Feres: Flexible and Efficient Replica Synchronization For Diverse Environments

Jiandan Zheng[†] Nalini Belaramani[*] Mike Dahlin[*]
[*]University of Texas at Austin [†]Amazon Inc.

Abstract:

This paper presents Feres, a peer-to-peer data synchronization protocol that can be used to construct new, flexible distributed file systems that share data across collections of devices with limited, varying, or intermittent connectivity. The amount of flexibility provided by Feres is not matched by existing protocols. In particular, Feres allows any device to synchronize any subset of data with any other node, provides options to chose either data or meta-data and log-based or state-based synchronization, detects conflicts and supports application specific commit policies without require rollback. In addition, the overheads associated with Feres are reasonable.

Because Feres is flexible, it can send the right data via the right network paths and dramatically outperform less flexible traditional client-server or server-replication protocols – we observe up to 15 times speedup for one scenario. At the same time, Feres is more efficient than existing similarly flexible protocols like PRACTI – we observe order 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. The key challenge for file systems targeting personal and mobile environments is the diversity the environment presents in terms of device mobility, connectivity, storage capacities and application needs in terms of consistency, performance and availability.

Our vision is to construct a single replication protocol that meets these needs. In contrast, past approaches [Guy et al., 1998] [Karypidis and Lalis, 2006] [Kistler and Satyanarayanan, 1992] [Mazzola et al., 2003] have built different protocols that embed different trade-offs in their mechanisms making them suitable for only a subset of applications. The benefit of having a single flexible protocol is threefold: First, a flexible protocol simplifies development since systems can be quickly constructed for a target environment by picking suitable protocol options. Second, systems can take advantage of the flexibility to adapt to changing workloads and network environment. Third, building a replication system over flexible mechanisms makes it possible to implement different synchronization policies for different subsets of data or at different times.

The key challenge is that such a protocol must have sufficient *flexibility* and *efficiency*. In particular, the protocol must support the following features:

- *Partial replication:* Every device may store different subsets of data, either due to storage limitations or user preference. The protocol must allow any device to store any subsets of data.
- *Wide range of consistency semantics:* 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 that a device be able to 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, log-based and state-based synchronization, and dynamic synchronization establishment.
- *Incremental progress:* Network interruptions may be common. The protocol must ensure 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 the conflicts so that applications can invoke appropriate resolution mechanisms.
- *Low overheads:* Network and resource limited devices are common in this environment. The flexible protocol should be competitive with non-flexible, hand-crafted protocols.

No existing protocol satisfies the set of features listed above. PRACTI [Belaramani et al., 2006] shares similar goals but is not sufficiently flexible: it does not support conflict detection for state-based synchronization; it cannot efficiently support protocols that require establishing dynamic fine-grained synchronization; and it does not support certain consistency semantics well because it does not differentiate between tentative and committed writes.

This paper presents Feres, a new data synchronization protocol that provides system designers with this wide range

of synchronization options at reasonable costs. In order to do this, Feres draws on some ideas from past protocols and then introduces three key new ideas. As in some past protocols, Feres uses peer-to-peer synchronization [Petersen et al., 1997, Sobti et al., 2004, Guy et al., 1990, 1998, Saito et al., 2002] via log exchange [Petersen et al., 1997] and state exchange [Novik et al., 2006], separation of invalidations and bodies [Belaramani et al., 2006, Kistler and Satyanarayanan, 1992], causal propagation of updates [Petersen et al., 1997, Belaramani et al., 2006] and summarization of unwanted meta-data [Belaramani et al., 2006] But, to meet the needs of this environment, Feres introduces the following new techniques:

- *Fine-grained multiplexing of synchronization requests.* If two nodes have a synchronization stream established, any new synchronization request is multiplexed on the same stream. Previous peer-to-peer protocols [Belaramani et al., 2006, Petersen et al., 1997] establish a new synchronization stream for each synchronization request between two nodes leading to inefficiencies. Multiplexing synchronization requests on a single stream enables efficient implementation of common replication techniques such as demand caching and callbacks.

- *Dependency summary vector scheme for conflict detection:* Some previous protocols only detect conflicts for either state-based [Malkhi et al., 2007, Novik et al., 2006] or log-based synchronization [Belaramani et al., 2006, Petersen et al., 1997]. Feres introduces a dependency summary vector (DSV) scheme that detects conflicts for both state-based and log-based synchronization. In addition, it uses consistency meta-data already maintained for synchronization and as a result does not incur any extra bandwidth or storage overhead despite network disruptions.

- *Flexible commit mechanism:* Previous protocols either do not differentiate between tentative and committed writes [Belaramani et al., 2006, Novik et al., 2006], or require roll-back and reordering of writes [Petersen et al., 1997]. Feres exposes a commit operation that assigns a final commit order to writes that is independent of the original write-order making it efficient because it does not require roll-back or reprocessing of already received writes. In addition, commit information is propagated together with update information in causal order making it easier to reason about the implementation of commit policies. As a result, it is easy for applications that need strong consistency (for e.g., linearizability, serializability) to provide it over Feres mechanisms.

This paper presents details of Feres and evaluates it under different synchronization scenarios. Details of how to implement specific replication policies with the Feres mechanisms can be found elsewhere [Belaramani et al., 2008]. We demonstrate that Feres possesses the flexibility and efficiency required for mobile environments. Feres is *flexible* – it is able to support all the above features. Feres is *efficient* – its overheads are proportional to the data required by a node. When compared to traditional client-server and server replication protocols, synchronization with Feres provides up to 15 times speedup for a significant range of workloads. Feres also provides several orders of magnitude of bandwidth savings when compared to PRACTI for common workloads.

2 System Model

In this section, we briefly describe the system model assumed by Feres.

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 object IDs. For example, the interest set "/a/*:/b" includes object IDs with the prefix "/a/" and also includes the object ID "/b".

Feres heavily relies on Lamport's clocks [Lamport, 1978] 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 3.2.

3 Synchronization

Synchronization between two nodes is carried out via unidirectional streams that allow Feres to meet all of its design requirements. We present an overview of the protocol followed by details of the implementation.

3.1 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

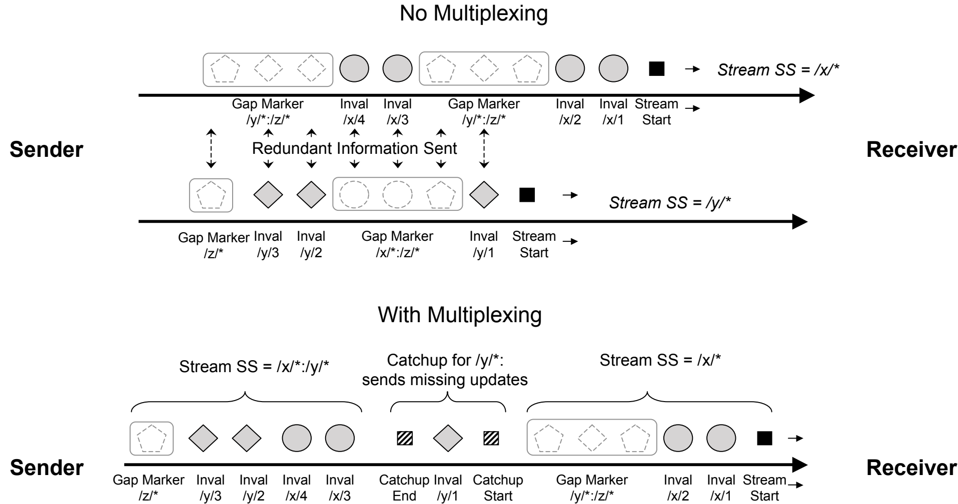

Fig. 1: Diagram comparing the messages sent on invalidation streams without and with multiplexing.

flexibility to enforce whatever level of consistency it needs. Synchronization in Feres tries to keep that information to a minimum.

Synchronization streams. Suppose all the objects stored in Node A lie in the interest set, $A.IS$. Node A knows about all updates to $A.IS$ up to its current time, $A.currentVV$. 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 Node B has to objects in $A.IS$, it 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* [Belaramani et al., 2006] 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 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 to all nodes that allows Feres to simultaneously support partial replication, topology independence, and still support a broad range of consistency semantics.

An update 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. This inefficiency prevents existing flexible protocols, like PRACTI, from meeting our needs. For example, in PRACTI, to support demand caching with per-object callbacks [Kistler and Satyanarayanan, 1992] [Nightingale and Flinn, 2004], each time a client caches a new object, it creates a new subscription by establishing a new independent stream from the server.

Feres 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. Feres 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* and *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* to *stream.VV*.

Log and checkpoint synchronizations have different trade-offs. 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. Feres separates meta-data and data synchronization by having separate 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 the invalidation streams is sufficient to guarantee consistency.

The separation of invalidation and body subscriptions 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.

3.2 Protocol Details

The protocol must allow any peers to exchange any subsets of data via log or incremental checkpoint. This is done by establishing multiple *subscriptions* between nodes. In this section, we explain the key ideas including subscriptions, state maintained in each node, and the processing carried out to update node state when messages are received.

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.

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. 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 meta-data 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 3.2 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, say 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 stream, Node A maintains a logical time that includes the last record and all the causally preceding records 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 [Zheng, 2008]. 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.

We introduce the concept of "attaching" an interest set to a stream, so as to eliminate the need for updating $IS.noGapVV$ every time an invalidation or gap marker is received. 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. Sender side processing of subscriptions is simple. 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 $stream.startVV$.

4 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 in conflict 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 [Kistler and Satyanarayanan, 1992] [Terry et al., 1995]

To meet these needs while still supporting the flexible and efficient synchronization described in the previous section, Feres introduces a dependency summary vector (DSV) scheme. DSVs are similar to WinFS's predecessor vectors [Malkhi and Terry, 2005] but they can detect conflicts for both log-based and state-based synchronizations and support the consistency guarantees of Feres's synchronization protocol. For efficiency, instead of storing DSVs explicitly in a new data structure, Feres derives them from the consistency meta-data already stored. Despite network disruptions, the metadata stored and sent during synchronization does not increase, Section 6 evaluates the overheads of Feres and demonstrates that Feres performs has overheads equivalent to other state-of-the-art schemes.

Conflict detection is carried out as follows: the derived DSV of a received update is compared with the derived DSV of the local version. 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. Feres provides mechanisms for conflict detection, but it leaves conflict resolution to application specific policies. For convenience, Feres

```
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.

provides a last-writer-wins policy by default. Other resolution mechanisms can be implemented and plugged into the protocol.

4.1 Dependency summary vectors

A dependency summary vector (DSV) is a vector associated with an update that summarizes all the causally preceding updates to the object being updated. It is similar to predecessor vectors [Malkhi and Terry, 2005]. 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 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.

4.2 Deriving DSV

It would be inefficient to transmit a DSV with each update and store a DSV with each object. Feres 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).

For a locally stored object, the DSV is equal to the *noGapVV* of its enclosing interest set. 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, Feres takes advantage of the causal property of the stream: For a received invalidation, the stream has sent all the causally preceding updates, and any newer updates will not come before the current received invalidation. *streamVV* includes all the current and all causally preceding updates. Hence, 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 *noGapVV* received.

Feres 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.

5 Commit Mechanism

Commit policies are often employed when applications differentiate between tentative and committed writes [Petersen et al., 1997] or when applications need to provide stronger consistency guarantees [Golding, 1992] [Thomas, 1978]. Commit policies greatly differ from system to system. For example, Bayou [Petersen et al., 1997] employs a primary commit protocol in which a single server is responsible for committing writes. Since the commit order corresponds to the write order, Bayou's commit protocol requires repeated roll-back and reapplication of reordered updates if the writes arrive out of order. Golding's algorithm [Golding, 1992] requires heartbeats to be sent to commit updates which availability when during periods of disconnection.

Feres provides mechanisms that can be used to implement various commit protocols including primary-commit [Petersen et al., 1997], Golding's algorithm [Golding, 1992] and quorum-based commit [Thomas, 1978]. In particular, Feres exposes a commit operation that assigns a *commit time* to a previous update. The mechanisms provide the flexibility to allow any node to commit an update. However, we expect that applications will restrict the nodes that can commit a write.

In addition, commit information is passed along invalidation subscriptions. Therefore, the casual order among commits is maintained. By reading only committed updates, a node can ensure that its view of the data is consistent with that of the committing node without the need to roll-back and re-order writes.

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 commit invalidations are 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.

Implementing 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 Feres 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 the invalidation subscription. The server commits the writes 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.

6 Evaluation

In this section, we examine the costs and benefits of Feres's flexibility. Ideally, the flexibility should come at minimal costs but yield significant efficiency benefits. We carry out our investigations by evaluating the three properties of the protocol as follows:

- *Synchronization:* We evaluate whether the flexibility of *FE synchronization* yields efficiency benefits it promises, and what cost is paid to support this flexibility.
- *Conflict detection:* Given that Feres has the flexibility to detect conflicts in any synchronization scenario, we evaluate overheads associated with conflict detection.
- *Commit mechanisms:* We evaluate the cost for supporting the flexible commit mechanisms.

We carry out our investigations on a Feres prototype implemented with Java and BerkelyDB. We demonstrate that the costs for flexibility are minimal but the efficiency benefits are significant in many workloads.

6.1 Flexible Synchronization

Feres 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 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 Feres demonstrates the similar benefits and then evaluate the benefits of Feres'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

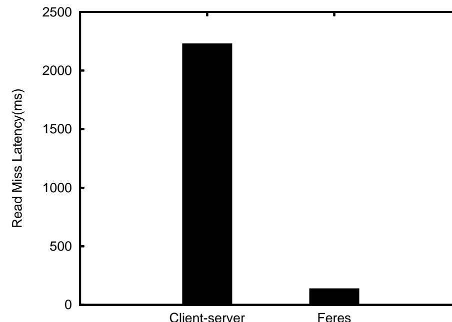

Fig. 4: Comparing read miss latency of client-server protocol with Feres.

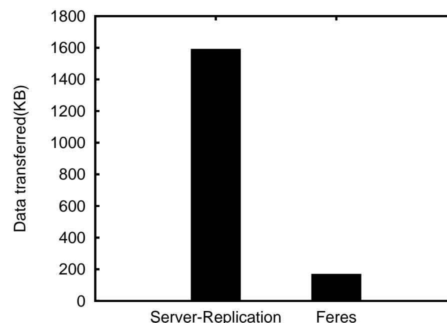

Fig. 5: Bandwidth required to synchronize 10 percent of a data set via server-replication and Feres.

synchronize with a server. Since Feres 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 Feres 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, Feres can achieve up to 15 times more efficiency.

Benefits of partial replication. In this experiment, we compare a Bayou-like server replication protocol with Feres. 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 Feres. Figure 5 demonstrates that Feres 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

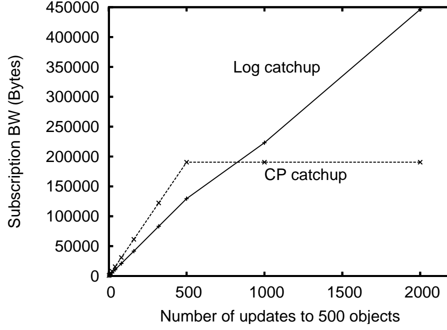

Fig. 6: Bandwidth to subscribe to varying number of updates to 500 objects sets for checkpoint and log synchronization.

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

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

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.

6.2 Efficient Synchronization

In this section, we evaluate the efficiency of Feres by quantifing the overhead associated with supporting consistency, comparing it to PRACTI, and evaluating the worst case overheads.

Cost of consistency. We first evaluate the cost Feres pays to support flexible consistency. 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 Feres. In a coherence-only system, only updates to objects in the synchronization set are sent on the stream. On the other hand, Feres 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, Feres sends a gap marker after every invalidation. Thus, Feres 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.

Feres vs. PRACTI. We compare Feres and PRACTI by evaluating the efficiency of establishing multiple dynamic subscription request. PRACTI establishes a separate invalidation stream for each request, whereas Feres multiplexes subscription requests on a single stream. The major cost saving comes from the reduction of redundant information received by a node.

A varying number of single-object invalidation subscriptions are established with PRACTI and Feresand the bandwidth cost for subscription establishment was measured. Figure 8 depicts the differences in bandwidth. Since both PRACTI and Feresare implemented in Java, the inefficiency of Java serialization does affect the bandwidth cost. However, it is not difficult to see that Feres achieves magnitudes of savings when compared to PRACTI.

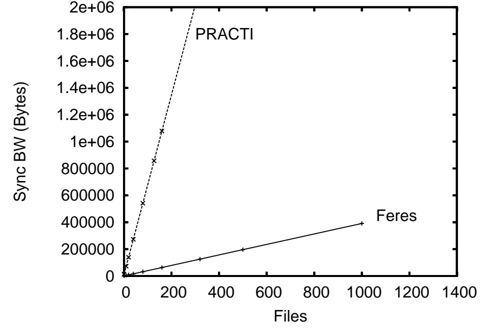

Fig. 8: Bandwidth to subscribe to varying number of single-object interest sets for Feres and PRACTI.

Worst-case overheads. We evaluate the worst-case overheads for Feres. For every object, in addition to the data, Feres stores a write time stamp, a commit time stamp, a valid flag and a commit flag. Feres 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 $O(s+q)$.

	Version vectors	PVE	Vector sets	Feres log sync	Feres 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 Feres, the node stores k interest sets.

6.3 Cost for conflict detection and commit

For conflict detection, Feres 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 Feres, 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 Feres supports, the conflict detection costs are reasonable, see Figure 6.3. 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.

7 Related Work

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

Client-server-based [Kistler and Satyanarayanan, 1992] and hierarchy-based [Demmer et al., 2008] 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 [Petersen et al., 1997], 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 [Novik et al., 2006], Rumor [Guy et al., 1998], Ficus[Guy et al., 1990], Pangaea [Saito et al., 2002], 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 [Mazzola et al., 2003] and OmniStore [Karypidis and Lalis, 2006], or to improve performance and availability [N.Tolia et al., 2004], rather than for data synchronization.

Segank [Sobti et al., 2004], 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 metadata. It employs a multi-cast like solution to request and locate data, which could lead to high network costs.

PRACTI [Belaramani et al., 2006] is another peer-to-peer synchronization protocol that resembles Feres. PRACTI uses *imprecise invalidations* to propagate consistency information in same way as Feres uses gap markers. However, it uses previous time stamps for conflict detection which do not work well for checkpoint-synchronization. Feres 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* [Gray et al., 1996, Belaramani et al., 2006], *hash histories* [Kang et al., 2003], and *version vectors* [Guy et al., 1998] [Kistler and Satyanarayanan, 1992] [Mazzola et al., 2003] [Nightingale and Flinn, 2004] [Reiher et al., 1994] [Walker et al., 1983][Saito et al., 2002]. 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) [Malkhi and Terry, 2005] and *vector sets* [Malkhi et al., 2007] are variations of the *version vectors* approach employed by WinFS [Novik et al., 2006] to reduce the total number of version vectors maintained and communicated. PVEs can reach an unbounded size if synchronizations are frequently disrupted making them unsuitable for environments with intermittent connections. Vector sets maintain predecessor vectors for subsets of data and in the worst case, have overheads equivalent to a simple version vector scheme. Feres's dependency summary vector scheme (DSV) are similar predecessor vectors. However, Feres can support conflict detection with more flexible synchronization policies.

8 Conclusion

Feres is a flexible peer-to-peer data synchronization protocol that can be used to construct new distributed file systems.

Feres's flexibility and efficiency stems from three 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 conflict detection scheme is able to support synchronization flexibility with minimal overhead and the performs as well as the current schemes. Third, its flexible commit mechanisms eliminate reordering and rollback of writes and enable applications to implement their own commit schemes.

Because of these properties, Feres can be used to build replication engines in a wide range of environments including more demanding environments with mobile devices and intermittent connections.

References

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

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, U. of Texas at Austin, May 2008.

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

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

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

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. URL `ftp://ftp.cs.ucla.edu/pub/ficus/usenix_summer_90.ps.gz`.

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.

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

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

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

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

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

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

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.

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

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.

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

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

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

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

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.

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.

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

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

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