it sends a `yes` message to the co-ordinator (a.k.a. the co-ordinator). Note that in this implementation, a participant cannot cast a `no` vote.

- *Step 3:* Once the co-ordinator receives the `yes` message from all other nodes, it will commit the write using the *AssignSeq* action. This action will cause the underlying layer to generate a commit invalidation that is sent to all other nodes via subscriptions. The commit invalidation serves as the `commit` message of the protocol. If the co-ordinator does not hear back from all nodes in time, it aborts the write. In PADS, aborting a write is equivalent to not committing a write. In this case, the co-ordinator simply does nothing.

- *Step 4:* On receiving the commit invalidation, the participants send an `ack` message to the co-ordinator. Note that the underlying layer will automatically commit the write when the commit invalidation is received.

- *Step 5:* If the write was aborted in Step 3, the co-ordinator generates a *2PC-complete* message to the blocking layer to unblock the write. Otherwise, the co-ordinator waits for *ack* messages from the participants before generating the *2PC-complete* message for the blocking layer.

5.5 Summary

In PADS, consistency and durability guarantees of a distributed storage system are cast as conditions that must be satisfied before and after access to data. These conditions are considered as part of the blocking policy of the system. For specifying blocking policy, PADS defines five *blocking points* in the data plane's processing of read, write, and receive-update actions. At each blocking point, a designer specifies a *blocking predicate* that indicates when the processing of these actions must block. A blocking predicate can use any combination of the five pre-defined *blocking conditions* and the extensible condition.

Chapter 6

The Mechanisms Layer

In PADS, the mechanisms layer has two main responsibilities: first, it implements the primitives defined by the policy layer, and second, it ensures that implementation is flexible and efficient. As a result, with the low-level details taken care of by the mechanisms, system designers only need to focus on writing higher level policy.

This chapter focuses on the primitives provided by mechanisms layer in order to support the policy API. A more complete discussion of the PRACTI mechanisms layer on which PADS is based is available in Zheng's dissertation [96] as well as in other publication [16] and technical report [97]. We include a discussion of the mechanisms here for context and for completeness.

6.1 Overview

The policy layer has defined primitives that provide designers with sufficient options to build a wide range of systems. The flexibility is useless if the underlying mechanisms layer cannot support these options efficiently. It is, therefore, imperative that the primitives are implemented such that the constructed systems do not pay high costs for generality.

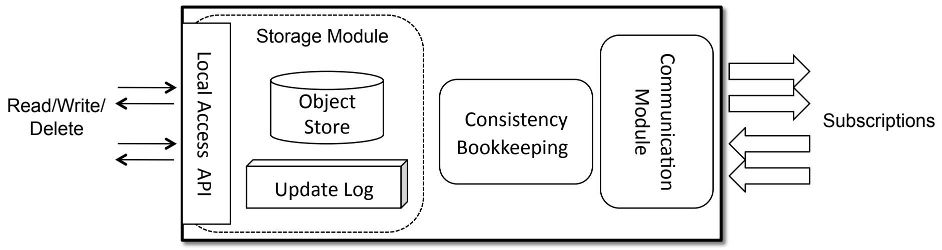

Figure 6.1: Internal structures of the mechanisms layer.

The mechanisms layer must satisfy three requirements:

- For basic operation, it must provide locally accessible data storage for any subsets of data.
- For routing policy specification, it must implement the subscription primitive so that all PR-AC-TI properties can be supported.
- For blocking policy specification, it must maintain sufficient consistency-related book-keeping so that blocking conditions can be supported.

In order to meet the above requirements, as depicted in Figure 6.1, the mechanisms layer consists of the following parts:

- *a storage module* that stores data objects and information about updates locally.
- *a communication module* that implements a flexible synchronization protocol in order to support subscriptions.

- *a consistency bookkeeping module* that keeps track of the consistency state of objects stored locally.

We detail each module in subsequent sections.

6.1.1 Basic Model

Data objects. In PADS, data are stored as objects identified by unique object identifier strings. Sets of objects can be compactly represented as *interest sets* that impose a hierarchical structure on object IDs. For example, the interest set "/a/*:/b" includes object IDs with the prefix "/a/" and also includes the object ID "/b". In addition, an interest set is also used to represent the set of objects a node is interested in replicating locally.

Keeping time. The mechanisms layer heavily relies on Lamport's clocks [52] and version vectors to keep logical time and consistent information. Every node maintains a *time stamp*, $lc@n$ where lc is a logical counter and n the node identifier. To allow events to be causally ordered, the time stamp is incremented whenever a local update occurs and advanced to exceed any observed event whenever a remote update is received. Every node also maintains a version vector, $currentVV$, that indicates all the local or remote updates it is aware of. The $currentVV$ is a representation of the current logical time of the node.

Invalidations and bodies. Whenever an object is updated an invalidation and a body is generated. An invalidation contains the object ID and the logical time of the update. A body contains the actual data of the update in the case of the write. When an object is assigned a sequence or is deleted, only an invalidation is generated.

6.2 Local Storage Module

The storage module stores data and updates in an objects store and an update log. It also provides an API so that applications can access the data stored in a node.

6.2.1 Update Log and Object Store

The *update log* stores local or received invalidations in causal order. In order to prevent the log from becoming arbitrarily large, the node truncates older portions of the log when the log hits a locally configurable size limit and maintains a version vector, $omitVV$, to keep track of the cut-off time.

The *object store* stores latest bodies of objects the node chooses to replicate along with per-object meta-data such as the logical time $writeTimeStamp$, the real time $realTimeStamp$ and additional consistency-related state (refer to Section 6.4) for the last-known write to each byte range. When an object is deleted, the body is deleted from the object store but the delete time, $deleteAS$ is maintained.

Invalidation type	**Components**
Write Invalidation	objId, offset, length, writeTimeStamp
Commit Invalidation	objId, targetTimeStamp, commitTimeStamp
Delete Invalidation	objId, deleteTimeStamp
Imprecise Invalidation	targetSet, startVV, endVV
Body	objId, offset, length, writeTimeStamp, data

Table 6.1: Components of invalidations and bodies

6.2.2 Local Access API

Data stored on a node can be accessed via four methods: read, write, delete, and commit. The *read* method requires the object Id and the byte range, i.e. <*objId, offset, length*>. When a *write* occurs, it is assigned a logical time stamp, *writeTimeStamp*, and a real time stamp, *realTimeStamp*. When an object is *deleted*, a logical time stamp, *deleteTimeStamp* is assigned. The *commit* method takes in an an object ID, and the time stamp, *targetTimeStamp*, of the update it wants to commit. The commit operataion is also assigned a time stamp, *commitTimeStamp*, to indicate the logical time of the commit.

The mechanisms also support multi-object writes. For the ease of discussion, we only consider single objects writes to whole objects instead of specific byte-ranges.

Whenever an object is updated (via a write, delete, or commit), an invalidation and possibly a body is generated. Table 6.1 summarizes the different types of invalidations and bodies and their components.

When a write occurs, a write invalidation and a body are generated.

A write invalidation consists of four fields. The first three fields specify the object and byte-range was updated. The last field, *writeTimeStamp* specifies the time at which the object was updated. A body has an additional field, *data*, that stores the actual contents of the update.

When an object is updated via a commit, a commit invalidation consisting of the object Id, *objId*, the time of the committed write, *targetTimeStamp*, and the commit time, *commitTimeStamp*, is generated. When an object is deleted, a delete invalidation consisting of the object Id, *objId*, and the delete time, *deleteAS*, is generated.

Another type of invalidation is an an *imprecise invalidation* It summarizes all the updates that have occurred to a set of objects, *targetSet*, between a start time, represented by a partial version vector *startVV*, and an end time, another partial version vector *endVV*. Imprecise invalidations are used to transfer update summaries from one node to the other (refer to Section 6.3). On the other hand, write invalidations, commit invalidations and delete invalidations are considered to be *precise invalidations* because they contain the complete meta-data of an update.

6.3 Communication Module

Propagating data and meta-data among nodes is an important design aspect of a distributed storage systems. In PADS, it is considered part of the routing policy. In order to aid the specification of various types of data propagation, the policy API defines *subscription* primitive—a uni-directional stream of updates from one node to another. A subscription is associated with a subscription set, SS, that specifies the set of objects a node is interested in synchronizing and a start time, $startVV$, which specifies that only the updates that have occurred after that time should be sent. It can be seen as an agreement that the sender will send all updates that it is aware of and that have occurred after the start time to the objects in the subscription set.

The communication module, in the mechanisms layer, implements a synchronization protocol that is able to provide all the PR-AC-TI and flexibility properties that subscriptions require and is sufficiently efficient to keep the costs for generality to a minimum. In particular, the protocol

- sends information that is proportional to the required updates on a synchronization stream (i.e. supporting PR)
- allows any node to establish a stream with any other node (i.e. supporting TI)
- sends sufficient information to implement flexible consistency guarantees but that information is kept to a minimum (i.e. supporting AC)
- supports separate invalidation and body subscriptions
- is incremental
- supports both log-based and state-based synchronization
- efficiently supports dynamic subscription establishment between two nodes

Subsequent subsections detail the implementation of the protocol.

6.3.1 Basic Subscriptions

Suppose all the objects stored in Node A lie in the interest set, $A.IS$ and Node A knows about all updates to $A.IS$ up to its current time, $A.currentVV$. Node A wants to receive new updates to $A.IS$ and requests a subscription from node B with the $A.IS$ as subscription set and $A.currentVV$ as the start time.

If node B stores the same objects as A, i.e. $A.IS = B.IS$, synchronization is simple. Node B just sends all the updates that have occurred from $A.currentVV$ to $B.currentVV$ in causal order. A causal steam is used because it provides flexibility for applications to implement a wide range of weaker or stronger consistency guarantees with little additional overhead. Also, a causal stream is incremental, i.e. in case of disconnection, the synchronization can continue where it left off.

A subscription can be seen as having two phases: a *catchup* phase in which the the sender, Node B, sends all updates to objects in the subscription set from the start time, $startVV$, to the sender's $currentVV$, and a *connected* phase in which the sender forwards any new updates it receives until the subscription is removed (i.e. dismantled).

6.3.2 Separate Invalidation and Body Subscriptions.

In order to provide greater flexibility, PADS separates meta-data and data synchronization by having separate invalidation and body subscriptions.

Invalidation subscriptions propagate meta-data of updates that occurred after $startVV$ in *causal order*. They satisfy the prefix property—if an invalidation,I_1, is sent before another invalidation, I_2, then I_1 causally precedes I_2. Body streams are simply *unordered* streams of the bodies of updates that occurred after $startVV$. Ordering of body streams is unnecessary because received bodies are applied to the object store only after their corresponding invalidations are received. The causality of invalidation streams is sufficient to guarantee causal consistency.

In fact, unordered-ness of body subscriptions has its benefits. First, it allows the propagation of high priority updates before other updates improving performance and availability. Second, for frequently updated objects, if multiple bodies to the same object byte-range are sitting in the send queue, only the latest body can be sent and hence reducing bandwidth [64].

An invalidation stream is associated with a subscription set, SS, a start time, $stream.startVV$, and a stream time, $stream.VV$ that keeps track of the logical time progression of the stream. Since a body stream is unordered, it is associated with a subscription set, SS and a start time, $stream.startVV$.

6.3.3 Supporting Flexible Consistency

Say a node wants to receive updates to a subset of data from another node. In addition to the updates, it is necessary to send enough information so that systems can enforce whatever level of consistency they require.

Consider the scenario that an invalidation subscription is established from Node B to Node A for the subscription set $A.IS$. Because of partial replication, however, different nodes may store different subsets of data. In other words, Node B may not store all the objects that Node A wants. In addition to sending all the updates to objects in $A.IS$, Node B must warn Node A if there are any causal gaps – updates that have occurred to objects that Node A cares about but Node B does not have. In this case, Node B sends an *imprecise invalidation*. Imprecise invalidations can be seen as a summary of multiple updates. They summarize updates that occurred to a set of objects, $targetSet$, between a start time, $startVV$, and an end time, $endVV$. Note that start and end time are partial version vectors rather than full version vectors. Note, imprecise invalidations are only sent over invalidation subscriptions and not body subscriptions.

Imprecise invalidation must be sent in two cases. First, the sender, Node B, does not know about the updates the receiver, Node A, has asked for. Second, imprecise invalidations are sent for updates to objects Node A did not ask for so that Node A can pass on the knowledge of the gaps in its information to another node, Node C, that it synchronizes with in the future. It is this propagation of imprecise invalidations allows PADS to simultaneously support partial replication, topology independence, and still support a broad range of consistency semantics.

6.3.4 State-based Synchronization

PADS also supports state-based synchronization by sending a checkpoint of the final state of objects in an interest set instead of sending an ordered log of updates. A checkpoint consists of an imprecise invalidation for $A.IS$ from $A.stream.startVV$ to $B.currentVV$ and the latest meta-data of objects in $A.IS$ updated between $stream.startVV$ and $B.currentVV$.

Log and checkpoint synchronizations have different tradeoffs. The bandwidth requirement for log synchronization is always proportional to the number of updates that occurred to objects in the subscription set. Log synchronization is useful for incrementally and continuously exchanging updates between pairs of nodes. On the other hand, the bandwidth requirement for checkpoint synchronization is proportional to the number of objects updated. Hence, for a small frequently updated subscription set, a checkpoint synchronization might be a better option. Also, a log synchronization is impossible to execute if the update log has been truncated beyond the start time of the subscription. i.e. the subscription requires invalidations that are older than what is currently stored in the log. The only option is to fall back to checkpoint synchronization.

6.3.5 Supporting Efficient Dynamic Subscriptions

Next, what if Node A decides that it wants to synchronize more objects? Say Node A already has an invalidation subscription established for $A.IS$ from Node B and it wants get updates for objects in $A.newIS$ from the time

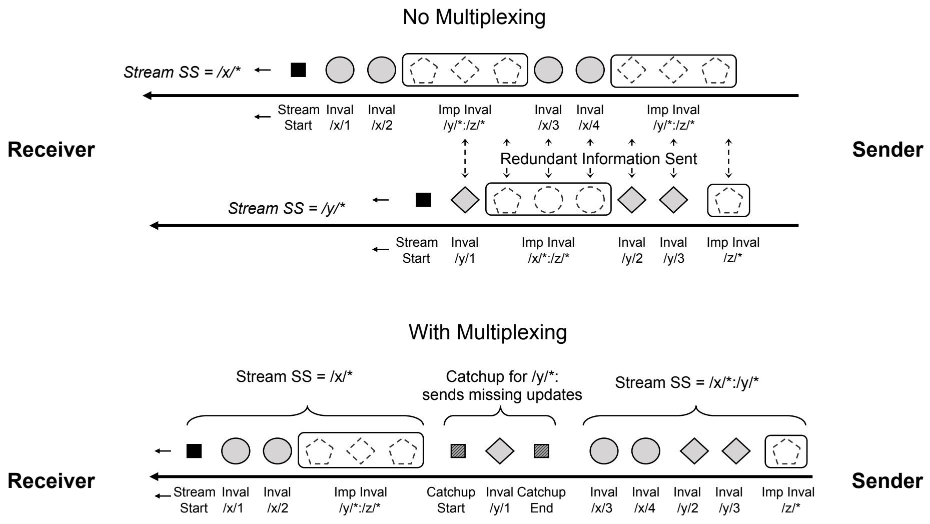

Figure 6.2: Diagram comparing the messages sent on invalidation streams without and with multiplexing.

$A.newIS.startVV$. A simple approach would be to establish a new separate stream for $A.newIS$. The problem is that a lot of redundant information will be sent—in order to ensure that each stream provides a causally ordered series of events from $stream.startVV$, each must include imprecise invalidations for any omitted events and Node A may receive imprecise invalidations for the same updates twice. If Node A creates a large number of subscriptions, then the redundant information sent can be quite substantial.

PADS reduces the overheads of dynamic synchronization requests by multiplexing subscription requests on to a single stream. Since updates on the stream are sent in causal order, simply sending updates to $A.newIS$ from the current logical time of the stream $stream.VV$ will not work because that would not “fix the gap” that Node A has for *A.newIS* from *A.newIS.startVV* to *stream.VV*. Instead, Node B “pauses” the stream at *stream.VV* and sends a *catchup stream* which includes all updates for *A.newIS* from *A.newIS.startVV* to *stream.VV* that node B knows about. After catchup, node B continues sending invalidations for *A.IS* and *A.newIS* and gap markers for everything else starting from *stream.VV*. Figure 6.2 illustrates the multiplexing of the stream.

6.3.6 Summary

The following list highlights the features that enable subscriptions to be flexible and efficient.

- Because subscriptions are peer-to-peer streams, they are able to support arbitrary synchronization topologies.
- The synchronization set for each subscription is not pre-defined. Hence, nodes can synchronize arbitrary subsets of data.
- By separating invalidation and body subscriptions, PADS provides the flexibility to propagate meta-data and data via separate paths.
- The causal propagation of updates provides systems with the flexibility to build a wide range of consistency semantics and allows synchronization to make incremental progress between failures.
- By supporting both log-based and state-based synchronization, subscriptions allows systems to make appropriate performance tradeoffs.

- By summarizing unwanted meta-data, PADS ensures that updates are propagated with minimal overhead while supporting partial replication and guaranteeing causal consistency.
- By multiplexing subscription requests over a single stream, PADS ensures that dynamic requests for synchronizations are efficiently handled.

6.4 Consistency Bookkeeping Module

The responsibility of the bookkeeping module is to maintain sufficient local state in order to support the primitives (i.e. blocking conditions) defined by the blocking policy API. The advantage of maintaining the state in the mechanisms layer is that it relives designers from writing tricky bookkeeping code. Also, since the state maintained is general, designers can quickly adapt the system if consistency requirements change.

In this section, we discuss the state maintained by the module and how it is updated as messages are received.

6.4.1 Dimensions of Local Consistency State

In PADS, every node keeps track of the consistency-related state of the objects it stores. Consistency has three dimensions:

- *Preciseness:* The interest set is *precise* if the node has received all the precise invalidations affecting the objects in the interest set up to the current logical time, *currentVV*. An interest set is *imprecise*, if the node has received imprecise invalidations for any subset the interest set. An object is precise if it lies in a precise interest set

- *Validity:* An object is *valid* if object store contains the body of the last received write invalidation, else it is *invalid*.

- *Commit:* An object is *committed* if the commit operation or a commit invalidation is received for the last-known write update.

In fact, different consistency semantics can be guaranteed by reading objects in specific states. For example, causal consistency can be easily guaranteed by accessing only precise and valid objects. Preciseness guarantees that the node is aware of the latest causal update to the object and validity implies that the node is storing the body of that update. On the other hand, systems that do not require causal consistency have the option of accessing data even from imprecise interest sets.

6.4.2 Local Consistency State

In order to keep track of the consistency of local objects, a node, Node A, maintains the following state:

- $currentVV$: Node A maintains a version vector that indicates the latest update it has seen, either due to a precise invalidation or to an imprecise invalidation. This implies that Node A is *not* aware of any updates after $currentVV$.

- *currRealVV*: Node A also maintains a vector of real timestamps rather than logical timestamps of the updates it has received. *currRealVV* can be seen as the real-time equivalent of *currentVV*.

- *stream.VV*: For every invalidation stream, Node A maintains a logical time that includes the last update and all the causally preceding updates received on the stream. It implies that Node A has seen, either via precise or imprecise invalidations, all events from the *stream.startVV* to *stream.VV*.

- *IS.noGapVV*: For every interest set, Node A maintains a *noGapVV* that indicates that Node A has seen all updates and no imprecise invalidations to the interest set until this time. For a particular interest set IS_1, if IS_1.*noGapVV* $<$ *currentVV* then the interest set is considered *imprecise* – Node A is missing one or more invalidations that affect IS_1 between IS_1.*noGapVV* and *currentVV*. Hence, causal consistency cannot be assured for reads of objects in IS_1.

- *obj.writeTimeStamp*: For every object currently stored, Node A stores the logical timestamp of the *latest* write invalidation it has received for the object.

- *obj.realTimeStamp*: For every object currently stored, Node A stores the real timestamp of the *latest* write invalidation it has received for the object.

Blocking Condition	**Consistency Module Equivalent**
isValid	obj.isValid = true
isComplete	IS.precise = true
isSequenced	obj.isCommited = true
maxStaleness *nodes count t*	*count* values corresponding to *nodes* in *currRealVV* are less than *t* milliseconds old.

Table 6.2: Mapping between blocking conditions and local consistency state maintained. *obj* represents the object for which the condition is specified and *IS* represents the interest set enclosing the object.

- *obj.isValid*: A flag, stored for every object, that indicates whether the node stores the body of the last received write invalidation for the object. If the *isValid* flag is not set, the object is considered to be *invalid*. Causal consistency cannot be assured for a read of that object, because the body is older than the last invalidation received.

- *obj.isCommitted*: A flag, stored for every object, that indicates whether the node has received a commit invalidation for the last received write invalidation it has received for the object. If the *isCommitted* flag is set, the object is considered to be *committed*.

Note that object-related consistency state is stored in the object store and interest set-related state is stored in a hierarchical structure called *ISSTATUS*.

The state maintained at the module is sufficient to implement the blocking conditions defined by the blocking policy API. Table 6.4.2 depicts how blocking conditions can be implemented.

6.4.3 Updating the State

When a receiver receives messages on an invalidation or body stream, in addition to updating the log and the store, the key job for the receiver is to make sure that the consistency state is correctly updated. We describe below how incoming invalidation streams and body streams are processed.

Processing incoming invalidation streams. In order to eliminate the need for updating *IS.noGapVV* every time an invalidation is received, we introduce the concept of "attaching" an interest set to a stream. An interest set is "attached" to a stream if no imprecise invalidations for the interest set have been received on the stream, i.e. *IS.noGapVV* includes *stream.VV*. When an invalidation is received, only *stream.VV* needs to be updated. The node also keeps track of which streams an interest set is attached to by maintaining a *IS.attachedStreams* set. If an imprecise invalidation, *IMP*, is received on a stream, the *IMP.targetSet* is "detached" from the stream by explicitly by storing its *noGapVV* for the interest set in *ISSTATUS*. The details of how an invalidation stream is processed as described in Figure 6.3 and 6.4.

Processing incoming body streams. Processing a body stream is simple. When a node receives a body, it will check if the *body.writeTimeStamp* matches local times stamp for the object, *obj.writeTimeStamp*. If there is a match, it implies that the body corresponds to the latest received invalidation for the object, the body is applied the store and the *obj.isValid* flag is set to

```
if received message is a subscription start message, $SubStart$ then
  //set up subscription stream:
  $stream.SS \Leftarrow subStart.SS$
  $stream.VV \Leftarrow subStart.VV$
else if received message is an write invalidation, $I$ then
  //update log, timing state and per object state:
  store $I$ in update log
  update $stream.VV$ and $currentVV$ to include $I.writeTimeStamp$.
  update $currRealVV$ to include $I.realTimeStamp$.
  $obj.writeTimeStamp \Leftarrow I.writeTimeStamp$
  $obj.realTimeStamp \Leftarrow I.realTimeStamp$
  $obj.isValid \Leftarrow false$
  $obj.isCommitted \Leftarrow false$
else if received message is a commit invalidation, $CI$ then
  //update log, timing state and per object state:
  store $CI$ in update log
  update $stream.VV$ and $currentVV$ to include $CI.commitTimeStamp$.
  update $currRealVV$ to include $CI.realTimeStamp$.
  //if target timestamp matches current timestamp:
  // update commit flag:
  if $obj.writeTimeStamp$ equals $CI.targetTimeStamp$ then
    $obj.isCommitted \Leftarrow true$
  end if
else if received message is a delete invalidation, $DI$ then
  //update log, timing state and per object state:
  store $DI$ in update log
  update $stream.VV$ and $currentVV$ to include $DI.deleteTimeStamp$.
  update $currRealVV$ to include $DI.realTimeStamp$.
  $obj.deleteTimeStamp \Leftarrow DI.deleteTimeStamp$
else if received message is an imprecise invalidation, $IM$ then
  //update log, timing state and interest set state:
  store $IM$ in update log
  update $stream.VV$ and $currentVV$ to include $IM.endVV$.
  update $currRealVV$ to include $IM.realEndVV$.
  check for intersecting set
  $IIS \Leftarrow stream.SS \cap IM.targetSet$
  if $IIS \neq \emptyset$ then
    // detach IIS from the stream
    $IIS.noGapVV \Leftarrow min(IIS.noGapVV, IM.startVV - 1)$
    $stream.SS \Leftarrow stream.SS \setminus IIS$
    remove $stream$ from $IIS.attachedStreams$
  end if
end if
```

Figure 6.3: Pseudocode for processing received invalidations.

```
if received message is a checkpoint, $CP$ then
  //apply received meta-data to local structures
  for all $IS$ in $CP$ do
    update local $IS.noGapVV$ to include $CP.IS.noGapVV$
  end for
  for all object metadata in $CP$ do
    if $CP.obj.metadata$ is newer than $local.obj.metada$ then
      update $local.obj.metadata$ to include $CP.obj.metadata$
    end if
  end for
else if received message is a catchup start message, $CStart$ then
  //switch to catchup mode
  $stream.pendingSS \Leftarrow Cstart.SS$
  $stream.pendingVV \Leftarrow Cstart.VV$
  for all invalidation or gap markers received do
    process as above, except, update $pendingVV$ instead of $stream.VV$
  end for
else if received messages is a catchup end message, $CEnd$ then
  //switch to normal mode
  if $stream.pendingVV$ equals or includes $stream.VV$ then
    //attach stream.pendingSS to the stream.
    $stream.SS \Leftarrow stream.SS \cup stream.pendingSS$
    add $stream$ to $consistencyModule.pendingSS.attachedStreams$
  end if
end if
```

Figure 6.4: Pseudocode for processing other received messages on invalidation streams.

true. If the body is older than the timestamp, then the body is discarded. If the body is newer than the timestamp, it implies that its corresponding invalidation has not been received yet. Instead of discarding it, the body is stored in a body buffer and is applied to the store when its corresponding invalidation arrives.

6.5 Conflict Detection

Conflict detection is an important feature for distributed storage systems. An object may be independently updated on multiple nodes leading to diverging versions. Updates to the same byte of the same object are considered to be conflicting if there is no causal relationship between them. Such conflicts need to be detected so that appropriate resolution, either automatic or manual, can be invoked to resolve the differences and achieve eventual consistency [49] [86].

Existing synchronization protocols [16, 33, 45, 59] often store or transmit extra information for the sole purpose of conflict detection. For mobile environments where network bandwidth and storage capacity can be limited, it is important that such information be minimal. Fortunately, the mechanisms layer in PADS can detect conflicts without having to store or transmit any extra information – it simply derives the information it needs from the state already maintained for consistency.

Conflict detection is carried out as described below. If no conflict is detected, the received update is applied; otherwise the conflict flag set and all the information is stored in a special file for resolution. PADS provides mechanisms for conflict detection, but it leaves conflict resolution to system specific policies. For convenience, PADS provides a default last-writer-wins policy.

Dependency summary vectors. PADS uses a dependency summary vector (DSV) scheme for conflict detection. A dependency summary vector (DSV) is a vector associated with an update that summarizes all the causally preceding updates to the object being updated.

In particular, a DSV of a update U,

- includes the timestamp of all causally preceding updates to the object.
- may include the timestamp of the current update, U.
- may include the timestamps of updates to other objects.
- excludes any updates that are causally ordered after U.

Note that, there is not necessarily a unique DSV for a single update. For example, suppose all the causally ordered updates on an object are $(1@A), (3@A), (10@B)$. Two legal DSVs for the the second update $(3@A)$ are $< 1@A, 9@B >$ and $< 2@A, 6@B >$ but not $< 0@A, 9@B >$ or $< 3@A, 10@B >$ because the former does not include the first update and the latter does not exclude the third update.

Conflict detection simply requires comparing the write times and the DSVs for two updates. In order to detect whether two different updates U_1 and U_2 to the same object conflict, we carry out the following comparisons: If $U_1.ts$ is included in $U_2.dsv$, then U_1 causally precedes U_2, by definition. Similarly, if $U_2.ts$ is included in $U_2.dsv$, then U_2 causally precedes U_1. Otherwise, U_1 and U_2 are marked as conflicts.

Deriving DSVs. It would be inefficient to transmit a DSV with each update and store a DSV with each object. PADS therefore derives DSVs from the meta-data already maintained by the synchronization protocol. In order to do that, it ensures that a node is aware of all the previous updates to the object before it is updated. Any new update (a local write or a received invalidation) can only be applied if there is no gap in the object update information (i.e. the enclosing interest set is precise).

By definition *noGapVV* of an interest set covers all the causally preceding updates to the objects in the interest up to that time. Hence, for an object in the interest set, *noGapVV* includes all the causally preceding updates to that object. If the interest set is precise, then the *noGapVV* and the DSV are equal to *currentVV*.

To determine the DSVs of received invalidations, PADS takes advantage of the causal property of the stream: For a received invalidation, all the causally preceding updates have been already received, and any newer updates will not arrive before the current received invalidation. *streamVV* includes all the current and all causally preceding updates. The DSV for invalidations in connected phase is $streamVV$. For invalidations received during log synchronization, the DSV is $pendingVV$, and for updates received via a checkpoint, the DSV is the received $noGapVV$.

Detecting conflicts. PADS detects conflicts by comparing the timestamp and the DSV of the received invalidation with the locally stored object timestamp and the $noGapVV$ of the enclosing interest set. Conflicting updates are flagged and logged for the application or the user to handle.

6.6 Summary

The PADS mechanisms layer provides a set of common primitives so that designers can simply invoke these mechanisms to construct systems without having to spend the effort to reimplement them. In particular, the mechanisms layer implements (1) an object store and an update log for storing and accessing data objects locally, (2) subscriptions for flexible and efficient update propagation between nodes, (3) bookkeeping of consistency state that can be used to implement various consistency semantics, and (4) a conflict detection scheme so that concurrent updates can be eventually detected and resolved without additional overhead.

Chapter 7

Bridging the Gap

The mechanisms layer implements the set of primitives required by PADS policy layer. Unfortunately, the mechanisms API does not match the policy API. This section describes how that gap is bridged. In particular, it briefly describes the API exposed by the mechanisms layer, provides reasons for why development on the raw API is difficult, and presents the modules implemented to transform the raw API to routing and blocking policy API.

7.1 Mechanisms API

The mechanisms layer implements sufficient primitives that a distributed storage system can be directly built over it. It provides an extensive set of API that consists of a set of 10 actions (listed in Table 7.1) that can be used to invoke mechanisms primitives, 37 triggers (listed in Table 7.2) that provide information about events or messages received by the mechanisms layer, and 6 flags at the read/write interface (listed in Table 7.3) to specify the consistency semantics. In order to build a distributed storage system over the raw interface, a system designer specifies the system design by setting flags for consistency at there read and write interface and defining a *Controller* that

Type	**List of Actions**
Subscription Related	subscribeInval subscribeBody subscribeCheckpoint removeSubscribeInval removeSubscribeBody removeConnection
Other Commands	issueDemandRead requestSync getCurrentVV unbind

Table 7.1: Actions available to policy

implements methods to handle the received triggers.

Unfortunately, it is difficult to build a storage system with the interface exposed for five reasons:

First, the API exposes too many abstractions. The API exposes at least 6 types of streams that send different types of information. For example, bodies are sent on a body stream whereas information about node state is sent over a sync-reply stream. Also, the API differentiates between streams and subscriptions—a subscription is instantiated over a stream which may be serving multiple subscriptions. On the contrary, the PADS routing API only exposes the single abstraction of a subscription keeping things simple.

Second, the difference between the triggers received is subtle and requires an understanding of the implementation of the underlying layer. Consider the two triggers *informInvalStreamInitiated* and *informSubscribeInval-Succeeded*. The first one indicates that an invalidation stream was initiated

Type	List of Triggers
Stream Initialization	informInvalStreamInitiated informCommitStreamInitiated informBodyStreamInitiated informUnbindStreamInitiated informSyncRplyStreamInitiated informCheckpointStreamInitiated informOutgoingInvalStreamInitiated informOutgoingBodyStreamInitiated informOutgoingCheckpointStreamInitiated
Stream Termination	informInvalStreamTerminated informCommitStreamTerminated informBodyStreamTerminated informUnbindStreamTerminated informSyncRplyStreamTerminated informCheckpointStreamTerminated informOutgoingInvalStreamTerminated informOutgoingBodyStreamTerminated informOutgoingCheckpointStreamTerminated
Message Received	recvSyncReply informReceiveInval informReceivePushBody informReceiveDemandReply informCheckpointStreamReceiveStatus
Local Events	informLocalWrite informLocalDelete informLocalReadImprecise informLocalReadInvalid
Demand Read Related	informDemandReadMiss informDemandReadHit informDemandImprecise
Consistency Related	informBecameImprecise informBecamePrecise
Invalidation Subscription Related	informGapExistForSubscribeInv informSubscribeInvalFailed informSubscribeInvalSucceeded informOutgoingSubscribeInvalInitiated informOutgoingSubscribeInvalTerminated
Body Subscription Related	informSubscribeBodySucceeded informSubscribeBodyRemoved informOutgoingSubscribeBodyInitiated informOutgoingSubscribeBodyTerminated

Table 7.2: Triggers sent from the underlying layer to the controller.

Operation	**Available Flags**
Read	blockInvalid, blockImprecise, maxTemporalError
Write	priority, bound, targetOrderError

Table 7.3: Flags available for specifying consistency

between two nodes and the second one indicates that an invalidation subscription was successfully established on a stream. If a designer is not careful, it is is easy to confuse the two triggers. Another pair of triggers that can be easily mixed up is the *informReceivePushBody* and *informReceiveDemandReply* triggers. Both indicate that a single body was received. However, the first one implies that the sender of the body initiated the transfer and the second one implies that the body was sent in response to a request. The PADS API combines the two triggers to a single *sendBodySuccess* trigger reducing opportunities for misuse.

Third, the level of detail exposed by some triggers is often not required in practice. For example, the triggers *informDemandReadMiss* and *informDemandReadImprecise* indicate that a request for a body from another node could not be satisfied because the local copy was either invalid (for the first trigger) or imprecise (for the second trigger). In all the systems we constructed with PADS, we did not need this level of detail—it was sufficient to know that a remote body request could not be locally satisfied but the reason was not necessary.

Fourth, despite its size, the API is missing some important triggers and consistency flags. Eventhough write operations expose a flag to set the

TACT order error constraint, there is no trigger that informs policy when the constraint is not satisfied. Also, the default range of consistency semantics supported is limited because the API only supports two of the three TACT parameters (it does not support the numerical error constraint).

Fifth, and most importantly, the API puts a lot of burden on the developers. Because the API does not impose a structure on system specification (i.e. no separation of policy into routing and blocking), programmers are left to decipher the different parts of system design (i.e. consistency, durability, and information propagation) and the pick the right methods in the large API to implement their systems. The API also does not make it easy to write a customized consistency implementation—a designer would need to explicit implement a wrapper that blocks read/write operations, sends a new trigger to the controller, receives messages from the controller, and unblocks the operation when appropriate.

7.2 Making it Easier to Use

For ease of development, we convert the raw API to PADS policy API. In particular,

- we expose a much smaller API to developers. We trim excessive methods and add methods to make the API more complete.
- we introduce a separation of concerns by splitting policy specification into blocking and routing, making systems easier to design and imple-

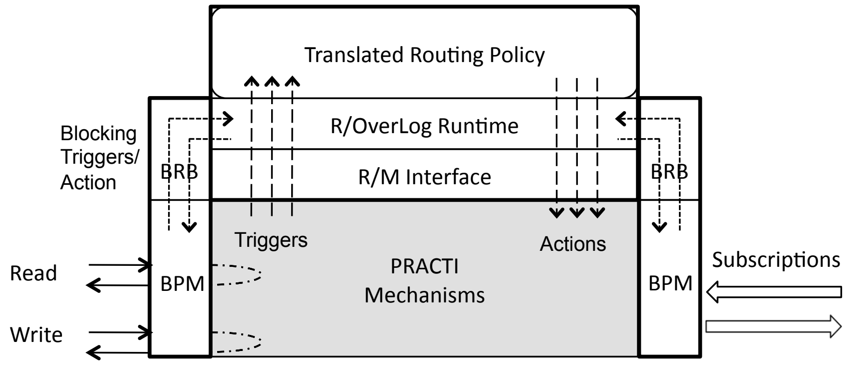

Figure 7.1: Bridging the mechanisms and policy layers.
BPM stands for BlockingPolicyModule and BRB stands for BlockingRoutingBridge.

ment.

- we implement common tasks such as retries on subscription establishment failure and communication between routing and blocking policies so that designers need not reimplement them.

In order to support the PADS API, we implement four modules, as illustrated by Figure 7.1: (1) the Routing/Mechanisms (R/M) interface that translates the API exposed by the mechanisms layer to routing policy API, (2) the R/OverLog runtime that translates R/OverLog policies into Java and executes them over the mechanisms, (3) the BlockingPolicyModule (BPM) that acts as a wrapper layer to intercept reads, writes, and update applications to support blocking points and expose the pre-defined blocking conditions, and (4) the BlockingRoutingBridge (BRM) that allows routing policies and blocking policies to communicate with each other. We provide details of each module in subsequent subsections.

7.2.1 R/M Interface

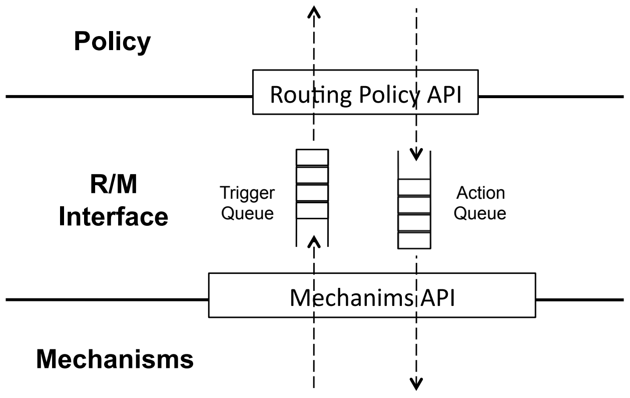

Figure 7.2: Internal queues in the R/M Interface

The R/M interface is the middle layer between the policy and mechanisms layers and serves three purposes: to translate the mechanisms API to the concise PADS routing API, to prevent policies from implementing deadlocks, and to implement common tasks.

Translating the API. As described earlier, the API exposed by the mechanisms layer is difficult to program on. The R/M interface translates the API so that developers have a small, intuitive API to work with. It derives the new API as follows:

- by reducing the abstractions exposed. Only a single abstraction for update flows is exposed—subscriptions. Designers only take care of body and invalidation subscriptions. They do not need to worry about streams or multiple subscriptions on a single stream.

- by hiding triggers or actions that are excessive, redundant, or do not fit in the PADS model. For example, we hide triggers pertaining to streams because we do not expose streams to designers. We also hide the actions and triggers that get information about another node's state (i.e. *requestSync* action and *recvSyncReply* triggers), because in PADS, this functionality is part of meta-data propagation and should be implemented by the routing rules rather than the underlying layer. Triggers that are related to consistency of local objects (i.e. *informBecamePrecise* and *informBecameImprecise*) are hidden from the routing policy layer, because PADS uses this information for blocking policy. Also, since the PADS model cleanly separates data propagation and meta-data propagation paths, the primitives for binding bodies together with invalidations for propagation are hidden (i.e. no bound writes and no *unbind* action).

- by combining similar triggers and in some cases, making them more general. For example, we combine the read miss triggers (*informLocalReadImprecise* and *informLocalReadInvalid*) to a single *operationBlock* trigger which not only informs the routing policy of read misses but also of write blocks. We combine the *informReceivePushBody* and *informReceiveDemandReply* triggers into a single *sendBodySuccess* trigger.

Even though the API exposed to the routing policy is much smaller than that provided by the mechanisms layer, we find that the routing API provides sufficient control to policy writers to specify system designs.

Preventing deadlocks. The R/M interface adds a level of indirection in the threading model to prevent deadlocks. Without the interface, a thread that is carrying out some important task in the mechanisms layer could get transferred to the policy layer via a trigger and then come back down to the mechanisms layer via an action without having finished the task it was originally performing. This situation can cause blocking or deadlocks. The R/M interface prevents deadlocks by cleanly separating policy threads from mechanisms threads. Whenever a mechanisms thread needs to send a trigger to policy, it puts a trigger object in an R/M interface queue and returns to its processing. A separate R/M thread retrieves the trigger in the queue and calls into the policy layer. This separation allows mechanisms threads to asynchronously send triggers to the policy. Policy actions are also passed to the mechanisms in a similar fashion. Figure 7.2 depicts the internal queues of the R/M interface.

Implementing common tasks. The R/M interface implements functionality for common tasks. For example, if there was a problem establishing a subscription, the R/M interface will automatically retry the subscription for 2 times after every 3 seconds before reporting failure. The number of retries and the time between each retry can be specified by the policy writer.

7.2.2 R/OverLog Runtime

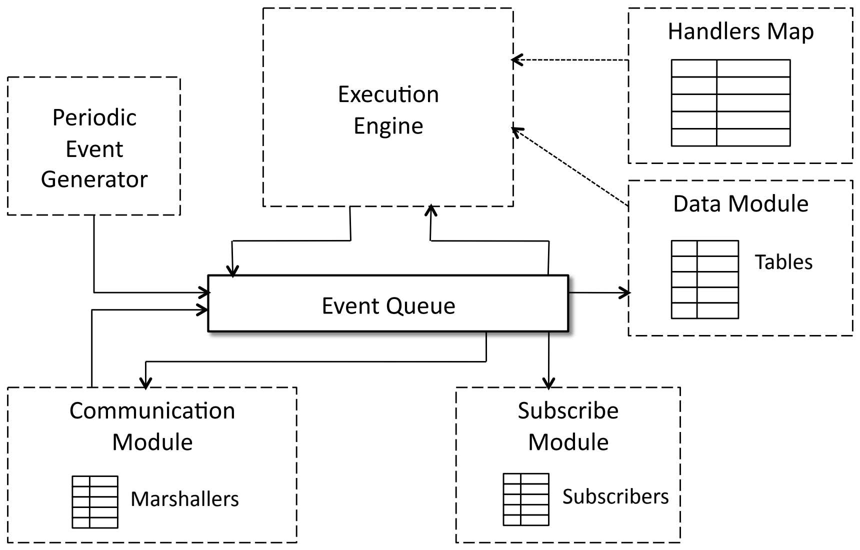

Figure 7.3: Major components of the R/OverLog Runtime.
The solid arrows represent event or table update flows and the dashed arrows represent handler or table lookups.

The R/OverLog runtime is a Java-based engine to run translated R/OverLog policies. As Figure 7.3 depicts, major parts of the runtime include:

- *Event Queue*: keeps events that have been generated or received but not yet been handled.
- *Periodic Event Generator*: generates periodic events.
- *Handler Map*: keeps track of the handlers associated with every event. A handler contains the logic of a rule that is fired by the event.

- *Communication Module*: responsible for sending and receiving remote events. The communication module also maintains a list of marshallers for serializing and deserializing events. When an remote event is received, it is put into the event queue.

- *Data Module*: responsible for storing the R/OverLog state as tables. Every table has interfaces for inserting tuples, removing tuples, and carrying out simple aggregation.

- *Subscribe Module*: allows an external source to listen for a particular event type by registering a *subscriber* for it. Whenever an event of the registered type is generated, the associated subscriber is invoked.

R/OverLog to Java Translation. The translation from R/OverLog to Java is done using the XTC toolkit [34, 41]—a framework for building extensible source-to-source translators and compilers. The translation is carried as follows:

- For every event or tuple, the translator generates a tuple class that contains fields and marshalling code.

- For every rule, the translator generates a handler class that implements the logic of the rule (i.e. table lookups, assignments, and constraints) to generate either another event or a table update.

- For a single R/OverLog program, the translator generates a main OLG class that initializes the tables, registers periodic events with the Periodic

Event Generator, registers print subscribers for *watch* statements, and for every event, registers the appropriate handlers and marshallers with the Handler Map and Communication Module respectively.

The execution of the program can be started by invoking the *start* method in the generated main OLG class.

Program execution. The main execution thread dequeues an event from the event queue, looks up the handler table and executes all the handlers associated with that event. Any new events generated as a result of the execution are put back in the event queue. The event queue checks if the events are local or remote. Local events are put to the back of the queue whereas remote events are handed over the communication module for transfer. Any table updates are handed over to the data module.

Fixed point semantics. The semantics guarantees that all rules triggered by the appearance of the same event are executed atomically in isolation from one another. Once all such rules are executed, their table updates are applied, their actions are invoked, and the events they produce are enqueued for future execution.

In order to support fixed point semantics, the event queue is implemented internally as two queues: an external queue that holds events belonging to different fixed points and an internal queue that holds event corresponding to the current fixed point. Events received from remote nodes and periodic events are put in the external queue.

The execution thread will pick an event from the internal queue, unless it is empty, and invoke the handlers associated with the event. All new local events generated from the handling of that event are put in the internal queue. Table updates and remote events are buffered. When the internal queue becomes empty (i.e. the end of the fixed-point is reached), table updates are committed and the remote events are sent. The next time the execution thread needs an event, it is picked from the external queue starting a new fixed point.

7.2.3 BlockingPolicyModule (BPM)

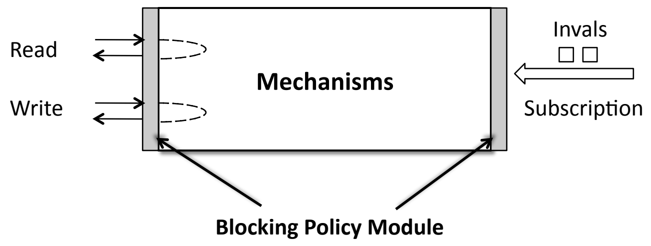

Figure 7.4: Blocking Policy Module intercepting operations.

The blocking policy API consists of 5 blocking points that correspond to access points and a list of conditions that can be specified for each access point. On the other hand, the mechanisms API only provides blocking flags for reads (equivalent to the *readNowBlock*) and an order error bound for writes (equivalent to the *writeBeforeBlock*). The gap between the two layers is

Flags at Blocking Points
isValid
isComplete
isSequenced
MaxStaleness

Table 7.4: Flags provided by the BlockingPolicyModule.

```
public class MyCausalBPM extends BlockingPolicyModule {
    readNowBlock.isValid = true;
    readNowBlock.isComplete = true;
    ApplyUpdateBlock.isValid = true;
}
```

Figure 7.5: Blocking policy implementation for causal consistency.

bridged by the *BlockingPolicyModule*—a wrapper layer, as depicted in Figure 7.4, that intercepts reads and writes before and after access to local state and received invalidations before they are applied locally. Each interception point corresponds to a blocking point.

The built-in blocking conditions are provided as flags (refer to Table 7.4). By default, the flags are set to false so no operation is blocked. Most flags are set to either *true* or *false* values. For the *maxStaleness* condition, the designer provides a list of *nodes*, a *count* and a *value* for the maximum staleness allowed. For example, as illustrated by Figure 7.5, the blocking policy for a causal consistency can be implemented by setting flags such that only valid and complete objects are read and application of invalidations is delayed until the corresponding body has arrived.

If an operation blocks due to built-in conditions, the BlockingPolicy-

Module will inform the routing policy of the block via *operationBlocked* triggers for each condition that is not satisfied. When all the conditions are satisfied, the operation will automatically unblock.

For custom conditions, the BlockingPolicyModule provides abstract methods for each point that a designer can extend and methods to look up consistency state for each object.

7.2.4 BlockingRoutingBridge (BRB)

Some systems implement custom blocking conditions based on routing conditions (i.e. the *B_ACT* condition) which requires the blocking policy to send and receive messages from routing policy, the BlockingPolicyModule provides a *BlockingRoutingBridge.* that allows a blocking policy to send a trigger to the routing policy and receive events from it. Triggers are inserted in the routing runtime's event queue. In order to receive an event, the blocking policy first needs to register the *eventName* for the routing event it wants to listen to. When an *eventName* is registered, the bridge sets up a producer-consumer mailbox for the received events that match that *eventName.* Whenever an event that matches a registered *eventName* is generated in the routing runtime, it is put in the mailbox. To receive a routing event, the blocking policy calls the *waitFor* method with the *eventName* and a *eventSpec.* The call will cause the policy to wait until a matching event is received from the routing policy, unless it is already available in the mailbox.

For example, for the simple client-server system described in Section 9.3.1,

```
public class SCSPolicy extends BlockingPolicyModule {

  //built-in conditions
  readNowBlock.isValid = true;
  readNowBlock.isComplete = true;
  readNowBlock.isSequenced = true;
  applyUpdateBlock.isValid = true;

  // initialization
   public SCSPolicy() {
       // register the events to listen to
       bridge.registerMsgToListen("writeComplete");
   }

  // custom write after block
  public void writeAfterBlock(ObjId objId, long offset,
                              long length, AcceptStamp as) {

        // inform routing policy of a block
        Object[] msgFeilds = {objId.toString, offset, length,
                           "writeAfter", "custom"};
        bridge.sendMsgToRouting("operationBlocked", msgFeilds);


      // create the required event spec
      Object[] spec = {objId.toString()};

      // wait for a writeComplete message  for the object
      bridge.waitFor("writeComplete", spec);
  }
 }

```

Figure 7.6: Blocking policy implementation for the simple client-server system (refer to Section 9.3.1)

the blocking policy specifies the *B_ACT writeComplete* condition for the *write-AfterBlock*. As Figure 7.6 illustrates, the implementation of this condition involves four steps: First, the blocking policy registers the event it wants to listen to (line 12). Second, the designer extends the appropriate method of the BlockingPolicyModule (line 16). Third, in the method implementation, the routing policy is informed that the operation is blocked (line 22). Finally, the method waits until the required routing event is received (line 29).

In fact, the bridge allows designers to specify custom event matchers, that determines how received events are matched with the provided event spec. Custom matchers are useful to implement optimizations. For example, on a client, each multiple outstanding write may be waiting for an acknowledgement from the server to ensure that the server has received that particular write. However, because invalidation streams are causal, when a server acknowledges a particular write, it is safe to assume that the server has already received all the previous writes from the client. When the server sends an ack for the last-received write, the default matcher will only unblock that particular write. By customizing the event matcher, all previous outstanding writes can be unblocked too.

7.3 Summary

There is a need to bridge the gap between the API expected by the policy layers and that provided by the mechanisms layer. This section provides details of the glue. In particular, it describes (1) the R/M Interface that allows deadlock-free translation between the mechanisms and the routing API, (2) the R/OverLog runtime that translates R/OverLog to Java and executes it, (3) the BlockingPolicyModule that intercepts access to local state at the blocking points and exposes blocking conditions as flags for each point, and (4) the BlockingRoutingBridge that handles communication between the routing and blocking policies for routing-based conditions.

Chapter 8

Evaluation Part 1: Microbenchmarks

Other than the ease of development, PADS's usefulness also depends on the performance of systems built with it. The systems should not have to pay a high cost for PADS's generality. This chapter examines the overheads associated with PADS. Evaluation of systems built with PADS is discussed in Chapter 9. To aid our evaluation, we consider three criteria

- *Synchronization overhead*: The overheads associated with storage and synchronization should be within a reasonable constant factor of ideal implementations and the cost for generality should be minimal.
- *Flexibility*: The synchronization options should allow designers to pick the appropriate tradeoffs for their target workloads.
- *Performance*: The absolute performance of the research prototype should be reasonable enough for prototyping and for supporting moderately workloads.

In order to determine whether PADS meets the above stated criteria, we carry out our evaluation in three steps. First, we evaluate the theoretical network and storage overheads associated with PADS primitives. Second, we quantify the overheads and evaluate the flexibility by running microbenchmarks on our prototype. Lastly, we examine the absolute read/write performance and R/OverLog communication performance.

8.1 Fundamental Overheads

We evaluate the fundamental network and storage costs associated with PADS prototype by examining the costs for synchronization via subscriptions, for data and update log storage, and for conflict detection.

	Ideal	**PADS Prototype**
Subscription Setup		
Inval Subscription with LOG catch-up	$O(N_{SSPrevUpdates})$	$O(N_{nodes} + N_{SSPrevUpdates})$
Inval Subscription with CP from time=0	$O(N_{SSObj})$	$O(N_{SSObj})$
Inval Subscription with CP from time=VV	$O(N_{SSObjUpd})$	$O(N_{nodes} + N_{SSObjUpd})$
Body Subscription	$O(N_{SSObjUpd})$	$O(N_{SSObjUpd})$
Transmitting Updates		
Inval Subscription	$O(N_{SSNewUpdates})$	$O(N_{SSNewUpdates})$
Body Subscription	$O(N_{SSNewUpdates})$	$O(N_{SSNewUpdates})$

Table 8.1: Network overheads associated with subscription primitives. Here, N_{nodes} is the number of nodes. N_{SSObj} is the number of objects in the subscription set. $N_{SSPrevUpdates}$ and $N_{SSObjUpd}$ are the number of updates that occurred and the number objects in the subscription set that were modified from a subscription start time to the current logical time. $N_{SSNewUpdates}$ is the number of updates to the subscription set that occur after the subscription has caught up to the sender's logical time.

Synchronization costs. Table 8.1 shows the network cost associated with our prototype's implementation of PADS's primitives and indicates that our costs are close to the ideal of having actual costs be proportional to the amount of new information transferred between nodes. Note that these ideal costs may not be able always be achievable in practice.

During invalidation subscription setup, the sender transmits a version vector indicating the start time of the subscription and catch-up information as either a log of past updates or a checkpoint of the current state. The start time is simply a partial version vector that is proportional to the number of nodes in the system. That cost is amortized over all the updates sent on the connection.

For log catchup, in an ideal implementation, the sender would send information proportional to the number of updates that have occurred to objects in the subscription set. In PADS, the sender does the same—it sends precise invalidations for those updates. However, in order to support flexible consistency, invalidation subscriptions also carry extra information such as imprecise invalidations. Imprecise invalidations summarize updates to objects not part of the subscription set and are sent to mark logical gaps in the casual stream of invalidations. The number of imprecise invalidations sent depends on the workload and is never more than the number of invalidations of updates to objects in the subscription set sent. The size of imprecise invalidations depends on the locality of the workload and how compactly the invalidations compress into imprecise invalidations.

For checkpoint catchup, the sender sends meta-data for all the objects that were updated after the start time. Hence the cost is proportional to the number of objects that were modified plus the initial start time. Also, by starting the subscription at logical time 0, the overhead for the start time can be avoided. Note, checkpoint catch-up is particularly cheap when interest sets are small.

Once the invalidation subscription is set up, the sender simply sends precise invalidations of any new updates occur to objects in the subscription set. The precise invalidations are interspersed with imprecise invalidations. But, as mentioned before, the number of imprecise invalidations is no more than the number of precise invalidations.

PADS body subscriptions send the same information as an ideal implementation. The sender sends the bodies of objects in the subscription set that were modified after the start time. Once the receiver has caught up, it sends bodies of any new updates it receives.

Two things should be noted. First, the 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 places. Second, there are only two ways that PADS sends extra information over the ideal: the start-time vector during invalidation subscription set up and the imprecise invalidations sent on a stream to maintain flexible consistency. Overall, we expect PADS to scale well to systems with large numbers of objects or nodes—the per-node overheads associated with the version vectors used to set up subscriptions is amortized over all of the updates sent on the stream, and subscription sets and imprecise invalidations ensure that the number of records transferred is proportional to amount of data of interest (and not to the overall size of the database).

Storage costs. The minimal storage requirement is that a node stores the objects it is interested in.

However, it is impossible for a system to provide any reasonable consistency semantics if no bookkeeping information is maintained (even monotonic-reads coherence requires some bookkeeping for accept stamps). Therefore, in addition to data objects in the store, PADS maintains per-object meta-information, consistency bookkeeping per-interest set, and an update log. For every object, PADS maintains a logical time stamp, a real time stamp, a deleted flag, a valid flag and a commit flag. PADS also stores a version vector for every interest set in $ISStatus$. The number of interest sets is generally fixed. However, in the worst case, a version vector is maintained per object. The update log has a fixed maximum size and hence imposes a constant storage overhead.

Conflict detection. For conflict detection, PADS utilizes the consistency information already maintained and therefore exerts no extra overhead. However, for several other conflict-detection schemes, the amount of bookkeeping information increases with network disruptions. For PADS, the bookkeeping information remains the same because the number of interest sets a node maintains is not affected by disruptions. Hence, for k interest sets, the storage

	Version vectors	**PVE**	**Vector sets**	**PADS**	
				log sync	cp sync
Storage lower bound	$O(N \times R)$	$O(N + R)$	$O(N + R)$	$O(N + k \times R)$	$O(N + k \times R)$
Storage upper bound	$O(N \times R)$	unbounded	$O(N \times R)$	$O(N + k \times R)$	$O(N + k \times R)$
Network lower bound	$O(p \times R)$	$O(p + R)$	$O(p + R)$	$O(p + R)$	$O(p + R)$
Network upper bound	$O(p \times R)$	unbounded	$O(N \times R)$	$O(p \times R)$	$O(N \times R)$

Table 8.2: Storage and network overheads under network disruptions.
R nodes and N objects stored on a node in k interest sets with p recent updates

overhead is $O(N + k \times R)$. If invalidations subscriptions are disrupted, they simply re-start where they left off incurring extra version vector overhead due to the resending of subscription start time. Hence, in the worst case, for log synchronization, there is a version vector overhead per update sent and for checkpoint synchronization, there is a version vector overhead per object sent.

Given the amount of flexibility that PADS supports, conflict detection costs are reasonable, see Table 8.2. In fact, the overheads of conflict detection is comparable to existing state-of-the-art approaches that do not provide such flexibility. We only compare the catchup phase because it is not clear whether the schemes have an equivalent "connected" phase. PADS performs as well as vector sets while providing stronger consistency guarantees.

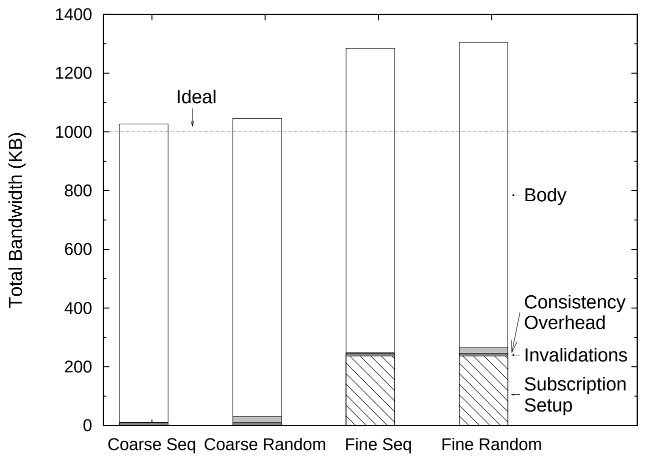

Figure 8.1: Network bandwidth cost to synchronize 1000 10KB files, 100 of which are modified.

8.2 Quantifying the Constants

We investigate whether the actual performance of the prototype matches the cost model by running experiments to quantify the overheads associated with subscription setup, with flexible consistency support, and with the different options exposed by the primitives.

Synchronization cost under different workloads. Figure 8.1 illustrates the synchronization cost for a simple scenario. In this experiment, there are 10,000 objects in the system organized into 10 groups of 1,000 objects each, and each object's size is 10KB. The reader registers to receive updates for one of these groups by establishing subscriptions. Then, the writer updates 100 of the objects in each group. Finally, the reader reads all 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, one for each 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" overheads associated with PADS subscription setup and flexible consistency over the best case is a small fraction of the total overhead in all cases. Third, when writes have locality, the overhead of flexible consistency drops further because larger numbers of invalidations are combined into an imprecise invalidation. Fourth, coarse-grained synchronization has lower overhead than fine-grained synchronization because it avoids per-object subscription setup costs.

Advantage of partial replication. Figure 8.2 demonstrates the bandwidth savings achieved due to PADS's support for partial replication. Node A updates 500 objects of size 3KB and Node B establishes subscriptions to synchronize these updates. Two scenarios are investigated: in one, subscriptions are established with the subscription set covering 100% of the data, and in the

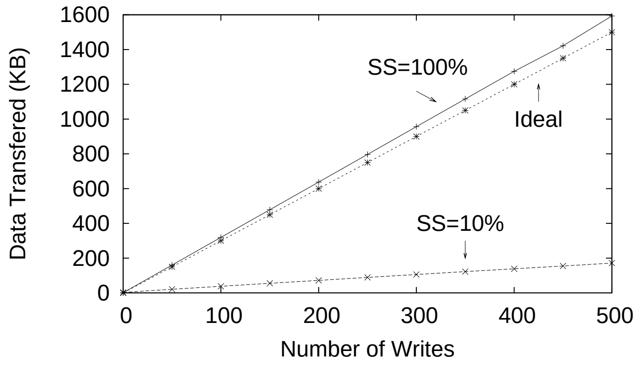

Figure 8.2: Bandwidth to synchronize 500 objects of size 3KB

	Coherence-only	**PADS**
Bursty workload (1 in 10)	1	1.1
Worst Case (1 in 2)	1	2

Table 8.3: Messages per interested update sent by a coherence-only system and PADS.

other, the subscriptions only cover 10% of the data. The bandwidth required by the subscriptions for synchronization is measured. The results depicted in Figure 8.2 allow us to come to two conclusions. First, the results demonstrate that the overhead for synchronization is relatively small even for small files compared to an *ideal* implementation (plotted by counting the bytes of data that must be sent ignoring all metadata overheads). More importantly, they demonstrate that if a node requires only a fraction (e.g., 10%) of the data, PADS keeps information that is not relevant to that subset to a minimum and in the process greatly reduces the bandwidth required for synchronization.

Cost of flexible consistency. We evaluate the cost PADS pays to support flexible consistency. For systems that require weak consistency such as coherence, they simply send updates without imprecise invalidations. For systems that require strong consistency, they need to send imprecise invalidations to ensure casual semantics over which stronger guarantees can be implemented. In particular, we quantify the cost of sending imprecise invalidations in an invalidation stream. Table 8.3 compares the number of messages per update between a coherence-only system and PADS. In a coherence-only system, only updates to objects in the synchronization set are sent on the stream. On the other hand, PADS also sends imprecise invalidations for updates outside the subscription set.

The number of imprecise invalidations sent depends on the workload. For a bursty workload, say if 9 out of 10 updates occur to objects in the subscription set, an imprecise invalidation is only sent after nine invalidations. In the worst case workload, PADS sends an imprecise invalidation after every precise invalidation. Thus, PADS sends at most twice the number of messages when compared to a coherence-only system. However, since imprecise invalidations are significantly smaller than actual bodies, the overhead remains within reasonable bounds.

Advantage of multiplexing subscriptions. Efficient support for multiple dynamic subscriptions is important because they are used to implement demand caching with per-object callbacks [49]. For example, each time a client

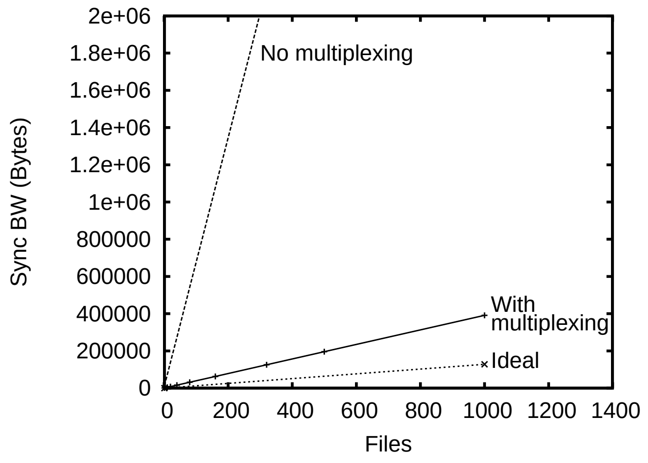

Figure 8.3: Bandwidth to subscribe to varying number of single-object interest sets with and without subscription multiplexing.

caches a new object, it creates a new subscription for that object.

We compare the efficiency of establishing multiple dynamic subscriptions with and without subscription multiplexing. Figure 8.3 depicts the bandwidth costs for establishing single-object invalidation subscriptions for the ideal case, without subscription multiplexing, and with subscription multiplexing (i.e. PADS). In the ideal case, only the object is sent for every subscription request. Without subscription multiplexing, a separate invalidation stream is established for each request. Other than subscription start and end messages, imprecise invalidations are sent on each stream. PADS multiplexes subscription requests on a single stream—only the catchup start, the object, and catchup end messages are sent for each request. The major cost savings comes from the reduction of redundant invalidation information received by a node. Since the prototype is implemented in Java, the inefficiency of Java serialization does affect the bandwidth cost. However, it is not difficult to see that the bandwidth subscription multiplexing achieves comes much closer to ideal when compared to no multiplexing.

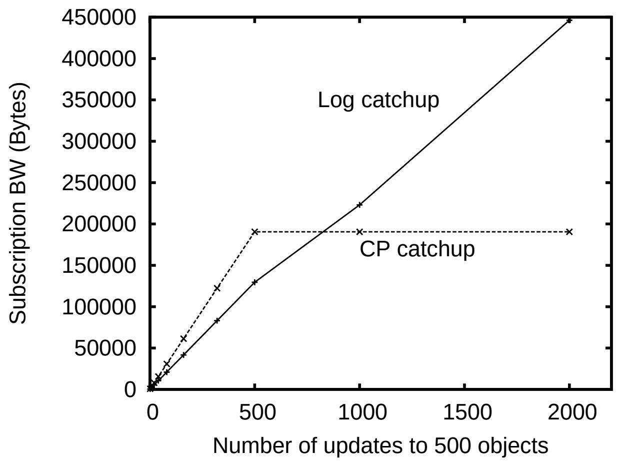

Figure 8.4: Bandwidth to subscribe to varying number of updates to 500 objects sets for checkpoint and log synchronization.

Log vs. checkpoint catchup. Figure 8.4 compares the bandwidth cost for log and checkpoint synchronization. A set of 500 objects were updated uniformly and invalidation subscriptions are established separately for each object. As the figure illustrates, the synchronization cost for both options are proportional to the number of updates when each object is not updated more than once. Checkpoint synchronization does worse because the size of the meta-data sent in a checkpoint is slightly larger than an imprecise invalidation sent for log synchronization. However, when an object is updated multiple times, checkpoint catchup outperforms log synchronization.

	Write (sync)	**Write (async)**	**Read (cold)**	**Read (warm)**
ext3	6.64	0.02	0.04	0.02
PADS Object Store	8.47	1.27	0.25	0.16

Table 8.4: Read/write performance for 1KB objects/files in milliseconds.

	Write (sync)	**Write (async)**	**Read (cold)**	**Read (warm)**
ext3	19.08	0.13	0.20	0.19
PADS Object Store	52.43	43.08	0.90	0.35

Table 8.5: Read/write performance for 100KB objects/files in milliseconds.

8.3 Absolute Performance

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

Local performance. Tables 8.4 and 8.5 summarize the performance for reading and writing 1KB and 100KB objects stored locally in PADS and compare it to the performance for reading or writing 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. We measure the performance of synchronous writes (i.e. the write returns after update is committed to disk), asynchronous writes (i.e. the write returns after the update is committed to memory), cold reads (i.e. the data need to be fetched from disk), and warm reads (i.e. the data is already located in memory). The values reported are averages of 5 runs.

Overheads are significant. Upon further investigation, we find the bottleneck in the mechanism layer object store and consistency bookkeeping implementation. When an update occurs, the disk may be accessed 4 times: to store the invalidation in the log, to store the body in the object store, to update the object meta-data, and to update the byte-range meta-data. Also, by allowing writes to occur to any object byte-range, a decent amount of complexity is introduced—to clean per-range meta-data and the bodies stored in the database when a new write overlaps multiple existing writes. We have strong evidence to believe that by optimizing object and meta-data storage, by restricting writes to fixed-size chunks, and by implementing a customized disk sync policy, performance can be greatly improved. In the meantime, the current prototype still provides sufficient performance for a wide range of systems.

R/OverLog performance. Execution of routing rules has significant impact on the performance of systems built on PADS. An earlier version of our system used P2 [56] to execute the routing rules. Unfortunately, the P2 run-

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

Table 8.6: Performance numbers for processing NULL trigger to produce NULL event.

time was not designed to receive external events. With a workaround, it was possible to insert and receive events from P2, but with inferior performance. As described in Chapter 7, we currently use a customized translator that converts R/OverLog programs to Java and a R/OverLog runtime that natively implements an interface to allow PADS to inject and receive events from the rules. In addtion, R/OverLog imposes a local-lookups-only limitation, i.e. a rule can only be look up local tables in its body. In OverLog, a single rule can look up tables on multiple nodes. The P2 runtime establishes and maintains tuple streams between nodes in order to support that. However, by imposing the local-lookups-only limitation, communcation between nodes in R/OverLog is reduced to explicit event transfers making the runtime smaller and more streamlined. As a result, we were able to achieve a significant performance boost. Table 8.6 quantifies the performance difference between the P2 and R/OverLog runtimes.

8.4 Summary

In this chapter, we evaluate the overheads associated with the generality of PADS. We find that the benefits of using PADS as a development platform compelling. Architecturally, PADS is sound— its network and storage overheads are within small constant factors of ideal implementations. Also, PADS provides sufficient flexibility to allow designers to pick the tradeoffs best suited for their target workloads. Finally, in terms of absolute performance, since the PADS prototype is a user-level Java implementation, the performance is sufficient for prototyping and moderately demanding workloads. However, this drawback can be seen as an artifact of the prototype implementation rather than that of the approach.

Chapter 9

Evaluation 2: Case-studies

This chapter evaluates PADS's approach for building new distributed systems. The goal of the evaluation is to demonstrate that the primitives are sufficient to implement a broad range of systems, that the policy architecture is easy to use, and that the systems developed with it are real and can be easily adapted to new requirements. One way to evaluate PADS would be to construct a new system for a new demanding environment and report on that experience. We choose a different approach—constructing a broad range of existing systems—for three reasons. First, a single system may not cover all of the design choices or test the limits of PADS. Second, it might not be clear how to generalize the experience from building one system to building others. Third, it might be difficult to disentangle the challenges of designing a new system for a new environment from the challenges of realizing a given design using PADS.

This chapter provides details of our experience. First, it defines the evaluation criteria for a good framework from a developer's point of view. Then, it defines notion of *architectural equivalence* that is used as a yardstick to compare our implementation of a system with its original description. It then details the implementation of each case-study system. Finally, it describes the experiments carried out to evaluate the performance, realism, or the agility of the systems developed.

9.1 Evaluation Overview

The goal of PADS is to make it easier to develop new systems. Our evaluation aims to answer the following questions from a system developer's point of view:

- *Is PADS flexible enough to support my system?* We build a wide range of systems inspired from literature, including client-server systems, server-replication systems, and ad-hoc peer-to-peer systems. These systems were chosen because they embody a wide range of replication techniques that are often used by new systems. PADS's ability to support this wide range of systems suggests that PADS is sufficiently powerful to define systems spanning a major portion of the design space.
- *Is development easier on PADS?* The fact that a small team could quickly build a dozen significant replication systems is evidence of our experience that PADS greatly reduces development time and effort. Developers only need to focus on implementing and debugging control plane policy instead of implementing data plane mechanisms such as object storage, consistency management, and communication. In addition, each system built on PADS requires relatively few lines of code – typically several

dozens of routing rules and a handful of blocking conditions. In our experience, it is easier to understand and debug a system that has a hundred lines rather than thousands of lines of code. There is, however, an initial learning curve to learn the routing language, to learn the API, and to change the development mindset to the blocking and routing approach. We note, however, that students in a graduate operating system class have used PADS as a development platform for their projects without much trouble after the initial learning curve.

- *Does PADS facilitate evolving system design to incorporate new features or meet new demands?* We added significant new features to several systems we built that allowed order-of-magnitude performance improvements over the baseline design. The features were added by changing fewer than 10 routing rules.

- *Can PADS be used to build real systems?* When building a concrete distributed storage system, a system designer often needs to deal with practical issues such as setting up configuration options, handling crashes and recovery, maintaining consistency and durability during periods of crashes, and dynamic addition and removal of nodes. PADS makes it easy to implement the above details: First, the stored events primitive allows routing policies to dynamically store and retrieve configuration options from data objects. Second, during recovery, the underlying mechanisms take care of complex details such as state recovery so that policy writers only need to re-establish subscriptions. Third, blocking predicates make it easy to reason about consistency during crashes because they prevent any "unsafe" data from being accessed. All systems we built are concrete in the sense that they handle the above issues. To demonstrate the "concreteness" of the systems, we examine the behaviour of some of the systems under adverse network conditions (refer to Section 9.4).

- *What are the overheads?* We evaluate the overheads associated with the generality of PADS. As described in Chapter 8, architecturally PADS seems sound—its network and storage overheads are within small constant factors of hand-tuned systems. In terms of absolute performance, since the PADS prototype is a user-level Java implementation, a system built on PADS is within ten to fifty percent of the original system in most cases and 3.3 times worse in the worst case we measured.

9.2 Architectural Equivalence

We build systems based on system designs from the literature, but constructing perfect, "bug-compatible" duplicates of the original systems using PADS is not a realistic (or useful) goal. On the other hand, if we were free to pick and choose arbitrary subsets of features to exclude, then the bar for evaluating PADS is too low: we can claim to have built any system by simply excluding any features PADS has difficulty supporting.

Section 3 identifies three aspects of system design—security, interface, and conflict resolution—for which PADS provides limited support, and our implementations of the above systems do not attempt to mimic the original designs in these dimensions.

Beyond that, we have attempted to faithfully implement the designs in the papers cited. More precisely, although our implementations certainly differ in some details, we believe we have built systems that are *architecturally equivalent* to the original designs. We define architectural equivalence in terms of three properties:

E1. *Equivalent overhead.* A system's network bandwidth between any pair of nodes and its local storage at any node are within a small constant factor of the target system.

E2. *Equivalent consistency.* The system provides consistency and staleness properties that are at least as strong as the target system's.

E3. *Equivalent local data.* The set of data that may be accessed from the system's local state without network communication is a superset of the set of data that may be accessed from the target system's local state. Notice that this property addresses 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 vs. Availability vs. Partition-resilience) [30] and PC (Performance vs. Consistency) [55] tradeoffs that any distributed storage 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. Similarly, one can improve the availability a system offers for a given level of consistency by using more network bandwidth to synchronize more often [94], or one can reduce the resources consumed by reducing the amount of data cached at a node.

9.3 Case-studies

In this section, we evaluate PADS's flexibility and ease of use as a development platform by constructing 8 distributed storage systems inspired from literature. The constructed systems include

- client-server systems such as Simple Client-Server(SCS), Full Client Server(FCS), Coda [49], and TRIP [64],
- server replication systems such as Bayou [69], and Chain Replication [88], and
- object-replication systems such as TierStore [26] and Pangaea [77].

These systems were chosen because they cover a wide range of design features in a number of key dimensions. For example,

- *Replication:* full replication (Bayou, Chain Replication, and TRIP), partial replication (Coda, Pangaea, FCS, and TierStore), and demand caching (Coda, Pangaea, and FCS),

- *Topology:* structured topologies such as client-server (Coda, FCS, and TRIP), hierarchical (TierStore), and chain (Chain Replication); unstructured topologies (Bayou and Pangaea). Invalidation-based (Coda and FCS) and update-based (Bayou, TierStore, and TRIP) propagation.

- *Consistency:* monotonic-reads coherence (Pangaea and TierStore), casual (Bayou), sequential (FCS and TRIP), and linearizability (Chain Replication); techniques such as callbacks (Coda, FCS, and TRIP) and leases (Coda and FCS).

- *Availability:* Disconnected operation (Bayou, Coda, TierStore, and TRIP), crash recovery (all), and network reconnection (all).

Table 1.1 on Page 5 summarizes the features for each system and the following subsections provide details of the implementations. In order to differentiate the PADS implementations from the original implementations, we add a prefix "P-" to the PADS implementations.

9.3.1 Simple Client Server (SCS)

This simple system includes support for a client-server architecture, invalidation callbacks [42], sequential consistency [53], correctness in the face of crash/recovery of any node, and configuration. For simplicity, it assumes that a write overwrites an entire file.

We choose this example not because it is inherently interesting but because it is simple yet sufficient to illustrate the main aspects of PADS, including support for coarse- and fine-grained synchronization, consistency, durability, and configuration. In Section 9.3.2, we extend the example with features that are relevant for practical deployments, including leases [32], cooperative caching [23], and support for partial-file writes.

9.3.1.1 System Overview

The server stores the full set of data. Clients cache a subset of data locally. On a read miss, a client fetches the missing object from the server and registers for callback for the fetched object. On a write, a client sends the update to the server and waits for an acknowledgement from the server. Upon receiving an update of an object, a server sends invalidations to all clients registered to receive callbacks for that object. Upon receiving an invalidation of an object, a client sends an acknowledgment to the server and cancels the callback. Once the server has received all the acknowledgements for an update from other clients, the server informs the original writer so it can continue. For durability reasons, a write by a client is not seen by any other client until the server has persistently stored the update.

Since multiple clients can issue concurrent writes to multiple objects, for sequential consistency, the system defines a global sequential order on those writes and ensures that they are observed in that order. In a client-server system, it is natural to have the server set that total order. Therefore, upon receiving acknowledgements for all of an update's invalidations, the server assigns the update a position in the global total order. The system guarantees sequential consistency by ensuring that a write of an object is blocked until all earlier versions have been invalidated, a read of an object is blocked until the reader holds a valid, consistent copy of the object, and a read or write of an object is blocked until the client is guaranteed to observe the effects of all earlier updates in the sequence of updates defined by the server.

9.3.1.2 P-SCS Implementation

The implementation of P-SCS requires 24 routing rules and 5 blocking conditions. In this section, we provide a brief overview of the implementation. Details of the implementation is described Appendix A.1. We include rule counts in our descriptions to emphasize how concisely the design features can be specified.

System configuration. Every client needs to know the address of the server to contact. At startup, a node looks up a configuration file using the stored events interface and stores the identifier in an R/OverLog table (2 rules: *ini1, ini2*).

Sending updates to server. As soon as a client receives the server address, it establishes invalidation and body subscriptions to the server so that local updates (2 rules: *csSb1, csSb2*) can be automatically transferred to the server. In order to deal with failures, the client tries to reestablish the subscriptions when they fail (2 rules: *csSb3, csSb4*).

Read misses and callbacks. Whenever a client suffers a read miss, the routing policy is informed by a *operationBlock* trigger for the *readNowBlock*. When that happens, the client informs the server and stores outstanding misses in a local table (2 rules: *rm1, rm2*). When the server is informed of a client read miss, it sends the body to the client and establishes a callback. In PADS, a callback is simply an invalidation subscription from the server to the client. The server also puts an entry into a table to keep track of the clients have that callbacks for objects (3 rules: *cb1, cb2, cb3*).

Client writes and ack management. Whenever a client updates an object, the invalidation and the body propagates to server via the subscriptions established at startup. An invalidation of the update automatically propagates to other clients that hold callbacks via the established invalidation subscriptions. When a client receives an invalidation, it will check if the received invalidation corresponds to an outstanding read miss (1 rule:*rinv0*). If not, it breaks the callback (by removing the invalidation subscription) and sends an ack to the server (2 rules: *rinv1, rinv2*). Otherwise, the client will consider the miss satisfied, and delete the entry from its read miss table. (1 rule: *rinv3*). The server gathers acks from the clients. Once all the required acks have been received, the server assigns a sequence number to the write and informs the original writer. The server-side ack management takes 8 rules in total: *ack1-ack8*.

Enforcing durability and consistency. The blocking policy helps enforce durability and consistency guarantees by specifying 5 conditions.

In order to ensure durability, the *isValid* condition is specified for the *applyUpdateBlock* at the server and at clients. Since a received invalidation is not applied until the corresponding body is received, it is guaranteed that no update can be seen by any other client until the server has stored it.

Consistency is guaranteed both by routing rules and access blocks. A routing rule (*ack7*), ensures that an update is assigned a sequence number only after all other copies of the data have been invalidated. On the client side, the *readNowBlock* is set to 3 conditions:*isValid* and *isComplete* and *isSequenced*. These conditions ensure that reads only return valid consistent copies that have been sequenced by the server. The *writeAfterBlock* on the client is set to *B_action writeComplete*. In other words, the write only unblocks after it receives a message from the server, which, due to the ack management routing rules, is only sent to the writer when all other copies have been invalidated. Even when multiple clients are updating the same object, the observed order at any client is as the same sequenced order observed at server.

9.3.2 Full Client Server (FCS)

The full client server adds several features to the simple client server implementation to make it more practical including volume leases [93], cooperative caching [23], partial-file writes, and blind writes. Complete details of the implementation can be found in Appendix A.2.

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

We add cooperative caching by replacing the rule that sends a body from the server with 5 rules: 3 rules (*cc2, cc3, cc4*) to find a helper and get data from the helper, and 2 rules to fall back on the server if no helper is found (*cc5*) or when the helper fails to satisfy the request (*cc1*). Note that reasoning about cache consistency remains easy because invalidation metadata still follows client-server paths, and the blocking predicates ensure that a body is not read until the corresponding invalidation has been processed. In contrast, some previous implementations of cooperative caching found it challenging to reason about consistency [18].

We add support for partial-file writes by removing one and adding six rules (*pfw1-pfw6*) to track which blocks each client is caching and to cancel a callback subscription for a file only when all blocks have been invalidated.

Finally, we add three rules (*bw1-bw3*) to the server that check for blind writes when no callback is held and to establish callbacks for them.

In total, P-FCS requires 43 routing rules and 6 blocking conditions.

9.3.3 Coda

We implement P-Coda, a system inspired by the version of Coda described by Kistler et. al. [49]. P-Coda supports disconnected operation, reintegration, crash recovery, whole-file caching, open/close consistency (when connected), causal consistency (when disconnected), and hoarding. We know of one feature from this version that we are missing: we do not support cache replacement prioritization. In Coda, some files and directories can be given a lower priority and will be discarded from cache before others. Coda is long-running project with many papers worth of ideas. We omit features discussed in other papers like server replication [78], trickle reintegration [60], and variable granularity cache coherence [61]. We see no fundamental barriers to adding them in P-Coda. We also illustrate the ease with which co-operative caching can be added to P-Coda.

9.3.3.1 System Overview

P-Coda is a client-server system, similar to the simple client server system (SCS) discussed earlier. The main differences between P-Coda and P-SCS are detailed below.

First, P-Coda provides open-to-close semantics which means that when a file is opened at a client, the client will return a local valid copy or retrieve the newest version from the server. Subsequent updates to the file are buffered and are sent to the server on file is closed.

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

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

9.3.3.2 P-Coda Implementation

As detailed in Appendix A.3, we extend the routing rules of the simple-client server model to implement P-Coda in 31 rules. First, like the full client-server example, we add 3 rules to check for blind writes when no callback is held and establish callbacks for them. Second, we add 4 rules (*ss1-ss4*) to keep track of server status and two rules (*li1* and *li2*) to let writes and reads continue when disconnected from the server. Third, we implement hoarding in 2 rules (*hd1-hd2*) by storing the hoard set as tuples in a configuration file and establishing invalidation and body subscriptions for each of them whenever the client connects to the server. Fourth, in order to allow a client to quickly get information about the updates it has missed, we add a rule (*csSb6)* so that when a client reconnects to a server, it establishes an invalidation subscription for an *empty* set. This action causes the server to send an imprecise invalidation for the missed updates.

The blocking policy adds the *B_ACT isDisconnected* condition to *readNowBlock* and *writeAfterBlock* so as to allow reads and writes to continue when operating in disconnected mode.

In order to support open-to-close semantics, we implement a wrapper layer over the underlying read/write interface. When a file is opened, a read for the whole object is issued at the read interface. All writes are buffered by the wrapper layer and are issued to the underlying write interface when the file is closed. The blocking policy ensures that the close returns after all updates are propagated to the server and all copies cached on other clients have been invalidated.

P-Coda and cooperative caching. In P-Coda, on a read miss, a client is restricted to retrieving data from the server. We add cooperative caching to P-Coda by adding 8 rules: 5 to monitor the reachability of nearby nodes, 1 to retrieve data from nearby peers on a read miss, and 2 to get invalidations from nearby clients if the server is not reachable. Cooperative caching has two advantages: first, it can greatly reduce the read latency on a miss if the peer is much closer than the server, and second it allows disconnected clients to access files they previously could not.

9.3.4 TRIP

TRIP [64] is a distributed storage system for large-scale information dissemination: all updates occur at a server and all reads occur at clients. TRIP uses a self-tuning prefetch algorithm to send higher priority updates sooner and delays applying invalidations to a client's locally cached data to maximize the amount of data that a client can serve from its local state. TRIP guarantees sequential consistency via a simple algorithm that exploits the constraint that all writes are carried out by a single server.

9.3.4.1 P-TRIP Implementation

Information propagation in P-TRIP follows a star topology—every client is connected to the server via invalidation and body subscriptions.

To support self-tuning, whenever the server issues a write, it assigns a priority to the update. Since body subscriptions have no ordering constraints, they send bodies of higher priority updates before lower-priority updates. Also, the application of invalidations is delayed up to a threshold by setting the *applyUpdateBlock* is to (*isValid or maxStaleness*). Since all writes occur at the server, sequential consistency can be guaranteed by setting the *readNowBlock* to (*isValid and isComplete*). As detailed in Appendix A.4, the routing policy consists of 6 rules in total and the blocking policy of 4 conditions.

P-TRIP and hierarchical topology. TRIP assumes a single server and a star topology. We can improve scalability by changing the topology from a star to a static tree by simply changing a node's configuration file to list a different node as its parent. PADS's mechanisms are general enough so that this "just works"—invalidations and bodies flow as intended and consistency is still maintained. Better still, if one writes a topology policy that dynamically reconfigures a tree when nodes become available or unavailable [56], a few additional rules to establish or remove subscriptions to produce a dynamic-tree version of TRIP that still enforces sequential consistency. Note that we have implemented the static tree policy (refer to Appendix A.4) but not the dynamic tree policy.

9.3.5 Bayou

Bayou [69] is a server-replication protocol that focuses on peer-to-peer data sharing. Every node has a local copy of all of the system's data. From time to time, a node picks a peer to exchange updates with via *anti-entropy sessions.* Actually, Bayou transfers updates as operations. However, since PADS does not support operation transfer, we implement a state-transfer version of Bayou. Bayou guarantees causal and eventual consistency.

9.3.5.1 P-Bayou Implementation

As detailed in Appendix A.5, an anti-entropy session in PADS is implemented by setting up invalidation and body subscriptions between two nodes. Once all the updates have been transferred, the subscriptions are removed. Note that if the log at the sender has been truncated to a point beyond the receiver's consistency state, the invalidation subscription will automatically send a checkpoint rather than the log; Bayou's approach is similar. The complete implementation of P-Bayou's routing policy requires 10 rules: 1 for picking a random peer, 4 for carrying out anti-entropy and 5 for connection management.

Causal consistency is guaranteed by setting the *readNowBlock* to (*isValid and isComplete*). Also, the *applyUpdateBlock* is set to *isValid* to ensure that no local data is invalid.

P-Bayou and small device support. Since the protocol propagates updates for the whole data set to every node, P-Bayou cannot efficiently support smaller devices that have limited storage or bandwidth.

It is easy to change P-Bayou to support small devices. In the original P-Bayou design, when anti-entropy is triggered, a node connects to a reachable peer and subscribes to receive invalidations and bodies for all objects using a subscription set "/*". In our small device variation, a node uses stored events to read a list of directories from a per-node configuration file and subscribes only for the listed sub-directories. This change required us to modify two routing rules.

This change raises an issue for the designer. If a small device C synchronizes with a first complete server $S1$, it will not receive updates to objects outside of its subscription sets. These omissions will not affect C since C will not access those objects. However, if C later synchronizes with a second complete server $S2$, $S2$ may end up with causal gaps in its update logs due to the missing updates that C doesn't subscribe to. The designer has three choices: to weaken consistency from causal to per-object coherence; to restrict communication to avoid such situations (e.g., prevent C from synchronizing with $S2$); or to weaken availability by forcing $S2$ to fill its gaps by talking to another server before allowing local reads of potentially stale objects. We choose the first, so we change the blocking predicate for reads to no longer require the *isComplete* condition. Other designers may make different choices depending on their environment and goals.

9.3.6 Chain Replication

Chain Replication [88] is a server replication protocol in which the nodes are arranged as a chain to provide high availability and linearizability. All updates are introduced at the head of the chain and queries are handled by the tail. An update does not complete until all live nodes in the chain have received it. Chain management is carried out by an always-available master.

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

9.3.6.1 P-Chain-Replication Implementation

P-ChainReplication 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 (Rules *sub1* and *sub2*). The master process is also implemented in R/OverLog and is assumed to never fail. We see no difficulty in extending the master to a Paxos-based [54] implementation. Appendix A.6 details the routing and blocking policies of P-Chain-Replication.

Volume support. P-Chain-Replication assumes that the prefix of every objectId indicates its volume. For example, the object */V1/a/b* belongs to volume *V1.* In order to aid the specification of the routing policy, we extend R/OverLog runtime to implement a function *f_getVol()* that will return the volume for a given object. We also assume that there is a *volume list* that specifies the volumes that should be replicated on each node.

Enforcing linearizability. In order to support linearizability, an update occurs only at the head and is blocked until it has propagated down all nodes to the tail of the chain. Once the tail receives an update, it sends an ack upstream. In the original chain replication system, every node keeps track of the updates it has sent downstream for which it has not received an ack. However, in P-ChainReplication, it is not necessary to carry out this complex bookkeeping. Because updates propagate along the links in causal order, there is no need to send an ack through each link. Instead, the tail simply sends an ack to the head. The ack indicates that all nodes in the chain have received that update and all casually preceding updates. Rules *con1* to *con6* provide details of ack management.

The blocking policy is defined so that reads return only causal data (i.e. *readNowBlock* is set to *isValid* and *isComplete*), and a write is blocked until an ack is received (i.e. *writeAfterBlock* is set to *B_Action(AckFromTail)*). Also, the *applyUpdateBlock* is set to *isValid* to ensure that every node has the body of an update before its invalidation is applied.

Chain management. As in the original system, chain management is carried out by the master. The master keeps track of the reachability of all nodes and maintains a table that stores the configuration of the volume chain. When the master detects a new node or a node failure, it rearranges the chain as described below.

Addition of new nodes. When the master detects a new node, it is added to the tail of the chain as follows: The master sends messages to the new node informing it of the current head and tail of the volume chain and its predecessor. The node then establishes subscriptions to its predecessor. Once, the subscription has caught-up (i.e. the node's state reflects that of the chain), it can start functioning as a tail and notifies all other nodes. Rules *nn1* to *nn15* handle new node addition.

Failure recovery. When a node fails, the master detects the failure and rearranges the chain. If the head fails, the next node in the chain becomes the the head. If the tail fails, the node before it in the chain becomes the new tail. If a node in the middle fails, then its predecessor and successor are linked by establishing subscriptions. Note that if a node recovers, it is added to the end of the chain, just like a new node.

In the original system, recovery from mid-chain failure involves a complex algorithm to ensure that the node preceding the failure sends all updates that the new successor is missing and that the successor sends all the missing acks to the predecessor. However, in the PADS implementation, all this complexity is eliminated. The preceding node only needs to establish subscriptions and the underlying mechanisms will automatically send the missing updates down the chain. Also, since the tail sends an ack directly to the head, hop-by-hop ack management is not needed. Rules *dn1* to *dn7* provide details of failure management.

9.3.7 TierStore

TierStore [26] is a distributed object storage system that targets developing regions where networks are bandwidth-constrained and unreliable. Each node reads and writes specific subsets of data. Since nodes must often operate in disconnected mode, the system prioritizes 100% availability over strong consistency.

9.3.7.1 System Overview

In order to achieve these goals, TierStore employs a hierarchical publish/subscribe system: all nodes are arranged in a tree. To propagate updates up the tree, every node sends all of its updates and its children's updates to its parent. To flood data down the tree, data are partitioned into "publications" and every node subscribes to a set of publications from its parent node covering its own interests and those of its children. For consistency, TierStore only supports single-object monotonic reads coherence.

9.3.7.2 P-TierStore Implementation

A 14-rule routing policy establishes and maintains the publication aggregation and multicast trees. A full listing of these rules is available in the Appendix A.7. In terms of PADS primitives, each connection in the tree is simply an invalidation subscription and a body subscription between a pair of nodes. Every PADS node stores in configuration objects the ID of its parent and the set of publications to subscribe to.

On start up, a node uses the stored events interface to read configuration objects and store the configuration information in R/OverLog tables (4 rules: *in0, pp0, pp1, pSb0*). When it knows of the ID of its parent, it adds subscriptions for every item in the publication set (2 rules: *pSb1, pSb2*). For every child, it adds subscriptions for "/*" to receive all updates from the child

(2 rules: *cSb1, cSb2*). If an application decides to subscribe to another publication, it simply writes to the configuration object. When this update occurs, a new stored event is generated and the routing rules add subscriptions for the new publication.

Recovery. If an incoming or an outgoing subscription fails, the node periodically tries to re-establish the connection (2 rules: *f1, f2*). Crash recovery requires no extra policy rules. When a node crashes and starts up, it simply re-establishes the subscriptions using its local logical time as the subscription's start time. The underlying subscription mechanisms automatically detect which updates the receiver is missing and send them.

Delay tolerant network (DTN) support. P-TierStore supports DTN environments by allowing one or more mobile PADS nodes to relay information between a parent and a child in a distribution tree. In this configuration, whenever a relay node arrives, a node subscribes to receive any new updates the relay node brings and pushes all new local updates for the parent or child subscription to the relay node (4 rules: *dtn1, dtn2, dtn3, dtn4*). An alternative approach would be to make use of existing DTN network protocols. This approach is straight-forward to implement if the DTN layer informs the policy layer when it has an opportunity to send to another node and when that opportunity ends. An opportunity could be that a TCP connection opens up or a USB drive was inserted. The routing policy would establish subscriptions to send updates within that connection opportunity as a DTN bundle.

Blocking policy. Blocking policy is simple because TierStore has weak consistency requirements. Since TierStore prefers stale available data to unavailable data, we set the *applyUpdateBlock* to *isValid* to avoid applying an invalidation until the corresponding body is received.

TierStore vs. P-TierStore. Publications in TierStore are defined by a container name and depth to include all objects up to that depth from the root of the publication. However, since P-TierStore uses a name hierarchy to define publications (e.g., /publication1/*), all objects under the directory tree become part of the subscription with no limit on depth.

Also, PADS provides a single conflict-resolution mechanism, which differs from that of TierStore in some details. Similarly, TierStore provides native support for directory objects, while PADS supports a simple untyped object store interface.

9.3.8 Pangaea

Pangaea [77] is a peer-to-peer distributed storage system for wide area networks that supports high degrees of replication and high availability. For each object, Pangaea maintains a connected graph and updates to that object are pushed along graph edges. Pangaea also maintains three gold replicas for every object to ensure data durability. The location of gold replicas for each object is stored in the object's parent (i.e. the directory entry for the file). For consistency, Pangaea guarantees only weak best-effort coherence.

9.3.8.1 P-Pangaea Implementation

P-Pangaea implements object creation, replica creation, update propagation, 3 gold nodes and 3-connected graph maintenance, temporary failure recovery, and permanent failure recovery. The graph edges in P-Pangaea are simply invalidation and body subscriptions. We currently do not implement harbingers. However, since harbingers map to invalidations, support for them can be easily added by only using invalidation subscriptions as graph edges and fetching the bodies from the fastest link as required. Also, 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 them. Appendix A.8 details the implementation of P-Pangaea.

Network management. The original system implements a gossip-based membership module to keep of track of node liveness and latency between nodes pairs. P-Pangaea uses a simple ping-based membership protocol and assumes that the latency between node pairs is static and is provided in a configuration file.

Gold node information. In P-Pangaea, instead of mapping a file/directory to a single object, it is mapped to two: a *.data* object to store the data and a *.meta* to store the gold node location of its children. For example, the file "/a/b" actually maps to objects "/a/b/.data" and "/a/b/.meta" and gold node locations of "/a/b" are stored in "/a/.meta". In order to aid the specification of routing rules, we extend the R/OverLog runtime with string manipulation functions to extract the parent directory of a file from its ID (*f_getParent()*), to get the fileId from the meta or data objectId (*f_getObj()*), and vice versa (*f_getData(), f_getMeta()*). Also, a wrapper is implemented at the local interface to translate all reads and writes to an object to its *.data* object. Like the original system, in P-Pangaea, all read and write requests are preceded by a directory lookup.

Whenever an object locally replicated, the routing rules use the stored events interface to read and watch the associated *.meta* object to be informed of any updates to gold node information.

Update propagation. Replicas of an object are arranged in a connected graph with invalidation and body subscriptions as the edges. Whenever an update occurs, the update automatically floods the graph via the subscriptions. The blocking policy sets the *applyUpdateBlock* to *isValid* so that both the body and invalidation of an update are stored locally before the update is propagated to other nodes.

Replica addition. When a node, N, tries to access an object that is not present locally (i.e. read miss), the system proceeds to create a replica locally. It will look up the location of the gold node for that object from the parent directory. If the parent directory is not locally available, it looks up its parent. This can continue recursively up to the root directory.

In order to create a local replica, N contacts the gold node of the object to retrieve the data and creates links to 3 replicas (1 gold and 2 others) so that the local replica is integrated in the connected graph with at least one direct link to a gold node.

File creation. In order to create a new object, a node will pick two other nodes and establish connections to them for that object. It then declares the two nodes and itself as gold gold nodes for that object and updates the parent directory object with that information. Note that the update automatically floods the parent object graph.

Failures. As in the original system, temporary failures are simply dealt with by retrying. For permanent failures, if a non-gold replica fails, its peers establish connections to other replicas to ensure the connectivity of the graph. If a gold node fails, a non-gold node is promoted to gold and connections are established to maintain the gold node clique.

In total, the implementation of P-Pangaea requires 59 routing rules and 1 blocking predicate.

9.4 Properties of Constructed Systems

We explore the properties of constructed systems in terms of their realism, agility, and performance. In particular, we run experiments to investigate how the systems handle failures, the ease of adding new features for performance benefits, and the absolute performance of constructed systems.

9.4.1 Realism

When building a distributed storage system, a system designer needs to address issues that arise in practical deployments such as configuration options, local crash recovery, distributed crash recovery, and maintaining consistency and durability despite crashes and network failures. PADS makes it easy to tackle these issues for three reasons. First, since the stored events primitive allows routing policies to access local objects, policies can store and retrieve configuration and routing options on-the-fly. For example, in P-TierStore, a node stores in a configuration object the publications it wishes to access. In P-Pangaea, the parent directory object of each object stores the list of nodes from which to fetch the object on a read miss.

Second, for failure and crash recovery, the underlying subscription mechanisms insulate the designer from implementing low-level recovery logic. Upon recovery, local mechanisms first reconstruct local state from persistent logs. Also, PADS's subscription primitives abstract away many challenging details of resynchronizing node state. Notably, these mechanisms track consistency state even across crashes that could introduce gaps in the sequences of invalidations sent between nodes. As a result, crash recovery in most systems simply entails restoring lost subscriptions and letting the underlying mechanisms ensure that the local state reflects any updates that were missed.

Third, blocking predicates greatly simplify maintaining consistency during failures. If there is a failure and the required consistency semantics cannot be guaranteed, the system will simply block access to "unsafe" data. On recovery, once the subscriptions are restored and the predicates are satisfied, the data become accessible again.

For example, in an anti-entropy session in P-Bayou, Node A receives updates from Node B via invalidation and body subscriptions. Every time A applies a received update to its local state, its current logical time *currentVV* advances to include that update. Let's say that the subscription disconnects before all the updates were transferred due to a network failure. Two things must be noted: First, reads of local data at A still guarantee causal consistency—since the *applyUpdateBlock* is set to *isValid*, invalidations are only applied if the corresponding body is received, and since invalidations are applied in causal order, all locally stored objects are valid and causally consistent. Second, recovery is as simple as re-establishing subscriptions to receive updates from another node. A's *currentVV* indicates the latest update in causal order it has received. By setting the subscription start time to *currentVV*, all the updates that A missed will be transferred. The new subscription, in effect, starts from where the previous on left off. Even if A crashes before the anti-entropy session was complete, on recovery, A will recover its *currentVV* from the persistent logs. By establishing subscriptions, it can retrieve the updates it missed.

In P-Coda, even if a client looses the connection to the server, either due to a network partition or due to server crash, it is not necessary to implement special logic to maintain consistency during failures. The blocking conditions will automatically prevent the read of locally invalid data by blocking the read until the connection to the server is re-established and a valid version of the object is retrieved.

In each of the PADS systems we constructed, we implemented support for these practical concerns. Due to space limitations we focus this discussion on the behaviour of two systems under failure: the full featured client-server system (P-FCS) and TierStore (P-TierStore). Both are client-server systems, but they have very different consistency guarantees. We demonstrate that the systems are able to provide their corresponding consistency guarantees despite failures.

Consistency, durability, and crash recovery in P-FCS and P-TierStore. Our experiment uses one server and two clients. To highlight the interactions, we add a 50ms delay on the network links between the clients and the server. Client C1 repeatedly reads an object and then sleeps for 500ms, and Client C2 repeatedly writes increasing values to the object and sleeps for 2000ms. We plot the start time, finish time, and value of each operation.

Figure 9.1 illustrates behavior of P-FCS under failures. P-FCS guarantees sequential consistency by maintaining per-object callbacks [42], maintaining object leases [32], and blocking the completion of a write until the server has stored the write and invalidated all other client caches. We config-

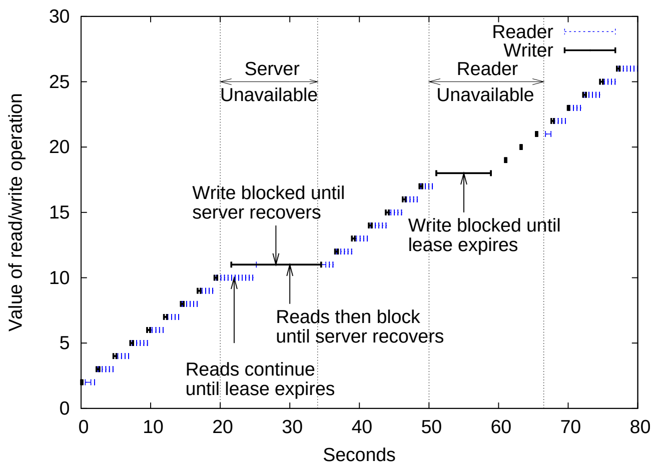

Figure 9.1: Demonstration of full client-server system, P-FCS, under failures. The x-axis shows time and the y-axis shows the value of each read or write operation.

ure the system with a 10 second lease timeout. During the first 20 seconds of the experiment, as the figure indicates, sequential consistency is enforced. We kill (kill -9) the server process 20 seconds into the experiment and restart it 10 seconds later. While the server is down, writes block immediately but reads continue until the lease expires after which reads block as well. When we restart the server, it recovers its local state and resumes processing requests. Both reads and writes resume shortly after the server restarts, and the subscription reestablishment and blocking policy ensure that consistency is maintained.

We kill the reader, C1, at 50 seconds and restart it 15 seconds later. Initially, writes block; but as soon as the lease expires, they proceed. When the reader restarts, reads resume as well.

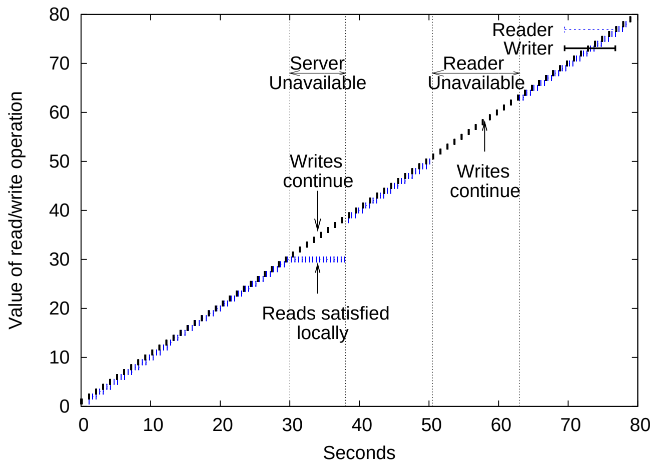

Figure 9.2: Demonstration of TierStore under a workload similar to that in Figure 9.1.

Figure 9.2 illustrates a similar scenario using P-TierStore. P-TierStore enforces monotonic reads coherence rather than sequential consistency, and propagates updates via subscriptions when network is available. As a result, all reads and writes complete locally without blocking during network failures. During periods of no failures, the reader receives updates quickly and reads return recent values. However, if the server is unavailable, writes still progress, and the reads return values that are locally stored even if they are stale.

9.4.2 Agility

A system's requirements also change, as workloads and goals change. We explore how systems built with PADS can be adapted. We highlight two cases in particular: our implementation of Bayou and Coda. Even though they are simple examples, they demonstrate that being able to easily adapt a

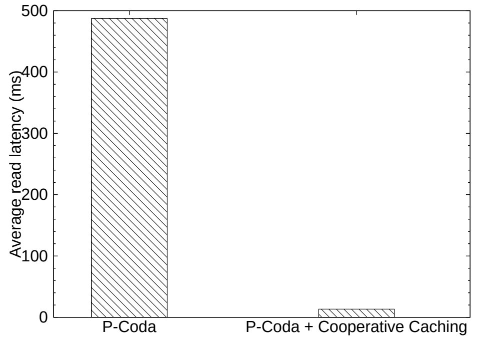

Figure 9.3: Average read latency of P-Coda and P-Coda with cooperative caching.

system to send the right data along the right paths can pay big dividends.

P-Bayou and small device enhancement. P-Bayou was extended to support small devices by a simple change of two routing rules. The results are consistent with that depicted in Figure 8.2. If a node requires only a fraction (e.g., say 10%) of the data, the *small-device enhancement* allows a node to synchronize only the required subset of data thereby reducing the bandwidth required for anti-entropy.

P-Coda and cooperative caching. In P-Coda, support for cooperative caching was implemented by adding only 8 rules. Figure 9.3 depicts the difference in read latencies for misses on a 1KB file with and without support for cooperative caching. For the experiment, the round-trip latency between the two clients is 10ms, whereas the round-trip latency between a client and server is almost 500ms. When data can be retrieved from a nearby client, read

	1KB objects		**100KB objects**	
	Coda	**P-Coda**	**Coda**	**P-Coda**
Cold read	1.51	4.95 *(3.28)*	11.65	9.10 *(0.78)*
Hot read	0.15	0.23 *(1.53)*	0.38	0.43 *(1.13)*
Connected Write	36.07	47.21 *(1.31)*	49.64	54.75 *(1.10)*
Disconnected Write	17.2	15.50 *(0.88)*	18.56	20.48 *(1.10)*

Table 9.1: Read and write latencies in milliseconds for Coda and P-Coda. The numbers in parentheses indicate factors of overhead. The values are averages of 5 runs.

performance is greatly improved. More importantly, with this new capability, clients can share data even when disconnected from the server.

9.4.3 Absolute Performance

Our goal is to provide sufficient performance to be useful. We compare the performance of a hand-crafted implementation of a system (Coda) that has been in production use for over a decade and a PADS implementation of the same system (P-Coda). We expect P-Coda to pay some overheads over Coda because PADS is a relatively untuned prototype rather than well-tuned production code.

Table 9.1 compares the client-side read and write latencies of Coda and P-Coda. The systems are set up in a two client configuration. To measure the read latencies, client C1 has a collection of 1,000 objects and Client C2 has none. For cold reads, Client C2 randomly selects 100 objects to read. Each read fetches the object from the server and establishes a callback for the object. C2 re-reads those objects to measure the hot-read latency. To measure the connected write latency, both C1 and C2 initially store the same collection of 1,000 objects. C2 selects 100 objects to write. The write will cause the server to store the update and break a callback with C1 before the write completes at C2. Disconnected writes are measured by disconnecting C2 from the server and writing to 100 randomly selected objects.

The performance of PADS's implementation is comparable to the original system in most cases and is at most 3.3 times worse in the worst case we measured.

9.5 Summary

This section evaluates the benefits of PADS as a development platform for distributed storage systems. First, by constructing 8 very different systems, this section proves that the primitives provided by PADS are sufficient to implement a wide range of systems. Second, the systems were easy to build—each required only a couple dozens of routing rules and a few blocking conditions; each requiring only a couple of weeks of development time. Third, despite the conciseness of the specification, each system was concrete in the sense that it could handle handle real-world issues such as recovery from failures and configuration options. Fourth, the systems could be easily adapted to new environments by changing just a handful of rules. Lastly, the performance of constructed systems was reasonable—in the case of P-Coda, read/write performance is within 50% of the original Coda system and three times worse in the worst case we measured. This evaluation has led us to the conclusion that PADS does indeed achieve the goal set out by this dissertation—to make development of new distributed replication systems easier.

Chapter 10

Related Work

What sets PADS apart is the flexibility of its replication protocol, its use of declarative specification, and its efficient conflict detection mechanisms. In this chapter, we describe other replication protocols and the limits of their flexibility, other applications for declarative domain specific languages, and other approaches for conflict detection.

10.1 Replication Protocols

The ability of PADS to support a wide range of systems stems from the bookkeeping information maintained at each node and the update transfer protocol. Every node is associated with an interest set—the set of data the node replicates locally and is interested in receiving updates about. In PADS, instead of maintaining data and meta-data pertaining to its interest set only, a node also maintains information about updates to objects not belonging to its interest set. This information is summarized as logical gaps in its causal update log. In addition, when a node transfers updates to another node, it transmits summaries of updates to objects not in the receiver's interest set as logical gap markers. This extra information allows a node maintain consistency invariants despite topology independence (TI) and partial replication (PR). With these invariants, various consistency semantics (AC) can be implemented. Other replication protocols do not send or maintain such information, and therefore, cannot simultaneously provide all three PR-AC-TI properties.

Client-server and hierarchical systems. Client-server systems like Sprite [65] and Coda [49] and hierarchical systems like TierStore [26] and hierarchical AFS [42] are able support a range of consistency semantics (AC), from monotonic read coherence to linearizability. They assume that there is a primary node (such as a central server) that maintains the full repository of data and other nodes replicate subsets of the data (PR). However, these systems allow updates to propagate via fixed paths only (i.e. along hierarchical connections). Even if some systems support co-operative caching, control messages propagate via the fixed topology.

Even though the systems support partial replication, the systems require that a child node's interest set is a subset of its parent's. Because of the restricted communication paths and replication options, every node is able track the interest sets of its children and only transmits updates and metadata pertaining to their interest sets. On the contrary, when these systems are implemented on PADS, extra information (i.e. logical gap markers) for updates not pertaining to the interest set is also sent down the tree.

Server-replication systems. The main requirements for server-replication systems is strong consistency (or at least causal) (AC) and high availability. Some server-replication systems like Replicated Dictionary [90] and Bayou [69] allow any node to send updates to any other node (TI), whereas others like Chain Replication [88] define specific paths for update propagation. These systems fundamentally assume full replication: all nodes store all data from any volume they export and all nodes receive all updates. Because of full replication, the required consistency semantics can be efficiently implemented. For example, in Bayou, a log-based update propagation protocol, per-object time stamps, and a single version vector are sufficient to order all received updates to enforce causal consistency. If Bayou's protocol is used in a scenario with partial replication and ad-hoc communication patterns, the bookkeeping information would not be sufficient, and the system would fail to meet its consistency requirements. On the other hand, PADS sends and maintains extra information that allows a node to order all updates and enforce its consistency invariants.

Object replication systems. Object replication systems like Ficus [36], Pangaea [77], and WinFS [67] allow nodes to store arbitrary subsets of data (PR) and support arbitrary transmission patterns (TI). Unfortunately, the order in which updates to different objects are sent from one node to another is not guaranteed. Therefore, these systems support state-based rather than log-based transfer and can only provide weak consistency guarantees such as monotonic-read coherence. On the other hand, PADS transmits updates in causal order with logical gap markers for missing updates. In addition, detecting concurrent updates to a single object requires more per-object bookkeeping to be maintained (refer to Section 10.3). PADS takes advantage of the update transfer ordering to keep the information maintained for an object to a minimum.

Some systems, such as Cimbiosys [70], distribute data among nodes not based on object identifiers or file names, but rather on content-based filters. We see no fundamental barriers to incorporating filters in PADS to identify sets of related objects. This would allow system designers to set up subscriptions and maintain consistency state in terms of filters rather than object-name prefixes.

DHT-based systems. DHT-based storage systems such as BH [87], PAST [76], and CFS [21], allow different nodes to store subsets of different data (PR). However, they implement a specific update propagation topology and replication policy that makes it difficult to maintain update ordering information across objects. Therefore, they only provide weak per-object coherence guarantees.

Tunable systems. Systems like the TACT toolkit [43] and Swarm [85] provide applications with a menu of options for consistency and dynamically initiate update propagation (or reconciliations) in order to meet the specified consistency levels. However, in order to support the range of consistency levels, they impose restrictions either on replication or topology. For example, the TACT toolkit only supports full replication and Swarm assumes hierarchical topology in addition to full replication. On the contrary, PADS is able to support the whole range of TACT consistency dimensions without imposing the replication and topology restrictions. Fluid replication [20] also exposes a range of consistency options. It dynamically creates replicas and carries out reconciliations to meet performance and consistency needs. However, it only supports hierarchical topology. In fact, like PADS, fluid replication employs update summaries to to quickly transfer update inform and establish update ordering at the server. However, unlike PADS, these summaries are only sent from waystations to the server, rather than on all communication links.

Other systems like Deceit [80], WheelFS [84], and Dynamo [25], provide applications with a wide range of options to control the level of replication, placement of replicas, and the sizes of read/write quorums for a given topology. Because every object may have a different replication policy, it is difficult to support cross-object consistency. Instead, these systems provide a range of single-object guarantees, from one-copy serializability to maximum staleness to eventual consistency, that can be specified with the provided options.

10.2 Declarative Domain Specific Languages

R/OverLog falls in the same category of declarative domain specific languages (DSL) such as SQL [3], HTML [1], OverLog [56], and SQCK [35], that have been used to aid development. The common benefit of these languages is that they allow designers to specify high-level intent rather than implement low-level details (i.e. *what* to do rather than *how* to do). For example, the SQL `select` statement describes what data is required and not how it should be retrieved, an HTML file describes what the interface should look like and not how it should be rendered, and an OverLog program describes what data flows should be established but does not explicitly establish them. Similarly, an R/OverLog policy describes what update paths should be established but does not implement the update propagation protocol.

These languages rely on an underlying layer that takes care of the low-level details required to implement the high-level intent. A high-level specification has multiple benefits. First, it leads to concise programs making them easy to maintain and modify. Second, it is portable in the sense that the program is isolated from specific implementations and optimizations of the underlying layer.

A common criticism for R/OverLog is the steep learning curve. However, as the popularity of SQL and HTML indicate, a declarative DSL can be the better approach for some tasks. R/OverLog demonstrates the feasibility of a declarative approach for the first generation of the systems. Perhaps in the next generation, effort should be spent on developing a more user-friendly syntax to further aid development.

On a different note, Mace [48]/Macedon [74] is another event driven language for setting up network overlays. It uses the abstraction of finite state machine for describing overlays. We do not use Mace in PADS because we worry about the number of states that need to be defined for a node in order to handle failures and recovery of multiple subscriptions and for various consistency policies. However, given that Ramasubramanian et. al. [70] have used Mace to implement the Cimbiosys distributed file system, it would be interesting to compare the ease of defining PADS policy with Mace and with R/OverLog.

10.3 Conflict Detection

PADS focuses on syntactic conflict detection based on the causal relationship [52] rather than relying on any application-specific semantics. In particular, any two updates to the same object are not causally related are considered to be *conflicting*.

Current schemes to detect conflicts fall into three main families:

- *Previous stamps.* In this approach [16, 33], whenever an object is overwritten, the time stamp of the previous version *previous stamp* is also stored. When the update is propagated to another node, the *previous stamp* is also sent with the *write stamp* of the update. When a node receives an update, it compares the *previous stamp* of the received update with the *write stamp* of the locally stored version. If they mismatch, then the update is marked as a conflict. This approach can accurately detect all conflicts in any log exchange protocol that ensures the prefix property [22], but it adds an extra per-update overhead for both storage and network bandwidth. More importantly, in the case when the log is truncated and a node falls back to state-based exchange, false positives are possible due to missing intermediary updates.

- *Hash histories.* Kang et. al. [45] use *hash histories* to detect conflicts. Whenever an update is locally applied, a node creates a new hash summarizing the entire current state. Each node keeps a list of hashes ordered by when they were generated. Whenever a node A synchronizes its state with another node B, it looks up B's last hash in its own hash history. If the hash exists, then A's version is a newer version. Similarly, if B finds A's last hash in B's hash history, then B's version is a newer version. If neither of the last hashes exists in the other's history, then the system marks the synchronization as conflicting. Although the size of hash history is independent of the number of replicas, it grows proportionally to the total number of updates. More importantly, because hashes summarize the entire local state, false positives are can occur when different objects are concurrently updated.

- *Version vectors.* Many systems [37, 49, 71, 77] use *version vectors* to detect conflicts. A version vector [44] accurately captures the causality relationship between two 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 a version vector for every object, especially in large-scale systems.

Predecessor vectors with exceptions (PVE) [59] and *vector sets* [58] are variations of the *version vectors* approach employed by WinFS [67] designed to reduce the total number of version vectors maintained and communicated. PVEs can reach an unbounded size if synchronizations are frequently disrupted, making them unsuitable for environments with intermittent connections. Vector sets maintain predecessor vectors for subsets of data, and in the worst case, have overheads equivalent to a simple version vector scheme. PADS's dependency summary vectors (DSV) are actually equivalent to predecessor vectors. However, due to the properties of the synchronization protocol, instead of storing DSVs, explicitly in a data structure, PADS derives them from the consistency meta-data already stored. In addition, the metadata stored and sent during synchronization does not increase with network disruptions. Chapter 8 evaluates the overheads of PADS and demonstrates that PADS's overheads are equivalent to other state-of-the-art schemes.

Chapter 11

Limits and Future Work

PADS's flexibility sufficient covers a broad range of systems. However, because of the small API, there are several design aspects and trade-offs it does not expose to designers. This chapter discusses the limits of the PADS architecture and the avenues of the future work that can be explored.

11.1 Limits

The limits of PADS can be roughly divided into three categories:

- *Limits imposed by ther model itself:* There are several high-level design aspects, such as security and transactions, that are not covered by the routing and blocking abstractions.

- *Limited extensibility of the implementation:* The mechanisms layer provides a default implementation of PADS primitives. Even though the default implementation is generally applicable, some parts may need to be reimplemented to build systems that are better suited for the target requirements.

- *Limited parameters for performance tuning:* The mechanisms layer provides limited options to tweak current primitive implementation leading to non-optimal system performance.

In this section, we discuss each category in turn.

11.1.1 PADS Model

The routing and blocking abstractions exposed by PADS do not cover the following aspects that play an important role in distributed storage system design:

Security. An important part of distributed data storage system design defines the security guarantees a system provides. Security guarantees cover the following properties:

- *Data access control:* ensures that data is accessible only to authorized users.
- *Data integrity:* ensures that any malicious or accidental altering of data can be detected.
- *Data privacy:* ensures that no information (such as the amount of data stored or pattern of access) is exposed to unauthorized users.

Different systems provide different levels of guarantees for each security property. In order to enforce these guarantees, a wide range of technologies, such as encryption, secure channels, access control lists and trusted hardware, are used.

Security specification does not match the blocking and routing abstractions provided by the current PADS model. We should consider augmenting the PADS architecture to allow security specification as a separate policy component. We suspect that the security abstraction would allow designers to address three aspects:

- *Basic access control:* It seems plausible that straightforward use of encryption and signatures could secure reads and writes so that one is only able to read and write an object if one has the appropriate credentials.
- *Flexible policy hooks:* It seems plausible to add something similar to the five blocking points to manage encryption and validation decisions.
- *Consistency:* In addition to enforcing security on reads and writes, securing on ordering is also needed. It may be plausible to apply the content-hash based protocol describe by Mahajan et. al. [57] for that purpose.

Distributed transactions. The current PADS model is sufficient for a small subset of distributed transactions. For example, for a single update, two-phase commit can be implemented by writing appropriate routing policy and blocking policy and using the *assign seq* action to commit updates (refer to Section 5.4.7). However, for transactions that involve reads and writes to multiple objects stored across multiple nodes, the PADS model falls short. Applications have no means to specify what reads and writes make up a transaction. Also, because only a single version of an object is maintained, if the transaction that created the updated is aborted, the mechanisms do not support rollback to a previously committed version.

We suspect that support for distributed transactions can be easily added by implementing a tentative buffer that stores intermediate results and a primitive that moves tentative updates to stable store once the transaction is committed. We would also require additional triggers that provide information about the propagation of tentative updates so that appropriate commit policies can be implemented.

Erasure coding. PADS provides data availability by employing data replication rather than erasure coding. Even though PADS objects cannot be erasure-coded, PADS can be used to replicate erasure-coded fragments. A library can be implemented at the read/write interface that translates a file into fragments and stores each fragment a separate PADS object. The routing policy and the blocking policy can be used to specify how the fragments are replicated.

Quorums. Some distributed storage systems [25] use quorums to provide consistency guarantees. Write quorums can be easily implemented with current blocking and routing abstractions—every node sends an ACK to the original writer when it receives an update and the *writeAfterBlock* predicate is set to unblock if sufficient ACKS have been received. However, implementing a read quorum with PADS is complex. It involves implementing a library over the read/write interface that enforces the read quorum—when an object is read, out-of-band communication is used to retrieve data from other nodes, consolidate the responses, and return the latest version. It would good to allow designers to easily specify read and write quorums.

11.1.2 Extensibility

The implementation of PADS makes specific choices that work well for a broad range of environments. These choices are not fundamental to the PADS architecture but have implications for performance, resource, and availability guarantees of the system. Sometimes a designer may want to change specific aspects of the system in order to optimize the system for the target environment. The aspects can include:

- *Local storage:* In the current implementation, BerkeleyDB is used as the back-end for the persistent update log and the in-memory data store. Designers may want to control how data is locally stored for performance reasons, such as allowing objects to be stored as fixed-size chunks rather than as variable-sized byte-ranges, compressing stored bodies, implementing an in-memory-only log, customizing when the log or store is synchronized to disk, or using a different back-end for storing data.

- *Replacement protocols:* Currently, objects in the store are never replaced. Designers may want to implement cache replacement protocols (such as LRU or priority-based schemes) to control the resources taken up by the storage.

- *Network protocol:* Currently, all communication occur over TCP connections. Designers may want to use connections for data propagation and policy communication that is best suited for their needs, such as TCP-Nice connections, UDP connections, secure connections, delay-tolerant connections, and pluggable protocols.

- *Namespace:* PADS imposes a hierarchical namespace on objects. For example, the string */a/** covers all objects with the prefix */a/* such as */a/x* and */a/y*. This hierarchy is used as a basis for subscription establishment and for specifying imprecise invalidation target sets. The hierarchical namespace is well suited for implementing systems that use the file-name abstraction for access. However, increasingly tags or filters [70] are being used to access data and the hierarchical namespace is not efficient for filter-based access. Designers may want to modify how the object namespace is defined so that alternate access schemes can be efficiently implemented. For example, by using bloom filters, a group of unrelated objects can be concisely summarized.

- *Conflict resolution:* Current implementation uses the last-writer-wins scheme to resolve conflicting updates (i.e. for two conflicting updates, one of them will overwrite the other). With this scheme, one of the updates is lost which may not be acceptable for all applications. Some applications require that the final version of the object consolidate both updates in a manner that may be application-dependent. For example, an object that represents a directory object in a file system will be consolidated differently from an object that represents a shopping-cart object [25].

Currently, if a designer wants to change a certain aspect of the implementation, she has to make that modification by hand. The PADS implementation is sufficiently modular—changing one of the above aspects requires updates to only a small part of the system. One can imagine making all this extensible—by defining interfaces so that designers can pick or plug in alternative implementations.

11.1.3 Current Implementation Limits

In addition to providing a single standard implementation of the primitives, the current mechanisms layer also hides options that if exposed, would lead to more efficient implementations. These options include:

- *Scheduling of outgoing updates:* Currently, if an invalidation subscription and a body subscription are established from Node A to Node B, whenever an update occurs at Node A, the invalidation and the body of the update are sent immediately to Node B. If the timing of when the updates are sent can be controlled, then techniques to improve performance can be incorporated [64]. These techniques include removing updates pertaining to temporary objects, and batching updates so that for quick consecutive updates to the same file, only a single body is sent.

- *Splitting and joining interest-sets:* Currently, consistency-related meta-data such as preciseness is maintained on a per interest set basis. The meta-data is stored as a version vector in a hierarchical structure, called *ISSTATUS*, in which each node corresponds to an interest set. Depending on the subscriptions established and the imprecise invalidations received, the hierarchical structure may be split to store version vectors associated with smaller interest sets. For example, assuming that initially the *ISSTATUS* stores a version vector for */a/**. If an incoming subscription is established for the object */a/x*, the interest set in the *ISSTATUS* is split so that it stores 2 version vectors—one for */a/x* and another version vector for all other objects in */a/**. If another incoming subscription is established for */a/y*, then the interest set is split further to store a version vector for */a/y* separately. If many such subscriptions are established, it is possible that over the course of time, a version vector is stored for every local object. By implementing the capability of joining interest sets and allowing designers to specify the policy for splitting and joining interest sets, designers gain fine grained control over how meta-data is maintained and possibly lead to storage savings.

- *Generation of imprecise invalidations:* Currently, for an outgoing subscription, an imprecise invalidation is generated for one of two reasons: a precise invalidation needs to be sent on the subscription or the time since the last precise invalidation was sent has reached the threshold time (i.e. a time out). In the first case, the imprecise invalidation summarizes *all* updates that have occurred to objects not in the subscription set since the last precise invalidation sent. In the second case, the imprecise invalidation summarizes *all* updates that have occurred to objects not in the subscription set since the last precise invalidation sent up to the current time.

  Since current generation policy aims to be conservative and to achieve good compression, the granularity of the imprecise invalidation may be too big and make a larger-than-necessary portion of the receiver's objects imprecise. For example, in order to summarize updates to */a/x/p* and to */a/y/q*, an imprecise invalidation for */a/** may be generated. This invalidation will make the whole */a/** subtree imprecise and possibly making all objects in */a/** unavailable for access. By exposing the policy for the generation of imprecise invalidations, the granularity of the imprecise invalidations can be controlled in order to improve the overall data availability.

11.2 Future Work

This dissertation opens up several avenues for future work that can be important improvements to PADS as a development platform.

11.2.1 Streamlined Implementation

The performance achieved by PADS is not sufficient for demanding workloads. We suspect that the following aspects of the mechanisms layer can be redesigned or reimplemented to improve the performance.

- Instead of allowing updates to cover variable-sized byte-ranges of an object, we should consider restricting updates to fixed-size chunks, like the NFS protocol. Because different parts of an object are stored as variable-sized chunks corresponding to different byte-ranges, extra complexity is introduced for storage, body look-up, and meta-data maintenance. For example, when an update is received, the update may overlap multiple chunks but partially overlap some chunks. Extra processing is needed to detect overlap boundary, consolidate the chunks and update the meta-data for each chunk. By storing only fixed size chunks, much complexity can be reduced.

- The data store currently a big bottleneck for local read/write performance. The back-end of the data store is BerkeleyDB. Even though, the BerkeleyDB environment is configured to asynchronously sync to disk, BerkeleyDB carries out a lot of processing under-the-hood, such as log garbage collection, transaction locks, etc. We may want to consider implementing a custom in-memory store that can give us better control over the performance.

- We may want to redesign how meta-stored is stored and how it is updated when an invalidation is received. Currently, when an invalidation is locally applied, it requires updating object meta-data, the meta-data associated with the byte-range, and clearing up invalid bytes. This operation involves several database accesses and can be pretty slow if multiple threads are updating the meta-data. Perhaps performance can be improved by implementing in-memory meta-data maintenance and finer-grained locks for meta-data and object stores.

- The current implementation of outgoing body subscriptions is inefficient. Whenever an outgoing body subscription is established, every byte-range in the object store is iteratively scanned in order to look for bodies that belong to the subscription set and that are newer than the subscription start-time. If the store contains a lot of objects, this scan can take a very long time. It is plausible that by introducing secondary keys or by maintaining extra bookkeeping at the object level, the time required for subscription establishment can be reduced.

- The mechanisms layer maintains several buffers that have no clean up policy allowing them to increase in size indefinitely. These include the body buffer that keeps bodies that arrived before their corresponding invalidations, and the object store that does not implement any object replacement policy. It will be beneficial spend some effort to track down such buffers and implement appropriate clean-up policies.

- We should consider implementing protocols to summarize version vectors. Since meta-data is maintained in terms of version vectors, the overheads are proportional to the number of nodes in the system. For environments that involve a large number of nodes, such as clusters and data centers, PADS imposes unacceptable overheads.

11.2.2 Formal Model and Verification

Designs and implementations of distributed data storage systems are complex and proving that they are correct can be difficult. In fact, ensuring that a system meets its consistency guarantees is further complicated by concurrent updates and failures.

With PADS, because a system is described with a high-level language over a small set of abstractions, we can reason about it more easily. It is plausible that we can take advantage of the abstractions provided by PADS to develop a formal model that can allow designers to check or prove high-level properties of their systems. The model should allow designers to specify, check, and/or verify both the safety and the liveness properties of systems (i.e. whether consistency guarantees are satisfied or whether the implementation is free from deadlocks).

One approach to verify a system is to create a model of the system, verify the model and generate an executable/deployable system from the model. This approach requires writing the PADS architecture, the system design, and the system constraints in a modeling language such as Murϕ [27], Promela [2], or MACE [48]. A model checker is run to ensure that the design meets the specified constraints. Once the model is verified, the model is translated so that it can be executed. There are two options for translation. First, the translator can generate R/OverLog code and blocking conditions that work with the current PADS implementation. Alternatively, the model can be translated directly into C++ or Java code. This option is relevant if we were to use MACE as the modeling language which directly generates C++ code. However, its state-machine abstraction is does not easily match current PADS abstractions. The implications of this option need to be further examined since it may require a rewrite of the underlying framework.

Another possible approach is to do just the opposite. Instead of specifying a system in a modeling language, the system is written in terms of R/OverLog rules and blocking conditions and is then either directly checked or is translated into a modeling language that can be checked.

Whatever the approach, the main challenge is to define a constraint language so that consistency, availability, performance and durability constraints of a distributed storage system can be easily written and verified. Pip [72] uses a declarative language for specifying expected behaviour of systems. Its declarative nature makes it an attractive candidate for specifying constraints in PADS. Its applicability in the context of distributed storage systems requires further study.

On a different note, some checkers, such as CystalBall [91], MaceODB [24], and Pip, verify that a system is running correctly during execution. The advantage of this approach is that in addition to safety and liveness properties, performance and resource constraints can also be specified.

11.2.3 Self-tuning File System

Current technology trends indicate that the amount of data users generate and access on a daily basis is on an increase. In addition, users have access to a wide range of devices including mobile phones, laptops, netbooks, ebooks, portable music players. Because multiple copies of a user's data may be spread over multiple devices, data management becomes a major headache. For an average user, locating latest versions of data, consolidating divergent copies, and ensuring that the required data is always available is no easy feat.

In fact, with most devices being equipped with multiple networking technologies, such as WIFI, Bluetooth, and 3G connections, and the almost-always availability of cloud storage, it should be possible to implement a distributed file system that transparently manages user data, enabling users to access the latest version of their data from any device without any effort. In addition, the file system should be self-tuning, i.e. it should be able take into account available Internet and peer connectivity, costs for cloud storage, usage patterns and available battery life in order to to propagate data along paths that yield the best monetary costs, power, or performance tradeoffs.

Our preliminary design for such a file system is based on a client-server scheme with support for peer-to-peer transmissions. Because of its availability, cloud storage is best suited to act as the server for meta-data and a large portion of recently accessed data. Updates are propagated based on an eager-meta-data-lazy-data scheme—whenever an update occurs, the meta-data is quickly propagated to the cloud whereas the data of the update is sent if the opportunity is appropriate. The advantage of this approach is that the eager meta-data propagation makes it easy to keep track of where the latest version of an object is, and the lazy data propagation makes it possible to delay sending the data if the network conditions are not good (i.e. either the network is too slow or not energy-efficient). As the first step, we have designed a policy that focuses on energy efficiency that can potentially lead to two orders of magnitudes of savings [15]. However, more work needs to be done to build a *hero* file system that self-tunes and ensures that data is accessible most the time.

11.2.4 New Applications

PADS makes it easy to implement and enforce consistency guarantees for distributed data. It would be interesting to explore whether we can apply PADS to domains other that data storage. One domain in which PADS can be useful is structured overlays. Structured peer-to-peer overlays are often used as the underlying substrate for content-dissemination, file sharing and VOIP applications. Most existing overlay protocols [75, 83, 95] focus on the design of overlay topology, reducing maintenance costs, and improving routing performance. They do not focus on enforcing consistency of the routing information stored at each node. The lack of consistency guarantees can result in routing errors especially during failures and churn. By enforcing routing consistency and reducing routing errors, it is possible to provide better overlay performance, availability, and in turn, better QOS guarantees to applications. Unfortunately, enforcing routing consistency is complex [19] and so most systems give up on consistency and implement mechanisms to detect and remove erroneous state instead [83].

We suspect that the complexity of enforcing consistency can be reduced if PADS is used as the infrastructure to maintain and propagate routing state. First, PADS provides high-level abstractions with which consistency semantics can be easily specified. Second, PADS mechanisms ensure that updates are sent in order. Third, with the support of formal verification, it is possible to prove that consistency guarantees are enforced even during failures.

Chapter 12

Conclusion

This dissertation aims to make it easier to build new distributed storage systems. In order to achieve this goal, it presents the PADS approach according to which development is carried out by specifying policy over an underlying mechanisms layer. In order to demonstrate the benefits of this approach, instead of building a single new distributed storage system, this dissertation constructs several systems inspired from literature that cover a broad spectrum of the design space. This chapter summarizes the contributions of this work and the research lessons learned.

12.1 Contributions

From a developer's point of view, using the PADS has several advantages: First, it provides a clean separation of system design into routing policy and blocking policy. Second, it is sufficiently flexible to support a broad range of systems which is demonstrated by the fact that we have built client-server systems, server-replication systems, and object-replication systems with PADS. Third, development on PADS is significantly easier and shorter than building a system from scratch. The systems constructed on PADS required less than a hundred routing rules and a couple of blocking conditions; each required about a few weeks of development time. Fourth, the systems developed on PADS can be easily modified to incorporate new features or address new requirements. Lastly, the overheads associated with the PADS prototype are within small constant factors of hand-tuned systems.

In the development and the evaluation of the PADS approach, this thesis makes the following contributions:

- It demonstrates that the effort required to build distributed storage systems can be significantly reduced by adopting a policy specification-based approach.
- It defines a new dichotomy for distributed storage system design. In particular, system design and implementation is separated into routing policy that addresses performance and availability goals, and blocking policy that addresses consistency and durability concerns.
- It demonstrates that a small API is sufficient to build a broad range of distributed storage systems.
- It defines and provides an implementation of a general set of replication mechanisms that are sufficiently flexible to support all three PR-AC-TI properties.
- It defines a new domain specific language, R/OverLog, that allows concise and elegant specification of routing policy.

- It provides a set of useful prototypes for different distributed storage systems that can be deployed for moderating workloads.

12.2 Research Experience

The most important lesson we learned during the pursuit of this thesis is that a great solution is often the result of well thought out evolution. Similarly, current PADS is the result of 4 stages of evolution.

We started with a basic mechanisms layer which had an extensive API. Unfortunately, we had no concrete way to map the designs of existing systems on to the provided API. We realized that a lot of the design of distributed storage systems is routing of information among nodes. That realization enabled us to develop the subscriptions primitive. We were also able to take advantage of OverLog, and later R/OverLog, to define routing policy.

In the second step of the evolution, we were struggling with how to cleanly define the "rest" of the system design. We realized that the "rest" boils down to consistency and durability guarantees which could be implemented by blocking access to data. At first, we required that systems designers implement libraries over the local interface and only some simple read-block flags were provided. However, after implementing several consistency libraries, we noticed that the enforcement of consistency guarantees often required looking up local consistency bookkeeping and sending or waiting for messages from routing policy. This observation led to the definition of blocking points and blocking predicates.

The next step in the evolution was the definition of the stored events interface. During the development of P-Pangaea, we realized that the routing and blocking API were not sufficient to allow us to persistently store gold node information. We designed the stored event interface so that local data objects can be easily accessed in a manner that works well with the event-driven model for routing policy. The stored events interface also allowed reading and writing configuration information such as server IDs into objects. We previously handled configuration information in ad-hoc ways such as hard-coding it in the routing policy.

Finally, the last step in the evolution was the addition of the commit operation and the *isSequenced* predicate. We found difficulty in implementing the primary-commit protocol due to the lack of ordering guarantees provided by R/OverLog. For example, when a primary commits updates, the commit information may be sent to different nodes in different orders. By introducing the commit operation and by ensuring that commit invalidations are causally transferred, we were able to implement the primary-commit protocol in a clean, concise manner.

Appendices

Appendix A

Code Listings

This appendix provides the code listings for the systems constructed with PADS. In order to make it the R/OverLog code easier to read, we add prefixes to R/OverLog tuples: `ACT` for actions that call into the mechanisms layer, `TRIG` for triggers from the mechanisms layer, `TBL` for table lookups, `RCV` for events received via the stored events interface, and `B_ACT` for actions that go to blocking policy. For the blocking policy, we only list the predicates that are not set to the default "true" value.

A.1 Simple Client Server (P-SCS)

The implementation of P-SCS requires 24 routing rules and 5 blocking conditions.

A.1.1 Routing Policy

```
/**********************************************************************/
//   Initialization:  Read server configuration, and
//   store serverId in a table
/**********************************************************************/

in1  ACT_readEvent(@X, ObjId)  :-
         initialize(@X), ObjId = "/.serverCFG".

in2  TBL_server(@X, S)  :-
         RCV_serverConfig(@X, S).
```

```
 /********************************************************************/
 //   Client establishes subscriptions to
 //   server for transferring updates.
 /********************************************************************/

csSb1 ACT_addInvalSub(@X, X, S, SS, CTP)  :-
          RCV_serverConfig(@X, S), SS="/*", CTP=="LOG", X≠S.

csSb2 ACT_addBodySub(@X, X, S, SS)  :-
          RCV_serverConfig(@X, S), SS="/*", X≠S.

csSb3 ACT_addInvalSub(@X, X, S, SS, CTP)  :-
          TRIG_subEnd(@X, X, S, SS, _, Type), Type=="inval", CTP=="LOG".

csSb4 ACT_addBodySub(@X, X, S, SS)  :-
          TRIG_subEnd(@X, X, S, SS, _, Type), Type=="body".

 /********************************************************************/
 //   On read miss, client store info in a table and informs server.
 /********************************************************************/

 rm1  TBL_readMiss(@C, Obj, Off, Len)  :-
          TRIG_operationBlock(@C, Obj, Off, Len, Bpoint, _),
          Bpoint == "readAt", TBL_serverId(@C, S), C≠S.

 rm2  clientRead(@S, C, Obj, Off, Len)  :-
          TRIG_operationBlock(@C, Obj, Off, Len, Bpoint, _),
          Bpoint == "readAt", TBL_serverId(@C, S), C≠S.

 /********************************************************************/
 //   On client read miss, server sends body and establishes callback
 /********************************************************************/

 cb1  ACT_sendBody(@S, S, C, Obj, Off, Len)  :-
          clientRead(@S, C, Obj, Off, Len).

 cb2  ACT_addInvalSub(@S, S, C, Obj, CTP)  :-
          clientRead(@S, C, Obj, Off, Len), CTP = "CP".

 cb3  TBL_hasCallback(@S, Obj, C) :-
          clientRead(@S, C, Obj, Off, Len),

 /********************************************************************/
 //   When client receives an invalidation, if it does
 //   not satisfy a read miss, ack server and cancel callback.
 //   Otherwise, remove it from the miss table
 /********************************************************************/

rinv0 satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, a_COUNT<*>) :-
          TRIG_invalArrives(@C, S, Obj, Off, Len, Writer, Stamp),
          TBL_serverId(@C, S), S≠C,
          TBL_readMiss(@C, S, Obj, Off, Len).

rinv1 ackServer(@S, C, Obj, Off, Len, Writer, Stamp)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count==0,

rinv2 ACT_removeInvalSub(@C, S, C, Obj)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count==0,

rinv3 delete TBL_readMiss(@C, S, Obj, Off, Len)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count > 1.
 /*********************************************************************/
 //   When server receives an invalidation,
 //   gather acks from others who have callbacks.
 //   Note:
 //   - Acks are accumulative so one received ack also
 //   satisfies all previous required acks.
 //   - needAck table keeps track of which acks are needed.
 /*********************************************************************/

ack1  TBL_needAck(@S, Obj, Off, Len, C2, Writer, Stamp, Need)  :-
          TRIG_invalArrives(@S, C, Obj, Off, Len, Writer, Stamp),
          TBL_hasCallback(@S, Obj, C2), C2 ≠ Writer,
          Need = 1, TBL_serverId(@S, S).

ack2  TBL_needAck(@S, Obj, Off, Len, C2, Writer, Stamp, Need)  :-
          TRIG_invalArrives(@S, C, Obj, Off, Len, Stamp, Writer),
          C2 == Writer, Need = 0, TBL_serverId(@S, S).

ack3  TBL_needAck(@S, Obj, Off, Len, C, Writer, NeedStamp, Need)  :-
          ackServer(@S, C, _, _, _, _, RecvStamp),
          TB_needAck(@S, Obj, Off, Len, C, Writer, NeedStamp, _),
          NeedStamp < RecvStamp, Need= 0, TBL_serverId(@S, S).

ack4  TBL_needAck(@S, Obj, C, Writer, NeedStamp, Need)  :-
          ackServer(@S, C, _, _, _, RecvWriter, RecvStamp),
          TBL_needAck(@S, Obj, C, NeedWriter, NeedStamp, _),
          NeedStamp == RecvStamp, NeedWriter <= RecvWriter,
          Need= 0, TBL_serverId(@S, S).

ack5  delete TBL_hasCallback(@S, Obj, C)  :-
          TBL_needAck@S, Obj, _, _, C, _, _, Need)
          Need = 0, TBL_serverId(@S, S).

ack6  acksNeeded(@S, Obj, Off, Len, Writer, Recvtamp, a_COUNT<*>) :-
          TBL_needAck(@S, Obj, Off, Len, C, Writer, RecvStamp, NeedTrig),
          TBL_needAck(@S, Obj, Off, Len, C, Writer, RecvStamp, NeedCount),
          NeedTrig == 0, NeedCount == 1.

 /*********************************************************************/
 //   If no more acks are needed, assign Sequence number,
 //   inform client, and clear out ack table
 /*********************************************************************/

ack7  ACT_assignSeq(@S, Obj, Off, Len, Stamp, Writer)  :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.

ack8  BACT_writeComplete(@Writer, Obj)  :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.

ack9  delete TBL_needACK(@S, Obj, C, WriterId, RecvStamp, Need) :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.
```

A.1.2 Blocking Policy

Blocking Point	**Predicate**
ReadNowBlock	isValid and isComplete and isSequenced (at Client)
WriteAfterBlock	B_Action(writeComplete, objId) (at Client)
ApplyUpdateBlock	isValid (at Client & Server)

A.2 Full Client Server (P-FCS)

The implementation of P-FCS requires 43 routing rules and 6 blocking conditions.

A.2.1 Routing Policy

```
 /*******************************************************************/
 //   Initialization:  Read server configuration, and
 //   store serverId in a table
 /*******************************************************************/

 in1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId = "/.serverCFG".

 in2  TBL_server(@X, S)  :-
          RCV_serverConfig(@X, S).

 /*******************************************************************/
 //   Client establishes subscriptions to
 //   server for transferring updates.
 /*******************************************************************/

csSb1 ACT_addInvalSub(@X, X, S, SS, CTP)  :-
          RCV_serverConfig(@X, S), SS="/*", CTP=="LOG", X≠S.

csSb2 ACT_addBodySub(@X, X, S, SS)  :-
          RCV_serverConfig(@X, S), SS="/*", X≠S.

csSb3 ACT_addInvalSub(@X, X, S, SS, CTP)  :-
          TRIG_subEnd(@X, X, S, SS, _, Type), Type=="inval", CTP=="LOG".

csSb4 ACT_addBodySub(@X, X, S, SS)  :-
          TRIG_subEnd(@X, X, S, SS, _, Type), Type=="body".

 /*******************************************************************/
 //   On read miss, client stores info in a table and informs server.
 /*******************************************************************/

 rm1  TBL_readMiss(@C, Obj, Off, Len)  :-
          TRIG_operationBlock(@C, Obj, Off, Len, Bpoint, _),
          Bpoint == "readAt", TBL_serverId(@C, S), C≠S.

 rm2  clientRead(@S, C, Obj, Off, Len)  :-
          TRIG_operationBlock(@C, Obj, Off, Len, Bpoint, _),
          Bpoint == "readAt", TBL_serverId(@C, S), C≠S.

 /*******************************************************************/
 //   If peer fails to send body, ask server to send body.
 /*******************************************************************/

 cc1  ACT_sendBody(@S, S, C, Obj, Off, Len)  :-
          TRIG_sendBodyFailed(@C, _, C, Obj, Off, Len, _, _).

 /*******************************************************************/
 //   On client read miss, server establishes callback and finds
 //   another client with callbacks to send the body.
 /*******************************************************************/
 cb1  ACT_addInvalSub(@S, S, C, Obj, CTP)  :-
           clientRead(@S, C, Obj, Off, Len), CTP = "CP".

 cb2  TBL_hasCallback(@S, Obj, C, T) :-
           clientRead(@S, C, Obj, Off, Len), T=f_now().

 cc2  anyPeerAvailable(@S, C, Obj, Off, Len, a_Count<*>)  :-
           clientRead(@S, C, Obj, Off, Len),
           TBL_hasCallback(@S, Obj, C2, _).

 cc3  selectedPeer(@S, C, Obj, Off, Len, a_RANDOM<C2>)  :-
           anyPeerAvailable (@S, C, Obj, Off, Len, Count), Count > 0.
           TBL_hasCallback(@S, Obj, C2, _).

 cc4  ACT_sendBody(@SP, SP, C, Obj, Off, Len, _, _)  :-
           seletedPeer(@S, C, obj, Off, Len, SP).

 cc5  ACT_sendBody(@S, S, C, Obj, Off, Len) :-
           anyPeerAvailable (@S, C, Obj, Off, Len, Count), Count==0.

  /*********************************************************************/
  //   When client receives an invalidation,
  //   checks if it satisfies a read miss.  If so,
  //   removes it from the table.  If not, ACK server.
  /*********************************************************************/

rinv0 satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, a_COUNT<*>) :-
           TRIG_InvalArrives(@C, S, Obj, Off, Len, Writer, Stamp),
           TBL_serverId(@C, S), S≠C, TBL_readMiss(@C, S, Obj, Off, Len).

rinv1 ackServer(@S, C, Obj, Off, Len, Writer, Stamp)  :-
           satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count==0,

rinv2 delete TBL_readMiss(@C, S, Obj, Off, Len)  :-
           satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count > 1.

  /*********************************************************************/
  //   When server receives an invalidation,
  //   gather acks from others who have callbacks.
  //   Note:
  //   - Acks are accumulative so one received ack also
  //   satisfies all previous required acks.
  //   - needAck table keeps track of which acks are needed.
  /*********************************************************************/

 ack1  TBL_needAck(@S, Obj, Off, Len, C2, Writer, Stamp, Need)  :-
           TRIG_invalArrives(@S, C, Obj, Off, Len, Writer, Stamp),
           TBL_hasCallback(@S, Obj, C2, _), C2 ≠ Writer,
           Need = 1, TBL_serverId(@S, S).

 ack2  TBL_needAck(@S, Obj, Off, Len, C2, Writer, Stamp, Need)  :-
           TRIG_invalArrives(@S, C, Obj, Off, Len, Stamp, Writer),
           C2 == Writer, Need = 0, TBL_serverId(@S, S).

 ack3  TBL_needAck(@S, Obj, Off, Len, C, Writer, NeedStamp, Need)  :-
           ackServer(@S, C, _, _, _, _, RecvStamp),
           TB_needAck(@S, Obj, Off, Len, C, Writer, NeedStamp, _),
           NeedStamp < RecvStamp, Need= 0, TBL_serverId(@S, S).
```

```
ack4  TBL_needAck(@S, Obj, C, Writer, NeedStamp, Need)  :-
          ackServer(@S, C, _, _, _, RecvWriter, RecvStamp),
          TBL_needAck(@S, Obj, C, NeedWriter, NeedStamp, _),
          NeedStamp == RecvStamp, NeedWriter <= RecvWriter,
          Need= 0, TBL_serverId(@S, S).

ack5  delete TBL_hasCallback(@S, Obj, C, _)  :-
          TBL_needAck@S, Obj, _, _, C, _, _, Need)
          Need = 0, TBL_serverId(@S, S).

ack6  acksNeeded(@S, Obj, Off, Len, Writer, Recvtamp, a_COUNT<*>) :-
          TBL_needAck(@S, Obj, Off, Len, C, Writer, RecvStamp, NeedTrig),
          TBL_needAck(@S, Obj, Off, Len, C, Writer, RecvStamp, NeedCount),
          NeedTrig == 0, NeedCount == 1.

 /********************************************************************/
 //   If no more acks are needed, assign Sequence number,
 //   inform client, and clear out ack table
 /********************************************************************/

ack7  ACT_assignSeq(@S, Obj, Off, Len, Stamp, Writer)  :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.

ack8  B_ACTwriteComplete(@Writer, Obj)  :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.

ack9  delete TBL_needACK(@S, Obj, C, WriterId, RecvStamp, Need) :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.

 /********************************************************************/
 //   Lease Support:
 //   Server writes to lease object every (lease interval/2)
 //   Client subscribes for lease object.
 /********************************************************************/

 cl1  ACT_writeEvent(@S, Obj, Value)  :-
          periodic(@S, I, -1), I=leaseTime/2,
          intervalObj="/volumelease", TBL_serverId(@S,S).

 cl2  ACT_addInvalSub(@X, X, S, SS, CTP)  :-
          RCV_serverConfig(@X, S), SS="/volumelease", CTP=="CP", X ≠S.

 cl3  ACT_addBodySub(@X, X, S, SS)  :-
          RCV_serverConfig(@X, S), SS="/volumelease", X ≠S.

 /********************************************************************/
 //   Lease maintenance:
 //   At the end of every interval, for all expired leases,
 //   remove inval subscriptions, generate acks if needed,
 //   and remove from hasCallback table.
 /********************************************************************/

ll01  expiredLease(@S, C, Obj)  :-
          periodic(@S, LeastTime, -1), hasCallback(@S, Obj, C, ST),
          TimeElapsed=f_now()-ST, TimeElapsed < LeaseTime, TBL_serverId(@S, S).

ll02  ACT_removeInvalSub(@C, S, C, Obj)  :-
          expiredLease(@S, C, Obj).

ll03  ackServer(@S, C,Obj, Off, Len, Writer, NeedStamp) :-
          TBL_needAck(@S, Obj, Off, Len, C, Writer, NeedStamp, Need),
          expiredLease(@S, C, Obj), Need= 0.
```

```
ll04  delete hasCallback(@S, Obj, C, _) :-
          expiredLease(@S, C, Obj).

 /********************************************************************/
 //   Support for partial file writes:
 //   - for every local write, add to currentValid Table.
 //   - When an inval arrives, and if it satisfies a read miss,
 //   add to currentValid table,
 //   - If the inval does not correspond to read miss,
 //   update table accordingly and remove callback only when
 //   all block ranges for the object are invalid
 /********************************************************************/

pfw1  TBL_currentValid(@C, Obj, Off, Len, State)  :-
          TRIG_write(@C, Obj, Off, _, _), State="Valid".
          TBL_serverId(@C, S), S≠C.

pfw2  TBL_currentValid(@C, Obj, Off, Len, State)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count),
          Count==1, State="Valid".

pfw3  TBL_currentValid(@C, Obj, Off, Len, NewState)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count),
          TBL_currentValid(@C, Obj, Off.  Len,_), Count==0, NewState="Invalid".

pfw4  isAnythingElseValid(@C, Obj, a_COUNT<*>) :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count),
          TBL_currentValid(@C, Obj, Off2, Len2, State),
          Count==0, State=="Valid", Off≠Off2, Len≠len2.

pfw5  ACT_removeInvalSub(@C, S, C, Obj)  :-
          isAnythingElseValid(@C, Obj, Count), Count==0.

pfw6  delete TBL_currentValid(@C, Obj, _, _, _)  :-
          isAnythingElseValid(@C, Obj, Count), Count==0.

 /********************************************************************/
 //   If server recieves a blind write,
 //   establishes a callback if there isn't one.
 /********************************************************************/

 bw1  blindWriteWithCallback(@S, C, Obj, a_COUNT<*>)  :-
          TRIG_invalArrives(@S, C, Obj, _, _, Writer, _),
          TBL_hasCallback(@S, Obj, C, _),
          C== Writer, TBL_serverId(@S, S),

 bw2  TBL_hasCallback(@S, Obj, C, T)  :-
          blindWriteWithCallback(@SS, C, Obj, Count),
          Count==0, T=f_now().

 bw3  ACT_addInvalSub(@S, S, C, Obj, CTP)  :-
          blindWriteWithCallback(@SS, C, Obj, Count), Count==0.
```

A.2.2 Blocking Policy

Blocking Point	Predicate
ReadNowBlock	isValid and isComplete and isSequenced and maxStaleness(Server, 1, leaseInterval) (at Client)
WriteAfterBlock	B_Action(writeComplete, objId) (at Client)
ApplyUpdateBlock	isValid (at Client & Server)

A.3 P-Coda

The implementation of P-Coda requires 37 routing rules and 7 blocking conditions.

A.3.1 Routing Policy

```
 /*******************************************************************/
 //   Initialization:
 //   Read server configuration and hoard list.
 //   Initialize server state to notConnected.
 /*******************************************************************/

 in1  ACT_readEvent(@X, ObjId)  :-
           initialize(@X), ObjId = "/.serverCFG".

 in2  TBL_server(@X, S)  :-
           RCV_serverConfig(@X, S).

 ss1  TBL_serverState(@X, State)  :-
           initialize(@X), State="notConnected".

 /*******************************************************************/
 //   Client establishes subscriptions to
 //   server for transferring updates.
 /*******************************************************************/

csSb1 ACT_addInvalSub(@X, X, S, SS, CTP)  :-
           RCV_serverConfig(@X, S), SS="/*", CTP=="LOG", X≠S.

csSb2 ACT_addBodySub(@X, X, S, SS)  :-
           RCV_serverConfig(@X, S), SS="/*", X≠S.

 /*******************************************************************/
 //   If subscriptions are successful, update server state
 //   otherwise try again
 /*******************************************************************/

 ss2  connectedToServer(@X)  :-
           TRIG_subStart(@X, X, S, _, _).

 ss3  TBL_serverState(@X, State)  :-
           TRIG_connectedToServer(@X), State="Connected".
```

```
csSb3 ACT_addInvalSub(@X , X, S, SS, CTP)  :-
          TRIG_subEnd(@X, X, S, SS, _, Type), Type=="inval", CTP=="LOG".

csSb4 ACT_addBodySub(@X, X, S, SS)  :-
          TRIG_subEnd(@X, X, S, SS, _, Type), Type=="body".

 ss4  TBL_serverState(@X, State)  :-
          TRIG_subEnd(@X, X, S, SS, _, Type), Type=="inval", State="notConnected".

 /*********************************************************************/
 //   If a server is detected, establish a subscription for "empty"
 //   to get information about missing updates.
 /*********************************************************************/

csSb6 ACT_addInvalSub(@X,S , X, SS, CTP)  :-
          connectedToServer(@X), TBL_serverId(@X, S), SS="EMPTY", CTP=="LOG".

 /*********************************************************************/
 //   If not connected to server, simply inform blocking policy so that
 //   local writes and local read misses do not block.
 /*********************************************************************/

 li1  BACT_writeComplete(@C, Obj)  :-
          TRIG_write(@C, Obj, _, _, _, _), TBL_serverId(@C, S), C≠S,
          TBL_serverState(@C, St), St=="notConnected".

 li2  BACT_writeComplete(@C, Obj)  :-
          TRIG_operationBlock(@C, Obj, Off, Len, Bpoint, _),
          Bpoint == "readAt", TBL_serverId(@C, S), C≠S,
          TBL_serverState(@C, St), St=="notConnected".

 /*********************************************************************/
 //   On read miss, client store it in a table and informs server.
 /*********************************************************************/

 rm1  readMiss(@C, Obj, Off, Len)  :-
          TRIG_operationBlock(@C, Obj, Off, Len, Bpoint, _),
          Bpoint == "readAt", TBL_serverId(@C, S), C≠S.
          TBL_serverState(@C, State), State=="Connected".

 rm2  TBL_readMiss(@C, Obj, Off, Len)  :-
          readMiss(@C, Obj, Off, Len), TBL_serverState(@C, State), State=="Connected".

 rm3  clientRead(@S, C, Obj, Off, Len)  :-
          readMiss(@C, Obj, Off, Len), TBL_serverState(@C, State), State=="Connected".

 /*********************************************************************/
 //   On client read miss, server sends body and establishes callback
 /*********************************************************************/

 cb1  ACT_sendBody(@S, S, C, Obj, Off, Len)  :-
          clientRead(@S, C, Obj, Off, Len).

 cb2  ACT_addInvalSub(@S, S, C, Obj, CTP)  :-
          clientRead(@S, C, Obj, Off, Len), CTP := "CP".

 cb3  TBL_hasCallback(@S, Obj, C) :-
          clientRead(@S, C, Obj, Off, Len),

 /*********************************************************************/
 //   When client receives an invalidation, if it does
 //   not satisfy a read miss, ack server and cancel callback.
 //   Otherwise, remove it from the table
 /*********************************************************************/
```

```
rinv0 satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, a_COUNT<*>) :-
          TRIG_InvalArrives(@C, S, Obj, Off, Len, Writer, Stamp),
          TBL_serverId(@C, S), S≠C,
          TBL_readMiss(@C, S, Obj, Off, Len).

rinv1 ackServer(@S, C, Obj, Off, Len, Writer, Stamp)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count==0,

rinv2 ACT_removeInvalSub(@C, S, C, Obj)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count==0,

rinv3 delete TBL_readMiss(@C, S, Obj, Off, Len)  :-
          satsifiesReadMiss(@C, S, Obj, Off, Len, Writer, Stamp, Count), Count > 1.

 /*********************************************************************/
 //   When server receives an invalidation,
 //   gather acks from others who have callbacks.
 //   Note:
 //   - Acks are accumulative so on received ack also
 //   acks all previous required acks.
 //   - needAck table keeps track of which acks are needed.
 /*********************************************************************/

 ack1  TBL_needAck(@S, Obj, Off, Len, C2, Writer, Stamp, Need)  :-
          TRIG_invalArrives(@S, C, Obj, Off, Len, Writer, Stamp),
          TBL_hasCallback(@S, Obj, C2), C2 ≠ Writer,
          Need := 1, TBL_serverId(@S, S).

 ack2  TBL_needAck(@S, Obj, Off, Len, C2, Writer, Stamp, Need)  :-
          TRIG_invalArrives(@S, C, Obj, Off, Len, Stamp, Writer),
          C2 == Writer, Need := 0, TBL_serverId(@S, S).

 ack3  TBL_needAck(@S, Obj, Off, Len, C, Writer, NeedStamp, Need)  :-
          ackServer(@S, C, _, _, _, _, RecvStamp),
          TB_needAck(@S, Obj, Off, Len, C, Writer, NeedStamp, _),
          NeedStamp < RecvStamp, Need:= 0, TBL_serverId(@S, S).

 ack4  TBL_needAck(@S, Obj, C, Writer, NeedStamp, Need)  :-
          ackServer(@S, C, _, _, _, RecvWriter, RecvStamp),
          TBL_needAck(@S, Obj, C, NeedWriter, NeedStamp, _),
          NeedStamp == RecvStamp, NeedWriter <= RecvWriter,
          Need:= 0, TBL_serverId(@S, S).

 ack5  delete TBL_hasCallback(@S, Obj, C)  :-
          TBL_needAck@S, Obj, _, _, C, _, _, Need)
          Need = 0, TBL_serverId(@S, S).

 ack6  acksNeeded(@S, Obj, Off, Len, Writer, Recvtamp, a_COUNT<*>) :-
          TBL_needAck(@S, Obj, Off, Len, C, Writer, RecvStamp, NeedTrig),
          TBL_needAck(@S, Obj, Off, Len, C, Writer, RecvStamp, NeedCount),
          NeedTrig == 0, NeedCount == 1.

 /*********************************************************************/
 //   If no more acks are needed, assign Sequence number,
 //   inform client, and clear out ack table
 /*********************************************************************/

 ack7  ACT_assignSeq(@S,Obj, Off, Len, Stamp, Writer)  :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.

 ack8  BACT_writeComplete(@Writer, Obj)  :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.
```

```
ack9  delete TBL_needACK(@S, Obj, C, WriterId, RecvStamp, Need) :-
          acksNeeded(@S, Obj, Off, Len, Writer, RecvStamp, Count), Count == 0.

 /*********************************************************************/
 //   Hoard when files when connected to the server,
 //   implement it like a read miss so server can establish call backs
 /*********************************************************************/

 hd1  ACT_readEvent(@C, ObjId)  :-
          connectedToServer(@C), ObjId = "/.hoardList".

 hd2  readMiss(@C, Obj, Off, Len)  :-
          RCV_hoardItems(@C, Obj), Off=0, Len=-1.

 /*********************************************************************/
 //   If server recieves a blind write,
 //   it establishes a callback if there isn't one.
 /*********************************************************************/

 bw1  blindWriteWithCallback(@SS, C, Obj, a_COUNT<*>)  :-
          TRIG_invalArrives(@S, C, Obj, _, _, Writer, _),
          TBL_hasCallback(@S, Obj, C, _),
          C== Writer, TBL_serverId(@S, S),

 bw2  TBL_hasCallback(@S, Obj, C)  :-
          blindWriteWithCallback(@SS, C, Obj, Count), Count==0,

 bw3  ACT_addInvalSub(@S, S, C, Obj, CTP)  :-
          blindWriteWithCallback(@SS, C, Obj, Count), Count==0.
```

A.3.2 Blocking Policy

Blocking Point	Predicate
ReadNowBlock	isValid and isComplete and (isSequenced or B_ACT(isDisconnected))
WriteAfterBlock	B_Action(writeComplete, objId) or B_ACT(isDisconnected)
ApplyUpdateBlock	isValid (at Client & Server)

A.3.3 Co-operative Caching

The following 8 rules were added to support co-operative caching.

```
 /*********************************************************************/
 //   Liveness monitoring:
 //   - At initialization, read from neighbor config.
 //   - Periodically ping a neighbor.
 //   - Update neighbor table when ping event is received
 //   - Periodically, check which neighbors are dead
 /*********************************************************************/

nin1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId="/.neighborList".
```

```
nin2  TBL_neighbor(@X, N, T, Live)  :-
          RCV_neighborConfig(@X, N), T=f_now(), Live=0.

 png  refresh(@N, X) :-
          periodic(@X, 5, -1), TBL_neighbor(@X, N, _, _).

rPng  TBL_neighbor(@X, N, T, Live)  :-
          refresh(@X, N), T=f_now(), Live=1.

 chk  TBL_neighbor(@X, N, T, newLive) :-
          periodic(@X, 20, -1), TBL_neighbor(@X, N, T, Live),
          Live==1, TimePassed=f_now()-T, TimePassed>20, newLive=0.

 /*********************************************************************/
 //   Co-operative Caching:
 //   On a read miss, ask all reachable peers for body.
 //   if server is not available, contact reachable peer
 //   and establish an inval subscription to get metada.
 //   Note:  we remove the inval subs when it is caught up
 //   (i.e.  we know the know about the status of peer)
 /*********************************************************************/

 cc1  ACT_sendBody(@N, N, X, Obj, Off, Off)  :-
          readMiss(@C, Obj, Off, Len), TBL_neighbor(@X, N, _, Live), L==1.

 cc2  ACT_addInvalSub(@X, N, X, Obj, CTP)  :-
          readMiss(@C, Obj, Off, Len), TBL_serverState(@C, State),
          TBL_neighbor(@X, N, _, Live), L==1, State=="notConnected", CTP="CP.

 cc3  ACT_removeInvalSub(@C, N, C, Obj)  :-
          TRIG_subCaughtup(@C, N, C, Obj), TBL_serverId(@X, S), S≠N.
```

A.4 P-TRIP

The implementation of P-TRIP requires 6 routing rules and 4 blocking conditions.

A.4.1 Routing Policy

```
 in1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId = "/.serverCFG".

 in2  TBL_server(@X, S)  :-
          RCV_serverConfig(@X, S).

 /*********************************************************************/
 //   Client establishes subscriptions to get updates.
 /*********************************************************************/

csSb1 ACT_addInvalSub(@X, S, X, SS, CTP)  :-
          RCV_serverConfig(@X, S), SS="/*", CTP=="LOG", X≠S.

csSb2 ACT_addBodySub(@X, P, X, SS)  :-
          RCV_serverConfig(@X, S), SS="/*", X≠S.
 /*********************************************************************/
 //   Re-try in cases of failure
 /*********************************************************************/

csSb3 ACT_addInvalSub(@X, S, X, SS, CTP)  :-
          TRIG_subEnd(@X, S, X, SS, _, Type), Type=="inval", CTP=="LOG".

csSb4 ACT_addBodySub(@X, S, X, SS)  :-
          TRIG_subEnd(@X, S, X, SS, _, Type), Type=="body".
```

A.4.2 Blocking Policy

Blocking Point	Predicate
ReadNowBlock	isValid and isComplete
ApplyUpdateBlock	isValid or maxStaleness(Server, 1, threshold)

A.4.3 Hierachical topology

We can easily change the topology from star to a static tree as follows.

```
 in1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId = "/.parentCFG".

 in2  TBL_parent(@X, P)  :-
          RCV_serverConfig(@X, X, P).

 /*********************************************************************/
 //   Client establishes subscriptions to get updates.
 /*********************************************************************/

csSb1 ACT_addInvalSub(@X, P, X, SS, CTP)  :-
          RCV_parentConfig(@X, P), SS="/*", CTP=="LOG", X≠P.

csSb2 ACT_addBodySub(@X, P, X, SS)  :-
          RCV_parentConfig(@X, P), SS="/*", X≠P.

 /*********************************************************************/
 //   Re-try in cases of failure
 /*********************************************************************/

csSb3 ACT_addInvalSub(@X, P, X, SS, CTP)  :-
          TRIG_subEnd(@X, P, X, SS, _, Type), Type=="inval", CTP=="LOG".

csSb4 ACT_addBodySub(@X, P, X, SS)  :-
          TRIG_subEnd(@X, P, X, SS, _, Type), Type=="body".
```

A.5 P-Bayou

The implementation of P-Bayou requires 10 routing rules and 3 blocking conditions.

A.5.1 Routing Policy

```
/*********************************************************************/
//   Periodically, pick a peer and set up subscriptions
//   to get updates.  Once caught up, remove subscriptions.
/*********************************************************************/

ae1  peerPicked(@X, a_random<N>) :-
          periodic(@X, Interval, -1), TBL_neighbor(@X, N, _, Live), Live==1.

ae2  ACT_addInvalSub(@X, N, X, SS, CTP) :-
          peerPicked(@X, N), SS="/*", CTP="LOG".

ae3  ACT_addBodySub(@X, N, X, SS) :-
          peerPicked(@X, N), SS="/*".

ae4  ACT_removeInvalSub(@X, N, X, SS)  :-
          TRIG_subCaughtup(@X,N,X,SS).

ae5  ACT_removeBodySub(@X, N, X, SS)  :-
          TRIG_subCaughtup(@X,N,X,SS).

/*********************************************************************/
//   Liveness monitoring:
//   - At initialization, read from neighbor config.
//   - Periodically ping a neighbor.
//   - Update neighbor table when ping event received.
//   - Periodically, check which neighbors are dead.
/*********************************************************************/

nin1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId="/.neighborList".

nin2  TBL_neighbor(@X, N, T, Live)  :-
          RCV_neighborConfig(@X, N), T=f_now(), Live=0.

png  refresh(@N, X) :-
          periodic(@X, 5, -1), TBL_neighbor(@X, N, _, _).

rPng  TBL_neighbor(@X, N, T, Live)  :-
          refresh(@X, N), T=f_now(), Live=1.

chk  TBL_neighbor(@X, N, T, newLive) :-
          periodic(@X, 20, -1), TBL_neighbor(@X, N, T, Live),
          Live==1, TimePassed=f_now()-T, TimePassed>20, newLive=0.
```

A.5.2 Blocking Policy

Blocking Point	**Predicate**
ReadNowBlock	isValid and isComplete
ApplyUpdateBlock	isValid

A.6 P-ChainReplication

The implementation of P-ChainReplication requires 45 routing rules and 4 blocking conditions.

A.6.1 Routing Policy

```
 /*********************************************************************/
 //   Initialization:  read master info
 //   If I am master, read in volume to node mappings
 /*********************************************************************/

ini1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId = "/.masterCFG".

ini2  TBL_master(@X, M)  :-
          RCV_masterConfig(@X, M).

ini3  ACT_readEvent(@M, ObjId)  :-
          RCV_masterConfig(@X, M), ObjId="/.volList".

ini4  TBL_volList(@M, Vol, N)  :-
          RCV_volListConfig(@X, Vol, N).

 /*********************************************************************/
 //   Keep head, tail, my successor, my predecessor information
 //   provided by the master in tables
 /*********************************************************************/

 s1   TBL_head(@X, Vol, H)  :-
          currHead(@X, Vol, H).

 s2   TBL_tail(@X, Vol, H)  :-
          currTail(@X, Vol, H).

 s3   TBL_predecessor@X, Vol, P)  :-
          currPredecessor(@X, Vol, P).

 /*********************************************************************/
 //   Establish subscriptions to predecessors to receive updates
 /*********************************************************************/

sub1  ACT_addInvalSub(@X, P, X , Vol, CTP)  :-
          predecessor(@X, Vol, P), CTP=="LOG".

sub2  ACT_addBodySub(@X, P, X , Vol)  :-
          predecessor(@X, Vol, P).

 /*********************************************************************/
 //   Inform master when the subscription is caught up.
 //   i.e.  if X is a new node, it can be used as a tail now
 /*********************************************************************/

sub4  connEstablished(@M, X, Vol)  :-
          TRIG_subCaughtup(@X,P, X, Vol), TBL_master(@X, M).
 /*********************************************************************/
 //   For consistency, every node keeps track of the latest
 //   received write.
 /*********************************************************************/

con1  rcvdWrite(@X, Vol, Writer, Stamp)  :-
          TRIG_invalArrives(@X, S, Obj, _, _, Writer, Stamp), Vol=f_getVol(Obj),

con2  TBL_LatestRcvdWrite(@X, Vol, NWriter, NStamp)  :-
          rcvdWrite(@X, Vol, NWriter, NStamp),
          TBL_LatestRcvdWrite(@X, Vol, Writer, Stamp), Stamp < NStamp.

con3  TBL_LatestRcvdWrite(@X, Vol, NWriter, NStamp)  :-
          rcvdWrite(@X, Vol, NWriter, NStamp),
          TBL_LatestRcvdWrite(@X, Vol, Writer, Stamp), Stamp == NStamp, Stamp < NStamp.

 /*********************************************************************/
 //   If I am the tail or I become the new tail, send an ACK to the head
 /*********************************************************************/

con4  BACT_ackFromTail(@H, Vol, Writer, Stamp)  :-
          rcvdWrite(@X, Vol, Writer, Stamp).
          TBL_head(@X, Vol, H), TBL_tail(@X, Vol, T), X==T.

con5  BACT_ackFromTail(@H, Vol, Writer, Stamp)  :-
          currTail(@X, Vol, T), X==T,
          TBL_LatestRcvdWrite(@X, Vol, Writer, Stamp).

 /*********************************************************************/
 //   If there is a new head and I am the tail, send an ACK to the head
 /*********************************************************************/

con6  BACT_ackFromTail(@H, Vol, Writer, Stamp)  :-
          currHead(@X, Vol, H), TBL_tail(@X, Vol, T), X==T,
          TBL_LatestRcvdWrite(@X, Vol, Writer, Stamp).

 /*********************************************************************/
 //   When master detects a new node, it looks up
 //   what volume chain to add it to.
 /*********************************************************************/

 nn1  nn1 newVolMember(@M,Vol, N) :- newNeighbor(@M, N), volList(@M,Vol, N). :-

 /*********************************************************************/
 //   If volume chain has not been set up, set the
 //   new node to head and tail and let it know.
 /*********************************************************************/

 nn2  currChainCount(@M, Vol, N, a_COUNT<*>)  :-
          newVolMember(@M, Vol, N), TBL_chain(@M, Vol, _, _).

 nn3  newHead(@N,Vol, N) :-
          currChainCount(@M, Vol, N, C), C==0.

 nn4  newTail(@N,Vol, N) :-
          currChainCount(@X, Vol, N, C), C==0.

 nn5  TBL_chain(@M, Vol, N)  :-
          currChainCount(@X, Vol, N, C), C==0.

 /*********************************************************************/
 //   When you have a new head or new tail,
 //   update tables and inform others.
 /*********************************************************************/
 nn6  currHead(@N, Vol, H)  :-
          newHead(@M, Vol, H), TBL_chain(@M, Vol, N, _).

 nn7  TBL_head(@M, Vol, H)  :-
          newTail(@M, Vol, H).

 nn8  currTail(@N, Vol, T)  :-
          newTail(@M, Vol, T), TBL_chain(@M, Vol, N, _).

 nn9  TBL_tail(@M, Vol, T)  :-
          newTail(@M, Vol, T).

 /*********************************************************************/
 //   If volume chain exists, send new node head and
 //   predecessor info and add it to the end of the chain
 /*********************************************************************/

nn10  currHead(@N, Vol, H)  :-
          currChainCount(@M, Vol, N, C), C≠0, TBL_head(@M, Vol, H).

nn11  currMaxChainPos(@M, Vol, N, a_MAX<Pos>)  :-
          currChainCount(@M, Vol, N, C), C≠0, TBL_chain(@M, Vol, _, Pos).

nn12  currPredecessor(@N,Vol, Pred)  :-
          currMaxChainPos(@M, Vol, N, PredPos), TBL_chain(@M, Vol, Pred, PredPos).

nn13  TBL_chain(@M, Vol, N, Pos)  :-
          currMaxChainPos(@M, Vol, N, PredPos), Pos = PredPos + 1.

 /*********************************************************************/
 //   When a node has caught up, if it is at the
 //   last position in the volume chain, set it as tail.
 /*********************************************************************/

nn14  currTailPos(@M, Vol, N, NPos, a_MAX<Pos>)  :-
          connEstablished(@M, Vol, N), TBL_chain(@M, Vol, N, NPos),
          TBL_chain(@M, Vol, _, Pos).

nn15  newTail(@M, Vol, T)  :-
          currTailPos(@M, Vol, N, NPos, MaxPos), NPos==MaxPos.

 /*********************************************************************/
 //   Dead node:  delete it from chain
 /*********************************************************************/

 dn1  delete TBL_chain(@M, Vol, N, NPos)  :-
          deadNeighbor(@M, N), TBL_chain(@M, Vol, N, NPos).

 /*********************************************************************/
 //   find predecessor if dead node is not head
 /*********************************************************************/

 dn2  findPredecessor(@M, N, Vol, NPos, a_MAX<PPos>)  :-
          deadNeighbor(@M, N), TBL_chain(@M, Vol, N, NPos),
          TBL_chain(@M, Vol,_, PPos), PPos < Pos,
          TBL_head(@M, Vol, H), H≠N.

 /*********************************************************************/
 //   If dead node is the current tail, set
 //   the predecessor as new tail.
 /*********************************************************************/
 dn3  newTail(@M, Vol, Pred)  :-
          findPredecessor(@M, N, Vol,NPos, PPos), TBL_tail(@M, Vol, T),
          T==N, TBL_chain(@M, Vol, Pred, PPos).

 /*********************************************************************/
 //   Otherwise, find successor and inform it of its
 //   new predecessor.
 /*********************************************************************/

 dn4  findSuccessor(@M, N, Vol, NPos, PPos, a_MIN<SPos>)  :-
          findPredecessor(@M, N, Vol, NPos, PPos), TBL_tail(@M, Vol, T),
          T≠N, TBL_chain(@M, Vol, _, SPos), SPos > NPos.

 dn5  currPredecessor(@Suc, Pred)  :-
          findSuccessor(@M, N, Vol, NPos, PPos, SPos), TBL_chain(@M, Vol, Succ, SPos),
          TBL_chain(@M, Vol, Succ, PPos),

 /*********************************************************************/
 //   if dead node is current head,
 //   find successor and set it to new head.
 /*********************************************************************/

 dn6  findNewHead(@M, N, Vol, NPos, a_MIN<PPos>)  :-
          deadNeighbor(@M, N), TBL_chain(@M, Vol, N, NPos),
          TBL_chain(@M, Vol,_, SPos), SPos > Pos,
          TBL_head(@M, Vol, H), H==N.

 dn7  newHead(@M, Vol, H)  :-
          findNewHead(@M, N, Vol, NPos, HPos), TBL_chain(@M, Vol, H, HPos).

 /*********************************************************************/
 //   Liveness monitoring:
 //   - At initialization, read from neighbor config.
 //   - Periodically ping a neighbor.
 //   - when a ping event is received,
 //   check if it is a new neighbor and
 //   udpate the neighbor table
 //   - Periodically, check which neighbors are dead
 /*********************************************************************/

nin1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId="/.neighborList".

nin2  TBL_neighbor(@X, N, T, Live)  :-
          RCV_neighborConfig(@X, N), T=f_now(), Live=0.

 png  refresh(@N, X) :-
          periodic(@X, 5, -1), TBL_neighbor(@X, N, _, _).

rPng1 newNeighbor(@X, N)  :-
          refresh(@X, N), TBL_neighbor(@X, N, _, Live), Live==0.

rPng2 TBL_neighbor(@X, N, T, Live)  :-
          refresh(@X, N), T=f_now(), Live=1.

 chk1 deadNeighbor(@X, N)  :-
          periodic(@X, 20, -1), TBL_neighbor(@X, N, T, Live),
          Live==1, TimePassed=f_now()-T, TimePassed>20.

 chk2 TBL_neighbor(@X, N, T, Live) :-
          deadNeighbor(@X, N), TBL_neighbor(@X, N, T,_), Live=0.
```

A.6.2 Blocking Policy

Blocking Point	Predicate
ReadNowBlock	isValid and isComplete
WriteAfterBlock	B_Action(ackFromTail, writer, Stamp)
ApplyUpdateBlock	isValid

A.7 P-TierStore

The implementation of P-TierStore requires 14 routing rules and 1 blocking conditions.

A.7.1 Routing Policy

```
 /*********************************************************************/
 //   Initialization:  Read parent id.
 /*********************************************************************/

 in0  TRIG_readEvent(@X, ObjId) :-
          EVT_initialize(@X), ObjId := "/.parent".

 /*********************************************************************/
 //   When node X receives its own parent id, store it in
 //   a table and read subscription list.
 /*********************************************************************/

 pp0  TBL_parent(@X, P) :-
          RCV_parent(@X, P).

 pp1  TRIG_readAndWatchEvent(@X, ObjId) :-
          RCV_initialize(@X), ObjId := "/.subList".

 /*********************************************************************/
 //   When node X receives a subscription event for
 //   one of its subscriptions, store it in a
 //   subscription table and establish an inval
 //   and body subscription from the parent.
 /*********************************************************************/

pSb0  TBL_subscription(@X, SS) :-
          RCV_subscription(@X, SS).

pSb1  ACT_addInvalSub(@X, P, X, SS, CTP) :-
          RCV_subscription(@X, SS), TBL_parent(@X, P),
          CTP=="LOG".

pSb2  ACT_addBodySub(@X, P, X, SS) :-
          RCV_subscription(@X, SS), TBL_parent(@X, P).

 /*********************************************************************/
 //   If parent subscription fails, retry.
 /*********************************************************************/
 f1   ACT_addInvalSub(@X, P, X, SS, CTP) :-
          TRIG_subEnd(@X, P, X, SS, _, Type),
          TBL_parent(@X, P), Type=="Inval", CTP:="LOG".

 f2   ACT_addBodySub(@X, P, X, SS)  :-
          TRIG_subEnd(@X, P, X, SS, _, Type),
          TBL_parent(@X, P), TYPE=="Body", CTP:="LOG".

 /*********************************************************************/
 //   If a child contacts me, establish subscriptions
 //   for "/*'' to receive updates.
 /*********************************************************************/

cSb1  ACT_addInvalSub(@X, C, X, SS, CTP) :-
          TRIG_subStart(@X, X, C, _, Type), C ≠ P,
          Type == "Inval", SS := "/*", CTP := "LOG".

cSb2  ACT_addBodySub(@X, C, X, SS, CTP)  :-
          TRIG_subStart(@X, X, C, _, Type), C ≠ P,
          Type == "Body", SS := "/*".

 /*********************************************************************/
 //   DTN Support:  if a relay node arrives,
 //   establish subscriptions to receive updates
 //   and to send local receive new updates.
 /*********************************************************************/

dtn1  ACT_addInvalSub(@X, R, X, SS, CTP) :-
          EVT_relayNodeArrives(@X, R),
          TBL_subscription(@X, SS), CTP=="LOG".

dtn2  ACT_addBodySub(@X, R, X, SS) :-
          EVT_relayNodeArrives(@X, R),
          TBL_subscription(@X, SS), CTP=="LOG".

dtn3  ACT_addInvalSub(@X, X, R, SS, CTP) :-
          EVT_relayNodeArrives(@X, R), SS:="/*", CTP=="LOG".

dtn4  ACT_addBodySub(@X, X, R, SS) :-
          EVT_relayNodeArrives(@X, R), SS:="/*", CTP=="LOG".
```

A.7.2 Blocking Policy

Blocking Point	**Predicate**
ApplyUpdateBlock	isValid

A.8 P-Pangaea

The implementation of P-Pangaea requires 59 routing rules and 1 blocking predicate.

A.8.1 Routing Policy

```
 /********************************************************************/
 //   Assumptions:
 //   - the FileEntries table stores the gold nodes for every file entry for
 //   a directory as tuples FileEntries(@X, dir, child, GoldNode).
 //   - Every node has a Peer table to store links to gold and bronze replicas.
 //   - Every server has a latency table that stores all pairs RTT information.
 /********************************************************************/

 /********************************************************************/
 //   Initalization:  Read in file entries of root directory
 /********************************************************************/

ini01 ACT_readAndWatchEvent(@S, ObjMeta)  :-
          initalize(@S), ObjMeta="/.meta".

 /********************************************************************/
 //   When you receive new gold node information, add it to file entry table.
 /********************************************************************/

 fe01  TBL_fileEntry(@S, Dir, Obj, G1)  :-
          RCV_fileEntryConfig(@S, Dir, Obj, G1, G2, G3).

 fe02  TBL_fileEntry(@S, Dir, Obj, G2)  :-
          RCV_fileEntryConfig(@S, Dir, Obj, G1, G2, G3).

 fe03  TBL_fileEntry(@S, Dir, Obj, G3)  :-
          RCV_fileEntryConfig(@S, Dir, Obj, G1, G2, G3).

 /********************************************************************/
 //   Initialization:  Read latency information
 /********************************************************************/

ini02 ACT_read(@S, Obj)  :-
          intialize(@S), Obj = "/.latencyConfig".

ini03 TBL_latency(@S, X, Y, Lat)  :-
          RCV_latencyConfig(@S, X, Y, Lat).

 /********************************************************************/
 //   On a read miss, find parentdir of the object
 //   and add to outstanding read miss table
 /********************************************************************/

 rm1  readMiss(@S, Obj)  :-
          TRIG_operationBlock(@S, ObjData, Off, Len, Bpoint, _),
          Bpoint == "readAt", Obj=f_getObjFromData(ObjData).

 rm2  missParentChild(@S, Parent, Child)  :-
          readMiss(@S, Child), f_getParent(Child).

 rm3  TBL_outstandingMiss(@S, Parent, Child)  :-
          missParentChild(@S, Parent, Child).

 /********************************************************************/
 //   Check FileEntries table:
 //   if no parent entry found, issue read miss for parent
 //   if parent entries exist but no obj entry, new file needs to be created
 //   if obj entry found, pick Gold Node, establish connections to it
 //   and ask it to pick peers.
 /********************************************************************/
 rm4  findParentEntries(@S, Parent, Child, a_count<*>) :-
          missParentChild(@S, Parent, Child), TBL_fileEntries(@S, Parent, _, _),

 rm5  readMiss(@S, Parent)  :-
          findParentEntries(@S, Parent,Child, Count), Count==0.

 rm6  findObjEntries(@S, Parent, Child, a_count<*>)  :-
          findParentEntries(@S, Parent,Child, Count), Count>0,
          TBL_fileEntries(@S, Parent, Child, _).

 rm7  createNewObj(@S, Parent, Child)  :-
          findObjEntries(@S, Parent, Child, Count), Count==0.

 rm8  pickGoldNode(@S, Child, a_RANDOM<GNode>)  :-
          findObjEntries(@S, Parent, Child, Count), Count ≠0.
          TBL_FileEntries(@S, Dir, File, GNode).

 rm9  needReplica(@GNode, S, Child)  :-
          pickGoldNode(@S, Child, GNode).

rm10  establishConnections(@S, GNode, Child)  :-
          pickGoldNode(@S, Child, GNode).

 /*********************************************************************/
 //   When a gold node recieves a needReplica message,
 //   it picks 2 peers from peer list for S to contact.
 /*********************************************************************/

rm11  pickClosestPeer(@G, S, Obj, a_MIN<Lat>)  :-
          needReplica(@G, S, Obj), TBL_peer(@G, Obj, P),
          TBL_neighbor(@G, P, _, Live), Live == 1,
          TBL_latency(@G, S, P, Lat).

rm12  pickRandomPeer(@G, S, Obj, CP, a_RANDOM<RP>)  :-
          pickClosestPeer(@G, S, Obj, MinLat),
          TBL_latency(@G, S, CP, MinLat), TBL_peer(@G, Obj, RP),
          TBL_neighbor(@G, RP, _, Live), CP ≠RP, Live==1.

 /*********************************************************************/
 //   Once peers are picked, ask S to establish connections
 /*********************************************************************/

rm13  establishConnections(@S, Obj, CP) :-
          pickRandomPeer(@G, S, Obj, CP, RP).

rm14  establishConnections(@S, Obj, RP)  :-
          pickRandomPeer(@G, S, Obj, CP, RP).

 /*********************************************************************/
 //   Establish connections by setting up bi-directional
 //   body and inval connections for both data and meta data
 /*********************************************************************/

con1  TRIG_addInvalSub(@S, P, S, SS, CTP)  :-
          establishConnections(@S, Obj, P), SS=Obj+"/*",
          CTP="CP".

con2  TRIG_addInvalSub(@S, S, P, SS, CTP)  :-
          establishConnections(@S, Obj, P), SS=Obj+"/*",
          CTP="CP".

con3  TRIG_addBodySub(@S, P, S, SS)  :-
          establishConnections(@S, Obj, P), SS=Obj+"/*".
```

```
con4  TRIG_addBodySub(@S, S, P, SS)  :-
          establishConnections(@S, Obj, P), SS=Obj+"/*".

 /**********************************************************************/
 //   When an inval connection catches up, add to peer table.
 //   If there is an outstanding miss for object,
 //   readAndWatch meta data and remove from outstanding miss table.
 /**********************************************************************/

con5  TBL_peer(@S, Obj, P)  :-
          ACT_subCaughtup(@S, P, S, SS), Obj=f_getObj(SS).

con6  receivedObj(@S, Obj)  :-
          ACT_subCaughtup(@S, P, S, SS), Obj=f_getObj(SS).

con7  TRIG_readAndWatchEvent(@S, Meta)  :-
          receivedObj(@S, Obj), TBL_outstandingMiss(@S, _, Obj), Meta=Obj+"/.meta".

con8  delete TBL_outstandingMiss(@S, Parent, Obj)  :-
          receivedObj(@S, Obj), TBL_outstandingMiss(@S, Parent, Obj).

 /**********************************************************************/
 //   As we get new meta-data, check if there is an outstanding
 //   miss for the child.  If so, reissue miss.
 //   Note:  we use a retryMiss table to ensure that we issue the miss
 //   after the new file entries have been added to the table.
 /**********************************************************************/

ret1  TBL_retryMiss(@S, Dir, Obj)  :-
          RCV_fileEntryConfig(@S, Dir, Obj, G1, G2, G3),
          TBL_outstandingMiss(@S, Dir, Obj).

ret2  missParentChild(@S, Dir, Obj)  :-
          TBL_retryMiss(@S, Dir, Obj).

 /**********************************************************************/
 //   if we have received all entries from parent of an outstanding
 //   miss, need to create the object
 /**********************************************************************/

ret3  createNewObj(@S, Dir, Child)  :-
          RCV_EndFileEvent(@S, ObjMeta), Dir = f_getObj(ObjMeta),
          TBL_outstandingMiss(@S, Dir, Child).

 /**********************************************************************/
 //   File Creation:
 //   find new gold nodes, establish connections.
 //   Then update directory object.
 /**********************************************************************/

 fc1  pickFirstPeerGold(@S, Dir, Obj, a_RANDOM<P>) :-
          createNewObj(@S, Dir, Obj) TBL_neighbor(@S, P,_, Live), Live==1.

 fc2  pickSecondPeerGold(@S, Dir, Obj, G1, a_RANDOM<P>) :-
          pickFirstPeerGold(@S, Dir, Obj, G1), TBL_neighbor(@, P, _, Live),
          G1 ≠P, Live==1.

 fc3  establishConnections(@S, Obj, G1) :-
          pickSecondPeerGold(@S, Dir, Obj, G1, G2).

 fc4  establishConnections(@S, Obj, G2) :-
          pickSecondPeerGold(@S, Dir, Obj, G1, G2).
```

```
fc5  ACT_WriteEvent((@S, ObjMeta, EName, Dir, Obj, S, G1, G2) :-
          pickSecondPeerGold(@S, Dir, Obj, G1, G2), ObjMeta =Obj+"/.meta",
          EName = "fileEntryConfig".

 /*********************************************************************/
 //   Temporary Failures:
 //   retry connections when they become live
 /*********************************************************************/

tf1  establishConnection(@S, Obj, P) :-
          liveNeighbor(@S, P), TBL_peer(@S, Obj, P).

 /*********************************************************************/
 //   Permanent Failures:
 //   check how long a node has been dead.
 //   if deadPeer, ask a gold node to pick a new peer
 /*********************************************************************/

pf1  deadNode(@S, Obj, P)  :-
          periodic(@S, 60*60, -1), TBL_neighbors(@S, P, T, Live),
          T > f_now() - 60*60, Live==0, TBL_peer(@S, Obj, P).

pf2  delete TBL_peer(@S, Obj, P) :-
          deadNode(@S, Obj, P).

pf3  isNodeGold(@S, Obj, P, a_COUNT<*>)  :-
          deadNode(@S, Obj, P), TBL_fileEntry(@S, Obj, G), G==P.

pf4  deadPeer(@S, Obj, P)  :-
          isNodeGold(@S, Obj, P, C), C==0.

pf5  deadGoldPeer(@S, Obj, P)  :-
          isPeerGold(@S, Obj, P, C), C>0.

pf6  pickGoldForRecovery(@S, Obj, a_random<G>)  :-
          deadPeer(@S, Obj, P), TBL_fileEntry(@S, Obj, G).

pf7  needNewPeer(@G, Obj, S)  :-
          pickGoldForRecovery(@S, Obj, G).

 /*********************************************************************/
 //   Gold node picks a peer and sends it over
 /*********************************************************************/

pf8  needPeer(@S, G, Obj, NP) :-
          needNewPeer(@G, Obj, S), TBL_peer(@G, Obj, NP),
          TBL_neighbor(@G, NP, _, Live), Live==1.

 /*********************************************************************/
 //   When receive new peer, check if it is already a peer.
 //   if not, establish connections.
 //   if so, ask gold node for another peer.
 /*********************************************************************/

pf9  checkIfExistingPeer(@S, G, Obj, NP, a_COUNT<*>) :-
          newPeer(@S, G, Obj,NP), TBL_peer(@S, Obj, P), NP==P.

pf10 establishConnections(@S, Obj, NP)  :-
          checkIfExistingPeer(@S, G, Obj,NP, C), C==0.

pf11 needNewPeer(@G, S, Obj)  :-
          checkIfExistingPeer(@S,G, Obj,NP, C), C>0.
 /*******************************************************************/
 //   Permanent Failures:
 //   if dead gold node, pick a random peer to upgrade
 //   to gold and update object meta data
 /*******************************************************************/

pf12  pickNewGold(@G, Obj, OP, a_RANDOM<NP>)  :-
          deadGoldPeer(@G, Obj, OP), TBL_peer(@G, Obj, NP),
          OP ≠ NP, TBL_neighbor(@G, NP, _, Live), Live==1.

pf13  findOtherGold(@G, Dir, Obj,OP, LG, NP)  :-
          pickNewGold(@G, Obj, OP, NP), TBL_FileEntry(@G,Dir,Obj, LG),
          LG≠G, LG≠OP.

pf14  ACT_WriteEvent((@G, ObjMeta, EName, Dir, Obj, S, G1, G2) :-
          findOtherGold(@G, Dir, Obj, OP, G1, G2), ObjMeta =Obj+"/.meta",
          EName = "fileEntryConfig".

pf16  establishConnections(@NP, Obj, LG) :-
          findOtherGold(@G, _, Obj,_, LG, NP).

 /*******************************************************************/
 //   Liveness monitoring:
 //   - At initialization, read from neighbor config.
 //   - Periodically ping a neighbor.
 //   - when a ping event is received,
 //   check if it is a new neighbor and
 //   udpate the neighbor table
 //   - Periodically, check which neighbors are dead
 /*******************************************************************/

nin1  ACT_readEvent(@X, ObjId)  :-
          initialize(@X), ObjId="/.neighborList".

nin2  TBL_neighbor(@X, N, T, Live)  :-
          RCV_neighborConfig(@X, N), T=f_now(), Live=0.

 png  refresh(@N, X) :-
          periodic(@X, 5, -1), TBL_neighbor(@X, N, _, _).

rPng1 liveNeighbor(@X, N)  :-
          refresh(@X, N), TBL_neighbor(@X, N, _, Live), Live==0.

rPng2 TBL_neighbor(@X, N, T, Live)  :-
          refresh(@X, N), T=f_now(), Live=1.

chk1  deadNeighbor(@X, N)  :-
          periodic(@X, 20, -1), TBL_neighbor(@X, N, T, Live),
          Live==1, TimePassed=f_now()-T, TimePassed>20.

chk2  TBL_neighbor(@X, N, T, Live) :-
          deadNeighbor(@X, N), TBL_neighbor(@X, N, T,_), Live=0.
```

A.8.2 Blocking Policy

Blocking Point	**Predicate**
ApplyUpdateBlock	isValid

Bibliography

[1] http://www.w3.org/html/.

[2] Spin formal verification. http://spinroot.com/spin/whatispin.html.

[3] Sql. http://en.wikipedia.org/wiki/SQL.

[4] *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[5] *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[6] *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[7] *Proceedings of the 5th ACM Symposium on Operating Systems Design and Implementation*, December 2002.

[8] *Proceedings of the 6th ACM Symposium on Operating Systems Design and Implementation*, December 2004.

[9] *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, May 2006.

[10] *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, April 2009.

[11] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[12] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th ACM Symposium on Operating Systems Design and Implementation* [7].

[13] M. Ahamad, P. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the International Conference on Distributed Computing Systems*, May 1991.

[14] 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.

[15] N. Belaramani and M. Dahlin. Achieving energy efficiency while synchronizing personal data. Technical Report TR-09-21, The University of Texas at Austin, August 2009.

[16] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation* [9].

[17] 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.

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

[19] W. Chen and X. Liu. Enforcing routing consistency in structured peer-to-peer overlays: Should we and could we? In *Proceedings of the 5th International workshop on Peer-to-Peer Systems*, February 2006.

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

[21] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* [6].

[22] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for large-scale systems. Technical Report TR-04-28, University of Texas at Austin, March 2004.

[23] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Per-

formance. In *Proceedings of the First ACM Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[24] D. Dao, J. Albrecht, C. Killian, and A. Vahdat. Live debugging of distributed systems. In *Proceedings of the 18th International Conference on Compiler Construction*, March 2009.

[25] 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 21st ACM Symposium on Operating Systems Principles*, October 2007.

[26] M. Demmer, B. Du, and E. Brewer. Tierstore: A distributed file-system for challenged networks. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[27] D. L. Dill. Ther murphi verification system. In *8th International Conference on Computer Aided Verification*, 1996.

[28] 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 15th ACM Symposium on Operating Systems Principles* [5], pages 201–212.

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

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

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

[32] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[33] J. Gray, P.Helland, P. E. O'Neil, and D. Shasha. Dangers of replication and a solution. In *Proceedings of SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[34] Robert Grimm. Better extensibility through modular syntax. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–51, June 2006.

[35] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Proceedings of the 8th acm symposium on operating systems design and implementation. In *Proceedings of the 8th ACM Symposium on Operating Systems Design and Implementation*, December 2008.

[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*, June 1990.

[37] R. Guy, P. Reiher, D. Ratner, M. Gunter, and W. Ma. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Workshop on Mobile Data Access*, pages 254–265, 1998.

[38] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 29–43, December 1993.

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

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

[41] Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. In *The International Conference on Object Oriented Programming, Systems, Languages and Applications*, October 2007.

[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] H. Hu and A. Vahdat. Building replicated internet services using TACT: A toolkit for tunable availability and consistency tradeoffs. In *Proceedings of 2nd International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems*, 2000.

[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*, May 2003.

[46] A. Karypidis and S. Lalis. Omnistore: A system for ubiquitous personal storage management. In *Annual IEEE International Conference on Pervasive Computing and Communications*, pages 136–147, 2006.

[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] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[49] J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[50] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[51] 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.

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

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

[54] L. Lamport. Part time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

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

[56] 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.

[57] P. Mahajan, S. Lee, J. Zheng, L. Alvisi, and M. Dahlin. Astro: Autonomous and trustworthy data sharing. Technical Report TR-08-24, The University of Texas at Austin, October 2008.

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

[59] D. Malkhi and D. Terry. Concise version vectors in WinFS. In *Proceedings of the 20th Symposium on Distributed Computing*, 2005.

[60] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* [5].

[61] L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *Proceedings of the Summer 1994 USENIX Conference* [4].

[62] 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.

[63] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th ACM Symposium on Operating Systems Design and Implementation* [7].

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

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

[66] E. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th ACM Symposium on Operating Systems Design and Implementation* [8].

[67] 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.

[68] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proceedings of the Summer 1994 USENIX Conference* [4].

[69] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings*

of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[70] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, and A. Vahdat C. C. Marshall. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* [10].

[71] 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* [4].

[72] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation* [9].

[73] 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.

[74] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the First USENIX Symposium on Networked Systems Design and Implementation*, March 2004.

[75] A. Rowstron and P. Druschel. Pastry: Scalable, distirbuted object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*, Nov 2001.

[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 18th ACM Symposium on Operating Systems Principles* [6].

[77] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the 5th ACM Symposium on Operating Systems Design and Implementation* [7].

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

[79] 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.

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

[81] 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.

[82] S. Sobti, N. Garg, F. Zheng, J. Lai, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: a distributed mobile storage system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[83] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Aug 2001.

[84] J. Stribling, Y. Sovran, I.Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* [10].

[85] 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.

[86] 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 15th ACM Symposium on Operating Systems Principles* [5].

[87] 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.

[88] R. van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th ACM Symposium on Operating Systems Design and Implementation* [8].

[89] R. Wang and T. Anderson. xFS: A Wide Area Mass Storage File System. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 71–78, October 1993.

[90] G. Wuu and A. Berstein. Efficient solutions to the replicated log and dictionary problem. In *Proceedings of the 3rd Symposium on the Principles of Distributed Computing*, pages 233–242, 1984.

[91] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Proceedings of the 6th usenix symposium on networked systems design and implementation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* [10].

[92] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[93] 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.

[94] 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.

[95] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-sclae overlay for service deployment. In *IEEE Journal on Selected Areas in Communications*, Jan 2004.

[96] J. Zheng. *URA: A Universal Data Replication Architecture.* PhD thesis, The University of Texas at Austin, August 2008.

[97] J. Zheng, N. Belaramani, and M. Dahlin. Pheme: Synchronizing replicas in diverse environments. Technical Report TR-09-07, University of Texas at Austin, February 2009.

Vita

Nalini Belaramani was born in Hong Kong on the 2nd of November 1978 to two loving parents—Rani and Moti Belaramani. She was soon joined by her sister, Kiran, who has been her constant source of entertainment ever since.

Nalini got her first computer in 1996 and was totally intrigued by it. She decided to "unravel" the mysteries of the computer by getting a BEng degree in Computer Engineering and a MPhil degree in Computer Science from The University of Hong Kong.

While working for Motorola Semi-Conductors HK Ltd as an engineer, she realized that there were more mysteries to be solved and more knowledge to be found. She join the PhD program at the University of Texas at Austin, where the weather and the people made her quest for knowledge a grueling, and yet delightful experience.

And now, with enough knowledge under her belt, she is ready to face the world.

Permanent address: 16/A South Sea Mansion,
81 Chatham Road,
TST, KLN, Hong Kong.

This dissertation was typed by the author.