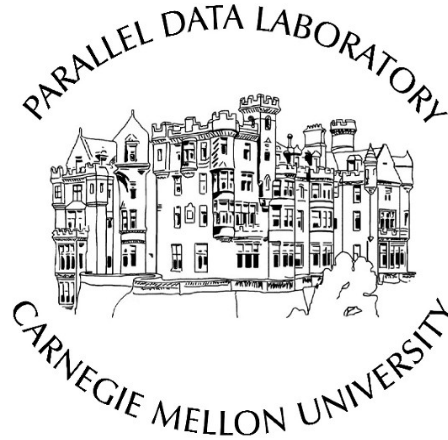

Putting home storage management into Perspective

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

CMU-PDL-06-110
September 2006

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

Abstract

Perspective is a decentralized data management system for the growing collection of consumer electronics that store and access digital content. Perspective uses a new construct, called the view, to concisely describe which data objects each device may store and access. By knowing the views in the system, a device can know which devices may need to hear about any particular update and which devices to contact for a particular search query. By exchanging views, an ensemble of devices can coordinate and share data efficiently without relying on a centralized server; it works the same in the home or for a subset of devices outside the home. Experiments with Perspective confirm that views improve ensemble creation and search performance, by avoiding costs that are linear in the number of objects stored, without penalizing performance relative to an ideal centralized server setup.

[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, and in part by an Intel Fellowship. Part of this work was done while Brandon Salmon was an intern at Intel Research Pittsburgh.

Keywords: home storage, views, optimistic concurrency, perspective, storage management, object storage, distributed systems, distributed reliability

1 Introduction

Digital content is now common in the home. An increasing number of home and personal electronic devices create, use, and display digitized forms of music, images, videos, as well as more conventional files. People are increasingly shifting important content from other forms of media, such as photographs and personal records, to online digital storage. The transition to digital homes with heterogeneous collections of devices for accessing content is exciting, but does bring challenges.

Perhaps the biggest challenge is data management for this environment. Most consumer devices (e.g., digital video recorders and digital cameras) are specialized, each with simplified interfaces to help users interact with that device and the data it stores. But, realizing the promise of the digital home requires moving past per-device interactions to easily coordinated data management across collections of storage and access devices. Users will need assistance with consistency of replicated/cached data, searching for particular data within the environment, ensuring reliability of important data, and so on. These features must be well-supported and made intuitive, as most users will not tolerate the manual effort nor be capable of the expertise required today.

One approach would be to reuse distributed file systems, having each device act as a client of a conventional central file server. We think that this approach has a number of shortcomings. For example, most file systems provide little assistance with search, but this may be one of the most common actions in this environment. Also, users will often take subsets of devices elsewhere (e.g., on a family vacation) and wish to use them collectively away from the home; while a few file systems provide support for disconnected operation, very few combine that with support for ad hoc coordination and sharing among disconnected clients. On a practical level, traditional file server administration does not create confidence that it will be easy for non-experts in a home environment. In addition, consumer device vendors may want more flexibility in tailoring device data management than is normally associated with distributed file system solutions.

This paper promotes a decentralized model based on semantic (a.k.a. attribute-based) file/object naming and metadata summaries that we call *views*. Semantic naming associates descriptive metadata (e.g., keywords) with data objects and allows users to search in various ways. Much new consumer electronics data, such as music and television, fits this model immediately, since it is distributed with rich metadata about the content. Desktop file management is also moving in this direction, with tools such as Apple's Spotlight and Google's Google Desktop. A view is a compact description of a set of metadata, such as "all objects with type=music and artist=Aerosmith," expressed much like a search query. A view provides a way of compactly expressing the data objects that a device has interest in and may store.

Perspective[1] is a distributed data management system designed for home/personal storage. It assumes semantic naming and uses views as a building block to efficiently support consistency, search, reliability, and ad hoc creation of device ensembles. By exchanging their views, which are small and change infrequently, devices can identify which other devices may need to know about particular new data (for consistency) and which other devices may have data relevant to a search. Views assist with redundancy management by making it possible to determine which devices promise to hold which replicas, which in turn makes it possible to verify appropriate reliability coverage. Ad hoc ensembles can be efficiently created by simply exchanging views among participating devices to enable appropriate consistency and search traffic. With views, these features can be supported without relying on a reliably maintained central server that stores all data in the environment.

Experiments with Perspective show that views provide significant performance benefits in creating device ensembles. They also show performance advantages over alternate methods in providing search and event notification in ad hoc ensembles, while automatically providing the same performance as a central server when similar resources exist. Experiments also show that the overhead of evaluating views is negli-

[1]In seeing many views, one gains Perspective.

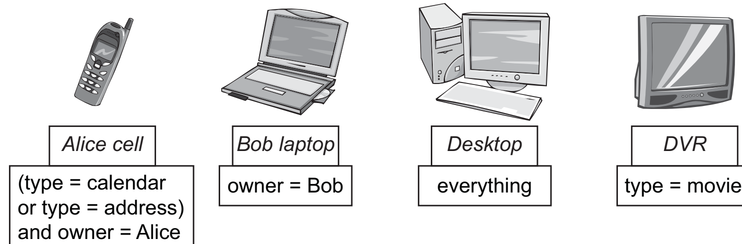

Figure 1: **An example set of devices.** In this diagram, we have four devices, each with a customized view. Alice's cell phone is only interested in "calendar" and "address" objects belonging to Alice, Bob's laptop is interested in all objects owned by Bob, the house desktop is interested in all data,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

gible.

2 Home storage and views

This section discusses challenges and goals for home storage, views and their uses, and related work.

2.1 Storage management at home

Storage management is not only a challenge for large enterprises, but it is also a challenge in our homes. Today's users are faced with the task of managing a collection of devices in their own home that store increasingly valuable and precious personal data. The digital data that we store is no longer just documents and the occasional recipe, but is personal, often irreplaceable data like family photographs, tax returns, and home videos. Users need ready access to their data, they need help configuring and maintaining their infrastructure, and they need as much protection from failure as their collection of devices can provide.

The devices that make up a user's home storage infrastructure vary widely from normal PCs and laptops to portable music players, digital video recorders, and network-attached storage servers. Devices will form ensembles that are at times static (e.g., everyone is at home and all devices are well-connected) and at other times dynamic (e.g., several devices are taken along while the family is on vacation). Consumer computing equipment is often not as reliable as enterprise equipment, so home users' devices are more prone to failure.

Automating storage management for the home will require a home storage infrastructure with manageability as a central focus. We believe that manageability is not something that can be added onto a system after the fact, but must be part of the design of a system from the outset. Home storage requires a distributed data management system that easily handles device addition and failure without user intervention. In addition, it must provide for data accessibility and search in arbitrary groups of devices. The system should also make it easier for users to reason about where their home data is stored and how it is protected. We believe that the views can be a powerful organizing concept for home data management supporting such features.

2.2 Views

A *view* describes a set of objects in the semantic store in the form of a query on per-object metadata. For example, a simple view could specify every object, i.e., "*". More sophisticated views could specify all objects that are of a particular type, are newer than a particular date, or are owned by a particular user. In fact, a view can specify an arbitrary query on any extensible metadata in the system.

In Perspective, each device registers with the system one or more views that express the data objects that the device is interested in. Devices only store data that matches at least one of their views. Whenever

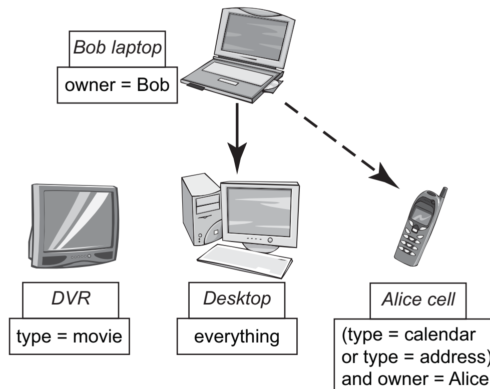

Figure 2: **Search.** Views can improve search efficiency. In this example, Bob's laptop performs a query for calendar entries for today. Because the laptop has a copy of all views in the system, it knows which devices could hold an object of that description. In this case, this would be both the desktop 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.

data is created, modified, or deleted in the system, a notification of that update is sent to all devices with views that match the modified object. Each device sends a copy of all its views to all other devices, allowing each device to easily determine which other devices should receive a notification message.

Figure 1 shows an example set of views. In the example, imagine that Bob (the owner of the laptop) creates a movie clip on his laptop. Perspective will send an update notification to those devices in the system with matching views, in this case the Desktop and the DVR. If Alice updates her address book on the Desktop machine, that update will only be propagated to the cell phone.

When an update notification is received by a device, that device can decide whether to accept that update and store the data or ignore that update and not store the data. If a device registers a view as being *complete*, that device promises that it will always accept updates for objects in that view and will never delete those objects once they're stored. If a view is instead registered as being *partial*, the device registering that view is free to ignore updates and delete its copies of objects, if it so chooses. The distinction between complete and partial views has ramifications for both search and reliability management.

This basic application of views enables automatic replica coherence with available devices. That is, as objects are modified in the system, replicas of those objects can be automatically kept in sync by Perspective based on update notifications delivered according to the views. In addition, views provide event notification; as new objects are created, devices with matching views will hear about their creation. In addition, views can be used to improve other important functions, including search and reliability maintenance.

2.2.1 Search

Search is likely to be one of the most common operations in home storage systems, so it is important to not only enable it but to make it efficient as well. Rich metadata provides a great deal of useful content for search, and views can enable some useful optimizations.

By comparing the search query to the views in the system, it is simple to exclude devices that could not store the desired data. For example, the search for calendar data illustrated in Figure2 is posed from the laptop. Using the views in the system, Perspective is able to exclude the DVR from the search since calendar entries do not match its view.

Complete views can simplify the search process even further. Because a device with a complete view is guaranteed to store all objects that match that view, if a complete view contains the search, the device only

needs to forward the search to that device. If the Desktop had a complete view on “everything”, the example search in Figure 2 could be executed just on the desktop machine and would not have to be propagated to the cell phone.

Efficient decentralized search is especially useful when dealing with devices from multiple administrative domains. One challenge in sharing data between domains is that they do not collaborate on name spaces, making it difficult to find data. However, by decentralizing the search process and using a semantic model, a device can see data on a device from another domain in exactly the same way it sees data from a local domain. It is free to do access control as it sees fit, and “domain” could be one part of the metadata.

2.2.2 Reliability

Providing strong reliability is important in home storage systems, as the data that users store is often irreplaceable and the devices that they use to store it are typically inexpensive and failure-prone. Relying on the user to buy more reliable (and more expensive) hardware like better disk drives or RAID arrays seems problematic. Home users’ purchase decisions are more likely to be driven by initial costs (i.e., “what’s on sale online today?”) rather than long-term costs. Instead, a home data management system should enable users to get the best reliability from the ensemble of devices they already have, using the natural redundancy of devices in the home.

For example, a user with several well-provisioned devices (e.g., several PCs, a digital video recorder, etc.) should be able to automatically replicate data across those devices to provide reliability. When a device is added, fails, or reaches its capacity, the system should automatically reconfigure itself to maintain reliability.

By configuring several devices with the same views, Perspective will automatically propagate updates to replicated data on those devices. Views can be simple (e.g., “*”) or more constrained (e.g., “all objects where type = jpeg”), potentially providing different levels of protection for different types of data. Views that are defined as complete will provide strong guarantees that the matching data will not be dropped or deleted from a device.

By observing the views in the system and the objects that are present, an administrative tool can quickly determine the level of protection that the current views provide. For example, if it observes that there are two views of “*”, and one view for “all objects where type = jpeg”, a tool can easily see that photographs will be protected with three replicas, and all other data will be protected with two replicas. Devices with partial views are not guaranteed to provide reliability, since they are free to ignore updates or evict objects, and so would not be considered.

Of course, home data management should not preclude the use of more reliable storage devices, like RAID arrays, in the home. Using storage devices that already provide redundancy can certainly help and will simply affect the reliability offered by a given number of complete views. In fact, adding a “reliability server” or “online backup” device could consist of simply plugging it in and allowing it to register a complete view for all objects.

Perspective provides efficient propagation of updates among whatever devices are available and based on the rules set by the views. In addition, Perspective takes care not to lose updates or replicas needed for reliability when adding and removing views, as described in Section5.2.

2.3 Managing with views

Views are a compact interface between management activities and a heterogeneous, highly-partitioned storage infrastructure. By setting the views correctly, management tools can assure that users automatically see the correct objects on the correct devices and assure that a given number of replicas exist for each object in the system.

Management tools could include applications that allow users to hand set views, automation tools that observe user behavior and set views appropriately, or defaults that are put onto devices at manufacture time.

These tools can perform such operations without having to be tied to a central server and without having to see all devices in the system. A management tool built on top of Perspective can be stateless, simply connecting to some device in the system to see the views in the system, and propagate changes using that device. So, an automation tool could, for example, be a Java applet downloaded over the web.

2.4 Related work

Perspective uses publish/subscribe methods to provide eventual consistency, search and device ensembles in a semantic, decentralized home storage system. It extends previous work on home storage, semantic storage, decentralized storage, and publish/subscribe.

Several recent systems have targeted personal storage. EnsemBlue [13] extends the Blue filesystem to provide functionality needed for consumer devices. It added event notification to the filesystem through *persistent queries*, and allowed ad-hoc ensembles of devices by creating a pseudo-server that fills in for the central server. Persistent queries are similar to views in that they allow event notification and search in the system. However, EnsemBlue implements these queries by utilizing the underlying filesystem cache coherence protocols which rely on a server or pseudo-server. Perspective extends the idea by using the views themselves to implement ad-hoc ensembles, replica coherence, more efficient search, and reliability without requiring a central server.

Several other projects have also addressed personal storage. Omnistore [9] and Segank [20] allow a distributed filesystem across mobile devices, using a central device to provide consistency. The Files Every Where [15] and Unison [14] focus on synchronizing sets of replicas by using epidemic propagation. The Data Furnace [3] project also addresses home storage, but targets temporary data from sensors and uses a centralized architecture.

Perspective uses search on data attributes to allow flexible access to data across heterogeneous devices. The Semantic Filesystem [5] proposed the use of attribute queries to access data in a file system, and implemented such techniques in a local filesystem. Subsequent systems, like HAC [6], showed how these techniques could be extended to include standard hierarchical schemes and user personalization. Newer systems like WinFS, Beagle, Google Desktop, and Spotlight borrow from the Semantic filesystem by adding semantic information to filesystems with traditional hierarchical naming, in order to facilitate local search. Perspective uses views to provide an efficient implementation of a decentralized semantic store, on top of local semantic access.

Diamond [8] also provided optimized search by pushing search into individual devices. Views could be used to guide similar kinds of content searches. Active Disks [17] also proposed pushing this kind of functionality into individual storage devices.

Perspective also builds on a rich set of previous work in disconnected distributed filesystems. Coda [18] 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. The Blue filesystem [10] extends Coda functionality to allow clients to read data from replicas stored on a variety of storage devices, and includes system plug-ins to optimize power consumption by going to the most efficient replica. This two-tiered architecture assumes that the servers are relatively static, while allowing clients to be dynamic. In contrast, Perspective makes every device into both a server and client, allowing all devices to exist in a highly dynamic environment.

This architecture has also been explored by several previous filesystems. 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 [11] proposed allowing individual devices to register for

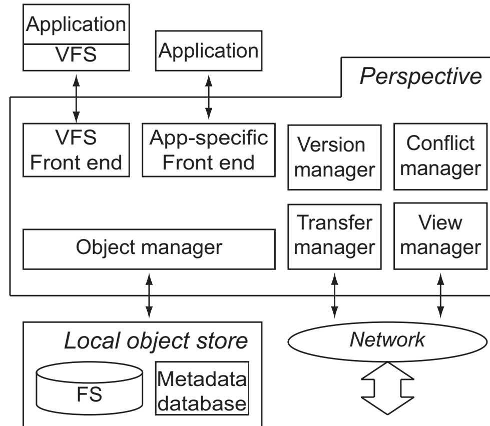

Figure 3: **Block diagram.** This block diagram shows the major components of the Perspective.

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. Perspective builds on the consistency techniques used in these systems by using views to provide structure for inter-device access like device ensembles, sync, and search.

Views follow 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. These systems focus of providing search in a wide area, high dynamic environment, while Perspective uses them to provide consistency and reliability in a smaller environment.

While not strictly publish/subscribe, the SharedViews [4] project also uses a similar technique to views to allow users to share read-only data with one another over the wide area network. They use capabilities with the views to provide access control, and provide search over views, but do not seek to use them to provide update propagation, replica coherence, consistency, ad hoc ensembles or reliability.

3 Design overview

Perspective is a decentralized semantic object store based on views. It allows devices to keep data they store in sync, search for and access data stored on other devices, and receive notification when objects of interest are created or modified. It enables arbitrary groups of devices to form ensembles without requiring arbitration through central servers to share data. Perspective uses version vectors to provide per-object eventual consistency in the face of device disconnections, a common occurrence in the target domain.

Because Perspective is a semantic object store, search is the primary way to locate data. In a traditional filesystem, an application accesses data by first searching through directories to find an inode number and location, which it uses to access a file. In Perspective, an application finds an object through a search on object metadata, which leads to an object ID and location that can be used to access the object. Views make this search efficient in a decentralized environment.

This section describes the major components and some fundamental concepts of Perspective.

3.1 Components

Figure 3 shows the major components of Perspective. We have designed Perspective in a modular fashion, to allow alternate implementations (for experimentation purposes) of many of the components.

View manager: The view manager handles all operations on views. It sends update messages to the appropriate other devices when local changes are made, and accepts messages from remote devices, handing them off to the object manager for processing. The view manager also decides which objects to return when another device asks for sync information. Search is also performed through the view manager, which passes the search on to the appropriate devices. Overall the view manager manages the control messages for Perspective by communicating with view managers on remote devices.

Transfer manager: The transfer manager is responsible for transferring data between devices. The transfer manager may contain a set of data transfer protocols, from which it can choose an appropriate method, depending on the data, the device and current connectivity.

Version manager: The version manager compares two versions of an object and decides which version is newest. If two versions conflict, it will ask the conflict manager to resolve it. Perspective uses version vectors to track object versions and identify conflicts.

Conflict manager: The conflict manager is responsible for resolving conflicts when the occur in the system. Perspective can use user directed conflict resolution or application level resolvers, just like Coda and Bayou. [21, 18]

Object manager: The object manager coordinates updates to objects, applying updates to the local store, passing messages to the frontends and the view manager, and processing front end requests.

Frontend: A frontend is responsible for connecting an application to Perspective. The frontend customizes the way in which it exports objects to an application. A frontend could implement a standard file system interface by mapping object attributes into a file system hierarchy, like the Semantic file system. [5] This allows an application to mount Perspective through a standard file system interface. The frontend can convert from filesystem directory operations into Perspective searches. Alternately, a frontend could communicate directly with an application to customize the interface. For example, a frontend might automatically include new music items into iTunes. Each frontend can register with the object manager for callbacks on object modification. This allows a frontend to decide if updates are ignored, maintain extra state if it chooses to do so, and perform extra operations, such as automatically transcoding objects on updates.

Local object store: The local object store stores object replicas and metadata. We have implemented our own local object store, but any semantic store would suffice.

3.2 General concepts

This section describes some concepts in Perspective.

Device ensembles: A device ensemble is a group of devices that communicate with one other to share data. [19] For example, in the home, computers, digital video recorders, and devices from visitors will share data. Alternately, a family on a road trip may have laptops and music players in the car and would be interested in sharing data between them.

Both of these cases are considered ensembles, and handled the same way in Perspective. Views allow Perspective to automatically adjust to the types of devices found in the ensemble, whether they are capable devices found in the home, or an ad-hoc collection of devices outside the home.

In order to form an ensemble, devices exchange views with one another, and sync with the other devices. When a device discovers a new device, it will send a copy of its views to the new device and sync any overlapping views with it. Sync is described in Section 4.1.

When a device makes an update to an object, or creates a new object, it will send an update message to all accessible devices with overlapping views. In this way, members of an ensemble will see new updates as

they occur in the ensemble.

Devices in an ensemble do not have to share an owner or central server in order to participate in an ensemble. An ensemble could contain devices from one household, and also devices owned by visitors, without requiring coordination through servers. Each device is free to decide whether to send searches to, or accept data or updates from, another device.

Section 7 shows that views make ensemble creation an efficient operation. This allows ensembles to be dynamic, such as the group of devices found in a coffee shop or in a meeting room.

Device IDs: Unique device IDs must be assigned for Perspective's schemes to work correctly. This could be implemented by assigning each device an ID at manufacture time, much like a MAC address, and supplemented by an epoch number on format, or by constructing the ID from a secure hash of personal information and an epoch number.

Object IDs: Each object in the system is uniquely identified by an *object ID*. To assure that an object ID is unique, a device assigns a new object an ID constructed from its device ID and a monotonically increasing counter on that device.

Replica IDs: Each replica of an object is uniquely identified by a *replica ID* . A device chooses a new replica ID based on the device ID and a local counter that is guaranteed to grow monotonically, just like object IDs. Replica IDs are required for consistency, as described in section5.1.

Version vectors: When an application updates an object in any way, it must increment the version vector associated with that object to reflect the change. A version vector stores a version number for each modified replica of the object in the system. It is used to determine which of two updates is more recent, or detect if updates have been made concurrently. [12] Version vectors in Perspective are discussed in more detail in Section 5.1.

4 Accessing and moving data

Applications access data through a frontend, or through a filesystem mounted on top of a frontend. Perspective provides standard filesystem operations like open, close, read and write, in addition to metadata read and write operations to each frontend. Perspective also allows frontends to perform searches. Perspective allows applications to access data stored on a local device or search and access data stored on remote devices. This section discusses the methods employed to provide these features.

Search: Frontends can find objects by performing searches based on queries on object metadata. When a frontend asks the object manager to perform a search, the object manager asks the view manager to forward the search to remote devices. The view manager checks with the system views and forwards the operation on to the appropriate devices, possibly including the local device. These searches return the metadata for matching objects, which the view manager combines. An frontend can use this metadata, to access an object.

Data access: This section discusses the implementation of the basic data manipulation commands in Perspective.

Create: When an frontend creates a new object, it passes the metadata for the object through the frontend into the object manager. The object manager assigns a new object ID and creates the object in the local object store. The object manager returns the new object ID to the frontend for reference. It also forwards the update to the view manager to forward on to remote devices with matching views.

Open: To open an object in Perspective, the frontend passes through the metadata. The object manager extracts the object ID and replica ID from the metadata. The object ID allows the object manager to index into the local object store and look for the object. If the object is not found locally, the object manager extracts the device ID from the replica ID and creates a local replica by asking the transfer manager to fetch the object from the remote device before opening it.

Because devices are only allowed to store objects that fall into one of its views, if no local view contains

the object, the object manager must also create a temporary view containing the individual object before creating a new local replica, in order to maintain consistency. While this is more expensive than a normal access, we expect it to be rare, since views should capture the normal usage patterns of devices. When the object is closed, the object manager can remove the replica and temporary view, or choose to keep the replica locally for future access.

Read: To read an object, Perspective simply opens the local copy of the object and allows the frontend to read data from it.

Write: When an frontend writes to an object, the object manager asks the version manager to update the version vector for the given object. The object manager then writes the change through to the local object store and asks the view manager to send the update on to the appropriate devices.

Delete: The delete operation replaces the object metadata with a tag, or *tombstone*, that marks it as deleted and then removes the actual data from the local object store, like many other decentralized systems. [7] It then forwards the update to the view manager to forward to the appropriate devices. This tombstone can be removed once the version vector for the tombstone has been completely garbage collected. Garbage collection is discussed in Section 5.1.

Forwarding updates: Any local modification to an object is converted into a update that is then passed to other devices. If it is a metadata only operation, like writeMetadata, the update can be marked as such, to inform devices they do not need to fetch the data. The view manager checks the views to determine which devices should receive the update. It also marks the update with a unique local timestamp and may add an entry to a synchronization log for this device, discussed in detail in the next section. The view manager queues the update to be sent, and lets a thread send it later, because all updates must be sent in order of the local timestamp to make the sync operation efficient. The object manager also notifies any interested frontends about the update.

4.1 Sync

Sync is necessary to bring two devices up to date with one another after a period of disconnection. It is a unidirectional operation that brings one device up to date with another. After a device A syncs with device B, A will contain the most recent version stored on either device of all objects stored on A, assuring that updates are not lost in the system.

To make this process efficient, many systems have devices keep a sync log, which is a log of operations a device has seen ordered by a timestamp, allowing devices to only exchange information about objects that have been modified. However, the system must also have a '“full” sync that works in cases where a log is not available, such as the addition of a new device or a device that has truncated or lost its log.

The naive approach to sync is to transfer data about all objects stored on device B back to device A. While this works from a correctness standpoint, it is inefficient, because it requires the devices to transfer large amounts of unnecessary information. This is especially true in Perspective, where device B could be a desktop machine storing thousands of objects, while device A could be a cell phone interested only in one or two objects.

Perspective uses views to optimize sync. By syncing views rather than full devices, we can make sync efficient, even if two devices store radically different subsets of data. Instead of having to exchange information about all objects stored on either device, they must only exchange updates to objects that fall on both devices. Perspective also uses sync logs to restrict the number of updates devices must exchange.

When a device A notices a new device B enter its ensemble, it will sync each of its views with device B, to receive any updates it may have missed. To sync a given view, device A passes the last local timestamp it received from device B for this view. Device B returns the metadata for any object modified after the given timestamp that falls within the given view.

If device B has not kept a sync log, or has truncated its log to a time after the given timestamp, it can choose one of two “full” sync options. By providing these two options, we handle the case where a small device with a view containing a limited number of objects can sync with a large device storing a huge number of objects without ever having to exchange all of the objects stored in the large device.

The first option is for device B to pass back the information for all objects it stores. However, if device B holds many more objects than device A, this may overwhelm A. Alternately, B can pass back all objects it stores within the view and also inform device A that this may not be a complete list. After applying the updates B has returned, A must send a list of all objects it stores within the view that were not in the list B returned to A. B can then forward any updates it may have to those objects to A, bringing A fully up to date with B, without them having to exchange all objects on A.

However, this still requires the devices to exchange information about all objects that fall on both devices. To eliminate this inefficiency, device B can keep a sync log. This log contains all of the updates the device has applied ordered by a local timestamp. It can find all applicable updates by searching all updates in the log that occur after the timestamp passed by A, and match the given view. The log is purely an optimization, and can thus be truncated if necessary.

This method keeps device B from returning an update to device A that device A has already seen from device B. However, device B cannot know if device A has already received the update from another device, so device A will see the same update once from each device that stores a replica of the object. To further prune the number of updates transferred, Perspective allows device B to prune some updates that device A has already seen from other devices. Specifically, we will allow device B to prune updates that were processed by some other device with with overlapping complete views, with which device A has already synced.

To enable this process, the metadata for each update contains a list of timestamps for devices that have processed it. When a device processes an update it adds a new timestamp entry for itself into the update. When a device applies an update the timestamps will be stored with the other object metadata making it available later. At sync time, in addition to the timestamp from device B, device A can pass along the timestamps obtained by syncing with other devices with complete views that overlap this view. This allows device B to prune updates that were processed by another device that has already synced with device A. Device B finds these updates by comparing the timestamps found in the update with the timestamps passed by device A. If the update contains a timestamp that is smaller than the corresponding timestamp passed through by device A, device B does not need to pass it back to device A. This is similar to the approach used in Bayou. [21]

Device A can only pass along timestamps from devices that only have complete views that overlap with the view we are currently syncing. If device A tried to use a timestamp from a device C with an overlapping partial view, we cannot be sure at sync time that device A saw all updates device C has processed, since device C may have processed an update and then dropped the corresponding object from its cache. In this case, even if device A has synced with device C up to a certain timestamp, there may be updates on device B that were processed by device C before the given time, but were not passed to device A when it synced with device C, because they did not exist on device C when it synced with device A. Thus, if device A passed through the timestamp for device C, device B might prune updates which it thinks A has seen, but A has really not yet seen.

We choose to sync a view at a time with a new device rather than synchronizing a full device with another so that devices can add new views without having to resynchronize old ones. However, this means that objects that fall into more than one view on a device would be synced multiple times. Fortunately, it is straightforward to allow devices to pass lists of views to sync in a single request, to eliminate passing any duplicate updates.

4.2 Remote updates

The view manager receives update messages from remote view managers about new or modified objects that match the device's views. These updates allow the device to fetch new objects in which it is interested and keep currently stored objects up to date.

To keep devices from missing updates due to dropped messages, each update message contains the timestamp of the update and the timestamp for the last update sent from this device to the given view. When a device receives an update message from a remote device, the view manager first checks the previous update timestamp contained in the update message against the current timestamp for the given device for this view. If the view manager detects that an update has been missed, it performs a sync with the remote device that sent the update.

The view manager then passes the update to the object manager, which must decide what to do with the update. It can accept the update by putting it into a queue to retrieve data and add to the local store when desired, or it can ignore the update. If the update is to a complete view, the Object Manager automatically accepts the update. If the view is partial, it can either accept or ignore the update. In this case the object manager queries the frontends on the device; if any frontend would like to accept the update, the object manger will accept the update; otherwise, it will ignore it. If the device already stores a replica of this object, and chooses to ignore the update, it will invalidate the current replica.

If the update is accepted and there is already a version of the object stored locally, the object manager queries the version manager to decide whether the update should be integrated into the object. The version manager checks the version vectors, and if they conflict, it calls the conflict manager to resolve the conflict. The conflict manager may retrieve the data for the conflicting update to resolve the conflict, or it may be able to use only the metadata.

If the version manager decides that the update should be applied, and the data has not already been fetched by the version manager or conflict manager, the object manager will retrieve the data from the remote device using the transfer manager. The transfer manager always requests a specific version of the object from a device. If it fails to find that object version, the object manager will look up the object again to see if the views have changed. If it finds a newer version of the object, it will fail and the object manager will try to get the newer version. If the lookup results are different, it will try to get the data again. If it cannot find the data, because some devices are not accessible, it will wait until one of the named devices reappears.

Once the data has been fetched, the object manager will check again to make sure the update has not been superseded and then apply the change to the object store.

5 Consistency and reliability

It is critical that user data in the home is kept consistent and reliable, even in the dynamic environment of the home. Perspective maintains data consistency and reliability in the system without requiring central system control.

5.1 Consistency

Because concurrent updates are expected to be infrequent and the system must support disconnected operation, Perspective uses an optimistic concurrency control method. Our focus is on providing object-level eventual consistency and allowing users to detect conflicts in the system, so we use techniques similar to Ficus [7], which had similar goals.

We exploit extensible metadata to implement version vectors. We store each entry in the version vector as a metadata tag containing a replica ID and the corresponding version number. 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 [16].

Identifying replicas: Replica IDs are used to identify a replica's entry in the version vector. In many systems the device ID is used as the replica ID. However, in Perspective this is not adequate, because it allows devices to drop modified replicas from their local stores.

If a device acquires a replica, modifies it, drops the replica from cache, and then reacquires a replica at a later time, it cannot reuse the replica ID it used previously. Version vectors will only work correctly if a particular version of an object is unique. If a device reused a replica id, it could enter a corner case that would generate two object versions with the same version vector; if it were to drop an object, later get an older version of the object and modify this older version, it might create a second version of the object with the same version vector, making it impossible to determine which update was more recent. Instead, we choose a new replica ID each time we create a new local replica.

The replica ID for a replica is stored in the metadata associated with the object. On a write to the object, the version manager uses the replica ID to determine the entry in the version vector that should be updated. When a device receives an update, it will clear the replica ID field and replace it with an appropriate local replica ID.

If devices make conflicting updates, the system must resolve them in some fashion. Any device can resolve a conflict, which will then be propagated to other devices in the system. Conflicts can be resolved with user intervention, or application-level resolvers as done in Coda and Bayou. [21, 18]

Garbage collection: Over time, object version vectors will grow in size because, whenever 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.

Perspective uses a garbage collection technique similar to that used in ROAM. [16] But, Perspective folds garbage collection into version vectors, allowing it to be performed using the same path as normal updates, rather than requiring a separate protocol.

5.2 Maintaining reliability

In this section we describe how Perspective maintains reliability in the system while still allowing management tools to modify views as they see fit. We describe the reliability guarantees in Perspective, compare it to a central server that allows disconnected operation, and show how Perspective provides these guarantees.

Perspective allows management tools to analyze and set the reliability for a given set of objects by manipulating complete views. For example, if three complete views cover a set of objects, the objects are guaranteed to have three replicas. However, Perspective needs to maintain these properties while allowing management tools to add and remove views from the system.

Because complete views never drop updates, in steady state these guarantees hold. However, once we allow the system to remove and add complete views, we run the chance of failing to meet the reliability goals. For example, if we remove a complete view covering some set of objects, and then add a new complete view covering them, the new complete view will not contain any objects until the devices synchronize, so the actual number of replicas of this set of objects will be one less than the number implied by the views. We also run the risk of dropping updates. For example, a device might update an object in a partial view and then drop the object replica before the update has been propagated to any other devices.

Perspective allows devices to add and remove views in the system without a central coordinator while still providing guarantees to not drop updates and keep given levels of reliability. It achieves these properties

by applying several simple rules to view addition and removal.

Reliability in a centralized system with disconnected operation: Because reliability in any system allowing disconnected operation has a slightly different definition than that in a system without disconnected operation, we will start by examining reliability in a system with a central server that allows disconnected operation, and then extend to the decentralized case. In a centralized system with disconnected operation, we will define two types of updates: *unstable* updates that have been made on some disconnected device, but not yet passed to the server, and *stable* updates that have been passed to the server.

Three guarantees about stability are desired. First, the system guarantees that in normal (non-failure) conditions no unstable update is lost. It cannot guarantee that an unstable update receives any particular level of protection, since the device making the update could be the only available device at the time, making the update sensitive to a single fault. Second, the system guarantees that any unstable update in the system will eventually become stable. This is assured if the disconnected device will eventually connect with the server. Third, the system guarantees that a stable update will receive the level of protection the server is configured to support.

Reliability in Perspective: In the decentralized case, we seek to provide these same stability guarantees, but we will change the definition of stable and unstable slightly. We first define that a view *covers* an update if the object containing the update falls within the query for the view. Next, we say that a view *contains* an update when that update has been propagated to the view and applied to the replica stored in that view. In Perspective, an update is considered stable when it is contained by all complete views that cover it. Until it reaches this state, it is unstable. We distinguish two types of unstable updates: *dirty* and *clean*. A *dirty* update is an update that is only contained by partial views. A *clean* update is contained by at least one complete view.

Perspective provides the same three stability guarantees as the centralized case: unstable updates will not be lost in normal conditions, unstable updates will eventually become stable, and stable updates will receive the level of protection the system is configured to provide: i.e., the update is contained by all complete views covering it. These guarantees are supported with three rules:

1. A device cannot evict a replica containing a dirty update from a partial view or remove a partial view that contains a replica with a dirty update. Once the replica has been pulled by a complete view covering it, it can be marked as clean and removed.

2. A view cannot be made a complete view until it has synchronized with a complete view covering it. We will show later how to add a complete view if no other complete view in the system contains it.

3. In order to remove a complete view, it is first turned into a partial view, then all replicas in it are marked as dirty. At this point, the view can be removed when it conforms to rule 1.

Unstable updates will not be lost in normal conditions. We must ensure that both dirty and clean updates will not be lost. Dirty updates will not be lost, because they will not be evicted by the creator until they are safely on a device with a complete view. We also provide the same guarantee for clean updates on complete views. Perspective can only lose a clean update in normal conditions if the update is only contained by one complete view and a device removes that view. However, when a device removes the complete view, it will mark the replica as dirty and not evict it until it is safely on another complete view. Thus, we will not lose the update.

Next, we must verify that stable updates will receive the appropriate level of protection. A stable update meets this requirement by definition; it is contained by all complete views covering it. Further, once an update is stable, it will always remain stable. This could only cease to be the case if one added a new complete view covering the update that does not contain it. This cannot happen because a new view cannot

be complete until it has synced with another complete view covering it. Since any complete view covering a stable update contains it, the new view will also contain the update.

Epidemic update propagation assures that dirty updates will eventually spread to a complete view, thus becoming clean, and that clean updates will eventually propagate to all complete views containing them and thus become stable. In the absence of failures and complete view removal, the number of complete views containing an update will only increase over time. If a device is concerned about an update, it can hold onto the update until it is sure that the update is complete.

When a new complete view is added, it is possible that there is no complete view in the system that covers the new view. If this is the case, there are several options. First, if the system contains a set of views that collectively contain the new view, we can sync with that set of views. Alternately, we can sync with all views in the system that overlap with the new complete view. In this case, we are guaranteed to see all stable updates contained by the new view, since any stable update contained by the new view will exist in at least one complete view the overlaps with the new view.

Running out of space: It is possible for a device that hosts a complete view to run out of space while trying to store the objects that fall into the complete view. In this case, the device must convert the complete view into a partial view, and continue to store objects as is possible. In terms of reliability guarantees, this event is similar to a failure in the system; when the management tool responsible for reliability inspects the system it will notice the data no longer has the correct number of complete views covering it, and take action to correct the problem.

6 Implementation

The Perspective prototype is implemented in C++ as a user-level process. It currently runs on both Linux and Macintosh OS X. The object store is implemented by storing data in a backing filesystem and metadata in a custom XML database that is backed by a file in the filesystem. Perspective stores and exchanges metadata as XML to allow for extensible semantic naming.

The prototype currently allows for data access from a client library, which communicates with the Perspective daemon through domain sockets and shared memory. The prototype correctly propagates updates throughout the system. It implements log-based sync, with a fall back of full object exchange. It also allows for ensemble creation, disconnected operation, introduction of new views, and addition of new devices. It uses TCP sockets to transfer view manager messages and data.

The prototype does not currently support deletion, garbage collection, or view demotion (from complete to partial). These features do not affect the performance experiments, but they should all be in place before the final copy deadline.

7 Evaluation

In this section we evaluate performance with our prototype system. It shows that our prototype system has reasonable performance in the base case. It also show how views are more efficient than other ways to implement search and event notification in a distributed system.

Experimental setup.: We use three laptops, two MacBook Pros, with 1.87 GHz processors and 1 and 2 GB RAM. We also use one Linux laptop with a 1.6 GHz processor and 1 GB RAM and a Linux desktop with a 3 GHz processor and 2 GB RAM. We connect these devices to a 10Mbps half duplex wired hub to approximate home wireless bandwidth conditions.

Baseline performance: Figure 4 shows the local performance of Perspective compared with a local filesystem. Perspective has significant overhead for a pure metadata workload due to IPC calls and a user-level implementation, but for more typical data workloads Perspective introduces a reasonable 4% overhead.

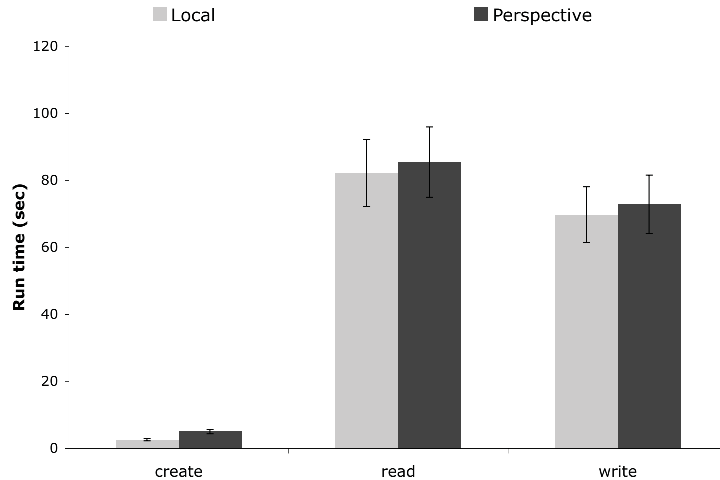

Figure 4: **Local performance.** "Create" creates 10,000 empty files, "read" reads 330 mp3s (2.4GB total), and "write" writes the same amount of data. We see a 90% overhead for a pure metadata workload, but only a 4% overhead for data workloads. This graph shows the average of 10 runs, and also shows the standard deviation.

Similarly, Figure 5 shows the overhead of Perspective in copying data from one device to another. Perspective only introduces 4% of overhead in the remote case where we copy data from one device to another. An interesting point of this experiment is that the inherent overhead of Perspective, the cost of doing sync, is a very small fraction of the overhead, meaning that most of the overhead is simply in the prototype's less efficient data transfer mechanism.

Figure 6 shows the cost of evaluating views. In these experiments we ran the same tests from Figure4 in Perspective, but varied the number of views in the system from 0 to 200. We constructed the views so that no objects matched the views, but the system would have to check the views. Each view contained two OR clauses. Even with 200 views, considerably higher than our expected number of views in a home deployment, impose negligible overhead on these tests.

Creating device ensembles: One key advantage of views is that they make creation of a device ensembles efficient. Figure 7 shows the cost for creating a device ensemble with views to a centralized approach that builds a central metadata repository. This experiment uses three laptops. As expected, views allow a basically constant ensemble creation time of .5 seconds. In contrast, the time to build a central metadata repository increases linearly with the number of objects in the system, reaching 19.9 seconds to create a device ensemble with 30,000 objects on devices other than the central metadata store. This time is spent almost exclusively in network traffic, meaning that other operations may also be difficult to perform while the store is being built.

Search and event notification: Figure 9 shows the performance of views in comparison with several approaches to implementing search and event notification. The high level outcome is that views match or beat all approaches, even the ideal case with a central server, without requiring centralization. The remainder of this section explains and discusses the graph.

Table 8 shows the alternative methods. The ideal case, in terms of bandwidth usage, is the "server" case, where there is a central server that already has a complete metadata store for the system. In this case a device only has to go to the server for any given query, and it is cheap to create the ensemble, since a central metadata store already exists. However, in a disconnected system, there will be cases where the central server is not accessible; in these cases another approach is to build a temporary central metadata store and make a device a temporary server. This approach is called "create store" in Figure9. While this still gives

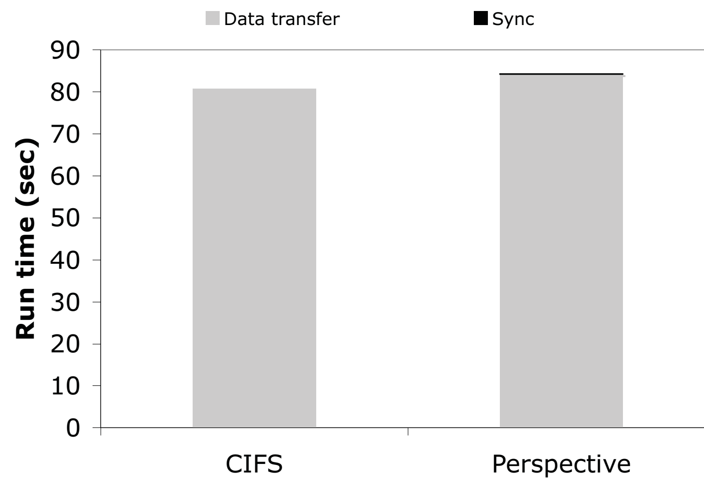

Figure 5: **Remote performance.** This test transfers 10 mp3s (64MB total) between devices. Perspective has a 4% overhead. However, the inherent protocol overhead is only .14 sec. This graph shows the average of 5 runs.

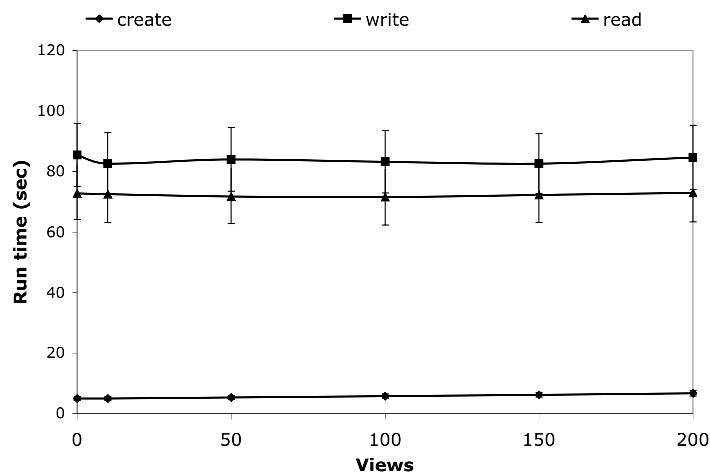

Figure 6: **View overhead.** View overhead. We expect 12-100 views in the system, at even 200 views the system has no noticeable overhead whem no data items match any view. This graph shows the average of 10 runs, and also shows the standard deviation.

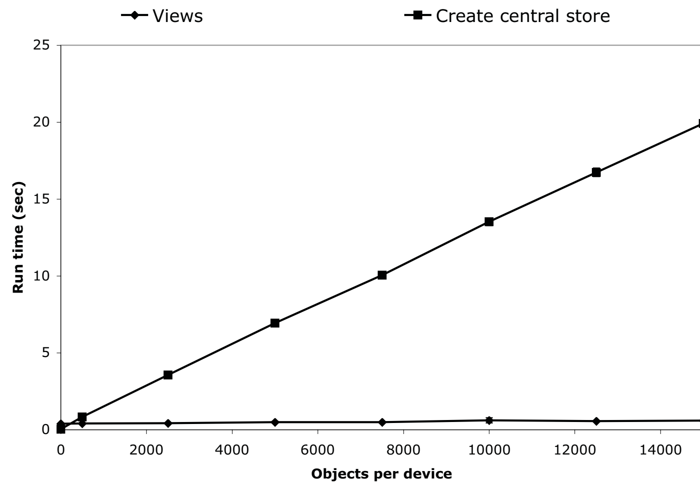

Figure 7: **Device ensemble creation.** Shows the time to create a device ensemble from three laptops. The "views" case shows the time to exchange the views and check for new updates. The "centralized" case shows the time to create a central metadata repository so that the ensemble can search and do event notification. This graph shows the cost to establish the ensemble, without having to exchange any new updates. If any new updates have been made in the system, these changes will also need to be transferred. Each data point is the average of 5 runs. Standard deviate bars are included, but not visible because they are so small.

Entry	Centralized	Decentralized
Pre-built	**Server** pre-built meta-data collection	**Local search** each device exports search interface
On demand	**Create store** create temporary metadata collection on device	**Full search** search through all metadata

Figure 8: **Search and event notification methods.**

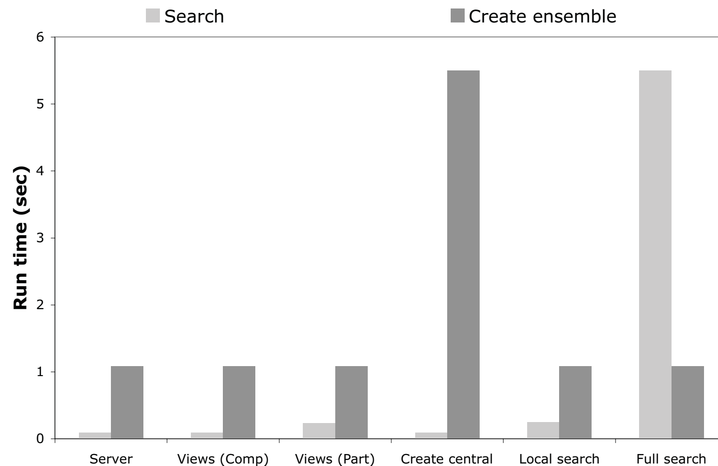

Figure 9: **Overall search performance.** Shows relative performance for creating a device ensemble, doing a search, and the number of devices that actually need to search. Views with a complete view do as well as ideal, central server, without complete view beat all other approaches. Uses 5000 total objects on four devices. The search we perform matches a total of 50 objects that are stored on two devices.

good performance for searches, it is costly to build the store at ensemble creation time.

One can avoid building a central metadata store and extend each individual device to have a search interface. One can then send searches to all devices in the system and collate the results. This case (called "local search" in Figure 9) avoids the cost of building the central store, buts adds a some extra time to searches, since searches must try all devices. However, this approaches introduces extra costs that are infeasible in the extremely heterogeneous home environment. In this approach, a cell phone participating in the network would have to handle all searches that much more powerful devices like desktop computers issued, even if the searches could not contain any of the data on the cell phone. A power and CPU limited device like a cell phone would have little chance of keeping up with the other devices.

Finally, one can perform search in a standard filesystem: by reading all the metadata in the system and searching for the correct items. While this approach (called "full search" in Figure9) avoids the cost of building a central metadata store, it is extremely costly at search time, since all searches read all metadata in the system.

In contrast, views provide efficient search without the cost of building a central metadata store and without the extra cost of sending queries to unneeded power-limited devices. If there is a complete view currently accessible that covers the given query (as shown by the "views (comp)" bars), views will act just like the central server case by noticing that the query only needs to go to that device. It thus obtains the same benefits without requiring a centralized server and without requiring any device to have a copy of all metadata in the system.

If an appropriate complete view is not available (as depicted by the "views (part)" bars), views allow the system to only query devices that could contain matching objects. This allows a power-limited device to participate in the ensemble without having to see most of the queries in the system. Of course, it is possible that the views in the system do not match the query, and we must search all devices. In this worst case scenario, views match the "local search" case.

We used Perspective to simulate the alternative systems. To do a central server approach, we use a complete view on one machine. This makes it equivalent to the "server" case because the client only sends a search to one device. To simulate the create local store case, we have one device search for all objects in

the system, and then send all queries to that device. The only extra overhead is view checking, which we have shown is negligible. For the "local search" case, we create views on the devices that do not match the query, so that the search is sent to all devices. We recreate "full search" by searching for all objects in the system; note that this is a best case, since it does not include any actual search time.

Note that we have used the "views" case as the device ensemble creation time for the "server", "full search" and "local search" cases. These values may be slightly smaller without views, since they only require device detection. However, the relative difference compared to other approaches is small. In addition, an optimized system should be able to piggyback view transfer on device detection, minimizing overhead.

This evaluation also assumes no duplicate objects. If duplicate replicas of objects exist in the system, the time to create a device collection for the create central metadata store case would increase, as would the search time for partial views and local search.

In summary Figure 9 shows the trade-offs involved in various techniques for event notification and search. The ideal "server" case allow ensemble creation in constant time, while still keeping searches dependent on the number of objects matching the search. However, a central server is not always available. In contrast, the time to create an ensemble for the "create store" case and the search time in the "full search" case are linear in the number of objects in the system, and will quickly become expensive. The "local search" approach keeps ensemble creation constant, and makes search more efficient, but requires all devices to participate in all searches, costly in a heterogeneous environment.

When a complete view is accessible, views provide identical performance to the ideal "server" case without requiring a server. If a complete view is not accessible, views still provide constant ensemble creation time and efficient search, while avoiding sending searches to unneeded devices. In the worst case, views will provide the same performance as "local search."

8 Conclusion

Views are a powerful structuring tool for home and personal data management. By concisely describing the data objects that a device may store and access, views enable a collection of devices to efficiently coordinate for consistency, search, reliability, and ad hoc ensemble creation without any central server. We believe that Perspective's use of views provides a strong foundation on which to build automated data management for home storage.

9 Acknowledgements

We would like to thank Craig Soules, Eric Toan and Michael Kaminsky for all of their help in brainstorming and refining ideas.

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] Minos Garofalakis, Kurt P. Brown, Michael J. Franklin, Joseph M. Hellerstein, Daisy Zhe Wang, Eirinaios Michelakis, Liviu Tancau, Eugene Wu, Shawn R. Jeffery, and Ryan Aipperspach. Probabilistic Data Management for Pervasive Computing: The Data Furnace Project. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. IEEE, 2006.

[4] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy. *Capability-Based Access Control for Peer-to-Peer Data Sharing*. Tech report 2006-07-02. University of Washington, July 2006.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Alexandros Karypidis and Spyros Lalis. OmniStore: A system for ubiquitous personal storage management. *IEEE International Conference on Pervasive Computing and Communications.* IEEE, 2006.

[10] 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.

[11] 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.

[12] 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.

[13] Daniel Peek and Jason Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. *Symposium on Operating Systems Design and Implementation* (Seattle, WA, 06–08 November 2006), 2006.

[14] Benjamin C. Pierce and Jerome Vouillon. *What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer*. Technical report MS-CIS-03-36. Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[15] Nuno Preguica, Carlos Baquero, J. Legatheaux Martins, Marc Shapiro, Paulo Sergio Almeida, Henrique Domingos, Victor Fonte, and Sergio Duarte. FEW: File Management for Portable Devices. *Proceedings of The International Workshop on Software Support for Portable Storage*, 2005.

[16] 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.

[17] Eric Riedel, Christos Faloutsos, Garth Gibson, and Dave Nagle. Active disks for large-scal data processing. *IEEE Computer*, pages 68–74, June 2001.

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

[19] Bill Schilit and Uttam Sengupta. Device Ensembles. *IEEE Computer*, **37**(12):56–64. IEEE, December 2004.

[20] 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.

[21] 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.