PADS: A Policy Architecture for Distributed Storage Systems

Nalini Belaramani*, Jiandan Zheng§, Amol Nayate†, Robert Soulé‡,
Mike Dahlin*, Robert Grimm‡

*The University of Texas at Austin §Amazon.com Inc. †IBM TJ Watson Research
‡New York University

Abstract

This paper presents PADS, a *policy architecture* for building distributed storage systems. A policy architecture has two aspects. First, a common set of mechanisms that allow new systems to be implemented simply by defining new policies. Second, a structure for how policies, themselves, should be specified. In the case of distributed storage systems, PADS defines a *data plane* that provides a fixed set of mechanisms for storing and transmitting data and maintaining consistency information. PADS requires a designer to define a *control plane policy* that specifies the system-specific policy for orchestrating flows of data among nodes. PADS then divides control plane policy into two parts: *routing policy* and *blocking policy*. The PADS prototype defines a concise interface between the data and control planes, it provides a declarative language for specifying routing policy, and it defines a simple interface for specifying blocking policy. We find that PADS greatly reduces the effort to design, implement, and modify distributed storage systems. In particular, by using PADS we were able to quickly construct a dozen significant distributed storage systems spanning a large portion of the design space using just a few dozen policy rules to define each system.

1 Introduction

Our goal is to make it easy for system designers to construct new distributed storage systems. 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). To cope with these varying demands, new systems are developed [12, 14, 19, 21, 22, 30], each making design choices that balance performance, resource usage, consistency, and availability. Because these tradeoffs are fundamental [7, 16, 34], we do not expect the emergence of a single "hero" distributed storage system to serve all situations and end the need for new systems.

This paper presents PADS, a *policy architecture* that simplifies the development of distributed storage systems. A policy architecture has two aspects.

First, a policy architecture defines a common set of mechanisms and allows new systems to be implemented simply by defining new policies. PADS casts its 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 the details of storing and transmitting data and maintaining consistency information. System designers then build storage systems by specifying a control plane policy that orchestrates data flows among nodes.

Second, a policy architecture defines a framework for specifying policy. In PADS, we separate control plane policy into *routing* 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?", and 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.

The PADS prototype is an instantiation of this architecture. It provides a concise interface between the control and data planes that is flexible, efficient, and yet simple. For routing policy, designers specify an event-driven program over an API comprising a set of *actions* that set up data flows, a set of *triggers* that expose local node information, and the abstraction of *stored events* that store and retrieve persistent state. To facilitate the specification of event-driven routing, the prototype defines a domain-specific language that allows routing policy to be written as a set of declarative rules. For defining a control plane's blocking policy, PADS defines five *blocking points* in the data plane's processing of read, write,

	Simple Client Server	Full Client Server	Coda [14]	Coda +Coop Cache	TRIP [20]	TRIP +Hier	Tier Store [6]	Tier Store +CC	Chain Repl [32]	Bayou [23]	Bayou +Small Dev	Pangaea [26]
Routing Rules	21	43	31	44	6	6	14	29	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	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*		√	√	√	√	√	√	√	√	√	√	√

Fig. 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.

and receive-update actions; at each blocking point, a designer specifies *blocking predicates* that indicate when the processing of these actions must block.

Ultimately, the evidence for PADS's usefulness is simple: two students used PADS to construct a dozen distributed storage systems summarized in Figure 1 in a few months. PADS's ability to support these systems (1) provides evidence supporting our high-level approach and (2) suggests that the specific APIs of our PADS prototype adequately capture the key abstractions for building distributed storage systems. Notably, in contrast with the thousands of lines of code it typically takes to construct such a system using standard practice, given the PADS prototype it requires just 6-75 routing rules and a handful of blocking conditions to define each new system with PADS.

Similarly, we find it easy to add significant new features to PADS systems. For example, we add cooperative caching [5] to Coda by adding 13 rules.

This flexibility comes 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 [14]) in most cases and 3.3 times worse in the worst case we measured.

A key issue in interpreting Figure 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 Section 6. 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.

2 PADS overview

Separating mechanism from policy is an old idea. As Figure 2 illustrates, PADS does so by defining a data plane that embodies the basic mechanisms needed for storing data, sending and receiving data, and maintaining consistency information. PADS then casts policy as defining a control plane that orchestrates data flow among nodes. This division is useful because it allows the designer to focus on high-level specification of control plane policy rather than on implementation of low-level data storage, bookkeeping, and transmission details.

PADS must therefore specify an interface between the data plane and the control plane that is flexible and efficient so that it can accommodate a wide design space. At the same time, the interface must be simple so that the designer can reason about it. Section 3 and Section 4 detail the interface exposed by the data plane mechanisms to the control plane policy.

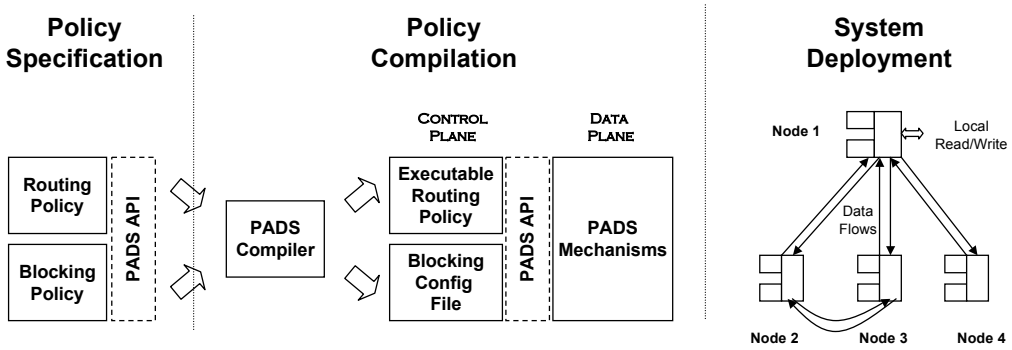

Fig. 2: PADS approach to system development.

To meet these goals and to guide a designer, PADS divides the control policy into a routing policy and a blocking policy. This division is useful 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, sending all updates to all nodes provides excellent response time and availability, whereas caching data on demand requires fewer network and storage resources. As described in Section 3, a PADS routing policy is an event-driven program that builds on the data plane mechanisms exposed by the PADS API to set up data flows among nodes in order to transmit and store the desired data at the desired nodes.

Second, a system's durability and consistency constraints are naturally expressed as conditions that must be met when an object is read or updated. For example, the enforcement of a specific consistency semantic might require a read to block until it can return the value of the most recently completed write. As described in Section 4, a PADS blocking policy specifies these requirements as a set of predicates that block access to an object until the predicates are satisfied.

Blocking policy works together with routing policy to enforce the safety constraints and the liveness goals of a system. Blocking policy enforce safety conditions by ensuring that an operation blocks until system invariants are met, whereas routing policy guarantee liveness by ensuring that an operation will eventually unblock—by setting up data flows to ensure the conditions are eventually satisfied.

2.1 Using PADS

As Figure 2 illustrates, in order to build a distributed storage system on PADS, a system designer writes a routing policy and a blocking policy. She writes the routing policy as an event-driven program comprising a set of rules that send or fetch updates among nodes when particular events exposed by the underlying data plane occur. She writes her blocking policy as a list of predicates. She then uses a PADS compiler to translate her routing rules into Java 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.

2.2 Policies vs. goals

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.

2.3 Scope and limitations

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 three design issues for which the current PADS prototype significantly 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 [18].

Second, the prototype exposes an object-store inter-

face 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, 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 [14, 31, 13, 28, 6].

3 Routing policy

In PADS, the basic abstraction provided by the data plane is *a subscription*—a unidirectional stream of updates to a specific subset of objects between a pair of nodes. A policy designer controls the data plane's subscriptions to implement the system's routing policy. 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.

Given this basic approach, the challenge is to define an API that is sufficiently expressive to construct a wide range of systems and yet sufficiently simple to be comprehensible to a designer. As the rest of this section details, PADS provides three sets of primitives for specifying routing policies: (1) a set of 7 *actions* that establish or remove subscriptions to direct communication of specific subsets of data among nodes, (2) a set of 9 *triggers* that expose the status of local operations and information flow, and (3) a set of 5 *stored events* that allow a routing policy to persistently store and access configuration options and information affecting routing decisions in data objects. Consequently, a system's routing policy is specified as an event-driven program that invokes the appropriate actions or accesses stored events based on the triggers received.

In the rest of this section, we discuss details of these PADS primitives and try to provide an intuition for why these few primitives can cover a large part of the design space. 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.

3.1 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 or *removes an established subscription* to stop sending updates. As Figure 3 shows, the subscription establishment API (*Add Inval Sub* and *Add Body Sub*) provides five parameters that allow a designer to control the scope of subscriptions:

Routing Actions	
Add Inval Sub	srcId, destId, objS, [startTime], LOG|CP|CP+Body
Add Body Sub	srcId, destId, objS, [startTime]
Remove Inval 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
B Action	<policy defined>

Fig. 3: 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.

- *Selecting the subscription type.* The designer decides whether *invalidations* or *bodies* of updates should be sent. Every 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 aid consistency enforcement by providing a means to quickly notify nodes of updates and to order the system's events. Conversely, a body contains the data for a specific update.
- *Selecting the source and destination nodes.* Since subscriptions are unidirectional streams, the designer indicates the direction of the subscription by specifying the source node (*srcId*) of the updates and the destination node (*destId*) to which the updates should be transmitted.
- *Selecting what data to send.* The designer specifies what data to send by specifying the *objects of interest* for a subscription so that only updates for those objects are sent on the subscription. PADS exports a hierarchical namespace in which objects are identified with unique strings (e.g., /x/y/z) and a group of related objects can be concisely specified. (e.g., /a/b/*).
- *Selecting the logical start time*. The designer specifies a logical *start time* so that the subscription can send all updates that have occurred to the objects of interest from that time. The start time is specified as a partial version vector and is set by default to the receiver's current logical time.
- *Selecting the catch-up method.* If the start time for an invalidation subscription is earlier than the sender's current logical time, the sender has two options: The sender can transmit either a *log* of the updates that have occurred since the start time or a *checkpoint* that includes just the most recent update to each byterange

Local Read/Write Triggers	
Operation block	obj, off, len, blocking_point, failed_predicates
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

Fig. 4: Routing triggers provided by PADS. *blocking_point* and *failed_predicates* indicate at which point an operation blocked and what predicate failed (refer to Section 4). *Inval | Body* indicate the type of subscription. *Reason* indicates if the subscription ended due to failure or termination.

since the start time. These options have different performance tradeoffs. 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 set consists of 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. For body subscriptions, if the start time of the subscription is earlier than the sender's current time, the sender transmits a checkpoint containing the most recent update to each byterange. The log option is not available for sending bodies. Consequently, the data plane only needs to store the most recent version of each byterange.

In addition to the interface for creating subscriptions (*Add Inval Sub* and *Add Body Sub*), PADS provides *Remove Inval Sub* and *Remove Body Sub* to remove established subscriptions, *Send Body* to send an individual body of an update that occurred at or after the specified time, *Assign Seq* to mark a previous update with a commit sequence number to aid enforcement of consistency [23], and *B Action* to allow the routing policy to send an event to the blocking policy (refer to Section 4). Figure 3 details the full routing actions API.

3.2 Triggers

PADS *triggers* expose to the control plane policy events that occur in the data plane. As Figure 4 details, these events fall into three categories.

- *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.

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. *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.

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

3.3 Stored events

Many systems need to maintain persistent state to make routing decisions. 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 [14]), but it must also scale up to per-file information (e.g., which nodes store the gold copies of each object in Pangaea [26].)

To provide a uniform abstraction to address this range of demands, PADS provides *stored events* primitives to store events into a data object in the underlying persistent object store. Figure 5 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.

For example, 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 event handlers in the routing policy that re-establish

subscriptions.

3.4 Specifying routing policy

A routing policy is specified as an event-driven program that invokes actions when local triggers or stored events are received. PADS provides R/OverLog, a language based on the OverLog routing language [17] and a runtime to simplify writing event-driven policies.[1]

As in OverLog, 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. This state does not need to be persistently stored, but is required for policy execution and can dynamically change. For example, a table might store the ids of currently reachable nodes. Rules are fired when an event occurs and the constraints associated with the rule are met. The input event to a rule can be a trigger injected from the local 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. For example, the following rule:

```
EVT_clientReadMiss(@S, X, Obj, Off, Len):-
        TRIG_operationBlock(@X, Obj, Off, Len, BPoint,_),
        TBL_serverId(@X, S),
        BPoint == "readNowBlock".
```

specifies that whenever node *X* receives a *operationBlock* trigger informing it of an operation blocked at the *readNowBlock* blocking point, it should produce a new event *clientReadMiss* at server *S*, identified by *serverId* table. This event is populated with the fields from the triggering event and the constraints—the client id (*X*), the data to be read (*obj, off, len*), and the server to contact (*S*). Note that the underscore symbol (_) is a wildcard that matches any list of predicates and the at symbol (@) specifies the node at which the event occurs. A more complete discussion of OverLog language and execution model is available elsewhere [17].

4 Blocking policy

A system's durability and consistency constraints can be naturally expressed as invariants that must hold when an object is accessed. In PADS, the system designer specifies these invariants as a set of predicates that block access to an object until the conditions 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 as predicates, and (3) exposes a *B_Action interface* that allows a designer to specify custom conditions based on routing information.

[1] 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
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

Fig. 6: Conditions available for defining blocking predicates.

The set of predicates for each blocking point makes up the blocking policy of the system.

4.1 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.*)
- *WriteEndBlock* 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 provides *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.

4.2 Blocking conditions

PADS provides a set of predefined conditions, listed in Figure 6, to specify predicates at each blocking point. A blocking predicate can use any combination of these predicates. 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 monotonic coherence on reads[2] and for maximizing availability by ensuring that invalidations received from other nodes are not applied until they can be applied with their corresponding bodies [6, 20].

- *IsComplete* requires that a node receives 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 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 want to ensure sequential or stronger consistency can use the *Assign Seq* routing action (see Figure 3) to allow a node to sequence other nodes' writes and specify the *isSequenced* condition as a *ReadNowBlock* predicate to block reads of unsequenced data.
- *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 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 [2, 35] 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 *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 system to produce better solutions than a generic implementation of the same condition. For example, in the client-server system we describe in Section 6, 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 *receivedAllAcks* event. Then, when an object is written and other clients receive an invalidation, they send acknowledgements to the server. When the server gathers acknowledgements from all other clients, it generates a *receivedAllAcks* action for the client that issued the write.

[2] Any read on an object will return a version that is equal to or newer than the version that was last read.

5 Constructing P-TierStore

As an example of how to build a system with PADS, we describe our implementation of P-TierStore, a system inspired by TierStore [6]. We choose this example because it is simple and yet exercises most aspects of PADS.

5.1 System goals

TierStore 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 the data. Since nodes must often operate in disconnected mode, the system prioritizes 100% availability over strong consistency.

5.2 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.

5.3 Policy specification

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

A 14-rule routing policy establishes and maintains the publication aggregation and multicast trees. A full listing of these rules is available elsewhere [3]. 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 stored events to read the configuration objects and store the configuration information in R/OverLog tables (4 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. 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). 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 data plane's 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).

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, as noted in Section 2.3, 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.

6 Experience and evaluation

Our central thesis is that it is useful to design and build distributed storage systems by specifying a control plane comprising a routing policy and a blocking policy. There is no quantitative way to prove that this approach is good, so we base our evaluation on our experience using the PADS prototype.

Figure 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 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 elaborate on 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 constructed 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 [2, 35] 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 [33]. For software, we use Fedora Core 8, BEA JRockit JVM Version 27.4.0, and Berkeley DB Java Edition 3.2.23.

6.1 System development on PADS

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

6.1.1 Flexibility

We constructed systems chosen from the literature to cover large part of the design space. 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 client-server (P-SCS) and a full featured client-server (P-FCS).

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 a valid version of an object and sends invalidations to them whenever the object is updated. The difference between P-SCS and P-FCS 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 [5] allows clients to retrieve data from a nearby client rather than from the server. Both P-SCS and P-FCS enforce *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. P-Coda includes the client-server protocol and the features described in Kistler et al.'s paper [14]. It does not include server replication features detailed in [27]. Our discussion focuses on P-Coda. P-Coda is similar to P-FCS—it implements callbacks and leases but not cooperative caching; also, it guarantees *open-to-close consistency*[3] 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

TRIP [20]. TRIP 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 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.

TierStore [6]. TierStore is described in Section 5.

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.

Bayou [23]. 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 to exchange updates with via anti-entropy sessions.

Pangaea [26] Pangaea is a peer-to-peer distributed storage system for wide area networks. Pangaea maintains a connected graph across replicas for each object, and it pushes updates along the graph edges. Pangaea maintains three gold replicas for every object to ensure data durability.

Summary of design features. As Figure 1 further details, these systems 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, P-FCS, and TierStore), demand caching (Coda, Pangaea, and P-FCS),
- *Topology:* structured topologies such as client-server (Coda, P-FCS, and TRIP), hierarchical (TierStore), and chain (Chain Replication); unstructured topologies (Bayou and Pangaea). Invalidation-based (Coda and P-FCS) and update-based (Bayou, TierStore, and TRIP) propagation.
- *Consistency:* monotonic-reads coherence (Pangaea and TierStore), casual (Bayou), sequential (P-FCS and TRIP), and linearizability (Chain Replication); techniques such as callbacks (Coda, P-FCS, and TRIP) and leases (Coda and P-FCS).
- *Availability:* Disconnected operation (Bayou, Coda, TierStore, and TRIP), 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 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 2.3 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

[3] Whenever a client opens a file, it always gets the latest version of the file known to the server, and the server is not updated until the file is closed.

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) [7] and PC (Performance vs. Consistency) [16] trade-offs that any distributed storage system must make.

6.1.2 Agility

As workloads and goals change, a system's requirements also change. We explore how systems build with PADS can be adapted by adding new features. 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 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 pairs of servers via an anti-entropy protocol. 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 subdirectories. 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 *S*1, 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 *S*2, *S*2 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: weaken consistency from causal to per-object coherence; restrict communication to avoid such situations (e.g., prevent *C* from synchronizing with *S*2); or weaken availability by forcing *S*2 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.

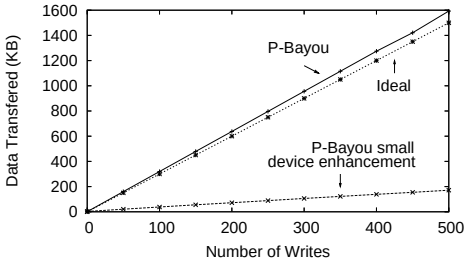

Fig. 7: Anti-Entropy bandwidth on P-Bayou

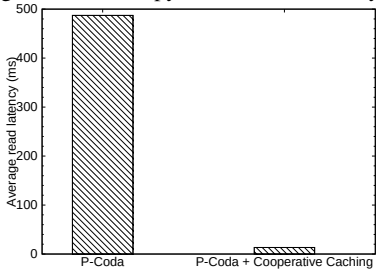

Fig. 8: Average read latency of P-Coda and P-Coda with cooperative caching.

Figure 7 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.

Figure 8 shows the difference in read latency for misses on a 1KB file with and without support for cooperative caching. For the experiment, the rount-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 performance is greatly improved. More importantly,

with this new capability, clients can share data even when disconnected from the server.

6.1.3 Ease of development

Each of these systems took a few days to three weeks to construct by one or two graduate students with part time effort. The time includes mapping the original system design to PADS policy primitives, implementation, testing, and debugging. 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.

Note that routing rules and blocking conditions are extremely simple, low-level building bocks. Each routing rule specifies the conditions under which a single tuple should be produced. R/Overlog lets us specify routing rules succinctly—across all of our systems, each routing rule is from 1 to 3 lines of text. The count of blocking conditions exposes the complexity of the blocking predicates: each blocking predicate is an equation across zero or more blocking condition elements from Figure 6, so the count of at most 10 blocking conditions for a policy indicates that across all of that policy's blocking predicates, a total of 10 conditions were used. As Figure 1 indicates, each system was implemented in fewer than 100 routing rules and fewer than 10 blocking conditions.

6.1.4 Debugging and correctness

Three aspects of PADS help 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 system's consistency and durability requirements 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 and 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.

6.2 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 nodes 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 consistency and crash recovery, the underlying subscription mechanisms insulate the designer from low-level details. 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 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 and the predicates are satisfied, the data become accessible again.

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 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 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 illustrates behavior of P-FCS under failures. P-FCS guarantees sequential consistency by maintaining per-object callbacks [11], maintaining object leases [8], and blocking the completion of a write until the server has stored the write and invalidated all other client

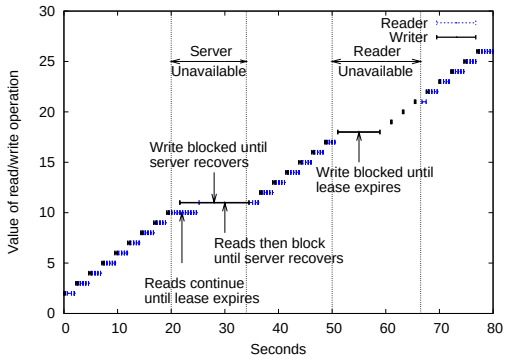

Fig. 9: Demonstration of full client-server system, P-FCS, under failures. The x axis shows time and the y axis shows the value of each read or write operation.

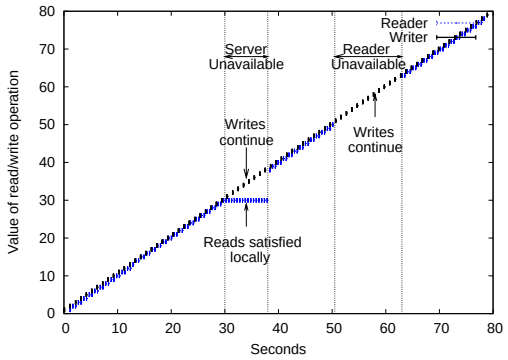

Fig. 10: Demonstration of TierStore under a workload similar to that in Figure 9.

caches. We configure 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 then 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, writes proceed. When the reader restarts, reads resume as well.

Figure 10 illustrates a similar scenario using P-TierStore. P-TierStore enforces monotonic reads coherence rather than sequential consistency, and it 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.

	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	$O(N_{SSNewUpdates})$	$O(N_{SSNewUpdates})$

Fig. 11: Network overheads of 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.

6.3 Performance

The programming model exposed to designers must have predictable costs. In particular, the volume of data stored and sent over the network should be proportional to the amount of information a node is interested in.

We carry out performance evaluation of PADS in two steps. First, we evaluate the fundamental costs associated with the PADS architecture. In particular, we argue that network overheads of PADS are within reasonable bounds of ideal implementations and highlight when they depart from ideal.

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 3.3 times worse than Coda.

6.3.1 Fundamental overheads and scalability

Figure 11 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.

There are two ways that PADS sends extra information.

First, during invalidation subscription setup in PADS the sender transmits a version vector indicating the start time of the subscription and catch-up information so that the receiver can determine if the catch-up information introduces gaps in the receiver's consistency state. That

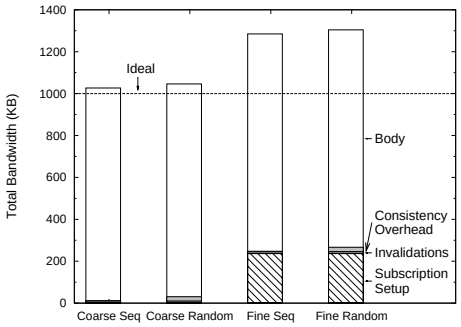

Fig. 12: Network bandwidth cost to synchronize 1000 10KB files, 100 of which are modified.

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 with a checkpoint rather than a log for catching up to the current time. Note, checkpoint catch-up 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 [2]. 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 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.

Overall, we expect PADS to scale well to systems with large numbers of objects or nodes—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), and the per-node overheads associated with the version vectors used to set up some subscriptions can be amortized over all of the updates sent.

6.3.2 Quantifying the constants

We run experiments to investigate the constant factors in the cost model and quantify the overheads associated with subscription setup and flexible consistency. Figure 12 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-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 a random order.

	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)*

Fig. 13: Read and write latencies in milliseconds for Coda and P-Coda. The numbers in parantheses indicate factors of overhead. The values are averages of 5 runs.

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.

Similarly, Figure 7 compares the bandwidth overhead associated with using a PADS system implementation with an ideal implementation. As the figure indicates, the bandwidth to propagate updates is close to ideal implementations. The extra overhead is due to the meta-data sent with each update.

6.3.3 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 to pay some overheads 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 particular hand-crafted system.

Figure 13 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 la-

tencies, 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 hand-crafted C implementation in most cases and is at most 3 times worse in the worst case we measured.

7 Related work

PADS and PRACTI. We use a modified version of PRACTI [2, 35] as the data plane for PADS. Writing a new policy in PADS differs from constructing a system using PRACTI alone for three reasons.

1. PADS *adds key abstractions* not present in PRACTI such as the separation of routing policy from blocking policy, stored events, and commit actions.
2. PADS *significantly changes* abstractions from those provided in PRACTI. We distilled the interface between mechanism and policy to the handful of calls in Figures 3, 4, and 5, and we changed the underlying protocols and mechanisms to meet the needs of the data plane required by PADS. For example, where the original PRACTI protocol provides the abstraction of *connections* between nodes, each of which carries one subscription, PADS provides the more lightweight abstraction of *subscriptions* which forced us to redesign the protocol to multiplex subscriptions onto a single connection between a pair of nodes in order to efficiently support fine-grained subscriptions and dynamic addition of 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 the 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 design 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.

R/OverLog and OverLog R/OverLog extends OverLog [17] by (1) adding type information to events, (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 [17] runtime system.

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 [24], 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.

PADS follows in the footsteps of efforts to define runtime systems or domain-specific languages to ease the construction of routing [17], overlay [25], cache consistency protocols [4], and routers [15].

8 Conclusion

Our goal is to allow developers to quickly build new distributed storage systems. This paper presents PADS, a policy architecture that allows developers to construct systems by specifying policy without worrying about complex low-level implementation details. Our experience has led us to make two conclusions: First, the approach of constructing a system in terms of a routing policy and a blocking policy over a data plane greatly reduces development time. Second, the range of systems implemented with the small number of primitives exposed by the API suggest that the primitives adequately capture the key abstractions for building distributed storage systems.

Acknowledgements

The authors would like to thank the anonymous reviewers whose comments and suggestions have helped shape this paper. We would also like to thank Petros Maniatis and Amin Vahdat for their valuable insights in the early drafts of this paper. Finally, we would like to thank

our shepherd, Bruce Maggs. This material is based upon work supported by the National Science Foundation under Grants No. IIS-0537252 and CNS-0448349 and by the Center of Information Assurance and Security at University of Texas at Austin.

References

[1] http://spinroot.com/spin/whatispin.html.

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

[3] N. Belaramani, J. Zheng, A. Nayate, R. Soule, M. Dahlin, and R. Grimm. PADS: A Policy Architecture for Distributed Storage Systems. Technical Report TR-09-08, U. of Texas at Austin, Feb. 2009.

[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. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proc. OSDI*, pages 267–280, Nov. 1994.

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

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

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

[18] 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, Oct. 2008.

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

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

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

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

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

[24] V. Ramasubramanian, T. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. Technical report, Microsoft Research, 2008.

[25] 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.

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

[27] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for distributed workstation environments. *IEEE Trans. Computers*, 39(4), 1990.

[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] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

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

[35] J. Zheng, N. Belaramani, and M. Dahlin. Pheme: Synchronizing replicas in diverse environments. Technical Report TR-09-07, U. of Texas at Austin, Feb. 2009.