Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09).
February 24-27, 2009, San Francisco, CA.

Perspective: Semantic data management for the home

Brandon Salmon, Steven W. Schlosser*, Lorrie Faith Cranor, Gregory R. Ganger
*Carnegie Mellon University, *Intel Research Pittsburgh*

Abstract

Perspective is a storage system designed for the home, with the decentralization and flexibility sought by home users and a new semantic filesystem construct, the *view*, to simplify management. A view is a semantic description of a set of files, specified as a query on file attributes, and the ID of the device on which they are stored. By examining and modifying the views associated with a device, a user can identify and control the files stored on it. This approach allows users to reason about what is stored where in the same way (semantic naming) as they navigate their digital content. Thus, in serving as their own administrators, users do not have to deal with a second data organization scheme (hierarchical naming) to perform replica management tasks, such as specifying redundancy to increase reliability and data partitioning to address device capacity exhaustion. Experiences with Perspective deployments and user studies confirm the efficacy of view-based data management.

1 Introduction

Distributed storage is coming 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 (e.g., financial records and contact lists). In-home networks enable these devices to communicate, and a variety of device-specific and datatype-specific tools are emerging. The transition to digital homes gives exciting new capabilities to users, but it also makes them responsible for administration tasks usually handled by dedicated professionals in other settings. It is unclear that traditional data management practices will work for "normal people" reluctant to put time into administration.

This paper presents the Perspective distributed filesystem, part of an expedition into this new domain for distributed storage. As with previous expeditions into new computing paradigms, such as distributed operating systems (e.g., [23, 27]) and ubiquitous computing (e.g., [41]), we are building and utilizing a system representing the vision in order to gain experience. In this case, however, the researchers are not representative of the user population. Most will be non-technical people who just want to use the system, but must (begrudgingly) deal with administration tasks or live with the consequences. Thus, organized user studies will be required as complements to systems experimentation.

Perspective's design is motivated by a contextual analysis and early deployment experiences [31]. Our interactions with users have made clear the need for decentralization, selective replication, and support for device mobility and dynamic membership. An intriguing lesson is that home users rarely organize and access their data via traditional hierarchical naming—usually, they do so based on data attributes. Computing researchers have long talked about attribute-based data navigation (e.g., semantic filesystems [12, 36]), while continuing to use directory hierarchies. However, users of home and personal storage live it. Popular interfaces (e.g., iTunes, iPhoto, and even drop-down lists of recently-opened Word documents) allow users to navigate file collections via attributes like publisher-provided metadata, extracted keywords, and date/time. Usually, files are still stored in underlying hierarchical file systems, but users often are insulated from naming at that level and are oblivious to where in the namespace given files end up.

Users have readily adopted these higher-level navigation interfaces, leading to a proliferation of semantic data location tools [42, 3, 13, 37, 19]. In contrast, the abstractions provided by filesystems for managing files have remained tightly tied to hierarchical namespaces. For example, most tools require that specific subtrees be identified, by name or by "volumes" containing them, in order to perform *replica management tasks*, such as partitioning data across computers for capacity management or specifying that multiple copies of certain data be kept for reliability. Since home users double as their own system administrators, this disconnect between interface styles (semantic for data access activities and hierarchical for management tasks) naturally creates difficulties.

The Perspective distributed filesystem allows a collection of devices to share storage without requiring a central server. Each device holds a subset of the data and can access data stored on any other (currently connected) device. However, Perspective does not restrict the subset stored on each device to traditional volumes or subtrees. To correct the disconnect between semantic data access and hierarchical replica management, Perspective replaces the traditional volume abstraction with a new

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
FEB 2009		**00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Perspective: Semantic data management for the home	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Parallel Data Laboratory,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**16**	

primitive we call a view. A *view* is a compact description of a set of files, expressed much like a search query, and a device on which that data should be stored. For example, one view might be *"all files with type=music and artist=Beatles stored on Liz's iPod"* and another *"all files with owner=Liz stored on Liz's laptop"*. Each device participating in Perspective maintains and publishes one or more views to describe the files that it stores. Perspective ensures that any file that matches a view will eventually be stored on the device named in the view.

Since views describe sets of files using the same attribute-based style as users' other tools, view-based management replica management is easier than hierarchical file management. A user can see what is stored where, in a human-readable fashion, by examining the set of views in the system. She can control replication and data placement by changing the views of one or more devices. Views allow sets of files to overlap and to be described independently of namespace structure, removing the need for users to worry about application-internal file naming decisions or unfortunate volume boundaries. Semantic management can also be useful for local management tasks, such as setting file attributes and security, in addition to replica management. In addition to anecdotal experiences, an extensive lab study of 30 users each performing 10 different management tasks confirms that view-based management is easier for users than volume-based management.

This paper describes view-based management and our Perspective prototype, which combines existing technologies with several new algorithms to implement view-based distributed storage. View-based data placement and view freshness allow Perspective to manage and expose data mobility with views. Distributed update rules allow Perspective to ensure and expose permanence with views (which can be thought of as semantically-defined volumes). Perspective introduces *overlap trees* as a mechanism for reasoning about how many replicas exist of a particular dataset, and where these files are stored, even when no view exactly matches the attributes of the dataset.

Our Perspective prototype is a user-level filesystem which runs on Linux and OS X. In our deployments, Perspective provides normal file storage as well as being the backing store for iTunes and MythTV in one household and in our research environment lounge. Experiments with the Perspective prototype confirm that it can provide consistent, decentralized storage with reasonable performance. Even with its application-level implementation (connected to the OS via FUSE [10]), Perspective performance is within 3% of native filesystem performance for activities of interest.

2 Storage for the home

Storage has become a component of many consumer electronics. Currently, most stored content is captive to individual devices (e.g., DVRs, digital cameras, digital picture frames, and so on), with human-intensive and proprietary interfaces (if any) for copying it to other devices. But, we expect a rapid transition toward exploiting wireless home networking to allow increased sharing of content across devices. Thus, we are exploring how to architect a distributed storage system for the home.

The home is different from an enterprise. Most notably, there are no sysadmins—household members generally deal with administration (or don't) themselves. The users also interact with their home storage differently, since most of it is for convenience and enjoyment rather than employment. However, much of the data stored in home systems, such as family photos, is both important and irreplaceable, so home storage systems must provide high levels of reliability in spite of lax management practices. Not surprisingly, we believe that home storage's unique requirements would be best served by a design different than enterprise storage. This section outlines insights gained from studying use of storage in real homes and design features suggested by them.

2.1 What users want

A contextual analysis is an HCI research technique that provides a wealth of in-situ data, perspectives, and real-world anecdotes of the use of technology. It consists of interviews conducted in the context of the environment under study. To better understand home storage, we extensively interviewed all members of eight households (24 people total), in their homes and with all of their storage devices present. We have also gathered experiences from early deployments in real homes. This section lists some guiding insights (with more detailed information available in technical reports [31]).

Decentralized and Dynamic: The users in our study employed a wide variety of computers and devices. While it was not uncommon for them to have a set of primary devices at any given point in time, the set changed rapidly, the boundaries between the devices were porous, and different data was "homed" on different devices with no central server. One household had set up a home server, at one point, but did not re-establish it when they upgraded the machine due to setup complexity.

Money matters: While the cost of storage continues to decrease, our interviews showed that cost remains a critical concern for home users. Note that our studies were conducted well before the Fall 2008 economic crisis. While the same is true of enterprises, home stor-

age rarely has a clear "return on investment," and the cost is instead balanced against other needs (e.g., new shoes for the kids) or other forms of enjoyment. Thus, users replicate selectively, and many adopted cumbersome data management strategies to save money.

Semantic naming: Most users navigated their data via attribute-based naming schemes provided by their applications, such as iPhoto, iTunes, and the like. Of course, these applications stored the content into files in the underlying hierarchical file system, but users rarely knew where. This disconnect created problems when they needed to make manual copies or configure backup/synchronization tools.

Need to feel in control: Many approaches to manageability in the home tout automation as the answer. While automation is needed, the users expressed a need to understand and sometimes control the decisions being made. For example, only 2 of the 14 users who backed up data used backup tools. The most commonly cited reason was that they did not understand what the tool was doing and, thus, found it more difficult to use the tool than to do the task by hand.

Infrequent, explicit data placement: Only 2 of 24 users had devices on which they regularly placed data in anticipation of needs in the near future. Instead, most users decided on a type of data that belonged on a device (e.g., "all my music" or "files for this semester") and rarely revisited these decisions, usually only when prompted by environmental changes. Many did regularly copy new files matching each device's data criteria onto it.

2.2 Designing home storage

From the insights above, we extract guidance that has informed our design of Perspective.

Peer-to-peer architecture: While centralization can be appealing from a system simplicity standpoint, and has been a key feature in many distributed filesystems, it seems to be a non-starter with home users. Not only do many users struggle with the concept of managing a central server, many will be unwilling to invest the money necessary to build a server with sufficient capacity and reliability. We believe that a decentralized, peer-to-peer architecture more cleanly matches the realities we encountered in our contextual analysis.

Single class of replicas: Many previous systems have differentiated between two classes: permanent replicas stored on server devices and temporarily replicas stored on client devices (e.g., to provide mobility) [32, 25]. While this distinction can simplify system design, it introduces extra complexity for users, and prevents users from utilizing the capacity on client devices for reliability, which can be important for cost-conscious home consumers. Having only a single replica class removes the client-server distinction from the user's perception and allows all peers to contribute capacity to reliability.

Semantic naming for management: Using the same type of naming for both data access and management should be much easier for users who serve as their own administrators. Since home storage users have chosen semantic interfaces for data navigation, replica management tools should be adapted accordingly—users should be able to specify replica management policies applied to sets of files identified by semantic naming.

In theory, applications could limit the mismatch by aligning the underlying hierarchy to the application representation. But, this alternative seems untenable, in practice. It would limit the number of attributes that could be handled, lock the data into a representation for a particular application, and force the user to sort data in the way the application desires. Worse, for data shared across applications, vendors would have to agree on a common underlying namespace organization.

Rule-based data placement: Users want to be able to specify file types (e.g., "Jerry's music files") that should be stored on particular devices. The system should allow such rules to be expressed by users and enforced by the system as new files are created. In addition to helping users to get the right data onto the right devices, such support will help users to express specific replication rules at the right granularity, to balance their reliability and cost goals.

Transparent automation: Automation can simplify storage management, but many home users (like enterprise sysadmins) insist on understanding and being able to affect the decisions made. By having automation tools use the same flexible semantic naming schemes as users do normally, it should be possible to create interfaces that express human-readable policy descriptions and allow users to understand automated decisions.

3 Perspective architecture

Perspective is a distributed filesystem designed for home users. It is decentralized, enables any device to store and access any data, and allows decisions about what is stored where to be expressed or viewed semantically.

Perspective provides flexible and comprehensible file organization through the use of *views.* A view is a concise description of the data stored on a given device. Each view describes a particular set of data, defined by a semantic query, and a device on which the data is stored. A view-based replica management system guarantees that

any object that matches the view query will eventually be stored on the device named in the view. We will describe our query language in detail in Section 4.1.

Figure 1 illustrates a combination of management tools and storage infrastructure that we envision, with views serving as the connection between the two layers. Users can set policies through management tools, such as those described in Section 5, from any device in the system at any time. Tools implement these changes by manipulating views, and the underlying infrastructure (Perspective) in turn enforces those policies by keeping files in sync among the devices according to the views. Views provide a clear division point between tools that allow users to manage data replicas and the underlying filesystem that implements the policies.

View-based management enables the design points outlined in Section 2.2. Views provide a primitive allowing users to specify meaningful rule-based placement policies. Because views are semantic, they unify the naming used for data access and data management. Views are also defined in a human-understandable fashion, providing a basis for transparent automation. Perspective provides data reliability using views without restricting their flexibility, allowing it to use a single replica class.

3.1 Placing file replicas

In Perspective, the views control the distribution of data among the devices in the system. When a file is created or updated, Perspective checks the attributes of the file against the current list of views in the system and sends an update message to each device with a view that contains that file. Each device can then independently pull a copy of the update.

When a device, A, receives an update message from another device, B, it checks that the updated file does, indeed, match one or more views that A has registered. If the file does match, then A applies the update from B. If there is no match, which can occur if the attributes of a file are updated such that it is no longer covered by a view, then A ensures that there is no replica of the file stored locally.

This simple protocol automatically places new files, and also keeps current files up to date according to the current views in the system. Simple rules described in Section 4.3 assure that files are never dropped due to view changes.

Each device is represented by a file in the filesystem that describes the device and its characteristics. Views themselves are also represented by files. Each device registers a view for all device and view files to assure they are replicated on all participating devices. This allows applications to manage views through the standard filesystem interfaces, even if not all devices are currently present.

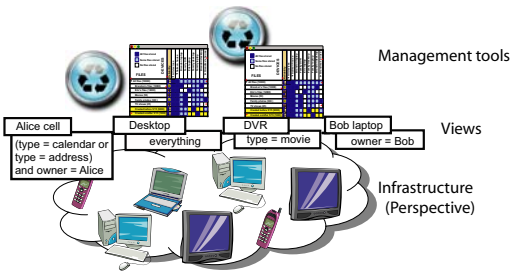

Figure 1: View-based architecture. Views are the interface between management tools and the underlying heterogeneous, disconnected infrastructure. By manipulating the views, tools can specify data policies that are then enforced by the filesystem.

3.2 View-based data management

This subsection presents three scenarios to illustrate view-based management.

Traveling: Harry is visiting Sally at her house and would like to play a new U2 album for her while he is at her house. Before leaving, he checks the views defined on his wireless music player and notices the songs are not stored on the device, though he can play them from his laptop where they are currently stored. He asks the music player to pull a copy of all U2 songs, which the player does by creating a new view for this data. When the synchronization is complete, the filesystem marks the view as complete, and the music player informs Harry.

He takes the music player over to Sally's house. Because the views on his music player are defined only for his household and the views on Sally's devices for her household, no files are synchronized. But, queries for "all music" initiated from Sally's digital stereo can see the music files on Harry's music player, so they can listen to the new U2 album off of Harry's music player on the nice stereo speakers, while he is visiting.

Crash: Mike's young nephew Oliver accidentally pushes the family desktop off of the desk onto the floor and breaks it. Mike and his wife Carol have each configured the system to store their files both on their respective laptop and on the desktop, so their data is safe. When they set up the replacement computer, a setup tool pulls the device objects and views from other household devices. The setup tool gives them the option to replace an old device with this computer, and they choose the old desktop from the list of devices. The tool then creates views on the device that match the views on the old desktop and deletes the device object for the old computer. The data from Mike and Carol's laptops is transferred to the new

desktop in the background over the weekend.

Short on space: Marge is working on a project for work on her laptop in the house. While she is working, a capacity automation tool on her laptop alerts her that the laptop is short on space. It recommends that files created over two years ago be moved to the family desktop, which has spare space. Marge, who is busy with her project, decides to allow the capacity tool to make the change. She later decides to keep her older files on the external hard drive instead, and makes the change using a view-editing interface on the desktop.

4 Perspective design

This section details three aspects of Perspective: semantic search and naming, consistent partial replication of sets of files, and reliability maintenance and reasoning.

The Perspective prototype is implemented in C++ and runs at user-level using FUSE [10] to connect with the system. It currently runs on both Linux and Macintosh OS X. Perspective stores file data in files in a repository on the machine's local filesystem and metadata in a SQLite database with an XML wrapper. Our prototype implements all of the features described in this paper except garbage collection and some advanced update log features.

The prototype system has supporting one researcher's household's DVR, which is under heavy use; it is the exclusive television for him and his four roommates, and is also frequently used by sixteen other friends in the same complex. It has also stored one researcher's personal data for about a year. It has also been the backing store for the DVR in the lounge for our research group for several months. We are preparing the system for deployment in several non-technical households for a wider, long-term user study over several months.

4.1 Search and naming

All naming in Perspective uses semantic metadata. Therefore, search is a very common operation both for users and for many system operations. Metadata queries can be made from any device and Perspective will return references to all matching files on devices currently accessible (e.g., on the local subnet), which we will call the current *device ensemble* [33]. Views allow Perspective to route queries to devices containing all needed files and, when other devices suffice, avoid sending queries to power-limited devices. While specialized applications may use the Perspective API directly, we expect most applications to access files through the standard VFS layer, just as they access other filesystems. Perspective provides this access using *front ends* that support a variety of user-facing naming schemes. These names are then converted to Perspective searches, which are then passed on to the filesystem. Our current prototype system implements four front ends that each support a different organization: directory hierarchies, faceted metadata, simple search, and hierarchies synthesized from the values of specific tags.

Query language and operations: We use a query language based on a subset of the XPath language used for querying XML. Our language includes logic for comparing attributes to literal values with equality, standard mathematical operators, string search, and an operator to determine if a document contains a given attribute. Clauses can be combined with the logical operators *and*, *or*, and *not*. Each attribute is allowed to have a single value, but multi-value attributes can be expressed in terms of single value attributes, if necessary. We require all comparisons to be between attributes and constant values.

In addition to standard queries, we support two operations needed for efficient naming and reliability analysis. The first is the *enumerate values* query, which returns all values of an attribute found in files matching a given query. The second is the *enumerate attributes* query, which returns all the unique attributes found in files matching a given query. These operations must be efficient; fortunately we can support them at the database level using indices, which negate the need for full enumeration of the files matching the query.

This language is expressive enough to capture many common data organization schemes (e.g., directories, unions [27], faceted metadata [43], and keyword search) but is still simple enough to allow Perspective to efficiently reason about the overlap of queries. Perspective can support any of the replica management functions described in this paper for any naming scheme that can be converted into this language.

Overlap evaluation is commonly used to compare two queries. The overlap evaluation operation returns one of three values when applied to two queries: one query *subsumes* the other, the two queries have *no-overlap*, or the relationship between them is *unknown*. Note that the comparison operator is used for efficiency but not correctness, allowing for a trade-off between language complexity and efficiency. For example, Perspective can determine that the query *all files where date* $<$ *January, 2008* is subsumed by the query *all files where date* $<$ *June, 2008*, and that the query *all files where owner=Brandon* does not overlap with the query *all files where owner=Greg*. However, it cannot determine the relationship between the queries *all files where*

type=Music and *all files where album=The Joshua Tree*. Perspective will correctly handle operations on the latter two queries, but at some cost in efficiency.

4.2 Partial replication

Perspective supports partial replication among the devices in a home. Devices in Perspective can each store disjoint sets of files — there is no requirement that any master device store all files or that any device mirror another in order to maintain reliability. Previous systems have supported either partial replication [16, 32] or topology independence [40], but not both. PRACTI [7] provided a combination of the two properties tied to directories, but probably could be extended to work in the semantic case. Recently, Cimbiosis [28] has also provided partial replication with effective topology independence, although it requires all files to be stored on some master device. We present Perspective's algorithms to show that it is possible to build a simple, efficient consistency protocol for a view-based system, but a full comparison with previous techniques is beyond the scope of this paper. The related work section presents the differences and similarities with previous work.

Synchronization: Devices that are not currently accessible at the time of an update will receive that update at synchronization time, when the two devices exchange information about updates that they may have missed. Device and view files are always synchronized before other files, to make sure the device does not miss files matching new views. Perspective employs a modified update log to limit the exchanges to only the needed information, much like the approach used in Bayou [40]. However, the flexibility of views makes this task more challenging.

For each update, the log contains the metadata for the file both before and after the update. Upon receiving a sync request, a device returns all updates that match the views for the calling device either before or after the update. As in Bayou, the update log is only an optimization; we can always fall back on full file-by-file synchronization.

Conventional full synchronization can be problematic for heterogeneous devices with partial replication, especially for resource- and battery-limited devices. For example, if a cell phone syncs with a desktop computer, it is not feasible for the cell phone to process all of the files on the desktop, even occasionally. To address this problem, Perspective includes a second synchronization option. Continuing the example, the cell phone first asks the desktop how many updates it would return. If this number is too large, the cell phone can pass the metadata of all the files it owns to the desktop, along with the view query, and ask the desktop for updates for any files that match the view or are contained in the set of files currently on the cell phone. At each synchronization, the calling device can choose either of these two methods, reducing synchronization costs to O($N_{smaller}$), where $N_{smaller}$ is the number of files stored on the smaller device.

Full synchronizations will only return the most recent version of a file, which may cause gaps in the update logs. If the update log has a gap in the updates for a file, recognizable by a gap in the versions of the before and after metadata, the calling device must pass this update back to other devices on synchronization even if the metadata does not match the caller's views, to avoid missing updates to files which used to match a view, but now do not.

Consistency: As with many file systems that support some form of eventual consistency, Perspective uses version vectors and epidemic propagation to ensure that all file replicas eventually converge to the same version. Version vectors in Perspective are similar to those used in many systems; the vector contains a version for each replica that has been modified. Because Perspective does not have the concept of volumes, it does not support volume-level consistency like Bayou. Instead, it supports file-level consistency, like FICUS [16].

To keep all file replicas consistent, we need to assure that updates will eventually reach all replicas. If all devices in the household sync with one another occasionally, this property will be assured. While this is a reasonable assumption in many homes, we do not require full pairwise device synchronization. Like many systems built on epidemic propagation, a variety of configurations satisfy this property. For example, even if some remote device (e.g., a work computer) never returns home, the property will still hold as long as some other device that syncs with the remote device and does return home (e.g., a laptop) contains all the data stored on the remote device. System tools might even create views on such devices to facilitate such data transfer, similar to the routing done in Footloose [24]. Alternately, a sync tree, as that used in Cimbiosis [28] could be layered on top of Perspective to provide connectedness guarantees.

By tracking the timestamps of the latest complete sync operation for each device in the household, devices provide a *freshness timestamp* for each view. Perspective can guarantee that all file versions created before the freshness timestamp for a view are stored on that view's device. It can also recommend sync operations needed to advance the freshness timestamp for any view.

Conflicts: Any system that supports disconnected operation must deal with *conflicts*, where two devices modify the same file without knowledge of the other device's modification. We resolve conflicts first with a *preresolver*, which uses the metadata of the two versions to

deterministically choose a winning and losing version. Our pre-resolver can be run on any device without any global agreement. It uses the file's modification time and then the sorted version vector in the case of a tie. But, instead of eliminating the losing version, the pre-resolver creates a new file, places the losing version in this new file. It then tags the new file with all metadata from the losing version, as well as tags marking it as a conflict file and tying it to the winning version. Later, a *full resolver*, which may ask for user input or use more sophisticated logic, can search for conflict objects, remove duplicates, and adjust the resolution as desired.

Capacity management: Pushing updates to other devices can be problematic if those devices are at full capacity. In this case, the full device will refuse subsequent updates, and mark the device file noting that the device is out of space. Until a user or tool corrects the problem, the device will continue to refuse updates, although other devices will be able to continue. However, management tools built on top of Perspective should help users address capacity problems before they arise.

File deletion: As in many other distributed filesystems, when a file is removed, Perspective keeps a *tombstone* marker that assures all replicas of the file in the system are deleted, but is ignored by all naming operations. Perspective uses a two-phase garbage collection mechanism, like that used in FICUS, between all devices with views that match the file to which the tombstone belongs. Note that deletion of a file removes all replicas of a file in the system, which is a different operation from dropping a particular replica of a file (done by manipulating views). This distinction also exists in any distributed filesystem allowing replication.

View and device objects: Each device is only required to store view and device objects from devices that contain replicas of files it stores, although they must also temporarily store view and device files for devices in the current ensemble in order to access their files. Because views are very small (hundreds of bytes), this is tractable, even for small devices like cell phones.

4.3 Reliability with partial replication

In order to manage data semantically, users must be able to provide fault-tolerance on data split semantically across a distributed set of disconnected, eventually-consistent devices. Perspective enables semantic fault-tolerance through two new algorithms, and provides a way to efficiently reason about the number of replicas of arbitrary sets of files. It also assures that data is never lost despite arbitrary and disconnected view manipulation using three simple distributed update rules.

Reasoning about number of replicas: Reasoning about the reliability of a storage system — put simply, determining the level of replication for each data item — is a challenge in a partially-replicated filesystem. Since devices can store arbitrary subsets of the data, there are no simple rules that allow all of the replicas to be counted. A naïve solution would be to enumerate all of the files on each device and count replicas. Unfortunately, this would be prohibitively expensive and would be possible only if all devices are currently accessible.

Fortunately, Perspective's views compactly and fully describe the location of files in terms of their attributes. Since there are far fewer views than there are file replicas in the system, it is cheaper to reason about the number of times a particular query is replicated among all of the views in the system than to enumerate all replicas. The files in question could be replicated exactly (e.g., all of the family's pictures are on two devices), they could be subsumed by multiple views (e.g., all files are on the desktop and all pictures are on the laptop), or they could be replicated in part on multiple devices but never in full on any one device (e.g., Alice's pictures are on her laptop and desktop, while Bob's pictures are on his laptop and desktop – among all devices, the entire family's pictures have two replicas).

To efficiently reason about how views overlap, Perspective uses *overlap trees.* An overlap tree encapsulates the location of general subsets of data in the system, and thus simplifies the task of determining the location of the many data groupings needed by management tools. An overlap tree is currently created each time a management application starts, and then used throughout the application's runtime to answer needed overlap queries.

Overlap trees are created using the enumeration queries described in Section 4.1. Each node contains a query, that describes the set of data the node represents. Each leaf node represents a subset of files whose location can be precisely quantify using the views and records the devices that store that subset. Each interior node of the tree encodes a subdivision of the attribute space, and contains a list of child nodes, each of which represents a subset of the files that the parent node represents. We begin building the tree by enumerating all of the attributes that are used in the views found in the household.

We create a root node for the tree to represent all files, choose the first attribute in our attribute list, and use the enumerate values query to find all values of this attribute for the current node's query. We then create a child node from each value with a query of the form *<parent query> and attribute=value*. We compare the query for each child node against the complete views on all devices. If the compare operator can give a precise answer

(i.e., not *unknown*) for whether the query for this node is stored on each device in the home, then this node is a leaf and we can stop dividing. Otherwise, we recursively perform this operation on the child node, dividing it by the next attribute in our list. Figure 2 shows an example overlap tree. The ordering of the attribute list could be optimized to improve performance of the overlap tree, but in the current implementation we leave it unordered.

When we create an overlap tree, we may not have all the information needed to construct the tree. For example, if we currently only have access to Brian's files, we may incorrectly assume that all music files in the household are owned by Brian, when music files owned by Mary exist elsewhere in the system. The tree construction mechanism makes a notation in a node if it cannot guarantee that all matching files are available via the views. When checking for overlaps, if a marked node is required the tree will return an *unknown* value, but it will still correctly compute overlaps for queries that do not require these nodes. To avoid this restriction, devices are free to cache and update an overlap tree, rather than recreating the overlap tree when each management application starts. The tree is small, making caching it easy. To keep it up to date, a device can publish a view for all files, and then use the updates to keep the cached tree up to date.

Once we have constructed the overlap tree, we can use it to determine the location and number of full copies in the system of the files for any given query. Because the tree caches much of the overlap processing, each individual query request can be processed efficiently. We do so by traversing all of the leaf nodes and finding those that overlap with the given view or query. We may occasionally need to perform more costly overlap detection, if the attribute in a leaf node does not match any of the attributes in the query. For example, in the overlap tree in Figure 2, if we were checking to see if the query *album=Joshua Tree* was contained in the node *owner=Mary and type=Music* we would use the enumerate values query to determine the values of "type" for the query *album=Joshua Tree and owner=Mary*. If "Music" is the only value, then we can count this node as a full match in our computations. Otherwise, we cannot. This extra comparison is only valid if we can determine via the views that all files in the query for which we are computing overlaps are accessible

Attributes with larger cardinalities can be handled more efficiently by selectively expanding the tree. For example, if a view is defined on $date < T$, we need only expand the attribute date into three sub-nodes, one for $date < T$, one for $date \geq T$, and one for *has no date attribute*.

Note that the number of attributes used in views at any one time is likely to be much smaller than the total number of attributes in the system, and both of these will be much smaller than the total number of files or replicas. For example, in our contextual analysis, most households described settings requiring around 20 views and 5 attributes. None of households we interviewed described more than 30 views, or more than 7 attributes. Because the number of relevant attributes is small, overlap tree computations are fast enough to allow us to compute them in real time as the user browses files. We will present a performance comparison of overlap trees to the naïve approach in Section 6.

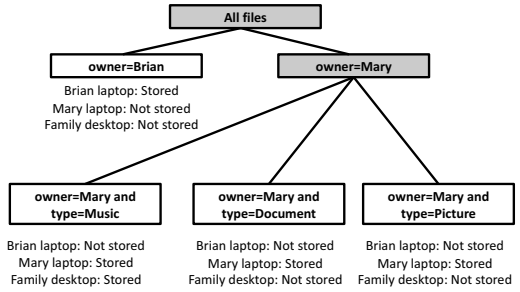

Figure 2: Overlap tree. This figure shows an example overlap tree, constructed from a three-device, three-view scenerio: Brian's files stored on Brian's laptop, Mary's files stored on Mary's laptop, and Mary's music stored on the Family desktop. Shaded nodes are interior nodes, unshaded nodes are leaf nodes. Each leaf node lists whether this query is stored on each device the household.

Update rules: Perspective maintains permanence by guaranteeing that files will never be lost by changes to views or addition or removal of devices, regardless of the order, timing, or origin of the changes, freeing the user from worrying about these problems when making view changes. We also provide a guarantee that, once a version of a file is stored on the devices associated with all overlapping views, it will always be stored in all overlapping views, which provides a strong assurance on the number of copies in the system based on the current views. View freshness timestamps, as described in Section 4.2, allow Perspective to guarantee that all updates created before a given timestamp are safely stored in the correct locations, and thus have the fault-tolerance implied by the views. These guarantees are assured by careful adherence to three simple rules: 1) When a file replica is modified by a device, it is marked as "modified." Devices cannot evict modified replicas. Once a modified replica has been pulled by a device holding a view covering it, the file can be marked as unmodified and then removed. 2) A newly created view cannot be considered complete until it has synced with all devices with overlapping views or synced with one device with a view that subsumes the new view. 3) When a view is removed, all replicas in it are marked as modified. The replicas are then removed when they conform to rule 1.

These rules ensure that devices will not evict modified replicas until they are safely on some "stable" location (i.e., in a completely created view). The rules also assure that a device will not drop a file until it has confirmed that another up-to-date replica of the file exists somewhere in the system. However, a user can force the system to drop a file replica without assuring another replica exists, if she is confident that another replica exists and is willing to forgo this system protection. With these rules, Perspective can provide permanence guarantees without requiring central control or limiting when or where views can be changed.

4.4 Security and cross-household sharing

Security is not a focus in this paper, but is certainly a concern for users and system designers alike. While Perspective does not currently support it, we envision using mechanisms such as those promoted by the UIA project [8]. Our current prototype supports voluntary access control using simplified access control lists. While all devices are able to communicate and share replicas with one another, even aside from security concerns it is helpful to divide households from one another to divide management and view specification. To do so, Perspective maintains a household ID for each device and each file. Views are specified on files within the given household, to avoid undesired cross-syncing. However, the fundamental architecture of Perspective places no restrictions on how these divisions are made.

5 View manager interface

To explore view-based management, we built a *view manager* tool to allow users to manipulate views.

Customizable faceted metadata: One way of visualizing and accessing semantic data is through the use of *faceted metadata* [43]. Faceted metadata allows a user to choose a first attribute to use to divide the data and a value at which to divide. Then, the user can choose another attribute to divide on, and so on. Faceted metadata helps users browse semantic information by giving them the flexibility to divide the data as needed. But, it can present the user with a dizzying array of choices in environments with large numbers of attributes.

To curb this problem, we developed *customizable faceted metadata* (CFM), which exposes a small user-selected set of attributes as directories plus one additional *other groupings* directory that contains a full list of possible attributes. The user can customize which attributes are displayed in the original list by moving folders between the base directory and the *other groupings* directory. These preferences are saved in a customization object in the filesystem. The file structure on the left side of the interface in Figure 3 illustrates CFM. Perspective exposes CFM through the VFS layer, so it can be accessed in the same way as a normal hierarchical filesystem.

View manager interface: The view manager interface (Figure 3), allows users to create and delete views on devices and to see the effects of these actions. This GUI is built in Java and makes calls into the view library of the underlying filesystem.

The GUI is built on Expandable Grids [29], a user interface concept initially developed to allow users to view and edit file system permissions. Each row in the grid represents a file or file group, and each column represents a device in the household. The color of a square represents whether the files in the row are stored on the device in the column. The files can be "all stored" on the device, "some stored" on the device, or "not stored" on the device. Each option is represented by a different color in the square. By clicking on a square a user can add or remove the given files from the given device. Similarly to file permissions, this allows users to manipulate actual storage decisions, instead of rule lists.

An extra column, labeled "Summary of failure protection," shows whether the given set of files is protected from one failure or not, which is true if there are at least two copies of each file in the set. By clicking on an unbacked-up square, the user can ask the system to assure that two copies of the files are stored in the system, which it will do by placing any extra needed replicas on devices with free space.

An extra row contains all unique views and where they are stored, allowing a user to see precisely what data is stored on each device at a glance.

6 Evaluation

Our experience from working with many home storage users suggests that users are very concerned about the time and effort spent managing their devices and data at home, which has motivated our design of Perspective, as well as our evaluation. Therefore, we focus our study primarily on the usability of Perspective's management capabilities and secondarily on its performance overhead.

We conducted a lab study in which non-technical users used Perspective, outfitted with appropriate user interfaces, to perform home data management tasks. We measured accuracy and completion time of each task. In order to insulate our results as much as possible from the particulars of the user interface used for each primitive, we built similar user interfaces for each primitive using

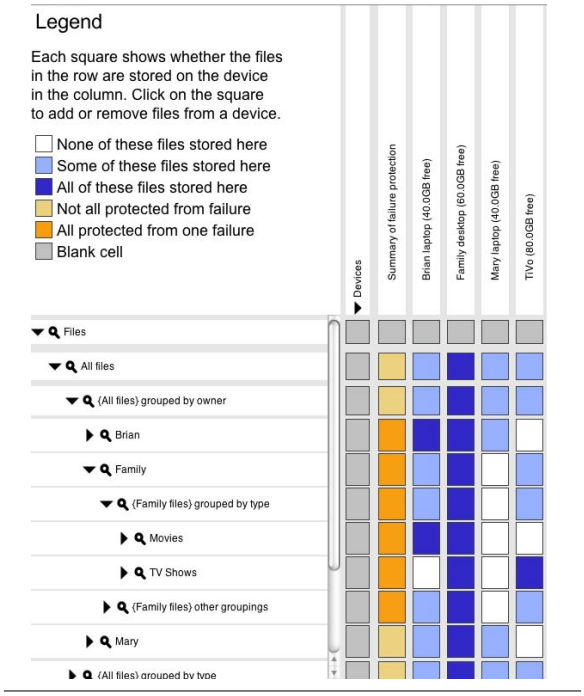

Figure 3: View manager interface. A screen shot of the view manager GUI. On the left are files, grouped using faceted metadata. Across the top are devices. Each square shows whether the files in the row are stored on the device in the column.

the Expandable Grids UI toolkit [29].

Views-facet interface: The views-facet interface was described in Section 5. It uses CFM to describe data, and allows users to place any set of data described by the faceted metadata on any device in the home.

Volumes interface: This user interface represents a similar interface built on top of a more conventional volume-based system with directory hierarchies. Each device is classified as a client or server, and this distinction is listed in the column along with the device name. The volumes abstraction only allows permanent copies of data to be placed on servers, and it restricts server placement policies on volume boundaries. We defined each root level directory (based on user) as a volume. The abstraction allows placement of a copy of any subtree of the data on any client device, but these replicas are only temporary caches and are not guaranteed to be permanent or complete. The interface distinguishes between temporary and permanent replicas by color. The legend displays a summary of the rules for servers and permanent data and for clients and temporary data.

Views-directory interface: To tease apart the effects of semantic naming and using a single replica class, we evaluated an intermediate interface, which replaces the CFM organization with a traditional directory hierarchy. Otherwise, it is identical to the views-facet interface. In particular, it allows users to place any subtree of the hierarchy on any device.

6.1 Experiment design

Our user pool consisted of students and staff from nearby universities in non-technical fields who stated that they did not use their computers for programming. We did a *between-group* comparison, with each participant using one of the three interfaces described above. We tested 10 users in each group, for a total of 30 users overall. The users performed a *think-aloud* study in which they spoke out loud about their current thoughts and read out loud any text they read on the screen, which provides insight into the difficulty of tasks and users' interpretation. All tasks were performed in a *latin square* configuration, which guarantees that every task occurs in each position in the ordering, and each task is equally likely to follow any other task.

We created a filesystem with just over 3,000 files, based on observations from our contextual analysis. We created a setup with two users, Mary and Brian, and a third "Family" user with some shared files. We modeled Brian's file layout on the Windows music and pictures tools and Mary's on Apple's iTunes and iPhoto file trees. Our setup included four devices: two laptops, a desktop, and a DVR. We also provided the user with iTunes and iPhoto, with the libraries filled with all of the matching data from the filesystem. This allowed us to evaluate how users convert from structures in the applications to the underlying filesystem.

6.2 Tasks

Each participant performed the same set of tasks, which we designed based on our contextual analysis. We started each user with a 5 to 10 minute training task, after which our participants performed 10 data management tasks. While space constraints preclude us including the full text for all of them, as we discuss each class of tasks, we include the text of one example task. For this study, we chose tasks to illustrate the differences between the approaches. A base-case task that was similar in all interfaces confirmed that, on these similar tasks, all interfaces performed similarly. The tasks were divided into two types: single replica tasks, and data organization tasks.

Single replica tasks: Two single replica tasks (LH and CB) required the user to deal with distinctions between permanent and temporary replicas to be successful.

Example task, Mary's laptop comes home (LH): *"Mary has not taken her laptop on a trip with her for a while now, so she has decided to leave it in the house and make*

an extra copy of her files on it, in case the Family desktop fails. However, Brian has asked her not to make extra copies of his files or of the Family files. Make sure Mary's files are safely stored on her laptop."

Mary's laptop was initially a client in the volume case. This task asked the user to change it to a server before storing data there. This step was not required for the single replica class interfaces, as all devices are equivalent.

Note that because server/client systems, unlike Perspective, are designed around non-portable servers for simplicity, it is not feasible to simply make all devices servers. Indeed, the volume interface actually makes this task much simpler than current systems; in the volume interface, we allow the user to switch a device from server to client using a single menu option, where current distributed filesystems require an offline device reformat.

Data organization tasks: The data organization tasks required users to convert from structures in the iTunes and iPhoto applications into the appropriate structures in the filesystem. This allowed us to test the differences between a hierarchical and semantic, faceted systems. The data organization tasks are divided into three types: aggregation, comprehension, and sparse collection tasks.

Aggregation: One major difference between semantic and hierarchical systems is that because the hierarchy forces a single tree, tasks that do not match the current tree require the user to aggregate data from multiple directories. This is a natural case as homes fill with aggregation devices and data is shared across users and devices. However, in a hierarchical system, it is difficult for users to know all the folders that correspond to a given application grouping. Users often erroneously assumed all the files for a given collection were in the same folder. The semantic structure mitigates this problem, since the user is free to use a filesystem grouping suited to the current specific task.

Example task, U2 (U2): *"Mary and Brian share music at home. However, when Mary is on trips, she finds that she can't listen to all the songs by U2 on her laptop. She doesn't listen to any other music and doesn't want other songs taking up space on her laptop, but she does want to be able to listen to U2. Make sure she can listen to all music by the artist U2 on her trips."*

As may often be the case in the home, the U2 files were spread across all three user's trees in the hierarchical interfaces. The user needed to use iTunes to locate the various folders. The semantic system allowed the user to view all U2 files in a single grouping.

Aggregation is also needed when applications sort data differently from what is needed for the current task. For example, iPhoto places modified photos in a separate folder tree from originals, making it tricky for users to get all the files for a particular event. The semantic structure allows applications to set and use attributes, while allowing the user to group data as desired.

Example task, Rafting (RF): *"Mary and Brian went on a rafting trip and took a number of photos, which Mary remembers they labeled as 'Rafting 2007'. She wants to show her mother these photos on Mary's laptop. However, she doesn't want to take up space on her laptop for files other than the 'Rafting 2007' files. Make sure Mary can show the photos to her mother during her visit."*

The rafting photos were initially in Brian's files, but iPhoto places modified copies of photos in a separate directory in the iPhoto tree. To find both folders, the user needed to explore the group in iPhoto. The semantic system allows iPhoto to make the distinction, while allowing the user to group all files from this roll in one grouping.

Comprehension: Applications can allow users to set policies on application groupings, and then convert them into the underlying hierarchy. However, in addition to requiring multiple implementations and methods for the same system tasks, this leads to extremely messy underlying policies, which make it difficult for users to understand, especially when viewing it from another application. In contrast, semantic systems can retain a description of the policy as specified by the application, making them easier for users to understand.

Example task, Traveling Brian (TB): *"Brian is taking a trip with his laptop. What data will he be able to access while on his trip? You should summarize your answer into two classes of data."*

Brian's laptop contained all of his files and all of the music files in the household. However, because iTunes places TV shows in the Music repository, the settings included all of the music subfolders, but not the "TV Shows" subfolder, causing confusion. In contrast, the semantic system allows the user to specify both of these policies in a single view, while still allowing applications to sort the data as needed.

Note that this particular task would be simpler if iTunes chose to sort its files differently, but the current iTunes organization is critical for other administrative tasks, such as backing up a user's full iTunes library. It is impossible to find a single hierarchical grouping that will be suited to all needed operations. This task illustrates how these kinds of mismatches occur even for common tasks and well-behaved applications.

Sparse collection: Two sparse collection tasks (BF and HV) required users to make policies on collections that contain single files from across the tree, such as song playlists. These structures do not lend themselves well

to a hierarchical structure, so they are kept externally in application structures, forcing users to re-create these policies by hand. In contrast, semantic structures allow applications to push these groupings into the filesystem.

Example task, Brian favorites (BF): *"Brian is taking a trip with his laptop. He doesn't want to copy all music onto his laptop as he is short on space, but he wants to have all of the songs on the playlist "Brian favorites"."*

Because the playlist does not exist in the hierarchy, the user had to add the nine files in the playlist individually, after looking up the locations using iTunes. In the semantic system, the playlist is included as a tag, allowing the user to specify the policy in a single step.

6.3 Results

All of the statistically significant comparisons are in favor of the facet interface over the alternative approaches, showing the clear advantage of semantic management for these tasks. For the single replica tasks the facet and directory interfaces perform comparably, as expected, with an average accuracy of 95% and 100% respectively, compared to an average of 15% for the volume interface. For the data organization tasks, the facet interface outperforms the directory and volume interfaces with an average accuracy of 66% compared to 14% and 6% respectively. Finally, while the accuracy of sparse tasks is not significantly different, the average time for completion for the facet interface is 73 seconds, compared to 428 seconds for the directory interface and 559 seconds for the volume interface. We discuss our statistical comparisons and the tasks in more detail in this section.

Statistical analysis: We performed a statistical analysis on our accuracy results in order to test the strength of our findings. Because our data was not well-fitted to the chi-squared test, we used a one-sided Fisher's Exact Test for accuracy and a t-test to compare times. We used Benjamini-Hochberg correction to adjust our p values to correct for our use of multiple comparisons. As is conventional in HCI studies, we used $\alpha = .05$. All the comparisons mentioned in this section were statistically significant, except where explicitly mentioned.

Single replica tasks: Figure 4 shows results from the single replica tasks. As expected, the directory and view interfaces, which both have a single replica class, perform equivalently, while the volume interface suffers heavily due to the extra complexity of two distinct replica classes. The comparisons between the single replica interfaces and the volume interface are all statistically significant. We do not show times, because they showed no appreciable differences.

Data organization tasks: Results from the three aggregation tasks (U2, RF, and TV), and the two comprehension tasks (TB and TM) are shown in Figure 5. As expected, the faceted metadata approach performs significantly better than the alternative approaches, as the filesystem structure more closely matches that of the applications. The facet interface is statistically better than both the other interfaces in the aggregation tasks, but we would need more data for statistical significance for the comprehension tasks.

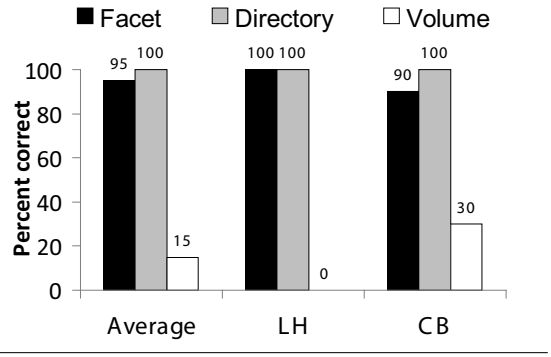

Figure 4: Single replica task results. This graph shows the results of the single replica tasks.

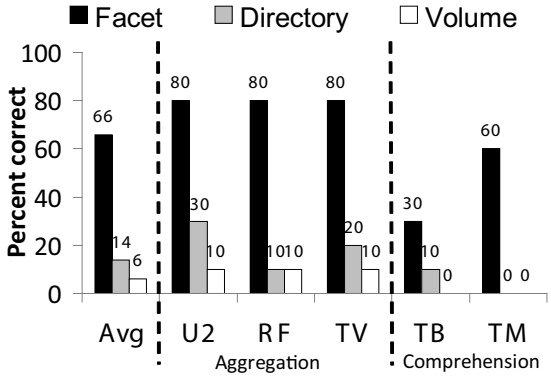

Figure 5: Data organization task results. This graph shows the results from the aggregation and comprehension tasks.

Figure 6 shows the accuracy and time metrics for the sparse tasks (BF and HV). Note that none of the accuracy comparisons are statistically significant. This is because in the sparse tasks, each file is in a unique location, making the correlation between application structure and filesystem structure clear, but very inconvenient. In contrast, for the other aggregation tasks the correlation between application and structures and the filesystem was hazy, leading to errors. However, setting the policy on each individual file was extremely time consuming, leading to a statistically significant difference in times. The one exception is the HV task, where too few volume

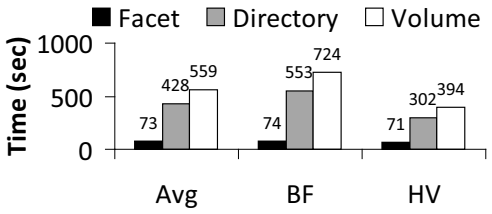

Figure 6: Sparse collection task results. This graph shows the results from all of the sparse collection tasks.

users correctly performed the task to allow comparison with the other interfaces. Indeed, the hierarchical interfaces took an order of magnitude longer than the facet interface for these tasks. Thus re-creating the groups was difficult, leading to frustration and frequent grumbling that "there must be a better way to do this."

6.4 Performance evaluation

We have found that Perspective generally incurs negligible overhead over the base filesystem, and its performance is sufficient for everyday use. Using overlap trees to reason about the location of files based on the available views is a significant improvement over simpler schemes. All our tests were run on a MacBook Pro 2.5GHz Intel Core Duo with 2GB RAM running Macintosh OS X 10.5.4.

Performance overhead: Our benchmark writes 200 4MB files, clearing the cache by writing a large amount of data elsewhere and then re-reading all 800MB. This sequential workload on small files simulates common media workloads. For these tasks, we compared Perspective to HFS+, the standard OS X filesystem. Writing the files on HFS+ and Perspective took 18.1 s and 18.6 s, respectively. Reading them took 17.0 s and 17.2 s, respectively. Perspective has less than a 3% overhead in both phases of this benchmark. In a more real-world scenario, Perspective has been used by the authors for several months as the backing store for several multi-tuner DVRs, without performance problems.

Overlap trees: Overlap trees allow us to efficiently compute how many copies of a given file set are stored in the system, despite the more flexible storage policies that views provide. It is important to make this operation efficient because, while it is only used in administration tasks, these tasks require calculation of a large number of these overlaps in real time as the user browses and manipulates data placement policies.

Figure 7 summarizes the benefits of overlap trees. We compared overlap trees to a simple method that enumerates all matching files and compares them against the views in the system. We break out the cost for tree creation and then the cost to compute an overlap. The "probe" case uses a query and view set that requires the overlap tree to probe the filesystem to compute the overlap, while the "no probe" case can be determined solely through query comparisons. Overlap trees take a task that would require seconds or minutes and turns it into a task requiring milliseconds.

Num files	Create OT	OT no probe	OT w/ probe	Simple
100	9.6ms	0.3ms	3.5ms	961ms (.9sec)
1000	29ms	0.6ms	3.8ms	12759ms (12sec)
10000	249ms	0.6ms	3.4ms	95049ms (95sec)

Figure 7: Overlap tree benchmark. This table shows the results from the overlap tree benchmark. It compares the time to create a tree and perform an overlap comparison, with or without probes, and compares to the simple enumerate approach. The results are the average of 10 runs.

7 Related work

A primary contribution of Perspective is the use of semantic queries to *manage the replication of data*. Specifically, it allows the system to provide accessibility and reliability guarantees over semantic, partially replicated data. This builds on previous semantic systems that used queries to *locate* data, and hierarchies to manage data. Our user study evaluation shows that, by supporting semantic management, Perspective can simplify important management tasks for end users.

Another contribution is a filesystem design based on insitu analysis of the home environment. This overall design could be implemented on top of a variety of underlying filesystem implementations, but we believe that a fully view-based system provides simplicity to both user and designer by keeping the primitives similar throughout the system. While no current system provides all of the features of Perspective, Perspective builds on a wealth of previous work in data placement, consistency, search and publish/subscribe event notification. In this section we discuss this related work.

Data placement: Views allow flexible data placement used to provide both reliability and mobility. Views are another step in a long progression of increasingly flexible data placement schemes.

The most basic approach to storing data in the home is to put all of the data on a single server and make all other devices in the home clients of this server. Variations of this approach centralize control, while allowing data to be cached on devices [18, 35].

To provide better reliability, AFS [34] expanded the single server model to include a tier of replicated servers,

each connected in a peer-to-peer fashion. However, clients cannot access data when they are out of contact with the servers. Coda [32] addressed this problem by allowing devices to enter a disconnected mode, in which devices use locally cached data defined by user hoarding priorities. However, hoarded replicas do not provide the reliability guarantees allowed by volumes because devices make no guarantee about what data resides on what devices, or how long they will keep the data they currently store. Views extend this notion by allowing volume-style reliability guarantees along with the flexibility of hoarding in the same abstraction.

A few filesystems suggested even more flexible methods of organizing data. BlueFS extended the hoarding primitive to allow client devices to access data hoarded on portable storage devices, in addition to the local device, but did not explore the use of this primitive for accessibility or reliability beyond that provided by Coda [21]. Footloose [24] proposed allowing individual devices to register for data types in this kind of system as an alternative to hoarding files, but did not expand it to general publish/subscribe-style queries, or explore how to use this primitive for mobility and reliability management or for distributed search.

Consistency: Perspective supports decentralized, topology-independent consistency for semantically-defined, partially replicated data, a critical feature for the home environment. While no previous system provides these properties out of the box, PRACTI [7] also provides a framework for topology-independent consistency of partially replicated data over directories, in addition to allowing a group of sophisticated consistency guarantees. PRACTI could probably be extended to use semantic groupings fairly simply, and thus provide consistency properties like Perspective. Recently, Cimbiosis [28] has also built on a view-style system of partial replication and topology independence, with a different consistency model.

Cimbiosis also presents a *sync tree* which provides a distributed algorithm to ensure connectedness, and routes updates in a more flexible manner. This sync tree could be layered on top of Perspective or PRACTIs consistency mechanisms to provide these advantages.

We chose our approach over Cimbiosis because it does not require any device to store all files, while Cimbiosis has this requirement. Many of the households in our contextual analysis did not have any such master device, leading us to believe requiring it could be a problem. Perspective also does not require small devices to track any information about the data stored on other devices, while PRACTI requires them to store imprecise summaries. However, there are advantages to each of these approaches as well. For example, PRACTI provides a more flexible consistency model than Perspective, and Cimbiosis a more compact log structure. A full comparison of the differences between these approaches, and the relative importance of these differences, is beyond the scope of this paper. We present Perspective's algorithms to show that it is possible to build a simple, efficient consistency protocol for a view-based system.

Previous peer-to-peer systems such as Bayou [40], FICUS [16] and Pangaea [30] extended synchronization and consistency algorithms to accommodate mobile devices, allowing these systems to blur or eliminate the distinction between server and client. However, none of these systems fully support topology-independent consistency with partial replication. EnsemBlue [25] takes a middle ground, providing support for groups of client devices to form device ensembles [33], which can share data separately from a server through the creation of a temporary pseudo-server, but requiring a central server for consistency and reliability.

Search: We believe that effective home data management will use search on data attributes to allow flexible access to data across heterogeneous devices. Perspective takes the naming techniques of semantic systems and applies them to the replica management tasks of mobility and reliability as well. Naturally, Perspective borrows its semantic naming structures and search techniques from a rich history of previous work. The Semantic Filesystem [12] proposed the use of attribute queries to locate data in a file system, and subsequent systems showed how these techniques could be extended to include personalization [14]. Flamenco [43] uses "faceted metadata," a scheme much like the semantic filesystem's. Many newer systems [3, 13, 19, 37] borrow from the Semantic Filesystem by adding semantic information to filesystems with traditional hierarchical naming. Microsoft's proposed WinFS filesystem also incorporated semantic naming [42].

Perspective also uses views to provide efficient distributed search, by guiding searches to appropriate devices. The most similar work is HomeViews [11], which uses a primitive similar to Perspective's views to allow users to share read-only data. HomeViews combines capabilities with persistent queries to provide an extended version of search over data, but do not use them to target replica management tasks like reliability.

Replica indices and publish/subscribe: In order to provide replica coherence and remote data access, filesystems need a replica indexing system that forwards updates to the correct file replicas and locates the replicas of a given file when it is accessed remotely. Previous systems have used volumes to index replicas [32, 34], but

did not support replica indexing in a partially replicated peer-ensemble. EnsemBlue [25] extended the volume model to support partially replicated peer-ensembles by allowing devices to store a single copy of all replica locations onto a temporarily elected *pseudo-server* device. EnsemBlue also showed how its replica indexing system could be leveraged to provide more general application-level event notification. Perspective takes an inverse approach; it uses a publish/subscribe model to implement replica indexing and, thus, application-level event notification. This matches the semantic nature of views.

This work does not propose algorithms beyond the current publish/subscribe literature [1, 4, 6, 26, 38], it applies publish/subscribe algorithms to the new area of file system replica indices. Using a publish/subscribe method for replica indexing provides advantages over a pseudo-server scheme, such as efficient ensemble creation, but also disadvantages, such as requiring view changes to move replicas. Again, a full comparison of alternative approaches is beyond the scope of the paper. We present Perspective's algorithms to show that replica indexing can be performed efficiently using views.

User studies: While we believe our contextual analysis is the first focused on home data organization and reliability, researchers have conducted a wealth of studies on technology use and management, especially in the home [2, 5, 9, 15, 17, 20, 22, 39]. We borrow our methods from these previous studies, and use them to ground our exploration and analysis.

8 Conclusion

Home users struggle with replica management tasks that are normally handled by professional administrators in other environments. Perspective provides distributed storage for the home with a new approach to data location management: the view. Views simplify replica management tasks for home storage users, allowing them to use the same attribute-based naming style for such tasks as for their regular data navigation.

Acknowlegements

We thank Rob Reeder, Jay Melican, and Jay Hasbrouck for helping with the users studies. We also thank the members and companies of the PDL Consortium (including APC, Cisco, DataDomain, EMC, Facebook, Google, HP, Hitachi, IBM, Intel, LSI, Microsoft, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0326453 and #CNS-0831407, and by the Army Research Office, under agreement number DAAD19–02–1–0389. Brandon Salmon is supported in part by an Intel Fellowship.

References

[1] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. *PODC.* (Atlanta, GA, 04–06 May. 1999), pages 53–61. ACM, 1999.

[2] R. Aipperspach, T. Rattenbury, A. Woodruff, and J. Canny. A Quantitative Method for Revealing and Comparing Places in the Home. *UBICOMP* (Orange County, CA, Sep. 2006), 2006.

[3] Beagle web page, http://beagle-project.org, 2007.

[4] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably supporting durable subscriptions in a publish/subscribe system. *DSN.* (San Francisco, CA, 22–25 Jun. 2003), pages 57–66. IEEE, 2003.

[5] A. J. Bernheim Brush and Kori M. Inkpen. Yours, Mine and Ours? Sharing and Use of Technology in Domestic Environments. *UBICOMP 07*, 2007.

[6] 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, Jul. 2000), pages 219–227, 2000.

[7] Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramana, Praveen Yalagandula, and Jiandan Zheng. PRACTI Replication. *NSDI.* (May. 2006), 2006.

[8] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. *OSDI.* (Seattle, WA, 06–08 Nov. 2006), pages 233–248. USENIX Association, 2006.

[9] David M. Frohlich, Susan Dray, and Amy Silverman. Breaking up is hard to do: family perspectives on the future of the home PC. *International Journal of Human-Computer Studies*, **54**(5):701–724, May. 2001.

[10] Filesystem in User Space. http://fuse.sourceforge.net/.

[11] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy. HomeViews: Peer-to-Peer Middleware for Personal Data Sharing Applications. *SIGMOD.*, 2007.

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

[13] Google desktop web page, http://desktop.google.com, Aug. 2007.

[14] Burra Gopal and Udi Manber. Integrating Content-based Access Mechanisms with Heirarchical File Systems. *OSDI.* (New Orleans, LA, Feb. 1999), 1999.

[15] Rebecca E Grinter, W Keith Edwards, Mark W New-

man, and Nicolas Ducheneaut. The work to make a home network work. *European Conference on Computer Supported Cooperative Work (ESCW)* (Paris, France, 18–22 Sep. 2005), 2005.

[16] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. PhD thesis, published as Ph.D. Thesis CSD-910018. University of California, Los Angeles, 1991.

[17] Thomas Karagiannis, Elias Athanasopoulos, Christos Gkantsidis, and Peter Key. *HomeMaestro: Order from Chaos in Home Networks*. MSR-TR 2008-84. Microsoft Research, May. 2008.

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

[19] Dahlia Malkhi and Doug Terry. Concise Version Vectors in WinFS. *DISC.* (Cracow, Poland, Sep. 2005), 2005.

[20] Catherine C Marshall. Rethinking Personal Digital Archiving, Part 1: Four Challenges from the Field. *DLib Magazine*, **14**(3/4), Mar. 2008.

[21] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the Blue file system. *OSDI.* (San Francisco, CA, 06–08 Dec. 2004), pages 363–378. USENIX Association, 2004.

[22] Jon O'Brien, Tom Rodden, Mark Rouncefield, and John Hughes. At home with the technology: an ethnographic study of a set-top-box trial. *CHI*, 1999.

[23] John K. Ousterhout, Andrew R. Cherenson, Fredrick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, **21**(2):23–36, Feb. 1988.

[24] 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 Oct. 2003), 2003.

[25] Daniel Peek and Jason Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. *OSDI* (Seattle, WA, 06–08 Nov. 2006), 2006.

[26] Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. *International Workshop on Distributed Event-Based Systems* (Vienna, Austria), 2002.

[27] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *United Kingdom UNIX systems User Group* (London, UK, 9–13 Jul. 1990), pages 1–9. United Kingdom UNIX systems User Group, Buntingford, Herts, 1990.

[28] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobbler, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A Platform for content-based partial replication. *NSDI.* (Boston, MA, Apr. 2009), 2009.

[29] Robert W. Reeder, Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, Kelli Bacon, Keisha How, and Heather Strong. Expandable grids for visualizing and authoring computer security policies. *CHI* (Florence, Italy, 2007), 2007.

[30] Yasushi Saito and Christos Karamanolis. *Name space consistency in the Pangaea wide-area file system*. HP Laboratories SSP Technical Report HPL–SSP–2002–12. HP Labs, Dec. 2002.

[31] Brandon Salmon, Frank Hady, and Jay Melican. *Learning to Share: A Study of Sharing Among Home Storage Devices*. Technical Report CMU-PDL-07-107. Carnegie Mellon University, Oct. 2007.

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

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

[34] Bob Sidebotham. VOLUMES – the Andrew file system data structuring primitive. *EUUGAutumn.* (Manchester, England, 22−24 Sep. 1986), pages 473–480. EUUG Secretariat, Owles Hall, Buntingford, Herts SG9 9PL, Sep. 1986.

[35] 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. *FAST.* (San Francisco, CA, 31 Mar.–02 Apr. 2004), pages 239–252. USENIX Association, 2004.

[36] Craig A. N. Soules and Gregory R. Ganger. Connections: Using Context to Enhance File Search. *SOSP.* (Brighton, United Kingdom, 23–26 Oct. 2005), pages 119–132. ACM, 2005.

[37] Spotlight web page, http://www.apple.com/macosx/-features/spotlight, Aug. 2007.

[38] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting Disconnectedness - Transparent Information Delivery for Mobile and Invisible Computing. *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2001.

[39] Alex S. Taylor, Richard Harper, Laurel Swan, Shahram Izadi, Abigail Sellen, and Mark Perry. Homes that make us smart. *Personal and Ubiquitous Computing*. Springer London, 2006.

[40] 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. *SOSP.* (Copper Mountain Resort, CO, 3–6 Dec. 1995). Published as *Operating Systems Review*, **29**(5), 1995.

[41] Mark Weiser. The computer for the 21st century. *Scientific American*, Sep. 1991.

[42] WinFS 101: Introducing the New Windows File System, March 2007. http://msdn.microsoft.com/en-us/library/-aa480687.aspx.

[43] Ping Yee, Kirsten Swearingen, Kevin Li, and Marti Hearst. Faceted Metadata for Image Search and Browsing. *CHI*, 2003.