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

ıcreasingly people manage and share information ɔross a wide variety of computing devices from cell hones to Internet services. Selective replication of ontent is essential because devices, especially portable nes, have limited resources for storage and communica- on. *Cimbiosys* is a novel replication platform that per- ıits each device to define its own content-based filtering riteria and to share updates directly with other devices.

In the face of fluid network connectivity, redefinable ontent filters, and changing content, Cimbiosys ensures vo properties not achieved by previous systems. First, very device eventually stores exactly those items whose ıtest version matches its filter. Second, every device epresents its replication-specific metadata in a compact orm, with state proportional to the number of devices ıther than the number of items. Such compact repre- entation results in low data synchronization overhead, hich permits ad hoc replication between newly encoun- ered devices and frequent replication between estab- shed partners, even over low bandwidth wireless net- orks.

Introduction

Delivering information that is relevant to different eople—or is appropriate for different devices—requires ystem support for a richer notion of data synchroniza- on, one that incorporates personalized content filtering. ı many social and work settings, where bandwidth, stor- ge, and human attention may be at a premium, filtering nables information to spread according to interests and equirements. Personal information needs do not always dhere to the rigid organizational structures imposed by ata providers [3], but rather can often be characterized y flexible query-like predicates over the contents of di- erse data collections.

At the same time, timely and robust information shar- ıg cannot always rely on established Internet connectiv- y or depend on centrally managed storage. Communi- ation between devices may be ad hoc, taking advantage f the proximity of neighboring devices and the avail- bility of particular content. For example, in the wake of lurricane Katrina, disaster workers needed to quickly set p ad hoc networks in which communication and control ere distributed and egalitarian [5].

In this paper, we present *Cimbiosys*, a replicated storage platform designed to support collaboration within loosely-organized communities with applications such as home media management and shared calendars and to facilitate the interplay between mobile devices and cloud-based services. The main contribution of this work is demonstrating how to permit content-based partial replication among peers while providing two important system properties:

- *Eventual filter consistency*: Each device eventually stores precisely those items that would be returned by running its custom filter query against the full data collection.
- *Eventual knowledge singularity*: The state that is transmitted between devices in synchronization requests and is used to identify unknown latest versions converges to a size that is proportional to the number of replicas in the system rather than the number of stored items.

Eventual consistency has long been demanded by applications and provided in replicated systems. Ensuring eventual filter consistency in a system that permits peer-to-peer synchronization between devices with individual, content-based filters is more challenging. Not only may a device's interest in specific items fluctuate over time as the items are updated, but a device may vary its filtering criteria, causing items with stable contents to enter and leave the device's interest set. The next section expands on the substantial challenges of content-based partial replication.

Eventual knowledge singularity is a new property we have defined to convey the importance of compact synchronization-specific state in making economical use of bandwidth and system resources. Essentially, eventual filter consistency is an important correctness property while knowledge singularity is hidden from applications but provides performance and convergence benefits. In particular, this property allows Cimbiosys to use brief intervals of connectivity between peer devices and permits more frequent exchanges between regular synchronization partners, thereby reducing convergence delays. By contrast, conventional synchronization techniques that exchange per-item version vectors or rely on

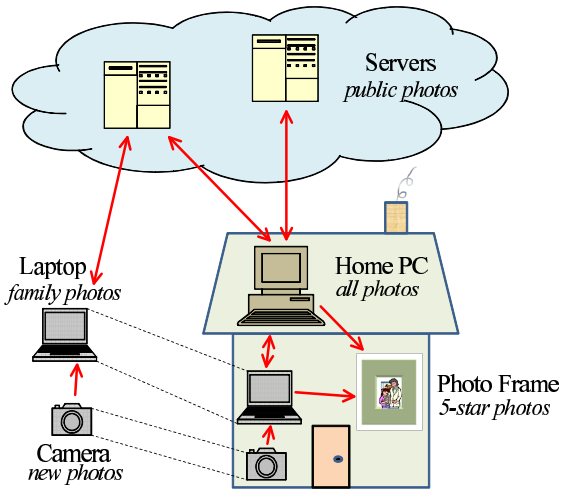

Figure 1: Photo sharing

operation logs make less effective use of relatively slow or intermittent connections. In such systems, 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 tens of thousands of items and items are updated repeatedly.

2 Challenges

To further illustrate the needs of applications that manage partially replicated data, consider the photo sharing scenario depicted in Figure 1. Alice is traveling in Thailand, photoblogging as she goes. Each night, the day's photos are copied from Alice's camera to her laptop. When she reaches a town with an Internet café, she uploads select photos to her Flickr account. After Alice returns from her trip, her photos are synchronized with the master collection on her PC. She spends several weeks working with her new photos on the PC, rating them using one to five stars, adding additional tags, and cropping or retouching photos. Five-star photos are uploaded via a direct WiFi connection to her living room's photo frame. Photos that Alice tags "public" are uploaded to a travel photoset on Flickr and onto a photojournalism web site. A copy of all of her family photos are retained on her laptop, so she'll have them with her when she travels again.

This scenario reveals an implicit set of requirements for a modern storage platform:

- Updates may originate from multiple sites and produce new versions of items that must be selectively disseminated to various devices.
- Interdevice communication may be ad hoc, taking advantage of device proximity and the availability of particular content.
- Not all devices (or even cloud-resident services) store complete collections, and the items of interest vary across devices according to their uses and capabilities.

At first blush, adding content-based filtering to a replication protocol may seem straightforward. Start with any protocol that fully replicates data and guarantees eventual consistency. Whenever a data item is about to be sent via this protocol, check the contents of the item against the destination device's filter. If the item matches, and hence is of interest to the destination, then continue to send the item; if it does not match, then ignore the item. Unfortunately, this simple scheme does not ensure eventual filter consistency.

Content-based filtering for devices with arbitrary communication topologies introduces five key challenges:

- *effective connectivity*: ensuring, in the face of varying device-specific filters, that every item has a path by which it can flow to all interested parties;
- *partial synchronization*: permitting incremental synchronization between peers with overlapping interests without wasting bandwidth on duplicate items or excessive exchanges of metadata;
- *item move-outs*: informing devices of items they store that no longer match their filters due to more recent updates;
- *out-of-filter updates*: determining how to propagate and when to safely discard updated items that do not match the updating device's own filter; and
- *filter changes*: allowing a device to modify its filter without completely discarding previously stored items or failing to receive items that match its new filter.

Unless these issues are explicitly addressed by the replication protocol design, they can prevent eventual filter consistency. We now describe each of these problems in more detail; solutions are presented in later sections.

A synchronization *topology* can be viewed as a graph where devices (or services) are the nodes, and edges indicate synchronization partnerships between pairs. Custom synchronization topologies that permit indirect communication between devices are desirable; Alice's photo frame never directly synchronizes with her camera, for instance. In a fully replicated system, eventual consistency can be achieved as long as the topology graph is connected and devices at least occasionally synchronize with their neighbors. As long as these basic conditions are met, *topology-independent* protocols accomodate arbitrary communication patterns. In a system with partially replicated collections, additional issues arise. For

example, in the scenario in Figure 1, if Alice's home PC never directly synchronized with her laptop, then the only path for routing new, tagged photos from Alice's laptop to her PC would be through services in the cloud, such as Flickr. In this case, the PC would only receive laptop-resident photos that are tagged as "public" and, hence, have been uploaded to a photo-sharing service. Section 7 discusses the topology constraints enforced by Cimbiosys to ensure effective connectivity.

The problem of *partial synchronization* arises when a device synchronizes from a partner that can only supply some of the items that match the device's filter. For example, while Alice is traveling and uploading select photos to Flickr, her home PC may synchronize daily with the service and obtain these new photos. When Alice returns home and her laptop synchronizes directly with her PC, the PC should not assume that it has already received from Flickr any photos taken more than a day ago. In general, a device may receive some items that match its filter from one synchronization partner and other items of interest from other partners. Section 4.2 introduces the notion of *item-set knowledge* to deal with this issue.

An item is said to *move out* of a device's interest set when an update to the item causes it to no longer match the device's filter. For example, suppose that Alice decides that one of her public photos is a bit too revealing, and so she edits the photo on her PC to remove the "public" tag. Using the simple replication approach outlined earlier, this updated photo would not be sent to Flickr when it next synchronizes with Alice's PC. However, the previous version of this photo, the one marked as public, would remain indefinitely on Flickr's web site, contrary to Alice's intentions (and violating eventual filter consistency). Replication protocols that support content-based filtering not only must selectively propagate updated items but also must inform devices of items that should be discarded. Section 5.1 indicates the conditions under which Cimbiosys delivers move-out notifications during synchronization.

The fourth challenge is dealing with *out-of-filter updates*. A device might update an item producing a version that does not match the device's own filter. For example, suppose that Alice is working on her laptop and edits one of her private photos to remove the "family" tag (perhaps a photo of her sister's ex-husband). In this case, Alice's laptop cannot discard the photo immediately, even though it does not match the laptop's filter, since doing so would prevent other devices from learning of this edit; the photo can only be discarded by the laptop after it synchronizes with the home PC and sends it the new version. In some situations, none of a device's regular synchronization partners may be interested in out-of-filter updates that it makes. Section 5.2 addresses this issue.

A final challenge arises from the need to support changing filters. A person's information needs may vary over time, causing her to change some devices' filters. For example, Alice might decide one day that she wants only 5-star photos uploaded to the photojournalism web site rather than all of her public photos. One option is for a device, upon a change to its filter, to discard all of its locally stored items, reset its synchronization state, and essentially restart as a new replica. However, this approach wastes critical resources, such as network bandwidth and energy, and may disrupt the person's work. Section 5.3 details our approach to filter changes.

3 Cimbiosys Platform

Cimbiosys is a platform developed to support a variety of applications that manage data on mobile devices, personal computers, and cloud-based services. It was developed as part of a research project exploring issues in community information management (CIM).

3.1 System model

In the Cimbiosys distributed architecture, 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 shared video library, or a company's customer 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* contains copies of some or all of the items in a given collection. A device can hold any number of replicas of different collections. For simplicity, all of the examples used in this paper involve a single collection and a single replica per device.

Each device sharing a collection maintains its own replica of the items of interest. The set of items included in a 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. Users can set different filters for each device and can change these filters over time.

Each device is allowed to read its locally stored items and update those items at any time, as long as such updates are in accordance with the collection's *access control policy*. Update operations are applied directly to

items in the device's local replica; such operations are not logged or explicitly recorded. Updates produce new *versions* of items that are later sent to other replicas via a device-to-device *synchronization protocol*. Devices generally have regular synchronization partners 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). Collections and their replicas can be discovered by a variety of means, including social networking web sites, e-mail invitations, naming directories, and wireless discovery protocols.

A replica may remain disconnected from the rest of the system for an arbitrary amount of time due to device failures or lack of network connectivity. However, we assume that each device eventually recovers with its persistent storage intact, occasionally communicates with other devices, and correctly executes the synchronization protocol. A device can permanently retire and discard its local replica but must first synchronize with some other device to ensure that updates are not lost.

At any point in time, a replica may hold older versions of items that have been updated elsewhere, and it may not have learned yet of recently created or deleted items. The Cimbiosys synchronization protocol guarantees eventual filter consistency. That is, a replica eventually receives all versions of items that match its filter and have not been overwritten by later versions, and the replica eventually discards items that are updated in such a way that their contents no longer match the replica's filter.

Cimbiosys does not provide other guarantees such as causal consistency or multi-item coherence. In particular, versions may be received by a device in a different order than they were produced. Moreover, a set of versions for items that were updated atomically at one device may be partially received by another device whose filter only matches a subset of the items.

Naturally, because Cimbiosys allows updates to be made at any replica without locking, two (or more) devices may perform concurrent updates to the same item. Such updates result in conflicting versions that are propagated throughout the system using the synchronization protocol. Any device whose filter selects both conflicting versions may detect the conflict and either resolve it automatically or store both versions pending manual resolution. Resolving a conflict produces a new version of the item that supersedes all known conflicting versions. Any existing technique for detecting conflicts, such as per-item versions vectors [16] or concise predecessor vectors [12], could be adopted for use with content-based partial replication. Thus, no further discussion of conflict management appears in this paper.

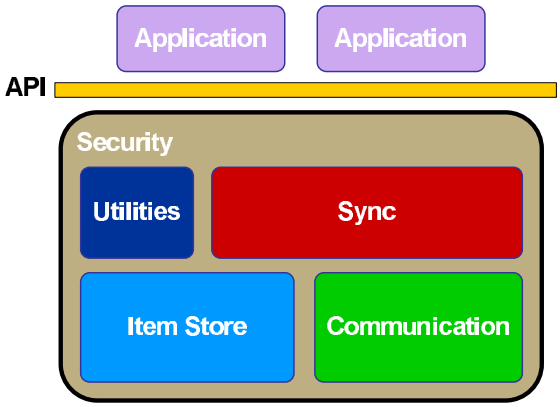

Figure 2: Cimbiosys software architecture

3.2 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 an 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 FeedSync, a set of simple extensions to RSS. Of course, any two devices must agree on the network and transport protocol that they use during synchronization.

The *Sync* module implements the synchronization protocol described in Section 4. During synchronization, it enumerates versions of items in the local Item Store that are unknown to the remote sync partner and sends these along with the appropriate metadata. The remote partner then adds the received items to its Item Store, possibly 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 performing other configuration functions.

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. Versions produced

by a device without write access to the collection (or to the specific items) are rejected during synchronization. A full discussion of the access control design can be found in a companion paper [22]. Additionally, techniques have been developed for recovering from corrupt versions that are introduced through malice or misuse [11].

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 change a replica's filter. Legacy applications that read and write local files, and do not use the Cimbiosys API, are supported by "watcher" processes that monitor file system directories and import files into (or delete items from) a local replica.

3.3 Implementation and validation

Cimbiosys has been implemented in two different environments. One implementation is in C# using Microsoft's .NET Framework running on Windows. We plan to port this code to Windows Mobile 6.0 so it can run on handheld mobile devices. The other implementation is in Mace, a C++ language extension that supports distributed systems development [8]. Both implementations are used in the evaluation presented in Section 8.

Additionally, the synchronization protocol has been fully specified in TLA+ [10]. Extensive model checking has been performed on both the TLA+ specification and the Mace implementation to ensure that the protocol meets the stated design goals, that is, achieves eventual filter consistency and eventual knowledge singularity under a variety of operating conditions.

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 [13].

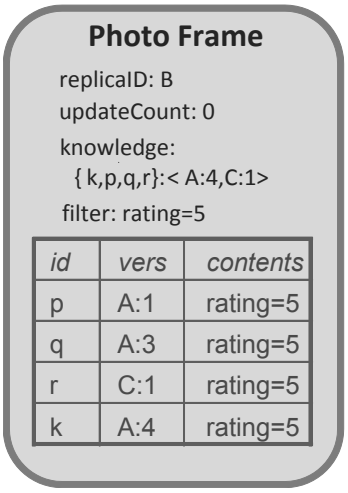

Figure 3: Sample metadata held on the photo frame

4 CIM Sync Basics

The next three sections focus on a key aspect of the Cimbiosys platform, the synchronization protocol. The basic protocol is introduced in this section; Sections 5 and 6 address how the protocol meets the challenges of filter consistency (storing the items that currently match a replica's filter and no other items) and knowledge singularity (operating efficiently by optimizing the metadata that is exchanged during synchronization).

4.1 Metadata

The CIM Sync protocol relies on both per-item and per-replica metadata. Each collection and each item in a collection has a unique identifier, as does each replica of a 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 treated as out-of-filter versions as discussed in Section 5.2 and are eventually discarded by all replicas.

For each item in a replica, the Cimbiosys item store maintains the item's unique identifier, version-id, XML+file contents, deleted bit, and additional information used to detect whether different versions of the item are in conflict (similar to the made-with knowledge used in WinFS [15]). Only the latest known version of each item is retained in the item store. Older versions are considered obsolete.

Figure 3 depicts the data and metadata maintained by a sample replica in our photo sharing scenario. This particular replica, the digital photo frame, is known as replica B. Note that uppercase letters are used through-

out this paper as unique replica identifiers while lowercase letters are used as unique item identifiers. This replica has not performed any local updates, and hence its $updateCount$ is zero. Its filter indicates that it is interested only in photos with a 5-star rating. The replica's item store is shown as a table at the bottom of the figure. It stores four photos: items p, q, r, and k. Every item has a unique version-id. Item p, for instance, has a version-id of A:1, meaning that this version was produced by replica A's first update operation, and has a rating of 5 stars. Each item has additional data and metadata that is not shown in the figure, such as the actual photo contents and the deleted bit. Finally, this replica has knowledge about the items that it stores as described next.

4.2 Item-set knowledge

Each replica maintains *knowledge* recording 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 the 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 [18] that do not support content-based partial replication.

Knowledge is represented as one or more fragments where each fragment is a version vector [16] 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. When a replica's knowledge contains multiple fragments, the replica's overall knowledge is the union of the version-ids from each fragment. Note that, from its knowledge alone, a replica cannot determine whether a known version is stored, obsolete, or out-of-filter.

For example, replica B in Figure 3 has a single knowledge fragment whose item-set is $\{k, p, q, r\}$, the ids of the four items that are stored by this replica, and a version vector of <A:4, C:1>. Replica B, the photo frame, does not appear in the version vector since it never directly updates items and hence does not generate any versions. Replica B's knowledge indicates that the device is aware of any versions of items k, p, q, or r with a version-id of A:1, A:2, A:3, A:4, or C:1. It does not mean, however, that each of these version-ids is for a current or obsolete version of one of these items. To permit a compact knowledge representation, the version vector may include version-ids for items that are not in the associated set; technically, those versions are not known to the replica. For instance, version A:2 could be the latest version of some item u that is not stored by replica B and that may or may not match its filter.

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 in addition to star-knowledge. In a system with a mix of full and partial replicas, any replica may have both star-knowledge and any number of item-set knowledge fragments, at least temporarily. For instance, after synchronizing from a partial replica, a full replica may end up with item-set knowledge reflecting the set of received items.

4.3 Filtered synchronization

Cimbiosys uses a one-way, pull-style synchronization protocol. A replica, called the *target replica*, initiates synchronization with another replica, called the *source replica*. Each device generally plays the role of the target replica for some synchronization sessions and the source replica for others. Two-way synchronization requires a pair of devices to synchronize, switch roles, and then synchronize again.

The target replica starts by sending a SyncRequest message that includes the target's knowledge and its filter. The target is not sent any versions that are already included in its knowledge or that are not of interest. In particular, the source replica 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 in Section 5.1, 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 version-ids as discussed in Section 6.1.

The messages received by the target replica can be applied to its item store individually or as a single atomic transaction. Updating items (and the replica's knowledge) as new versions are received allows progress to be made even when a connection is interrupted before the synchronization protocol completes. The knowledge-driven nature of the protocol makes it resilient to device crashes and lost messages.

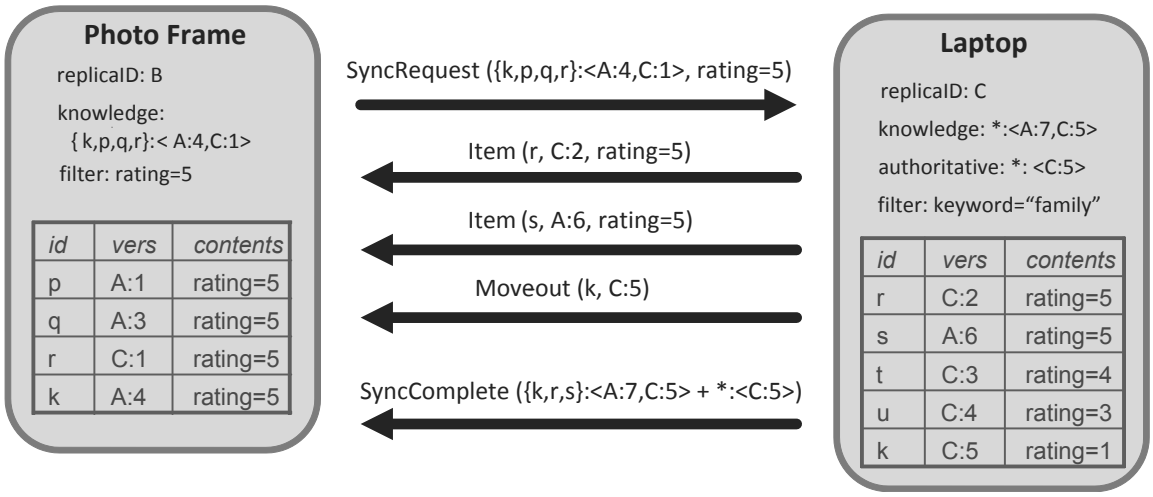

Figure 4: Example synchronization between a target replica, the photo frame, and a source replica, the laptop

Figure 4 illustrates a synchronization session from our scenario in which the digital photo frame (replica B) requests items from the laptop (replica C). 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 specifies that it knows about four items, but has not seen any updates from the laptop since version C:1. The laptop, the source replica in this example, returns a more recent version of item r that it produced and a new item s that had been created at replica A. Item k had also been updated on the laptop to reduce the photos rating; hence the laptop notifies that 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. This learned knowledge consists of two knowledge fragments, separated by a plus sign, which means that the photo frame will end up with three knowledge fragments after processing the SyncComplete message.

The following sections describe in more detail specific protocol features devised to support the requirements of partial replication.

5 Eventual Filter Consistency

Although the use of item-set knowledge in the CIM Sync protocol guarantees that replicas eventually receive all items of interest (assuming sufficient effective connectivity), it does not ensure eventual filter consistency. This section presents additional techniques needed to deal with move-outs, out-of-filter updates, and filter changes.

5.1 Move-out notifications

During synchronization, the target replica may receive *move-out notifications* from the source replica when items have later versions that no longer match its filter. These cause the target to remove specified items from its item store. There are two conditions under which the source returns move-out notifications.

The simplest condition is when the source replica stores an item whose version is not known to the target replica and whose contents do not match the target's filter. The source can send a move-out notification for any such item. This is the condition illustrated in Figure 4 where the laptop sends a move-out notification for item k, whose rating had been reduced.

A target replica may receive move-out notifications for items that it does not store, such as items that are updated and continue to not match the target's filter, a potentially common occurrence. For example, suppose that the laptop in Figure 4 updated item t producing version C:6 in which the rating was unchanged but a new caption was added to this photo. In this case, when the photo frame next synchronizes from the laptop, it would be sent a move-out notification for item t even though it does not store this item and perhaps never did. Such spurious notifications do not affect eventual filter consistency since they 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 lap-

top 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 notification 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. This relies on the source being informed of the set of items that are stored by the target.

5.2 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 might 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 7 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.

5.3 Changing filters

Cimbiosys permits arbitrary filter changes while allowing replicas to retain as many items as possible. 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, if 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 items since it may be the only replica that holds the latest versions. 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 previously out-of-filter versions may now match the new filter. Such versions need to be removed from the replica's knowledge so that the replica 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 retracted to include only versions of items that it already stores. The representation of item-set knowledge makes retraction easy. Knowledge fragments with explicit item-sets retain the same version vector but with a possibly smaller set of items; any star-knowledge fragments are converted 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 retract its knowledge.

Since replicas are allowed to change their filters at any time, a replica may receive out-of-date move-out notifications based on a previous filter. 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 number is included in each synchronization request and is returned in each move-out notification. Move-out notifications that include old filter version numbers are simply ignored by the receiving replica.

6 Eventual Knowledge Singularity

In this section, we propose mechanisms by which replicas acquire and compact their knowledge. Although the number of fragments in a replica's knowledge may temporarily grow after synchronization, the knowledge tends to converge towards a single star-knowledge fragment represented as a single version vector. This section shows how we achieve the desired state of knowledge singularity for both full and partial replicas.

6.1 Acquiring knowledge

As replicas receive items during synchronization, they add the items' version-ids to their knowledge, but require 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; any out-of-filter versions included in the source's knowledge will also be out-of-filter with respect to the target replica. In other cases in which the target has a broader filter or a disjoint filter compared to the source, the source replica must restrict the conveyed learned knowledge to those items that it actually stores. Figure 4 shows an example of disjoint filters; the photo frame's filter is based on the rating attribute and the laptop's filter is based on the value of the photo's keyword (in this case, "family").

6.2 Compacting knowledge

Whenever a replica synchronizes with another replica, it receives new knowledge fragments. 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 (i.e. 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. Table 1 enumerates compaction rules that can be applied to any pair of knowledge fragments.

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, u, and k, producing versions C:3, C:4, and C:5. Suppose that

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

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

Table 1: Knowledge compaction rules

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 4. When Alice's home PC (replica A) synchronizes from her laptop, it will receive these items and the associated learned knowledge. The home PC's knowledge would become something similar to $*{:}\langle A{:}9\rangle + \{k, r, t, u\}{:}\langle A{:}7, C{:}5\rangle$. Unfortunately, this knowledge cannot be compacted. This problem is addressed in the remainder of this section.

6.3 Authoritative versions

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 6.1 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 "*" (i.e. 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 $*{:}\langle C{:}5\rangle$ 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. A replica's star-knowledge grows so that it eventually dominates other knowledge fragments, which can then be discarded using the compaction rules in Table 1.

6.4 Transferring authority

One practical issue remains, namely how to transfer authority when an item is no longer of interest to the authoritative replica, whether due to out-of-filter updates or to 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 or updates 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 a 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 three. 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), the laptop 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 other replicas, 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). As long as each replica regularly synchronizes with a set of partners that collectively know about all versions in the system, each replica will converge towards singular knowledge. Clearly, a device that synchronizes directly with every other device will receive a complete set of star-knowledge. The following section describes how Cimbiosys ensures that replicas are configured in a suitable topology without requiring full interconnectivity.

7 Filter-based Tree Topologies

The CIM Sync protocol can be used by 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 partner that match its filter, and it will receive whatever move-out notifications can be generated by the partner. Moreover, a replica never receives the same version from multiple synchronization partners (unless it engages in parallel synchronizations or changes its filter). But additional constraints must be placed on the synchronization topology in order to achieve eventual filter consistency and 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 a 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, but may also synchronize with other replicas.

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 is unreachable for an extended period of time.

The tree synchronization topology provides four important benefits.

One, the synchronization topology ensures effective connectivity. That is, groups of replicas for the same collection cannot remain disconnected indefinitely, assuming periodic synchronization between parents and children. Moreover, 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.

Two, 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 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.

Three, 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.

Four, 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 their descendants. Eventually, all authoritative versions arrive at the reference replica, which 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. In the photo sharing scenario presented in Section 2, 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. Cloud-based services might replicate data among themselves for geographic scaling, unbeknownst to the reference replica. 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.

8 Evaluation

In this section, we present an evaluation of Cimbiosys based on our two implementations, one in C# for Windows platforms and one in Mace for Linux platforms. In particular, we answer the following questions with respect to the goals of Cimbiosys:

- Does Cimbiosys achieve eventual filter consistency in the presence of move-outs, out-of-filter updates, and changing filters?
- How concise is the knowledge representation in Cimbiosys as compared to protocols with per-item knowledge, and does the reduction in knowledge size lead to more efficient synchronizations?
- What are the benefits of leveraging filter relationships between replicas, and how do non-hierarchical synchronizations affect the performance of Cimbiosys?

8.1 Experiments on the C# implementation

We performed experiments on the C# implementation by running 10 replicas on the same computer. The replicas formed a three-level hierarchy based on filter relationships with one full replica at the top, three partial replicas in the middle, and six more partial replicas at the bottom. Each replica's filter was less restrictive than the filters of any replica at a lower level.

The experimental workload had five serial phases consisting of different kinds of updates to the system. Each update consisted of a randomly chosen replica modifying the content of a randomly chosen item in its item store. Throughout the experiment, replicas synchronized with randomly chosen partners at regular intervals.

1. *insert phase*: Randomly chosen replicas inserted a total of 1000 items into their respective item stores at the start of the experiment. 600 synchronizations followed the inserts.
2. *update phase*: 1000 updates were performed, none of which triggered move-outs at any replicas. There

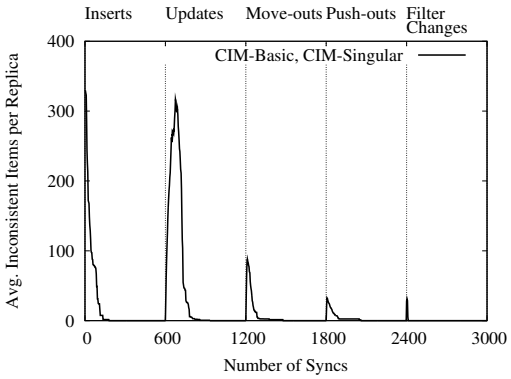

Figure 5: Average inconsistent items per replica vs. time

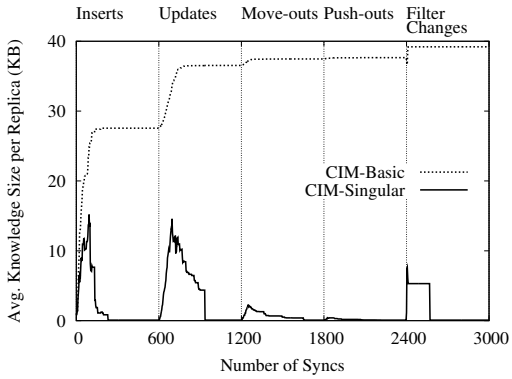

Figure 6: Average size of knowledge per replica vs. time

were 600 synchronizations in this phase, and the updates happened during the start of the phase at the rate of 10 updates between each synchronization.

3. *move-out phase*: Replicas updated 100 items; the updated content continued to match the updater's filter even though it might move out of other replicas' filters. 600 synchronizations followed.

4. *push-out phase*: Replicas performed a total of 50 out-of-filter updates. That is, the updated content did not match the updating replica's filter. Another 600 synchronizations followed.

5. *filter-change phase*: Three randomly chosen partial replicas changed their filters to new non-overlapping filters. A final 300 synchronizations ended the experiment.

We evaluated two variants of the Cimbiosys system. The first variant, called *CIM-Basic*, implemented all the core mechanisms described in Section 5 for achieving eventual filter consistency. The second variant, called *CIM-Singular*, implemented the additional mechanisms for the accumulation of authoritative knowledge in order to achieve eventual knowledge singularity as presented in Section 6.

Results

We first show the progress made by replicas in achieving eventual filter consistency. Figure 5 plots the average number of inconsistencies in a replica's item store over time. Here, an inconsistency at a replica R at a certain time includes three cases: a) an item present in R's store is obsolete, b) the latest version of an item matches R's filter but no version of the item is present in R's store, and c) an item is present in R's store but does not match R's filter. We counted these inconsistencies by tracking the global state of the system.

Figure 5 confirms that both CIM-Basic and CIM-Singular eventually achieve a state of zero inconsistencies in the presence of partial synchronization, move-outs, out-of-filter updates, and filter changes. They also converge at the same rate (and the graphs are identical) because they share the same core mechanisms to support partial replication.

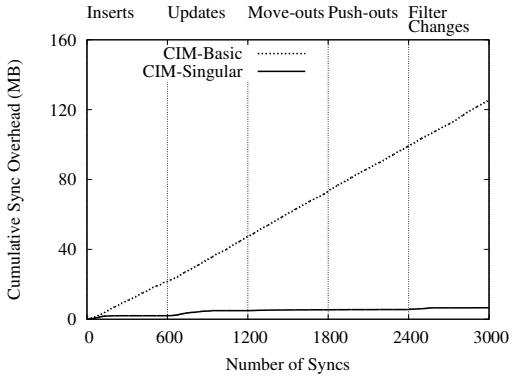

Figure 7: Cumulative synchronization overhead incurred vs. time

We next evaluate knowledge compaction in Cimbiosys. Figure 6 shows the average size of the knowledge of each replica over time. As expected, the size of knowledge in CIM-Basic increases as updates are performed and reaches a peak value dependent on the number of items stored in the replica and the number of updates performed to each item. In CIM-Singular, however, knowledge is fragmented in the initial stages but eventually converges to the size of a single version vector at the end of each phase. In other words, CIM-Singular achieves eventual knowledge singularity.

Figure 7 demonstrates the positive effect that knowledge compaction has on synchronization overhead. It shows the cumulative overhead incurred during synchronizations in the insert and the update phases. The overhead includes the cost of transmitting knowledge from the target to the source in the initial SyncRequest message and from the source to the target in the final SyncComplete message.

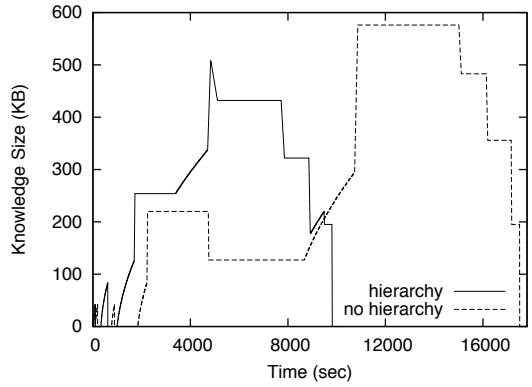

Figure 8: Effect of leveraging filter relationships

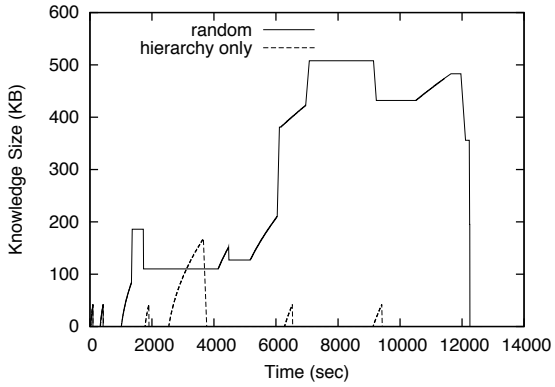

Figure 9: Effects of out-of-hierarchy synchronization

Knowledge compaction provides a significant reduction in the sync overhead over a period of time as evident from the difference between CIM-Basic and CIM-Singular in the figure. Low synchronization overhead means that replicas can synchronize more often and learn updates sooner with the same bandwidth budget. It also enables effective synchronization for replicas on bandwidth-constrained mobile devices.

8.2 Experiments on the Mace implementation

We evaluated the Mace implementation of Cimbiosys using ModelNet [21] to simulate a variety of network topologies on a cluster of machines.

For these experiments, we used a system of 10 replicas, a binary-tree filter hierarchy, and a collection size of 10,000 items, which reflects the average size of a consumer photo collection. Using ModelNet, we emulated a clique of 10 routers, each connected to a single replica. The link speed between all routers and replicas was set to 100 Mbps. The trends in the experimental results were similar with lower bandwidths.

Each experiment consisted of two phases. During phase 1, replicas created items such that 10,000 total items existed in the system at the conclusion of this phase. During phase 2, synchronizations proceeded until the knowledge at all replicas converged to a stable state.

Results

The general trends in the size of knowledge and the sync overhead for the MACE experiments were similar to the results of the C# experiments discussed earlier, and so we do not present them here. Instead, we focus on evaluating the impacts of filter relationships and synchronization patterns.

We first discuss the effects of leveraging the hierarchical filter relationships overlaid upon the network topology. We performed experiments where each replica chose a parent or a child as its synchronization partner 50% of the time and an arbitrary replica at other times. In the first experiment, called *hierarchy*, replicas would synchronize as parents or children when their filters were in the proper relation according to the filter hierarchy. In the second experiment, called *no hierarchy*, every synchronization was treated as if the filters were unrelated.

Figure 8 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 after synchronizing with a parent that all versions included in the parent's knowledge should be included in their own. Similarly, replicas can become authoritative for versions authored by their descendants, and this information can flow up the hierarchy until it reaches a reference replica, at which point it flows downward in a compact form. Without a hierarchy, replicas can only claim authority over versions they themselves store. We can still achieve eventual knowledge singularity without a filter hierarchy but it takes longer for replicas to reach that state.

Finally, we discuss how the choice of synchronization partners (only parent or children versus arbitrary replicas) affects the performance of Cimbiosys. Figure 9 compares an experiment in which replicas only synchronized with their parents and children with an experiment in which the replicas selected synchronization peers at random. As the figure shows, 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. On the other hand, synchronizations with arbitrary peers may allow quicker exchange of updated items between replicas at the cost of increased fragmentation in knowledge.

9 Related Work

The Cimbiosys design presented in this paper builds upon previous work on content-based filtering and especially weak-consistency replication protocols. In this section, we discuss related work with an eye toward how the systems fall short of meeting the challenges intro-

System	Selection criteria	Partial sync	Effective connectivity	Move-outs	Out-of-filter updates	Filter changes
Cimbiosys	Content-based filters	Item-set knowledge	Filter-constrained embedded tree topology	Explicit move-out notifications	Push-out store	Knowledge retraction and push-out store
Ficus	File IDs	Metadata exchange	Per-file ring topology	Cannot occur	Cannot occur	Not addressed
PRACTI	File IDs / directories	Log exchange	Policy	Not addressed	Not addressed	Not addressed
EnsemBlue	File IDs + persistent queries	Client-server	Client-server	Not addressed	Write back to server	Not addressed
Perspective	Views, i.e. attribute-based filters	Log or metadata exchange	Not addressed	Logged pre and post versions	Retain until pulled by device	Not addressed

Table 2: Key design decisions in Cimbiosys and related work.

duced by content-based replication with a peer-to-peer synchronization model, particularly in an environment characterized by changing content, user interests, and device connectivity.

The HomeViews system has the similiar goal of supporting selective data sharing in a peer-to-peer system model [6]. It allows users to export their data, including digital photos and other files, as views defined by content-based queries written in SQL. Although views are essentially equivalent to filters in Cimbiosys, they are defined by the data exporter rather than by the devices that import the data. Moreover, data is not replicated among devices but rather views are accessed remotely and searched via distributed queries.

The filters supported in Cimbiosys also resemble those of content-based publish/subscribe systems, though such systems offer a completely different replication model [1, 4]. Subscribers in a pub/sub system advertise their filters to a collection of brokers, which build routing tables used to route events from a publisher to the set of interested subscribers. Each event is independent and stored temporarily in the brokers' message queues. New subscribers (or those with new filters) observe only future events. In Cimbiosys, on the other hand, replicas eventually and persistently store all items that match their filters, can update items, and disseminate new and updated items among themselves through direct communication.

Some systems support partial replication but with a client-server model. Coda, for instance, allows clients to cache some or all of the files residing on a server, thereby supporting disconnected operation on mobile devices [9]. A hoard profile, which could be considered a type of filter, specifies the files of interest to each client, though Coda clients may cache other files based on access patterns. Clients reconcile their local changes directly with the server(s). BlueFS [14] provides a similar system model but emphasizes energy efficiency when dealing with small, mobile devices. As opposed to Cimbiosys, neither Coda nor BlueFS permits clients to share updates directly with each other.

EnsemBlue [17] extends BlueFS by allowing disconnected clients to organize into a temporary ensemble headed by a client acting in place of the server. Notably, EnsemBlue supports persistent queries that can be used by clients, along with server-provided callbacks for cache invalidation, to provide a form of content-based replication. Select operations on files that match a persistent query are logged by the server in a special file that can be retrieved and read by clients. A client then explicitly fetches new files that match its query and discards updated files that no longer match the query. Unlike Cimbiosys, the burden is placed on servers to record which files are cached where and on clients to fetch updated files in order to determine whether the contents are of interest.

Some topology-independent replication systems allow arbitrary communication patterns but lack support for content-based filters. Bayou, for instance, includes an efficient log-based, peer-to-peer synchronization protocol but assumes that all replicas are interested in all items [18]. WinFS, like Bayou, maintains a single version vector per replica that is transmitted on every synchronization, but uses state-exchange rather than log-exchange [15]. WinFS supports replication of arbitrary file folders but not per-replica filters. Cimbiosys extends the WinFS design to support content-based filtering while ensuring eventual filter consistency; the eventual knowledge singularity property ensures that the per-replica overhead converges to a single version vector as in Bayou and WinFS.

A few other systems have combined topology independence with some form of partial replication. One early peer-to-peer replication system, Ficus [7], was extended to support selective replication [19]. Each replica can store an arbitrary subset of a file system volume and can alter the set of locally stored files at any time. Because the set of interesting files is explicitly specified by file ids, and not based on file contents, several of the key concerns with content-based filtering do not arise in Ficus, including out-of-filter updates and move-outs. Syn-

chronization is a heavy weight operation since a replica must pull information about all of the files stored on a remote replica in order to determine those that have been updated or newly created. To reduce communication costs and ensure effective connectivity, the sites replicating a given file are organized into a ring where synchronizations occur between neighbors in the ring, essentially renouncing topology-independence.

PRACTI is another replication system with topology-independence and partial replication (and arbitrary consistency) [2]. In PRACTI, each replica maintains a log of invalidations for objects that have been updated. A synchronization protocol similar to Bayou's exchanges log entries between pairs of replicas. Partial replication is achieved by allowing replicas to selectively fetch invalidated objects. Imprecise invalidations that cover a range of objects let partial replicas maintain smaller logs. While PRACTI permits each replica to define its own "interest set", the current design equates interest sets with file folders, and issues such as effective connectivity are left as policy decisions. Adding practical support for content-based filtering to PRACTI would require many of the techniques developed in Cimbiosys.

More recently, the Perspective project at CMU has been exploring a replication paradigm most closely resembling that of Cimbiosys, but with a very different system design [20]. Each device in Perspective defines an attribute-based filter called a "view". Only files included in a device's view are stored on the device. Unlike Cimbiosys, each device is aware of all other devices and their views; hence, Perspective is more suitable for a small, fixed set of devices, such as those in a consumer's home media system. Upon updating a file, a device sends a notification to all other available devices. Devices, in turn, fetch the updated files on demand. A disconnected device that misses update notifications is later brought up-to-date by synchronizing directly with other devices. A device can modify its view at any time, but it must inform the other devices and behave as a new replica during synchronization to obtain the files that match its new view. Cimbiosys, by contrast, allows content-based filters, bandwidth-efficient synchronization, incremental filter changes, incomplete knowledge of other replicas, and arbitrary synchronization partners.

Table 2 summarizes the key design decisions in previous partial replication systems as well as Cimbiosys. It focuses on the steps taken by the designers of these systems to address the five key challenges of content-based partial replication presented in Section 2.

10 Conclusion

Cimbiosys is a new storage platform that provides filtered replication of content through peer-to-peer synchronization. Its design was motivated by the needs of loosely-organized communities and of individuals managing multiple devices. Cimbiosys allows each device to express its individual information needs as a content-based filter, permits devices to enter or leave the system without global coordination, accommodates dynamically changing content and filters, efficiently propagates updated items while avoiding duplicate delivery, exploits opportunistic encounters between devices with overlapping filters, and supports flexible synchronization topologies (within certain constraints).

Eventual filter consistency, whereby a device's replica converges towards a state containing exactly those items that match its filter and nothing more, is achieved through a combination of novel technologies and pragmatic design decisions. Item-set knowledge, compactly represented as one or more version vectors and associated items, records not only the versions that have been received by a device but also obsolete versions and versions of items that no longer match its filter. Given a device's knowledge and filter, the synchronization protocol can readily determine exactly those versions of interest, thus meeting the challenge of partial synchronization. Under specific conditions, devices receive move-out notifications during synchronization and can discard out-of-filter versions without losing updates. When modifying its filter, a device can adjust its knowledge so that its local item store is incrementally updated to match its new filter.

Remarkably, knowledge converges towards a single version vector for all devices, with full or partially replicated contents. This eventual knowledge singularity property is achieved by ensuring that at least one device is authoritative for every version ever generated, transmitting star-knowledge for authoritative versions during synchronization, and compacting knowledge fragments. Our experimental evaluation, which was based on implementations of our protocol, as well as model checking performed on a formal specification, demonstrate that eventual knowledge singularity is indeed realized if updates cease for a sufficiently long period. In a system with frequent updates and filter changes, devices may never actually reach knowledge singularity, but the techniques used to drive the system in that direction serve to keep knowledge to a manageable size.

Using the CIM Sync protocol, eventual filter consistency and knowledge singularity will be attained in systems where every device synchronizes occasionally with every other device. However, requiring full inter-device connectivity is unrealistic in many of the scenarios that we wish to support. By enforcing a hierarchically filtered tree topology, Cimbiosys maintains the desired properties while providing some degree of flexibility in establishing synchronization partnerships and still allowing ad hoc communication between peers.

Acknowledgements

We thank Rama Kotla, who made considerable suggestions for improving this paper, Daniel Peek for many early design discussions, Mike Dahlin for aiding our understanding of PRACTI, Brandon Salmon for answering our questions about Perspective, Robbert van Renesse for his insightful comments on an earlier version, and our shepherd, Steve Gribble, for his advice and guidance in producing the final version of this paper. Chip Killian, James W. Anderson, and Ryan Braud provided much assistance with Mace and ModelNet. Lev Novik, Mike Clark, and Moe Khosravy were a valuable source of information on the Microsoft Sync Framework and contributed product code to our C# implementation.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–61, May 1999.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 59–72, May 2006.

[3] P. Dourish, W. K. Edwards, A. Lamarca, and M. Salisbury. Presto: An experimental architecture for fluid interactive document spaces. *ACM Transactions on Computer-Human Interaction*, 6(2):133–161, June 1999.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[5] S. Farnham, E. Pedersen, and R. Kirkpatrick. Observation of Katrina/Rita Groove deployment: Addressing social and communication challenges of ephemeral groups. In *Proceedings of the 3rd International ISCRAM Conference*, pages 39–49, May 2006.

[6] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. HomeViews: Peer-to-peer middleware for personal data sharing applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–246, June 2007.

[7] 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 USENIX Conference*, pages 63–71, June 1990.

[8] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 179–188, June 2007.

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

[10] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[11] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proceedings of the EuroSys 2009 Conference*, Apr. 2009.

[12] D. Malkhi and D. B. Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, Oct. 2007.

[13] 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)*, pages 251–260, June 2008.

[14] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the Blue file system. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 363–378, Dec. 2004.

[15] 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.

[16] 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, May 1983.

[17] 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)*, pages 219–232, Nov. 2006.

[18] 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)*, pages 288–301, Oct. 1997.

[19] 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)*, pages 169–181, Dec. 1999.

[20] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009.

[21] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becke. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 271–284, Dec. 2002.

[22] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Access control for peer-to-peer replication. Technical Report MSR-TR-2009-15, Microsoft Research, Feb. 2009.