Consumer Distributed File Systems

by

Daniel N. Peek

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:
Associate Professor Jason N. Flinn, Chair
Professor Farnam Jahanian
Associate Professor Brian D. Noble
Assistant Professor Mark W. Newman

For family and friends.

ACKNOWLEDGEMENTS

This work is the result of the efforts of a group of people. The foremost of these is my advisor, Professor Jason Flinn. I have had the privilege of being Jason's fourth graduate student and the last who started at the University of Michigan when he did. I have been the beneficiary of his enormous patience, experience, insights, and indulgence. I couldn't have asked for a better mentor.

I would like to thank my committee members: Professors Brian Noble, Farnam Jahanian, and Mark Newman. Their advice and insight improved this work.

I am also grateful to the members of my research group: Manish Anand, Ed Nightingale, Ya-Yunn Su, Kaushik Veeraraghavan, Mona Attariyan, Brett Higgins, Puspesh Kumar, and Michelle Noronha. They always provided advice, support, problems, and ideas. I would also like to thank my officemates and housemates: Benjamin Wester, Arnab Nandi, David Watson, Evan Cooke, Jon Oberheide, Sushant Sinha, Kaushtubh Nyalkalkar, Amy Kao, Darshan Karwat, and Ali Besharatian, who made sure there was always something interesting going on.

Thanks also to my internship mentor and collaborators: Doug Terry, Tom Rodeheffer, Ted Wobber, Rama Ramasubramanian, and Meg Walraed-Sullivan.

I would like to acknowledge the dozens of program committee members and anonymous reviewers who have endured my papers over the years. Their efforts make the software systems community possible.

Finally, I would like to thank my family. They have been very supportive of this academic pursuit over the years it took to complete.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

Consumers have long stored their data on local file systems on personal computers. However, these file systems are no longer sufficient to manage consumer data due to the emergence of consumer electronics (CE) devices such as cell phones and digital cameras and the rapid increase of multimedia data.

CE devices are a problem for local file systems because they store user data outside the purview of the local file system on heterogeneous devices. As a result, intervention is required to synchronize the state of the CE devices and a local file system. For instance, pictures can be downloaded from a digital camera or iPod contents can be synchronized by using iTunes. However, these management tasks burden the user. They require the user to remember which data needs to be transferred and to learn to operate new application interfaces.

Our insight into simplifying this situation is that although each instance of the stored data is physically distinct, conceptually, all instances of a stored data item are a single logical entity. Distributed file systems embody a similar realization for general-purpose computers, presenting an interface in which all computers can access all data as if they were all connected to a single large shared disk and maintaining this illusion by automatically propagating data. This eases data management issues by allowing users to operate on a single conceptual data item instead of the dispersed physical instances of that data. By creating a distributed file system capable of including both

general-purpose computers and CE devices as clients, we can eliminate the chore of manual synchronization.

We have also experienced an immense expansion of multimedia data [2, 16]. Studies [61] have shown that higher level management techniques are needed to help manage the burgeoning data. An example of this new management is the ability to list files based not on their position within a hierarchy of file system directories, but based on other properties of files such as the quality of a JPEG image file. These properties, called metadata, are typically encoded into file data in a manner governed by their file type. This arrangement is opaque to current file systems, preventing them from accessing and this metadata and consequently, from providing type-aware functionality such as metadata indexing, automatic conflict resolution, and generating document previews.

Our response is to build a new mechanism that allows the file system to be extended with code that provides type-aware features. We have also developed a technique that can provide many type-aware functions and does not require the creation of code for each file type.

These two problems, the lack of CE device support and type-awareness, are an obstacle to consumer storage use. This work demonstrates techniques to address these difficulties. Thus, our thesis is that the inclusion of CE devices and type-awareness enable distributed storage systems to better manage personal multimedia data.

1.1 Principles

In our pursuit of a consumer distributed file system, we will be guided by three principles: type-awareness, the reuse of existing work, and maximizing data availability.

Type-awareness is the application of information about file types to activities in the storage system. Knowledge of file types lets the file system support features such as type-specific conflict resolution, the ability to automatically resolve concurrent modifications to orthogonal fields of a file that would otherwise have to be handled as

an actual conflict [35]. Type-awareness also yields benefits for user-level applications. Providing an understanding of type-specific metadata permits applications to select files on the basis of metadata contained by files instead of file names.

Reuse of existing work will also be critical to our efforts. There are far too many file types and CE devices for us to understand them individually. To address this multitude of entities, we must take advantage of existing tools that operate on them and the interfaces they present, even if they were not originally intended for our purposes or reuse of any sort. Fortunately, external forces often encourage similar entities to implement common interfaces. For example, most applications with a GUI interface expose an accessibility API designed for use by tools to assist the blind because the default GUI widgets (buttons, scroll bars, etc.) in modern popular windowing systems implement these interfaces by default. Because this single interface is used across many applications, it is simpler to gather information from many applications, giving us access to much more functionality than we could produce ourselves.

Finally, maximizing data availability is a common goal of many distributed file systems. Data is considered available if it can be delivered to a requesting application. Failure to provide such data is harmful because it may prevent the user from effectively using the computing system. It also dispels the illusion of a single large shared disk that distributed file systems provide. With the addition of CE devices, simple data transmission is sometimes insufficient to deliver content. Many CE devices only operate on a few types of data. Files may have to be transcoded into a suitable format before they can be accessed on these devices.

1.2 Distributed file systems for consumers

Distributed file systems are a natural starting point for this work. As mentioned before, they already treat distributed instances of a file as a single conceptual file. Distributed file systems such as CIFS [39] (also called Windows file sharing), Apple Filing Protocol, NFS [50], and AFS [29] are already widely used by consumers. Academic projects in this area include Coda [34], Segank [67], and BlueFS [52].

Consumer distributed file systems are distributed file systems that aim to serve the needs of nonprofessional single users or small groups of users such as a family. Such a distributed file system should be able to store hundreds of thousands of files, operate with a few clients performing actions concurrently, have a strategy for dealing with concurrent updates (either by resolving conflicts, or preventing them from occurring), and support disconnected operation to improve data availability. We propose that consumer distributed file systems should also include support for CE devices and type-awareness to adapt to the popularity of CE devices and the profusion of multimedia data.

1.3 Adding type-awareness

In our pursuit of type-aware functionality in the file system, we first implemented persistent queries, a file system feature that allows programs to receive notifications when files matching a specified criteria are changed anywhere in the distributed file system. This simplifies the development of services that take type-specific actions such as transcoding files.

We then sought to build a new mechanism into the file system to allow the insertion of type-specific behavior. However, integrating this new functionality into a distributed file system must be done with care so that the file system remains reliable despite the incorporation of large amounts of code. We devised a binary instrumentation-based mechanism called sprockets that allows type-aware functionality to run in the distributed file system address space while protecting the distributed file system from bugs in the loaded code.

Once we had a mechanism for incorporating type-aware behavior, we were ready to create type-aware functionality for the file system to use. Existing systems rely on the creation of pieces of custom code that parse each file type. Unfortunately, this approach requires substantial developer effort for each file type to be supported. As a result, such a system is expensive to construct or unable to parse many of the vast array of file types. Our TrapperKeeper project takes a new approach to

type-awareness by eliminating custom code and instead reusing the functionality of existing applications that can parse these file types.

1.4 Connecting CE devices

CE devices are combined hardware and software products sold to consumers and designed to perform a specific set of functions. This is opposed to general-purpose computers which handle a very wide set of applications. Examples of CE devices include digital cameras, TiVos, iPods, cell phones, and video game consoles. They may contain specialized electronics such as sensors, microphones, and cable decoders to allow them to perform their functions. They may or may not be programmable.

As predicted by Weiser [81], these computers disappear into the background because they present a specialized interface that is limited to the particular application for which they are designed. However, they are an important topic of research because the popularity of these devices has increased to the point where they dominate the computing experience of many users.

The most immediate difficulty is connecting CE devices to the distributed file system. Because many CE devices are not programmable and thus cannot execute distributed file system code, they cannot implement the distributed file system protocol and hence, cannot directly act as clients. Instead, we make another file system client act on its behalf, bridging the file system protocol and CE device interface.

Another interesting property of CE devices is that many of them are designed to be carried with the user. Consequently, they will be frequently disconnected from a distributed file system central file server. However, they may still be able to connect to each other, forming an ensemble of devices. They can still supply data and updates to each other and together, improving data availability, and present a consistent view of the file system. This extra flexibility in propagating data permits more data propagation than a distributed file system built around a central file server, while still preserving many of the beneficial properties of a central file server.

1.5 The road ahead

The remainder of this thesis is divided into 8 chapters. The second chapter provides a detailed motivation for the remainder of the work, describing how the advances presented here can be used together to enhance a consumer distributed file system. Chapters 3-5 describe features we are adding to the distributed file system to support type-awareness: persistent queries, sprockets, Trapper-Keeper. Chapters 6 and 7 discuss the methods we use to connect CE devices to the distributed file system, CE device integration and ensembles. Finally, chapter 8 describes related work and chapter 9 concludes this thesis.

CHAPTER 2

Background and Motivation

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

2.1 Integrating consumer electronics devices

Historically, when users possessed many storage devices, those devices were components of general-purpose computers such as a servers, desktops, and laptops. Many distributed storage solutions appeared for these computers [29, 50, 73, 80, 34] offering the convenience of automatic update propagation and the ability to access any data on any client by providing the illusion of having a single large shared disk. However, all of these systems make the assumption that the platform the distributed storage system runs on is programmable and that the interfaces provided by each device are similar.

CE devices violate these historical assumptions about programmability and interfaces and as a result, these distributed storage systems cannot incorporate these CE devices as clients. Devices such as iPod and digital cameras run custom software and generally do not permit users to run arbitrary code such as a distributed file system

client at either user or kernel level.

Instead of using a distributed storage system, users propagate data to and from CE devices by means of special-purpose software such as iTunes, gPhoto, or similar programs [6, 48, 25] that synchronize a CE device with a general-purpose computer. As new devices are added to the system, new software must also be added, increasing complexity and placing a greater burden on users who must learn how to operate all of these software interfaces. Even worse, no single system determines which data much be propagated to which devices to ensure availability of the most recent version of files, leaving that burden to the user. Due to the enormous popularity of CE devices, this is not only a substantial quantitative change in the fraction of user storage covered by distributed storage systems, but also a qualitative change in the usefulness of distributed storage — it can no longer cover all storage in regular use.

We adapted BlueFS [52], a distributed file system developed by our research group, as the basis for the EnsemBlue file system, our platform for the work in this dissertation. Our choice of BlueFS was driven by several factors. BlueFS aims to meet the storage needs of a single user or a small group of users such as a family — a good match for our focus on consumer-level data management.

In BlueFS, a central file server stores all files and receives file updates from clients. Clients send their updates to the central file server shortly after they are made by applications. This prevents updates from being lost in the event a client crashes, is destroyed, or otherwise lost; this is an important consideration for CE devices which are often small, portable, and valuable.

BlueFS also has support for mobile clients and storage. It allows clients to operate disconnected, a critical ability for devices such as MP3 players and laptops that often operate without a network connection. BlueFS has first class support for portable storage, tracking these devices independently of clients, allowing portable storage devices to be dynamically connected and removed from any client. We felt that this feature would be useful for CE devices for which we wanted to provide similar flexibility.

It is also designed to conserve the battery lifetime of mobile clients. BlueFS uses

a flexible cache hierarchy; whenever data must be read from storage, the file system chooses to service each request by using the most efficient storage device available. The device is selected by estimating the time and energy it would take to service the access on each storage option, taking into account the power management state of each device. The flexible cache hierarchy is likely to have storage options to choose from because whenever data is written by a client, that data is stored on all storage devices connected to that client as well as the central file server.

Since we focus on 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. BlueFS'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, BlueFS requires the user to choose which version to keep.

Over the past three years, 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 watching video 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.

2.2 Ensembles

Although the above techniques successfully connected CE devices to the file system, 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 band them together into an ensemble to exchange data between them when they were disconnected from the server. This arrangement not only improves the availability of data, as ensemble members can access data residing any ensemble member, but also improves freshness, as updates can be propagated between ensemble members.

2.3 Persistent queries

We also wished that BlueFS were more extensible. The heterogeneity of CE devices creates the need to customize file system behavior. For instance, many CE devices are associated with files of only one type; e.g., a digital camera with JPEGs. The network, storage, and processing resources of different CE devices 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.

An example of the customization that is needed is transcoding. Since several of our media players supported a limited set of formats, we found it necessary to

transcode files in order to make data available on all our devices. 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 prefer to extend the file system to do transcoding automatically.

We also found that the mechanisms for managing caching in BlueFS were insufficiently expressive. The existing 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.

Persistent queries are a file system facility that allows programs to receive notifications of file system activity. Notifications can be delivered to applications running on any EnsemBlue client; they are robust in the presence of failures and network disconnection. This allows many new file system features, including the examples above, to be constructed as user-level programs. For the transcoding example, a program on the powerful workstation can watch for the addition of new files and perform the transcoding. Similarly, in the second situation, by watching for the creation of files of a particular type, we can easily specify that those files be cached.

2.4 Sprockets

After a further year of experience with EnsemBlue, we returned to address another set of problems. Although we could now receive notifications to build applications that customize file system behavior, it was still not possible to precisely express the set of files we wanted to process. It could not handle queries not expressed in

terms of generic file metadata that EnsemBlue tracks. For instance, we were unable to create a query for MP3s by a certain artist, because artist information is type-specific metadata residing inside MP3 files. We needed a way to further customize the behavior of a distributed file system.

In addition to these immediate concerns, a mechanism to easily extend the file system would also create many new opportunities. In recent years, the file systems research community has proposed a number of new innovations that extend the functionality of storage systems. Yet, most production file systems have been slow to adopt such advances. This slow rate of change is a reasonable precaution because the storage system is entrusted with the persistent data of a computer system. However, if file systems are to adapt to new challenges presented by scale, widespread storage of multimedia data, new clients such as consumer electronic devices, and the need to efficiently search through large data repositories, they must change faster.

We will explore a method called *sprockets* that safely extends the functionality of distributed file systems. Our goal is to develop methodologies that let third-party developers create binaries that can be linked into the file system. Sprockets target finer-grained extensions than those supported by Watchdogs [11] and user level file system toolkits [21, 47], which offer extensibility at the granularity of VFS calls such as `read` and `open`. Sprockets are intended for smaller, type-specific tweaks to file system behavior such as querying type-specific metadata and resolving conflicts in distributed file systems. Sprockets have even been used to select among various formats of the same data depending on resource or performance constraints [77]. Sprockets are akin to procedure calls linked into the code base of existing file systems, except that they *safely* extend file system behavior.

2.5 TrapperKeeper

When the sprockets project was completed, we realized that most of the sprockets we had implemented were devoted to handling type-specific metadata. An example of this is a sprocket that has code that parses MP3 files in order to find all of the songs

performed by a particular artist. Building these sprockets was a laborious process, as it requires either building or using code that parses file types of interest.

More generally, this sort of type-awareness is increasingly important in modern storage systems. Traditionally, such systems managed files as a simple array of bytes; they were generally agnostic to the internal content of the files. However, current file systems store more files that have rich internal structure, such as music files that have ID3 headers, photos that have EXIF information, and documents that contain formatting data and information about their pedigree. Storage systems are quickly adding exciting new functionality based on understanding data internal to the files they store. For example, Apple's Spotlight tool [69], Windows Desktop Search [82], and Google Desktop [24] allow users to locate files based on internal metadata such as artist names in music files and comments in photo files. Graphical user interfaces (GUIs) such as the Mac OS X Finder and Gnome display icons preview what documents would look like if they were opened by applications that parse their particular file type.

However, much human effort is required to support this new functionality. The almost universal approach to understanding the internal structure of files is to require a software developer to write a plug-in that parses the file type and generates output for a general-purpose file system tool. For instance, given a new file type, a developer must write a plug-in to enable Spotlight to search through its metadata. She must write another parser to enable Google Desktop search for Windows, and yet another for Windows Desktop Search. To enable document preview, she must write a different parser to generate a suitable image for files of that type.

As the above discussion highlights, there are several problems with the existing plug-in approach to adding type-awareness to file systems:

- **It does not scale.** In total, developers must write a different parser for each file type, for each feature (e.g., preview and search), and for each file utility (e.g., Spotlight and Google Desktop). Thus, type-awareness has a large development cost; this cost is incurred not only during the initial creation of plug-ins, but also during the lifetime of that file type because developers must

keep the plug-ins in sync with the applications that parse that file type (e.g., document preview should reflect the current appearance of documents in their corresponding applications).

- **It inhibits innovation.** If an organization holds a dominant position in markets such as operating systems or search, then that organization is in a position to dictate to developers that they must write plug-ins to support new features. Developers will generally acquiesce since they want their applications to work correctly for the majority of their users. But, innovators, who often do not enjoy a position of dominance in the marketplace, are left out in the cold. It is hard to convince developers to write plug-ins for operating systems or utilities that currently have small market shares, since the developers will only satisfy a small percentage of their users in return for their hard work. If an innovator creates a new feature, such as a novel preview format, he will have a hard time convincing developers to support that feature. Thus, the innovator will usually find himself in the position of having to write plug-ins for most popular file types in order to bootstrap his innovation, even though he is probably unfamiliar with the applications that parse those file types.

- **It ignores the long tail.** While the most common file types may account for the majority of the files on a computer, the distribution of file types has a long tail. This means that even well-funded organizations willing to invest substantial amounts of software developer time may find it difficult to cover a very large percentage of files on a typical computer. For instance, by analyzing the trace data collected by Agrawal et al. [2], we found that even if one were to write plug-ins to support the 50 most popular file extensions, 24% of the files observed during a large-scale study of corporate file system usage would still not have a corresponding plug-in. Thus, the total development effort required to write plug-ins for a new feature is quite large. While it may be economically feasible for a large organization to write and support plug-ins for a few of the most popular file types, rarer file types will necessarily be unsupported. This

will create an intrusive disruption for users of the new feature because they must remember which file types are unsearchable, do not have previews, etc. when interacting with their computers.

TrapperKeeper is a solution to these problems. At the heart of this work is the observation that the application associated with a given file type already understands how to parse, manipulate, and display files of that type. Thus, there is no need to write separate plug-ins. Instead, TrapperKeeper uses virtualization to run such applications in isolation in order to extract specific features such as index terms or an image of a document being displayed.

By adding CE devices, persistent queries, ensembles, Sprockets, and TrapperKeeper to distributed file systems, we create a distributed storage system that better handles not only the new profusion of devices but also their growing multimedia content.

CHAPTER 3

Persistent Queries

Persistent queries are a robust event notification mechanism that lets users customize the file system with applications that automate tasks. Persistent query-based applications allow us to easily add features such as file transcoding and type-specific caching preferences on a file system-wide basis. This chapter describes the design rationale, implementation, performance, and experiences using persistent queries.

3.1 Design considerations

When we first decided to build custom functionality on top of EnsemBlue, the first design issue we considered was how tightly to integrate custom functionality with the file system. 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. This was an issue we would return to later. 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 [46] or the lame transcoder [38]. 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 [69], 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.*

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 [52], 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

`pq_create`	`(IN String query, IN Long event_mask, OUT Id fileid);`	Creates a query, returns its unique id
`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 next record in the query (if any)
`pq_truncate`	`(IN Int file_descriptor, IN Int record_number);`	Deletes records up to the specified record

Figure 3.1: Persistent query interface

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 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.

3.2 Implementation

Figure 3.1 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 notified 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 CE devices 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.

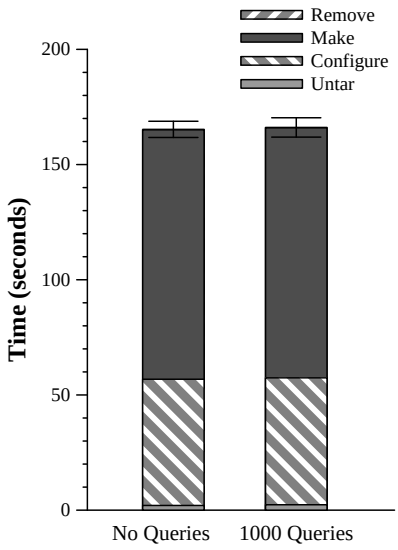
A drawback of making queries persistent is that queries that are no longer useful may accumulate over time. This can be addressed with a tool that periodically examines the outstanding queries and deletes ones that have not been used for a substantial period of time. Another potential drawback is that 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 usually does not require extra disk seeks.

3.3 Evaluation

We 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 an IBM X40 laptop with a 1.2 GHz Pentium M processor and 768 MB of RAM. 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



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 3.2: Overhead of persistent queries

which we untar the Apache 2.0.48 source tree into EnsemBlue, run `configure` in an object directory within EnsemBlue, run `make` in the object directory, and finally remove all files. Figure 3.2 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. 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 3.3.

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.

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 3.3: Time to create a persistent query

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.

3.4 Examples

To explore the success of this approach, 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 CE device 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.

3.5 Conclusion

The four applications above demonstrate the simplicity of using persistent queries to build custom features for distributed file systems. By reusing the distributed file cache coherence mechanism to provide reliable notifications, persistent queries provide an efficient way to extend file system functionality and will serve as an important component of our future work in both type-awareness and CE device integration.

CHAPTER 4

Sprockets

After the work on persistent queries, we could now build applications that customize file system behavior. However, in some cases, it was still not possible to precisely express the set of files we wanted to process. Persistent query criteria could only be specified in terms of metadata understood by the file system such as file names. Criteria such as the artist of an MP3, a value encoded in MP3 files, could not be used as the file system has no understanding of the MP3 file type that would allow it access to that information. We needed a way to add this type-awareness to the distributed file system. This chapter discusses *sprockets*, a method to safely extend the functionality of distributed file systems. Our goal is to develop methodologies that let third-party developers create binaries that can be linked into the file system. Sprockets are akin to procedure calls linked into the code base of existing file systems, except that the file system is protected from errors in sprockets.

While one might think that extending the behavior of a distributed file system requires one to alter kernel functionality, many distributed file systems such as AFS [29], BlueFS [52], and Coda [34] implement their core functionality at user level. It is these file systems that we target; extending file system functionality in the kernel can be accomplished through other methods [10, 18, 65, 71]. In many ways, extending user level code is easier than extending kernel code since the extension implementation can use operating system services to sandbox extensions to user level components. However, we have found that existing services such as `fork` are often prohibitively

expensive for commonly-used file system extensions that are only a few hundred lines of code. Further, isolation primitives such as `chroot` can be insufficiently expressive to capture the range of policies necessary to support some file system extensions.

The sprocket extension model is based upon software-fault isolation. Sprockets are easy to implement since they are executed in the address space of the file system. They may query existing data structures in the file system and reuse powerful functions in the code base that manipulate the file system abstractions. To ensure safety, sprockets execute inside a transaction that is always partially rolled back on completion, even if an extension executes correctly. A sprocket may execute arbitrary user level code to compute its results, but it must express those results in a limited buffer shared with the file system. Only the shared buffer is not rolled back at the end of sprocket execution. The results are verified by the core file system before changes are made to file state.

4.1 Design goals

What is the best way to extend file system functionality? To answer this question, we first outlined the goals that we wished to achieve in the design of Sprockets.

4.1.1 Safety

Our most important goal is safe execution of potentially unreliable code. The file system is critical to the reliability of a computer system — it should be a safe repository to which persistent data can be entrusted. A crash of the file system may render the entire computer system unusable. A subtle bug in the file system can lead to loss or corruption of the data that it stores [84]. Since the file system often stores the only persistent copy of data, such errors are to be avoided at all costs.

We envision that many sprockets will be written by third-party developers who may be less familiar with the invariants and details of the file system than core developers. Sprockets may also be executed more rarely than code in the core file system, meaning that sprocket bugs may go undetected longer. Thus, we expect the

incidence of bugs in sprockets to be higher than that in the core file system. It is therefore important to support strong isolation for sprocket code. In particular, a programming error in a sprocket should never crash the file system nor corrupt the data that the file system stores. A buggy sprocket may induce an incorrect change to a file on which it operates since the core file system cannot verify application-specific semantics within a file. However, the core file system can verify that any changes are semantically correct given its general view of file system data (e.g., that a file and its attributes are still internally consistent) and that the sprocket only modifies files on which it is entitled to operate.

Like previous systems such as Nooks [71], our design goal is to protect against buggy extensions rather than those that are overtly malicious. In particular, our design makes it extremely unlikely, but not impossible, for a sprocket to compromise the core file system. Our design also cannot protect against sprockets that intentionally leak unauthorized data through covert channels.

4.1.2 Ease of implementation

We also designed sprockets to minimize the cost of implementation. We wanted to make only minimal changes to the existing code of the core file system in order to support sprockets. We eliminated from consideration any design that required a substantial refactoring of file system code or that added a substantial amount of new complexity. We also wanted to minimize the amount of code required to write a new sprocket. In particular, we decided to make sprocket invocation as similar to a procedure call as possible.

Sprockets can call any function implemented as part of the core file system. Distributed file systems often consist of multiple layers of data structures and abstractions. A sprocket can save substantial work if it can reuse high-level functions in the core file system that manipulate those abstractions.

We also let sprockets access the memory image of the file system that they extend in order to reduce the cost of designing sprocket interfaces. If a sprocket could only

access data passed to it when it is called, then the file system designer must carefully consider all possible future extensions when designing an interface in order to make sure that the set of data passed to the sprocket is sufficient. In contrast, by letting sprockets access data not directly passed to them, we enable the creation of sprockets that were not explicitly envisioned when their interfaces were designed.

4.1.3 Performance

Finally, we designed sprockets to have minimal performance impact on the file system. Most of the sprockets that we have implemented so far can be executed many times during simple file system operations. Thus, it is critical that the time to execute each sprocket be small so as to minimize the impact on overall file system performance. Fortunately, most of the sprockets that we envision can be implemented with only a few hundred lines of code or less. These features led us to bias our choice of designs toward one that had a low constant performance cost per sprocket invoked, but a potentially higher cost per line of code executed.

An alternative to the above design bias would be batch processing so that each sprocket does much more work when it is invoked. Batching reduces the need to minimize the constant performance cost of executing a sprocket by amortizing more work across the execution of a single sprocket. However, batching would considerably increase implementation complexity by requiring us to refactor file system code wherever sprockets are used.

4.2 Alternative designs

In this section, we discuss alternative designs that we considered, and how these led to our current design.

4.2.1 Direct procedure call

There are many possible implementations for file system extensions. The most straightforward one is to simply link extension code into the file system and execute the extension as a procedure call. This approach is similar to how operating systems load and execute device drivers. Direct execution as a procedure call minimizes the cost of implementation and leads to good performance. However, this design provides no isolation: a buggy extension can crash the file system or corrupt data. As safety is our most important design goal, we considered this option no further.

4.2.2 Address space sandboxing

A second approach we considered is to run each extension in a separate address space. A simple implementation of this approach would be to `fork` a new process and call `exec` to replace the address space with a pristine copy for extension execution. This type of sandboxing is used by Web servers such as Apache to isolate untrusted CGI scripts. A more sophisticated approach to address sandboxing can provide better performance. In the spirit of Apache FastCGI scripts, the same forked process can be reused for several extension executions.

However, both forms of address space sandboxing suffer from two substantial drawbacks. First, they provide only minimal protection from persistent changes made by an extension through the execution of a system call. In particular, a buggy extension could corrupt file system data by incorrectly overwriting the data stored on disk. Potentially, such modifications could even violate file system invariants and lead to a crash of the file system when it reads the corrupted data. While operating systems do provide some tools such as the `chroot` system call and changing the effective userid of a process, these tools have a coarse granularity. It is hard to allow an extension access to only some operations, but not others. For instance, one might want to allow an extension that does transcoding to access only an input file in read mode and an output file in write mode. Restricting its privilege in this manner using the existing API of an operating system such as Linux requires much effort. Thus, address space

sandboxing does not provide completely satisfactory isolation on current operating systems.

A second drawback of address space sandboxing is that it considerably increases the difficulty of extension implementation. If the extension and the file system exist in separate address spaces, then the extension cannot access the file system's data structures, meaning that all data it needs for execution must be passed to it when it starts. Further, the extension cannot reuse functions implemented as part of the file system. While one could place code of potential interest to extensions in a shared library, the implementation cost of such a refactoring would be large.

4.2.3 Checkpoint and rollback

The above drawback led us to refine our design further to allow extensions to execute in the address space of the original file system. As before, the file system forks a new process to run the extension. However, instead of calling `exec` to load the executable image of the extension, the extension is instead dynamically loaded into the child's address space and directly called as a procedure. After the extension finishes, the child process terminates.

One way to view this implementation is that each extension executes as a transaction. However, in contrast to transactions that typically commit on success, these transactions are *always* rolled back. Since `fork` creates a new copy-on-write image, any modifications made to the original address space by the extension are isolated to the child process — the file system code never sees these modifications.

Extensions may often make persistent changes to file system state. Since it is unsafe to allow the extension to make such changes directly, we divide extension execution into two phases. During the first phase, the extension generates a description of the changes to persistent file state that it would like to make. This description is expressed in a format specific to each extension type that can be interpreted by the core file system. In the second phase, the core file system reads the extension's output and validates that it represents an allowable modification. This validation is

specific to the function expected of the extension and may be as simple as checking that changes are made only to specific files or that returned values fall within a permissible range.

If all validations pass, the core file system applies the changes to its persistent state. This approach is similar to that taken by an operating system during a system call. From the point of view of the operating system, the application making the call can execute arbitrary untrusted code; yet, the parameters of the system call can be validated and checked for consistency before any change to persistent state is made as a result of the call. This implementation relies on the fact that while the particular *policy* that determines what changes need to be made can be arbitrarily complex (and thus is best described with code), the *set of changes* that will be made as a result of that policy is often limited and can be expressed using a simple interface.

For example, consider the task of application-specific resolution, as is done in the Coda file system [35]. A resolver might merge conflicting updates made to the same file by reading both versions, performing some application-specific logic, and finally making changes that merge the conflicting versions into a single resolved file. While the application logic behind the resolution is specific to the types of files being merged, the possible result of the resolution is limited. The conflicting versions of the file will be replaced by new data. Thus, a extension that implements application-specific resolution can express the changes it wishes to make in a limited format such as a patch file that is easily interpreted by generic file system code. That core file system then verifies and applies the patch.

In the transactional implementation, the extension needs some way to return its result so that it can be interpreted, validated, and applied by the file system. We allow this by making the rollback at the end of extension execution partial. Before the extension is executed, the parent process allocates a new region of memory that it shares with its child. This region is exempted from the rollback when the extension finishes. The parent process instead reads, verifies, and applies return values from this shared region, and then deallocates it.

The transactional implementation still has some drawbacks. Like address space

isolation, we must rely on operating system sandboxing to limit the changes that an extension can make outside its own address space.

A second drawback occurs when the file system code being extended is multi-threaded. The extension operates on a copy of the data that existed in its parent's address space at the time it was forked. However, this copy could potentially contain data structures that were concurrently being manipulated by threads other than the one that invoked the extension. In that case, the state of the data structures in the extension's copy of the address space may violate expected invariants, causing the extension to fail. Ideally, we would like to fork an extension only when all data structures are consistent. One way to accomplish this would be to ask extension developers to specify which locks need to be held during extension execution. We rejected this alternative because it requires each extension developer to correctly grasp the complex locking semantics of the core file system. Instead, the extension infrastructure performs this task on behalf of the developer by relying on the heuristic that threads that modify shared data should hold a lock that protects that data. We use a barrier to delay the `fork` of an extension until no other threads currently hold a lock. This policy is sufficient to generate a clean copy as long as all threads follow good programming practice and acquire a lock before modifying shared data.

A final substantial drawback is that `fork` is a heavyweight operation on most operating systems: when an extension consists of only a few hundred lines of code, the time to fork a process may be an order of magnitude greater than the time to actually execute the extension. During `fork`, the Linux operating system copies the page table of the parent process—this cost is roughly proportional to the size of the address space. For instance, we measured the time to fork a 64 MB process as 6.3 ms on a desktop running the Linux 2.4 operating system [51]. This cost does not include the time that is later spent servicing page faults due to flushing the TLB and implementing copy-on-write. Overall, while the transactional implementation offers reasonably good safety and excellent ease of implementation, it is not ideal for performance because of the large constant cost of `fork`.

4.3 Sprocket design and implementation

Performance considerations led to our current design for sprocket implementation, which is to use the transactional model described in the previous section but to implement those transactions using a form of software fault isolation [79] instead of using address space isolation through `fork`.

4.3.1 Adding instrumentation

We use the PIN [45] binary instrumentation tool to modify the file system binary. PIN generates new text as the program executes using rules defined in a PIN *tool* that runs in the address space of the modified process. The modified text resides in the process address space and executes using an alternate stack. The separation of the original and modified text allows PIN to be turned on and off during program execution. We use this functionality to instrument the file system binary only when a sprocket is executing. Instrumenting and generating new code for an application is a very expensive operation, but the instrumentation must be performed only once for each instruction. Unfortunately, since PIN is designed for dynamic optimization, it does not support an option (available in many other instrumentation tools) to statically pre-instrument binaries before they start running. To overcome this artifact of the PIN implementation, we can pre-instrument sprockets by running them once on dummy data when the file system binary is first loaded.

We have implemented our own PIN tool to provide a safe execution environment in which to run sprockets. When a sprocket is about to be executed, the PIN instrumentation is activated. Our PIN tool first saves the context of the calling thread (e.g., register states, program counter, heap size, etc.). As the sprocket executes, for each instruction that writes memory, our PIN tool saves the original value and the memory location that was modified to an undo log.

When the sprocket completes execution, each memory location in the undo log is restored to its original value and the program context is restored to the point before the sprocket was executed. The PIN tool saves the sprocket's return code and passes

this back to the core file system as the return value of the sprocket execution. Like the fork implementation, the sprocket infrastructure allocates a special region of memory in the process address space for results — modifications to this region are not rolled back at the end of sprocket execution. If sprocket execution is aborted due to an exception, bug, or timeout, the PIN tool substitutes an error return code. Prior to returning, the PIN tool disables instrumentation so that the core file system code executes at native speed.

The ability to dynamically enable and disable instrumentation is especially important since sprockets often call core file system functions. When the sprocket executes, PIN uses a slow, instrumented version of the function that is used during all sprocket executions. When the function is called by the core file system, the original, native-speed implementation is used. Instrumented versions are cached between sprocket invocations so that the instrumentation cost need be paid only once.

Running the instrumented sprocket code, which saves modified memory values to an undo log, is an order of magnitude slower than running the native, uninstrumented version of the sprocket. However, since most sprockets are only a few hundred lines of code, the total slowdown due to instrumentation can be substantially less than the large, constant performance cost of `fork`.

We perform a few optimizations to improve the performance of binary instrumentation. We observed that many modifications to memory occur on the stack. By recording the location of the stack pointer when the sprocket is called, we can determine which region of the stack is unused at the point in time when the sprocket executes. We neither save nor restore memory in this unused region when it is modified by the sprocket. Similarly, we avoid saving and restoring areas of memory the sprocket allocates using `malloc`. Finally, we avoid duplicate backups of the same address.

Binary instrumentation also allows us to implement fine-grained sandboxing of sprocket code. Rather than rely on operating system facilities such as `chroot`, we use PIN to trap all system calls made by the sprocket. If the system call is not on a whitelist of allowed calls, described in Section 4.3.3, the sprocket is terminated with

an error. Calls on the whitelist include those that do not change external state (e.g., `getpid`). We also allow system calls that enable sprockets to read files but not modify them.

4.3.2 Sprocket interface

Figure 4.1 shows an example of how sprockets are used. From the point of view of the core file system, sprocket invocation is designed to appear like a procedure call. Each sprocket is passed a pointer argument that can contain arbitrary data that is specific to the type of sprocket being invoked. Since sprockets share the file system address space, the data structure that is passed in may include pointers. Alternatively, a sprocket can read all necessary data from the file system's address space.

The `SPROCKET_SET_RETURN_DATA` macro allocates a memory region that will hold the return value. In the example in Figure 4.1, this region is one memory page in size. The `DO_SPROCKET` macro invokes the sprocket and rolls back all changes except for data modified in the designated memory region. In the example code, the core file system function `get_needed_data` parses and verifies the data in the designated memory region, then deallocates the region. As shown in Figure 4.1, the core file system may also include error handling code to deal with the failure of sprocket execution.

4.3.3 Handling buggy sprockets

Sprockets employ a variety of methods to prevent erroneous extensions from affecting core file system behavior and data. Because changes to the process address space made by a sprocket are rolled back via the undo log, the effects of any sprocket bug that stomps on core file system data structures in memory will be undone during rollback. Similarly, a sprocket that leaks memory will not affect the core file system. Because the data structures used by malloc are kept in the process address space, any memory allocated by the sprocket is automatically freed when the undo log is replayed and the address space is restored. Additional pages acquired by memory

```
/* Arguments passed to sprocket */
help_args.buf = NULL;
help_args.len = 0;
help_args.file1_size = server_attr.size;
help_args.file2_size = client_file_stat.st_size;

/* Set up return buffer and invoke sprocket */
SPROCKET_SET_RETURN_DATA (help_args.shared_page, getpagesize());
rc = DO_SPROCKET(resolver_helper, &help_args);

if (rc == SPROCKET_SUCCESS) {
    /* Verify and read return values */
    get_needed_data(help_args.shared_page, &help_args,
                    NULL, fid, &server_attr, path);
} else {
    /* handle sprocket error */
    ...
```

Figure 4.1: Example of sprocket interface

allocation during sprocket execution are deallocated with the `brk` system call.

Other types of erroneous extensions are addressed by registering signal handlers before the execution of the socket. For instance, if a sprocket dereferences a NULL pointer or accesses an invalid address, the registered `segfault` handler will be called. This handler sets the return value of the sprocket to an error code and resets the process program counter in the saved execution context passed into the handler to the entry point of the rollback code. Thus, after the handler finishes, the sprocket automatically rolls back the changes to its address space, just as if the sprocket had returned with the specified error code. To handle sprockets that consume too much CPU (e.g., infinite loops), the sprocket infrastructure sets a timer before executing the extension.

The final type of errors currently handled by sprockets are erroneous system calls. Sprockets allow fine-grained, per-system-call capabilities via a *whitelist* that specifies the particular system calls that a sprocket is allowed to execute. We enforce the whitelist by using the PIN binary instrumentation tool to insert a check before the execution of each system call. If the system call being invoked by a sprocket is not on its whitelist, the sprocket is aborted and rolled back with an error code.

We support per-call handling for some system calls. For instance, we keep track of the file descriptors opened by each sprocket. If a sprocket attempts to close a descriptor that it has not itself opened, we roll back the sprocket and return an error. Similarly, after the sprocket finishes executing, our rollback code automatically closes any file descriptors that the sprocket has left open, preventing it from leaking a consumable resource.

One remaining way in which a buggy sprocket can affect the core file system code is to return invalid data via the shared memory buffer. Unfortunately, since the return values are specific to the type of sprocket being invoked, the sprocket execution code cannot automatically validate this buffer. Instead, the code that invokes the sprocket performs a sprocket-specific validation before using the returned values. For instance, one of our sprockets (described in Section 4.4.2) returns changes to a file in a `patch`-compatible file format. The code that was written to invoke that particular sprocket

verifies that the data in the return buffer is, in fact, compatible with the patch format before using it.

4.3.4 Support for multithreaded applications

Binary instrumentation introduces a further complication for multithreaded programs: other threads should never be allowed to see modifications made by a sprocket. This is an important consideration for file systems since most clients and servers are designed to support a high level of concurrency. We first discuss our current solution to multithreaded support, which is most appropriate for uniprocessors, and then discuss how we can extend the sprocket design in the future to better support file system code running on multiprocessors.

Our current design for supporting multithreaded applications relies on the observation that the typical time to execute a sprocket (0.14–0.62 ms in our experiments) is much less than the scheduling quantum for a thread. Thus, if a thread would ordinarily be preempted while a sprocket is running, it is acceptable to let the thread continue using the processor for a small amount of time in order to complete the sprocket execution. If the sprocket takes too long to execute, its timer expires and the sprocket is aborted. Effectively, we extend our barrier implementation so that sprockets are treated as a critical section; no other thread is scheduled until the sprocket is finished or aborted. Although our barrier implementation is slightly inefficient due to locking overheads, we would require a more expressive interface such as Anderson's scheduler activations [4] to utilize a kernel-level scheduling solution.

On a multiprocessor, the critical section implementation has the problem that all other processors must idle (or execute other applications) while a sprocket is run on one processor. If sprockets comprise a small percentage of total execution time, this may be acceptable. However, we see two possible solutions that would make sprockets more efficient on multiprocessors. One possibility would be to also instrument core file system code used by other threads during sprocket execution. If one thread reads a value modified by another, the original value from the undo log is supplied instead.

This solution allows other threads to make progress during sprocket execution, but imposes a performance penalty on those threads since they also must be instrumented while a sprocket executes.

An alternative solution is to have sprockets modify data in a shadow memory space. Instructions that read modified values would be changed to read the values from the shadow memory rather than from the normal locations in the process address space. For example, Chang and Gibson [14] describe one such implementation that they used to support speculative execution.

4.4 Sprocket uses

In order to examine the utility of sprockets, we have taken two extensions proposed by the file systems research community and implemented them as sprockets. The next subsections describe our implementation of transducers and application-specific conflict resolution.

4.4.1 Transducers

The first type of sprocket implements application-specific semantic queries over file system data. The functionality of this sprocket is similar to that of a transducer in the Semantic File System [22] or in Apple's Spotlight [69] in that it allows users to search and index type-specific attributes contained within files. For example, one might wish to search for music produced by a particular artist or photos taken on a specific date. This information is stored as metadata within each file (in the ID3 tag of music files and in the JPEG header of photos). However, since the organization of metadata is type-specific, the file system must understand the metadata format before it can search or index files of a given type. Our sprocket transducers extend EnsemBlue by providing this type-specific knowledge.

To support type-specific metadata, we extended the persistent query interface to allow applications to optionally specify a sprocket that will be called to help evaluate the query. For each potential match, the server first performs the generic type-

independent evaluation (for instance, the query might verify that the filename ends in ".mp3"). If the generic evaluation returns true, the server invokes the sprocket specified for the query.

The query sprocket reads the type-specific metadata from the file, evaluates the contents, and returns a boolean value that specifies whether or not the file matches the query. If the sprocket returns true, the server appends a record to the persistent query object; the server takes no action if the sprocket returns false.

Reading data from a server file is a relatively complex operation. File data may reside in one of three places: in a file on disk named by the unique EnsemBlue identifier for that file, in the write-ahead log on the server's disk, or in a memory cache that is used to improve read performance. Executing the sprocket within the address space of the server improves performance because the sprocket can reuse the server's memory cache to avoid reading data from disk. Further, when the cache or write-ahead log contains more recent data than on disk, executing the sprocket in the server's address space avoids the need to flush cached data and truncate the write-ahead log. If the sprocket were a stand-alone process that only read data from the on-disk file, then it would read stale data if the cache were not flushed and the write-ahead log truncated.

The sprocket design considerably reduces the complexity of transducers in EnsemBlue. The sprocket can reuse existing server functions that read data and metadata from the diverse sources (cache, log, and disk storage). These functions also encapsulate EnsemBlue-specific complexity such as the organization of data on disk (e.g., on-disk files are hashed and stored in a hierarchical directory structure organized by hash value to improve lookup performance). Due to this reuse, the code size of our transducers is relatively small. For example, a transducer that we wrote to search ID3 tags and return all MP3 files with a specific artist required only 239 lines of C code.

4.4.2 Application-specific resolution

The second type of sprocket performs application-specific resolution similar to that proposed by Kumar et al. for the Coda file system [35]. Like Coda, EnsemBlue uses optimistic concurrency and supports disconnected operation. Therefore, it is possible that concurrent updates may be made to a file by different clients. When this occurs, the user is normally asked to manually resolve the conflict. As anyone who has used CVS knows, manual conflict resolution is a tedious and error-prone process.

Kumar et al. observed that many types of files have an internal structure that can be used by the file system to *automatically* resolve conflicts. For example, if one client adds an artist to the ID3 tag of an MP3 file, while another client adds a rating for the song, a file system with knowledge of this data type can determine that the two updates are orthogonal. An automatic resolution for these two updates would produce a file that contains both the new artist and rating. However, like the transducer example in the previous section, EnsemBlue cannot perform such automatic resolution because it lacks the required knowledge about the data type.

To allow for automatic conflict resolution, we extended the conflict handling code in the EnsemBlue client daemon to allow for the optional invocation of a handler for specific data types. When the daemon tries to reintegrate an update that has been made on its client to the server, the server may detect that there has been a conflicting update made by another client (EnsemBlue stores a version number and the identifier of the last client to update the file in order to detect such conflicts). The client daemon then checks to see if there is a conflict handler registered for the data type (specifically, it checks to see if the name of the file matches a regular expression such as files that end in ".mp3"). If a match is found, the daemon invokes the sprocket registered for that data type.

Our original design had the sprocket do the entire resolution by reading and fetching the current version of the file stored at the server, comparing it to the version stored on the client, and then writing the result to a temporary file. However, this

approach was unsatisfying for two reasons. First, it violated our rule that sprockets should never persistently change state. The design required the sprocket to communicate with the server, which is an externally visible event that changes persistent state on the server. The communication increments sequence numbers and perturbs the next message if the stream is encrypted. Second, the design did not promote reuse. Each resolution sprocket must separately implement code to fetch data from the server, read data from the client, and write the result to a temporary file.

Based on these observations, we refactored our design to perform resolution with two separate sprockets. The first sprocket determines the data to be fetched from the server; it returns this information as a list of data ranges. For example, an MP3 resolver would return the bytes that contain the ID3 tag. After executing the sprocket, the daemon fetches the required data. The first sprocket may be invoked iteratively to allow it to traverse data structures within a file. Thus, the work that is generic and that makes persistent changes to file system state is now done outside of the sprocket. A second benefit of this approach is that only a limited subset of a file's data needs to be fetched from the server; for large multimedia files, this substantially improves performance.

The daemon passes the second sprocket the range of data to examine that was output by the first sprocket, as well as the corresponding data in the client and server versions of the file to be resolved. The second sprocket performs the resolution and returns a *patch* that contains regions of data to add, delete, or replace in the server's version of the file. The daemon validates that the patch represents an internally consistent update to the file (e.g., that the bytes being deleted or replaced actually exist within the file). It sends the changes in the patch file to the server to complete the resolution. This design fits the sprocket model well since the format of the patch is well understood and can be validated by the file system before being applied; yet, the logic that generates the patch can be arbitrarily complex and reside within the sprocket. A bug in the sprocket could potentially produce an invalid ID3 header; however, since the application-specific metadata is opaque to the core file system, such a bug could not lead to a subsequent crash of the client daemon or server.

We have written an MP3 resolver that compares two ID3 tags and returns a new ID3 tag that merges concurrent updates from the two input tags. The first sprocket is invoked twice to determine the byte range of the ID3 tag in the file. The second sprocket performs the resolution and requests that the daemon replace the version of the ID3 tag at the server with a new copy that contains the merged updates. Typically, the patch contains a single entry that replaces the data in the original ID3 tag. However, if the size of the ID3 tag has grown, the patch may also request that additional bytes be inserted in the file after the location of the original ID3 tag. These two sprockets required a combined 474 lines of C code.

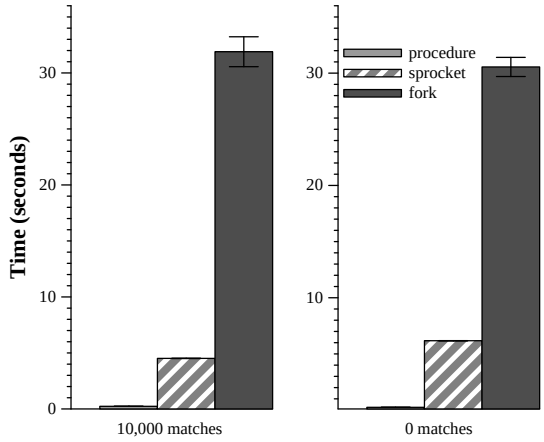
4.5 Evaluation

Our evaluation answers the following questions:

- What is the relative performance of extensions implemented through binary instrumentation, address-space sandboxing, and direct procedure calls?
- What are the effects of our binary instrumentation optimizations on performance?
- What is the isolation provided for extensions implemented through binary instrumentation, address-space sandboxing, and direct procedure calls?

4.5.1 Methodology

For our evaluation we used a single computer with a 3.02 GHz Pentium 4 processor and 1 GB of RAM — this computer acts as both a EnsemBlue client and server. The computer runs Red Hat Enterprise Linux 3 (kernel version 2.4.21-4). When a second EnsemBlue client is required, we add a IBM T20 laptop with a 700 MHz Pentium III processor and 128 MB of RAM connected over a 100 Mbps switch. The IBM T20 runs Red Hat Linux Enterprise 4 (kernel version 2.6.9-22). Each EnsemBlue client is configured with a 500 MB write log and does not cache data on disk. We used



This figure compares the time to create a persistent query that lists MP3 songs by the band Radiohead using extensions implemented via procedure call, sprocket, and `fork`. The graph on the left shows the results when the file system contains 10,000 files, all of which match the persistent query; the graph on the right shows the results when none match. Each result is the mean of five trials — error bars show 90% confidence intervals.

Figure 4.2: Performance of the Radiohead transducer

the PIN toolkit version 3585 compiled for gcc version 3.2. All results were measured using the `gettimeofday` system call.

4.5.2 Radiohead transducer

The first experiment measures the performance of a transducer sprocket that determines whether or not an MP3 has the artist tag: "Radiohead". The sprocket is executed by the EnsemBlue server to generate the contents of a persistent query. It directly calls core file system functions within the file server to read the id3 tag of a file; it then parses the tag to determine the artist name. The sprocket returns a boolean value (whether or not the artist matches). If the value is true, the file server adds the file being evaluated to the results of the persistent query. This sprocket is a good example of the benefits of providing an extensible interface to the file system. The sprocket is implemented with only 239 lines of code, yet it allows the user to semantically search files without having the file system understand type-specific file formats.

Figure 4.2 shows the performance of the sprocket in two different scenarios. In the

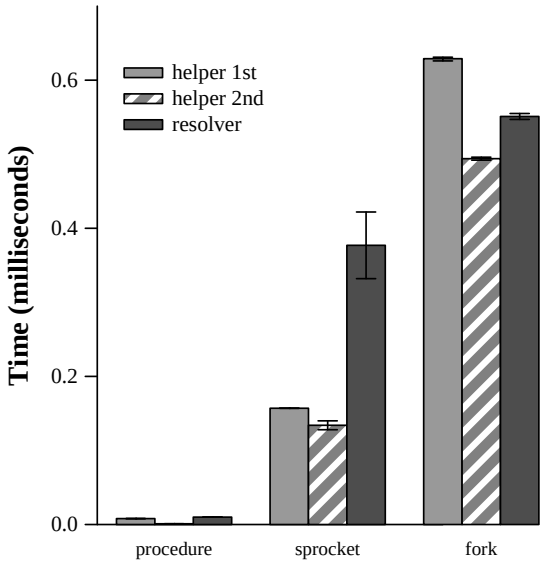
left hand graph, the file system is first populated with 10,000 MP3 files with the id3 tag designating Radiohead as the artist. The first bar in the graph shows the time to run the sprocket 10,000 times (for each file in the file system) and generate the persistent query when the sprocket is executed as a function call inside the EnsemBlue address space. As expected, executing the sprocket as a function call is extremely quick since no isolation is provided. The second bar shows the performance running the sprocket with PIN and our checkpoint and rollback tool. Using binary instrumentation slows down performance by a factor of 20, but it ensures that a buggy sprocket will not affect the execution of the server or the data it stores.

The last bar in the graph shows the performance when executing the sprocket using `fork` as described in section 4.2.3. While `fork` provides many of the same benefits as the PIN implementation, its performance costs are much higher. Using `fork` is 120 times slower than executing as a function call, and an order of magnitude slower than using PIN. When a sprocket's functionality is simple, the cost of binary instrumentation is outweighed by the costs of copying the file server's page table when executing `fork`.

The right hand graph in Figure 4.2 shows the performance of the Radiohead transducer when EnsemBlue is populated with 10,000 MP3 files, none of which are by the band Radiohead. Therefore, the resulting persistent query will be empty. The results of the second experiment are similar to the first. However, the extension executes more code in this scenario because it checks for the possible existence of a version 2 ID3 tag when it finds that no version 1 ID3 tag exists. In the first experiment, the second check is never executed. The additional code has a proportionally greater affect on the sprocket implementation because of its high per-instruction cost.

4.5.3 Application-specific conflict resolution

The next experiment measures the performance of a set of sprockets that resolve a conflict within the id3 tag of an MP3. When a client sends operations to the server (e.g., a file system write) that conflicts with the version at the server, the file system



This figure compares performance when resolving a conflict using an application-specific ID3 tag resolver using procedure call, sprocket, and fork-based implementations. A helper extension is invoked twice to determine which data needs to be resolved, and a resolver extension performs the actual resolution. Each bar shows the time to resolve conflicts with 100 files. Each result is the mean of five trials — error bars show 90% confidence intervals.

Figure 4.3: Application-specific conflict resolution

client first attempts to invoke a sprocket to resolve the conflict before requiring the user to resolve the conflict manually. We populated EnsemBlue with 100 3 MB MP3 files. We then modified two different fields within the ID3 tag of each file on two different EnsemBlue clients. After ensuring one client had reconciled its log with the server, we flushed the second client's log, creating a conflict in each of the 100 files. The client then invokes two sprockets to resolve the conflict. The first sprocket is invoked twice to determine where the ID3 tag is located in the file. The first invocation reads the ID3 header, which determines the size of the rest of the tag; the second invocation reads the rest of the tag. The second sprocket creates a patch that resolves the conflict. This process is repeated for each of the 100 files that are in conflict.

Figure 4.3 shows the performance of each implementation. The sprocket implementation is substantially faster than the fork-based implementation on all extensions, though the difference in performance is greater for the first two helper invocations (because they execute fewer instructions). The resolver extension is still faster with

optimization	transducer	conflict resolver
Malloc (total)	0.00%	2.78%
Duplicate (total)	82.44%	81.71%
Stack (total)	99.45%	93.35%
Malloc (unique)	0.00%	2.01%
Duplicate (unique)	0.38%	2.68%
Stack (unique)	17.39%	15.08%

This table shows the fraction of memory backups prevented by the three optimizations. The first three rows show the fraction of memory backups prevented by each optimization. The second three rows show the fraction of memory backups prevented by only that optimization and no other. Results are the average of five runs of a single execution of each extension.

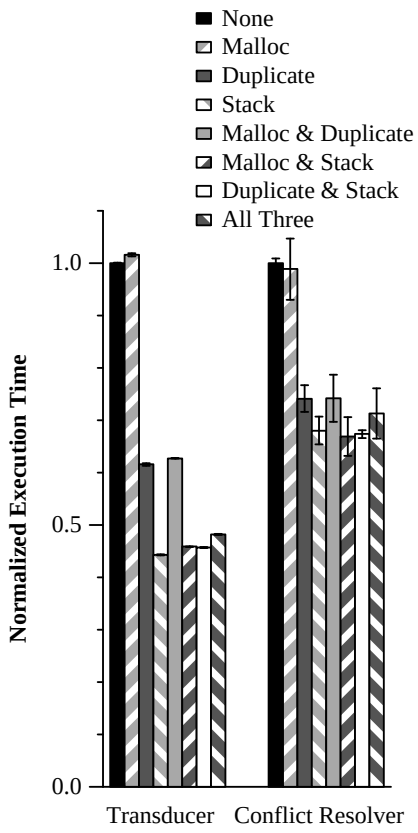
Table 4.1: Effects of optimizations

the sprocket implementation than with `fork`, but shows a substantially smaller advantage than the others. This is because the resolver extension runs longer, causing the cumulative cost of executing instrumented code to approach the cost of `fork`. While the performance of binary instrumentation might improve with further code optimization, we believe that sprockets of substantially greater complexity than this one should probably be executed using `fork` for best performance.

4.5.4 Optimizations

Given this set of experiments, we next measured the effectiveness of our proposed binary instrumentation optimizations which eliminate saving and restoring data at addresses allocated by the sprocket, duplicated in the undo log, or in the section of the stack used by the sprocket. These three techniques are intended to improve performance by inserting an inexpensive check before each write performed by the sprocket that tests whether overwritten data needs to be saved in the undo log.

Since each optimization adds a test that is executed before each write, optimizations must provide a substantial reduction in logging to overcome the cost of testing. Figure 4.4 shows the time taken for both the Radiohead transducer and conflict re-



This figure shows effects of combinations of our optimizations on the performance of our sprocket tests: the Radiohead transducer with 10,000 matches and a run of the application specific conflict resolver. Results are the average of five pre-instrumented executions with 90% confidence intervals and are normalized to the unoptimized performance.

Figure 4.4: Optimization performance

solver with combinations of optimizations turned on.

To understand these results, we first measured the fraction of writes each optimization prevents from creating an undo log entry. As shown in the upper half Table 4.1, avoiding either stack writes or duplicates of already logged addresses prevents almost all new log entries. For these sprockets, the malloc optimization is less effective; the Radiohead transducer does not use malloc and the conflict resolver performs few writes to the memory it allocates.

Seeing the large overlap in writes covered by these optimizations, we next investigated how much each contributed to the total reduction in logging. The lower half of Table 4.1 shows the fraction of writes that are uniquely covered by each optimization. In this view, the malloc optimization looks more useful as writes it covers are usually

Buggy Sprocket	fcn call result	fork result	PIN result
Memleak	crash	correct	correct
Memstomp	crash	correct	correct
Call exec	exec & exit	exec executed	sprocket terminated
Segfault	crash	correct	correct
File leak	crash	correct	correct
Infinite loop	hang	sprocket terminated	sprocket terminated
Wrong close	hang	correct	sprocket terminated

This table shows the results when a buggy sprocket is executed under three different execution environments. "Correct" means that the sprocket completed successfully without a negative effect on the EnsemBlue file system. "Sprocket terminated" means that PIN detected a problem and preemptively rolled back the sprocket's actions and returned an error on its behalf.

Figure 4.5: Results of buggy sprockets

not covered by the other optimizations.

Since the Radiohead transducer sprocket does not use malloc, the malloc optimization simply imposes additional cost. For this sprocket, the stack optimization alone is the most effective; adding the duplicate optimization prevents an additional 0.38% of writes from creating undo log entries, but this benefit is less than the cost of its test on every write.

On the conflict resolver sprocket, the effects are somewhat different. Again, the stack optimization is the most effective. Adding the other optimizations produces no significant difference. This suggests that very simple tests, such as the malloc optimization, that test if the address to be logged is within a certain range, can break even if they prevent around 2% of writes from triggering logging.

4.5.5 When good sprockets go bad

Since we expect that some number of sprockets will exhibit buggy behavior, we implemented three of our own buggy sprockets and recorded the effects on the file system, which are shown in Figure 4.5. The first buggy sprocket emulates a memory

leak by allocating a 10 MB buffer and then returning control to the calling process. We invoked the sprocket 10,000 times and recorded the results. When run as a function pointer, the file system crashed as it quickly used up its entire address space. When `fork` is used, the address space is reclaimed each time the sprocket exits, so there are not any negative effects. Likewise, our PIN tools' rollback mechanism undoes the memory allocation, and the sprocket executes correctly.

The second row of Figure 4.5 shows the results of a buggy sprocket that overwrites an important data structure in the EnsemBlue server address space (the pointer to the head of the write-ahead log) with `NULL`. As expected, the sprocket crashes the server when run as a function pointer, and it has no effect when run in a forked address space. Finally, when run within our PIN tool, the sprocket's actions have no effect on the server, as the memory stomp is undone when the sprocket completes.

The last row of Figure 4.5 shows the results when a sprocket is run that attempts to execute a new program by calling the `exec` system call. When executed as a function call, the server simply ceases to exist since its address space is replaced by `exec`. With our PIN tool, the system call whitelist detects that a sprocket is attempting a disallowed system call. The tool immediately rolls back the sprocket's execution, and returns an error to the file system. The `fork` implementation allows the sprocket to `exec` the specified executable, which is probably not a desirable behavior.

4.6 Conclusion

Sprockets are designed to be a safe, fast, and easy-to-use method for extending the functionality of distributed file systems implemented at user level. Our results are encouraging in many respects. We were able to implement almost every sprocket that we attempted in a few hundred lines of code. Our sprocket implementation using binary instrumentation caught several serious bugs that we introduced into extensions and allowed the file system to recover gracefully from programming errors. Sprocket performance for very simple extensions can be an order of magnitude faster than a `fork`-based implementation. Yet, we also found that there are upper limits to the

amount of complexity that can be placed in a sprocket before binary instrumentation becomes more expensive than `fork`. Our ID3 tag resolver lands approximately near this balancing point. In general, we believe that sprockets are a promising technique for meeting the type-awareness needs of distributed file systems.

CHAPTER 5

TrapperKeeper

TrapperKeeper is a system that assists the development of type-aware file system functionality. Existing plug-in-based architectures require software developers to write and maintain separate code modules for each new file type. TrapperKeeper aims to simplify this development by supplying type-specific information without requiring type-specific code. TrapperKeeper executes existing software applications that parse the desired file type inside virtual machines. It then uses accessibility APIs to control the application and extract information from the application's graphical user interface. We have implemented metadata extraction and document preview features that use TrapperKeeper, and we have used TrapperKeeper to capture the type-specific cognizance of over 20 applications that collectively parse more than 100 distinct file types. Our experimental results show that TrapperKeeper can execute these two features on hundreds of files per hour, a pace that far exceeds the rate that files are modified or created on the average desktop.

We begin in the next section by discussing the goals we hope to achieve with the design of TrapperKeeper. Section 5.2 describes our implementation, and Section 5.3 details our file indexing and document preview features. Sections 5.4 describes our evaluation.

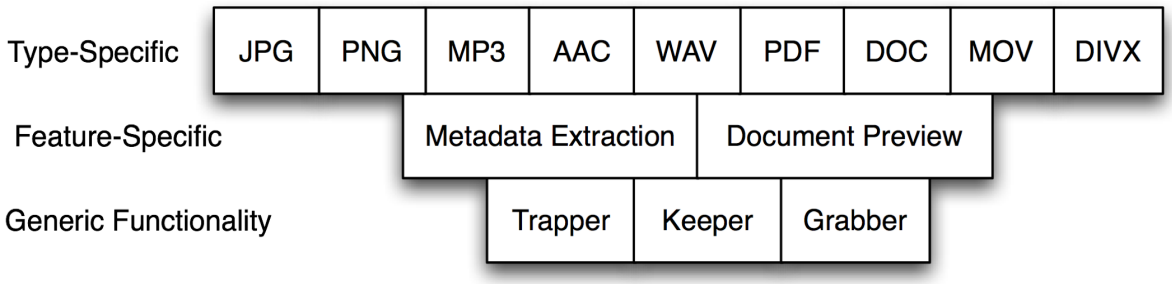


Figure 5.1: Tiers of increasing specificity

5.1 Design goals

Before we started on this project, we listed the most important properties for a type-awareness system to have. These properties differentiate TrapperKeeper from previous type-awareness systems.

5.1.1 Minimize work per file type

Ideally, we would like to allow new file types to benefit from features such as metadata indexing and document preview without requiring any additional work to support the new type. While this goal may be impossible to satisfy fully, we can take a large step toward achieving it by moving functionality out of type-specific components and into type-agnostic components, such as feature-specific code and TrapperKeeper's components: Trapper, Keeper, and Grabber.

As shown in Figure 5.1, our design for TrapperKeeper has three tiers. The bottom tier, which consists of Trapper, Keeper, and Grabber, supplies generic functionality that is common across different features and file types. For instance, Trapper and Keeper checkpoint and resume virtual machines, while Grabber implements shared functionality to query and manipulate GUIs through their accessibility APIs.

The second tier is feature-specific functionality. For instance, our metadata extraction feature uses Grabber to query the data in text fields and other GUI widgets. Our document preview feature uses Grabber to generate a screen snapshot of the

application within the virtual machine. Functionality in this tier may have to be implemented for several different features, so we prefer to move any feature-agnostic code to the bottom tier. However, we expect the number of features to be quite small, say 10-20, compared to the number of file types, which we estimate to be several thousand based on our analysis of trace data in Section 5.4.6. Thus, it is reasonable to have a small amount of feature-specific functionality.

The top tier is type-specific functionality. Some examples of type-specific functionality are manipulating an application to generate the best screen shot for a document preview and selecting particular widgets that contain metadata to index. We wish to avoid implementing type-specific behavior whenever possible since such work is potentially multiplied by thousands of file types.

When we cannot avoid type-specific tasks, we aim to make them as easy as possible. For instance, TrapperKeeper can amortize the support for a single application across many distinct file types. Since TrapperKeeper operates on the application GUI, which is typically common for all file types, a single application snapshot can often be used for all types parsed by an application (e.g., for all image types accepted by an image viewer). Making type-specific tasks easier is also the motivation for our next design goal.

5.1.2 No code per file type

Current indexing tools such as Spotlight [69], Windows Desktop Search [82], and Google Desktop [24] rely on software developers to create and maintain plug-ins to parse files. Each plug-in is a piece of code that parses a particular file type, such as JPEGs or MP3s.

Although this model of plug-in development is acceptable for the few most popular file types, there are a large number of less popular file types for which the benefit gained by adding them to the metadata system is outweighed by the development costs of the associated plug-in. Further, the different metadata systems require varying functionality from the plug-in, compounding the effort. For many file types, the

original developers may be the only ones who know how to parse that file type. Yet, these developers may have gone out of business, lost interest, forgotten how to parse the file type, declared the file type obsolete, or may not have the engineering resources required to build plug-ins for each metadata system.

Metadata features that require the writing of new code for each file type will always run afoul of these problems. Thus, our approach is to *eliminate* all code specific to a file type. Instead, we run an application associated with each file type (e.g., Exaile for music files) and use the application to parse files of that type. When some human guidance is required, that guidance is provided through the application's GUI. A user runs an application using Trapper and clicks on widgets in the application's GUI to indicate which metadata should be extracted.

Our "no type-specific code" manifesto aims to change the economics of supporting file types. With TrapperKeeper, users with no programming expertise who are familiar with an application can extract information from new file types simply by manipulating an application's GUI.

5.1.3 Isolation

Our third design goal is to isolate applications used to parse files for features such as metadata extraction from the rest of the computer system. In plug-in-based systems, the plug-ins are typically run as part of a background process. Using applications to provide parsing behavior requires care because these applications often have complex, stateful interactions with other parts of the computing system such as the network, file system, and windowing system. A bug in a plug-in may cause other software to crash unless the plug-in is run in a sandbox. TrapperKeeper is potentially even more dangerous because it runs a complete copy of an application.

For this reason, TrapperKeeper provides strong isolation by running each parsing application in its own virtual machine. Virtualization prevents changes made within the virtual machine from being externalized to the host computer. Further, TrapperKeeper prevents buggy applications or malicious files designed to trigger bugs [12, 13]

from affecting the parsing of subsequent files by reverting the virtual machine to a clean checkpoint after parsing each file.

5.2 Implementation

As shown in Figure 5.2, TrapperKeeper has three components that enable type-aware file system features. Trapper captures the file parsing behavior of an application and stores it for later use as an application-specific virtual machine checkpoint. Keeper resumes the virtual machine from the checkpoint to apply the captured application's behavior to a new file, generating a file-specific virtual machine. Grabber provides routines that allow features such as metadata extraction and document preview to examine and use the application interface within the virtual machine. We next describe each component.

5.2.1 Trapper: Capturing application behavior

Trapper checkpoints an application just as it is about to execute its file parsing behavior but after it has completed its startup routines, displayed initial messages, shown open file dialogs, and so on. This checkpoint is later used by Keeper to parse file types associated with the application in the checkpoint.

Trapper is only executed once for each application. The checkpoint it generates is used by all features that employ Keeper and Grabber. To create the checkpoint, Trapper is given a virtual machine that has the application to be captured installed. This virtual machine encapsulates the application, its dependencies, interactions with other processes, disk and memory state, and output. The application runs in as natural a situation as possible while still isolating all of the application's output and side effects to prevent them from becoming visible to applications running on the host computer. The virtual machine also constrains input to prevent misbehaving applications from accessing confidential data.

Virtual machine encapsulation makes the checkpoint captured by Trapper portable, which allows remote servers or workstations with spare resources to perform the pars-

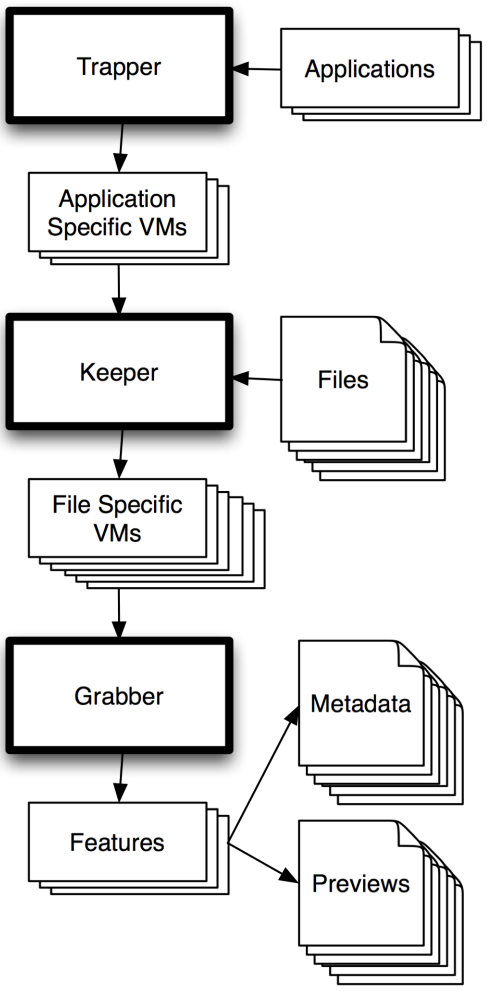


Figure 5.2: TrapperKeeper overview

ing on behalf of other computers. This is especially convenient in the context of a distributed file system, in which all computers share a common namespace and can thus collaborate to extract type-specific features from new files. For mobile, resource-constrained clients such as cell phones, offloading parsing and feature execution to servers is especially attractive.

Trapper runs on the host operating system. Our current implementation uses VMware Workstation version 6.0.2 to execute the guest operating system and the parsing application. Trapper creates a file system that is shared between the host and guest operating system for communication. Trapper also creates a shim file system in the guest OS that is used to detect when the application opens a file. The shim file system is implemented using FUSE [21]. It appears to contain a single file, which we refer to as the *dummy file*.

Trapper is given the name of the application and any arguments on its command line. It uses the VMware VIX API to start the execution of the named application within the virtual machine. The VIX API provided by VMware Workstation allows applications on the host to invoke virtual machine features such as snapshot creation, virtual machine suspension, and running applications in the guest.

If the application can be directed to open the dummy file with a command-line argument, no further intervention is needed. Otherwise, the user must open the file using the application's GUI.

The shim file system detects and blocks the `open` system call on the dummy file and, in response, creates a file in the communication directory shared between the guest and host operating systems. Trapper checkpoints the virtual machine using the VIX API when this file appears. It stores the resulting checkpoint of the application about to open the dummy file in a database of captured applications.

5.2.2 Keeper: Running application parsers

Keeper uses the application checkpoints produced by Trapper to run feature-specific code on individual files. Keeper induces the application to load a specific

file and continue running from the checkpoint. The resulting GUI will be parsed and manipulated by feature-specific code using the Grabber library. This is typically done whenever a new file enters the system or a file is modified.

Given a file to parse, Keeper must first determine which application checkpoint to use. By default, Keeper uses file extensions to determine the file type (e.g., files that end in ".mp3" are currently parsed using a checkpoint of the Exaile music player). While this default method adheres to the common practice of using file extensions to specify the type of each file, Keeper is not limited by this assumption. Because Keeper activity has no side-effects and the failure to parse a file can be detected, Keeper can try several parsers on a file, searching for one that parses the file correctly.

Keeper places the file of interest in the communication directory and uses the VIX API to resume execution from the checkpoint taken by Trapper. In the checkpoint, the application was captured in the middle of an `open` system call that was being blocked by the shim file system. Through the shared file system, Keeper signals the shim file system to return from `open`. The shim file system unblocks the application and returns normally from `open`. However, instead of returning a file descriptor for the dummy file the application was opening at the moment the checkpoint was taken, the shim file system instead returns a FUSE handle that directs future operations for that file to the shim file system. The shim file system allows applications to open a file for both reading and writing. It applies read-only operations to the file of interest rather than the dummy file. Write operations are temporarily buffered and returned to the application if it reads the same data that it wrote. However, no modifications are ever applied to the original file. This sleight-of-hand replaces the contents of the dummy file with those of the file of interest. Thus, when the application proceeds, it blithely parses and displays the file of interest.

5.2.3 Grabber: Capturing the interface

Grabber waits for the application to load the file and display its contents. Unfortunately, there is no standard method to detect when an application has finished

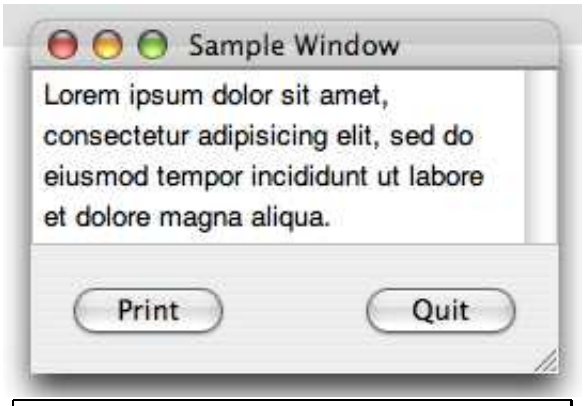
parsing the file and its interface has reached a stable state in which it displays the file contents.

One possible solution would be for Grabber to wait a fixed amount of time after the file is opened. This solution is undesirable because choosing the right timeout is hard. If a timeout is chosen that is too small, features may read incorrect values from the display. If one is chosen that is too large, much time would be spent idle and the rate at which TrapperKeeper can process new files would be limited. Further complicating matters, applications may take varying amounts of time to load files depending on the size of the file being loaded or the current load on the host CPU and disk.

Another solution we considered was for Grabber to wait until no changes have been made to the GUI for a fixed amount of time. However, this approach would fail for applications that do not have a stable final state due to activity such as animated GUI elements or automatic playback of a media file.

The solution we chose detects a final state by comparing the current state with the final state generated by the previous behavior of the same application. The first time Grabber uses an application checkpoint to parse a file, it waits a few minutes to be sure that the application GUI has reached a final state. Grabber then uses accessibility APIs to access a descriptive, hierarchical view of the elements of the GUI. Accessibility APIs are a part of every modern windowing system and allow applications to examine the GUI elements of other processes and their properties such as displayed text and size. Grabber writes a representation of each GUI element of the display to a file, with each GUI element on a single line. Figure 5.3 shows a simplified example of how a window and its child widgets appear when accessed through the accessibility API. The file generated by Grabber during the initial execution of the application checkpoint serves as an example of what the final state of the application GUI looks like after a successful parsing.

During subsequent invocations of the same application, Grabber periodically reads the state of the application GUI through the accessibility API. It then computes the difference between the current state and the example state. Our current implementa-




```
Window - ‘‘Sample Window’’
   Close Button
   Minimize Button
   Zoom Button
   Window Resizer
   Button - ‘‘Print’’
   Button - ‘‘Quit’’
   Scroll Area
     Scroll Bar
     Text Area - ‘‘Lorem Ipsum...’’
```

Figure 5.3: Accessibility API view of a window

tion calculates the difference by counting the number of lines generated by the `diff` tool when given the current and example GUI data. More sophisticated methods are possible, but we have found that the `diff` approach works well in practice.

Our method assumes that there should be a large difference between the current state of the application GUI and that in the example while the application is parsing the file. Once the file is loaded, some of the details of the GUI may be different from the example, but we expect much of the interface, such as the buttons, toolbars, and menu items, to be the same.

Grabber checks the state of the interface every 100 ms. If the fraction of lines that differ is above a threshold, Grabber does not consider the interface to be in a final state and it continues to wait. Grabber starts the threshold at 40% and gradually increases it by 0.2% every 100 ms to guarantee termination. When the parsing and display is complete, Grabber invokes feature-specific code that uses the current GUI state to implement features such as metadata extraction and document preview.

5.2.4 Discussion

Unlike the plug-in approach, TrapperKeeper does not need type-specific code to support a new file type. However, it does require support in the form of other software systems, most of which are readily available.

First, TrapperKeeper needs an application that can parse the file type in question. Ideally, the application should be able to open a file specified by a command-line argument. If this is not the case, a user must open a particular file using the application's GUI once. While this does represent human effort, the level of involvement is clearly much less than writing a plug-in. Further, the activity can be performed by any user of an application, not just by software developers.

TrapperKeeper also needs an instance of the file type to let Grabber observe a successful parsing that establishes a baseline for comparison with future parsings. The application must implement an accessibility interface to provide the windowing system access to information about the elements that make up the interface. Accessibility

APIs were originally designed to support tools that assist visually impaired users by exporting information exposed by application GUIs. They are a part of every popular modern windowing system.

There are several forces driving applications to implement the accessibility interface. First, the default GUI elements (buttons, windows, text fields, etc.) automatically implement accessibility functionality. Because many applications use these default widgets rather than implement their own, those applications support accessibility without any extra development effort. Implementing the accessibility interface is also desirable for its original purpose, allowing visually impaired people to use the application. Finally, implementing an accessibility solution has become a requirement to sell software to the United States government [76].

5.3 Features

Once Grabber determines that the parsing application within the virtual machine has successfully reached its final GUI state, it executes feature-specific code that uses the interface state to gather information about the file that has just been parsed. Grabber implements generic code that each feature can use to query and manipulate each application's GUI. Besides reducing feature complexity by providing a common mechanism, this implementation allows us to abstract away specific implementation details of window managers and operating systems that may run within the virtual machine.

We have implemented two features, metadata extraction and document preview, which are described in Sections 5.3.1 and 5.3.2. Additional features can be implemented by hooking into the Grabber API.

Note that feature extensibility does not recreate the original problem of requiring too many plug-ins. A developer only needs to write one feature; in contrast, she would have to write a plug-in for every application on each platform (because the plug-in interfaces are currently platform-dependent) to support the same feature.

5.3.1 Metadata extraction

The metadata extraction feature produces attribute-value pairs for each file. It stores these pairs in a file indexing system. Since the attributes extracted by the feature are themselves type-specific, the file indexing tool may be type-agnostic. For instance running the extraction feature on an MP3 file might produce the pairs {"composer","Beethoven"} and {"rating","five stars"}.

The metadata feature uses Grabber to acquire a dump of all GUI widgets such as text fields, choice boxes, tables, etc. The feature then sifts through the GUI information to find attribute-value pairs that describe the file.

We have implemented two ways to select which metadata to extract. The first way is completely automatic. The metadata feature searches through the GUI information to find widgets that supply both names and values. For instance, a photo viewer might have a label widget with an attribute called "name" that has the value "location" and an attribute called "text" that has the value "Paris". From this data, the metadata feature identifies {"location","Paris"} as an attribute value pair. Besides labels, the automatic metadata extractor looks for text and tables. Tables are especially valuable as they often have column or row headers that describe cell contents.

The above method can gather more data than is strictly necessary, so we implemented an alternative method for extracting metadata. The person who uses Trapper to create an application checkpoint subsequently runs Keeper on a sample file and selects the GUI widgets displaying metadata of interest by moving the mouse cursor over them and pressing a special key sequence (Ctrl-F11). The metadata feature makes a list of selected widgets. It extracts metadata from only those selected widgets in subsequent parsings. The selection is only done once per file type. It does not require any programming skills, just the ability to use the application.

In many applications, a button or menu item can be used to display more detailed information about a file after it is opened. To access this information, we have added an option that allows the user to specify buttons and/or menu items to be activated when parsing is complete, but before features are executed. The user chooses the

buttons or menu items by parsing a sample file and then pressing a special key sequence (Ctrl-F10) while the mouse is hovering over the element to be activated. These selections are saved for use on further parsings.

Once the metadata feature has extracted attribute-value pairs for a file, it stores them in an indexing database from which they can later be used as search terms when searching for particular files. Our implementation uses Beagle [8] to store metadata. With TrapperKeeper, Beagle's insertion and retrieval functionality is similar to that provided by Apple's Spotlight and Microsoft's Windows Desktop Search, except that no plug-ins are required to parse files and generate metadata.

However, one difference between plug-ins and TrapperKeeper is that plug-in-based metadata systems generally come with a set of metadata keys, standardized names for frequently used attributes and prescribed value formats. Because TrapperKeeper only extracts information displayed by applications, it does not know which displayed information might correspond to system-defined metadata keys and how to format the resulting values. For example, a music player may display a "play time" value for each song while the metadata key for the same value may refer to it as "length." This difference prevents TrapperKeeper from providing a complete replacement to plug-ins. Automatically creating ontologies to bridge similar gaps has proven to be difficult [9].

5.3.2 Document preview

The second feature we implemented is document preview. This feature creates an image of the file being displayed by its parsing application; the image can then be used as an icon for that particular file by a graphical file browser. Windows, Mac OS X, and Gnome all provide mechanisms to set a file's icon. These captured images can easily be used to provide custom icons for files that reflect the contents of that file when rendered.

The document preview feature uses Grabber to generate a screen shot of the application window after it has loaded a file. Grabber triggers the particular platform-

specific screen shot functionality for the guest operating system running in the virtual machine. For better screenshots, the user can again use Ctrl-F10 to specify GUI elements to be activated to perform actions such as switching software into presentation mode.

5.4 Evaluation

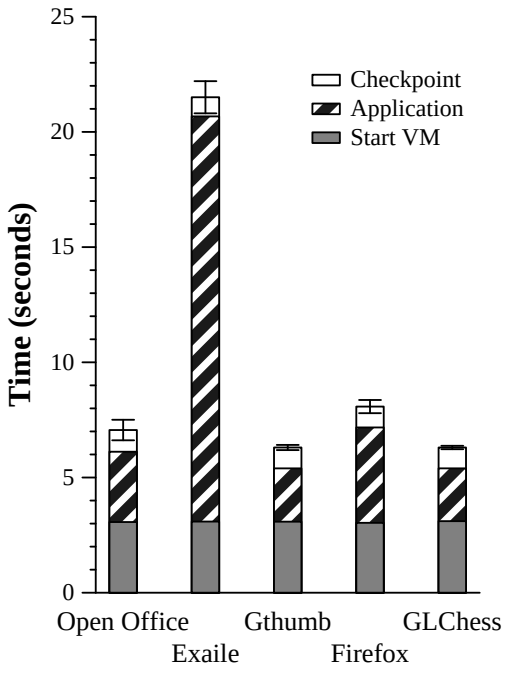
Our evaluation of TrapperKeeper sought to answer the following questions:

- How fast can TrapperKeeper extract metadata?
- How fast can TrapperKeeper generate previews?
- Can TrapperKeeper handle a variety of applications?
- What is the typical distribution of file types?

5.4.1 Methodology

In the following experiments, the computer we used was a Dell 690, with two quad-core 2.66 GHz Core 2 processors and 4 GB of RAM. The virtual machine we used was the 64-bit version of VMware workstation 6.0.2, and the guest OS was Ubuntu Linux 7.10.

We measured the time to extract metadata from and generate document previews of five files. The first is a 1 MB Microsoft Word file that uses several fonts and includes several images; the document is opened by Open Office Writer. The second is a 4.9 MB MP3 music file opened by the Exaile media player. Exaile cannot be directed to open a file through its command line arguments, so we used Trapper to checkpoint the application after specifying the file to be opened using Exaile's GUI. The remaining files are a 4.1 KB JPEG image, a 6.6 KB HTML file, and a 130 B saved chess game. These files are opened by the Gthumb image viewer, the Mozilla Firefox Web browser, and the GLChess chess program, respectively; all three applications can open a file through command line arguments.



This figure shows the amount of time needed by Trapper to capture checkpoints of five applications. Each result is the average of 5 trials. The error bars show 90% confidence intervals.

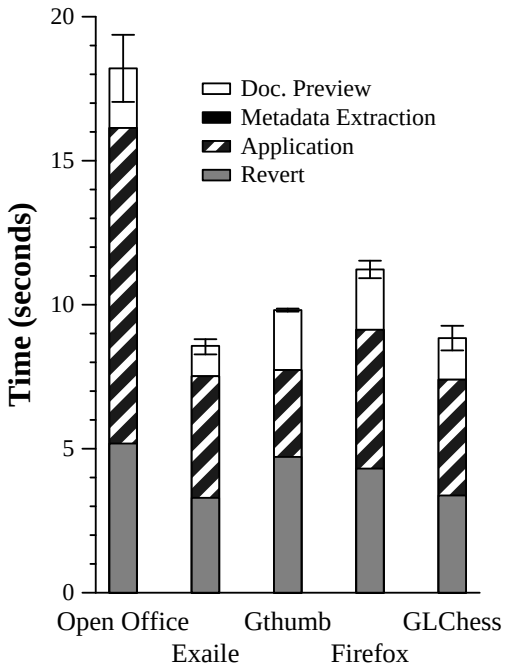
Figure 5.4: Trapper performance

5.4.2 Trapper performance

Figure 5.4 shows the time needed to create a checkpoint for each application. Creating a checkpoint for Exaile took the longest time because we had to manually open the MP3 file using Exaile's GUI. However, even with a manual step, it still took less than 22 seconds to create the checkpoint. Trapper created a checkpoint of all remaining applications in less than 9 seconds each.

We also measured the storage space consumed by checkpoints. The Ubuntu virtual machine took 4 GB. Each checkpoint consumed an additional 143-151 MB of storage. The checkpoints are relatively small because they are based on deltas from the virtual machine they were created from.

As the shadings within each bar in Figure 5.4 show, starting a new virtual machine took less than three seconds. Our initial checkpoint of Ubuntu Linux had all five applications installed. If any of our applications did not come with the base in-



This figure shows the amount of time needed by Keeper and Grabber to execute two features, metadata extraction and document preview, on five files parsed by different applications. Each result is the average of 5 trials. The error bars show 90% confidence intervals.

Figure 5.5: Keeper performance

stallation of the operating system, we would have had to install them before running Trapper. Checkpointing the virtual machine took 1-2 seconds, and the remaining time was spent waiting for the application to load and open the dummy file.

From these results, it is clear that creating a TrapperKeeper checkpoint is far less time-consuming than writing a type-specific plug-in.

5.4.3 Executing features

Figure 5.5 shows the time needed to resume from a virtual machine checkpoint and execute both features. As shown by the bottom segment of each bar, Keeper takes 3.3–5.2 seconds to resume the virtual machine from a Trapper checkpoint. Resuming from the Open Office Writer checkpoint takes slightly longer than the other applications, probably because Writer is more resource-intensive and uses more memory.

The second segment of each bar shows the amount of time that Grabber waits for

the application to reach a stable GUI state. Grabber waits an average of 5.4 seconds for each application. Again, Open Office Writer takes the longest, 10.9 seconds, because it must convert and display a complex document. In contrast, if we resume Writer with a simple text document, a stable GUI state is reached in only 4.2 seconds. The difference between these two times, 6.7 seconds, shows the benefit of detecting a stable GUI state rather than using a fixed timeout. An algorithm with a fixed timeout would either wait too long for simple documents or not correctly capture complex ones.

As shown by the third segment of each bar, the execution of the metadata feature is almost instantaneous for all applications. The reason is that Grabber must dump the state of each application's GUI to determine that the interface has reached a stable state. Since the metadata feature needs the identical information, Grabber can simply provide its cached values. Parsing the metadata to extract attribute-value pairs takes negligible time.

The top segment of each bar shows the amount of time for the document preview feature to capture a screen shot of each application displaying its files. Document preview takes an average of 1.7 seconds for all applications.

When the metadata extraction feature does not modify the state of the application (as is true for these 5 applications), TrapperKeeper executes the document preview feature immediately after the metadata extraction feature finishes. When this is not the case, TrapperKeeper must discard the virtual machine and again resume each application from its checkpoint prior to executing the document preview feature.

Extrapolating from these results, TrapperKeeper could extract metadata and make previews of 318 files per hour. TrapperKeeper takes the longest amount of time, 18.2 seconds, to process the Word document. At this rate, it could still process 198 documents per hour.

According to a file system study performed by Agrawal et al. [2], in 2004, the last year of their study of corporate desktops, the average file system contained approximately 90,000 files, 22% of which had been modified or created locally within the last year, a rate of only 2.26 per hour. The actual rate of file creation is higher

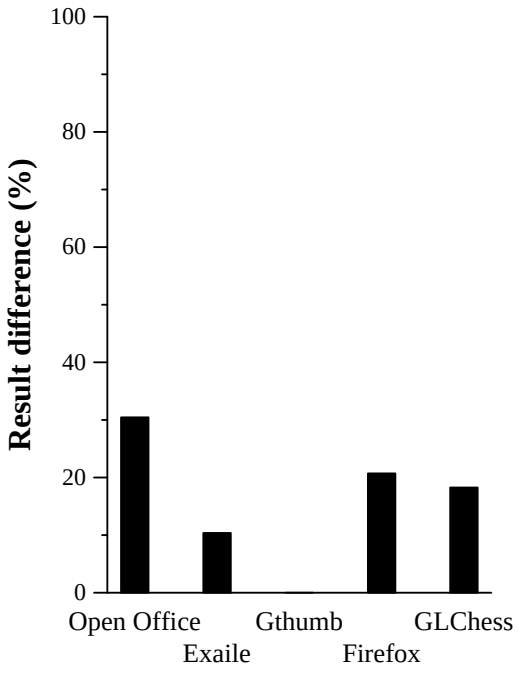


Figure 5.6: Difference in GUI states for different files

than this figure because the trace data does not capture files that are created and subsequently deleted between file system snapshots. Nevertheless, the more than two orders of magnitude difference between the long-term file creation rate and TrapperKeeper's parsing rate gives us confidence that TrapperKeeper can keep up with the file creation rate of the average user. Further, all TrapperKeeper activity takes place asynchronously in the background, so it never blocks user file creation or modification.

Further, the increasing popularity of multicore computers on the desktop creates opportunities for parallelism. First, TrapperKeeper can parse new files on an idle core in the background while foreground applications use other cores. Additionally, because each TrapperKeeper application is encapsulated in a separate virtual machine, TrapperKeeper can be executed in parallel when more than one idle core is available. Since we had a multicore computer available to us, we verified this possibility by creating a multithreaded version of TrapperKeeper. We verified that two instances of TrapperKeeper could run in parallel and produce almost perfect speedup on 2 cores (e.g., approximately 395 documents per hour or 836 music files per hour). Beyond

two cores, our implementation was limited by thread safety issues with the version of the VMware VIX API we used.

Further parallelism can be achieved in the context of a distributed file system. Keeper can run on any client of the distributed file system, leveraging the spare resources of multiple computers. This is made easy by the portability of virtual machines and distributed notification mechanisms such as persistent queries [55].

Performance can also be improved by hybridizing the TrapperKeeper approach with a plug-in based approach. By using plug-ins for a few of the most popular file types, such as MP3 and JPEG, we can make the these common cases fast for a small additional effort.

5.4.4 Detecting stable GUI states

We next verified that applications reach a stable GUI state with a relatively small number of differences from the initial example parsing (described in Section 5.2.3) when they open a different file of the same type.

To test this, we opened each application with a file of similar size but different contents than the one used by Grabber to create the example parsing. Rather than use Grabber's algorithm for detecting the final state, we let each application run and manually determined when the final state had been reached. Figure 5.6 shows the results. There is almost no difference in the final GUI state of Gthumb when it displays two different images because the image data is not part of the GUI state exported by the application, which makes sense since the original purpose of the accessibility APIs is to help visually impaired users. The GUI state of the remaining applications differ by 10.4–30.4%.

5.4.5 Experiences with TrapperKeeper

To demonstrate the general applicability of TrapperKeeper, we next captured the type-specific behavior of every application listed in the applications menu on a fresh installation of Ubuntu 7.10. We restricted our experiment to the 20 listed applications

that open user-specified files (not just configuration files).

All 20 applications were able to work with TrapperKeeper. However, not every application behaved as we expected. We discovered that choosing a minimal set of devices for the initial virtual machine is a bad idea. Two applications, Gnome Sound Recorder and Serpentine Audio CD Creator required audio devices and a CD writer, respectively. We corrected this problem by having the initial virtual machine come with a robust set of virtual devices.

For some applications, the actions required to execute the `open` system call are not obvious. For instance, Gnome Sound Recorder only records the name of files selected using its open file dialog. It does not execute the `open` system call until the play button is pressed. We handled such applications by experimenting with the GUI until an `open` system call was trapped by the shim file system.

Some applications were picky about file locations. Pidgin, an instant messenger client, only opens saved chat logs in a particular location in the file system, so we occupied that space with a symlink pointing to Trapper's dummy file. Fiddling was sometimes necessary to get an application's GUI to display file contents. For example, Evolution, an email client, requires several buttons to be pressed after opening a contact file before displaying its contents. We encountered one application, Exaile, that derives the type of a file from its filename extension. Since our dummy file can have only one name per application instance, we simply created snapshots with different dummy file names for each file type.

Despite these unexpected behaviors, we found it quite easy to capture type-specific behavior with TrapperKeeper. In fact, Applying TrapperKeeper to all of these applications took a single person less than eight hours. Further, because many applications parse more than one file type, the 20 applications we handle allow us to support metadata extraction and document preview for over 100 distinct file types.

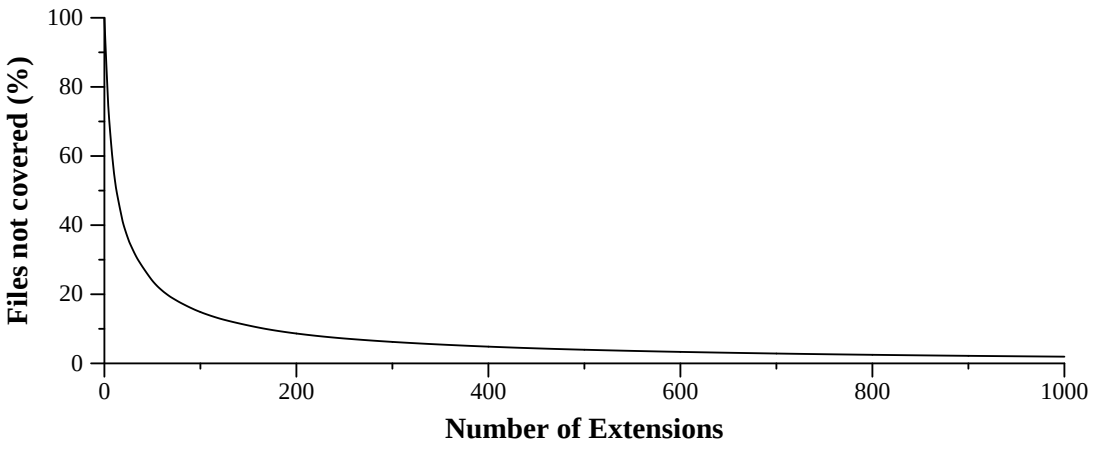


Figure 5.7: Fraction of files left uncovered

5.4.6 The long tail

One of our motivations for developing TrapperKeeper is that there are a large number of file types in use. Our hypothesis is that would take a Herculean effort to get near total coverage of file types using a plug-in approach. To get an idea of how many file types are in common usage, we examined the file system snapshots from the most recent year (2004) of the file system study performed by Agrawal et al. [2]. Their study captured file system snapshots on 8,729 desktop computers at Microsoft.

We used the methodology of Agrawal et al. to count the number of unique file name extensions and the percentage of files that appear for each extension. Their methodology defines a file name to have an extension if there are five or fewer characters following the last period in the file name. The endings of compressed files ending in .gz, .bz2, or .Z are removed before determining the file name extension. Using this methodology, approximately 7% of the file names captured by the study have no extension. In total, we found 102,869 distinct filename extensions in the study data. Although filename extensions and file types are not necessarily in one-to-one correspondence, this evidence certainly suggests that there are a large number of file types.

Figure 5.7 shows that the distribution of file name extensions has a very long tail. If one were to write type-specific code for each extension, even writing 50 plug-ins

Number of Most Popular Extensions	Files Covered
10	44.94%
20	59.94%
50	76.08%
100	85.14%
1000	98.05%

Figure 5.8: Coverage of most popular file name extensions

would not cover 23.92% of the files in the study. Table 5.8 shows the maximum number of files that would be covered by writing type-specific code for different numbers of extensions. These figures are roughly consistent with a 1999 file system study of 10,568 file systems in a corporate setting [16] which found that the top thirty file name extensions represented approximately 70% of files.

Because there are so many file types in the long tail, a strategy that involves substantial developer effort per file type will not achieve coverage. The cost of writing and maintaining code is high. Thus, it does not make economic sense to invest developer effort in supporting a file type whose percentage of the overall number of files is small. However, as our results show, although many individual file types are relatively unpopular, the collection of such unpopular types represents a substantial portion of the total number of files.

The solution that we propose is to reduce the effort needed to support each file type. TrapperKeeper requires no programming per file type. For other types, ordinary users can create snapshots using Trapper and an application's GUI. By greatly reducing the work required to support new file types, TrapperKeeper aims to increase the number of file types for which the benefit outweighs the cost of adding support.

5.5 Conclusion

Understanding type-specific metadata encapsulated within files has a great many benefits. In the past, unlocking these benefits has required that software developers build and maintain type-specific plug-ins for a wide variety of features, tools, and operating systems. Since the cost of developing such plug-ins is high, it is hard to deploy innovative new features that exploit type-awareness. Even for the most popular features, many files on a computer will be unsupported because the distribution of file types has a long tail and it is only possible to support the most popular file types.

TrapperKeeper changes the economics of this equation by making it much easier to create type-aware components. Our results show that checkpoints of new file types can be created in less than a minute using TrapperKeeper. Further, any user of an application can create a Trapper checkpoint since no programming is required. Our results also show that TrapperKeeper can process hundreds of files per hour, a rate that far exceeds the amount of files created or modified by a typical user.

CHAPTER 6

Consumer Electronics Device Integration

At first glance, it appears that closed-platform CE devices cannot participate in a distributed file system because they lack the ability to execute distributed file system code. For instance, most DVRs and MP3 players require substantial hacking to modify them to run arbitrary executables. Even CE devices 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 CE device 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 CE device for all EnsemBlue activities. If the user modifies data in the local CE device namespace, Wolverine detects these changes and makes corresponding updates to the distributed EnsemBlue namespace. Similarly, when data is modified in the EnsemBlue namespace, Wolverine propagates relevant modifications to the CE device.

The requirements for a closed-platform CE device to participate in EnsemBlue are minimal. The CE device must support communication with a general-purpose computer; e.g., via a wireless interface or USB cable. The CE device must provide a method to list the files it stores, as well as methods to read and update each file. Currently, EnsemBlue supports CE devices that provide a file system interface such as FAT32.

6.1 Making CE devices self-describing

In contrast to current models that require CE devices to synchronize with particular computers, EnsemBlue allows CE devices 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 CE devices self-describing. Each CE device locally stores metadata files that contain all the information needed for an EnsemBlue client to attach and interact with that CE device. For each file system object on the CE device that is replicated in the EnsemBlue namespace, EnsemBlue stores a *receipt* on the CE device that describes how the state of the object on the CE device relates to the state of a corresponding object in the EnsemBlue namespace. EnsemBlue also stores device-level metadata on the CE device that uniquely identify the CE device 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 CE device attaches.

6.2 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 CE device that mandates a custom organization, EnsemBlue stores data in a local namespace that matches the mandated organization.

EnsemBlue views files stored on a CE device as replicas of files in its distributed namespace. Each receipt maintains a one-to-one correspondence between the file in the CE device 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 CE device if it did not have all relevant links cached when the CE device attached. The alternative of using symbolic links in the CE device's local file system is unattractive because many CE device file systems do not support links. Receipts avoid both pitfalls since they are file system independent and reside on the CE device.

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 7.2.1. When a CE device 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.

6.3 Reintegrating changes from CE devices

On a general-purpose EnsemBlue client, a kernel module intercepts file modifications and redirects them to Wolverine. However, most CE devices do not allow the insertion of kernel modules, making this method infeasible. Thus, for a closed-platform CE device, Wolverine uses a strategy similar to the one used by file synchronization tools such as rsync [75] and Unison [56] in which it scans the local file system of the CE device to detect modifications. Wolverine scans a CE device when it is first attached and subsequently at an optional device-specific interval.

When a CE device attaches to a general-purpose client, Wolverine lists all files on the CE device using the interface exported by that device. For instance, for CE devices 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 CE device has a modification time later than the time stored in its receipt, the file has been modified since the last time the CE device detached from an EnsemBlue client. Wolverine copies the file from the CE device to the EnsemBlue namespace and updates its receipt.

If a file or directory is found on the CE device 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 CE device. 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 CE device with an EnsemBlue directory the first time the CE device is attached. Wolverine replicates the local file system of the CE device 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 CE device.

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 CE device, 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 CE devices such as DVRs and cameras on which files are often deleted due to storage constraints.

6.4 Propagating updates to CE devices

Modifications made to files in EnsemBlue are automatically propagated to corresponding files in local CE device namespaces. When Wolverine creates a receipt for an object stored on a CE device, it also sets a callback with the server for that file on behalf of the CE device. If the file is subsequently modified by another client, the server sends an invalidation to the client to which the CE device 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 CE device. If the CE device is not attached to a client when a file is modified, the server queues the invalidation and delivers it when the CE device next attaches.

EnsemBlue uses affinity to determine whether the local namespace of a CE device 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 CE device namespace. At that time, Wolverine replicates the subtree in the specified directory on the CE device. When setting affinity, the user may optionally specify that files created in the future in the EnsemBlue subtree should also be replicated on the CE device.

CE devices support type-specific affinity. A command line tool lets the user create a persistent query as a hidden file in any directory on the CE device. 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 CE device for the new query, the client to which the CE device 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 CE device directory. In this manner, the CE device directory is populated with all files that match the query. This use of type-specific affinity is inspired by the Semantic File System [22]. However, in contrast to SFS where directories populated by semantic queries are virtual, EnsemBlue replicates data on the CE device so that the results of a query are available when the CE device is disconnected.

6.5 Evaluation

In the next two sections, we present case studies in which we examine how well CE devices can be integrated with EnsemBlue.

6.5.1 Case study: Integrating a digital camera

In our first case study, 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 organization.

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 CE device manufacturers, third-party software developers, and technically-savvy users.

We measured the time to import photos from our camera into EnsemBlue. The camera, containing 201 new photos approximately 256 MB in size, is attached to an EnsemBlue client, a X40 laptop with a 1.2 GHz Pentium M processor and 768 MB of RAM. 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.

6.5.2 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 GnuPod. 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 3.4, 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.

These examples show that this system of CE device integration can effectively integrate real devices, incorporating their storage into the EnsemBlue distributed file system.

CHAPTER 7

Ensembles

The final work in this dissertation is ensembles. Ensembles are collections of devices that share a common view of the file system over a local network. Ensembles improve data propagation by allowing clients disconnected from the central file server to share their cache contents and propagate updates to each other. This is particularly important for CE devices that frequently travel with users. 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) and a client joins only one ensemble at a time.

7.1 Design considerations

In designing support for ensembles, we wished to reach a middle ground between file systems such as BlueFS [52] and Coda [34] that support disconnected operation but do not let two disconnected clients communicate, and systems such as Bayou [73] 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 CE devices 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 [53] and Segank [67]. 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 CE device 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).

7.2 Implementation

An ensemble is formed when two or more disconnected clients share a local area network. For wireless devices, we leverage PAN-on-Demand [3], 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. CE devices can join an ensemble by attaching to a general-purpose ensemble member.

7.2.1 Tracking modifications

EnsemBlue tracks modifications for each object using a version vector [54] 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.

7.2.2 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.

7.2.3 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 [52]. 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 7.2.1. 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.

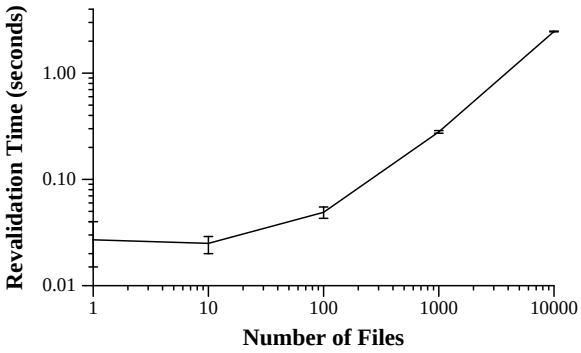
Any ensemble client that modifies an object sends an RPC to the castellan. The castellan forwards the modification to all ensemble members that cache the object. These clients update their cached objects and append a record to their disconnection logs. Thus, most clients are only informed of updates 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.

7.2.4 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.



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 7.1: Ensemble formation time

7.3 Evaluation

This evaluation measures the overhead of forming an ensemble.

7.3.1 Ensemble formation

We measured the time two disconnected clients take to form an ensemble. During this experiment, an IBM X40 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 7.1 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.

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 7.2: Time to reconcile updates in an ensemble

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 7.2. 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.

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.

7.4 Conclusion

Ensembles are an effective mode of communication that can provide improved data availability and consistency. They are essentially a stronger form of disconnected operation that is well suited to situations in which the central file server cannot be reached or has poor connectivity to the client.

CHAPTER 8

Related Work

AFS -¿ Coda -¿ BlueFS

Consumer data management has become the subject much research interest recently. The intersection of distributed consumer storage and metadata has been investigated by Perspective [61]. In Perspective, the contents of storage devices are described by metadata criteria (e.g. file type = MP3 and artist = Radiohead) called *views*. These views form a publish/subscribe system that can make ensemble formation more efficient. The Cimbiosys [59] system provides *filters*, which are similar to views, and are used to improve pairwise synchronization between storage devices.

8.1 Persistent queries related work

Persistent queries are the first system to deliver notifications to applications by reusing the distributed file system server cache coherence messages. Many operating systems provide application notifications about changes to local file systems; Linux's inotify [44], the Windows Change Journal [15], and Mac OS X FSEvents [20] are three examples. Watchdogs [11], 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.

The evaluation of 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 [60] and Diamond [30]. Salmon et al. [62] have 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 [22]. However, the directory contents produced by type-specific affinity are persistent, meaning that they can be accessed by a mobile computer or CE device that is disconnected from the file server. Persistent queries should not be confused with applications such as Glimpse [46] and Connections [68] 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.

8.2 Sprockets related work

To the best of our knowledge, sprockets are the first system to use binary instrumentation and a transactional model to allow arbitrary code to run safely inside a distributed file system's address space.

Our use of binary instrumentation to isolate faults builds on the work done by Wahbe et al. [79] to sandbox an extension inside a program's address space. However, instead of limiting access to the address space outside the sandbox, we provide unfettered access to the address space but record modifications and undo them when the extension completes execution.

VINO [65] used software fault isolation in the context of operating system extensions. However, VINO extensions do not have access to the full repertoire of kernel functionality and are prevented from accessing certain data. Permitted kernel functions are specified by a list. Those functions must check parameters sent to them by the extension to protect the kernel. Nooks [71] used this technique for device drivers.

Similarly, XFI [85] isolated loaded code at either kernel or user level, providing plug-ins with access to specified memory regions and allowed function entry points. In contrast, sprockets can call any function and access any data.

The Exokernel [18] allowed user-level code to implement many services traditionally provided by the operating system. This is typical of the microkernel approach embodied by systems such as Minix [72] and L4 [41]. Rather than focus on kernel extensions, sprockets target functionality that is already implemented at user-level. This has several advantages, including the ability to access user-level tools and libraries. The sprocket model also introduces minimal changes to the system being extended because it requires little refactoring of core file system code and makes extensions appear as much like a procedure call as possible.

Type-safe languages are another approach to protection. The SPIN project [71] used the type-safe Modula-3 language to guarantee safe behavior of modules loaded into the operating system. However, this approach may require extra effort from the extension developer to express the desired functionality and limits the reuse of existing file system code, much of which is not currently implemented in type-safe languages. Languages can be taken even further [40] by allowing provable correctness in limited domains. However, this is not applicable to our style of extension which can perform arbitrary calculation.

Other methods for extending file system functionality such as Watchdogs [11] and stackable file systems [27, 33] provide safety, but operate through a more restrictive interface that allows extensions only at certain pre-defined VFS operations such as `open`, `close`, and `write`. The sprocket interface is not necessarily appropriate for such course-grained extensions; instead, we target fine-grained extensions that are a few hundred lines of source code at most.

The use of sprockets to evaluate file system state was inspired by the predicates used by Joshi et al. [31] to detect the exploitation of vulnerabilities through virtual machine introspection. Evaluating a predicate provides similar functionality to a transaction that is never committed. The evaluation of a sprocket has similar goals in that it extracts a result from the system without perturbing the system's address

space. However, since the code we are isolating runs only at user level, we can provide the needed isolation by using existing operating system primitives instead of a virtual machine monitor.

Like projects on software [66] and hardware [28] transactional memory, sprockets rely on hiding changes to memory from other threads to ensure that all threads view consistent state. One transactional memory implementation, LogTM [49], also uses a log to record changes to memory state. In the future, it may be possible to improve the performance of sprockets, particularly on multicore systems, by leveraging these techniques.

8.3 TrapperKeeper related work

As far as we know, TrapperKeeper is the first project to leverage existing applications in order to extract metadata from files. This is in contrast to the technique first proposed by the Semantic File System [22], which uses small, special-purpose programs to extract metadata from files. The Semantic File System approach is used by today's popular metadata indexing and retrieval systems including Spotlight [69], Windows Desktop Search [82], Beagle [8], Google Desktop [24], and X1 [83]. For these commercial products, the special-purpose program is often a plug-in. The plug-in approach has also been used in academic projects including Roma [70] and Stuff I've Seen [17], which is the basis for Windows Desktop Search. More recently, document preview techniques [7] based on the same principles have emerged. Application developers provide plug-ins that are invoked to render a preview of the document.

TrapperKeeper makes use of accessibility APIs to get information from an application's GUI. These interfaces were designed to expose the structure and content of GUI elements to screen reader software, such as JAWS [19], VoiceOver [5], Narrator [1], and Gnopernicus [23], that help visually impaired users operate a computer. TrapperKeeper uses accessibility APIs for a different purpose — easy access to structured GUI information to extract information about displayed files.

DejaView [36] previously used accessibility APIs to archive and search the text

displayed by applications. Thus, DejaView's purpose is similar to that of TrapperKeeper's metadata extraction feature. Both have different strengths. DejaView indexes more than just file system data. However, TrapperKeeper's metadata extraction feature indexes data that the user has yet to view. It also can manipulate application GUIs through recorded actions similar to GUI scripting [42] to extract more information.

More generally, interposing on calls between the application and windowing system has been used as a way to access and even modify [63] the user-visible aspects of applications.

These are not the only ways to access information displayed to users. Screen scrapers have long been used to access information that programs display, but do not expose through other means. However, they are generally undesirable as they require substantial custom code to extract the desired information and can easily be broken by changes in the application or its configuration.

TrapperKeeper also relies on a checkpoint and restart mechanism to capture application activity at the moment it begins to parse a file. This task is potentially difficult because we need to capture not only the application itself, but also its interactions with the windowing system and any processes or other entities with which the process communicates. Encapsulating the application in a virtual machine isolates the actions of the captured application from other applications and provides a convenient snapshot ability that captures the application at the moment a file is opened. Capturing an application in this way is similar to making a virtual machine appliance [78] whose only function is to parse files.

We chose to use virtual machine checkpoint and restart for TrapperKeeper. Potentially, we could have used a different checkpointing solution implemented in the operating system or at user level [37, 43, 57, 74]. Such a solution might provide better performance, but the implementation would be more challenging. For instance, special care would be needed to correctly handle the stateful interactions between the captured process and the window manager.

8.4 CE device integration 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.

EnsemBlue's strategy of scanning the local file system of closed-platform CE devices is similar to the way that file synchronizers operate [56, 75]. 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. Receipts can be combined with persistent queries to automate the remapping of individual files between namespaces.

8.5 Ensembles related work

EnsemBlue's model of supporting isolated disconnected clients owes much to Coda [34]. 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 [64], 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 [73] 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 [53] 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 CE devices with limited storage. In contrast, EnsemBlue clients can discard update records once they have been reconciled with its server. Footloose targets CE devices 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 [67], in which a disk with a wireless network interface carried by a user at all times ensures consistency among computers that share a namespace, Ficus [26], Files Every Where [58], and OmniStore [32].

CHAPTER 9

Conclusion

Consumer electronics devices and multimedia data are challenges to consumer data management. This dissertation has addressed these issues by incorporating CE devices into a distributed storage system and developing techniques for gathering and using file type information.

Consumer electronics devices are popular computing platforms, serving as intermediaries in many computing activities. Yet, it is challenging to integrate them into existing distributed systems. CE device 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 CE devices should be flexible. It must customize its interactions with each device according to the interface presented. Even if a CE device 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 benefits of this flexibility. By supporting namespace diversity, device ensembles, and persistent queries, EnsemBlue is able to incorporate devices such as MP3 players, cameras, and media players full-fledged participants.

Sprockets furthers our ability to build services on top of the distributed file system by providing a safe, fast, and easy-to-use method for extending the functionality of file systems implemented at user level. With sprockets, we could now build type-aware

persistent queries and implement type-specific conflict resolution. We were able to implement every sprocket that we attempted in a few hundred lines of code. Our sprocket implementation using binary instrumentation caught several serious bugs that we introduced into extensions and allowed the file system to recover gracefully from programming errors.

Finally, TrapperKeeper eases the extraction of type-specific metadata encapsulated within files for use by applications. In the past, unlocking these benefits has required that software developers build and maintain type-specific plug-ins for a wide variety of features, tools, and operating systems. Since the cost of developing such plug-ins is high, it is hard to deploy innovative new features that exploit type-awareness. Even for the most popular features, many files on a computer will be unsupported because the distribution of file types has a long tail and it is only possible to support the most popular file types.

TrapperKeeper changes the economics of this equation by making it much easier to create type-aware components. Our results show that checkpoints of new file types can be created in less than a minute using TrapperKeeper. Further, any user of an application can create a Trapper checkpoint since no programming skills are required.

In addition to these demonstrations of new concepts and techniques, this work has contributed to a file system that is both practical to use and serves as a foundation for ongoing work in consumer storage. One area we are investigating is techniques for incorporating consumer online storage services such as Flickr or Facebook into the distributed file system. This is the next step in the process of expanding the file system to cover all of a user's data.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Accessibility in Windows Vista. http://www.microsoft.com/enable/products/windowsvista/.

[2] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. A five-year study of file-system metadata. In *FAST'07: Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), USENIX Association, pp. 31–45.

[3] 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.

[4] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 95–109.

[5] Apple - Mac OSX - VoiceOver. http://www.apple.com/macosx/features/voiceover/.

[6] Apple. "iTunes Music Store Customer Service - Authorizing your computer". http://www.apple.com/support/itunes/musicstore/authorization/.

[7] Quick look programming guide. http://developer.apple.com/documentation/UserExperience/Conceptual/Quicklook_Programming_Guide/Quicklook_Programming_Guide.pdf.

[8] Main page - Beagle. http://beagle-project.org.

[9] Berners-Lee, T., Hender, J., and Lassila, O. The semantic web. *Scientific American 284* (May 2001), 34–43.

[10] Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C., and Eggers, S. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995), pp. 267–284.

[11] Bershad, B. B., and Pinkerton, C. B. Watchdogs - extending the UNIX file system. *Computer Systems 1*, 2 (Spring 1988).

[12] Cadar, C., and Engler, D. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software* (August 2005).

[13] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (2006).

[14] Chang, F., and Gibson, G. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 1–14.

[15] Cooperstein, J., and Richter, J. Keeping an eye on your NTFS drives: the Windows 2000 Change Journal explained. *Microsft Systems Journal* (September 1999).

[16] Douceur, J. R., and Bolosky, W. J. A large-scale study of file-system contents. In *Proceedings of the international conference on measurement and modeling of comuter systems (SIGMETRICS)* (New York, NY, 1999), ACM, pp. 59–70.

[17] Dumais, S. T., E. Cutrell, E., Cadiz, J. J., Jancke, G., Sarin, R., and Robbins, D. C. Stuff i've seen: A system for personal information retrieval and re-use. In *Proceedings of SIGIR 2003* (Toronto, Canada, 2003).

[18] Engler, D., Kaashoek, M., and J. O'Toole, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 251–266.

[19] JAWS for Windows. `http://www.freedomscientific.com/`.

[20] FSEvents programming guide. `http://developer.apple.com/documentation/MacOSX/Conceptual/OSX\_Technology\_Overview/SystemTechnology/chapter\_3\_section\_5.html`.

[21] Filesystem in userspace. `http://fuse.sourceforge.net/`.

[22] 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.

[23] The gnome accessibility project. `http://developer.gnome.org/projects/gap/`.

[24] Google desktop. `http://desktop.google.com`.

[25] gphoto - gphoto home. http://www.gphoto.org.

[26] 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.

[27] Heidemann, J. S., and Popek, G. J. File-system development with stackable layers. *ACM Transactions on Computer Systems (TOCS) 12*, 1 (1994), 58–89.

[28] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), pp. 289–300.

[29] 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).

[30] 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).

[31] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 91–104.

[32] 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.

[33] Khalidi, Y. A., and Nelson, M. N. Extensible file systems in spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 1–14.

[34] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[35] Kumar, P., and Satyanarayanan, M. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Winter Technical Conference* (New Orleans, LA, January 1995).

[36] Laadan, O., Baratto, R., Phung, D., Potter, S., and Nieh, J. DejaView: A personal virtual computer recorder. In *Proceedings of the Twenty-first ACM Symposium on Operating Systems Principles* (Stevenson, WA, Oct 2007), pp. 279–292.

[37] Laadan, O., and Nieh, J. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *Proceedings of the 2007 USENIX Annual Technical Conference* (Santa Clara, CA, June 2007).

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

[39] Leach, P., and Perry, D. CIFS: A Common Internet File System. In *Microsoft Interactive Developer* (November 1996).

[40] Lerner, S., Millstein, T., Rice, E., and Chambers, C. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2005), ACM Press, pp. 364–377.

[41] Liedtke, J. On u-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 237–250.

[42] Little, G., Lau, T. A., Cypher, A., Lin, J., Haber, E. M., and Kandogan, E. Koala: capture, share, automate, personalize business processes on the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2007), ACM, pp. 943–946.

[43] Litzkow, M., and Solomon, M. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter Conference* (January 1992).

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

[45] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[46] 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.

[47] Mazières, D. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference* (Boston, MA, June 2001), pp. 261–274.

[48] Microsoft. Share Audio Files Zune to Zune. `http://www.zune.net/en-us/support/howto/zunetozune/sharesongs.htm`.

[49] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. Logtm: Log-based transactional memory. In *HPCA-12* (February 2006).

[50] Network Working Group. *NFS: Network File System protocol specification*, March 1989. RFC 1094.

[51] Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[52] 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.

[53] 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).

[54] 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.

[55] Peek, D., and Flinn, J. EnsemBlue: Integrating consumer electronics and distributed storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[56] 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.

[57] Plank, J. S., Beck, M., Kingsley, G., and Li, K. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference* (January 1995), pp. 213–223.

[58] 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.

[59] Ramasubramanian, V., Rodeheffer, T., Terry, D. B., Walraed-Sullivan, M., Wobber, T., Marshall, C., and Vahdat, A. Cimbiosys: A platform for content-based partial replication. Tech. Rep. MSR-TR-2008-116, Microsoft Research, Mountain View, CA, 2008.

[60] Riedel, E., Faloutsos, C., Gibson, G. A., and Nagle, D. Active disks for large-scale data processing. *IEEE Computer* (June 2001), 68–74.

[61] Salmon, B., Schlosser, S. W., Cranor, L. F., and Ganger, G. R. Perspective: Semantic data management for the home. Tech. Rep. CMU-PDL-08-105.

[62] 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.

[63] Satyanarayanan, M., Flinn, J., and Walker, K. R. Visual proxy: Exploiting OS customizations without application source code. *ACM SIGOPS Operating Systems Review 33*, 3 (1999), 14–18.

[64] Schilit, B. N., and Sengupta, U. Device ensembles. *Computer 37*, 12 (December 2004), 56–64.

[65] Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, Washington, October 1996), pp. 213–227.

[66] Shavit, N., and Touitou, D. Software transactional memory. In *Symposium on Principles of Distributed Computing* (1995), pp. 204–213.

[67] 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).

[68] 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.

[69] Spotlight overview, May 2007. `http://developer.apple.com/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.pdf`.

[70] Swierk, E., Kiciman, E., Laviano, V., and Baker, M. The Roma personal metadata service. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, 2000).

[71] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, 2003), pp. 207–222.

[72] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems Design and Implementation*. Prentice Hall, 2006.

[73] 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.

[74] Theimer, M. M., Lantz, K. A., and Cheriton, D. R. Preemptable remote execution facilities for the V-System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Santa Clara, CA, December 1985).

[75] 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.

[76] Section 508 of the rehabilitation act, as amended by the workforce investment act of 1998.

[77] Veeraraghavan, K., Nightingale, E. B., Flinn, J., and Noble, B. qufiles: A unifying abstraction for mobile data management. In *The Ninth Workshop on Mobile Computing Systems and Applications* (February 2008).

[78] Virtual appliance marketplace. `http://www.vmware.com/appliances/`.

[79] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 203–216.

[80] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. The locus distributed operating system. *SIGOPS Oper. Syst. Rev. 17*, 5 (1983), 49–70.

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

[82] Windows desktop home page. `http://www.microsoft.com/windows/desktopsearch`.

[83] X1 - unified, actionable search. `http://www.x1.com`.

[84] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 273–288.

[85] lfar Erlingsson, Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, November 2006), pp. 75–88.