Pheme: Synchronizing Replicas In Diverse Environments

Jiandan Zheng
Amazon Inc.
jiandanz@amazon.com

Nalini Belaramani, Mike Dahlin
The University of Texas at Austin.
nalini,dahlin@cs.utexas.edu

Abstract: This paper presents Pheme[1], a peer-to-peer data synchronization protocol that can be used to construct new distributed file systems that share data across collections of devices with limited, varying, or intermittent connectivity.

The suitability of Pheme for such environments is not matched by existing protocols. With current technology trends, devices will almost-always be connected to a server or to another device, synchronizing different sets of data as they please. Existing protocols often assume a coarse-grained synchronization model which makes changing the data set of an already established synchronization stream expensive. Pheme introduces fine-grained synchronization so that devices can cheaply change the data set they want to synchronize while they are connected to the other device. In addition, it uses a new dependency summary vector (DSV) scheme to detect conflicts with no extra overhead despite network interruptions and exposes flexible commit mechanisms to allow applications to implement their own commit schemes.

Because Pheme provides developers with various synchronization options, systems can be built that send the right data via the right network paths and dramatically outperform traditional client-server or server-replication protocols. At the same time, Pheme is more efficient than existing similarly flexible protocols like PRACTI – we observe several orders of magnitude of bandwidth savings in some experiments.

1 Introduction

This paper addresses a simple question: How can a replication protocol support the diverse needs of personal and mobile storage environments? Users are increasingly storing and accessing data from a large collection of devices with vastly different network capabilities including laptops, phones, eBooks, media-players, set-top boxes. These devices sometimes operate independently, but they sometimes share data with a local server, a cloud server, or with other devices. The key challenge for file systems targeting personal and mobile environments is the diversity the environment presents both in the dimension of device characteristics i.e. mobility, connectivity, and storage capacity, and in the dimension of application needs i.e. consistency, performance, and availability.

[1]Pheme is the Greek goddess of fame, renown, and gossip. She has been described as she who initiates and furthers communication.

Our vision is to construct a synchronization protocol that meets these needs of a wide range of applications targeting this environment. PRACTI [1] comes close. PRACTI allows devices to store different subsets of data (*partial replication*) and to synchronize with any other device (*topology independence*) while providing flexible consistency guarantees (*any consistency*) at reasonable costs. It provides synchronization optimizations that provide good performance, including partial replication of both data and metadata, separation of meta-data and data synchronization, and a hybrid log-based and state-based synchronization protocol.

Unfortunately, PRACTI has a significant limitation that makes it less than ideal in this environment. PRACTI, like many other existing protocols [17, 19], assumes a coarse-grained synchronization model – once two devices have a synchronization stream established, it is expensive to change the data set they want to synchronize. The coarse-grained model works well for environments in which devices operate in disconnected mode most of the time and occasionally connect with another device or a server to synchronize data. However, considering current technology trends where devices have more than one networking capability, such as WiFi, 3G, and Bluetooth, we expect that devices will be most of the time connected to another device or a server. In such environments, the coarse-grained model falls short. Devices may dynamically change the synchronization set due to a user or application directive. Also, common replication techniques such as callbacks and demand caching require fine-grained changes to the synchronize set between two devices. The workaround is to establish separate synchronization streams for every new synchronization set. However, the workaround, has high overheads because redundant consistency information is sent on each stream (explained in Section 4). For other protocols [17, 19], establishing separate streams does not provide any ordering guarantees between the updates received on different streams.

We present Pheme, a new synchronization protocol that supports both coarse-grained and fine-grained synchronization. In order to support fine-grained synchronization, Pheme uses a new technique to *multiplex synchronization requests* on to a single stream – if two nodes have a synchronization stream established and a request to synchronize a new data set is received, the new data set is added to the same stream. Synchronization streams in Pheme maintain the prefix property [19] and send up-

dates in causal order. The technical challenge lies in maintaining the prefix property and causal consistency even when new data sets are incorporated into the stream.

Pheme is also able to naturally deal with other challenges imposed by the environment.

Disconnections are common in such environments. On reconnection, it is necessary to detect conflicting updates so that appropriate mechanism can be invoked to reconcile the differences. Pheme employs a new *dependency summary vector (DSV)* scheme to detects conflicts without putting any extra network or storage overhead. In fact, the scheme is as efficient as existing protocols, but unlike some [17], the overheads do not increase with network disconnections.

No all applications have the same consistency or durability requirements. Some applications may require stronger consistency guarantees, such as serializability, when devices are connected to a server, or that all updates be marked as tentative until they reach a specific device [19]. Pheme incorporates a *flexible commit mechanism* for applications to implement their own commit schemes. In fact, Pheme takes advantage of the causality of synchronization to propagate commit information in casual order making it easier to implement stronger consistency semantics.

This paper presents details of Pheme and evaluates it under different synchronization scenarios. Details of how to implement specific replication policies with the Pheme mechanisms can be found elsewhere [2]. We demonstrate that Pheme an ideal choice for synchronization in mobile environments: Pheme is able to support a wide range of synchronization options efficiently. Pheme, like PRACTI, provides significant speedup and bandwidth savings when compared to traditional client-server and server-replication protocols. However, unlike PRACTI, Pheme provides up to tens of times of bandwidth savings for workloads that dynamically establish fine-grained subscriptions.

2 Design Requirements

Replication engines targeting personal and mobile environments must deal with the diversity the environment presents in terms of device mobility, connectivity and storage capacities. In addition, applications have vastly different consistency, availability and performance needs.

The key challenge for a replication protocol is to be sufficiently flexible and efficient so as to greatly simplify the development of replication engines for this environment. In particular, it must support the following features:

- *Partial replication:* Every device may store different subsets of data, either due to storage limitations or user preference. Ideally, the protocol should allow any device to store any subsets of data.
- *Wide range of consistency semantics:* Users' natural expectations is to see updates to different objects in order. Also, different applications have different consistency requirements. The protocol should allow applications that require strong consistency guarantees to provide them, whereas others should not have to pay the availability or performance costs of the consistency guarantees they do not need.
- *Arbitrary synchronization topologies:* It is often necessary for a device to be able to directly synchronize with a nearby peer when connectivity to the server is limited. The protocol should allow any device to carry out synchronization with any other node.
- *Various synchronization options:* Every replication system may have different synchronization requirements. The protocol must provide sufficient options so that policies implement synchronization schemes with the best tradeoffs including: the *separation of data and meta-data paths* so that a device can inform another device about an update without having to send the entire update, *log-based and state-based synchronization* so that the system can pick the one which is better suited for the experienced workload, and, most important, *dynamic synchronization establishment* so that two devices can efficiently synchronize a new data set while the synchronization of another set is taking place.
- *Incremental progress:* Network interruptions are commonplace in mobile environments. The protocol must make sure that synchronization makes progress despite network disruptions.
- *Conflict detection:* Due to network partitions, a data item may be concurrently updated at multiple devices. The protocol must detect conflicting updates so that applications can invoke the appropriate resolution mechanisms.
- *Application-specific commit policies:* Some applications differentiate between tentative and committed writes and implement their own commit policies. The protocol must allow engines to implement their own commit policies easily.
- *Low overheads:* Network and resource limited devices are common in this environment. The flexible protocol should be competitive with non-flexible, hand-crafted protocols.

In short, the synchronization protocol must allow any device to synchronize any subset of data with any other node and provide options to chose either data or meta-data and log-based or state-based synchronization while detecting conflicts and supporting application specific commit policies at reasonable costs. Previous efforts provide a subset of these requirements [8] [10] [11] [15]. Because they do not provide the whole set of features,

they impose restrictions on how synchronization should be carried out and thus cannot be used as general-purpose protocols.

Pheme is the first to provide the whole list of flexibility to system designers. The details of how Pheme supports these features is presented in later sections. Building a replication engine with Pheme becomes as simple as specifying when, where and which node to synchronize with and with what options. Details on how to build replication engines with Pheme is provided in [2]

3 System Model

Objects and time. Data are stored as objects identified by unique object identifier strings. Sets of objects can be compactly represented as *interest sets* that impose a hierarchical structure on object IDs. For example, the interest set "/a/*:/b" includes object IDs with the prefix "/a/" and also includes the object ID "/b".

Pheme heavily relies on Lamport's clocks [12] and version vectors to keep logical time and consistency information. Every node maintains a *time stamp*, $lc@n$ where lc is a logical counter and n the node identifier. To allow events to be causally ordered, the time stamp is incremented whenever a local update occurs and advanced to exceed any observed event whenever a remote update is received. Every node also maintains a version vector, $currentVV$, that indicates all the updates, local or remote, it is aware of.

Whenever an object is updated, the update is divided into an *invalidation* and a *body*. An invalidation contains the object ID and the logical time of the update. A body contains the actual data of the update.

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

The *object store* stores the latest bodies of objects the node chooses to replicate along with per-object meta-data used to ensure consistency of that data (described later).

The *consistency module* keeps track of other consistency information in a concise manner by organizing objects into hierarchical interest sets and storing the information on a per-interest set basis. Details of the consistency information maintained is provided in Section 4.2.

Failures. Pheme ensures progress despite network disconnections and device failures. On reconnection after a network disruption, synchronization continues where it left off. On recovery after a failure, the device consults the log to reconstruct its consistency state. Pheme does not handle byzantine failures.

4 Synchronization

Pheme carries out synchronization between two nodes via ordered, unidirectional. Pheme uses techniques used by past protocols including peer-to-peer synchronization [7, 8, 19, 21, 22] via log exchange [19] and state exchange [17], separation of invalidations and bodies [1, 11], causal propagation of updates [1, 19] and summarization of unwanted meta-data [1]. However, to meet all its design requirements, Pheme introduces a new technique: multiplexing synchronization requests on a single stream while maintaining the casual order.

In this section, we describe the synchronization protocol and provide details of the processing that occurs at a node when updates are sent and received.

4.1 Protocol Overview

Say a node wants to receive updates to a subset of data from another node. In addition to the updates, it is also necessary to send enough information to give each application the flexibility to enforce whatever level of consistency it needs. Synchronization in Pheme tries to keep that information to a minimum.

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

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

Gap markers. Because of partial replication, however, different nodes may store different subsets of data. In other words, Node B may not store all the objects that Node A wants. In addition to sending all the updates to objects in $A.IS$, Node A must warn Node A if there are any causal gaps – updates that have occurred to objects that Node A cares about but Node B does not have. Node B sends a *gap marker* [1] in that case. Gap markers can be seen as a summary of multiple updates. They summarize updates that occurred to a set of objects, $targetSet$, between a start time, $startVV$, and an end time, $endVV$. Note that start and end time are partial version vectors rather than full version vectors.

Gap markers must be sent in two cases. First, the sender, Node B, does not know about the updates the receiver, Node A, has asked for. Second, gap markers are sent for updates to objects Node A did not ask for so that

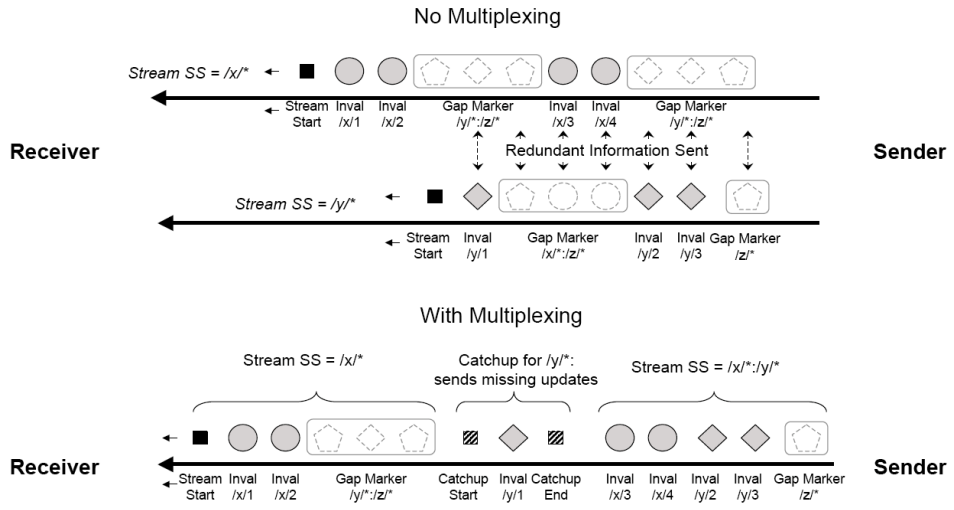

Fig. 1: Diagram comparing the messages sent on invalidation streams without and with multiplexing.

Node A can pass on the knowledge of the gaps in its information to another node, Node C, that it synchronizes with in the future. It is this propagation of gap markers allows Pheme to simultaneously support partial replication, topology independence, and still support a broad range of consistency semantics.

An synchronization stream consists of a start time, $stream.startVV$, and a causally ordered series of updates and gap markers. Every stream is associated with a $stream.VV$ that keeps track of the logical time progression of the stream.

Dynamic synchronization. Next, what if Node A decides that it wants to synchronize more objects? Say Node A already has a synchronization stream established for $A.IS$ from Node B and it wants get updates for objects in $A.newIS$ from the time $A.newIS.startVV$. A simple approach would be to establish a new stream for $A.newIS$. The problem is that a lot of redundant information will be sent – in order to ensure that each stream provides a causally ordered series of events from $stream.startVV$, each must include gap markers for any omitted events and Node A may receive gap markers for the same updates twice. If Node A creates a large number of subscriptions, then redundant information is sent on each stream.

Pheme reduces the overheads of dynamic synchronization requests by multiplexing requests on to a single stream. Since updates on the stream are sent in causal order, simply sending updates to $A.newIS$ from the current logical time of the stream $stream.VV$ will not work because that would not "fix the gap" that Node A has for $A.newIS$ from $A.newIS.startVV$ to $stream.VV$. Instead, Node B "pauses" the stream at $stream.VV$ and sends a *catchup stream* which includes all updates for $A.newIS$ from $A.newIS.startVV$ to $stream.VV$ that node B knows about. After catchup, node B continues sending invalidations for $A.IS$ and $A.newIS$ and gap markers for everything else starting from $stream.VV$. Figure 1 illustrates the multiplexing of the stream.

State-based synchronization. Pheme also supports state-based synchronization by sending a checkpoint of the final state of objects in an interest set instead of sending an ordered log of updates. A checkpoint consists of a gap marker for $A.IS$ from $A.stream.startVV$ to $B.currentVV$ and the latest meta-data of objects in $A.IS$ updated between $stream.startVV$ and $B.currentVV$. In case of a multiplexed catchup stream, a checkpoint consists of the meta-data of objects in $A.newIS$ updated between $A.newIS.startVV$ and $stream.VV$.

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

Invalidation and body streams. Pheme separates meta-data and data synchronization by having separate

Messages sent on an invalidation stream	
Subscription start	SS, startVV
Checkpoint	SS, per-obj meta-data, IS gap information
Invalidation	objId, offset, length, timeStamp
Gap marker	targetSet, startVV, endVV
Catchup start	SS, VV
Catchup end	VV
Messages sent on an body stream	
Body	objId, offset, length, timeStamp, data

Fig. 2: Components of different messages sent on subscription streams.

invalidation and body streams. Invalidation streams propagate meta-data of updates that occurred after *startVV* in causal order. Body streams are simply unordered streams of the bodies of updates that occurred after *startVV*. Ordering body streams is unnecessary because received bodies are applied to the object store only after their corresponding invalidations are received. The causality of invalidation streams is sufficient to guarantee consistency.

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

4.2 Protocol In Action

This section describes the protocol in action. It explains how a node initiates synchronization (via subscriptions), the state maintained at each node, and the processing carried out when messages are received.

Subscriptions. A synchronization request is called a subscription and is associated with a subscription set that specifies the set of objects, *SS*, a node is interested in synchronizing and a start time, *startVV*, which indicates that only the updates that occurred after that time should be sent. Subscriptions between two nodes are multiplexed on a single update stream. A subscription has two phases: a *catchup phase* in which the sender sends all updates to objects in the subscription set from the start time *startVV* to the sender's *currentVV*, and a *connected phase* in which the sender forwards any new updates it receives.

For invalidation subscriptions, in the catchup phase, the subscription can either carry out log-synchronization – with invalidations and gap markers sent causally over the stream, or checkpoint-synchronization – with a checkpoint of the object meta-data sent instead. In the connected phase, only invalidations and gap markers are sent. In particular, an invalidations stream can be made up of the following messages:- a *subscription start* message that includes the subscription start time, *startVV*, and the subscription set, *SS*; a *checkpoint* of the the sender's metadata for *SS*; *invalidations* of updates to objects in *SS* that the sender knows about; *gap markers* of updates out of *SS* or if the sender is missing invalidations of updates to *SS*; and perhaps a *catchup stream*, enclosed by *catchup start* and *catchup end* messages, if a new subscription set is multiplexed to the stream. Figure 2 provides details of the messages sent.

A body stream simply consists of bodies of updates that occurred after the *startVV* of each subscription multiplexed on it.

Node state. Every node maintains state to keep track of its logical time and the consistency of the objects it stores. Whenever it receives messages over streams, it updates the consistency state and the object store accordingly. In particular, a node, Node A, maintains the following state:

- *currentVV*: Node A maintains a version vector that indicates the latest update it has seen, either via an invalidation or via a gap marker. This implies that Node A is *not* aware of any updates after *currentVV*.
- *stream.VV*: For every invalidation stream, Node A maintains a logical time that includes the last update and all the causally preceding updates received on the stream. It implies that Node A has seen, either via invalidations or gap markers, all events from the *stream.startVV* to *stream.VV*.
- *IS.noGapVV*: For every interest set, Node A maintains a *noGapVV* that indicates that Node A has seen all updates and no gaps to the interest set until this time. *IS.noGapVV* is maintained in the consistency module. For a particular interest set IS_1, if $IS_1.noGapVV < currentVV$ then the interest set is considered *gapped* – Node A is missing one or more invalidations that affect IS_1 between $IS_1.noGapVV$ and *currentVV*. Hence, consistency cannot be assured for reads of objects in IS_1.
- *obj.timeStamp*: For every object currently stored, Node A stores the timestamp of the *latest* invalidation it has received for the object.
- *obj.isValid*: A flag, stored for every object, that indicates whether the node stores the body of the latest invalidation it has received for the object. If the *isValid* flag is not set, the object is considered *invalid* and the consistency cannot be assured for a read of that object, because the body is older than the invalidation received.

In fact, causal consistency can be guaranteed for reads to *valid* objects in *not gapped* interest sets. Because the interest set is not gapped, the node is aware of all the causal updates to the object up to *currentVV* and the validity implies that the node is actually storing the body of the latest causal update. Furthermore, causal consistency provides a baseline over which stronger guarantees like sequential consistency or linearizability can easily be

added [26]. On the other hand, applications that do not require causal consistency have the option of accessing data even from gapped interest sets.

A checkpoint for a subscription set *SS* consists of *noGapVV* information for every enclosed interest set, and the object meta data i.e. $(timestamp, isValid)$ for every object updated after $startVV$.

Processing received updates. When a receiver receives messages on the stream, in addition to updating the log and the store, the key job for the receiver is to make sure that the consistency state is correctly updated.

In order to eliminate the need for updating $IS.noGapVV$ every time an invalidation or gap marker is received, we introduce the concept of "attaching" an interest set to a stream. An interest set is "attached" to a stream if no gap markers for the interest set have been received on the stream, i.e. $IS.noGapVV$ includes $stream.VV$. The consistency module keeps track of which streams an interest set is attached to by maintaining a $IS.attachedStreams$ set. If a gap marker, *GM* is received on a stream, the $GM.targetSet$ is "detached" from the stream by explicitly storing its *noGapVV* in the consistency module.

An invalidation stream is, therefore, processed as described in Figure 3.

Processing a body stream is simple. When a node receives a body, it will check if the *body.timeStamp* matches local times stamp for the object, $obj.timeStamp$. If there is a match, it implies that the body corresponds to the latest received invalidation for the object and the body is put into the store. If the body is older than the timestamp, then the body is discarded. If the body is newer than the timestamp, it implies that its corresponding invalidation has not been received yet. Instead of discarding it, the body is stored in a body buffer and is applied to the store when its corresponding invalidation arrives.

Sending updates. For invalidation subscriptions, a sender iterates through the entries in its log from the subscription start time $stream.startVV$ to $currentVV$. Invalidations and gap markers of updates to objects in $stream.SS$ are sent as is. Invalidations of object not in $stream.SS$ are summarized into gap markers before being sent. For checkpoint catch-up, the sender creates a checkpoint by looking through the object store and consistency module and sends per-object state of objects updated after $stream.startVV$ and the *noGapVV* information on the stream.

When a body subscription is initiated, the sender searches through the object store and sends bodies of all valid objects that are in $stream.SS$ and whose timestamp is newer than $stream.startVV$. Note that bodies of invalid objects are not sent because the object store keeps track only of latest timestamp per object and once a body has been invalidated, it could be much older than $stream.startVV$.

5 Conflict Detection

Conflict detection is an important feature for synchronization protocols. An object may be independently updated on multiple nodes leading to diverging versions. Updates are considered to be conflicting if there is no causal relationship between these updates. Such conflicts need to be detected so that appropriate resolution, either automatic or manual, can be invoked to resolve the differences and achieve eventual consistency [11] [23].

Existing protocols [1, 6, 9, 14] often store or transmit extra information for the sole purpose of conflict detection. For mobile environments where network bandwidth and storage capacity can be limited, it is important that such information be minimal. Fortunately, Pheme can detect conflicts without having to store or transmit any extra information – it simply derives the information it needs from the state already maintained for consistency.

Conflict detection is carried out as described below. If no conflict is detected, the received update is applied, else the conflict flag set and all the information is stored in a special file for resolution. Pheme provides mechanisms for conflict detection, but it leaves conflict resolution to application specific policies. For convenience, Pheme provides a last-writer-wins policy by default. Other resolution mechanisms can be implemented and plugged into the protocol.

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

In particular, a DSV of a update U,

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

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

Conflict detection becomes as simple as comparing the write times and the DSVs for two updates. In order to detect whether two different updates U_1 and U_2 to the same object conflict, we carry out the following comparisons: If $U_1.ts$ is included in $U_2.dsv$, then U_1 causally

```
if received message is a subscription start message, SubStart then
   //set up subscription stream:
   stream.SS ⇐ subStart.SS
   stream.VV ⇐ subStart.VV
else if received message is an invalidation, I then
   //update log, timing state and per object state:
   store I in update log
   update stream.VV to include I.timeStamp.
   update currentVV to include I.timeStamp.
   obj.timeStamp ⇐ I.timeStamp
   obj.isValid ⇐ false
else if received message is a gap marker, GM then
   //update log, timing state and interest set state:
   store GM in update log
   update stream.VV to include GM.endVV.
   update currentVV to include GM.endVV.
   check for intersecting set
   IIS ⇐ stream.SS ∩ GM.targetSet
   if IIS ≠ ∅ then
      // detach IIS from the stream
      IIS.noGapVV ⇐ min(IIS.noGapVV, GM.startVV − 1)
      stream.SS ⇐ stream.SS \ IIS
      remove stream from IIS.attachedStreams
   end if
else if received message is a checkpoint, CP then
   //apply received meta-data to local structures
   for all IS in CP do
      update local IS.noGapVV to include CP.IS.noGapVV
   end for
   for all object metadata in CP do
      if CP.obj.metadata is newer than local.obj.metada then
         update local.obj.metadata to include CP.obj.metadata
      end if
   end for
else if received message is a catchup start message, CStart then
   //switch to catchup mode
   stream.pendingSS ⇐ Cstart.SS
   stream.pendingVV ⇐ Cstart.VV
   for all invalidation or gap markers recevied do
      process as above, except, update pendingVV instead of stream.VV
   end for
else if received messages is a catchup end message, CEnd then
   //switch to normal mode
   if stream.pendingVV equals or includes stream.VV then
      //attach stream.pendingSS to the stream.
      stream.SS ⇐ stream.SS ∪ stream.pendingSS
      add stream to consistencyModule.pendingSS.attachedStreams
   end if
end if
```

Fig. 3: Pseudocode for processing invalidation streams.

precedes U_2, by definition. Similarly, if $U_2.ts$ is included in $U_2.dsv$, then U_2 causally precedes U_1. Otherwise, U_1 and U_2 are marked as conflicts.

Deriving DSVs. It would be inefficient to transmit a DSV with each update and store a DSV with each object. Pheme therefore derives DSVs from the meta-data already maintained by the synchronization protocol. In order to do that, it ensures that a node is aware of all the previous updates to the object before it is updated. Any new update (a local write or a received invalidation) can only be applied if there is no gap in the object update information (i.e. the enclosing interest set is not gapped).

By definition *noGapVV* of an interest set covers all the causally preceding updates to the objects in the interest up to that time. Hence, for an object in the interest set, *noGapVV* includes all the causally preceding updates to that object. If the interest set is not gapped, then the *noGapVV* and so the DSV is equal to *currentVV*.

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

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

6 Flexible Commit Mechanisms

Commit policies are required when applications differentiate between tentative and committed writes [19] or when applications need to provide stronger consistency guarantees [5] [24]. Commit policies greatly differ from system to system. For example, Golding's algorithm [5], requires heartbeats to be sent to commit updates. However, it may not be able to commit updates when during periods of disconnection. Bayou [19] employs a primary commit protocol in which a single server is responsible for committing writes. Since the final total order is dependent on the commit order, Bayou's commit protocol requires repeated roll-back and reapplication of reordered updates if the commit order does not match the original write order.

The key characteristic of Pheme's commit mechanisms is that they do not require reordering of already received updates. In Pheme, the final order reflects the write order rather than the commit order. Pheme exposes a commit operation that commits a previous update by assigning *commit time* to the update. The commit time does not impose any order on the update; it simply is a record that the update was committed. For example, Node A receives two concurrent updates $u1$ and $u2$ to the same object and because of the last-writer-wins policy, orders $u1$ before $u2$. In Pheme, even if $u2$ is committed *before* $u1$, in the final order, $u2$ is still ordered *after* $u1$. Therefore, Node A does not need to reorder the updates. In Bayou, however, the final order reflects the commit order and so $u2$ would have to be reordered before $u1$.

In addition, Pheme transmits commit information via invalidation subscriptions, maintaining causal order among commits. Because of the casual order, a node can ensure that its view of committed data is consistent with that of the committing node by reading only committed updates. Say there are two updates to different objects, $u1$ and $u2$. Node A commits both updates but $u2$ is committed before $u1$. Node A then synchronizes with Node B. If there is no ordering guarantees among the commit information sent, there is a possibility that due to network disconnection, Node B will get the commit information for $u1$ but not for $u2$. In Node B's view, $u1$ was committed before $u2$ which is different from that of Node A. Maintaining consistent view of committed data is of utmost importance for implementing strong consistency semantics.

The commit mechanisms provide a building block over which various commit schemes and stronger consistency semantics can be implemented including primary-commit [19] and Golding's algorithm [5]. In fact, the mechanisms allow any node to commit an update. However, we expect that applications will restrict the set of nodes that can commit a write. We have used these mechanisms to guarantee serializability in a client-server environment. The implementation details and the proof can be found in [3].

Details. The commit operation takes in object ID, *objId*, and the time stamp, *targetTimeStamp* of the update to be committed. The operation is assigned a commit time, *commitTime*, and a commit invalidation is generated. Note that a commit invalidation is propagated along invalidations subscriptions and summarized into gap markers like any other invalidation.

The object store maintains the commit information for each object, including an *isCommitted* flag and a *commitTime*. When a commit invalidation *CI* is received, the local object *timeStamp* is compared with *CI.targetTimeStamp*. If they match, then the object is committed, i.e. *obj.isCommitted* flag is set to true and the *obj.commitTime* is set to *CI.commitTime*. When a checkpoint is generated, the commit information is included in the per-object meta-data.

6.1 Implementing A Primary Commit Scheme

Consider a primary commit scheme in a client-server system – clients write to objects and the server is responsible for committing writes. In Pheme, this scheme can be implemented as follows: Every client has subscriptions to and from the server. Whenever a client writes to an object, an invalidation is propagated to the server via an invalidation subscription. The server commits the updates as it receives the invalidations, and the commit invalidation is propagated back to the client via an invalidation subscription. An invalidation stream from the server to Node A will include commit invalidations of Node A's writes, invalidations of writes by other nodes followed by, after some lag, the commit invalidations for those writes.

Because all the writes are committed by the server, the commit time reflects the server's view of the data. If the client only reads committed objects, its view of the data will be consistent with that of the server. Even if a client receives an update directly from another client, it cannot access that update unless the server has received and committed the update.

7 Evaluation

In this section, we examine whether Pheme lives up to its promise of being a protocol that can meet the needs of a wide range of applications. We carry out our investigations by evaluating the three properties of the protocol as follows:

- *Synchronization:* We evaluate how well Pheme performs with traditional workloads with coarse-grained synchronization and with workloads with multiple fine-grained synchronization.
- *Conflict detection:* We evaluate overheads associated with conflict detection and compare it to existing schemes.
- *Commit mechanisms:* We evaluate the cost for supporting the flexible commit mechanisms.

We carry out our investigations on a Pheme prototype implemented with Java and BerkeleyDB. We demonstrate that like PRACTI, Pheme provides significant speedup and bandwidth savings when compared to traditional client-server and server-replication protocols. However, unlike PRACTI, Pheme provides up tens of times of bandwidth savings for workloads that dynamically establish fine-grained subscriptions.

7.1 Synchronization

Pheme falls under the class of protocols that support the "PR-AC-TI" properties. The PRACTI paper has demonstrated that by sending the right data on the right paths, this class of protocols can improve availability and achieve orders of magnitude more efficiency than

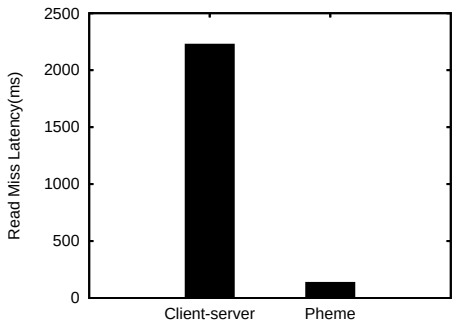

Fig. 4: Comparing read miss latency of client-server protocol with Pheme.

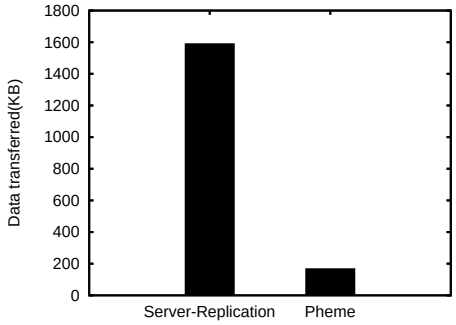

Fig. 5: Bandwidth required to synchronize 10 percent of a data set via server-replication and Pheme.

client-server and server-replication protocols for some key workloads. The efficiency stems from the fact that such protocols do not put any restrictions on how synchronization should take place. Nodes have the flexibility to retrieve updates over a fast connection from nearby peers instead of the server and to choose the data they want to synchronize instead of having to synchronize all objects.

Instead of repeating the same in-depth experiment as PRACTI, we validate that Pheme demonstrates the similar benefits and then evaluate the benefits of Pheme's efficiency by comparing against PRACTI.

Benefits of topology independence. Client-server or hierarchical protocols have the restriction that synchronization only occurs via specific nodes, for example a client can only synchronize with a server. Since Pheme supports topology independence, clients can retrieve updates from other peers based on availability and the cost of doing so. Significant benefits are achieved if the connection between clients is faster than the connection to the server or if the server is unavailable.

Figure 4 measures the time it takes to retrieve an object on a cache miss. For the client-server protocol, the object is retrieved from the server. However, with Pheme the object is retrieved from a nearby client. The client is connected to the server via a 1Mb/s 300ms RTT connection, and to other clients via a 100Mb/s 10ms RTT. As the figure illustrates, Pheme can achieve up to 15 times more efficiency.

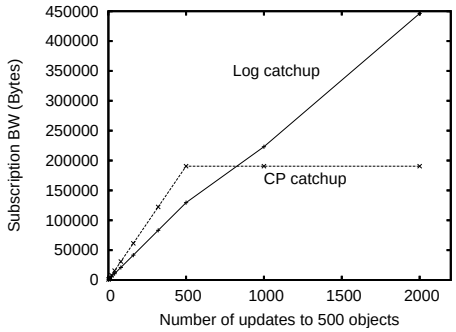

Fig. 6: Bandwidth to subscribe to varying number of updates to 500 objects sets for checkpoint and log synchronization.

	Coherence-only	Pheme
Bursty workload (10)	1	1.1
Worst Case	1	2

Fig. 7: Messages per interested update sent by a coherence-only system and Pheme.

Benefits of partial replication. In this experiment, we compare a Bayou-like server replication protocol with Pheme. Node A stores 500 objects of size 3KB, each of which have been updated. Node B is only interested in 10 percent of the data and synchronizes with Node A via the server-replication protocol and Pheme. Figure 5 demonstrates that Pheme achieves significant bandwidth efficiency when compare to the server-replication protocol because of its support for partial replication.

Log vs. checkpoint synchronization. Figure 6 compares the bandwidth cost for log and checkpoint synchronization. A set of 500 objects were updated uniformly and invalidation subscriptions are established separately for each object As the figure illustrates, the synchronization cost both options are proportional to the number of updates when each object is not updated more than once. Checkpoint synchronization does worse because the size of the meta-data sent in a checkpoint is slightly larger than a gap marker sent for log synchronization. However, when an object is updated multiple times, checkpoint catchup outperforms log synchronization.

Cost of flexible consistency. We first evaluate the cost Pheme pays to support flexible consistency. For systems that require weak consistency such as coherence, they simply send updates without gap markers. For systems that require strong consistency, they need to send gap markers to ensure casual semantics over which stronger guarantees can be implemented. In particular, we quantify the cost of sending gap markers in an invalidation stream. Figure 7 compares the number of messages per update between a coherence-only system and Pheme. In a coherence-only system, only updates to objects in the synchronization set are sent on the stream. On the other hand, Pheme also sends gap markers for updates outside the subscription set. For a bursty workload, say if 9 out of 10 updates occur to objects in the subscription set, a gap marker is only sent after nine invalidations. In the worst case workload, Pheme sends a gap marker after every invalidation. Thus, Pheme sends at most twice the number of messages when compared to a coherence only system. However, since gap markers are significantly smaller than actual bodies, the overhead remains within reasonable bounds.

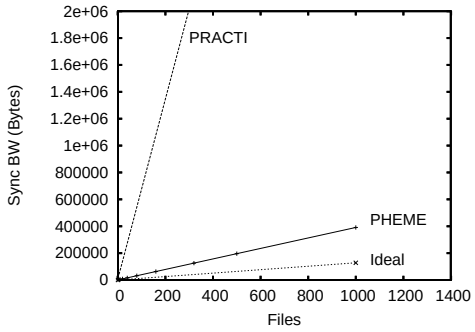

Fig. 8: Bandwidth to subscribe to varying number of single-object interest sets for Pheme and PRACTI.

Multiple subscriptions. Efficient support for multiple dynamic subscriptions is important because they are used to implement demand caching with per-object callbacks [11]. For example, each time a client caches a new object, it creates a new subscription for that object.

We compare the efficiency of establishing multiple dynamic subscriptions. A varying number of single-object invalidation subscriptions are established with PRACTI and Pheme. Figure 8 depicts the bandwidth costs for subscription establishment for the ideal case, PRACTI and Pheme. In the ideal case, only the object is sent for every subscription request. PRACTI establishes a separate invalidation stream for each request. Other than subscription start and end messages, gap markers are sent on each stream. Pheme multiplexes subscription requests on a single stream - only the catchup start, the object and catchup end messages are sent for each request. The major cost saving comes from the reduction of redundant gap markers received by a node. Since both PRACTI and Pheme are implemented in Java, the inefficiency of Java serialization does affect the bandwidth cost. However, it is not difficult to see that Pheme achieves comes much closer to ideal when compared to PRACTI.

Worst-case overheads. We evaluate the worst-case overheads for Pheme. For every object, in addition to the data, Pheme stores a write time stamp, a commit time stamp, a valid flag and a commit flag. Pheme also maintains a version vector, $noGapVV$, for every interest set in the consistency module. In the worst case, every interest set only covers a single object, hence the worst-case storage overhead is $O(N \times R)$ for N objects with R-element version vector per object.

Consider a subscription established between two nodes for a subscription set *SS* with s objects. Say, p updates occurred before the subscription was established and q updates occurred until the subscription is disconnected.

An invalidation subscription with log synchronization will send a start version vector, p invalidations during catchup and q in the connected phase and the number of gap markers depending on workload. Gap markers store partial version vectors. Hence if a gap maker summarizes K updates, it has at most $min(R,k)$ elements. In the worst case, on an invalidation subscription, there is a gap marker between every two updates, and hence the overhead is $O((p+q)\times R)$.

With checkpoint synchronization, the sender sends per-object meta-data for updated objects and *noGapVV*s for every interest set in *SS* during catchup. In the connected phase, q invalidation are sent. In the worst case, the checkpoint includes meta-data and a *noGapVV* for every object in *SS* and there is a gap marker after every invalidation in the connected phase. The worst-case overhead $O((s+q)\times R)$.

A body subscription, during catchup, includes the latest bodies of updated objects in *SS* and in the connected phase bodies of all updates to *SS*. Every body is sent with its timestamp. In the worst case, all object in the *SS* are sent. Hence, the overhead is $O(s+q)$.

7.2 Cost for conflict detection and commit

For conflict detection, Pheme utilizes the consistency information already maintained and hence exerts no extra overhead. However, for several conflict-detection schemes, the amount of book-keeping information increases with network disruptions. For Pheme, the book-keeping information remains the same because the number of interest sets a node maintains is not affected by disruptions. Hence, for k interest sets, the storage overhead is $O(N+k\times R)$. If invalidations subscriptions are disrupted, they simply re-start where they left off incurring extra version vector overhead due to the resending of subscription start time. Hence, in the worst case, for log synchronization, there is a version vector overhead per update sent and for checkpoint synchronization, there is a version vector overhead per object sent.

Given the amount of flexibility that Pheme supports, the conflict detection costs are reasonable, see Figure 9. In fact, the overheads of conflict detection is comparable to existing state-of-the-art approaches that do not provide such flexibility.

Despite the flexibility afforded by the commit mechanism, the overheads is minimal. For every commit, one commit invalidation is generated which contains two time stamps. Therefore, the overhead is O(1) per commit. For N objects, a commit time stamp is stored per object, so the storage overhead of $O(1)$ per object.

8 Related Work

What sets Pheme apart from other synchronization protocols is that Pheme is a peer-to-peer protocol that supports partial replication, is able to provide consistency guarantees, and provides flexible synchronization options.

Client-server-based [11] and hierarchy-based [4] protocols have limited use in mobile environments because they do not support arbitrary synchronization topologies.

Existing peer-to-peer synchronization protocols support arbitrary synchronization topologies but they fall short of providing other requirements. For example, Bayou [19], one of the most influential peer-to-peer protocols in the literature, often cannot be applied in such environments because it does not support partial replication of data. Peer-to-peer protocols that support partial replication such as WinFS [17], Rumor [8], Ficus[7], Pangaea [21], give up on cross-object consistency and only support single object coherence. Some of them support only state-based or log-based synchronization, making them less flexible switch to the scheme with better tradeoffs for different scenarios. Some systems, targeting personal environments, employ peer-to-peer communication as a conduit to a repository, such as Footloose [15] and OmniStore [10], or to improve performance and availability [18], rather than for data synchronization.

Segank [22], a mobile storage system, supports partial replication with peer-to-peer synchronization and consistency guarantees. However, it requires users to always carry with them a device that holds the latest meta-data. It employs a multi-cast like solution to request and locate data, which could lead to high network costs.

PRACTI [1] is another peer-to-peer synchronization protocol that resembles Pheme. PRACTI uses *imprecise invalidations* to propagate consistency information in same way as Pheme uses gap markers. However, it uses previous time stamps for conflict detection which do not work well for checkpoint-synchronization. Pheme provides better efficiency of dynamic synchronization request, a conflict detection scheme that supports synchronization flexibility, and flexible commit policies.

There are three main approaches for conflict detection: *previous stamps* [1, 6], *hash histories* [9], and *version vectors* [8] [11] [15] [16] [20] [25][21]. Both *previous stamps* and *hash histories* impose per-update storage overhead and might have false negatives under certain scenarios. *Version vectors* can accurately detect conflicts but impose a one vector per object overhead which is prohibitive when the number of replicas is large.

Predecessor vectors with exceptions (PVE) [14] and *vector sets* [13] are variations of the *version vectors* approach employed by WinFS [17] to reduce the total number of version vectors maintained and communicated. PVEs can reach an unbounded size if synchronizations are frequently disrupted making them unsuitable for en-

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

Fig. 9: Storage and network overheads under network disruptions for a node with N objects, p recent updates and R-element version vectors. For Pheme, the node stores k interest sets.

vironments with intermittent connections. Vector sets maintain predecessor vectors for subsets of data and in the worst case, have overheads equivalent to a simple version vector scheme. Pheme's dependency summary vectors (DSV) are actually equivalent to predecessor vectors. However, due to the properties of the synchronization protocol, instead of storing DSVs, explicitly in a data structure, Pheme derives them from the consistency meta-data already stored. In addition, the meta-data stored and sent during synchronization does not increase with network disruptions. Section 7 evaluates the overheads of Pheme and demonstrates that Pheme's overheads are equivalent to other state-of-the-art schemes.

9 Conclusion

Pheme is a peer-to-peer data synchronization protocol that can be used to construct new distributed file systems targeting personal and mobile environments.

Pheme's suitability for this environment stems form four key properties. First, its synchronization mechanism is able to support synchronization of any subsets of data between any peers, with support for both log-based and state-based exchange. Second, its support for fine-grained synchronization yields superior performance in scenarios with always-connected devices. Third, its conflict detection scheme is able to support synchronization flexibility with minimal overhead and performs as well as current schemes. Fourth, its flexible commit mechanisms eliminate reordering and rollback of writes and enable applications to implement their own commit schemes.

We conclude that Pheme realizes our vision of a protocol that can more than adequately meet the needs of a wide range of applications targeting mobile environments.

References

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

[2] N. Belaramani, M. Dahlin, A. Nayate, and J. Zheng. PADS: A Policy Architecture for building Distributed Storage systems. In *Proc NSDI*, 2009.

[3] N. Belaramani, J. Zheng, A. Nayate, R. Soule, M. dahlin, and R. Grimm. PADRE: A Policy Architecture for building Data REplication systems. Technical Report TR-08-25, University of Texas at Austin, May 2008.

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

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

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

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

[8] R. Guy, P. Reiher, D. Ratner, M. Gunter, and W. Ma. Rumor: Mobile data access through optimistic peer-to-peer replication. In *In Workshop on Mobile Data Access*, pages 254–265, 1998.

[9] B. Kang, R. Wilensky, and J. Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *ICDCS*, 2003.

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

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

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

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

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

[15] 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.

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

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

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

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

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

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

[22] 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.

[23] 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*, December 1995.

[24] R. Thomas. A Solution to the Concurrency Control Problem for Multiple-Copy Databases. In *Proceedings of the Sixteenth IEEE Computer Society International Conference*, 1978.

[25] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *SOSP*, pages 49–69, October 1983.

[26] J. Zheng. *URA: A Universal Data Replication Architecture*. PhD thesis, The University of Texas at Austin, August 2008.