PADS: A Policy Architecture for Building Distributed Storage Systems

Nalini Belaramani*, Jiandan Zheng§, Amol Nayate†, Robert Soulé‡, Mike Dahlin*, Robert Grimm‡
*UT Austin §Amazon.com Inc. †IBM TJ Watson Research ‡NYU

Abstract: This paper presents PADS, a new *policy architecture* that makes it easier to develop distributed storage systems. PADS is based on two key ideas. First, a distributed storage system is implemented by specifying a *control plane* that embodies the design policy of the system over a *data plane* that provides a set of common mechanisms. Second, the control plane policy is separated into *routing policy* which specifies how data flows through the system and *blocking policy* which forces reads, writes, and data propagation to wait until system consistency and durability invariants are met. The argument for PADS is simple: PADS qualitatively reduces the effort to build new systems. For example, using PADS we were able to construct a dozen significant distributed storage systems spanning a large portion of the design space.

1 Introduction

Our goal is to make it easy for system designers to construct new distributed storage systems for challenging environments. Distributed storage systems need to deal with a wide range of heterogeneity in terms of device 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). To cope with these varying demands, new systems are developed [6, 12, 14, 19, 20, 23, 24, 30], each system making design choices that balance performance, resource usage, consistency, and availability. Because these tradeoffs are fundamental [7, 17, 33], we do not expect a single "hero" distributed storage system to emerge to serve all situations and end the need for new systems in new environments and workloads.

In previous work [3, 22], we developed the PRACTI (Partial Replication, Arbitrary Consistency, Topology Independence) mechanisms. We speculated that PRACTI could serve as a "replication microkernel" over which a broad range of new distributed storage systems could quickly be built and we imagined demonstrating this hypothesis by constructing half a dozen diverse and demanding systems modeled on those from the literature [5, 14, 21, 25, 27, 32]. These efforts bogged down. We concluded that PRACTI was only half the story. It was not sufficient to define a set of replication mechanisms; we also had to define an architecture to easily implement policies that define a desired design.

This paper presents PADS, a policy architecture that makes it easier to construct new distributed storage systems. PADS is based on two key ideas.

First, a distributed storage system is built by specifying directives in the *control plane* over a *data plane*. The data plane encapsulates the underlying mechanisms to handle the details of storing and transmitting data and maintaining consistency information. System designers only need to write control plane policy that orchestrates data flows. Cleanly separating the control plane from the data plane simplifies a designer's job by abstracting away many complex implementation details.

Second, the PADS architecture divides control plane policy into two aspects: *routing* and *blocking*.

- *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? which node to contact on a read miss? etc. Routing policy sets up data flows among nodes to meet a system's performance, availability, and resource consumption goals.

- *Blocking policy*: Blocking policy specifies when nodes must block incoming updates or local read/write requests so as to maintain system invariants. Blocking is particularly useful for specifying consistency and durability goals. For example, a policy might block the completion of a write until it reaches at least 3 other nodes to meet a system's durability requirement.

The main challenge for PADS's detailed design is to provide an API for system development that is simple, flexible, and efficient. For routing policy, PADS provides a set of *actions* that set up data flows, a set of *triggers* that expose local node information, and the abstraction of *stored events* to allow persistent state in the data plane to affect routing decisions by the control plane. In order to further simplify the definition of routing policies in terms of these primitives, PADS also defines R/OverLog[1], an extension of the OverLog [18] declarative routing language. For blocking policy, PADS provides a set of *blocking predicates* that block access to data until *blocking conditions* are satisfied.

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

	Simple Client Server	Full Client Server	Coda [14]	Coda +Coop Cache	TRIP [21]	TRIP +Hier	Tier Store [5]	Tier Store +CC	Chain Repl [32]	Bayou [25]	Bayou +Small Dev	Pangaea [27]
Routing Rules	21	43	31	44	6	6	19	32	75	9	9	75
Blocking Conditions	5	6	5	5	3	3	1	1	4	3	3	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	Seq.	Seq.	Open/ Close	Open/ Close	Seq.	Seq.	Coher.	Coher.	Linear.	Causal	Coher.	Coher.
Callbacks	√	√	√	√	√	√						
Leases		√	√	√								
Invalidation vs. whole update propagation	Inval.	Inval.	Inval.	Inval.	Inval.	Inval.	Update	Update	Update	Update	Update	Update
Disconnected operation			√	√	√	√	√	√		√	√	√
Crash recovery	√	√	√	√	√	√	√	√	√	√	√	√
Object store interface*	√	√	√	√	√	√	√	√	√	√	√	√
File system interface*		√	√	√	√	√	√	√	√	√	√	√

Fig. 1: Features covered by case-study systems. *Note that the original implementations of some of these systems provide interfaces that differ from the object store or file system interfaces we provide in our prototypes.

Ultimately, the evidence for PADS's usefulness is simple: we were able to use PADS to construct a dozen distributed storage systems summarized in Fig. 1. These systems were chosen because they span a large portion of the design space, and PADS's ability to support these systems suggests that PADS captures key abstractions for distributed systems. In addition, the fact that two students could construct such a range of systems in a matter of months illustrates the qualitative simplification PADS represents for system implementers. Notably, in contrast with the ten thousand or more lines of code it typically takes to construct such a system using standard practice, given the 32K lines of code of the PADS framework, it requires just 6-75 routing rules and a handful of blocking conditions to define each new system with PADS. As an example of the benefits of this approach, we note that, to the best of our knowledge, we provide the first implementations of Chain Replication [32] and TRIP [21], which were evaluated via simulations in the original papers.

A key issue in interpreting Fig. 1 is understanding how complete or realistic these PADS implementations are. The PADS implementations are *not* bug-compatible recreations of every detail of the original systems, but we believe they do capture the overall architecture of these designs by storing approximately the same data on each node, by sending approximately the same data across the same network links, and by enforcing the same consistency and durability semantics; we discuss our definition of *architectural equivalence* in §5. We also note that our PADS implementations are sufficiently complete to run file system benchmarks and that they handle important and challenging real world details like configuration files and crash recovery.

Benchmarking of our PADS prototype indicates that PADS is competitive with hand-built distributed storage systems with respect to storage space and network bandwidth. However, the performance of a PADS implementation of one system (P-Coda) built on our user-level Java prototype, is up to 4 times worse than the original hand-tuned system (Coda). Overall, our performance evaluation leaves us confident that system builders can use PADS for rapid prototyping and for deployment of systems for moderately demanding applications. For demanding applications, we believe that PADS's storage and network overheads suggest that the overall PADS architecture is sound, but the absolute performance may need improvement over that offered by the prototype.

The rest of this paper describes PADS, demonstrates how to build systems with PADS, and evaluates the approach.

2 Architecture

Distributed storage systems cover a large design space. Some guarantee strong consistency while others sacrifice consistency for higher availability; some invalidate stale objects, while others push updates; some cache objects on demand, while others replicate all data to all nodes; and so on. Our design choices for PADS are driven by the need to accommodate this broad design space while allowing policies to be simple and efficient.

Fig. 2 shows how a system designer uses PADS to construct a system. She begins with a set of *high-level goals* regarding factors like performance, availability, resource consumption, consistency, and durability. She then considers the trade-offs among these goals, and architects a system design that optimizes these trade-offs.

Without PADS, she would then need to implement the system design from scratch spending several months for implementation and debugging. With PADS the system design is implemented as *policy* over a provided set of

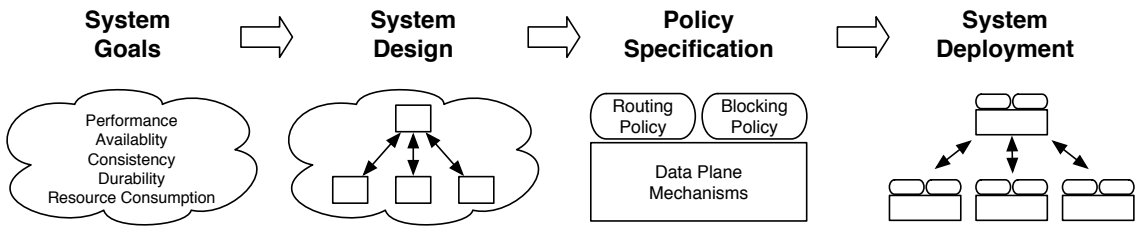

Fig. 2: Overview of PADS architecture and use.

common *mechanisms*. The system designer focuses only on policy specification.

Note that a PADS policy is a specific set of directives rather than a set of high-level goals. For example, a policy designer might decide on a client-server architecture and specify "When an update occurs, a client should send the update to the server within 30 seconds" rather than stating the high level goals "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. Distributed storage design is a creative process and PADS does not attempt to automate it. Instead, PADS aids a designer by allowing her to easily express and implement her design decisions.

As Fig. 2 indicates, PADS defines a policy architecture that is based on two principles. First, it casts policy as a *control plane* and defines an interface to system's *data plane* that embodies a set of common mechanisms. The control plane orchestrates communication among nodes, while the data plane handles low-level details like data storage, data transmission, and consistency bookkeeping. Second, PADS's policy architecture splits the control plane policy into two parts: *routing policy* and *blocking policy*.

- *Routing policy* defines when nodes should send what information to which other nodes. The intuition for routing policy is that many of the design choices that distinguish how different replication systems work (e.g., Coda vs. Bayou vs. TierStore) can be regarded as routing decisions: Where should a node go to satisfy a read miss? When and where should a node send updates it receives? Where should a node send invalidations when it learns of a new version of an object?
- *Blocking policy* specifies when a node must prevent a local read or write from proceeding or when it must delay applying an update received from another node. Blocking is crucial to implement consistency and durability semantics of a system. For example, block until a write reaches at least 3 nodes for durability, block until a server acknowledges a write for consistency, or block until local storage reflects all updates that occurred before the start of the current read for consistency.

In PADS, routing policy is specified as an event-driven program that implements these *routing decisions*. and blocking policy is specified as a set of *blocking predicates* that state the conditions that must be satisfied before a local read/write or an application of a remote update can proceed.

This division of policy into routing and blocking works well because it introduces a separation of concerns for a system designer. First, a system's trade-offs among performance, availability, and resource consumption goals largely map to routing rules. For example, Bayou sends all updates to all nodes to provide excellent response time and availability. Second, a system's durability and consistency constraints are naturally expressed as a combination of blocking predicates (e.g., block a read until it will return the most recent write), and routing rules (e.g., send a notification to all nodes caching a block when that block is updated).

Given a routing and blocking policy, PADS defines a runtime system that executes the policy across a set of nodes. The designer places her system's blocking predicates in a configuration file and she uses the PADS compiler to translate her routing rules into Java. She then distributes a Java jar file containing PADS's standard mechanisms and her system's policies to the system's nodes. Once the system is running at each node, users can read and write data via a local interface, and the system synchronizes data among nodes according to the policy.

2.1 Scope and limitations

PADS is not designed with the goal of aiding the construction of any conceivable replication system for any conceivable environment. By restricting our scope, we can craft abstractions that are well suited to a particular range of environments rather than building a generic programming system that can accommodate all designs but that is useful for none.

We target 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 where other abstractions may be of more immediate use to programmers [2].

We also restrict our attention to the construction of distributed storage systems with a read/write/delete interface, rather than trying to be an all-purpose distributed programming environment. Designers looking for a toolkit to help implement Paxos [16] should look elsewhere.

Within this scope, there are at least three properties that PADS does not address or for which it provides limited choice to designers: security, interface and conflict resolution. Thus, PADS may represent only a first step towards a fully general framework.

First, PADS does not support security specification. We believe that ultimately our policy architecture should also define flexible security primitives. Providing this capability is important future work, but it is outside the scope of this paper.

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

Third, PADS only assumes a simple conflict resolution mechanism. Write-write conflicts are detected and logged in a way that is data-preserving and consistent across nodes to support a broad range application-level resolvers. We implement a simple last-writer wins resolution scheme and believe that it is straightforward to extend PADS to support other schemes [5, 13, 14, 28, 31].

3 The PADS policy architecture

As discussed above, system development on PADS architecture entails specifying a per-system control policy over data plane as separate routing and blocking policies.

The data plane on which the control plane operates provides basic replication mechanisms in terms of 3 key abstractions. The data plane handles the details of

- Storing data locally.
- Sending and receiving updates among nodes.
- Maintaining consistency bookkeeping information.

The details of how the data plane implements these mechanisms [3, 22] are not the focus of this paper. What is important here is the abstractions it exposes to the control plane: it must expose an API that is simple, flexible, and efficient enough for a system designer to easily express her intent and for the runtime system to efficiently realize the intended design. The rest of this section details the routing and blocking abstractions PADS exposes to policy writers. We provide an example of how a designer uses this API to build a system in the next section.

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

Fig. 3: Routing actions provided by PADS.

3.1 Routing policy

In PADS, a routing policy sets up update flows among nodes to meet a designer's goals. The basic abstraction provided by the data plane for a data flow is *subscription* – a unidirectional stream of updates established between two nodes. 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. 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 exchanges at opportune times. Etc.

PADS's routing primitives consist of *actions* that establish or remove subscriptions to direct the communication of specific subsets of data among nodes and *triggers* that expose the status of local operations and information flow. A system's routing policy is specified as an event-based program that invokes actions in response to triggers received.

The basic idea of having an event driven program that responds to triggers is not new. Our design task in constructing PADS is to provide the right set of actions and triggers to control a replication data plane. PADS augments the basic event-driven programming model with a new abstraction: *stored events*, which allows a routing policy to store events in a named object in the underlying system and later cause the system to re-produce those events to trigger routing actions. This path between the data plane and the control plane allows routing policies to use persistent state (e.g., configuration files) and object-specific state (e.g., a directory of related files that should be replicated together) to drive its routing decisions.

Finally, to aid writing an event-driven routing program, we adopt the R/OverLog routing language. R/OverLog allows policy writers to construct routing policies in a concise manner.

3.1.1 Actions

The basic abstraction provided by a PADS action is simple: *an action sets up a subscription to route updates from one node to another.* The abstraction is so simple because the data plane handles the implementation details, allowing the system designer to focus on specifying what information should propagate to where. To that end,

the subscription actions API gives the designer 5 choices:

1. *Select invalidations or bodies.* In the data plane, each update comprises an invalidation and a body. An invalidation indicates that an update of a particular object occurred at a particular instant in logical time; invalidations help enforce consistency by notifying nodes of updates and by ordering the system's events. Conversely, a body contains the data for a specific update.
2. *Select objects of interest.* A subscription specifies which objects are of interest to the receiver, and the sender only includes updates for those objects. PADS exports a hierarchical namespace so a group of related objects can be concisely specified (e.g., /a/b/*).
3. *For a body subscription, select streaming or single-item mode.* A subscription for a stream of bodies sends updated bodies for the objects of interest until the subscription terminates; such a stream is useful for coarse-grained replication or for prefetching. Alternatively, a policy can send a single body by having the sender push it or the receiver fetch it. For reasons discussed in §3.2, invalidations are always sent in streams.
4. *Select the start time for a subscription.* A subscription specifies a logical start time, and the stream sends all updates that have occurred since that time.
5. *Specify a catchup mode for a subscription.* If the start time for a subscription is earlier than the sender's current logical time, then the sender can transmit either a *log* of the events that occurred between the start time and the current time or a *checkpoint* that includes just the most recent update to each byterange since the start time. Sending a log is more efficient when the number of recent changes is small compared to the number of objects covered by the subscription. Conversely, a checkpoint is more efficient if (a) the start time is in the distant past (so the log of events is long) or (b) the subscription is for only a few objects (so the size of the checkpoint is small). Note that once a subscription catches up with the sender's current logical time, updates are sent as they arrive, effectively putting all active subscriptions into a mode of continuous, incremental log transfer.

In addition to the interface for creating subscriptions, PADS provides actions to remove subscriptions, send an individual body or mark a previous update with a commit sequence number to aid in enforcing consistency [25]. Fig. 3 details the full routing actions API.

3.1.2 Triggers

Routing policies invoke PADS actions when PADS *triggers* signal important events that must be handled according to the system's design. These events fall into three categories.

Local Read/Write Triggers	
Read block	obj, off, len, EXIST|VALID|COMPLETE|SEQ
Write	obj, off, len, writerId, time
Delete	obj, writerId, time
Message Arrival Triggers	
Inval arrives	srcId, obj, off, len, writerId, time
Body send success	srcId, obj, off, len, writerId, time
Body send 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

Fig. 4: Routing triggers provided by PADS.

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

Fig. 5: PADS's stored events interface.

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

Fig. 4 details the full triggers API.

3.1.3 Stored events

Systems often need to maintain hard state to make routing decisions. Supporting this need is challenging both because we want an abstraction that meshes well with our event-driven, rule-based policy language and because the techniques must handle a wide range of scales. In particular, the abstraction must handle not only simple, global configuration information (e.g., the server identity in a client-server system like Coda [14]), but it must also scale up to per-volume or per-file information (e.g., which children have subscribed to which volumes in a hierarchical dissemination system [5, 21] or which nodes store the gold copies of each object in Pangaea [27].)

To provide a uniform abstraction to address this range of concerns, PADS provides *stored events.* To use stored events, policy rules produce one or more events that are stored into a data object in the underlying persistent object store. Rules also define when the events in an object should be retrieved, and the events thus produced can then trigger other policy rules. Fig. 5 details the full API for stored events.

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

Fig. 6: Performance for processing a NULL trigger to produce a NULL event.

3.1.4 Liveness policies in R/OverLog

PADS provides R/OverLog, a language and runtime to simplify writing event-driven routing programs.[2] R/OverLog is based on the OverLog routing language [18], and a R/OverLog program is a set of table declarations for storing tuples and a set of rules that specify how to create a new tuple or table update when an event occurs and the system's state matches some constraints.

R/OverLog extends OverLog by (1) adding type information to tuples, (2) providing an interface to pass *triggers*, *actions*, and *stored events* as tuples between PADS and the R/OverLog program, and (3) restricting the syntax slightly to allow us to implement a R/OverLog-to-Java compiler that produces executables that are more stable and faster than programs under the more general P2 [18] runtime system. Figure 6 compares the performance of the two systems for processing a simple event.

3.2 Blocking policy

A blocking policy specifies a set of *blocking predicates* that prevent state observation or updates until desired invariants are satisfied. Two sets of design choices determine the blocking policy interface: where operations can block and what conditions can the invariants be based on.

3.2.1 Blocking points

PADS defines five points for which a policy can supply a predicate and a timeout value that blocks a request until the predicate is satisfied or the timeout is reached. 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 results causally consistent with prior reads.*)
- *WriteEndBlock* blocks a write request after it has updated the local store 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.*)

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

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 object's consistency state reflects all updates before the node's current logical time
isSequenced	Block until object's total order is established
maxStale *nodes, count, t*	Block until I have received all writes up to $(operationStart - t)$ from *count* nodes in *nodes*.
User Defined Conditions on Local or Distributed State	
event *event-spec*	Block until an event matching *tuple-spec* is received from routing policy

Fig. 7: Conditions available for defining blocking predicates.

- *ApplyUpdateBlock* blocks an update received from the network before it is applied to the local store. Blocking here is useful for servers that must always be able to supply any data and for clients that want to be able to operate in disconnected mode (e.g., *block applying this invalidation until I have received the corresponding body.*)

PADS also provides *WriteBeforeBlock* to block a write before it modifies the underlying local store and *ReadEndBlock* to block a read after it has retrieved data from the local store but before it returns.

3.2.2 Blocking conditions

A blocking predicate can use any combination of the conditions listed in Fig. 7. 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 enforcing coherence on reads and for maximizing availability by ensuring that invalidations received from other nodes are not applied until they can be applied with their corresponding bodies [5, 21].
- *IsComplete* requires a node to receive all invalidations for the target object up to the node's current logical time. *IsComplete* is needed because liveness policies can direct arbitrary subsets of invalidations to a node, so a node may have gaps in its consistency state for some objects. If a node only reads objects for which *isValid* and *isComplete* is true, it is guaranteed to see FIFO consitency (aka writes-follow-writes aka PRAM) and also causal consistency.
- *IsSequenced* blocks until the most recent write to the target object has been assigned a position in a total order. Policies that want to ensure sequential or stronger consistency can use the *Assign Seq* routing action (see Fig. 3) to allow a node to sequence other nodes' writes and use the *isSequenced* condition to block reads of unsequenced data.

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

The *event* condition provides an interface to the control plane with which a routing policy can signal an arbitrary condition to a blocking predicate. An operation waiting for *tuple* unblocks when the routing rules produce an event matching a specified pattern.

Rationale. The built-in consistency bookkeeping primitives were chosen because they are simple and inexpensive to maintain within the data plane, but they 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 tuple *receivedAllAcks*).

Event 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 *events* from the event-driven routing policy is that when conditions reflect distributed state, policy designers can exploit knowledge of their system to produce better solutions than a generic topology-oblivious implementation of the same condition. For example, in the client-server system we describe in §5, to ensure its consistency semantics, a client blocks a write until it is sure that all other clients caching the object have been invalidated. Hence, when an object is written, all other clients send acknowledgements to the server when they receive an invalidation. The server gathers acknowledgements then generates a *receivedAllAcks* event for the client that issued the write so that the write can unblock.

4 Constructing P-TierStore

As an example on how to build a system with PADS, we describe our implementation of P-TierStore, a system inspired by TierStore [5].

4.1 System goals

TierStore is a distributed object storage system for developing regions where networks are bandwidth-constrained and unreliable. Each node reads and writes some specific subsets of the data. Since nodes must often operate in disconnected mode, ensuring 100% availability is more important than providing strong consistency.

4.2 System design

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.

4.3 Policy specification

In order to construct P-TierStore, we decompose the design into routing policy and blocking policy.

The 19 rule routing policy is responsible for establishing the publication aggregation and multicast trees. In terms of PADS primitives, each connection in the tree is simply an invalidation and a body subscription between nodes. Every PADS node stores, in configuration objects, the ID of its parent, the IDs of its children, and the set of publications to subscribe to. On start up, a node uses stored events to read the configuration objects and stores the configuration information in R/OverLog tables (6 rules). When it knows of the ID of its parent, it adds subscriptions for every item in the publication set (2 rules). For every child, it adds subscriptions for "/*" to receive all updates from the child (2 rules). If an application decides to subscribe to another publication, it simply writes to the configuration object. A new stored event is generated and the routing rules add subscriptions for the new publication (4 rules).

Recovery. Whenever an incoming or an outgoing subscription fails, the node periodically tries to re-establish the connection (1 rule). Crash recovery requires no extra policy rules. When a node crashes and starts up, it simply re-establishes the subscriptions. The underlying subscription mechanisms automatically detect which updates are missing and send them over.

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).

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 body is received.

TierStore v. 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. The

workaround is to define publications in separate subtrees and to stitch them together via symbolic links.

Also, as noted in §2.1, PADS provides a single conflict-resolution mechanism, which differs from TierStore's in some details. Similarly, TierStore provides native support for directory objects, while PADS supports a simple untyped object store interface.

5 Experience and evaluation

In this section, we explore PADS's usefulness as a platform for developing distributed storage systems. There is no quantitative way to prove that PADS is a better platform than any other, so we base our evaluation on our experience.

Fig. 1 conveys the main result of this paper: *using* PADS*, a small team was able to construct a dozen significant systems with a large number of features that cover a large part of the design space.* PADS qualitatively reduced the effort to build these systems and quantitatively increased our team's capabilities: we do not believe a small team such as ours could have constructed anything approaching this range of systems without PADS.

In the rest of this section, we detail this experience by first discussing the range of systems studied, the development effort needed, and our debugging experience. We then explore the realism of the systems we construct by examining how PADS handles key system-building problems like configuration, consistency, and crash recovery. Finally, we examine the costs of PADS's generality: what overheads do our PADS implementations pay compared to ideal or hand-crafted implementations?

Approach and environment. The goal of PADS is to help people develop new systems. 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 design using PADS.

The PADS prototype uses PRACTI [3] to provide the data plane mechanisms. We implement a R/OverLog to Java compiler using the XTC toolkit [9]. Except where noted, all experiments are carried out on machines with 3GHz Intel Pentium IV Xeon processors, 1GB of memory, and 1Gb/s Ethernet. Machines and network connections are controlled via the Emulab software. We use Fedora Core 8, BEA JRockit JVM Version 27.4.0, and Berkeley DB Java Edition 3.2.23.

5.1 System development on PADS

This section examines our experience by detailing the design space we have covered, how the agility of the resulting implementations makes them easy to change, the design effort needed to construct a system under PADS, and our experience debugging and analyzing our implementations.

5.1.1 Flexibility

We constructed systems chosen from the literature to cover large part of the design space. including client-server systems like We refer to our implementation of each system as P-system (e.g., P-Coda). To provide a sense of the design space covered, we provide a short summary of each of the system's properties below and in Figure 1.

Generic client-server. We construct a simple (P-SCS) and a full featured (P-FCS) version of a generic client-server system. Objects are stored on the server, and clients cache the data from the server on demand. Both systems implement *callbacks* in which the server keeps track of which clients are storing an object and sends invalidations to them whenever the object is updated. The difference between simple client server and full client server is that P-SCS assumes full object writes while P-FCS supports partial-object writes and also implements *leases* and *cooperative caching*. Leases [8] increase availability by allowing a server to break a callback for unreachable clients. Cooperative caching allows clients to retrieve data from a nearby client rather than from the server. Both P-SCS and P-FCS enforce demanding *sequential consistency* semantics and ensure durability by making sure that the server always holds the body of the most recently completed write of each object.

Coda [14]. Coda is a client-server system that supports mobile clients. Note that we implement Coda's client-server protocol, but we omit its server-to-server replication protocol. Coda is similar to P-FCS—it implements callbacks and leases but not cooperative caching; also, it guarantees *open/close consistency* instead of sequential consistency. A key feature of Coda is its support for *disconnected operation*—clients can access locally cached data when they are offline and propagate offline updates to the server on reconnection. Every client has a *hoard list* that specifies objects to be periodically fetched from the server to ensure that valid versions of these objects are locally cached.

TRIP [21]. TRIP 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 maximize the amount of data that a client can serve from its local state subject

to a staleness limit. TRIP guarantees sequential consistency via a simple algorithm that exploits the constraint that all writes are by a single server. TRIP was originally evaluated via simulation [21]; we believe that P-TRIP is the first implementation of the protocol.

TierStore [5]. TierStore is described in §4.

Chain replication [32]. Chain replication is a server replication protocol that guarantees linearizability and high availability. All the nodes in the system are arranged in a chain. Updates occur at the head and are only considered complete when they have reached the tail. Chain replication was originally evaluated via simulation [32].

Bayou [25]. Bayou 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 with whom to exchange updates via anti-entropy sessions.

Pangaea [27] Pangaea is a peer-to-peer distributed storage system for wide area networks. It employs a gossip-based protocol to propagate updates between replicas. Pangaea maintains a connected graph across replicas for each object, and it pushes updates along the graph edges. Pangaea maintains 3 gold replicas for every object to ensure data durability. Our implementation also includes the optimizations described in the paper including delta propagation and harbingers.

Summary of design features. As Fig. 1 further details, these systems cover a wide range of design features in a number of key dimensions. For example,

- *Replication:* full replication [21, 25, 32], partial replication [5, 14, 27] (and P-FCS), demand caching [14, 27] (and P-FCS), prefetching [5, 21, 27], and cooperative caching [27] (and P-FCS),
- *Topology:* structured topologies such as client-server [14, 21] (and P-FCS), hierarchical[5], and chain [32]; unstructured topologies [25, 27]. Invalidation-based [14] (and P-FCS) and update-based [5, 21, 25] propagation.
- *Consistency:* monotonic-reads coherence [5, 27], casual [25], sequential [21] (and P-FCS), and linearizability [32]; techniques such as callbacks [14, 21] (and P-FCS) and leases [14] (and P-FCS).
- *Availability:* Disconnected operation [5, 14, 21, 25], crash recovery (all), and network reconnection (all).

Goal: Architectural equivalence. We build systems based on the above designs from the literature, but constructing perfect, "bug-compatible" duplicates of the original systems using PADS is probably 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 2.1 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, except where explicitly noted above. 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 super-set of the set of data that may be accessed from the target system's local state. Notice that this property encompasses several factors including latency, availability, and durability.

There is a principled reason for believing that these properties capture something about the essence of a replication system: they highlight how a system resolves the fundamental CAP (Consistency vs. Availability vs. Partition-resilience) [7] and PC (Performance vs. Consistency) [17] trade-offs that any replication system must make.

5.1.2 Agility

As workloads and goals change, a system's requirements also change. We explore using PADS to adapt a system by adding new features. We highlight two cases in particular: our implementation of Bayou (P-Bayou) and Coda (P-Coda). Due to space constraints, we omit discussion of converting P-TRIP from a client-server to a hierarchical topology. Even though they are simple examples, they demonstrate that being able to easily adapt a distributed storage system to send the right data along the right paths can pay big dividends.

P-Bayou small device enhancement. P-Bayou is a server-replication protocol that exchanges updates between two servers via an anti-entropy protocol. Since anti-entropy propagates updates to the whole data set, P-Bayou cannot efficiently support smaller devices which have limited storage or bandwidth.

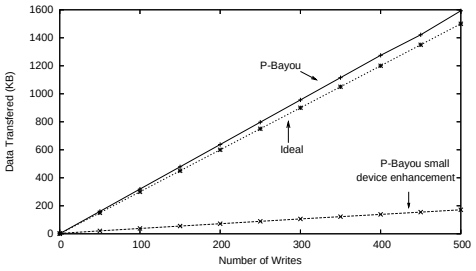

Fig. 8: Anti-Entropy bandwidth on P-Bayou

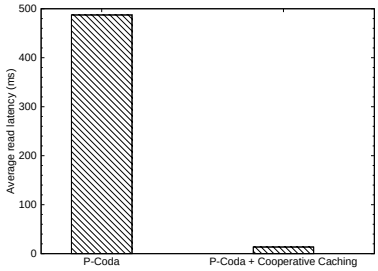

Fig. 9: Average read latency of P-Coda and P-Coda with cooperative caching.

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 any reachable peers and subscribes to receive invalidations and bodies for all objects using a subscription set "*". In our small device variation, a node instead reads a list of directories from a per-node configuration file (via stored events) and subscribes only for the listed subdirectories. This change required us to modify two routing rules.

This change raises a design question for the designer. If a small device S synchronizes with a first complete server $C1$, it will not receive updates to objects outside of its subscription sets. These omissions will not affect S since S will not access those objects. However, if S later synchronizes with a second complete server $C2$, $C2$ may end up with causal gaps in its update logs due to the missing updates. We have three choices: we can weaken consistency from causal to per-object coherence; we can restrict communication to avoid such situations (e.g., prevent S from synchronizing with $C2$); or we can weaken availability by forcing $C2$ to fill its gaps 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.

Fig. 8 examines the bandwidth consumed to synchronize 3KB files in P-Bayou and serves two purposes. First, it demonstrates that the overhead for anti-entropy in P-Bayou is relatively small even for small files compared to an *ideal* Bayou implementation (plotted by counting the bytes of data that must be sent ignoring all metadata overheads.) More importantly, it demonstrates that if a node requires only a fraction (e.g., 10%) of the data, the *small device enhancement*, which allows a node to synchronize a subset of data, greatly reduces the bandwidth required for anti-entropy.

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 13-rules: 9 to monitor the reachability of nearby nodes, 2 to retrieve data from a nearby client on a read miss, and 2 to fall back to the server if the client cannot satisfy the data request.

Fig. 9 shows the difference in read latency for misses with and without support for cooperative caching. For the experiment, the latency between the two clients is 10ms, whereas the latency between a client and server is 500ms. When data can be retrieved from a nearby client, read performance is greatly improved. More importantly, with this new capability, clients can share data even when disconnected from the server.

5.1.3 Ease of development

Each of these systems took a few days to three weeks to construct by one or two graduate students in part time effort. The time includes mapping the original system design to PADS policy primitives, implementation, testing and debugging. In our experience, mapping the design of the original implementation to routing and blocking policy was challenging at first but became progressively easier. Once the design work was done, the implementation did not take long since most of the low-level mechanisms were already provided by the data plane. As Fig. 1 indicates, each system was implemented in fewer than 100 routing rules and fewer than 10 blocking conditions.

5.1.4 Debugging and correctness

Three aspects of PADS's can simplify debugging and reasoning about the correctness of PADS systems.

First, the conciseness of PADS policy greatly facilitates analysis, peer review, and refinement of design. It was extremely useful to be able to sit down and walk through an entire design in a one or two hour meeting.

Second, the abstractions themselves divide work in a way that simplifies reasoning about correctness. For example, we find that the separation of policy into routing and blocking helps reduce the risk of consistency bugs. A consistency policy's safety conditions are specified and enforced by simple blocking predicates, so it is not difficult to get them right. We must then design our routing policy to deliver sufficient data to a node to eventually satisfy the predicates to ensure liveness.

Third, domain-specific languages can facilitate the use of model checking [4]. As future work, we intend to implement a translator from R/Overlog to Promela [1] so

that policies can be model checked to test the correctness of a system's implementation.

5.2 Realism

When building a distributed storage system, a system designer needs to address issues that arise in practical deployment such as configuration options, handling local crash recovery, distributed crash recovery, and, most importantly, maintaining consistency and durability during periods of crashes.

PADS makes it easy to tackle the above 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, publications a node wishes to access are stored in a configuration object. In P-Pangaea, each object's parent directory objects stores the list of nodes from which to fetch an object on a read miss.

Second, for consistency and handling crash recovery, the underlying subscription mechanisms insulates the designer from low-level details. Upon recovery, local mechanisms first reconstruct local state from persistent logs. Then, PADS's subscription primitives abstract away many of 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 local state reflects any updates that were missed.

Third, blocking predicates greatly simplify maintaining strong consistency during crashes. If there is a crash and the required consistency semantics cannot be guaranteed, the system will simply block access to "unsafe" data. On recovery, once the subscriptions have been restored so that the predicates are satisfied, data becomes accessible again.

Each of the PADS system we constructed, we implemented support for these practical concerns. Due to space limitations we focus this discussion on investigating the behaviour of two systems under failure. These systems include our implementations of the full-featured client server system (P-FCS) and TierStore (P-TierStore). Both are client-server based systems, but they have very different consistency guarantees. We demonstrate 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 consists of 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.

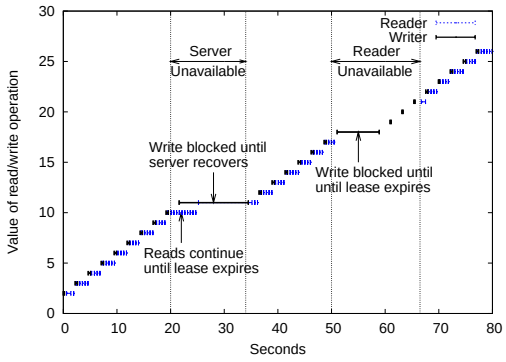

Fig. 10: Demonstration of full client-server system under failures. The x axis shows time and the y axis shows the value of each read or write operation.

Fig. 10 illustrates behaviour of P-FCS under failures. P-FCS guarantees sequential consistency by maintaining per-object callbacks [11], object leases [8] and blocking the completion of a write until the server has stored the write and invalidated all other client caches. We configure the system with a 5 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 then resumes processing requests. Both reads and writes resume shortly after the server restarts, and the subscription reestablishment and policy implementation ensure that consistency is maintained.

We kill the reader, C1, at 50 seconds and restart it 10 seconds later. Initially, writes block, but as soon as the lease expires, writes proceed. When the reader restarts, reads resume as well.

Fig. 11 illustrates a similar scenario using P-TierStore. P-TierStore enforces monotonic reads coherence rather than sequential consistency and propagates updates via subscriptions when the network is available. As a result, all reads and writes complete locally and without blocking despite 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.

5.3 Performance

We carry out performance evaluation of PADS in two steps. First, we evaluate the fundamental costs associated

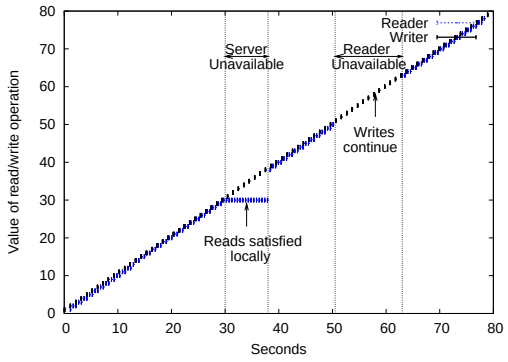

Fig. 11: Demonstration of TierStore under a workload similar to that in Figure 10.

	Ideal	PADS Prototype
Subscription setup		
Inval Subscription with LOG catchup	$O(N_{pUpdates})$	$O(N_{nodes}+N_{pUpdates})$
Inval Subscription with CP from time=0	$O(N_{obj})$	$O(N_{obj})$
Inval Subscription with CP from time=VV	$O(N_{objUpd})$	$O(N_{nodes}+N_{objUpd})$
Body Subscription	$O(N_{objUpd})$	$O(N_{objUpd})$
Transmitting updates		
Inval Subscription	$O(N_{nUpdates})$	$O(N_{nUpdates})$
Body Subscription	$O(N_{nUpdates})$	$O(N_{nUpdates})$

Fig. 12: Network overheads of primitives. Here, N_{nodes} is the number of nodes; N_{obj} is the number of objects in the subscription set. $N_{pUpdates}$ and N_{objUpd} 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; $N_{nUpdates}$ is the number of updates that to the subscription set that occur after the subscription has caught up to the sender's logical time.

with the PADS architecture. In particular, we demonstrate that network overheads of the PADS approach are within reasonable bounds of the best case implementations.

Second, we evaluate the absolute performance of the PADS prototype. We quantify overheads associated with the primitives via micro-benchmarks and compare the performance of two implementations of the same system: the original implementation with the one built over PADS. We find that P-Coda is as much as 4 times worse than Coda.

5.3.1 Fundamental overheads

PADS needs to have predictable costs. In particular, the amount of information stored and sent over the network should proportional to the amount of data a node is interested in. We first provide the cost model of PADS and then run experiments to confirm that the constant factors are indeed small.

Fig. 12 shows the cost model of our implementation of PADS's primitives and compares these costs to the costs of ideal implementations. As it can be seen from Fig. 12,

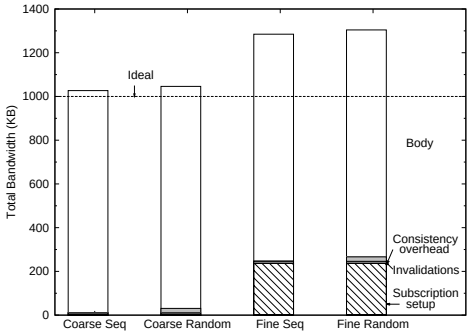

Fig. 13: Network bandwidth cost to synchronize 1000 10KB files, 100 of which are modified.

costs associated with PADS in most cases are competitive with ideal costs. Note that these ideal costs are optimistic and may not be able always be achievable.

There are two ways that PADS sends more information than an ideal or a hand-crafted implementation.

First, during invalidation subscription setup, PADS sends a version vector indicating the start time the subscription and catchup information so that the receiver can determine if the catchup information introduces gaps in the receiver's consistency state. That cost is then amortized over all the updates sent on the connection. Also, this cost can be avoided by starting a subscription at logical time 0 and with a checkpoint rather than a log for catching up to the current time. Note, checkpoint catchup is particularly cheap when interest sets are small.

Second, in order to support flexible consistency, invalidation subscriptions also carry extra information such as imprecise invalidations [3]. Imprecise invalidations summarize updates to objects out 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 precise invalidations sent. The size of imprecise invalidations is generally much smaller than bodies of updates and hence they do not impose a large overhead, as demonstrated in the next section.

5.3.2 Quantifying the overheads

We run experiments to investigate the constant factors in the cost model and quantify the overheads associated with subscription setup and flexible consistency. Fig. 13 illustrates the synchronization cost for a simple scenario. In this experiment, there are 10,000 objects in the system organized into 10 groups of 1,000 objects each, and each object's size is 10KB. The reader registers to receive invalidations for one of these groups. Then, the writer updates 100 of the objects in each group. Finally, the reader reads all 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-

	1KB objects		100KB objects	
	Coda	P-Coda	Coda	P-Coda
Cold read	13.59	44.2	28.78	46.11
Hot read	0.14	0.22	0.34	0.44
Hot Write	17.4	72.77	72.77	73.32
Disconnected Write	17.17	19.67	15.52	19.87

Fig. 14: Read and write latencies in milliseconds for Coda and P-Coda

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

Four things should be noted. First, the synchronization overheads are small compared to the body data transferred. Second, the "extra" overheads associated with PADS subsciption 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 they avoid per-object subscription setup costs.

Similarly, Fig. 8 compares the bandwidth overhead associated with using PADS system implementation with an ideal implementation. As the figure indicates, the bandwidth required by subscriptions to propagate updates comes close to ideal implementations. The extra overhead can be attributed to the small amount of metadata sent with each update.

5.4 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 to pay some overheads relative to a tuned kernel implementation 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 some of which may not be required by a hand-crafted system.

Fig. 14 compares the client-side read and write latencies under Coda and P-Coda. The systems are set up in a two client configuration. To measure the read latencies, we set up the scenario as follows: Client C1 has a collection of 1,000 objects/files and Client B has none. For cold reads, Client C2 randomly selects 100 objects/files to read. Each read fetches the object from the server and will establish a callback for the object. C2 re-reads those objects to measure the hot-read latency. To measure the hot-write latency, we set up the scenario as follows: Both C1 and C2 have the same collection of 1,000 objects/files. C2 selects 100 objects/files to write. The write will cause the server to break a callback with C1. Disconnected writes are measured by disconnecting C2 from the server and writing to 100 randomly selected objects. The values are averages of 5 runs.

As expected, PADS's a user-level Java implementation cannot compete with a hand-crafted C implementation. P-Coda's cold read performance is two times worse and the hot write performance is allmost 4 times worse than the original implementation for small files.

6 Related work

PADS and PRACTI. We use a modified version of PRACTI [3, 34] as PADSś data plane. Writing a new policy in PADS differs from constructing a system using PRACTI alone for three reasons that together reflect a major rethinking of the abstractions a data plane should export to a control plane.

1. PADS *adds key abstractions* not present in PRACTI such as the separation of routing policy from blocking policy, blocking predicates, stored events, and commit actions.
2. PADS *significantly changes* abstractions from those provided in PRACTI. For example, where PRACTI provides the abstraction of *connections* between nodes, each of which carries one subscription, PADS provides the abstraction of *subscriptions* and multiplexes subscriptions onto a single connection per pair of nodes, which enables fine-grained subscriptions and dynamically adding new items to a subscription. Similarly, where PRACTI provides the abstraction of *bound invalidations* to make sure that bodies and updates propagate together, PADS provides more flexible *blocking predicates*, and where PRACTI hard-coded several mechanisms to track the progress of updates through the system, PADS simply triggers the routing policy and lets the routing policy handle whatever notifications are needed.
3. PADS provides *R/OverLog* which has proven to be a convenient way to to think about, write, and debug routing policies.

The whole is more important than the parts. Building systems with PADS is much simpler than without. In some cases this is because PADS provides abstractions not present in PRACTI. In others, it is "merely" because

PADS provides a better way of thinking about the problem.

Other frameworks. A number of other efforts have defined frameworks for constructing distributed storage systems for different environments. Deceit [29] focuses on distributed storage across a well-connected cluster of servers. Stackable file systems [10] seek to provide a way to add features and compose file systems, but it focuses on adding features to local file systems.

Some systems, such as Cimbiosys, distribute data among nodes not based on object identifiers or file names, but rather on content-based filters. We see no fundamental issue in incorporating filters in PADS to identify a set 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.

PADS follows in the footsteps of efforts to define runtime systems or domain-specific languages to ease the construction of routing [18], overlay [26], cache consistency protocols [4], and routers [15].

7 Conclusion

Our goal is to provide a framework that allows developers to quickly build new distributed storage systems. This paper presents PADS a policy architecture that provides a control plane over which system design can be constructed in terms of routing and blocking policy. By providing the right policy abstractions, PADS allows developers to concentrate on policy without worrying about the complex low-level implementation details. Our experience suggests that PADS achieves our goal.

References

[1] Spin - formal verification. `http://spinroot.com/spin/whatispin.html`.

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

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

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

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

[6] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *SOSP*, 2003.

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

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

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

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

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

[12] A. Karypidis and S. Lalis. Omnistore: A system for ubiquitous personal storage management. In *PERCOM*, pages 136–147. IEEE CS Press, 2006.

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

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

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

[16] L. Lamport. Paxos made simple. *ACM SIGACT News Distributed Computing Column*, 32(4), Dec. 2001.

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

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

[19] D. Malkhi and D. Terry. Concise version vectors in WinFS. In *Symp. on Distr. Comp. (DISC)*, 2005.

[20] J. Mazzola, P. David, S.Tom, and Y. K. Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *IEE WMCSA*, 2003.

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

[22] M. D. N.Belaramani, J. Zheng. Feres: Flexible and efficient replica synchronization. In *In review*, 2008.

[23] E. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. OSDI*, Dec. 2004.

[24] N.Tolia, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proc. FAST*, pages 227–238, 2004.

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

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

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

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

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

[30] S. Sobti, N. Garg, F. Zheng, J. Lai, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: a distributed mobile storage system. In *Proc. FAST*, pages 239–252. USENIX Association, 2004.

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

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

[33] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP*, 2001.

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