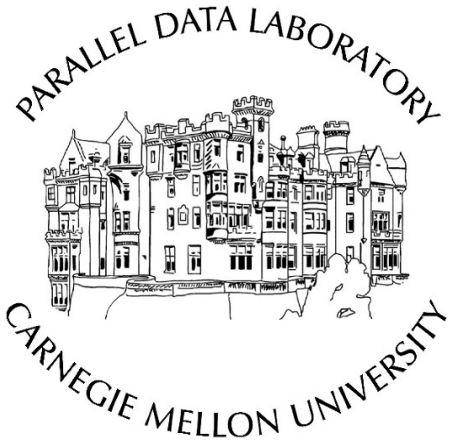
5-2006

Towards Efficient Semantic Object Storage for the Home (CMU-PDL-06-103)

Brandon Salmon
Carnegie Mellon University

Steven W. Schlosser
Carnegie Mellon University

Gregory R. Ganger
Carnegie Mellon University



Towards efficient semantic object storage for the home

Brandon Salmon
Steven W. Schlosser[1], Gregory R. Ganger

CMU-PDL-06-103
May 2006

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The home provides a new and challenging environment for data management. Devices in the home are extremely heterogeneous in terms of computational capability, capacity, and usage model. Yet, ideally, information would be shared easily across them. Current volume-based filesystems do not provide the flexibility to allow these specialized devices to keep an up-to-date copy of the information they require without seeing large amounts of traffic to other, unrelated pieces of information. We propose the use of views to allow devices to subscribe to particular classes of objects. Data views allow devices to see up-to-date information from available devices, and eventual consistency with unavailable devices, for objects of interest without seeing updates to other objects in the system. Views also provide a basis on which to build reliability, data management and search.

[1]Intel Research Pittsburgh

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Symantec) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453. Brandon Salmon is supported in part by an NSF Fellowship.

Keywords: home storage, views, optimistic concurrency, perspective, storage management, object storage

1 Background

Recent years have seen an explosion in the number and capability of digital devices in the home. Digital music players, DVDs, cameras and digital video recorders have started producing much of the media in the home digitally. As new functionality is created, the number of devices found in a home has increased, and many more are expected over the next few years.

This increase in devices naturally leads to an increase in the headache of device management. While many of these new devices are specialized and simplified to make them easier to use, coordinating a large number of devices is almost always difficult. Users are unlikely to adopt useful new consumer devices if managing them is painful or time consuming. Management difficulties are exacerbated by the burgeoning amount of digital data in the home, much of which is both frequently used and valuable. Distributing and protecting this data in an automatic and intuitive way will be critical to the adoption of innovative devices, and the continued movement from analog to digital formats.

1.1 Usage

As the number of devices in the home grows, so does the need for a home data infrastructure to connect them. This infrastructure is not defined by the physical wireless or Ethernet found in the home, but rather by the collection of devices used by the residents. These devices will travel to different locations, and will be accessible by different sets of other devices. The "home system" should exist on an airplane, where two children are both using laptops or music players, as well as in the home.

Data replicated on these devices should be accessible from any other "visible" devices (i.e. those with which a given device can communicate) even if the home is unavailable or only available over a low bandwidth link. Even in high bandwidth conditions, using a nearby replica will often be preferable to improve performance or reduce power consumption.

In addition, users of the system must be able to easily specify which information belongs on which devices, and easily see where that information resides. The user should be able to set high-level policies that are then enforced by the system. Some devices may even be able to determine user preferences by observing previous usage and suggest policies automatically.

1.2 How is the home space unique?

There are many challenges in addressing these problems in the home environment. First, most home users have little or no computer experience and limited patience with set-up processes or maintenance tasks. In order to be useful to home users, devices must be self-configuring and require minimal or no input from the user to work effectively. The devices may even need to anticipate demands in order to avoid explicit user requests. Second, the home is an environment that includes an extremely heterogeneous set of devices. It includes resource-rich devices like desktop machines, power-limited devices like portable music players, and resource-scarce devices like cell phones. Many of these characteristics are also dynamic. Many devices, such as car accessories or portable music players, will frequently leave the home. Many power-constrained devices will alternate between power-scarce disconnected periods and power-rich connected periods. Any infrastructure will need to utilize the resources provided by these devices at any given time, and anticipate their availability in the future.

The home environment does, however, provide new resources to solve these challenges. For example much of the data in the home, such as music and television, is produced by an outside creator and thus is often tagged with rich metadata about the content. The home environment is also populated with a small number of users, each of whom will often have fairly static usage patterns.

1.3 A semantic store

Semantic file (or storage) systems assume that objects within the system are tagged with extensible metadata, and allow users to search for content based on this metadata. The access patterns for most "home data" fit well into a semantic mold. Music, movies and photos are all put into simple databases and queried based on metadata. OS makers are busy extending current filesystems so that they also operate in such a fashion (i.e. Apple's Spotlight, Microsoft's WinFS, and Google's Google Desktop.) We view this model of extensible metadata as a given for data in the future, especially data in the home. This argues for a semantic system in the home. Additionally, many of the devices in the home will be specialized to a certain set of usage patterns — some devices only play music, others only movies; some devices are single user, others are multi-user.

This specialization of devices makes it difficult to divide data using the traditional volume approach which only allows an object to be classified into one volume and requires a device to subscribe to an entire volume at a time. A single classification is insufficient in an environment like the home, where users want to subscribe to specific classes of information from specific devices, which may not coincide with a single classification. The problem is exacerbated by the fact that many of the devices in the home are resource-limited and cannot be expected to track and receive updates for irrelevant objects. The volume approach also forces the user to make the difficult decision of classifying data into the correct volume. Instead, we propose using queries on semantic information as a foundation for finding objects and ensuring their consistent replication.

The semantic model is even more appealing as the namespace becomes a single image across a large number of machines. Storing data in separate trees on each machine is difficult for users, while a single, merged filesystem tree could lead to difficult false sharing and comprehension problems.

1.4 Design goals

In short, the home environment presents several unique challenges and properties that require a new data architecture.

- Home devices have widely varied data needs and capabilities, so our system must allow for flexible data placement, and only require devices to see updates for specific data.
- Home devices are also varied in their usage, so we must allow devices to support their own caching and data management policies.
- Home devices are portable, so our system must allow for flexible communication patterns, handle partitions gracefully, and function without a central server.
- Home data is metadata rich, so our system should allow for flexible metadata, and utilize this metadata when useful.
- Home users use search as a common tool, so our system must efficiently handle data search.
- Home users are usually non-technical, so our system must make it easy for a user to understand the state of the system if they require this information.
- Home data is valuable and long lived, so our system must facilitate data reliability, by utilizing both redundancy across in-home devices, and external backup sources.

In Section 2 we discuss the fundamental points of our design, and then show how they achieve these design goals. In Section 2.4 we discuss data consistency in our system. Finally, we present related work and conclude.

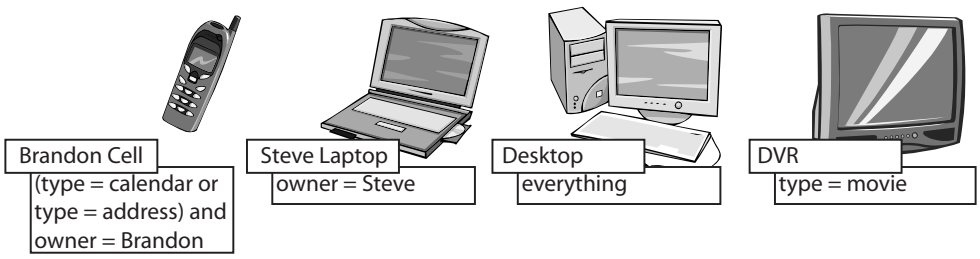


Figure 1: **Views.** This diagram shows an example set of devices and views. In this diagram, we have four devices, each with a customized view. Brandon's cell phone is only interested in "calendar" and "address" objects belonging to Brandon, Steve's laptop is interested in all objects owned by Steve, the house desktop is interested in all data, and the DVR is interested in all "movie" objects.

2 Design

We promote an approach based on providing efficient eventually-consistent copies of system-wide queries in a decentralized environment, while also keeping data fresh with all currently accessible devices. The design allows devices to specialize the information in which they are interested and ignore any data not fitting the description. It also provides a transparent way for users to understand the system state.

2.1 Views

The system is built on the concept of a *view*. A view is a cross between a semantic directory and a publish/subscribe registration. A view describes the objects in which a device is interested as a query on the metadata attached to them. Figure 1 illustrates several devices and their views.

Each device publishes its views into the system, where each view is replicated on each device. Because the number of devices in the home is moderate and these views are compact, this is not a problem in terms of space. Since these views should not change frequently, it should not require many messages, either.

For example, a cell phone might register a view on all objects of type "calendar" or "address book" that are owned by the cell phone's user. In contrast, a digital video recorder (DVR) might register a view for all objects of compatible movie types, and a laptop might register a view for all objects belonging to the primary laptop user.

Once a view is published to the devices in the system, all devices know objects in which the corresponding device is interested. When any device adds or updates an object, it will check its local copy of the views in the system to see which views this change affects. The device will then send a message with the object metadata and update type to each device with a view containing the object. Figure2 illustrates this process. Resource limited devices are also free to pass an update on to other more powerful devices for forwarding throughout the system.

An update notification does not actually contain the object data. When a device receives an update notification, it is free to decide whether and when to actually fetch the object data. This allows each device to have up-to-date knowledge of all of the objects in its published views, without requiring them to store any specific data. This gives each device the ability to use its own caching and eviction policies.

Devices are not allowed to store "home system" objects that do not fall into one of the views that they have registered. Any object not falling into one of the device's views will not be kept up-to-date or be visible to other devices in the system. However, devices can keep their own data outside of the system or make temporary copies of objects as long as they do not require them to be up-to-date.

Devices could also publish messages about their caching strategy; (e.g. when they accept an item into cache, or drop an item from cache). These messages are not critical to maintaining consistency or

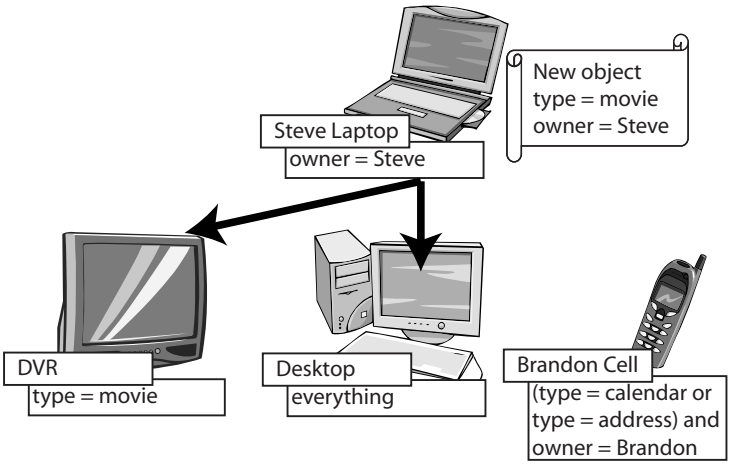


Figure 2: **Object modification.** This diagram shows how views facilitate object update or addition. In this case, Steve's laptop has received a new movie (perhaps from the Internet). Because it holds all the views in the system, it knows that both the desktop and DVR need to see update messages for this new object. Note that the cell phone's view does not include the object and, therefore, does not need to see the update.

availability, but without them devices may not be aware of whether given data items are actually currently available.

2.1.1 Complete views

Some views not only specify that a given device is interested in the given data items, but states that the device will always store objects matching the view. Effectively, the device guarantees that it will never drop updates to these objects. These kind of views are called *complete views*. A normal, partial view simply informs the system that a device must see all updates to a given set of objects, while a complete view also informs the system that a device will store the given objects, and not drop any of them. In essence, partial views describe temporary cached replicas that may disappear from the system, while complete views describe permanent replicas. Complete views are essential for guaranteeing reliability of data, and can be used to improve the efficiency of search and update.

2.2 Administrative tool

A stateless administrative tool runs occasionally to check the state of the home system. By contacting any other device in the system, the administrative tool can know all of the views in the system and consider options to provide various properties. For example, this administrative tool can help provide reliability of data, and assure that data flows through the entire system. While it is essential that only one instance run in the system at a time, it possible for the tool to be run on any device in the system. This means it could be something like a java applet that could run on any system device.

2.3 Features

Views provide an efficient solution to the design goals listed in Section 1.4, making them a good fit for the unique requirements of the home space.

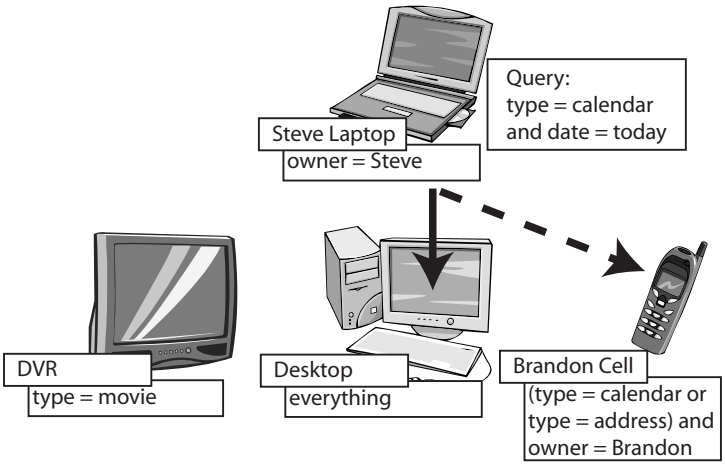


Figure 3: **Search.** Views also allow for efficient search. In this example, Steve's laptop performs a query for calendar entries for today. Because the laptop has a copy of all views in the system, it knows which devices could hold an object of that description. In this case, this would be both the desktop and the cell phone. However, if the desktop provided a complete view, then the laptop would only need to query the desktop, and not the cell phone.

2.3.1 Loosely connected devices

One key advantage to the use of views is that they allow resource limited devices to keep an up to date view of the data that is important to them, without having to see updates to data that does not interest them, without requiring devices to track extensive metadata about other devices in the system. Devices can implement their own caching policies, since data movement is initiated by the device at its leisure.

2.3.2 Flexible communication topology

Another advantage of views is that they allow for the communication topology to adapt to current conditions. In the home, desktop machines will probably subscribe to all information, and provide a complete view, since they have ample resources and are frequently running. This means that the communication patterns could easily follow a server/client approach. However, when devices are in a different environment, where there is no powerful central device, they can share information between themselves. Views also allow devices to function normally in partitions, and do not require a central server.

2.3.3 Search

Views also help make search more efficient. Given a query and the set of views in the system, a querying device can determine whether a given target device could store objects that could fall within the given query, by comparing the query to the views on the target device. The querying device only needs to forward the query on to target devices that have overlapping views. Complete views can be used to reduce the number of devices that must be queried. For example, if a target device has a complete view that covers the query, the querying device only needs to forward the query to that target device, if it is available. A querying device could also choose to query some number of target devices initially, and then only query other target devices if the user does not find the information she needs. Views can similarly guide content searches performed by the target devices, like those used in Diamond [6].

2.3.4 Data reliability

Reliability can be improved by the use of complete views. If an object falls within a certain number of complete views, in stable state it will have at least that many replicas. So, for example, the data of various devices could be stored redundantly by having more resourceful devices (e.g., a PC or storage server) maintain a complete view on their data. Adding off-line backup services of various types is also simple in this kind of system. A backup utility can simply subscribe to all data it must backup and perform whatever method of backup is required. Off-site backup could be easily added by having a device proxy for the off-site backup method. Views provide an easy way for users to define what type of data is protected by each backup service.

Reliability can be ensured by the administrative tool. Given a system goal for number of replicas for given objects, the tool can check to see if enough complete views cover the relevant objects. If the system does not meet the reliability goals, the tool can then recommend new views to ensure that data is correctly protected, given user preferences and input. With a little additional information, such as disk space availability, it should also be able to recommend upgrades if necessary to assure reliability of the system. As long as the administrative tool's decisions take into account the time between runs of the tool, this should provide adequate reliability.

To keep a user from deleting complete views that are needed to maintain reliability, devices should check with the administrative tool before removing complete views. The device could allow the user to delete the view, then simply queue the operation until after talking with the administrative tool. Alternately, the system could require the user to delete the view from the tool itself.

Alone, maintaining enough complete views does not guarantee reliability. It is also important to ensure that replicas containing updates are not prematurely deleted, and that updates are propagated to complete views in a timely fashion.

2.3.5 User interface

Views facilitate understanding the layout of the system for the advanced users who wish to do so. A user can see where data resides in the system by knowing the views currently in the system. The user can also know which views are complete and thus contain all objects in the view, and which views are not. The system should provide this information to the user in a human readable format and allow the user to make changes. This would allow a user to put data on a specific device in preparation for a trip, for example.

Views may also be an effective tool for users to specify system goals. A user could specify that all photos from their digital camera and email should be backed up remotely, for example. By allowing users to specify these requirements in an intuitive format, the system can make it easier for users to get the results they desire and to understand what the system is doing for them. We believe that part of the advantage of views is their utility both in providing important system properties like consistency, and in providing an easily understandable interface to the user.

2.4 Data consistency and freshness

Views provide a way to organize and locate data, but they do not provide data consistency or freshness on their own. We have designed methods to provide data consistency, freshness and convergence in the context of views and loosely connected devices.

2.4.1 Data consistency

Because devices can be disconnected from one another, it is important to have the protocols provide for an appropriate form of consistency. Because concurrent updates are expected to be infrequent and the system

Replica ID	Version Number
1	3
12	5
15	2

Table 1: **Version vector.** This table shows an example of a version vector. In this example replica 1 has been updated 3 times, while replica 15 has been updated twice. Note that replicas other than 1, 12 and 15 may exist in the system, if they have not been modified.

must support disconnected operation, a system using optimistic concurrency control is most fitting. Our focus is on providing eventual consistency and allowing users to detect conflicts in the system. We use techniques similar to Ficus [5], which had similar goals.

Version vectors: Each object has some number of replicas throughout the system, and keep a version vector for each object in the system, with a version number for each of the object's replicas in the system [9]. When a device edits an object replica, it increments the counter associated with that replica. Table1 shows an example version vector.

Given these version vectors, the system can tell if one version has been superseded by another, or if there is a conflict between them. If all of the entries in one version vector *a* are greater than the entries vector *b*, then *a* is more recent than *b*. If an entry is larger in *a* than *b*, and an entry is larger in *b* than *a* then the two versions conflict.

Version vectors create an ordering over all updates to an object, even when updates are done in separate partitions. If two updates are done concurrently in separate partitions, the version vector will detect that they conflict.

We store the version number for each replica as a metadata tag. To keep replica IDs unique, each device constructs a replica ID by prepending the device ID to a counter which it increments each time it creates a new object replica. Note that a replica ID is not equivalent to a device ID; if a device drops an object replica and then adds it again, it will assign the new object replica a new replica ID. We can decrease the storage consumed by version vectors by simply omitting version numbers of zero, meaning that each object need only contain version numbers for replicas that have actually been modified [11].

This method also allows a user to know which devices performed the conflicting modifications, which may help the user in resolving them. Like many similar systems, devices are required to keep *tombstones* for deleted objects until the tombstone can be garbage collected. A tombstone is a marker that notes that a particular object existed, but has since been deleted.

Garbage collection: Over time, object version vectors will grow in size because each time a new replica is modified for the first time, a new entry must be added to the version vector. Thus, the system must garbage collect old versions or the vectors will grow unbounded. While this garbage collection is necessary, version vectors are not expected to grow very rapidly, since a new entry is only added when a new replica is created and modified for the first time. It is expected that replica creation will be a fairly infrequent event, and that many of these replicas will be read-only. This means that garbage collection can be a periodic background maintenance activity, much like filesystem defragmentation.

Our approach only requires that a few devices in the system track garbage collection information. Garbage collecting devices track the latest version vector they have seen from each device that may contain a replica of the object. Occasionally, in the background, garbage collecting devices check for the latest version vector which all replicas have seen and send a message to each device that might contain a replica telling it about this stable version vector. This garbage collection approach allows simple devices to avoid being responsible for garbage collection, while still keeping the functionality available in the system. Multiple garbage collecting devices are safe. Since no device can garbage collect a version unless all devices have

seen it, adding another garbage collecting device will not mark erroneous versions as stable. Since garbage collecting a version is idempotent, it is safe to have the same version garbage collected multiple times. The administrative tool can be used to assure at least one garbage collecting node will clean up each object in the system. This method contrasts with methods like those proposed by Ratner, et. al., that require all system devices to track garbage collection information [11].

When a machine receives a stable version vector, it can discard any version vector entry that is equal to the entry in the stable version vector. For example, if the stable version vector contained version 1 for replica *a* and version 2 for replica *b*, and the device had a version vector containing version 1 for replica *a* and version 3 for replica *b*, the device can remove the entry for replica *a*, since all devices are guaranteed to have seen this particular update. This method can also be used to garbage collect tombstones on delete; once the version vector of a deleted object replica becomes empty, the tombstone can be removed.

To avoid ambiguity on subsequent updates, each device stores the most recent version number that has used for each replica it is currently storing in addition to the version vector. When it updates a replica, it updates the replica version number, and then copies the number into the appropriate location in the version vector. In this way, any new updates will supersede any previous updates, whether they have been garbage collected or not.

It is possible that an entry in a version vector will not be removed from the system after a garbage collecting step, since a device with the entry garbage collected could sync with a node that has not yet been garbage collected, but the entry will be garbage collected again later, after it has moved through the system. We expect this to be infrequent, since it can only occur in certain cases when an object or view is modified during the garbage collection process.

If a device that could store a given object disappears from the system, the system will not be able to garbage collect versions for this object while the device is absent. However, since we believe version vectors will grow slowly, a device can be absent for a long period of time without significantly impairing the system.

View addition and removal: Each device stores all of the views it has published in a single control object, which also contains other device-specific information. Since only the device edits this object, we can use a single version number. Views can be added or removed by any device at any time, as long as devices exchange view updates before doing other operations on reconnection. We will show that addition is safe with garbage collection. Assume device A adds a new view and gets an object with version v from device B while disconnected from G, the garbage collecting node. This will only be unsafe if the system can make a version v' stable, without A knowing about v'. There are two cases. Case 1: B has already seen update v'. In this case, A will know about v' since it received the object from B. Case 2: B has not yet seen v'. In this case, G cannot mark v' as stable until it has communicated with B. When it communicates with B, it will discover the view on A and will be unable to mark v' as stable until it has communicated with A.

We must also show that adding a view cannot result in a device missing or receiving additional updates. To remove addition updates the device checks that the update actually falls within the view; if it does not, it drops the update. A device will not miss updates because it will eventually sync the new view with all devices in the system.

Removing a view removes a replica and cannot introduce inconsistency, because no new versions are created. Thus, it is always safe to remove a view in terms of consistency. However, the system must take care when removing views that it does not remove replicas containing untransferred updates, as discussed in Section 2.3.4.

When devices permanently leave the system, a background task can notice that a device has not been seen for a long period of time and query the user to determine if the device has permanently left the system and can be deleted. If the device has permanently left the system, the administrative tool can delete the control object for that device, which will prompt other devices to remove the device's views.

Assigning device and object ids: Device IDs must be unique in the home environment, as in any network. This could be accomplished a variety of ways. The simplest is to use globally unique IDs, as is

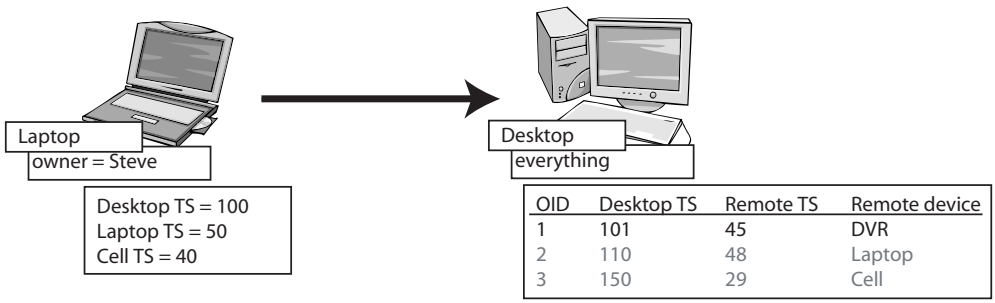


Figure 4: **Sync.** This shows Steve's laptop syncing its view with the Desktop machine. Notice that Steve's laptop has kept the most recent timestamps it has seen on this view from three devices. By passing these extra timestamps, it can prune unnecessary object data. Notice that object 1's metadata will be passed back to the laptop, because the timestamp for both the desktop and the remote machine are larger than the corresponding entries for the laptop. Object 2's metadata is not transferred because the timestamp for the desktop is smaller than the timestamp that the laptop has stored on this view for the desktop. Since the laptop also passed the timestamp for the cell phone, object 3's metadata does not need to be transferred.

done with Ethernet networks. Alternately, a system could use the hash of a user-supplied name and require users to choose unique names.

To avoid conflicts in the object ID space, each device must choose an ID constructed out of the device ID and a local counter that is guaranteed to grow monotonically.

2.4.2 Data freshness

In the home storage environment, devices will frequently exist in different partitions. The system must be able to keep data fresh even as devices leave and enter any particular partition. To accomplish this task, devices synchronize state when they discover one another after a period of disconnection.

To keep devices from having to transfer all state when reconnecting, each device labels each update it sees with a local timestamp. It also keeps the device number of the update creator and the local timestamp for the update on the update creator. Each device assures that it passes updates for each view in the order specified by the local timestamp. Note that these timestamps are logical, they must only be monotonically increasing on that device. The system does not require any synchronization of clocks.

When a device A synchronizes a view with another device B, device A passes the last local timestamp it received from device B for this view. In this way, device B knows it must pass the metadata for any object modified after the given timestamp. To further prune the number of updates transferred, device A can also pass along the most recent local timestamps for the view from other devices. This allows device B to prune any updates whose originator timestamp is smaller than the corresponding provided entry. This is similar to the approach taken by Bayou [15].

To protect against lost messages, each update also contains the local timestamp of the last request forwarded to that view. In this way, if a device receives an update with a previous local timestamp that is not equal to the current local timestamp for the given device and view, it knows it must synchronize with the device to obtain missed updates. See Figure 4 for an example of synchronization.

Each device can customize how it stores and updates internal information to determine what metadata must be synchronized. Devices are required to pass on any updated metadata, but similar to Bayou, they can also pass on metadata that has not been updated. Thus, devices can always resort to swapping metadata for all objects they store, or other lossy or lazy ways of tracking updates.

One significant optimization is that devices need only pass on objects with non-empty version vectors, so stable objects that have had metadata garbage collected (see section 2.4.1) do not need to be transferred. In the case where metadata is stored in a database, the device can forgo using a log and instead store the local timestamps in the database. On sync, it can retrieve all applicable objects using a query on the timestamps.

2.4.3 Data convergence

In our system, like many similar systems, we rely on devices swapping information as they come in contact with one another to propagate updates through the system. In this kind of loosely connected system, it is critical that updates to an object eventually flow through the system to all devices that store replicas of the object, so that the replicas in the system eventually converge to the same value. In many cases, this should naturally occur in the home, where devices tend to communicate with most other devices at least occasionally.

Bayou only requires that the graph of connections between devices in the system be connected, because all data is stored on all machines. However, because our system doesn't store all data on all machines, we effectively have a graph for each object, which includes all devices with views that may contain the object. Each of these graphs must be connected. If the graph for an object is not connected, then some updates may not reach some replicas.

However, by having devices track how frequently the other devices in the system are accessible, it is possible to detect that the system is not fully connected. By occasionally pushing this information to the administrative tool, the tool can check that updates will reach all replicas. If some device is not fully connected, the tool could suggest new complete views in the system to assure that this property does hold.

3 Related work

The Semantic Filesystem [3] proposed the use of attribute queries to access data in a file system. Subsequent systems, like HAC [4], showed how these techniques could be extended to include standard hierarchical schemes and user personalization. Semantic filesystems have traditionally used one of three approaches to support queries. The first approach is to use a centralized metadata service to store metadata and process queries. However, this approach will not be feasible in the home space. Some devices will often be outside of the home environment. A central service would require them to always be connected, not likely on long car trips, airplane locations, etc. Even when two devices have network connectivity, firewalls can prevent them from communicating with one another. Even if connectivity is available, power and cost constraints may prohibit making network round trips on all operations.

Relying on a single server also reduces flexibility and may increase need for active administration. In particular, one must make sure it is always on and available. A UPS would be needed to protect against unavailability during power failure, and it would need to protect communication paths as well as the central server itself. A decentralized approach means that any subset of devices should be able to function, eliminating problems if the central server fails or must be turned off for convenience or maintenance. The central server approach also makes it difficult for devices from different homes to coordinate, perhaps a common procedure as friends and family enter the home temporarily with data to be temporarily or permanently shared.

Some systems, such as Roma [14] and Segank [13] take a related approach, in which they store metadata on a small storage device meant to be carried with a user wherever they go. While this may work well for a single user, the home environment contains a large amount of data that is shared between the different home users, making this approach inappropriate.

The second approach is to perform a query, cache the results, and then occasionally update the cached copy. HAC [4] and Wayfinder [10] take this approach. This approach is not ideal in a system like the home, where updates often need to be immediately visible. Users will not tolerate large delays to see new objects or requirement of manual commands to refresh. In addition, this approach may be expensive if the workload is read dominated, as expected, since it requires re-querying many devices rather than only seeing the small number of updates.

A third approach is to only perform queries on demand. The Semantic Filesystem [3] takes this approach. A naive implementation of this approach will not work in a distributed system, since it requires querying all nodes in the system. A more sophisticated approach may cut down some of the overhead. However, these systems still perform most of the work on reads rather than writes, which will be less efficient in a read mostly workload.

Our approach is based on a publish/subscribe model. There has been a large amount of work in extending publish/subscribe systems to large scale environments. Siena is one example that allows a set of clients to subscribe to a set of events created by content publishers [1]. Siena uses a SQL-like query language to express these queries, expanding previous work using channels to direct messages.

Bayou [15] and Ficus [5] provide eventual consistency in a decentralized, partitionable network environment. Bayou supported full volume consistency using a log of updates and a central device to decide the order of updates. Ficus offered per-object consistency using version vectors. Footloose [8] proposed allowing individual devices to register for data types in this kind of system, but did not complete the system or expand it to general publish/subscribe-style queries. PRACTI [2] expanded these techniques allowing devices to avoid seeing all updates by using update "summarizations," which describe a range of updates without having to describe each individual update. However, they did not specify how these update decisions would be made and assume extra complexity to provide full volume consistency.

Coda [12] pioneered the use of disconnected devices and reintegration in a two-tier system, with a set of clients and a set of servers. Coda allowed clients to disconnect from the system and use cached data, which was then reintegrated into the system when the device reconnected. It also allowed servers to reintegrate if they became disconnected. Coda built application-level resolvers and showed that the disconnected mode could work in real systems. The Blue filesystem [7] extends Coda functionality to allow clients to read data from some number of replicas stored on a variety of storage devices. It also includes system plug-ins to optimize power consumption by going to the most efficient replica.

Our work differs from this related work in that it combines publish/subscribe-style flexibility in the context of a decentralized eventually-consistent, semantic storage system.

4 Conclusion

In conclusion, we have described the concept of *views* to effectively manage data in the home environment. The system provides eventual-consistency and data freshness in a decentralized environment using these views. We have also discussed some data management problems that views simplify, such as data search, reliability, and user transparency. We are currently building and testing a prototype system using views, called Perspective, to validate our ideas and more fully solve the various data management problems digital data in the home will create.

References

[1] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. *Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000)* (Portland, OR, July 2000), pages 219–227, 2000.

[2] Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramana, Praveen Yalagandula, and Jiandan Zheng. PRACTI Replication. *Symposium on Networked Systems Design and Implementation* (May 2006), 2006.

[3] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. *ACM Symposium on Operating System Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):16–25, 13–16 October 1991.

[4] Burra Gopal and Udi Manber. Integrating Content-based Access Mechanisms with Heirarchical File Systems. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), 1999.

[5] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page Jr, Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. *Summer USENIX Technical Conference* (Anaheim, California), pages 63–71, 11–15 June 1990.

[6] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 73–86. USENIX Association, 2004.

[7] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the Blue file system. *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 06–08 December 2004), pages 363–378. USENIX Association, 2004.

[8] Justin Mazzola Paluska, David Saff, Tom Yeh, and Kathryn Chen. Footloose: A Case for Physical Eventual Consistency and Selective Conflict Resolution. *IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, 09–10 October 2003), 2003.

[9] D. Stott Parker, Gerald J. Popek, Gerald Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, **9**(3):240–247, May 1983.

[10] Christopher Peery, Francisco Matias Cuenca-Acuna, Richard P. Martin, and Thu D. Nguyen. Wayfinder: Navigating and Sharing Information in a decentralized world. *International Workshop On Databases, Information Systems and Peer-to-Peer Computing*, 2004.

[11] David Ratner, Peter Reiher, and Gerald J. Popek. *Dynamic Version Vector Maintenance*. Tech report CSD-970022. Department of Computer Science, University of California, Los Angeles, 1997.

[12] M. Satyanarayanan. The evolution of Coda. *ACM Transactions on Computer Systems*, **20**(2):85–124. ACM Press, May 2002.

[13] Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Junwen Lai, Yilei Shao, Chi Zhang, Elisha Ziskind, Arvind Krishnamurthy, and Randolph Y. Wang. Segank: a distributed mobile storage system. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 239–252. USENIX Association, 2004.

[14] Edward Swierk, Emre Kiciman, Nathan C. Williams, Takashi Fukushima, Hideki Yoshida, Vince Laviano, and Mary Baker. The Roma Personal Metadata Service. *In Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, December 2000), 2000.

[15] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5), 1995.