Cimbiosys: A platform for content-based partial replication

Venugopalan Ramasubramanian[1], Thomas L. Rodeheffer[1], Douglas B. Terry[1],
Meg Walraed-Sullivan[2], Ted Wobber[1], Catherine C. Marshall[1], Amin Vahdat[2]

[1]*Microsoft Research, Silicon Valley*
[2]*University of California, San Diego*

Abstract

People increasingly use multiple devices and Internet services to manage and share information. Since portable devices have limited resources for storage and bandwidth, it is essential to take advantage of proximity and selected replication of content. To this end we present *Cimbiosys*, a replication platform that permits each device to define its own content-based filtering criteria and to share updates with any other device.

Cimbiosys ensures two properties not achieved by previous systems. First, every device stores exactly those items whose latest version meets arbitrary filter criteria that are independent of any hierarchical namespace. Second, every device represents its metadata in a compact form, with state proportional to the number of devices rather than the number of items. Such compact representation enables low synchronization overhead, which permits frequent synchronization even for bandwidth-limited devices.

We have implemented Cimbiosys in C# and Mace. We evaluated the performance of the CIM Sync protocol in both simulation and using the Mace implementation.

1 Introduction

Alice keeps her favorite recent photos on her iPhone. When she runs into Bob at the supermarket, all of the photos of Bob's daughter at the last rehearsal of Our Town *are synchronized with Bob's cell phone, without involving any of the other private photos stored on Alice's device.*

People who use mobile devices have a growing expectation that they can take a significant measure of their personal information with them wherever they go and share selected portions of it with ease. In spite of this expectation, people report difficulties managing information across their own devices so that they have the desired portions of different collections at hand when they need them [3].

Delivering information that is relevant to different people—or is appropriate for different devices—requires system support for a richer notion of data synchronization, one that incorporates personalized content filtering. In many social and work settings, communication between devices may be ad-hoc, taking advantage of proximity and the availability of particular content, rather than relying on established communication patterns or depending on centrally managed storage.

In this paper, we present Cimbiosys, an application platform that supports content-based partial replication and synchronization with arbitrary peers. We designed Cimbiosys with the expectation that it will support applications and uses such as home media management [17] and shared calendars (especially those maintained outside the normal workplace environment, where general public information is mixed with more private, local scheduling) [9].

The main contribution of this work is in demonstrating how the platform can provide two important system properties:

- *Eventual filter consistency*: Over time, each device will receive all items of interest to it (i.e. all items that meet its content-based filtering criteria), and it will discard all items that have been deleted or that no longer meet these filtering criteria.
- *Eventual knowledge singularity*: The metadata replicas exchange when they synchronize that enables them to detect overlapping interests and identify missing versions may vary in size over time, but eventually, it converges to a size that is proportional to the number of replicas in the system rather than the number of stored items, even for partial replicas.

Eventual consistency has long been demanded by applications and provided in replicated systems, but it is more challenging to ensure in a system that permits ad-hoc synchronization between partial replicas. Not only may individual items match different filtering criteria as the items are updated, but a device may vary its criteria over time, causing items with stable contents to enter and leave the device's interest set. Eventual knowledge singularity, on the other hand, is a new property we have defined to convey the importance of compact metadata in making efficient use of bandwidth and system resources. In particular, this property allows Cimbiosys to use brief intervals of connectivity with peer devices effectively. By contrast, other known techniques that maintain per-item version vectors, such as in Ficus [5], or rely on operation logs, such as in PRACTI [2], make less effective use of a relatively slow or intermittent connection because the data exchanged during synchronization is roughly proportional to the collection size or dependent on the update rate; this limitation becomes important as collection sizes grow into the 10s of thousands of items, especially if the items are updated frequently.

After the next section motivates this research using scenarios, Section 3 discusses related work on replication protocols. Section 4 presents the Cimbiosys platform, including the system model, software components, and the implementation details. Section 5 provides the details of the replication protocol (CIM Sync), including a description of our knowledge compaction method. Section 6 explains why the synchronization topologies used in Cimbiosys embed a hierarchically-filtered tree topology to ensure that changes propagate appropriately and to guarantee eventual knowledge singularity. Following this discussion, Section 7 presents the results of our evaluation of CIM Sync. The conclusion summarizes the contributions of the Cimbiosys platform and our future work.

2 Motivation

Collaboration within loosely-organized communities is a central motivation for our work. In these organic-growth scenarios, circumstances often demand information sharing along an arbitrary topology, without relying on Internet connectivity or contact with a centralized store. For example, in the wake of Hurricane Katrina, disaster workers needed to quickly set up ad hoc networks where communication and control were distributed and egalitarian [4]. Because not everyone in the collaborative setting wants or needs access to the entire collection—bandwidth, storage, and human attention may be at a premium—filtering enables information to spread according to interests and requirements. A related motivation is peoples' need to easily replicate content among the multiple devices that they use.

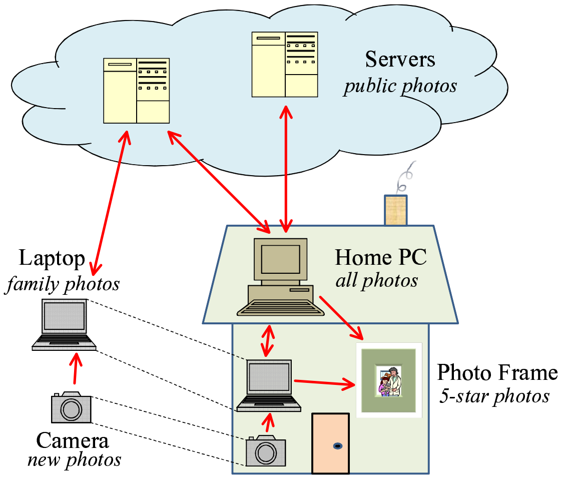

Figure 1: Photo sharing

We illustrate the advantages of our approach using two scenarios that combine information management across devices with informal sharing based on common interests. The first scenario tracks Alice's photo sharing (see Figure 1) and a second describes how Bob updates and shares the video content that he watches during his commute.

Alice is traveling alone in Thailand, photoblogging as she goes. Every night, the day's photos are copied to her laptop. When she reaches a town with an Internet café, she uploads photosets to her Flickr account to share with her friends and family, partly to narrate her trip, and partly to say, "I'm still alive!" When she returns from her trip, Alice is anxious to keep her new Thailand photo collection safe. She adds them to the master collection on her PC, which has grown quite large over the years. For the next few weeks, Alice works on her Thailand collection, rotating some photos, cropping others, and touching up the lighting on a few. Alice tags the ones she likes with a few descriptive words and rates them with one to five stars. Five-star photos are her favorites, the ones that will show up—via a direct WiFi connection—in her photo frame; they're also uploaded to a public travel photoset on Flickr, onto a photojournalism web site (where she hopes to win a prize), and onto her Facebook account. Comments she receives on any of these public sites are aggregated, so that she can look at them when she's browsing the collection on her laptop or on the household PC. A copy of all of her tagged photos, the ones Alice likes well enough to put in the extra effort, are stored with her family photos on her laptop, so she'll have them with her when she travels again.

During his daily commute on Caltrain, Bob entertains himself by watching videos on his iPod. Each night he syncs his player with the YouTube channels he subscribes to as well as with *The Daily Show* and *Colbert Report* video feeds so he'll have fresh content to watch. Yesterday while Bob was waiting for the train, he saw a electronic billboard advertising a new TV show. His iPod, which was set up to grab commercial content from sources he trusts, copied a trailer for the show over a local Bluetooth connection. Instead of watching his normal videos, Bob found himself watching the trailer on the train. A fellow commuter, who had caught a glimpse of it over his shoulder asked him for a copy for her own iPod, as did a third man, who was reading the *Times* on his laptop. With a flick of his finger, Bob shared the promo with his fellow commuters. Next time he had Internet access, Bob subscribed to the promising new show. After he'd watched the pilot on a subsequent commute, he deleted it and, when he got home in the evening, his delete eventually propagated to his home media center.

These scenarios reflect common usage patterns and reveal an implicit set of requirements for a modern storage platform:

- Interdevice communication may be ad-hoc, taking advantage of device proximity and the availability of particular content.
- These ad hoc networks are fluid, and may involve a changing set of people and devices.
- Not all device-based stores have complete collections; furthermore, the most complete version of a collection may not be in the cloud.
- Updates may originate from multiple sites and be contributed by different people.

The platform described in the remainder of the paper has been designed with these requirements—and the scenarios driving them—in mind.

3 Related Work

While we are not aware of other weakly-consistent replicated systems that combine content-based filtering with peer-to-peer communication, other systems have demonstrated several of these aspects.

Disconnected operation is supported by Coda [7], in which clients prepare by taking a hoard of files before disconnection and then reconcile updates with the server when reconnected. BlueFS [10] is similar but emphasizes energy efficiency when dealing with small, mobile devices. As opposed to Cimbiosys, neither Coda nor BlueFS permit clients to share updates with each other. EnsemBlue [13] extends BlueFS by allowing disconnected clients to organize into a temporary ensemble headed by a client acting in place of the server. EnsemBlue reflects a compromise between a full peer-to-peer architecture and a server-centric architecture.

Peer-to-peer file replication systems such as Ficus [5] support partitioned operation and permit updates to be shared between any two replicas. Subsequent work [16] extends Ficus to support selective, folder-based replication. Roam [15] is a further extension to the environment of mobile devices, in which bandwidth limitations may make it infeasible for all pairs of devices to communicate regularly. Ficus and Roam store a version vector per item. When synchronizing two replicas, they suffer the overhead of exchanging version vectors for every item in order to determine which items are newer.

Bayou [14] and PRACTI [2] reduce communication overhead by maintaining a single version vector per replica and utilizing a log-exchange protocol. Because of the use of logs, their storage use grows proportional to the update rate. Bayou supports full replicas only. PRACTI supports partial replicas through the notion of an interest set, which reflects the file system's folder structure. Although all invalidations must be logged at all replicas, including partial replicas, imprecise invalidations can reduce the logging overhead for partial replicas. Cimbiosys, on the other hand, does not require partial replicas to maintain any information about items that are not of interest. Cimbiosys also recognizes that users often wish to specify arbitrary filtering criteria that do not necessarily match entries in some filesystem namespace.

WinFS [11], like Bayou, maintains a single version vector per replica that is transmitted on every synchronization, but uses state-exchange rather than log-exchange. Cimbiosys shares these characteristics with WinFS. WinFS supports folder-based partial replication. Cimbiosys extends the WinFS design to add support for content-based filtering while ensuring eventual knowledge singularity.

4 Cimbiosys Platform

4.1 System model

Cimbiosys is a platform developed to support a variety of applications that manage data on mobile devices, personal computers, and cloud-based services. It provides a distributed, peer-to-peer architecture in which each participating node, hereafter simply called a *device*, stores full or partial copies of one or more data collections. A *collection*, for instance, might be an individual's digital photo album, a family's calendar, a compilation of videos from different people, or a company's customer relationship database. Each collection is managed separately and consists of a set of items that are not shared with other collections.

An *item* is an XML object plus an optional associated file. For example, a photo item stores its JPEG data in a conventional file and the associated XML object holds descriptive information, such as when the photo was taken, its resolution, a quality rating, and human-supplied keywords.

A *replica* is a copy of some or all of the items in a given collection. The set of items included in device's replica is specified by a *filter*, which is a selection predicate over the items' XML contents. For example, a filter might select e-mail messages from a particular individual, files tagged with certain keywords, or photos with a 5-star rating. The default "*" filter indicates that the device is interested in all items, and hence stores a full replica of the collection.

Each device holding a replica is allowed to read its locally stored items and update those items, as long as such updates are in accordance with the collection's *access control policy*. Updated items are then sent to other replicas via a device-to-device *synchronization protocol*. Devices generally have regular partners with which they periodically synchronize their replicas, but may also synchronize with any replica that they encounter. A device can join the system simply by creating a new (empty) replica of some collection and then synchronizing with some existing replica(s).

At any point in time, a replica may hold older versions of items that have been updated elsewhere or may not have learned yet of recently created or deleted items. The CIM Sync protocol guarantees *eventual filter consistency*, meaning that a replica eventually receives all versions of items that match its filter and have not been overwritten by later versions, and also that the replica eventually discards items that are updated in such a way that their contents no longer match the replica's filter. It does not provide other guarantees such as causal consistency or multi-item coherence since these are less meaningful for partial replicas.

4.2 Software components

Each device in Cimbiosys runs the set of software modules depicted in Figure 2. The *Item Store* manages the items for local replicas of one or more collections. The file portion of each item is stored in a special directory in the device's local file system. XML objects are stored in a SQL Server (Compact Edition) database where they can be queried and updated transactionally.

The *Communication* module is responsible for transmitting data to other devices using available networks, such as the Ethernet, WiFi, cellular, or Bluetooth. It also encapsulates the transport protocol used by the Sync module. Devices are free to use a variety of transport protocols, including SOAP-based RPC, HTTP, and Microsoft's simple sharing extensions (SSE) to RSS. Of course, two devices must agree on the network and transport protocol that they use during synchronization.

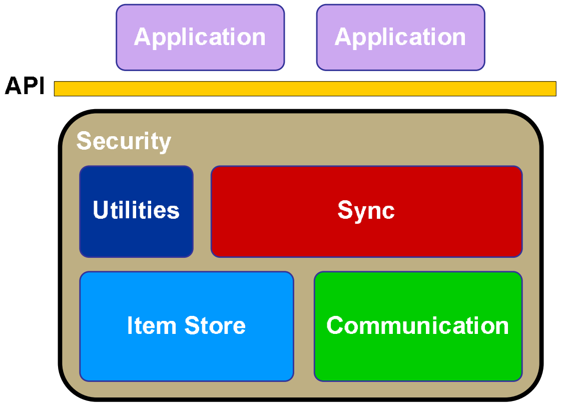

Figure 2: Cimbiosys software architecture

The *Sync* module implements the synchronization protocol described in detail in Section 5. During synchronization, it enumerates versions of items in the Item Store that are unknown to the remote sync partner and sends these along with the appropriate metadata, such as versioning information. The remote partner then adds the received items to its Store, possible replacing older versions of these items. We are considering allowing devices to keep multiple versions if requested by an application, but our current implementation retains only the latest known version of each item.

Cimbiosys also includes a number of *Utilities* for recording information about regular synchronization partners, naming collections and devices, managing access controls, and so on.

Security considerations permeate the Cimbiosys design. For example, all versions of items are digitally signed by the originating device, and collection-specific policies dictate which devices are allowed to create, update, and delete items in a collection. A full discussion of the security design is beyond the scope of this paper.

Applications interact with the Cimbiosys platform using a specially developed application programming interface (API). Through this API, an application can create a new collection, create a local replica for an existing collection, add items to a collection, update and delete items, run queries over items, initiate synchronization between a local and a remote replica, establish regular synchronization partnerships, change access permissions, and even change a replica's filter. Legacy applications that read and write local files, rather than using the Cimbiosys API, are supported by "watcher" processes that monitor file system directories and import files into (or delete items from) a local replica.

4.3 Implementation

Cimbiosys has been implemented in two different environments. One implementation is in C# using Microsoft's .NET Framework running on Windows. We have not yet ported this code to Windows Mobile 6.0, where it could run on handheld mobile devices. The other implementation is in Mace, a C++ language extension that supports distributed systems development [6]. This Mace implementation serves as the basis for the evaluation presented in Section 7.

Two applications have been designed and are intended for deployment in our lab. Cimetric, implemented in C#, is a collaborative authoring tool. It coordinates access and updates to the complex, heterogeneous set of text, graphics, and data files created and modified in the process of writing a paper. Authors receive their own replicas of the paper, perform local updates, and make those updates visible to coauthors when they are ready to share a new version. CimBib is designed as a bibliographic database and personal digital library in which colleagues can share references to local and remote copies of published papers as well as personal annotations and recommendations; this application is still in a user-centered design phase. The designs of both Cimetric and CimBib were informed by a qualitative field study of scholarly writing and reference use [8].

5 CIM Sync

This section focuses on one key aspect of the Cimbiosys platform, namely the CIM Sync protocol. Supporting content-based filtering with a fluid synchronization topology while ensuring eventual filter consistency and knowledge singularity requires novel per-replica metadata as well as new techniques for acquiring and discarding items as the collection is updated and replicas possibly change their filters.

5.1 Metadata

The sync protocol relies on a variety of per-item and per-replica metadata. Each collection and each item in the collection has a unique identifier, as does each replica of the collection. Each version of an item also has a unique identifier called its *version-id*. Whenever an item is created, updated, or deleted, the replica on which this operation is performed creates a new version-id for the item consisting of the replica's identifier coupled with a counter of the number of update operations that have been performed by that replica. Deleted items are simply marked as deleted; such items are referred to as *tombstones*.

For each item in a replica, the Cimbiosys item store maintains the item's unique identifier, version-id, XML+file contents, and other information used to detect whether different versions of the item are in conflict (similar to the made-with knowledge used in WinFS [11]). Only the latest known version of each item is retained in the item store. Older versions are considered obsolete.

Each replica maintains *knowledge* indicating the set of versions that are known to the replica. Conceptually, a replica's knowledge is simply a set of version-ids. It contains identifiers for any versions that (a) match the replica's filter and are stored in its item store, (b) are known to be obsolete, or (c) are known to not match the replica's filter. Including this third class of versions, *out-of-filter versions*, and using a novel representation called *item-set knowledge* distinguishes the knowledge used in CIM Sync from that of other replication protocols like Bayou that do not support partial replication.

Knowledge consists of one or more fragments where each fragment is a version vector [12] and an associated explicit set of item ids. The version vector component indicates, for each replica that has updated any item in the collection, the latest known version-id generated by the replica. Semantically, if a replica holds a knowledge fragment S:V then the replica knows all versions of items in the set S whose version-ids are included in the version vector V. Note that the version vector may include versions of items whose ids are not in the associated set, but those versions have no significance. When a replica's knowledge contains multiple fragments, the replica's overall knowledge is the union of the version-ids from each fragment.

A knowledge fragment may specify "*" as the item-set, meaning that the set includes all items in the collection. Such fragments are called *star-knowledge*. In a system consisting entirely of full replicas, each replica's knowledge is always a single star-knowledge fragment. Partial replicas introduce the need for item-set knowledge fragments in addition to star-knowledge. For instance, when synchronizing from a partial replica, the requesting replica obtains item-set knowledge reflecting the set of items stored by the partial replica.

5.2 Basic protocol

Cimbiosys uses a one-way, pull-style synchronization protocol. A replica, called the *target replica*, initiates synchronization with another replica, called the *source replica*, by sending a SyncRequest message. This message includes the target's knowledge and its filter. The source replica then checks its item store for any items whose version-ids are not known to the target replica and whose XML contents match the target's filter. The XML contents, file contents, and metadata for each of these items are returned to the target. If possible, as discussed below in Section 5.4, the source replica also informs the target replica of items that no longer match its filter. Finally, the source replica responds with a SyncComplete message including one or more knowledge fragments that are added to the target's knowledge. At the very least, this *learned knowledge* includes knowledge pertaining to items transmitted during this synchronization session but may include additional versions as discussed in Section 5.3. All of the messages received by the target replica result in updates that are applied to its item store as a single atomic transaction.

Figure 3 illustrates a synchronization session in which the digital photo frame (replica B) requests items from the laptop (replica C) in our previously discussed photo sharing scenario. The state shown for each device is the metadata and item store *before* synchronization. The arrows show the messages that are sent during synchronization. Note that the photo frame's knowledge that is sent in the SyncRequest message indicates that it knows about four items, but has not seen any updates from the laptop since version C:1. The laptop returns a more recent version of item r and a new item s. The laptop also had updated item k to reduce its rating, and hence notifies the photo frame that this item is no longer of interest. The final message informs the photo frame of the knowledge it learned from the laptop. The following sections present in more detail the protocol features resulting from the need to support partial replicas and refer back to aspects of this figure.

5.3 Acquiring knowledge

As replicas receive items during synchronization, they add the items version-ids to their knowledge, but need some other means of learning about obsolete and out-of-filter versions. The SyncComplete message at the end of the synchronization protocol conveys knowledge that the target replica learned during this sync session. The target replica adds this learned knowledge to its own knowledge, generally as new knowledge fragments. This knowledge can include any version-ids for items currently stored by the source replica as well as any ids for versions that the source knows to be obsolete. It may not, however, include versions that are out-of-filter at the source replica but could match the target replica's filter as this would cause the target replica to fail to receive such versions from other replicas.

The learned knowledge, therefore, depends on the relationship between the filters of the synchronizing replicas. If the source replica's filter is no more restrictive than the target's filter, that is, if any item that matches the target's filter also matches the source's filter, then the source replica can send its complete knowledge in the SyncComplete message. The reason is that any out-of-filter versions included in the source's knowledge are also out-of-filter with respect to the target replica. In other cases where the target has a broader filter or an incomparable filter compared to the source, the source replica must restrict the conveyed learned knowledge to those items that it actually stores. For example, this is the case in Figure 3 where the photo frame wants 5-star photos while the laptop holds photos that have been tagged with the "family" keyword.

5.4 Move-out notifications

A partial replica not only needs to receive newly created and updated items that match its filter but also wants to be informed when currently stored items have later versions that no longer match its filter. Such items are said to have *moved out* of the replica's interest set. During synchronization, the target replica may receive *move-out notifications* from the source replica, causing the target to remove specified items from its item store. In Figure 3, for instance, the laptop sends such a notification for item k, which it had updated. There are two conditions under which the source returns move-out notifications.

If the source replica stores an item whose version is not known to the target replica and whose contents does not match the target's filter, the source can send a move-out notification for this item. However, a target replica would receive move-out notifications for items that it does not store whenever those items are updated and continue to not match the target's filter, a potentially common occurrence. Such spurious messages will simply be ignored by the receiving replica, but they do consume network and processing resources.

To avoid spurious move-out notifications, a SyncRequest message may optionally include a set of identifiers for items that are stored by the requesting replica. The source replica only sends move-out notifications for items that are in this set. Replicas cache this item set for their regular synchronization partners, allowing these partners to send deltas, that is, to send just the set of newly acquired items.

Sending move-out notifications for items that are stored at the source replica is insufficient. Consider the case of a replicated customer relationship database in which a server holds the complete database, Bob's laptop holds items for all California customers, and his cell phone stores items for customers that live in Los Angeles. Bob's cell phone synchronizes periodically with his laptop but never directly with the server database. Suppose that a customer moves from Los Angeles to Chicago. When Bob's laptop synchronizes with the server, it receives a move-out notification causing the laptop to drop this customer from its local replica. But then how does Bob's cell phone learn that it also should discard this item?

The second condition for sending a move-out notifi-

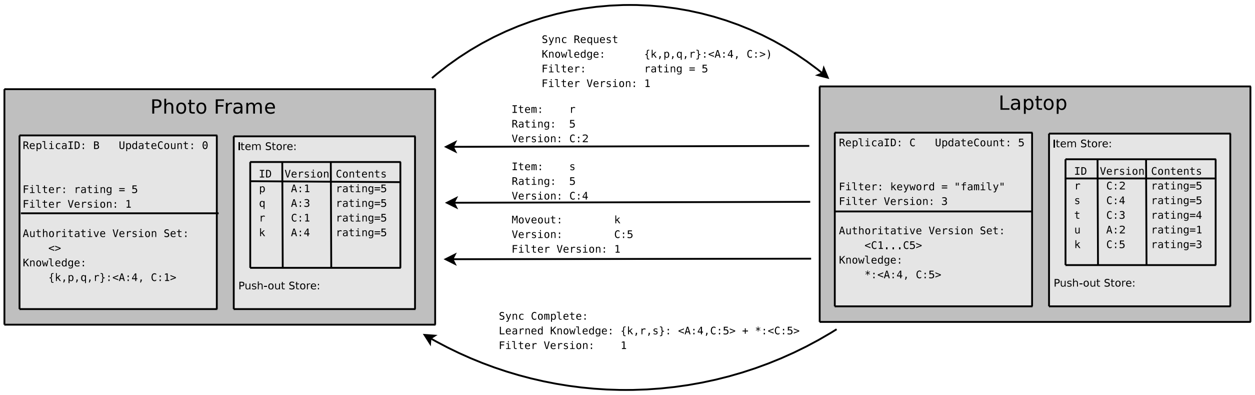

Figure 3: Example synchronization between two replicas

cation for an item is as follows: the target replica stores the item, the source replica does not store the item, the source replica's filter is no more restrictive than the target's filter, and the source's knowledge for this item is greater than the target's knowledge. In other words, if the source is interested in all items of interest to the target and is more knowledgeable than the target, it can deduce that any items it does not store should also be removed from the target's item store. Once again, this relies on the source being informed of the set of items that are stored by the target.

Since replicas are allowed to change their filters at any time, a target replica may receive move-out notifications based on a previous filter that no longer apply. To guard against processing out-of-date notifications, a replica increments a counter whenever it updates its filter. Essentially, this counter serves as a version identifier for the replica's filter. The filter version is included in each synchronization request and is returned in each move-out notification. Move-out notifications that include old filter versions are simply ignored by the receiving replica.

5.5 Out-of-filter updates

When updating a locally stored item, an application may cause that item to no longer match the local replica's filter. For example, consider a cell phone whose filter selects unread e-mail messages. After displaying a message, the cell phone updates the corresponding item to set the read-flag. Eventually, the cell phone should discard this item, but it cannot be discarded immediately since other replicas need to learn of this update. Operations that produce new versions of items that do not match the local replica's filter are called *out-of-filter updates*.

To preserve versions produced by out-of-filter updates, the updated items are placed in a special portion of the updating replica's item store called the *push-out store*. Items in the push-out store are not visible to applications, but are treated like any other item during synchronization. In particular, such items are sent to a synchronization partner if they match its filter, and may be overwritten by items received from a sync partner, possibly causing the item to move back into the regular item store.

Unfortunately, a replica may not have any synchronization partner whose filter matches the items in its push-out store. Thus, when synchronizing with any replica with an equal or less restrictive filter, a replica sends all items in its push-out store, and then optionally discards these items once it learns that they were successfully received by the target replica. This partner accepts these items even if they don't match its filter. Such items may end up in the target replica's push-out store, from where they are passed to another replica. However, this could lead to situations in which two replicas play "hot potato" by passing back and forth an item that matches neither of their filters. Section 6 discusses restrictions that Cimbiosys places on the synchronization topology to avoid the hot potato problem and guarantee that out-of-filter updates eventually reach all interested replicas.

5.6 Changing filters

When a replica changes its filter it may need to discard items or knowledge or both depending on the nature of the filter change. If the new filter is more restrictive than the previous filter, that is, it matches fewer items, then items that no longer match the filter are moved to the replica's push-out store. The replica cannot simply discard such an item since it may be the only replica that holds the latest version. As discussed above, items from the replica's push-out store will eventually be discarded after they are passed to another replica (or it is determined that they are already stored by another replica). Although some in-filter versions may become out-of-filter versions, the replica's knowledge does not change.

If the new filter is less restrictive than the previous filter, then out-of-filter versions may now match the new filter. Such versions need to be removed from the replica's knowledge so that it will receive them during future synchronizations. Unfortunately, the replica cannot determine which versions in its knowledge are out-of-filter and which are obsolete. So, conservatively, its knowledge must be reduced to include only versions of items that it already stores. Any star-knowledge fragments are reduced to item-set knowledge.

If the new filter is neither less restrictive nor more restrictive than the previous filter, that is, if the old and new filters are incomparable, then both cases apply. The replica may need to move non-matching items to its push-out store. The replica also needs to reduce its knowledge.

As mentioned earlier, each replica maintains a counter of the number of times that its filter has been changed. During synchronization with regular partners, a replica can send its filter version counter as a shorthand for its actual filter, which could be a complicated and long query. Replicas maintain soft-state mapping their partners' filter versions to the actual filter queries. If a replica discards this soft-state, it responds to a sync request with a special error code, causing the requesting replica to send its full filter in a retransmitted sync request.

5.7 Compacting knowledge

Whenever a replica synchronizes with another replica, it receives new knowledge fragments as described in the previous section. To reduce the number of fragments in its knowledge and the overall size, a replica can compact its knowledge using a set of simple rules. For example, suppose the replica's knowledge includes two fragments, S_1:V_1 and S_2:V_2. If the set S_1 is a subset of set S_2 and the version vector V_2 dominates V_1, meaning that any versions in V_1 are also included in V_2, then the fragment S_1:V_1 is redundant and can be discarded. If V_1 and V_2 are identical, then the sets S_1 and S_2 can be combined into a single knowledge fragment. Figure 4 depicts these and other compaction rules that can be applied any pair of knowledge fragments.

$S_1{:}V_1 + S_2{:}V_2 \Rightarrow$

	$S_1 \subset S_2$	$S_1 = S_2$	$S_1 \supset S_2$	*otherwise*
$V_1 \subset V_2$	$S_2{:}V_2$	$S_2{:}V_2$	$S_2{:}V_2$ + S_1-$S_2{:}V_1$	$S_2{:}V_2$ + S_1-$S_2{:}V_1$
$V_1 = V_2$	$S_2{:}V_2$	$S_1{:}V_1$	$S_1{:}V_1$	$S_1 \cup S_2{:}V_1$
$V_1 \supset V_2$	$S_1{:}V_1$ + S_2-$S_1{:}V_2$	$S_1{:}V_1$	$S_1{:}V_1$	$S_1{:}V_1$ + S_2-$S_1{:}V_2$
otherwise	$S_1{:}V_1 \cup V_2$ + S_2-$S_1{:}V_2$	$S_1{:}V_1 \cup V_2$	$S_1{:}V_1 \cup V_2$ + S_1-$S_2{:}V_1$	$S_1{:}V_1$ + $S_2{:}V_2$

Figure 4: Knowledge compaction rules

While these knowledge compaction rules are effective, they don't always lead to compact knowledge in practice. Consider the case of Alice who edits photo r on her laptop (replica C) producing a new version with version-id C:1, then edits this same photo again to produce a newer version C:2. Alice also adds keywords to photos t, s, and k, producing versions C:3, C:4, and C:5. Suppose that these items all match replica C's filter and are never updated by other replicas. The state of replica C on Alice's laptop is as shown in Figure 3. When Alice's home PC synchronizes from her laptop and receives some of these items, it will acquire the knowledge fragment $\{k, r, s\}$:<C:5> indicating that it learned about versions C:1 through C:5 for items k, r, and s. Unfortunately, this knowledge fragment would never be compacted with other knowledge fragments.

Key to reducing the number of fragments in a replica's knowledge is the notion of authority. A replica is *authoritative* for a version of an item if it either stores the item or knows the item to be obsolete. Recall from Section 5.3 that version-ids for any stored or obsolete versions can be included in the learned knowledge acquired by a target replica at the completion of the synchronization process. The source replica, therefore, can return a learned knowledge fragment in which the item-set is "*", meaning all items in the collection, and the associated version vector includes identifiers for its authoritative versions. In other words, during synchronization, the target replica learns of any versions of any items for which the source replica is authoritative. Moreover, when the target replica's filter is equal to or less restrictive than the source's filter, the target replica becomes an authority for all of the source replica's authoritative versions.

In our previous example, the laptop (replica C) is authoritative for all of the versions that it produced, that is, for versions C:1 through C:5. Thus, replica C sends $*$:<C:5> as learned knowledge when synchronizing to any other replica. This knowledge fragment is merged into the receiving replica's star-knowledge, and hence does not lead to an increase in the overall number of knowledge fragments.

One practical issue remains, namely how to deal with out-of-filter updates and filter changes. Such operations cause items to be placed in a replica's push-out store. The replica will cease to be authoritative for its own versions that are pushed to another replica and then discarded. Requiring a replica to store indefinitely all of the items that it creates would be unreasonable. For instance, a digital camera often offloads its photos to a laptop in order to free up storage space for new photos. In practice, the system simply needs to maintain the invariant that there exists at least one replica that is authoritative for every version ever generated.

In Cimbiosys, when a replica sends the items in its push-out store to replica with a less restrictive filter, the receiving replica becomes authoritative for these items. The sending replica can then discard such items without violating the system-wide invariant. Each replica records the version-id of the most recent version it has generated for which it is no longer authoritative. The replica then knows that it is authoritative for any versions it has produced with greater version-ids. The learned knowledge sent by a replica is a star-knowledge fragment containing the range of version-ids from the first version generated after its last push-out to its most recently generated version. A replica that has received multiple star-knowledge fragments containing overlapping or contiguous version ranges can combine these together into a single fragment.

For example, suppose Alice's laptop (replica C) changes its filter so that it no longer wants items with ratings below four. Version C:5 of item k no longer matches. After pushing this item to Alice's home PC (replica A), as well as sending the latest versions of all other items, the home PC will have learned $*$:<C:5>. At this point, the laptop discards item k and records C:5 as its last unauthoritative version. Now, suppose that Alice performs three more updates from her laptop producing versions with identifiers C:6, C:7, and C:8. During synchronization to another replica, say Alice's photo frame (replica B), C will pass $*$:<C:6..C:8> as learned knowledge. When the photo frame synchronizes from the home PC, it will receive learned knowledge of $*$:<C:5> in addition to knowledge of other versions for which Alice's home PC is authoritative. The photo frame then combines the knowledge received from the laptop with that received from the home PC to get a knowledge fragment of $*$:<C:8>, which in turn is merged with its other star-knowledge.

As a replica synchronizes from various other replicas in the system, it acquires star-knowledge fragments from each of these sync partners. Such fragments are combined together into a single star-knowledge fragment that is monotonically increasing (provided the replica does not expand its filter). Given a synchronization topology in which each replica regularly synchronizes with a set of partners that collectively are knowledgeable about all versions in the system, each replica will converge towards singular knowledge. The following section describes how Cimbiosys ensures that replicas are configured in such a topology.

6 Guaranteeing Synchronization Properties through Tree Topologies

The CIM Sync protocol presented in the previous section performs correctly for any set of replicas with arbitrary filters and arbitrary synchronization patterns. When a replica synchronizes with any other replica, it will receive all versions stored by its sync partner that match its filter. Likewise, it will never receive items that do not match its filter, and it will receive whatever move-out notifications can be generated by the sync partner given its current state. Moreover, a replica never receives the same version from multiple sync partners (unless it engages in parallel synchronizations or changes its filter). But these guarantees are insufficient to ensure that Cimbiosys performs appropriately in a range of real-world situations; additional constraints must be placed on the synchronization topology in order to achieve other desired system properties, such as eventual knowledge singularity.

Cimbiosys forces replicas of a given collection to configure themselves into a hierarchically filtered tree topology. In particular, each replica has a single parent replica, except for the replica at the root of the tree, and a replica's filter must be at least as restrictive as that of its parent. In other words, a parent replica stores any items that are stored by any of its children. The replica at the root of the tree has the "*" filter that matches all items; that is, it stores a full copy of the collection. This root replica is called the *reference replica* for the collection. Parent and child replicas are required to perform synchronization in both directions, at least occasionally.

Constructing the tree is easy. When a new replica is created for a collection, it asks an existing replica to be its parent. If the filter of the requested parent is too restrictive, then the new replica walks up the existing tree until it finds a replica that can serve as its parent. At the very least, the reference replica can always serve as a parent for any replica with an arbitrary filter. If a replica wishes to retire gracefully from a collection, then this replica should notify its children so they can select a new parent. The retiring replica's parent, for instance, can serve as the new parent for its children, or, in some cases, one of the existing children can be promoted to be the parent of its siblings. A replica can change its parent at any time as long as it chooses a new parent with a suitable filter and does not violate the tree structure. For instance, a replica may be required to find a new parent when it expands its filter or its previous parent dies unexpectedly.

The tree synchronization topology provides five important benefits.

One, the synchronization topology is well-connected. That is, groups of replicas for the same collection cannot remain disconnected indefinitely, and hence cannot prevent eventual convergence.

Two, each version of an item has a guaranteed path by which it can travel from the originating replica to any other replica whose filter matches the version. Specifically, when a new version is created, it can flow up the tree from child to parent replicas until it reaches common ancestors, including the reference replica. Any versions held by the reference replica can flow to any other replica over a path of replicas with increasingly restrictive filters.

Three, move-out notifications can be delivered by a parent to any of its children. Recall from Section 5 that move-out notifications can be sent only when the source replica has a filter than is no more restrictive than the target. This is exactly the case for replicas with a parent-child relationship. Thus, the tree topology guarantees that all replicas are able to receive appropriate move-out notifications. Essentially, such notifications flow down the tree.

Four, out-of-filter versions in a replica's push-out store flow up the tree until they reach replicas that are interested in those items. During synchronization from a child replica to its parent, the child sends all of the items in its push-out store, regardless of whether they match the parent's filter. The tree topology prevents replicas from playing "hot potato" with out-of-filter versions.

Finally, the tree topology ensures eventual knowledge singularity. As authoritative versions are passed up the tree, a parent replica assumes authority for any versions generated by any of its children or descendants. Eventually, all authoritative versions arrive at the reference replica, which then produces a single star-knowledge fragment containing all of these versions. This star-knowledge fragment is then passed down the tree from the reference replica to all other replicas during parent-to-child synchronizations. In the absence of further updates or filter changes, each replica's knowledge will eventually converge to that of the reference replica.

Although these benefits argue convincingly for having a tree-structured synchronization topology, extended synchronization patterns are not prevented. In Cimbiosys, a replica can choose arbitrary synchronization partners (in addition to its parent and children). The only restriction is that the overall synchronization topology must include an embedded tree with a reference replica.

All practical usage scenarios that we've envisioned meet this condition, including the motivating scenarios presented in Section 2. For example, in the photo sharing scenario, Alice's home PC serves as the reference replica for her photo collection. Her laptop and digital photo frame synchronize directly with this PC, and treat it as their parent, as do the cloud-based services that contain selected photos. However, Alice's laptop might also sync with such services on occasion or sync directly with friends laptops. Services might replicate data among themselves within the cloud, unbeknownst to the reference replica, such as for geographic scaling. The digital camera, which only synchronizes with the laptop, uses the laptop as its parent replica. The overlaid tree topology ensures that Alice's new photos will eventually find their way into her master photo collection as well as onto other devices with selective filters.

7 Evaluation

7.1 Simulation Results

We simulated a scenario consisting of 10 replicas. Their filter relationships form a three-level hierarchy of one replica at the top, three in the middle, and six at the bottom. A replica's filter subsumes the filters of all replica at lower levels and forms a DAG. The simulation workload had three serial phases: 1) 1000 inserts of new items at randomly chosen replicas, 2) 400 syncs between randomly chosen sync partners, and 3) another 400 random syncs interleaved with 1000 updates to random items. Even though the syncs were peer-to-peer, a replica syncs as a child or a parent whenever the randomly selected sync partner turns out to have a superior or an inferior filter.

We compare the behavior of three related techniques to represent knowledge: CIM, which is Cimbiosys; PIVV, which maintains individual version vectors for each item that matches a replica's filter, sends this entire knowledge during syncs, and receives in return version vectors pertaining only to items in its filter; and CIM-PIVV, which is a modified version of Cimbiosys in which item set knowledge is fully expanded into per-item version vectors but is discarded whenever the star knowledge subsumes them. In other words, CIM-PIVV maintains per-item version vectors but uses the notion of star knowledge accumulated through the hierarchy to converge to a single knowledge fragment.

Figure 5 shows the average size of knowledge kept at each replica over simulation time. This figure confirms our expectations about the behavior of the studied techniques; size of knowledge in PIVV increases as items are replicated and reaches a high value proportional to the number of items stored in the replica; whereas, in CIM and CIM-PIVV, knowledge is fragmented in the initial stages but eventually converges to a small size not dependent on the number of stored items; in other words, it eventual knowledge singularity. This trend also holds for the size of knowledge transmitted over the network (not shown here) during synchronization.

In addition, Figure 5 illustrates the effect of workload characteristics on the potential benefits of CIM. CIM provides significant savings over CIM-PIVV when synchronizing the initial phase of inserts (0 to 4000 time

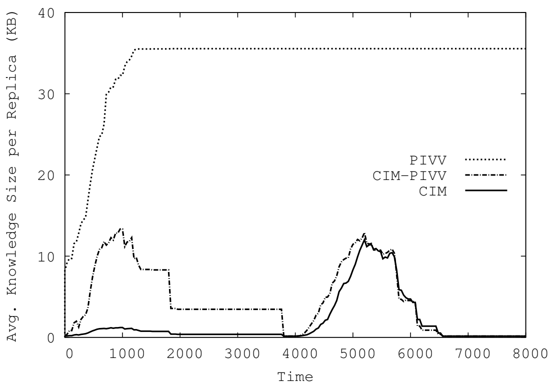

Figure 5: Average size of knowledge per replica vs. time

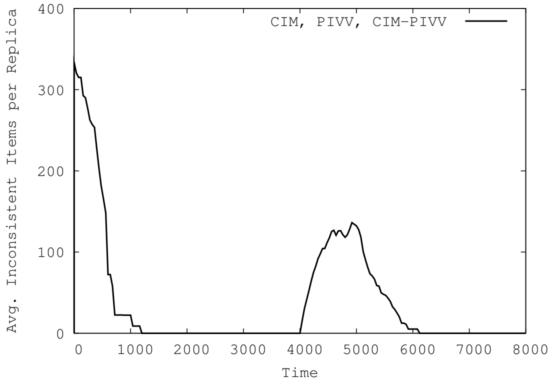

Figure 6: Average inconsistent items per replica vs. time

units), whereas the savings is marginal when synchronizing the updates in the third phase (4000 to 8000 time units). The key reason for this behavior is that the initial inserts were performed in batches, providing opportunities for CIM to compact knowledge into item sets. In contrast, the subsequent updates are spread over multiple rounds of syncs, leading to a highly fragmented state close to CIM-PIVV. Yet, CIM and CIM-PIVV are far superior to a vanilla version vector technique through the use of star knowledge to expunge subsumed knowledge fragments.

We next show the progress made by replicas in achieving eventual filter consistency. Figure 6 plots the average number of inconsistencies in a replica's store over simulation time. Here, an inconsistency at a replica R at a certain time includes: a) an item in R's filter but not in R's store, b) an item in R's store that is obsolete, and c) an item not in R's filter but in R's store. We used the simulator's global view of the system to count inconsistencies.

Figure 6 confirms that all three simulated solutions for partial replication eventually converge (at the same rate), both after the insert and the update phases. Moreover, a comparison of Figures 5 and 6 indicates that the system could achieve filter consistency much earlier than knowledge singularity.

7.2 Experimental Results

As a follow-on to our simulation, we ran experiments with the Mace implemention to further compare the CIM Sync protocol to traditional version vector schemes and examine the effects of various tunable parameters of CIM Sync. In particular, we answer the following questions with respect to the goals of Cimbiosys:

- Are the knowledge storage requirements for CIM Sync significantly smaller than those for per-item version vector protocols?
- Does the bandwidth usage of CIM Sync leverage the reduction in knowledge size?
- What are the benefits of requiring a filter hierarchy?
- How does the incorporation of non-hierarchical synchronizations affect the knowledge size, bandwidth usage, and time to convergence?

7.2.1 Experimental Environment

We evaluate the Mace implementation of the CIM Sync protocol using ModelNet [18] to simulate a variety of network topologies, and Plush [1] to deploy and run the topologies and protocol code on a cluster of machines. Each machine in the cluster is a Dell PowerEdge *SC*1425 with dual Intel Xeon 2.8GHz processors and 2GB of RAM, running Linux 2.6.9-22. For these tests, we use a system of 10 replicas, and a collection size of 10,000, which reflects the average size of a collection for our motivating scenario, photo sharing. Using ModelNet, we emulate a clique of 10 routers, each connected to a single replica. The link speed between all routers and replicas is set to 100 Mbps so that large experiments complete within a reasonable time frame. The trends in the experimental results are similar with lower bandwidths.

We vary the number of partial replicas in the system, the hierarchy created by filters (where appropriate), and the likelihood that a replica will synchronize with its parent or children, or with a random peer during any given synchronization operation.

Each experiment consists of 2 phases. During Phase 1, replicas create items such that the total number of items in the system at the conclusion of phase 1 is 10,000. During phase 2, synchronizations proceed until the knowledge at all replicas converges to a stable state.

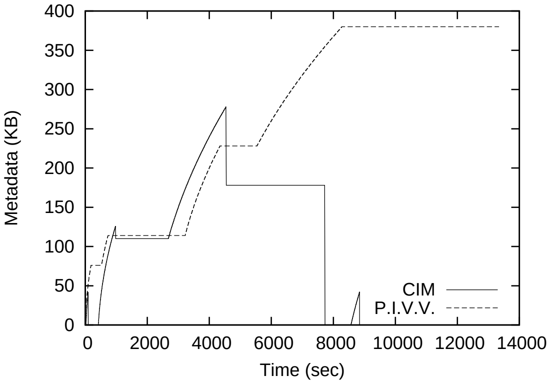

Figure 7: Knowledge Size: Partial Replica

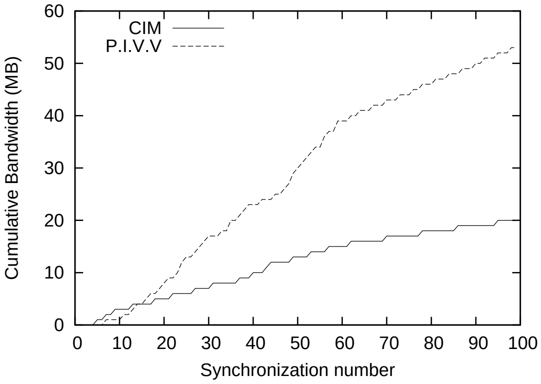

Figure 9: Bandwidth Usage: Reference Replica

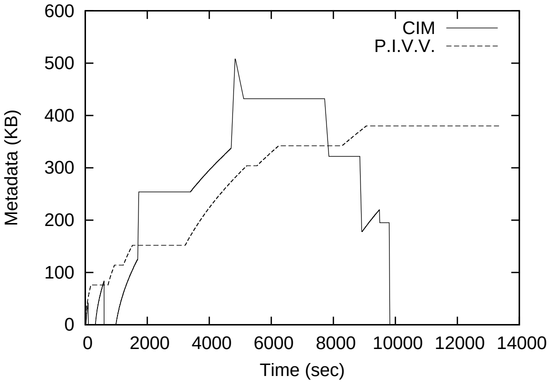

Figure 8: Knowledge Size: Reference Replica

7.2.2 Knowledge Storage

Figures 7 and 8 compare the knowledge storage requirements of the CIM Sync representation to those of the per-item version vector implementation, for a partial and reference replica, respectively. The system shown has 10 replicas, a collection size of 10,000, and a binary tree filter topology. Each replica is biased to send 50% of its synchronization requests to a parent or child (if possible), and to send the remainder of requests to a random peer.

During the initial synchronizations, the CIM Sync representation shows spikes in the knowledge storage requirements; these spikes demonstrate the significance of the knowledge compaction process. As the target replica receives knowledge about individual items from the source replica, it generates item set fragments to add to its knowledge. However, once it receives learned knowledge at the completion of the synchronization, the target compacts its knowledge. The target may not be able to completely compact its knowledge, as it may be synchronizing with a source replica that cannot speak authoritatively about all versions sent. Once the target has synchronized with enough peers and has received enough learned knowledge, it can compact its knowledge into a single star-knowledge fragment. Contrastingly, the per-item version vector representation requires that a version vector be stored per item in the system. Therefore, the knowledge stored for this representation remains on the order of the number of items in the system.

Additionally, as is apparent during the initial synchronizations, especially those of the reference replica, the CIM Sync protocol incurs overhead as a result of storing the authoritative version set, but this penalty is on the order of the number of replicas in the system, which is expected to be much smaller than the number of items.

7.2.3 Bandwidth Usage

The experiment that measures cumulative bandwidth usage of the two systems with respect to synchronization iteration is identical to the one used to study knowledge storage requirements. Figure 9 shows the result of this experiment, where bandwidth is measured as the sum of all data as it is queued into the transport layer. Each data item for this experiment is a 6 character string and therefore data content does not contribute significantly to the amount of bandwidth used. Since the reference replica has the largest filter, synchronizations in which the reference replica is the source generally result in the transmission of more data than do synchronizations between replicas with arbitrary filter overlap. As such, we compare bandwidth usage for the two systems at the reference replica.

During early synchronizations, the two knowledge representations are comparable, as information must be sent for each item which is new to the target replica. However, once the system reaches a stable state, the bandwidth usage of the CIM Sync knowledge representa-

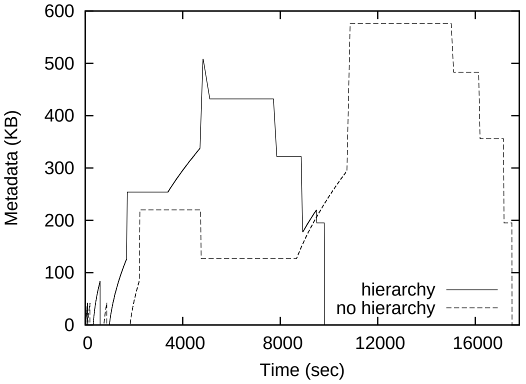

Figure 10: Effect of Topology Restrictions

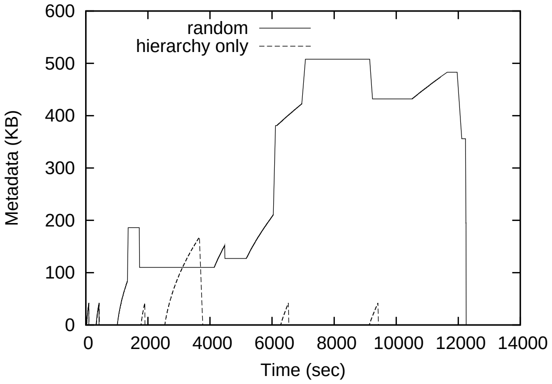

Figure 11: Effects of Out-of-Hierarchy Synchronization

tion are reduced drastically. This is because at this point, the knowledge for any given replica has been reduced to a single version vector. When the target of a synchronization operation sends knowledge to the source replica, it needs only send this single version vector, whereas the per-item version vector implementation requires that a version vector per item be sent.

7.2.4 Topology Restrictions

We examine the effects of the restriction that Cimbiosys replicas should have a hierarchy, overlaid upon the network topology. For these experiments, we measure Cimbiosys with 10 replicas and a collection size of 10,000, a binary tree filter topology, and an identical set of filters to that of the previous experiments. For one of the experiments depicted in Figure 10, each replica treats every sync as though it occurs with a peer outside of its filter hierarchy, whereas the filter relationships are leveraged in the second experiment, with a 50% bias toward synchronization with a replica's parent or children as opposed to synchronization with an arbitrary peer.

This experiment shows the benefits of leveraging parent-child relationships between replicas. Replicas can accept knowledge from their parents and can then directly merge this knowledge with their own, as they know at the completion of a synchronization with a parent that all versions included in the parent's knowledge should be included in their own. Similarly, replicas can become authoritative on versions authored by their descendents, and this information can flow up the hierarchy until it reaches a reference replica, at which point it flows downward in compact form. Without a hierarchy, replicas can only claim authority over versions they themselves have authored, and therefore, even if a replica R hears about a particular version authored by a replica S, until R synchronizes with S, R is unable to compact its knowledge. Thus, while we can still achieve eventual knowledge singularity even without a filter hierarchy, convergence time does improve with the use of a hierarchy.

7.2.5 Out-of-Hierarchy Synchronizations

Finally, we examine the effects of allowing for synchronizations with peers outside of a replica's filter hierarchy. Figure 11 compares a system in which replicas only synchronize with their parents and children with a system in which replicas select synchronization peers at random.

As is clear from the figure, restricting synchronizations to parents and children allows knowledge to converge much more quickly. This is because knowledge tends to flow within a hierarchy in a more compact form, as discussed above. However, synchronizations with arbitrary peers do allow knowledge about data items to potentially flow more quickly between replicas, at the cost of less compact knowledge for a period of time.

8 Conclusion

Cimbiosys, a new platform that provides filtered replication of content through peer-to-peer synchronization, was motivated by the needs of loosely-organized communities and of individuals managing multiple devices. The chief contributions of the protocol and architecture lie in the platform's ability to support efficient partial replication using a compact representation of accumulated per-replica knowledge. By using metadata that describes obsolete items and items that no longer match a replica's filter (move-outs and push outs) in addition to metadata that describes the replica's in-scope item updates, Cimbiosys guarantees efficient delivery of items and that each collection replica achieves eventual consistency. Our evaluations demonstrate the key property of eventual knowledge singularity, that is, the ultimate convergence of knowledge to a form that supports a minimal exchange of data at synchronization time, especially in comparison to other version vector replication methods.

Although we have evaluated Cimbiosys using artificial workloads that reflect real-world conditions (e.g. we used the current capacity of a popular player to estimate the size of consumer music and photo collections), we feel it is important to deploy an initial set of Cimbiosys applications to test the platform's real-world performance and to explore the utility of our approach. Development and fielding two complementary applications will also allow us to assess any possible gaps in our assumptions (e.g. about collection characteristics, device synchronization patterns, and update frequency) and to identify the practical strengths and weaknesses of our approach.

Acknowledgements

We would like to thank Daniel Peek for many early design discussions. Rama Kotla helped our understanding of PRACTI and other related work. Chip Killian, James W. Anderson, and Ryan Braud provided much assistance with Mace and ModelNet. Lev Novik, Mike Clark, and Moe Khosravy helped us with our C# implementation.

References

[1] J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote control: Distributed application configuration, management, and visualization with Plush. In *USENIX Large Installation System Administration Conference (LISA)*, Nov. 2007.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX symposium on Networked systems design and implementation (NSDI'06)*, pages 59–72, May 2006.

[3] D. Dearman and J. S. Pierce. It's on my other computer!: Computing with multiple devices. In *Proceeding of the 26th annual SIGCHI conference on Human factors in computing systems (CHI'08)*, pages 767–776, New York, NY, USA, 2008. ACM.

[4] S. Farnham, E. Pedersen, and R. Kirkpatrick. Observation of Katrina/Rita groove deployment. In *Proceedings of the 3rd International ISCRAM Conference (ISCRAM'06)*, May 2006.

[5] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–71, June 1990.

[6] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pages 179–188, New York, NY, USA, 2007. ACM.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[8] C. C. Marshall. From writing and analysis to the repository: Taking the scholars' perspective on scholarly archiving. In *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries (JCDL'08)*, New York, NY, USA, 2008. ACM.

[9] C. Neustaedter and A. J. Brush. "LINC-ing" the family: The participatory design of an inkable family calendar. In *Proceeding of the 24th annual SIGCHI conference on Human factors in computing systems (CHI'06)*, pages 141–150, New York, NY, USA, 2006. ACM.

[10] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th symposium on Opearting systems design and implementation (OSDI'04)*, pages 363–378, Berkeley, CA, USA, 2004. USENIX Association.

[11] L. Novik, I. Hudis, D. B. 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.

[12] D. S. Parker, Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. A. Edwards, S. Kiser, and C. S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.

[13] D. Peek and J. Flinn. EnsemBlue: integrating distributed storage and consumer electronics. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI'06)*, pages 219–232, Berkeley, CA, USA, 2006. USENIX Association.

[14] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM symposium on Operating systems principles (SOSP'97)*, pages 288–301, New York, NY, USA, 1997. ACM.

[15] D. Ratner, P. Reiher, and G. J. Popek. Roam: a scalable replication system for mobility. *Mobile Networks and Applications*, 9(5):537–544, 2004.

[16] D. Ratner, P. L. Reiher, G. J. Popek, and R. G. Guy. Peer replication with selective control. In *Proceedings of the First International Conference on Mobile Data Access (MDA'99)*, pages 169–181, London, UK, 1999. Springer-Verlag.

[17] B. Salmon, F. Hady, and J. Melican. Learning to share: a study of sharing among home storage devices. Technical Report CMU-PDL-07-107, Parallel Data Laboratory, Carnegie Mellon University, Oct. 2007.

[18] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becke. Scalability and accuracy in a largescale network emulator. In *Proceedings of the 5th ACM/USENIX symposium on Operating system design and implementation (OSDI'02)*, Dec. 2002.