EnsemBlue: Integrating Distributed Storage and Consumer Electronics

Daniel Peek and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

EnsemBlue is a distributed file system for personal multimedia that incorporates both general-purpose computers and consumer electronic devices (CEDs). EnsemBlue leverages the capabilities of a few general-purpose computers to make CEDs first class clients of the file system. It supports namespace diversity by translating between its distributed namespace and the local namespaces of CEDs. It supports extensibility through persistent queries, a robust event notification mechanism that leverages the underlying cache consistency protocols of the file system. Finally, it allows mobile clients to self-organize and share data through device ensembles. Our results show that these features impose little overhead, yet they enable the integration of emerging platforms such as digital cameras, MP3 players, and DVRs.

1 Introduction

Consumer electronic devices (CEDs) are increasingly important computing platforms. CEDs differ from general-purpose computers in both their degree of specialization and the narrowness of their interfaces. As predicted by Weiser [27], these computers "disappear into the background" because they present a specialized interface that is limited to the particular application for which they are designed. Nevertheless, CEDs are often formidable computing platforms that possess substantial storage, processing, and networking capabilities.

In this paper, we explore how CEDs can be integrated into a distributed file system. Our focus on storage is motivated by the difficulty of managing personal multimedia such as photos, video, and music. Current approaches to organizing data (e.g., manual synchronization) do not scale well as the number of computers and CEDs owned by a single user increases. For instance, when files are manually associated with specific devices, users must intervene to decide which items are replicated on which device. Users must also manage consistency of replicated data when files are updated. To guard against data loss, users must place copies of data in reliable, well-maintained storage locations.

Since distributed file systems have successfully automated these time-consuming and error-prone tasks in workstation environments [3, 7], we posit that they can perform a similar function for the home user. To explore our hypothesis, we have created EnsemBlue, a distributed file system that is designed to store personal multimedia. EnsemBlue, which is based on the Blue File System [14], adds several new capabilities to support consumer electronic devices:

- **persistent queries.** The heterogeneity of CEDs creates the need to customize file system behavior. For instance, many CEDs are associated with files of only one type; e.g., a digital camera with JPEGs. The network, storage, and processing resources of different CEDs vary widely since each is equipped with only the resources required to perform its particular function. Rather than treat each device equally, EnsemBlue provides persistent queries that customize its behavior for each client. A persistent query delivers event notifications to applications that specialize file system behavior. Notifications can be delivered to applications running on any EnsemBlue client; they are robust in the presence of failures and network disconnection. Persistent queries have low overhead because they reuse the existing cache consistency protocols of the file system to deliver notifications. They serve as the foundation on which to build custom automation such as type-specific caching, transcoding, and application-specific indexing.

- **namespace diversity.** Many CEDs have custom organizations for the data they store (e.g., an iPod stores music files in several different subdirectories in its local file system). The organization of data on a CED is application-specific; it is not necessarily the way the user most naturally thinks about the data. Since no single namespace can suffice for all CEDs and general-purpose computers, EnsemBlue supports namespace diversity. It creates a distributed namespace for a user's personal data that is shared among general-purpose computers — the user can organize this namespace in any fashion. It supports CEDs with

custom file organizations by automatically translating between the distributed namespace and each device-specific namespace. Changes made in the EnsemBlue namespace trigger equivalent changes in CED namespaces. Modifications made within a CED namespace are automatically propagated to the EnsemBlue namespace and shared with other clients.

- **ensemble support.** A mobile user may carry several CEDs; e.g., a cell phone, an MP3 player, and a portable DVD player. Device ensembles (sometimes called personal area networks) let multiple mobile computers and CEDs owned by the same user self-organize and share data [21]. For instance, an MP3 player might play not just music stored on its local hard drive, but also music stored on a co-located cell phone or laptop. EnsemBlue allows its mobile clients to form ensembles and directly access data from other clients. It presents a consistent view of data within each ensemble by propagating changes made on one device to replicas on other ensemble members.

Our results show that the overhead of these new capabilities is minimal. We present case studies in which we use these capabilities to integrate a digital camera and an iPod with EnsemBlue. We demonstrate that the extensibility of EnsemBlue enables a high degree of automation, including the ability to automatically organize photos and music, index text and multimedia data, and transcode content to support different media players.

2 Background

When we began this work, our ambition was to develop a distributed file system to store personal multimedia. We focused on multimedia because of the explosion of computing and consumer electronics devices that consume that type of data. Anecdotally, we found that many of our acquaintances owned several devices and that managing personal multimedia was increasingly a chore.

We adapted BlueFS [14], a distributed file system developed by our research group, to accomplish this task. Our choice of BlueFS was driven by several factors. BlueFS allows clients to operate disconnected, a critical ability for devices such as MP3 players and laptops that often operate without a network connection. BlueFS is also designed to conserve the battery lifetime of mobile clients. BlueFS has first class support for portable storage, which we felt would be useful for devices such as iPods and cameras. Finally, BlueFS lets clients dynamically add and remove storage devices — this seemed to mesh well with users who occasionally synchronize their CEDs with a general-purpose computer.

Over the past five months, our research group has used BlueFS to store personal multimedia. Our initial experience has been encouraging in several respects. We have found the common namespace of a distributed file system to be a useful way to organize data. One member of the group uses BlueFS to play music on a home PC, a laptop, a work computer, a TiVo DVR in his living room, and a D-Link wireless media player. After adding a new song to his BlueFS volume on any computer, he can play it from any of these locations. We have used support for disconnected operation to display content on long plane flights. The abilities of BlueFS to cache files on local disk and reintegrate modifications asynchronously have also proven useful. The group BlueFS file server is located at the University of Michigan, while the majority of clients are in home locations connected via broadband links. Disk caching reduces the frequency of skips when listening to music at home because it avoids a potentially unreliable broadband link. Further, when storing large video files that have been recorded at home, the ability to reintegrate modifications asynchronously has helped hide network constraints.

Unfortunately, our initial experience storing personal multimedia in BlueFS also revealed several problems. Most troubling was our inability to use BlueFS with many of our favorite CEDs such as cameras and MP3 players. Because these devices are closed platforms, they could not run the BlueFS client code. These devices required specific file organizations that did not match the way we had organized our files in the distributed namespace. The CEDs with which we had the most success were ones such as the TiVo DVR that use third-party software to interface with the local file system of a home computer. If the home computer exports the BlueFS namespace, such CEDs can indirectly read from and write to BlueFS files.

We found that the mechanisms for managing caching in BlueFS were insufficiently expressive. These mechanisms let us control caching according to the location of files within the hierarchical directory structure. However, we often wanted richer semantics. For instance, we wanted to cache all files of a certain type on particular devices (e.g., all MP3s on a laptop). Since large files were particularly time-consuming to transfer between home and work over a broadband connection, we wanted to specify policies where large files would be cached in both locations. Unfortunately, limited caching semantics constrained how we organized files. For example, to control the caching of JPEG photos, we put all files of that type in a single file system subtree.

We wished that BlueFS were more extensible. Since several of our media players supported a limited set of formats, we found it necessary to transcode files. As transcoding is CPU intensive, we used workstations for this task, which required us to log in to these machines

remotely. Instead of performing this task manually each time new media files were added to the file system, we would have preferred to extend the file system to do transcoding automatically.

Finally, we were sometimes frustrated by the need to propagate updates between clients through the file server. When both a producer and consumer of data were located at home, the broadband link connecting the home computers with the file server was a communication bottleneck. For instance, it would take many hours to propagate a video recorded on a DVR to a laptop because data had to traverse the bottleneck link twice. While we appreciated the file server as a safe repository for our data that was regularly backed up, we also wanted the ability to ship data between clients directly. We also believed that as we came to use more mobile devices, it would be useful to exchange data directly between them when they were disconnected from the server.

To address these problems, we have created a new file system, called EnsemBlue, that is based on the original BlueFS code base. EnsemBlue provides three novel capabilities, which we will describe in Sections 4–6: persistent queries to support customization of file system behavior, explicit support for closed-platform CEDs which require particular file system organizations, and ensemble support that allows multiple clients to exchange data without communicating with the file server.

3   Target environment

Individual EnsemBlue deployments are targeted at meeting the storage needs of a single user or a small group of users such as a family. A well-maintained file server might reside at home or with an ISP; its clients could include desktops, laptops, MP3 players, cell phones, and digital cameras. As many clients are mobile, EnsemBlue supports disconnected operation for isolated devices and allows collections of disconnected devices to form ensembles. Read-only sharing of content among different servers is enabled through a loosely-coupled federation mechanism.

Since EnsemBlue targets multimedia data, we expect that most files stored in the system will be large, and that reads will dominate writes. Updates, when they occur, will typically be small changes to file metadata such as song ratings and photo captions. EnsemBlue's consistency model is designed for a read-mostly workload. It uses a callback-based cache coherence strategy in which a client sets a callback with the server when it reads an object. The callback is a promise by the server to notify the client when the object is modified. Similar to Coda's weakly connected mode of operation, updates are propagated asynchronously to the file server. As in any system

that uses optimistic concurrency, conflicts can occur if two updates are made concurrently; if this occurs, EnsemBlue supports Coda-style conflict resolution.

4   Persistent queries

Persistent queries are a robust event notification mechanism that lets users customize the file system with applications that automate common tasks. With persistent queries, one can tune EnsemBlue's replication strategy to meet the needs of different CEDs. Persistent queries also let applications index or transcode files created by other computers and CEDs, even if those clients were disconnected from the network when files were added.

4.1   Design considerations

The first design issue we considered was how tightly to integrate custom functionality with EnsemBlue. We initially considered a tight integration that would allow custom code to be directly injected into the file system. However, we felt this approach would require careful sandboxing to address reliability, security, and privacy concerns. Therefore, we opted for a simpler, more loosely-coupled approach. We observed that, for local file systems, custom functionality is often implemented by standalone applications like the Glimpse indexer [13] or the lame transcoder [11]. However, existing distributed file systems do not provide a way for applications to learn about events that happen on other clients. Our approach, therefore, was to broaden the interface of EnsemBlue to better support standalone applications that extend file system behavior.

The functionality most sorely lacking was a robust event notification mechanism that supports multiple clients, some of which are mobile. Although current operating systems can notify applications about local file system changes [24], their notification mechanisms do not scale to distributed environments. For instance, a transcoder running on a laptop should be notified when JPEG files are added by other file system clients. The laptop may frequently disconnect from the network, complicating the delivery of notifications. Further, if JPEG files are added by a digital camera, the laptop and camera may rarely be connected to the network at the same time.

Potentially, we could have implemented a separate event notification framework. However, distributed file systems already provide notifications when files are modified in order to maintain cache consistency on multiple clients. Thus, *by expressing event notifications as modifications to objects within the distributed file system, we can reuse the existing cache coherency mechanism of the distributed file system to deliver those notifications.*

`pq_create`	`(IN String query, IN Long event_mask, OUT Id fileid);`	Creates a query and returns its unique identifier
`pq_delete`	`(IN Id fileid);`	Deletes the specified query
`pq_open`	`(IN Id fileid, OUT Int file_descriptor);`	Opens an existing query
`pq_close`	`(IN Int file_descriptor);`	Closes the specified query
`pq_wait`	`(IN Int file_descriptor, IN Timeval tv);`	Blocks until a record is available to read
`pq_next`	`(IN Int file_descriptor, OUT event_record);`	Returns the next record in the query log (if any)
`pq_truncate`	`(IN Int file_descriptor, IN Int record_number);`	Deletes records up to the specified record

Figure 1. Persistent query interface

A persistent query is a new type of file system object that is used to deliver event notifications. An application creates a persistent query to express the set of events that it is interested in receiving. The file server appends log records to the query when an event matching the query occurs. An application extending file system behavior reads the records from the query object, processes them, then removes them from the object. Since the persistent query is an object within the file system, the existing cache consistency mechanisms of EnsemBlue automatically propagate updates made by the server or application to the other party. EnsemBlue inherits the callback-based cache consistency of BlueFS [14], which ensures that updates made by a disconnected client are propagated to the server when the client reconnects. Similarly, invalidations queued by the server while the client was disconnected are delivered when it reconnects.

For example, an application that transcodes M4A music to the MP3 format creates a persistent query so that it is informed when new M4A files are added. It opens the query and selects on the file descriptor to block until new events arrive. The EnsemBlue client sets a callback with the file server for the query (if one does not already exist) when the query is opened. If another client adds a new M4A file, the EnsemBlue server appends an event record to the query, which causes an invalidation to be sent to the client running the transcoder. That client refetches the query and unblocks the transcoder. After reading the event record, the transcoder creates the corresponding MP3 file.

We next considered the semantics for event notification. Given our decision to implement custom functionality in standalone applications, semantics that deliver each event exactly once did not seem appropriate. Events could be lost if an application or operating system crash occurs after a notification is delivered but before the event is processed. While we could potentially perform event notification and processing as an atomic transaction, this would necessitate a much tighter coupling of the file system and applications than we want.

Instead, we observed that customizations can usually be structured as idempotent operations. For instance, an indexing application can insert a newly created file into its index only if and only if it is not already present. Therefore, EnsemBlue provides *at least once* semantics

for event notification. A customization application first receives a notification, then processes it, and finally removes the event from the query. If a crash occurs before the application processes the event, the notification is preserved since the query is a persistent object. If a crash occurs after the application processes the event but before it removes it from the query, it will reread the same notification on restart. Since its event processing is idempotent, the result is the same as if it had received only one notification.

4.2   Implementation

Figure 1 shows the interface for persistent queries. Applications running on any EnsemBlue client can create a new query by calling `pq_create` and specifying both a query string expressed over file metadata and an event mask that specifies the set of file system events on which the query string should be evaluated. Currently, the query string can be expressed over a subset of metadata fields (e.g., file name, owner, etc.). The event mask contains a bit for each modification type; e.g., it has bits for file creation and deletion.

Like directories, queries are a separate type of file system object that have a restricted application interface. A query contains both header information (the query string and event mask) and a log of events that match the query. Each record contains the event type as well as the 96 bit EnsemBlue unique identifier for the matching file.

The server keeps a list of outstanding queries. When it processes a modification, it checks all queries for the modification type to see if the state of any modified object matches any query string. If a match occurs, the server appends an event record to the query. The server guarantees that the appending of any event record is atomic with the modification by committing both updates to disk in the same transaction. Since queries are persistent, if an update is made to the file system, matching event notifications are eventually delivered.

An event mask also contains an *initial* bit that applications set to evaluate a query over the current state of the file system. If this bit is set, the server adds a record to the query for each existing file that matches the query string. If an application sets both the initial and file creation bits, the server guarantees that the application is no-

tified about all files that match the query string, including those created concurrently with the creation of the query.

Several implementation choices improve response time when evaluating persistent queries. First, queries are evaluated at the server, which typically has better computational resources than CEDs and other clients. Evaluating queries at the server also benefits from moving computation close to the data since the server stores the primary replica of every object. In contrast, evaluating queries at a client would require the client to fetch data from the server for each uncached object. Second, the server maintains an in-memory index of file metadata that it uses to answer queries over existing file system state. Use of an index means that the server does not need to scan all objects that it stores to answer each query. Finally, after the query is initially created, all further evaluation is incremental. Because the server makes each new record persistent atomically with the operation that caused the record to be written, query records are not lost due to crash or power failure. This eliminates the need to rescan the entire file system on recovery.

A drawback of making queries persistent is that queries that are no longer useful may accumulate over time. We plan to address this with a tool that periodically examines the outstanding queries and deletes ones that have not been used for a substantial period of time. Another potential drawback is that updating persistent queries creates additional serialization delays due to lock acquisition; however, since locks are held only briefly, the serialization costs in the server are usually negligible when compared with disk I/O costs. Since a query update is committed in the same disk transaction as the file operation that caused the update, the query update does not require extra disk seeks.

4.3 Examples

We have built four examples of customized functionality that use persistent queries. The first is a multimedia transcoder that converts M4A music to the MP3 format. When the transcoder first runs, it creates a query that matches existing files that have a name ending in ".m4a", as well as update and creation events for files of that name. As with many current multimedia programs, applications built on persistent queries typically infer a file's type from its name. All such applications share an implicit assumption that common naming conventions are employed while creating files.

When the transcoder is notified of a new M4A file, it invokes the Linux `faad` and `lame` tools to convert between the two formats. It stores the resulting MP3 in the same directory as the original M4A file. Since persistent queries deliver notifications to any EnsemBlue

client, we run the transcoder on a PC with ample computational resources. If an M4A file were to be added by a disconnected CED such as a cell phone, the notification reaches the transcoder once the phone reconnects to the file server. This transcoder is only 108 lines of code.

We also use persistent queries to support *type-specific affinity*. A command line tool lets users specify caching behavior for any storage device as a query string. The tool sets the initial bit in the event mask, as well as the bits for events that create new files, delete files, or modify them. When an event for an existing or newly created file is added to the query, the file is fetched and cached on the storage device. For example, using type-specific affinity, one can cache all music files on an MP3 player to allow them to be played while disconnected, or one can cache large files on a home PC to avoid communication delays associated with a broadband link.

The last two examples of persistent queries perform type-specific indexing. Applications such as iTunes, Spotlight, and Glimpse are examples of popular tools that index data stored on a local file system. However, because these tools rely on event notification mechanisms that are confined to a single computer, they do not scale well to distributed file systems. We augmented two existing tools, Glimpse and GnuPod, to use persistent queries. The Glimpse indexer creates a query that matches all events (creation, deletion, update, etc.) for files that contain textual data. The music indexer matches the same events for MP3 and other music files. The first time these tools execute, they index the files currently in a user's EnsemBlue namespace. Afterward, they incrementally update their databases when they are notified of the addition or deletion of matching files. The indexers run on powerful machines with spare cycles to reduce latency, yet their results are accessible from any client because they are stored in EnsemBlue. We used 139 lines of code to add persistent query support to Glimpse and 86 lines of code for GnuPod.

5 Integrating consumer electronic devices

5.1 Leveraging general-purpose computers

At first glance, it appears that closed-platform CEDs cannot participate in a distributed file system because they lack the extensibility to execute custom file system code. For instance, most DVRs and MP3 players require substantial hacking to modify them to run arbitrary executables. Even CEDs that are extensible at the user level may not allow kernel modifications.

EnsemBlue circumvents this problem by leveraging the capabilities of general-purpose computers. A closed-platform CED can participate in EnsemBlue by *attaching* to any general-purpose client of its file server. The

EnsemBlue daemon, Wolverine, running on the general-purpose client acts on behalf of the CED for all Ensem-Blue activities. If the user modifies data in the local CED namespace, Wolverine detects these changes and makes corresponding updates to the distributed Ensem-Blue namespace. Similarly, when data is modified in the EnsemBlue namespace, Wolverine propagates relevant modifications to the CED.

The requirements for a closed-platform CED to participate in EnsemBlue are minimal. The CED must support communication with a general-purpose computer; e.g., via a wireless interface or USB cable. The CED must provide a method to list the files it stores, as well as methods to read and update each file. Currently, Ensem-Blue supports CEDs that provide a file system interface such as FAT and cameras that support the Picture Transfer Protocol.

5.2 Making CEDs self-describing

In contrast to current models that require CEDs to synchronize with particular computers, EnsemBlue allows CEDs to attach to *any* general-purpose client, even if that client is currently disconnected from the file server. In order to provide this flexibility, EnsemBlue makes CEDs self-describing. Each CED locally stores metadata files that contain all the information needed for an EnsemBlue client to attach and interact with that CED. For each file system object on the CED that is replicated in the Ensem-Blue namespace, EnsemBlue stores a *receipt* on the CED that describes how the state of the object on the CED relates to the state of a corresponding object in the En-semBlue namespace. EnsemBlue also stores device-level metadata on the CED that uniquely identify the CED and describe its policies for propagating updates between its local namespace and the EnsemBlue namespace. Since these metadata files are small, Wolverine improves performance by reading them and caching them in memory when a CED attaches.

5.3 Supporting namespace diversity

EnsemBlue supports *namespace diversity*. It maintains a common distributed namespace that the user can organize — this namespace is exported by all general-purpose clients. On a CED that mandates a custom organization, EnsemBlue stores data in a local namespace that matches the mandated organization.

EnsemBlue views files stored on a CED as replicas of files in its distributed namespace. Each receipt maintains a one-to-one correspondence between the file in the CED namespace and the file in the distributed namespace. It stores the fully-qualified pathname of the file in the local namespace and its unique EnsemBlue identifier.

A receipt can be viewed as a type of symbolic link since it relates two logically equivalent files. We considered using symbolic links directly. However, if such links were to reside in EnsemBlue, a disconnected client would be unable to interpret the files on a CED if it did not have all relevant links cached when the CED attached. The alternative of using symbolic links in the CED's local file system is unattractive because many CED file systems do not support links. Receipts avoid both pitfalls since they are file system independent and reside on the CED.

Each receipt also contains version information. For the local namespace, the receipt stores a modification time. For the EnsemBlue namespace, the receipt stores the version vector described in Section 6. When a CED attaches to a general-purpose client, Wolverine detects modifications by comparing the versions in the receipt with the current version of the file in both namespaces. The next two subsections describe how it propagates updates between namespaces when versions differ.

5.4 Reintegrating changes from CEDs

On a general-purpose EnsemBlue client, a kernel module intercepts file modifications and redirects them to Wolverine. However, most CEDs do not allow the insertion of kernel modules, making this method infeasible. Thus, for a closed-platform CED, Wolverine uses a strategy similar to the one used by file synchronization tools such as rsync [26] and Unison [17] in which it scans the local file system of the CED to detect modifications. Wolverine scans a CED when it is first attached and subsequently at an optional device-specific interval.

When a CED attaches to a general-purpose client, Wolverine lists all files on the CED using the interface exported by that device. For instance, for CEDs that export a file system interface, Wolverine does a depth-first scan from the file system root. Usually, this scan is quick: on an iPod mini with 540 MP3s comprising 3.4 GB of storage, the scan takes less than 2 seconds. If a file on the CED has a modification time later than the time stored in its receipt, the file has been modified since the last time the CED detached from an EnsemBlue client. Wolverine copies the file from the CED to the EnsemBlue namespace and updates its receipt.

If a file or directory is found on the CED for which no receipt exists, Wolverine creates a corresponding object in the EnsemBlue namespace. It first retrieves the receipt of the object's parent directory on the CED. From the receipt, it determines the EnsemBlue directory that corresponds to the parent directory. It replicates the new object in that EnsemBlue directory.

To bootstrap this process, a user associates the CED with an EnsemBlue directory the first time the CED is attached. Wolverine replicates the local file system of the

CED as a subtree rooted at the specified directory. The user can then reorganize the data by moving files and directories from that subtree to other parts of the EnsemBlue namespace. Moving objects within EnsemBlue does not affect the organization of files on the CED.

For example, a cell phone user might download MP3s from a content provider and take pictures with a built-in camera. If the phone is associated with the directory /ensemblue/phone and the phone has separate music and photos subdirectories, EnsemBlue creates two directories /ensemblue/phone/photos and /ensemblue/phone/music that contain the new content. The user may change this behavior by moving the directories within EnsemBlue; e.g., to subtrees that store other types of music and photos. Not only will the files currently in these directories be moved to the new location, but future content created by the cell phone will be placed into the directories at their new locations.

The user can exert fine-grained control over the placement of data in EnsemBlue by relocating individual files. For instance, the user might move MP3s into a directory structure organized by artist and album. Since moving each file manually would be quite tedious, one can use persistent queries to automate this process. For instance, a music organizer could create a query to learn about new files that appear in the /ensemblue/phone/music directory. For each file, it would read the artist and album from the ID3 tag, then move the file to the appropriate directory (creating the directory if needed). In this example, the combined automation of namespace diversity and persistent queries lets the user exert substantial control over the organization of data with little effort.

If a file is deleted from a CED, Wolverine detects that a receipt exists without a corresponding local file. Depending on the policy specified for the device, Wolverine may either delete both the receipt and its corresponding file in the EnsemBlue namespace, or it may delete only the receipt. We have found the latter policy appropriate for CEDs such as DVRs and cameras on which files are often deleted due to storage constraints.

5.5 Propagating updates to CEDs

Modifications made to files in EnsemBlue are automatically propagated to corresponding files in local CED namespaces. When Wolverine creates a receipt for an object stored on a CED, it also sets a callback with the server for that file on behalf of the CED. If the file is subsequently modified by another client, the server sends an invalidation to the client to which the CED is currently attached (this client may be different from the one that set the callback). Upon receiving a callback, Wolverine fetches the new version of the file and updates the replica

and its receipt on the CED. If the CED is not attached to a client when a file is modified, the server queues the invalidation and delivers it when the CED next attaches.

EnsemBlue uses affinity to determine whether the local namespace of a CED should be updated when a new file is created. The user may specify that a subtree of the EnsemBlue namespace has affinity with a directory in the local CED namespace. At that time, Wolverine replicates the subtree in the specified directory on the CED. When setting affinity, the user may optionally specify that files created in the future in the EnsemBlue subtree should also be replicated on the CED.

CEDs support type-specific affinity. A command line tool lets the user create a persistent query as a hidden file in any directory on the CED. The EnsemBlue server initially appends event records for all existing files that match the specified query. When a file matching the query is created, the server appends an additional record. Since a callback is set on behalf of the CED for the new query, the client to which the CED is attached receives an invalidation when the server inserts new records in the query. Wolverine fetches the file referenced by each record and creates a corresponding file in the CED directory. In this manner, the CED directory is populated with all files that match the query. This use of type-specific affinity is inspired by the Semantic File System [5]. However, in contrast to SFS where directories populated by semantic queries are virtual, EnsemBlue replicates data on the CED so that the results of a query are available when the CED is disconnected.

6 Ensemble support

EnsemBlue supports ensembles, which are collections of devices that share a common view of the file system over a local network. Ensembles allow clients disconnected from the file server to share their cache contents and propagate updates to each other. For instance, a mobile user who lacks Internet access may carry a laptop, a cell phone, and an MP3 player. EnsemBlue lets these devices share a mutually consistent view of the distributed namespace. Data modifications made on one client will be seen on the others. Only clients that share a common server can form an ensemble (such devices would typically be owned by the same user or family). A client joins only one ensemble at a time.

6.1 Design considerations

In designing support for ensembles, we wished to reach a middle ground between file systems such as BlueFS [14] and Coda [10] that support disconnected operation but do not let two disconnected clients communicate, and systems such as Bayou [25] that eliminate the file server

and propagate data only through peer-to-peer exchanges. There is an important role for a central repository of data in personal file systems. A system that stores personal multimedia is entrusted with data such as family photos that have immense personal significance. Storing the primary replica of such files at the server ensures that one copy is always in a reliable location. The server is also a highly-available location from which any client may obtain a copy of the file. Yet, the peer-to-peer model is appealing in the ensemble environment. As the number of personal computing devices increases, a mobile user will often have two or more co-located devices that are disconnected from the server. Devices that cache content of interest to others should be able to share their data.

One important difference between ensembles and peer-to-peer propagation is the length of interaction. Bayou devices communicate through occasional synchronization: two clients come into contact, exchange updates, and depart. Ensembles, in contrast, allow long-lived interactions among disconnected devices. A client that joins an ensemble first reconciles the state of its cache with the view of the file system shared by the ensemble. It participates in the ensemble until it either loses contact with the other devices or reconnects with the server.

One general-purpose computer, called the *castellan*, acts as a pseudo-server for the ensemble. The castellan maintains a *replica list* that tracks the cache contents of each member. When a member suffers a cache miss, it contacts the castellan, which fetches the file from another member, if possible. Clients also propagate updates to the castellan when they modify files — the castellan in turn propagates the modifications to interested clients within the ensemble. For example, a PDA might be used to display photos taken with a cell phone. The cell phone updates a shared directory when it takes a new photo, causing the castellan to invalidate the replica of the directory cached on the PDA. Subsequently, the PDA would contact the castellan to fetch the modified directory and new photo from the phone.

Ensembles are designed to support specialized CEDs that cache only a small portion of the objects in the file system. Devices such as MP3 players and cell phones are incapable of storing a complete copy of all objects in a data volume, as is done in systems such as Bayou, or keeping a log of updates to all objects, as is done in systems such as Footloose [15] and Segank [22]. For instance, a single video exceeds the storage capacity of a typical cell phone. EnsemBlue lets a client cache only the objects with which it has affinity. Thus, a CED need not waste storage, CPU cycles, or battery energy processing updates for files it will never access.

We struggled to balance the concerns of consistency and availability. When a client is disconnected from the server, its cached version of a file may be stale since it cannot receive invalidations. Other clients within the ensemble would see the stale version if they read the file, assuming that no other member cached a more recent version. In the worst case, a client may have previously seen an up-to-date version of the file, evicted that version from its cache, then joined the ensemble while disconnected. The client would see an older version of the file than it previously viewed under this scenario.

We initially devised many solutions to improve consistency in ensembles. For example, we considered having each client remember the versions of all objects that it ever read. We also considered having each client store complete version information for all objects in the file system. However, we felt that these solutions did not fit well with a storage system that focuses on personal multimedia. The types of file updates that we envision a user making while disconnected are often trivial; e.g., changing the rating on a song, or adding a tag to a photo. We therefore asked ourselves: is it better to present data that might possibly be stale or to present no data at all? In our target environment, we believe the former answer is correct (although in a workstation environment, we would give a different answer).

6.2 Implementation

An ensemble is formed when two or more disconnected clients share a local area network. For wireless devices, we leverage PAN-on-Demand [1], which lets devices owned by the same user discover each other and form a personal area network. Since PAN-on-Demand targets devices carried by a single user, it assumes that its members can communicate via a single radio hop. To form an ensemble, at least one client must be a general-purpose computer. This device serves as the castellan; the other devices are its clients. CEDs can join an ensemble by attaching to a general-purpose ensemble member.

6.2.1 Tracking modifications

EnsemBlue tracks modifications for each object using a version vector [16] that contains a `<client, version>` tuple for each client that modified the object. A modifying client increments the version in its tuple. If it has not yet modified the object, it appends a tuple with its unique EnsemBlue client identifier and a version of one.

Version vectors are used to compare the recency of replicas. If two version vectors are the same, the replicas are equivalent. For non-equivalent replicas, we say that a replica is more recent than another if for every client in the version vector of the second replica, the client exists in the version vector of the first replica with an equal or greater version number. If two replicas are not equivalent and neither is more recent than the other, concurrent

updates have been made by different clients. In this case, EnsemBlue asks the user to manually resolve the conflict. However, if concurrent updates have been made to a directory and EnsemBlue can automatically merge the updates, it will do so without user involvement; e.g., it will merge updates that create two different files in the same directory. This strategy is the same as Coda's strategy for reintegrating changes from disconnected clients.

While a client is connected to the server, its locally cached replicas are the same as the primary replicas stored on the server. Once a client disconnects, it appends operations that modify its cached objects to a *disconnection log*. When the client reconnects with the server, it replays the disconnection log to reintegrate changes. Each disconnection log entry contains a modification (write, file creation, rmdir, etc.), the version vectors of all objects modified, and the unique identifier of the client that made the modification.

Each client maintains the invariant that *for each cached object, the disconnection log contains all operations needed to recreate the cached version of the object starting with a version already seen by the server.* A client that never joins an ensemble maintains this invariant trivially since it appends an entry to the disconnection log every time it modifies an object. When it disconnected, all cached objects were versions seen by the server. By replaying the log, the server can reconcile each primary replica with the modified version on the client.

6.2.2 Joining an ensemble

A disconnected client joins an ensemble when it discovers another disconnected client or an existing ensemble on its local network. The joining computer becomes a client of the castellan of the existing ensemble. If the discovered device is an isolated disconnected client, then the discovered device becomes the castellan and the discoverer becomes its client. While the current method of choosing the castellan is arbitrary, selecting a less-capable device as the castellan, e.g., a PDA instead of a laptop, impacts only performance, not the correctness of the system. In the future, we plan to add heuristics that select the most capable client to be the castellan.

Before a disconnected client joins an ensemble, it must reconcile its view of the EnsemBlue namespace with that of the ensemble. After reconciliation, if an object is replicated on more than one ensemble client, all replicas are the same, most up-to-date version.

The joining client sends the castellan the version vectors of its cached objects. The castellan stores the version vectors of all objects cached on any ensemble member in the replica list. By comparing the two sets of version vectors, it first determines the set of objects that are replicated on both the ensemble and the joining client. It then determines the subset of these objects for which the ensemble version and the version on the joining client differ — this is the set of objects to be reconciled.

If an object being reconciled is in conflict due to concurrent updates on disconnected clients, the user must resolve the conflict before the new device joins the ensemble. Otherwise, EnsemBlue brings the less recent replica up-to-date. If the disconnection log on the client with the more recent replica contains sufficient records to update the less recent replica, the log records are transmitted to the out-of-date client. The client applies the logged operations and adds the records to its disconnection log. If the ensemble version is out-of-date, the castellan applies the log records to its local cache if necessary, then forwards the records to other members that cache the object. We anticipate that transmitting and applying log records will usually be more efficient than transmitting the entire object. Most multimedia files are large, yet updates are often trivial; e.g., setting the fields in an ID3 tag.

Sometimes, log records alone are insufficient to bring the less recent replica up-to-date. This occurs when the less recent replica is older than the version associated with any record in the disconnection log of the client with the most recent replica. In this case, EnsemBlue simply invalidates the stale replica. Any client that has affinity to the invalidated object fetches the most recent version and its associated log records after the ensemble forms.

Reconciliation ends when all out-of-date replicas on the joining client and the existing ensemble clients have been updated or invalidated. The castellan updates the replica list to include objects cached by the joining client.

6.2.3 Ensemble operation

After joining an ensemble, a client can fetch data from other ensemble members via the castellan. EnsemBlue inherits the dynamic cache hierarchy of BlueFS [14]. It fetches data from the location predicted to have the best performance and use the least energy. A client that decides to fetch data from the ensemble sends an RPC to the castellan. The castellan examines the replica list to determine which client, if any, caches the data. It either services the request itself or forwards it to another client. If the data is not cached within the ensemble, the castellan returns an error code.

If the requesting client does not have an up-to-date version of the object, the responding client includes all records in its disconnection log that pertain to the object — this maintains the invariant described in Section 6.2.1. The requesting client appends the records to its disconnection log and caches the object in its local storage. The castellan updates its replica list to note that the object is now cached by the requesting client.

Any ensemble client that modifies an object sends an RPC to the castellan. The castellan forwards the modification to all ensemble members that cache the object. These clients update their cached objects and append a record to their disconnection logs. Thus, most clients are only informed of updates that are relevant to them. The castellan does not send clients updates and requests for objects they do not cache. For instance, an MP3 player that caches only music files will not be informed about updates to photos or asked to service requests for e-mail.

6.2.4 Leaving an ensemble

When the castellan loses contact with a client, the client is considered to have left the ensemble. The castellan removes the objects cached by that client from the replica list. If the castellan departs, the ensemble is disbanded. The remaining clients form a new ensemble if they remain in communication.

A client that leaves an ensemble operates disconnected until it joins another ensemble or reconnects with the server. Its disconnection log may now contain updates made by other clients. Upon reconnection, the client sends its log to the server. Since the first entry in the log for any object modifies a version previously seen by the server, the server can use the log to update the primary replicas. However, since multiple clients can reintegrate the same log record, the server must not apply the same record twice. It uses the version vectors in the log records to eliminate duplicates. If the version that it caches is more recent than the log record, it ignores the modification. Otherwise, it applies the modification to the primary replicas. If a primary replica is more recent than the replica on the reconnecting client, the server sends the client an invalidation.

This design lets mobile clients reconcile changes on behalf of other clients. Thus, a client can remain up-to-date even though it never reconnects with the server. For example, a car stereo could retrieve MP3s from a disconnected client such as a laptop that is transported in the car. The stereo could also reintegrate changes to play lists and song ratings back to the server via the laptop.

7 Evaluation

Our evaluation answers the following questions:

- What is the overhead of forming an ensemble?
- What is the overhead of persistent queries?
- How effectively can CEDs such as cameras and MP3 players be integrated with EnsemBlue?

7.1 Ensemble formation

We measured the time two disconnected clients take to form an ensemble. During this experiment, an IBM X40
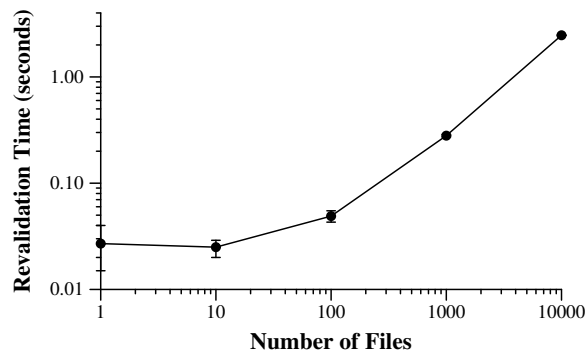
This figure shows how the time to form an ensemble varies with the number of files stored on each device. The graph is log-log. Each result is the mean of 7 trials — the error bars are 90% confidence intervals.

Figure 2. Ensemble formation time

laptop with a 1.2 GHz Pentium M processor and 768 MB of RAM becomes the castellan, and an IBM T20 laptop with a 700 MHz Pentium 3 processor and 128 MB of RAM becomes its client. The computers communicate via an 802.11b wireless ad-hoc connection.

Figure 2 shows how the time to form the ensemble changes as we vary the number of files cached on both clients. This data is displayed using a log-log graph due to the disparity in reconciliation time. At the beginning of each experiment, both laptops cache the same version of each file. Since only metadata is exchanged in this experiment, file size is unimportant and all files are zero length. In the absence of out-of-date replicas, the time to form an ensemble is quite small. Beyond an approximately 20 ms constant performance cost for the initial message exchange, formation time is roughly proportional to the number of cached items. Even with 10,000 files cached on both machines, the ensemble forms in less than three seconds.

We next measured the effect of reconciling out-of-date objects on ensemble formation time. At the beginning of this experiment, the X40 client contains 18 MP3 files that total 115 MB in size, as well as 40 photos comprising a total of 110 MB of data. The T20 contains only the 18 MP3 files. It does not have affinity for the photos and does not cache them.

We consider the four scenarios in Figure 3. In the first scenario, none of the files are modified. As predicted by the previous experiment, the ensemble is formed in a fraction of a second. In the second scenario, the X40 modifies the ID3 tag of all MP3s prior to joining the ensemble. The ensemble is formed in slightly less than a second. The additional delay reflects the time for the X40 to transmit its disconnection log records that correspond to the ID3 tag modifications.

Scenario	Formation time (seconds)
None	0.34 (0.32–0.36)
ID3 Tags	0.95 (0.91–0.96)
Music	226 (219–233)
All	227 (221–234)

This figure shows how the number of updates reconciled affects ensemble formation time. In the first row, no files are updated. In the second row, ID3 tags on 18 MP3s are updated. In the third row, the 18 MP3s are re-encoded, and in the last row, the 18 MP3s and 40 photos are completely modified. Each result is the mean of 5 trials — minimum and maximum values are in parentheses.

Figure 3. Time to reconcile updates in an ensemble

In the third scenario, the X40 overwrites the contents of all MP3s before joining the ensemble — this corresponds to a user re-encoding the MP3s from a CD. It takes 227 seconds to form the ensemble since each large file on the T20 is completely overwritten. For comparison, the time to manually copy the files is 209 seconds. Thus, the overhead due to EnsemBlue is only 8%. The difference between the second and third scenarios illustrates the benefit of shipping log records rather than entire objects during reconciliation. When modifications are a small portion of the total size of each file, EnsemBlue realizes a substantial performance improvement over a manual copy of the files.

In the final scenario, the X40 overwrites both the MP3 files and the photos in its cache. However, the time to form the ensemble is virtually identical to the third scenario. Since the T20 does not cache photos, it does not have to be informed about the additional updates; the X40 ships only log records that pertain to the MP3 files. From these results, we conclude that EnsemBlue can achieve substantial performance benefit by limiting reconciliation to the set of objects cached on both clients. In contrast, a file system that transfers all updates would nearly double the reconciliation time.

7.2 Persistent query overhead

We next measured the overhead of persistent queries, which is exhibited in two ways. First, evaluating a large number of queries might slow the server as it processes modifications. Second, the evaluation of individual queries might exhibit a high latency that would preclude them from being used by interactive applications.

For these experiments, the EnsemBlue server was a Dell Precision 370 desktop with a 3 GHz Pentium 4 processor and 2 GB of RAM. The client was the X40 laptop from the previous experiments. The two computers are connected via 100 Mb/s Ethernet.

To evaluate the first source of overhead, we ran an I/O intensive benchmark in which we untar the Apache 2.0.48
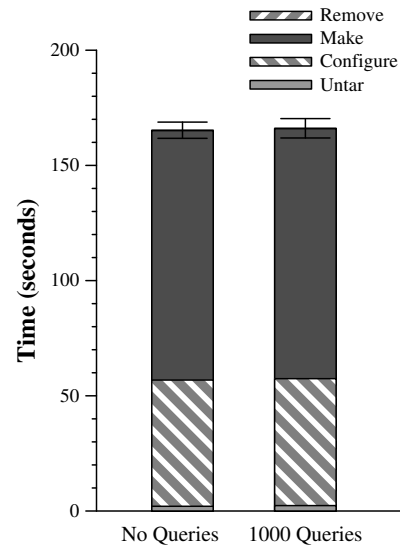
This figure compares the time to run the Apache build benchmark with no queries outstanding and 1,000 queries outstanding. Each result is the mean of 5 trials — the error bars are 90% confidence intervals.

Figure 4. Overhead of persistent queries

source tree into EnsemBlue, run `configure` in an object directory within EnsemBlue, run `make` in the object directory, and finally remove all files. Figure 4 compares the time to perform the benchmark when the server is evaluating 1000 outstanding queries with the time to perform the benchmark when no queries are outstanding. The results are identical within experimental error, indicating that persistent query evaluation is not a substantial source of overhead.

To evaluate the second source of overhead, we measured the time to create a query while varying the number of records returned. Before running each experiment, we populated EnsemBlue with the data from the personal volume of a user of the prototype described in Section 2. This data set contains 5,276 files and is over 7 GB in size. We created persistent queries with the initial bit set in the event mask and with the query string matching the various file types shown in Figure 5.

We measured the time for an application to create each query and read all matching records. The results show a fixed cost of approximately 126 ms. While the latency increases with the number of matching records, the incremental cost is small. If all 5,276 files match the query string, the experiment takes 62 ms longer.

From these results, we conclude that persistent queries impose minimal overhead during both creation and evaluation. We are encouraged that these results indicate that persistent queries are cheap enough to be employed by a wide variety of applications.

File type	Matches	Creation time (seconds)
None	0	0.126 (0.125–0.126)
TiVo	2	0.126 (0.125–0.126)
text	45	0.128 (0.126–0.133)
jpeg	132	0.127 (0.126–0.128)
postscript	712	0.135 (0.116–0.146)
MP3	1729	0.150 (0.147–0.160)
All	5276	0.188 (0.161–0.198)

This figure shows the time to create a persistent query matching varied numbers of files. Each result is the mean of 8 trials — the values in parentheses are the minimum and maximum trials.

Figure 5. Time to create a persistent query

7.3 Case study: Integrating a digital camera

In the next two sections, we present case studies in which we examine how well CEDs can be integrated with EnsemBlue. In the first, we take pictures using a Canon PowerShot S40 digital camera. The camera produces JPEG photos that it stores in a FAT file system. We registered the camera with EnsemBlue by specifying a root directory to which photos should be imported. The camera groups the photos it takes into subdirectories that contain 100 photos each. The default behavior of EnsemBlue is to recreate the camera directory structure within the root directory specified during registration. This is not the most user-friendly of organizations.

We first decided to organize our photos by date. The camera stores metadata in each JPEG that specifies when it took the photo. We created a photo organizer application that creates a persistent query to be notified when a new photo is added to EnsemBlue. The organizer reads the JPEG metadata and moves the file to a subdirectory specific to that date, creating the directory if necessary. After moving each JPEG file, the organizer removes the notification from the query.

We next enhanced our organizer to arrange photos by appointment. The organizer takes as a parameter the location of an ical `.calendar` file. When a new JPEG is added, it searches the file for any appointment entered for the time the photo was taken. If it finds such an appointment, it moves the photo to a subdirectory for that appointment within the directory for the date when the photo was taken (again, creating the subdirectory as needed). The photo organizer required only 123 lines of code, indicating that such applications can be created with minimal effort by CED manufacturers, third-party software developers, and technically-savvy users.

We measured the time to import photos from our camera into EnsemBlue using the same experimental setup as in the previous section. The camera, containing 201 new photos approximately 256 MB in size, is attached to the X40 laptop client. EnsemBlue takes approximately 188 seconds to import the photos. In contrast, copying all the files manually takes 174 seconds. The approximately 7% overhead imposed by EnsemBlue seems a reasonable price to pay for automatic replication and organization of the imported photos, especially when one considers the time required to manually organize 201 images.

7.4 Case study: Integrating an MP3 player

Our second case study integrates an iPod mini MP3 player and a D-Link media player with EnsemBlue. The iPod is a mobile device that stores and plays music files of several different formats. It presents two challenges for integration with a distributed file system. First, music files are stored in specific subdirectories of its local file system — the music files should be spread between these subdirectories to improve lookup latency. Second, the iPod uses a custom database to store information about the music files in its local storage. This database must be updated when files are added.

We address the first challenge with type-specific affinity. To place files in specific subdirectories on the iPod, we create a query for each directory that matches music files stored within EnsemBlue. To divide files between subdirectories, we take the simple approach of partitioning the namespace by filename. For example, the query for the first subdirectory matches on music files that begin with 'a'. When a new music file beginning with 'a' is added to EnsemBlue, the server inserts a record in the query for that directory. When the iPod next attaches to a client, that client fetches the query, reads the record, and replicates the file on the iPod within that subdirectory.

We address the second challenge by creating a standalone application to update the iPod database. This application creates a query that matches all music files. When a file is added, the application updates the iPod database within the EnsemBlue namespace using Gnu-Pod. The database on the iPod's local storage has affinity to this file. Thus, when the iPod attaches to a client, that client receives a callback on behalf of the iPod for the database file. It fetches the database and replicates it on the iPod. This application required only 86 lines of code.

We also added support for a D-Link media player that can only play music files encoded in the MP3 format. Since many of our music files are encoded in the M4A format, we wrote a transcoder, described in Section 4.3, to convert files so that they could be played on the media player. The D-Link media player can read files using a client program that exports the file system of a general-purpose computer. Thus, we simply run that program on an EnsemBlue client; the media player can play music in EnsemBlue without further customization. One of our

group members uses an identical strategy to play music files stored in EnsemBlue on a TiVo DVR.

8 Related work

To the best of our knowledge, EnsemBlue is the first distributed file system to provide explicit support for consumer electronics devices. Its novel contributions include persistent queries, which leverage the underlying cache consistency mechanisms of the file system to deliver application-specific event notifications, and receipts, which allow namespace diversity.

Many operating systems provide application notifications about changes to local file systems; Linux's inotify [12], the Windows Change Journal [4], and Apple's Spotlight [24] are three examples. Watchdogs [2], proposed by Bershad and Pinkerton, combine notification with customization by allowing user-level applications to safely overload file system operations for particular files and directories. Unlike persistent queries, these mechanisms are limited to a single computer and do not scale to the distributed environment targeted by EnsemBlue. Our approach of evaluating persistent queries on the server realizes the same benefit of pushing evaluation to the data that has been previously shown in projects such as Active Disks [19] and Diamond [8]. Salmon et al. [20] have recently proposed *views* that allow pervasive devices to publish interest in particular sets of objects with peer devices. Views are similar to persistent queries in that they allow clients to specify the objects in which they are interested as a semantic query. Since views target a serverless system, each view must be propagated to all peers.

Type-specific affinity is similar to the virtual directories presented by the Semantic File System [5]. However, the directory contents produced by type-specific affinity are persistent, meaning that they can be accessed by a mobile computer or CED that is disconnected from the file server. Persistent queries should not be confused with applications such as Glimpse [13] and Connections [23] that index file system data. Rather, persistent queries are a tool that such applications can use; they notify indexing applications about changes to file system state.

EnsemBlue's strategy of scanning the local file system of closed-platform CEDs is similar to the way that file synchronizers operate [17, 26]. The main difference between the two strategies lies in EnsemBlue's use of receipts. Receipts are a more general mapping between namespaces that allow files to be moved within the distributed namespace, yet retain the same location in the local device namespace. As shown in Section 7.3, receipts can be combined with persistent queries to automate the remapping of individual files between namespaces.

EnsemBlue's model of supporting isolated disconnected clients owes much to Coda [10]. From Coda, EnsemBlue inherits the use of a disconnection log to store updates, as well as its conflict resolution strategy. However, EnsemBlue differs from prior distributed file systems in its explicit support for ensembles of disconnected clients. While others have identified ensembles as an emerging paradigm for personal mobile computing [21], EnsemBlue is the first server-based distributed file system to explicitly support this model of computing.

Many previous storage systems have eschewed a central server in favor of propagating data through peer-to-peer exchanges. Bayou [25] replicates data collections in their entirety. A Bayou client can read and write any accessible replica. Replicas are reconciled by applying per-write conflict resolution based on client-provided merge procedures. EnsemBlue differs from Bayou in that it allows its clients to cache only a portion of a data volume. Further, ensembles remain consistent after formation, whereas Bayou replicas can diverge after each reconciliation.

Footloose [15] and EnsemBlue both present a consistent view of objects on devices located close to the user. Like Bayou, Footloose allows clients to exchange data through propagation of update records. *Wishes* in Footloose provide an analogous event notification mechanism to persistent queries. However, unlike persistent queries, wishes do not guarantee at-least-once delivery. Footloose requires that update records be preserved until every interested client is known to have received the record; this could be a problem for CEDs with limited storage. In contrast, EnsemBlue clients can discard update records once they have been reconciled with its server. Footloose targets CEDs as clients, but requires that any client be sufficiently open to run their Java code base. Footloose does not export a file system interface, and thus cannot be used with legacy applications.

Other systems that support peer-to-peer update propagation are Segank [22], in which a MOAD carried by a user at all times ensures consistency among computers that share a namespace, Ficus [6], Files Every Where [18], and OmniStore [9].

9 Conclusion

Consumer electronics devices are increasingly important computing platforms. Yet, it remains challenging to integrate them into existing distributed systems. CED architectures are typically closed, admitting only a narrow interface for interaction with the outside world. Their capabilities are non-uniform since available resources are chosen to support a particular application. This heterogeneity implies that a distributed system that supports

CEDs should be flexible. It must customize its interactions with each device according to the interface presented. If a CED lacks the necessary resources to participate in a distributed protocol, the protocol should allow for other, more general-purpose participants to supply needed resources on its behalf.

EnsemBlue shows the benefit of flexibility. By supporting namespace diversity, device ensembles, and persistent queries, EnsemBlue is highly extensible. As the case studies in this paper demonstrate, extensibility is crucial to making devices such as MP3 players, cameras, and media players full-fledged participants in EnsemBlue. Based on these results, we are hopeful that similar principles can be applied to other distributed systems to allow them to support CEDs.

Acknowledgments

References

[1] ANAND, M., AND FLINN, J. PAN-on-Demand: Building self-organizing WPANs for better power management. Tech. Rep. CSE-TR-524-06, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 2006.

[2] BERSHAD, B. B., AND PINKERTON, C. B. Watchdogs - extending the unix file system. *Computer Systems 1*, 2 (Spring 1988).

[3] CALLAGHAN, B., PAWLOWSKI, B., AND STAUBACH, P. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, IETF, June 1995.

[4] COOPERSTEIN, J., AND RICHTER, J. Keeping an eye on your NTFS drives: the Windows 2000 Change Journal explained. *Microsft Systems Journal* (September 1999).

[5] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 16–25.

[6] GUY, R. G., HEIDEMANN, J. S., MAK, W., PAGE, T. W., POPEK, G. J., AND ROTHMEIER, D. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Conference* (June 1990), pp. 63–71.

[7] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[8] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the USENIX FAST '04 Conference on File and Storage Technologies* (March 2004).

[9] KARYPIDIS, A., AND LALIS, S. Omnistore: A system for ubiquitous personal storage management. In *Proceedings of the 4th IEEE International Conference on Pervasive Computing And Communications* (Pisa, Italy, March 2006), pp. 136–147.

[10] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[11] http://lame.sourceforge.net/.

[12] LOVE, R. Kernel korner: Intro to inotify. *Linux Journal 2005*, 139 (2005), 8.

[13] MANBER, U., AND WU, S. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the 1994 Winter USENIX Conference* (San Francisco, CA, January 1994), pp. 23–32.

[14] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[15] PALUSKA, J. M., SAFF, D., YEH, T., AND CHEN, K. Footloose: A case for physical eventual consistency and selective conflict resolution. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, Oct. 2003).

[16] PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistencies in distributed systems. *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983), 240–247.

[17] PIERCE, B. C., AND VOUILLON, J. What's in Unison? A formal specification and reference implementation of a file synchronizer. Tech. Rep. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[18] PREGUICA, N., BAQUERO, C., MARTINS, J. L., SHAPIRO, M., ALMEIDA, P. S., DOMINGOS, H., FONTE, V., AND DUARTE, S. Few: File management for portable devices. In *Proceedings of the International Workshop on Software Support for Portable Storage* (San Francisco, CA, March 2005), pp. 29–35.

[19] RIEDEL, E., FALOUTSOS, C., GIBSON, G. A., AND NAGLE, D. Active disks for large-scale data processing. *IEEE Computer* (June 2001), 68–74.

[20] SALMON, B., SCHLOSSER, S. W., AND GANGER, G. R. Towards efficient semantic object storage for the home. Tech. Rep. CMU-PDL-06-103, Carnegie Mellon University, May 2006.

[21] SCHILIT, B. N., AND SENGUPTA, U. Device ensembles. *Computer 37*, 12 (December 2004), 56–64.

[22] SOBTI, S., GARG, N., ZHENG, F., LAI, J., SHAO, Y., ZHANG, C., ZISKIND, E., KRISHNAMURTHY, A., AND WANG, R. Y. Segank: A distributed mobile storage system. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies* (San Francisco, CA, March/April 2004).

[23] SOULES, C. A. N., AND GANGER, G. R. Connections: using context to enhance file search. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 119–132.

[24] Spotlight overview. Tech. Rep. 2006-04-04, Apple Corp., Cupertino, CA, 2006.

[25] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, 1995), pp. 172–182.

[26] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Tech. Rep. TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.

[27] WEISER, M. The computer for the 21st century. *ACM SIGMOBILE Mobile Computing and Communications Review 3*, 3 (July 1999), 3–11.