Device-Transparent Personal Storage

by

Jacob Alo Strauss

S.B., Massachusetts Institute of Technology (2001)
M.Eng., Massachusetts Institute of Technology (2002)

Submitted to the Department of Electrical Engineering and
Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

Author . . . .
Department of Electrical Engineering and Computer Science
September 3, 2010

Certified by . . . .
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Robert T. Morris
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Professor Terry P. Orlando
Chair, Department Committee on Graduate Students

Device-Transparent Personal Storage

by

Jacob Alo Strauss

Submitted to the Department of Electrical Engineering and Computer Science
on September 3, 2010, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Users increasingly store data collections such as digital photographs on multiple personal devices, each of which typically presents the user with a storage management interface isolated from the contents of all other devices. The result is that collections easily become disorganized and drift out of sync.

This thesis presents *Eyo*, a novel personal storage system that provides *device transparency*: a user can think in terms of "file X", rather than "file X on device Y", and will see the same set of files on all personal devices. *Eyo* allows a user to view and manage the entire collection of objects from any of their devices, even from disconnected devices and devices with too little storage to hold all the object content.

Eyo separates metadata (application-specific attributes of objects) from the content of objects, allowing even storage-limited devices to store all metadata and thus provide device transparency. Fully replicated metadata allows any set of *Eyo* devices to efficiently synchronize updates. Applications can specify flexible placement rules to guide *Eyo*'s partial replication of object contents across devices. *Eyo*'s application interface provides first-class access to object version history. If multiple disconnected devices update an object concurrently, *Eyo* preserves each resulting divergent version of that object. Applications can then examine the history and either coalesce the conflicting versions without user direction, or incorporate these versions naturally into their existing user interfaces.

Experiments using *Eyo* for storage in several example applications—media players, a photo editor, podcast manager, and an email interface—show that device transparency can be had with minor application changes, and within the storage and bandwidth capabilities of typical portable devices.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Robert T. Morris
Title: Professor of Computer Science and Engineering

Prior Publication

This thesis includes material from an earlier workshop paper presented the case for a device transparent storage system and argued for global metadata distribution as one enabling mechanism toward that goal [55].

Acknowledgments

I am deeply indebted to my advisors, Frans Kaashoek and Robert Morris, for years of guidance, advice, and encouragement. I can't imagine a better group of mentors. I am continually amazed that they have managed to bring together such a wonderful group of people.

I did not work alone on this thesis alone, or on the earlier projects that are mentioned here. Chris Lesniewski-Laas has been, at various times over many years, a lab partner, an officemate, a housemate, a coauthor, an occasional co-conspirator, but most importantly a friend. Bryan Ford provided the vision and drive behind the UIA project; without this, *Eyo* certainly would not exist. Even though Justin Mazzola Paluska got involved a little bit later, he's been invaluable ever since; much of the application-level designs and modifications are due to him. Sean Rhea formed a core part of the UIA team when we were doing the initial design and implementation. Sam Madden, a thesis committee member, made many helpful suggestions that demonstrated the utility of a fresh view towards the project.

Jamey Hicks and John Ankcorn provided feedback and encouragement throughout the design of this work. Franklin Reynolds, Zoe Antoniou, Dimitris Kalafonos, Paul Wisner, and Max Van Kleek helped tremendously while serving as an initial user group for UIA.

I wish to thank all of PDOS, both present and past, for providing such a supportive and animated atmosphere. May the spirit continue unabated. While I was first working on network measurements, along with Eddie Kohler, Sachin Katti, and Chuck Blake, much of the advising came from Dina Katabi.

Looking farther back in time, I should note that without the influence of several years of 6.001 students and staff, I would never have considered continuing on to graduate school. It was only after then that Frans and Robert started to teach me a little bit about what CS research actually is, and that I actually enjoyed it. Even farther back, Bob Welling for put code I'd written to real use for the first time.

I wish to thank my parents and family for pointing me toward the right path, and checking in along the way.

To Jessica, I simply say thank you, and that yes, you win.

I graciously thank Quanta Computer, Nokia Research Center Cambridge, and the National Science Foundation for funding support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Users often own many devices that combine storage, networking, and multiple applications managing different types of data: e.g., photographs, music files, videos, calendar entries, and email messages. When a single user owns more than one such device, that user needs a mechanism to access their data objects from whichever device they are using, in addition to the device where they first created or added the object to their collection. The storage capacities and network availability of these devices can vary significantly. Some fixed machines may always have a working network connection, and sufficient storage to hold an entire user's collection of media objects. Small mobile devices, in contrast, may contain significantly less local storage, and consequently can only store a small subset of a user's data locally. Mobile devices also frequently move to different locations with different network availability and cost, and may often be powered off to save energy. These events result in periods where devices have a slow connection to other devices if they are reachable at all. Systems that include such devices must be designed to handle network partitions as normal occurrences rather than an exceptional event. Providing highly available access to data in these settings therefore requires policies and mechanisms for replicating data across devices.

An individual person could now have a laptop, a tablet computer, a phone, a camera, a television, a digital video recorder, a photo frame, a desktop computer, a video camera, and a networked backup disk. All of these devices could display or manipulate the same type of data, such as digital photographs, and each of these devices can contain one or more network wireless or wired network interfaces. Thus, it would be useful to join such a set of devices into a distributed storage system to manage the same photo collection. The smallest of these devices may currently contain only a few gigabytes of storage, whereas the largest could easily hold multiple terabytes of stored objects. While these values will certainly increase over time, the relative disparity between the smallest and largest may not.

In settings with intermittent network connectivity, each device can manage only locally stored data, in isolation from other devices. As a result, today users see a storage abstraction that looks like "object a on device x", "object b on device y," etc.: it is up to the user to keep track of where an object lives and whether a and b are different objects, copies of the same object, or different versions of the same object. At a higher level, the user bears the burden of organizing object collections larger than a single device's storage, and synchronizing the collections on different devices. While a user could manually identify each object that needs copying, this approach quickly becomes infeasible given large numbers of files and only a few devices.

This thesis focuses on the problem of managing personal data objects in sets of devices including those that cannot hold the entire collection. In this situation, the user must partition the data collection among the devices, as well as duplicate objects that should be available from multiple devices. The overall goal of this work is to limit the complexity that end users and application developers encounter while managing data over a distributed collection of personal devices. The main approach this thesis takes toward this goal is the introduction of a new system property, *device transparency*, that allows users to think about their data collection in its entirety, rather than as the union of objects on a set of devices, as well as the design and implementation of a personal storage system, *Eyo*, that provides device transparency through a new storage API.

The remainder of this chapter describes existing approaches used in these situations, expands on the motivation for a device-transparent storage systems, and provides an outline for the rest of the thesis.

1.1 Existing Approaches

Storage systems that share data between different computers have a long history. Those systems were usually meant for settings where several different people, each with a workstation, shared a common set of data. For example, distributed file systems such as NFS [49] and AFS [24] have long allowed workstations to share files between multiple users while connected to a centralized set of servers. Alternatives to these systems supported types of disconnected operations, for example Coda [27] and Ficus [23]. These systems all aim to provide *network transparency*, where applications and users did not need to know whether a given object was stored local on the local machine, or on which remote server.

In contrast to these managed systems, the introduction of small mobile devices have lead to individual people needing to share data between devices that spend significant time disconnected from networks, or powered off, and that lack a traditional managed server. The lack of a managed server combined with disconnected device use has led users toward two main approaches for managing personal data over device collections, neither of which involve a traditional distributed filesystem.

Both models free the user from the complexity of manually managing files as they move between devices, and from having to remember which files should reside where. In one model, the set of devices is split into one master hub device and some number of edge devices that pass updates through the hub. The other model replaces the hub device with a cloud Internet service which all other devices use to access the data collection.

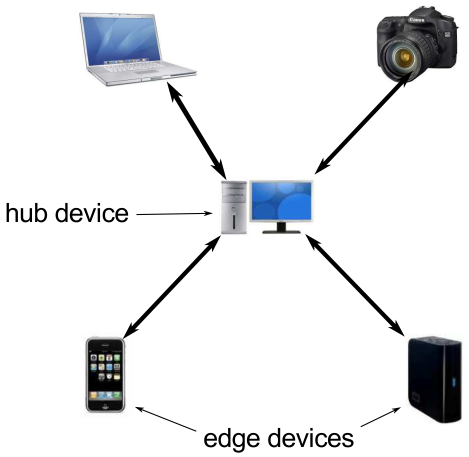

Figure 1-1: Hub-and-Spoke synchronization model

1.1.1 Hub-and-Spoke

In a hub-and-spoke usage model, as illustrated in figure 1-1 the user designates a single master device to hold a complete copy of the data collection. To copy data to another spoke device, the user brings that other device to the master and copies objects from it via a fast local connection, such as a direct cable or a local network. Any updates that should pass to other spoke devices first must be synchronized back to the master device first, and then passed from there to the other hub devices.

This arrangement has several advantages that aid management. Because the single master device holds all objects, viewing the complete collection simply requires viewing it on that master device. Handling concurrency is also simpler, because the hub device holds the authoritative copy of each object. If a user updates an object on an edge device, he must synchronize it with the copy already on the hub device before it can replace the original version as the authoritative version. The remaining edge devices will learn about the update when they next fetch new updates from the hub device.

Apple's iTunes [2] is one popular example of a current system using this model, helping the user to organize objects and synchronize with storage-limited music player devices. iTunes allows users to view their complete collection of media such as music and videos from one device. When users plug music players into a hub device via a USB cable, iTunes automatically passes updates in both directions so that each contains the updated files. Users can choose what subset of their collection should reside on each connected device based on sophisticated but easy to set rules based on artists, playlists, recent use or ratings. Edge devices generally cannot edit data other than metadata about recent uses, so the hub device can handle those kinds of conflicts without user intervention, though iTunes does fall back to user intervention when the specified collection doesn't fit on a mobile device.

iTunes does demonstrate several limitations to the hub-and-spoke model. (1) The hub device must be available to exchange updates. (2) It is limited to star-shaped device topologies. Edge devices cannot exchange updates directly without the hub device, even if they

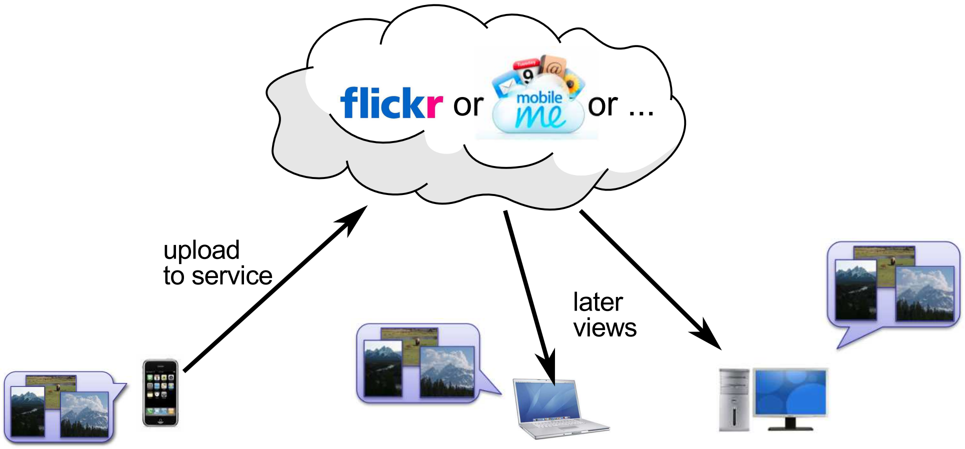

Figure 1-2: Cloud Service synchronization model

are directly connected or on the same local network. (3) Edge devices cannot even show users the complete list of files unless they have enough storage capacity to hold all the file content. (4) Hub device must hold the entire collection, which means the total collection size is limited by that one device. (5) There must be exactly one hub device per user, which means that a collection cannot be composed only of edge devices, nor can it have two or more hub-class devices. While these limitations may not affect all users, they are fundamental to the hub-and-spoke device organization and cannot easily be addressed without moving to a different usage model.

1.1.2 Cloud Storage

The key limitations in the hub and spoke model (dependence on a single hub device) have been recognized before, and are one of the many reasons for recent proliferation of online photo sharing websites (e.g., Flickr [15], SmugMug [52], Picasa [41]), as well as more general storage systems such as MobileMe [3] and infrastructure cloud storage services such as Amazon's S3 [1]. These services exemplify an alternate cloud-based storage model. In this organization, as shown in figure 1-2, a single website replaces the central hub device of the hub-and-spoke model. Though the website may in fact itself be constructed as a distributed system, from the point of view of the end user, there is once again a single location to view their entire data collection. All of the user's devices can access the single collection via web browsers, as long as they have a working network connection to the cloud service. Edge devices need not be able to store the complete collection via local storage, as edge devices generally only hold copies of objects when they are imported into or exported from the system.

The cloud service approach gains several advantages over the hub and spoke model, as the service can usually maintain much higher availability with less geographical dependence than a single user's device. The cloud approach also adds several new limitations, however: (1) The service provider may limit the types of supported data to those usable by a single application. (2) Edge devices must have a working Internet connection. (3) Edge devices still cannot directly exchange updates, even when they have a fast local network available. (4) All data accesses need to go through the central service rather than be handled locally, which can be slow and expensive due to latency and bandwidth limitations.

1.2 Device Transparency

Both the cloud storage and the hub-and-spoke models for managing data across devices share a key feature: there is a designated location to view the complete collection, but both place limitations on the devices and conditions that can provide such a global view.

An improvement to both such organization would provide the same global view on all of a user's devices, regardless of storage capacity or network connectivity. Inspired by location transparency [60] for network file systems, we name this the goal *device transparency*: the principle that the user should be able to view and manage the entire data collection from any device, even if the data is partitioned and replicated across many devices, and even when not all data objects may be available all the time (e.g., when the user is off the Internet carrying only one or two storage-limited devices). A device transparent storage system presents each object as a first-class entity, rather than as an object-on-a-device, and hence allows the user to manage data as a unified collection rather than in isolated device-sized fragments. Unlike location transparency, which states that the name of objects should remain the same even if its location changes, device transparency states that the set of objects visible should not change by viewing from a different device.

Given the possibility of disconnection, each device must store a replica of some or all of the data associated with each object in order to provide device transparency. A consequence of replication is that the user may make concurrent conflicting modifications. It is well known that such conflicts can often only be resolved with application-specific logic. Existing systems resolve conflicts at synchronization time using separate resolvers that understand application file formats (e.g., Coda [27] and Ficus [23]). This approach is often insufficient. For example, a user may wish to defer resolving conflicts, and instead explicitly see and use multiple versions of an object. This can only be handled by the storage system preserving multiple versions after synchronization, and by applications being aware during ordinary operation of the existence of multiple versions.

This thesis's core contribution is two design observations critical for any system that is to provide device transparency. First, *content must be separated from metadata*, so that a complete set of metadata may be copied to storage-limited devices which can use it to present the user with a device-transparent interface. Second, *applications must see divergent versions and conflicts as first-class entities*, so that they can automatically resolve most common divergent version histories without user intervention, and incorporate presentation and resolution of other conflicts to the user as part of ordinary operation. For a storage system to be used by multiple applications, these observations must be reflected as critical features in the API. In contrast to application-specific resolvers [28], the storage API must separate data from metadata, and must present objects in terms of version histories.

1.3 *Eyo* personal storage system

A central question is how much developers must do to gain the benefits of device transparency, and whether these benefits are substantial enough. In order to answer this question, we implemented a prototype storage system for personal data, named *Eyo*, which provides device transparency. *Eyo* is a special-purpose storage system designed for a single user with a small number of devices.

Eyo faces several design challenges driven by the need of supporting disconnected operation: (1) Limited storage space on devices, (2) Concurrent updates while disconnected, (3) Continuous synchronization without user direction, (4) Applications must automatically resolve conflicts arising from concurrent object modifications, and (5) the system must be able to locate data held on disconnected devices.

To address these challenges, *Eyo* provides a new storage API to applications. *Eyo* expects applications to separate object metadata from object content. *Eyo* replicates the metadata to all of a user's devices. *Eyo*'s API allows applications to create and locate objects via metadata attributes, examine and manipulate recent object version histories, and to register for notifications when other devices and applications add new changes. *Eyo*'s API allows applications to specify placement rules, as in Cimbiosys [44] and Perspective [48], which instruct *Eyo* to store copies of objects on selected devices.

Applications delegate most of the work of inter-device synchronization to *Eyo*. In order to minimize the cases where devices do not all see the most recent versions of a file, which can lead to conflicts, *Eyo* uses a fast synchronization protocol to automatically pass updates between devices without user intervention. When disconnected operations do lead to update conflicts, *Eyo*'s API provides notifications to applications which in many cases permit those applications to automatically resolve conflicts without user intervention.

1.4 Evaluation

To evaluate the feasibility of interacting with a device-transparent storage system, both for application developers and for end users, we used *Eyo* to examine the following questions: (1) What can end users do by using *Eyo* that they could not do otherwise? (2) Is *Eyo*'s API a good match for real applications? (3) Do *Eyo*'s design decisions, such as splitting metadata from content, and global metadata replication, unduly burden devices' storage capacity and network bandwidth?

We modified five applications to use *Eyo* as their storage system: a photo editor, two media players, a podcast manager, and an email interface. The original versions each implemented a specialized storage system atop a traditional file system. Our modifications transformed these applications into distributed applications that no longer act in isolation on a single device. Replacing these internal storage systems with *Eyo*'s was simple, didn't increase application complexity, and required no changes to the user interface to present the application's global data collection. Separation of metadata and content, version histories, and placement rules allowed the applications to provide a device-transparent storage systems to users. For example, in the photo editor, users can perform basic tasks such as adding tags to photos, searching for photos matching tags, and viewing thumbnail versions of photos, even from devices that cannot store the complete collection. In most cases, the modified applications can automatically handle concurrent changes to the same objects without user intervention.

To investigate the storage and communication costs of device transparency, and to evaluate *Eyo*'s metadata-everywhere design, we ran experiments with our modified applications using personal data sets. The costs of storage and bandwidth proved reasonable for typical portable devices. These experiments show that the metadata-everywhere approach to implementing device transparency imposes only modest storage and bandwidth costs for typical usage. *Eyo*'s synchronization protocol aims to quickly identify changes to permit devices to synchronize continuously without user intervention. To evaluate this design, we compared *Eyo* to several other data synchronization systems, and found that *Eyo* propagates updates faster than stand-alone file synchronization tools as well as cloud synchronization services.

1.5 Contributions

This thesis makes the following contributions:

- The articulation of the goal of device transparency, whereby each device shows the same data collection to applications and users.

- A new storage API for applications that separates object metadata from content, and provides first-class version histories.

- The design and implementation of *Eyo*, which is the first device-transparent storage system for disconnected collections of personal devices.

- Distinct metadata and content synchronization protocols that permit *Eyo* to continuously pass updates between devices whenever connectivity permits, without user or application direction.

- An evaluation of *Eyo* with real applications that shows that the new API is a good fit for users and applications, provides new features to end users not available previously, permits applications to handle many types of concurrent updates automatically, all within the storage and bandwidth capabilities of typical portable devices.

While *Eyo* uses many techniques pioneered by existing systems (e.g., disconnected operation in Coda [27], application-aware conflict resolution in Bayou [58], placement rules in Cimbiosys [44] and Perspective [48], version histories in Git [19] and Subversion [56], update notifications in EnsemBlue [39]), it is the first system to provide device transparency for disconnected storage-limited devices.

1.6 Limitations

There are several limitations inherent in the goal of a device-transparent storage system, in addition to more specific limitations that apply only to *Eyo*. This section describes several of these limitations; chapter 8 discusses ways to address these limitations through extensions or modifications to *Eyo*'s design.

Any device-transparent storage system that supports viewing an entire data collection from disconnected devices must place all of that collection's metadata (or alternately, all content as well) on each device. This requirement means that the smallest device in each group must be able to hold a copy of all of the metadata. For systems that do not need to support disconnected operation, the storage capacity is instead limited by the largest device, rather than the smallest.

While the ideal of a device-transparent storage system could be generally applicable to a variety of systems, *Eyo* is designed for a more limited set of uses. It is meant for small groups of devices owned by a single user, so that devices within such a group do not need to control access to individual files within the group, and the number of devices is small enough that replicating small messages to each device is reasonable. Section 8.1.2 describes a possible extension to multiple users.

Eyo's storage system is targeted towards data types where the metadata changes frequently, but the underlying data objects rarely change often, if they change at all. All of the applications we have examined satisfy this assumption. If the applications did change data frequently, the tools available to applications for resolving conflicts would need augmenting from the current set targeted towards metadata-only changes. An additional assumption about the types of data that *Eyo* makes is that the data objects will be larger than the metadata that describes those objects, which once again is true for all of the media-file examples we examined, but might not be true in other types of uses, such as if the data objects were individual measurements collected by devices and stored in a centralized storage system.

The ability to allow a heterogeneous group of devices to participate in a single storage system that requires changes both to the system software on those devices, and to individual applications requires that some mechanism exist to deploy storage system software to those devices in the first place. This thesis will not address this requirement, though there are several different paths by which *Eyo* could be deployed onto devices. The most straightforward would be if the device's manufacturer built in support for *Eyo*. An increasing set of portable devices now permit end users to install applications onto their devices, though the capabilities granted to applications differ by system. Ensemblue [39] describes a method for supporting devices that expose only a simple storage interface but do not allow user-supplied software, and *Eyo* could in principle adopt some of these approaches for similar devices. For more traditional desktop and laptop operating systems, application developers could of course choose to build *Eyo* directly into applications without needing any further coordination. Though this approach would not result in a fully-general deployment, it would still provide a direct benefit to end users while using those specific data types and applications.

1.7 Outline

The remainder of this thesis is organized as follows. Chapter 2 describes *Eyo*'s design goals, and provides an overview of the overall system design. Chapter 3 presents the storage API that *Eyo* provides for applications. Chapter 4 describes *Eyo*'s synchronization mechanisms, and Chapter 5 summarizes *Eyo*'s prototype implementation. Chapter 6 evaluates *Eyo*'s design with existing applications and object collections. Chapter 7 puts *Eyo*'s contributions in the context of previous systems, Chapter 8 considers extensions and alternatives, and Chapter 9 concludes.

Chapter 2

Overview

This chapter provides a an overview of the main challenges in building a device-transparent storage system, outlines the main approach to solving those challenges along with application assumptions, and provides a high-level description of the design of the *Eyo* storage system.

The main challenge in providing a device-transparent storage system is supporting disconnected operations. To illustrate this challenge, consider how users tackle this problem today. If a user has several devices which can display photos, limited storage space means that the user must manually decide which subset of the collection to copy to each. Additionally, they must organize the devices into a star topology, where one master device holds the authoritative copy of each object.

Using devices while they cannot communicate with the master replica means that changes to individual photos will only be eventually consistent with the master replica. The user cannot exchange updates between two non-hub devices, even if they are on the same local network, since those two devices might not hold the same collection to begin with.

The process of synchronizing updates between devices is entirely dependent on user direction, both for remembering which devices hold updates, which other devices need those updates, and how to handle any conflicts between the edge devices and the hub. In many cases, users need to manually examine each replica to decide what the final state of an object should be. Merging conflicting changes made to separate replicas may be entirely up to the user, making any method other than simply choosing one ‘winner’ version to replace both infeasible.

The risk of this mode of operation is that replicas can diverge if the user forgets to reconcile updates between replicas. Each new difference makes it more complicated to manage the data collection, and hence more likely that the collection will diverge even more over time.

A device-transparent storage system should automate as much of the data management process as possible. By limiting opportunities for the data collection to diverge, the system can provide a better approximation to a truly device-transparent collection, even in the face of disconnected updates. Doing so requires enough automation that the user no longer needs to keep track of which devices and data objects need updates.

We have built a storage system, *Eyo*, that combines three main approaches to constructing a storage system with several simplifying assumptions about application behavior to build a specialized device-transparent storage system for personal media collections.

2.1 Approach

Eyo's approach to device-transparency includes three main components:

- **Separating Metadata from Content:** In order to ensure that all devices, including disconnected devices, know about all objects, the metadata for those objects must be replicated to each device. Object content, however, cannot fit on all devices, so each device will hold some subset of the total collection. Individual content objects may be replicated to more than one device if they are important. This separation affects all layers of the system design, as both the applications, and the end users, need to interact with data items that may have only the metadata accessible and not the content.

- **Peer-to-Peer Continuous Synchronization:** Any pair of devices that communicate should be able to exchange updates. This approach both limits possibilities for object replicas to diverge while out of touch with a hub device, and also aids in object availability. If a user doesn't have content available on one device, it might be available from a nearby device instead. The process of synchronizing devices must proceed continuously without direction from users so that devices present the same data collection as soon as network connectivity permits. If two devices have a working network connection, updates from one should appear immediately on the other device without any additional user action.

- **Automated Conflict Resolution:** Even with continuous synchronization to pass updates as quickly as possible, intervals of no communication between devices will result in concurrent changes to the same object. These cases need to be resolved automatically as often as possible without user intervention. Automating conflict resolution requires application cooperation, as only applications understand both the format of data, and the types of reasonable changes that may occur. The storage system needs to provide an API that makes it easy for applications to identify concurrent updates, reason about their meaning, easily resolve the common types of conflicts automatically without user intervention.

2.2 Application Assumptions

In addition to the three approaches described earlier to handle disconnected operations in a device-transparent storage system, *Eyo* makes several assumptions about the types of applications, their organization, and the types of data stored in the system. These assumptions transform a problem that is quite difficult in general into one that is feasible in practice, by limiting the number of participating devices to one personal group, limiting the amount of metadata in the system to one person's collection, and limiting the update patterns to structured metadata instead of arbitrary data of unknown types. *Eyo* requires that these assumptions hold in order to perform reasonably well. Even with these limitations, however, *Eyo* proves to be well suited to personal data collections.

Eyo is meant to be used by applications that manage large collections of objects for the user, such as e-mail messages, song tracks, images, videos, etc. These applications must keep separate notions of *object metadata* (author, title, size, classification tags, play count, etc.) and *object content* (image data, message body, etc.). This separation of metadata from content must be carried through the application so that the user interface makes sense even when the device can show only metadata but does not have the associated content available locally. For example users could view lists of songs or message headers, search by name, genre, or composer, sort by rating or play count. All of these uses would not require the associated content, which would be the message body text or the song's audio data.

In addition to the difference between always-present metadata and sometimes-present object content, *Eyo* assumes that each class of data undergoes different update patterns. Modification of metadata is common, as are creation and deletion of objects, but modifications to object content is rare. For example, a user may modify the set of folders in which an email message appears, but the message content itself does not change after its initial addition to the system. *Eyo* does not require that content be immutable. If content does change, *Eyo* assigns a new identity to each content version, unlike version control systems that merge source code content changes line-by-line.

Although *Eyo* allows applications to place arbitrary data in metadata, metadata must be small enough that a user's entire collection of metadata can fit on each of that user's devices. This requirement enforces a relation between the smallest device in a personal group and the amount of metadata that the collection can hold. Object content is instead limited by the sum total of the devices' storage capacity in a group of devices.

Eyo assumes that application developers agree upon a basic set of semantics regarding metadata for common data types in order to permit multiple applications to share the same data objects. Applications can still attach arbitrary data to metadata in addition to the commonly agreed upon portions. For example, applications could agree to use the standard header fields for email messages, fields analogous to the exif data in jpeg image files, or the ID3 tags from MP3 for audio files.

2.3 Eyo

We have designed and built a personal storage system, *Eyo*, that uses the approach described in this chapter to provide device-transparent personal storage. *Eyo* provides a new storage API to applications which separates object content from metadata. *Eyo* continually synchronizes updates between peer devices as soon as network connectivity permits. The *Eyo* API provides applications with two explicit history information to automatically

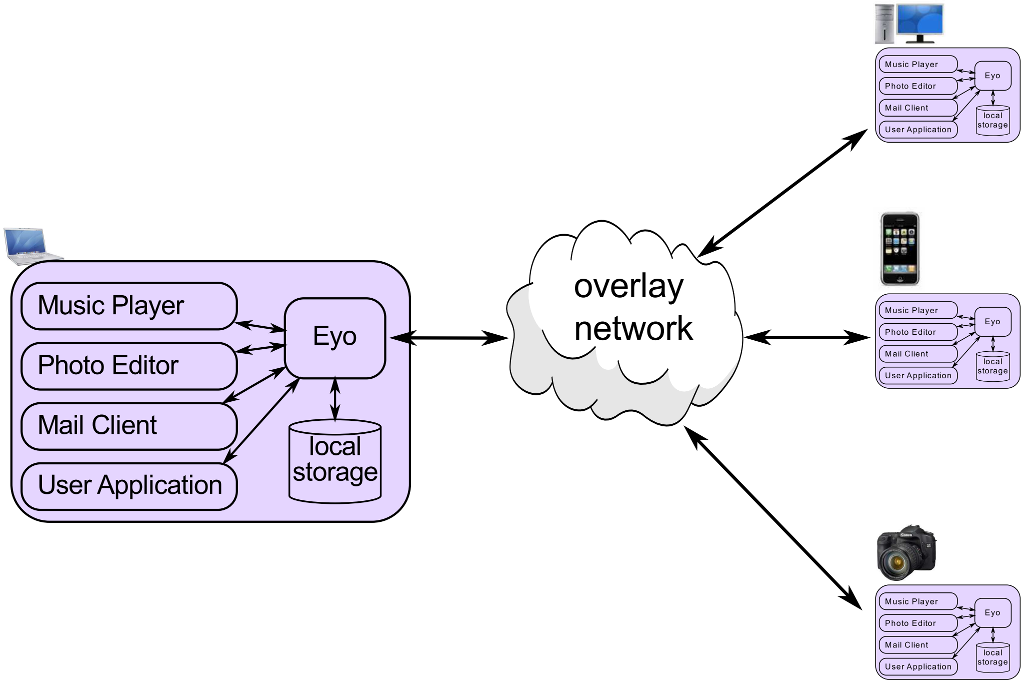

Figure 2-1: Eyo sits between applications and local storage. Eyo uses an overlay network to manage all inter-device communication.

resolve conflicts, and placement rules to specify which devices should hold which content objects.

2.3.1 Design Overview

Eyo sits between applications, local storage, and remote network devices, as shown in Figure 2-1. Placing *Eyo* directly between applications and local storage means that *Eyo* learns about all local updates directly as a result of application requests. *Eyo* then uses an overlay network to identify a user's devices, and track them as they move to different locations and networks. *Eyo* manages all communication with these devices directly, and determines which updates it must send to each peer device whenever those devices are reachable. As a consequence of this involvement, *Eyo* learns about all remote updates directly and notifies applications as appropriate.

2.3.2 API Features

The *Eyo* storage API provides several features not found in traditional filesystem interfaces, such as separate content and metadata, explicit version histories, and event notifications. Applications using *Eyo*'s storage API can attach arbitrary metadata tags to objects, and then use those application-specified metadata attributes to search for files.

Unlike the extended attributes found in many Unix filesystems, *Eyo* ensures that metadata searches are efficient enough to serve as the primary naming mechanism applications use to locate objects. Multiple applications can thereby share the same objects without needing to agree on a single directory and path name, or on an identical directory hierarchy for each device.

The storage API provides a notification mechanism for applications to learn about updates immediately without polling. For example, applications can learn about writes performed on other devices, as well as events such as changes to the set of content objects are locally available.

Eyo tracks the recent modification history for objects, and provides that history to applications. If after modifying an object, if *Eyo* can ensure that no other application or device modified the same object at the same time, *Eyo* will supply just that version to applications.

If an object has been concurrently modified on multiple devices so that there are multiple newest versions, *Eyo* will preserve and synchronize those versions and all versions back to the most recent common ancestor version. *Eyo* will present all of these versions to the application. The application can automatically reconcile the changes (e.g., if the only changes are increments of play counts), ask the user which version should supersede the others, or let the user see and use all versions much as if they were separate objects. *Eyo* attempts to pass updates between devices as soon as possible to minimize opportunities for conflicts to occur.

Eyo will automatically and continuously perform pairwise synchronizations between devices so that all devices know about all metadata and (subject to space availability) content. *Eyo* propagates both new data and modifications to data. Thus, for example, if a user changes a song title on an iPhone device, and adds a new song on a laptop, both devices will see both changes as soon as they are able to communicate. If the user then takes the iPhone to work, where it can communicate with a desktop machine, the desktop machine can learn about both changes by synchronizing with the iPhone, even if there is no direct communication path between the laptop and desktop.

Eyo automatically copes with devices with too little storage to fit all content. It allows applications (and thus users) to guide its decisions about which devices should store which content. Applications can modify these placement policies from any device in a user's personal group. Devices can change policies without being able to contact the affected device, though the new policies will not take effect until after the update later reaches the affected device. If the device group has adequate storage space and network capacity to satisfy the user's desired object placement policies, *Eyo* devices will eventually converge to a state where each device holds the specified objects.

Eyo provides applications with a reasonably accurate guess of which device(s) hold the content for each object. Thus an application can allow a user to search for objects by metadata (e.g., search for all images taken in a certain location), and then tell the user which device probably has the associated content. The location information does not require any network communication, though it will lack all changes (additions as well as deletions) since last receiving remote updates.

Many of the properties described here are similar to properties described in earlier systems, though *Eyo* provides different eventual consistency guarantees for metadata and content. For example, metadata synchronization fulfills a *prefix-property* [40] where devices learn of all earlier updates that either knew of prior to communicating. *Eyo* does not enforce any ordering relation between versions of different objects. Metadata updates to a single object define a partial order of happened-before relationships [29], rather than an *eventually-serializable* [13] set as in Bayou [58]. *Eyo* does not provide the prefix property for object content: devices may learn about updates before they can see the related content. When space permits, *Eyo* provides *eventual filter consistency* [44] for object content, meaning that, subject to space and bandwidth, devices eventually hold objects best matching placement policies.

2.3.3 Design Challenges

Eyo faces two main design challenges: a storage API with automatic conflict resolution, and protocols for fast synchronization between devices.

The API must provide applications with enough information so that they can easily handle conflicts automatically without requiring that users manually clean up objects after accessing them from multiple devices. The data model that *Eyo* provides must match applications' needs well enough that common uses require only straightforward resolution strategies instead of arbitrarily difficult procedures.

Eyo's synchronization protocols need to efficiently pass updates between each of the devices in a user's collection, and do so quickly in order to minimize both the chance of out-of-date collection state leading to conflicts, and to limit the amount of bandwidth consumed passing file updates. In order to pass updates as soon as possible to other devices, *Eyo* must learn about changes as soon as applications make them. *Eyo* cannot rely on scanning the local storage system at synchronization time (for example, whenever another device becomes reachable), as that approach would take too long to identify which changes need be sent to run continuously if the devices remain in contact.

The next chapter describes the details of *Eyo*'s storage API, and how applications use it, followed by a chapter that describes how *Eyo* synchronizes updates between different devices.

Chapter 3

A Device-Transparent Storage API

This chapter describes the features of a device-transparent storage API, explains how *Eyo* provides those features, and illustrates the need for those features in the context of managing photograph collections.

3.1 Objects, metadata, and content

In order to provide device-transparent storage, *Eyo* provides a storage API that makes the split between metadata and content explicit.

Eyo stores a set of objects on each device, as Figure 3-1 shows. Each object has a unique non-human-readable identifier, and corresponds to one user-visible application object, e.g., one photo. An object consists of a directed acyclic graph of object versions. Edges in the version graph denote parent-child relationships between those versions, which child versions note through predecessor pointers to the parent versions. Each object version consists of metadata. An object version's metadata consists of a set of *Eyo*- and application-defined key/value pairs, or attributes; for example, a digital photo's metadata may include the key/value pair `(ISO, 400)`. The metadata also contains a content identifier; the associated content might or might not be present on the device. A content item consists of application-defined data; for example, a JPEG-encoded image.

Eyo stores a flat set of objects, without structure such as directories or file names. Applications are expected to organize their own objects for presentation to the user, perhaps by storing various tags in metadata attributes. *Eyo* lets applications retrieve objects via queries on the metadata attributes. For example, a photo application may add `date`, `subject`, and `location` tags to photos in order to help it organize and retrieve photos for the user. Applications are expected to store enough information in the metadata to be able to display meaningful information to the user about an object even on devices not storing the content. Applications should use care in setting attribute names when multiple applications may access the same object by using commonly agreed upon fields, and prefixing special-purpose fields with an application-specific prefix, just as applications need to respect the meaning of, e.g., id3 tags on music files or exif data in image files.

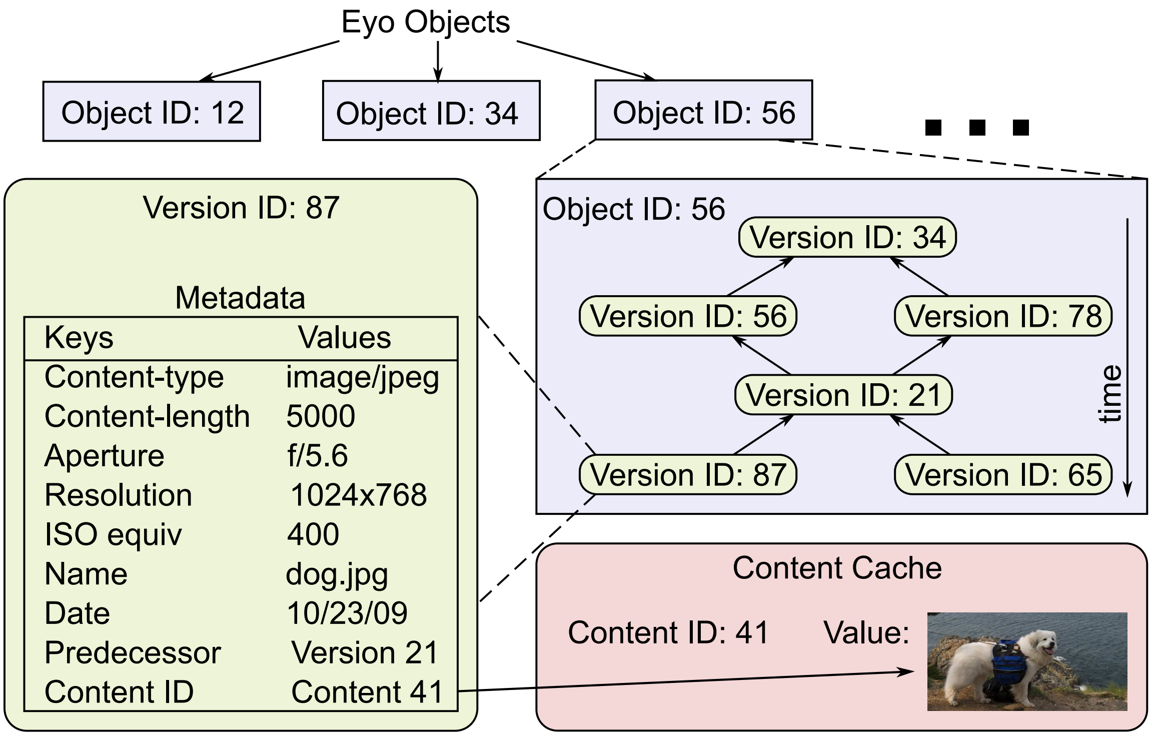

Figure 3-1: *Eyo* object store.

Eyo's API provides applications with operations to explore the data store, to read, create, and update metadata and content, to learn about and repair conflicts, to specify content placement rules, and to receive notices about changes to the object store. Figure 3-2 lists the commonly used *Eyo* methods. The figure omits alternate iterator-based versions of these methods for constructing or viewing large collections, and combined versions of these base operations for common usages. For example, applications might read the metadata for an object, add one new attribute to the metadata, and write a new versions with that tag. All of these methods access only device-local data, so no method calls will block on communication with remote devices.

If an application tries to read the content of an object, but the content is not present on the device, *Eyo* will signal an error. A user can perform useful operations on metadata even from a device that doesn't store content, such as classifying and reorganizing MP3 files. If the user wants to use content that is not on the current device, the system can use the metadata to help the user find a device that has the content, or ask *Eyo* to try to fetch the content, using the placement methods in the API (Section 3.5). Finally, having the metadata everywhere allows for efficient synchronization (see Chapter 4).

Eyo usually assigns random object identities when creating new objects. Applications which import external objects may pass an optional hint to CREATE to ensure that importing the same object from multiple devices does not result in duplicates. Section 6.3 describes an example use of creation hints.

object creation and manipulation:
(*objectID*, *versionID*) ← CREATE(*ID hint*)
(*objectID*, *versionID*)[] ← LOOKUP(*query*)
versionID[] ← GETVERSIONS(*objectID*)
(*key*,*value*)[] ← GETMETADATA(*objectID*, *versionID*)
contentID ← OPEN(*objectID*, *versionID*)
contents ← READ(*contentID*, *offset*, *length*)
versionID ← NEWVERSION(*objectID*, *versionID*[], *metadata*, *contents*)
versionID ← DELETEOBJECT(*objectID*)
placement rules:
ruleID ← ADDRULE(*name*, *query*, *devices*, *priority*)
(*ruleID*, *query*, *devices*, *priority*) ← GETRULE(*name*)
(*ruleID*, *query*, *devices*, *priority*)[] ← GETALLRULES()
REMOVERULE(*ruleID*)
event notifications:
watchID ← ADDWATCH(*query*, *watchFlags*, *callback*)
REMOVEWATCH(*watchID*)
callback(*watchID*, *event*)

Figure 3-2: *Eyo* API summary. Event notifications are discussed in section 3.4, and placement rules in section 3.5.

3.2 Object Version Histories

Much of the design of the *Eyo* API and storage model is motivated by the requirements of device consistency for potentially disconnected devices. Such devices must carry replicas of the *Eyo* object store and might make independent modifications to their replicas. Therefore, devices must be prepared to cope with divergent replicas.

When an *Eyo* application on a device modifies an object, it calls NEWVERSION() to create a new version of that object's metadata (and perhaps content) in the device's data store. The application specifies one or more parent versions, with the implication that the new version replaces those parents. In the ordinary case there is just one parent version, and the versions form a linear history, with a unique latest version. *Eyo* stores each version's parents as part of the version.

Pairs of *Eyo* devices synchronize their object stores with each other (see Chapter 4 for the protocol details). Synchronization replaces each device's set of object versions and metadata attributes with the union of the two devices' sets.

For example, in Figure 3-1, suppose device A uses *Eyo* store a new photo, and to do so it creates a new object $O56$, with one version, $O56$:34, and metadata and content for that version. If A and B synchronize, B's object store will then also contain the new object, its one version, that version's metadata, and perhaps the content. If an application on B then modifies the metadata by replacing the default camera-defined file name with a user-specified value for $O56$ and perhaps replacing the content after editing the content, the application will call NEWVERSION(O56, [O56:34], *newmetadata*, *newcontent*), indicating that the newly created version during the call, $O56{:}78$, should supplant the existing version. When A and B next synchronize, A will learn about $O56{:}78$, and will know from its parent that it supersedes $O56{:}34$. Again, the version history is linear, and *Eyo* applications will use the unique most recent version.

A more interesting situation arises if A had produced a new version of $O56$ before the second synchronization with B, such as adding additional `category` or `location` tags to the photo. In that case, both new versions would have parent version $O56{:}34$. After synchronization, A and B would both have two "latest" versions of $O56$ in their object stores. These are called *head* versions.

It is this case, in which there is no unique head version, that motivates much of the *Eyo* API. One strategy is for the application to continue on with divergent versions, presenting both to the user in object lists, and letting the user modify either or both. Another strategy is for the application to automatically merge the two head versions, producing a single new version that supersedes both by indicating that it has two parents; version 21 in Figure 3-1 is the result of such a merge. Another possibility is for the application to allow the user to specify how to merge the versions, perhaps by indicating that one should override the other.

Eyo's version graphs with explicit multiple parent versions are inspired by version control systems [19, 56], though used for a different purpose. Where version control systems keep history primarily for users to examine, *Eyo* uses version history to hide concurrency from users as much as possible. When combined with synchronization, version graphs automatically capture the fact that concurrent updates have occurred, and also indicate the most recent common ancestor. Many procedures for resolving conflicting updates require access to the most recent ancestor. Since *Eyo* preserves and synchronizes complete version graphs back to those recent ancestors, applications and users can defer the merging of conflicting updates as long as they want. For example, instead of either missing a fleeting functioning network opportunity or interrupting the user at an inopportune time to ask about an irrelevant data object, *Eyo* allows the user to wait until some more convenient time to merge conflicts, or perhaps ignore the conflicts forever. In order to ensure that parent pointers in object version histories always lead back to a common ancestor, *Eyo* transfers older versions of metadata before newer ones during synchronization [40].

3.3 Conflicts

The primary goal of *Eyo*'s API is to enable automated conflict management. In order to carry out these functions, applications need access to history information, notifications when conflicts arise, and a way to describe the resolution of conflicts.

Because applications hold responsibility for handling concurrent updates of the same object on different devices, those applications should structure the representation of objects in a way that makes concurrent updates either unlikely or easy to merge automatically whenever possible. Applications must notice when concurrent updates arise, and when they do, applications should either resolve them transparently to the user, or provide ways for users to resolve them.

When it detects concurrent updates, *Eyo* presents to the application each of the head versions along with their common ancestors. Alternative designs could have (1) chosen a single arbitrary head version and discarded the rest; (2) presented the application with all head versions but no older versions; or (3) given the application all the head versions along with corresponding version vectors. *Eyo* does not use these approaches because alternative 1 would silently drop changes, and alternatives 2 and 3 would place a heavy burden on applications to figure out what changes happened on which devices and thus to compose a reconciled version which reflects the user's intent.

Eyo's version history approach permits many concurrent updates to be resolved automatically and straightforwardly by the application. For example, a user may move a mail message between folders on one device, and set the 'replied' attribute flag from another, or two devices may each update the playcount on a song while disconnected. Applications can arrange for these pairs of operations to be *composable*. For mail messages, folder tags and status bits can be set independently in the metadata. For songs, the merged playcount should include the sum of the differences between the most recent common ancestor and each of the concurrent versions. *Eyo* identifies these conflicting modifications, but the applications themselves merge the changes. The applications know the uses of these attribute types, and so can clearly determine the correct final state for these classes of concurrent changes.

Some concurrent updates, however, require user intervention in order to merge them into a single version. For example, a user might change the same song's title in different ways on different devices. In such cases it is sufficient for *Eyo* to detect and preserve the changes for the user to either fix at some later time or ignore entirely. Because *Eyo* keeps all of the relevant ancestor versions, it is simple for the application to show the user what changes correspond to each head version.

Eyo can discard versions prior to the most recent common ancestor of an object's multiple versions to reclaim unneeded storage space. Figure 3-3(c) shows a graph with one resolved conflict followed by an unresolved conflict. In this graph, once all devices know about the version B:2, it is a *unique ancestor* for the object, and *Eyo* may *prune* the version graph, deleting the older versions (A:1, C:1, and A:0). Applications may not intentionally create conflicts: when calling NEWVERSION(), applications may only list head versions as predecessors. This requirement means that once a unique ancestor is known to all devices in a personal group, no version that came before the unique ancestor can ever be in conflict with any new written version or any newly learned version. These pre-unique-ancestor versions can thus be removed without affecting later conflict resolution schemes. If a single device writes several successive versions of an object (i.e., a linear version graph), it may coalesce those into a single version before synchronizing. Section 6.6 discusses storage costs when devices do not agree on a unique ancestor.

Applications permanently remove objects from *Eyo* via DELETEOBJECT(), which is just a special case of creating a new version of an object. When a device learns that a

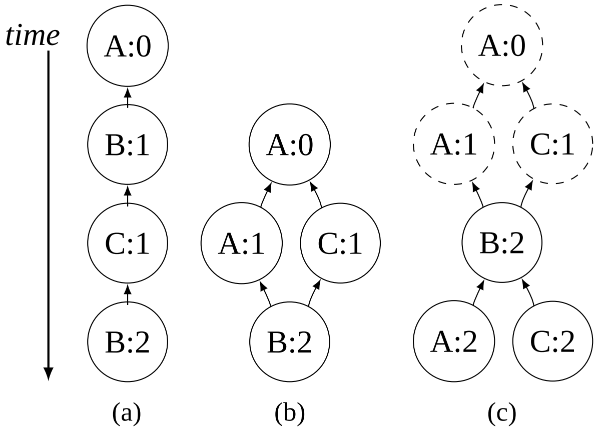

Figure 3-3: Example version graphs showing predecessor relations between versions of a single object. (a) contains a sequence of four versions with no conflicts, even though there were three different writers, devices A, B, and C. (b) shows a resolved conflict. (c) shows an unresolved conflict. There are two head versions, A:2 and C:2, with the unique ancestor B:2. The dashed versions may be pruned after all devices learn about the unique ancestor.

delete-version is a unique ancestor (or that all head versions are deletes, and seen by all other devices), *Eyo* deletes that object from the metadata collection.

3.4 Queries

While *Eyo* does not provide human-readable object identifiers, it does allow queries with which applications can implement their own naming and grouping schemes. Several of *Eyo*'s API methods (e.g., LOOKUP(), ADDRULE(), ADDWATCH()) use these queries to search for objects and to define placement rules. Queries return object IDs for all objects that have metadata attributes matching the query.

Eyo's LOOKUP() call performs a one-time search, whereas ADDWATCH() creates a persistent query. Watch queries allow applications to learn of new objects and object versions, and to observe the progress of *Eyo* inter-device synchronization. *Eyo* watch callbacks fulfill a purpose similar to single-filesystem notification schemes such as inotify [32].

Eyo's queries use a subset of SQL, allowing boolean combinations of comparisons of metadata values with constants. Such queries are efficient to execute but limited in expressiveness. For example, the language does not directly support searching for the 10 most-played songs or the newest mail message. An application can instead specify queries such as *all music with a rating above 4*, or add tags directly to the objects that should be included in an automatically-maintained collection. *Eyo* limits queries to these restricted forms to simplify those uses (watch events and placement rules) that must evaluate queries in two different contexts: evaluating new or changed queries to identify which objects match, and determining which existing queries match new or modified objects. As in Perspective [48], users never see *Eyo* queries; applications create queries on their behalf.

3.5 Placement Rules

Eyo allows applications to specify *placement rules* controlling which objects' content has highest priority for storage on storage-limited devices, much as related systems do [44, 48]. Applications are expected to generate placement rules based on user input.

Applications specify placement rules to *Eyo* using the query language. A placement rule is the combination of a query and the set of devices that should hold objects matching the query. For example, an application might give every object in a playlist the same tag, and present a UI allowing the user to indicate which devices should hold the complete playlist. An application can also let users specify particular objects and the devices on which they should be placed.

Each rule has a priority, and a storage-limited device tries to store high-priority content in preference to low-priority. Devices trade responsibility for content to avoid deleting the last copy of any item (see section 4.6). When space permits, *Eyo* provides *eventual filter consistency* [44] for object content, meaning that each device eventually gathers the set of objects that best matches its preferences. *Eyo*'s synchronization mechanism, as described in section 4.6, ensures that at least one copy of content persists even if no placement rule matches.

Eyo ensures that all devices know all placement rules by storing each rule as an object with no content, but with attributes containing the query, priority, and device set. Any device can modify a placement rule, and if a conflict arises between rule versions, *Eyo* conservatively applies the union of the requirements of all head versions. Similarly, if any of an object's head version matches a placement query, then *Eyo* acts as if the query had matched all versions back to the common ancestor. This ensures that devices have the content associated with all the versions required to recognize and resolve conflicts.

Experience suggests that users are not very good at predicting what objects they will need, or at describing those objects with rules [48]. *Eyo*'s metadata-everywhere approach makes it easy to find missing objects by searching the metadata, to discover which devices currently have copies of the object, and to fix the placement rules for the future.

Because placement rules operate at the granularity of objects, applications that store related content together should express these links as separate objects with links from the metadata from one to the other so that different placement rules can apply to the variations. For example, an application may wish to store both a full size and a thumbnail size image of the same base photo, but assign a high priority placement rule to place the thumbnail size objects on most devices, but only place the full size versions on a few high-capacity devices.

Placement rules do not guarantee that a group of devices reaches the optimal placement of objects to devices in the face of limited storage capacity. As one pathological example, consider a group of two devices, A, and B, each of which stores a single object, a on device A and b on device B, and has placement rules rule that specifies that A should instead hold b and B should instead hold a. If each device's storage capacity can hold only one of these objects at a time, and the group doesn't contain a third device, then neither device can fetch its preferred object due to lack of swap space. Arbitrarily large versions of this scenario can occur when devices have no free space. If devices can reserve enough free space for duplicating objects while moving, then these kinds of suboptimal stable configurations will not occur.

Chapter 4

Connectivity & Synchronization

Eyo faces two broad categories of challenges to fulfill its device-transparent storage API: device-to-device connectivity, and continuous synchronization.

In order to provide device-to-device connectivity, *Eyo* needs to be able to (1) identify the set of devices in user's personal group, (2) locate those devices as they move to different network locations, and (3) set up secure communication channels between the devices. *Eyo* uses an overlay network provided by an earlier project, UIA [16], to solve these challenges.

Several challenges remain towards the goal of providing continuous synchronization between devices. First, to approximate device transparency, *Eyo* systems should synchronize devices frequently. Frequent synchronization allows devices to check for updates whenever a local application writes new data, or when network connectivity may have been interrupted.

Second, to approximate device transparency when a collection of devices is disconnected from the network, *Eyo* should synchronize over any topology: any two devices that can communicate should be able to exchange objects. When disconnected, this feature allows local personal devices to access each others' objects transparently, and to show users the same set of objects from either device. Most existing synchronization tools require a central server to be able to provide consistency, and therefore don't support arbitrary topologies.

This section describes how *Eyo* synchronizes updates between devices, extending well-known techniques to take advantage of the separation between metadata and content to allow for frequent, efficient synchronization. *Eyo* identifies new updates using a single message (i.e., a constant amount of information is sufficient to to determine whether a collection is up to date, independent of the number of objects stored), and without a time-consuming local search that systems like Unison perform.

4.1 Device Identity and Communication

Eyo uses UIA [16] to manage groups of devices. UIA provides two basic functions to the applications using it (which is *Eyo* in this case): naming and routing. UIA allows users to construct a personal namespace, where the user can use any name to describe their devices, and then use those names from any of their devices. UIA then constructs an overlay network that allows applications on any of those devices to use those personal names to reach any other device regardless of whether it is on the same local network or at another location across the Internet. UIA allows users to also create links to friend's namespaces, in effect allowing each user to view a hierarchy of groups rooted at their own set of devices, and use user-relative names to reach the other devices. *Eyo* does not currently use UIA's multiple-user naming capabilities, but a possible extension (see Section 8.1.2) would.

When users get new devices, they add them to their device group by introducing the new device to an older one over some local network connection. After this introduction, each device sees a group with all member devices. *Eyo* uses this group information to authenticate metadata and content synchronization requests. UIA sends all inter-device communication over an SSL tunnel authenticated by the device's public keys, which are bound to the user-visible names during the introduction process.

UIA maintains active connections in the overlay network between each of the user's devices whenever possible, and informs *Eyo* when the set of reachable devices changes, or when devices join or leave the group. *Eyo* attempts to synchronize with each device in the group whenever UIA finds a new working path to it, either as it turns on and off, or moves between working networks.

UIA thus provides the communication properties *Eyo* requires: device identity, device location, and secured communication between those devices.

4.2 Synchronization Overview

Eyo needs to synchronize two classes of data between devices, metadata and content, and faces very different needs for these classes. Metadata is usually small, and updates must be passed as quickly as possible in order to provide the appearance of device-transparency. The goal of *Eyo*'s metadata synchronization protocol is to produce identical metadata collections after synchronizing two devices.

Content, on the other hand, can be comprised of many large objects which change infrequently. Content can take a long time to send over slow network links. Synchronizing content, unlike metadata, results in identical copies of individual objects, but not of the entire collections. The primary goal of synchronizing content is to move objects to their correct location to best match placement policies.

Given the different needs for these two classes of data, *Eyo* uses different protocols for each type.

4.3 Metadata Synchronization

The primary goal of *Eyo*'s metadata synchronization protocol is to produce identical copies of the entire metadata collection. This process must be efficient enough to run continuously:

when devices are on the same network and not disconnected, updates should flow immediately. If connectivity changes frequently, devices must quickly identify which changes to send to bring both devices up to date.

The main approach that *Eyo* takes to synchronize metadata is to poll for changes whenever connectivity changes, push notifications to reachable devices whenever a local application writes a new version of an object, and use immutable structures to pass updates over the network.

The primary challenge is, at each synchronization opportunity, to quickly identify the set of changed objects from among a much larger set of unchanged objects. More concretely, if two devices synchronize their metadata collections, and there were m new object versions created since the last synchronization time in a larger collection of M objects, the amount of work to identify those m new changes, as well as the network communication must both be bounded by $\mathrm{O}(m)$ rather than $\mathrm{O}(M)$. The metadata protocol described here takes $\mathrm{O}(n \times m)$ processing time and communication, where n is the number of devices in the user's group. For *Eyo*'s intended use cases, though, n will be a small constant. Several existing synchronization tools [4, 61] iterate over their data collections to identify changes at synchronization time, and consequently take longer than $\mathrm{O}(m)$ time to do so. *Eyo*'s metadata synchronization protocol identifies and organizes changes as they occur, rather than by iterating over the complete collection.

The split between content and metadata synchronization allows for a simple and efficient synchronization protocol. Figure 4-1 shows a diagram of the internal state of the metadata store for one device, showing the information each device needs to keep about which updates other devices know about. The following paragraphs introduce and define several of the internal structures *Eyo* uses to track metadata:

- A *generation* is a grouping of metadata updates into a permanent collection. Generations are named uniquely by the device that created them, along with an id field indicating how many generations that device has created. A generation includes complete metadata updates, but only the identifiers and new status bits for content updates. Generations are serialized for exchanging updates between devices, so all synchronization occurs at the granularity of individual generations. All devices that hold a copy of a given generation will have an identical copy.

- A *generation vector* is a vector denoting which generations a device has already received. These vectors are equivalent to traditional version vectors [26], but named differently to avoid confusion with the versions of individual objects. For a personal group with n devices, each *Eyo* device keeps a single n-element vector of $(device, id)$ tuples updated indicating the newest generation authored by that device it holds. This value is usually denoted as $generationVector$ in the following pseudocode. Additionally, each generation contains an attribute, usually noted as gv in the figures and pseudocode, that notes what the $generationVector$ of the authoring device was at the time it created that generation.

- The *archive generation* is a special generation used for garbage collecting fully-communicated generations. The archive groups together updates made by different

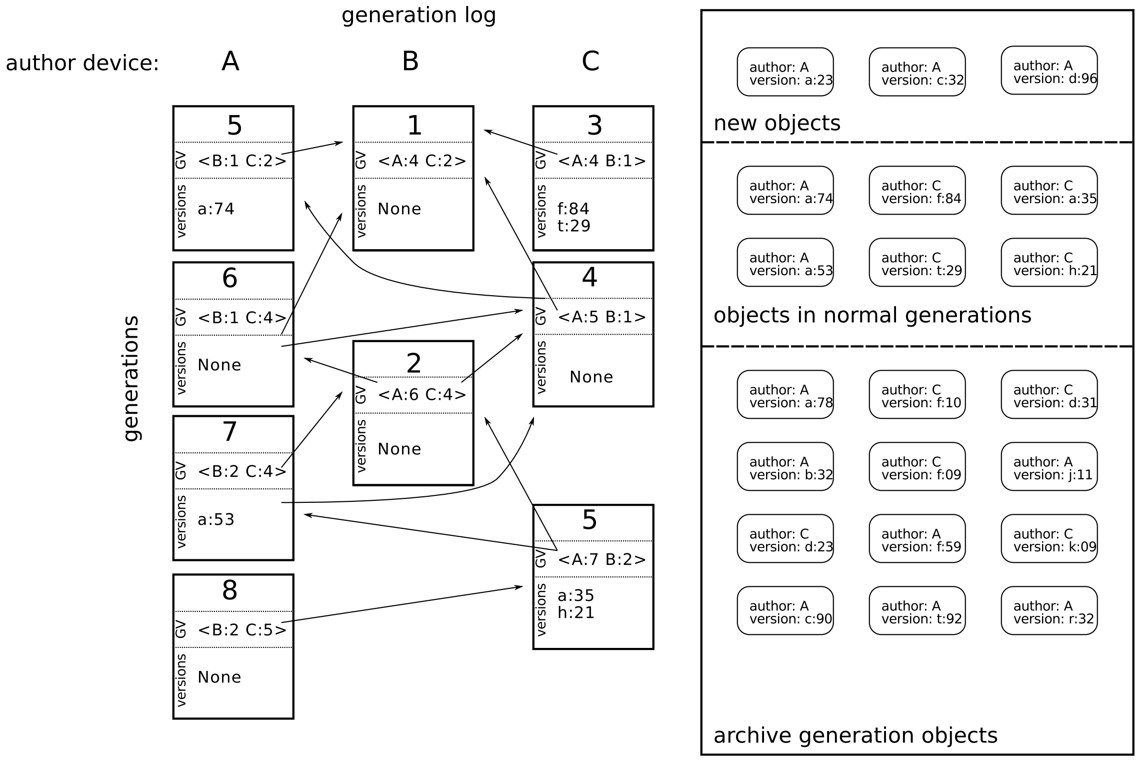

Figure 4-1: Metadata Synchronization: The state *Eyo* stores to track metadata synchronization on device A. Device B has not written any objects. Section 4.4 describes the archive generations. This figure omits data to track content locations (Section 4.6), object attributes, and predecessor relations between object versions.

devices and from different original generations, and does not retain those origins. The archive does have an associated generation vector, which tracks the newest generation from each device that has been subsumed into the archive. Section 4.4 discusses the uses of the archive.

- The *pending generation* is where devices store changes made by local applications before they are fixed into a permanent generation. The pending structure does not yet have a generation vector associated with it, as it is always converted to a permanent generation before sending its contents to other devices.

Figure 4-2 contains client-side pseudocode for requesting changes from other devices, and incorporating replies into the local metadata store. Each device regularly sends GETGENERATIONS requests to other reachable devices. When local applications modify or create new objects (via NEWVERSION calls), *Eyo* adds these uncommunicated changes to a *pending* structure, and attempts to contact reachable peers. With each of these requests, the client includes either it's local generation vector, or the next generation vector it will write if it has new changes pending. When a devices receives a reply, it incorporates the newly learned changes into it's local data store, updates it's generation vector accordingly,

```
function sendGetGenerationsRequest($peer$)
    $gv \leftarrow generationVector$
    if $pending \neq \emptyset \vee$ needAckGeneration() then        ▷ defined in Figure 4-3
        $gv[self]$ ++
    sendRPC($peer$, (GetGenerations, $gv$), handleSyncReply)
    return

function handleSyncReply($peer, res$)
    if $res.archive \neq \emptyset$ then
        $archive.c \leftarrow archive.c \cup res.archive.c$
        for all $(dev, id) \in res.archive.gv$ do
            $archive.gv[dev] \leftarrow \max(id, archive.gv)$
            if $dev \notin generationVector \vee id > generationVector[dev]$ then
                $generationVector[dev] \leftarrow id$
    for all $g \in$ sort($res.generations$) by generation vector do
        if $g.id = generationVector[g.author] + 1$ then
            if $g.c \neq \emptyset$ then
                $toPoll \leftarrow alldevices$
            $generations[g.author][g.id] \leftarrow g$
            $generationVector[g.author] \leftarrow g.id$
    //Notify applications of newly learned changes.
    //Apply newly changed placement rules against all objects.
    //Apply existing placement rules to newly learned objects.
    //Lazily check for generations that may be archived, and versions to prune.
```

Figure 4-2: Pseudocode to send metadata synchronization requests and handle replies.

```
function handleGetGenerations($gv, peer$)
    if $pending \neq \emptyset \vee$ needAckGeneration then
        $newgen \leftarrow$ new $Generation()$
        $generationVector[self]$ ++
        $newgen.author \leftarrow self$
        $newgen.id \leftarrow generationVector[self]$
        $newgen.gv \leftarrow generationVector$
        $newgen.c \leftarrow pending$
        $pending \leftarrow \emptyset$
        $generations[self][newgen.id] \leftarrow newgen$
    if $gv \not\preceq generationVector$ then
        //send a getGenerations request to $peer$ as soon as possible
        $toPoll \leftarrow toPoll \cup peer$
    $needarchive \leftarrow$ False
    for all $g \in archive.gv$ do
        if $g \notin gv \vee archive.gv[g] > gv[g]$ then
            $needarchive \leftarrow$ True
    $r \leftarrow$ new $SyncReply()$
    if $needarchive$ then
        $(r.archive, r.generations) \leftarrow archive, generations$
        return $r$
    $(r.archive, r.generations) \leftarrow (\emptyset, \emptyset)$
    for all $(d, id) \in gv$ do
        for all $g \in generations[d][id + 1 : -1]$ do
            $r.generations \leftarrow r.generations \cup g$
    return $r$

function needAckGeneration
    for all $g \in generations[\neg self]$ do
        if $g.gv \not\preceq generations[self][-1].gv \wedge g.c \neq \emptyset$ then
            return True
    return False
```

Figure 4-3: Pseudocode to handle incoming metadata synchronization requests.

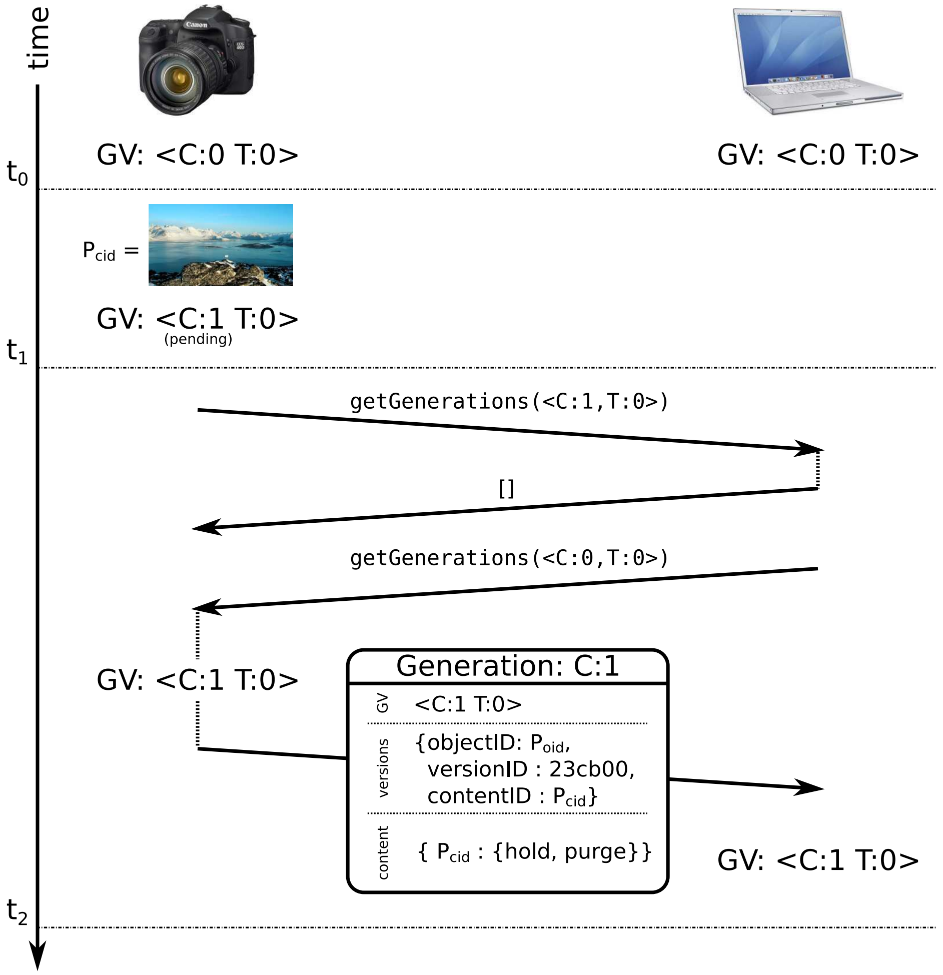

Figure 4-4: Metadata Synchronization: Messages sent between two devices for one new object

notifies applications about newly learned changes, and updates and applies placement rules to the newly learned changes.

When a device receives an incoming GETGENERATIONS requests, as described in Figure 4-3, it first fixes pending changes into a new generation if any such pending changes exist. It then identifies all the changes the other device lacks, and replies with those changes. If the request includes a generation vector with some component larger than the device handling the request knows about, it queues a GETGENERATIONS request in the reverse direction, either immediately, or when next reachable if the request fails. In cases where no new devices have joined the group, the reply will not include a complete archive, so the message size, and time to identify changed objects, depends only on changes authored since they last communicated.

Figure 4-4 presents an example use of these structures between two devices: a camera C that temporarily stores photos when the user takes a picture, and a target device T that archives the user's photos. To match the user's workflow, the target device has a placement rule matching photos the camera creates; the camera has no such rule and thus tries to push its photos to other devices.

Initially, at t_0 in Figure 4-4, both devices hold no objects and agree on an initial generation vector `<C:0,T:0>`. When the user takes a picture P at time t_1, the camera adds the contents of the picture to its local content store with content identifier P_{cid}, creates a new *Eyo* object with object id P_{oid}, and adds P_{oid} to the metadata store. *Eyo* adds each of these updates to the next generation under construction (noted $pending$ in the figure).

At time t_2, C holds uncommunicated updates, so it sends GETGENERATIONS() requests to all reachable devices with the single argument `<C:1,T:0>`: C's generation vector with the `C` element incremented. T compares the incoming generation vector to its own and determines that it has no updates for C and replies with an empty generation list. However, since C's generation vector was larger than its own, T now knows that C has updates it has not seen, so T immediately makes its own GETGENERATIONS() call in the opposite direction with argument `<C:0,T:0>` since T has no uncommunicated updates of its own. Upon receiving the incoming request from T, C increments its generation vector and permanently fixes all uncommunicated updates into generation `C:1`. C then replies with generation `C:1` and its newly-updated generation vector to T. The camera makes no further call back to T, as T's generation vector was not larger than its own. Both devices now contain identical metadata.

Although for the sake of clarity this example only included two devices and did not include a large existing data collection, it does illustrate the protocol's scaling properties. For a group containing n devices, the *Eyo* metadata synchronization protocol sends only a single generation vector of length n to summarize the set of updates it knows about in a GETGENERATIONS() request. Upon receiving an incoming vector, an *Eyo* device needs only a simple lookup to identify what generations to send back, rather than an expensive search. This lookup requires one indexed read into the generation log per element in the incoming generation vector. This low cost means that devices can afford to push notifications instantaneously, and poll others whenever network connectivity changes.

4.4 History and Version Truncation

Eyo must have a way to prune version histories. It must identify which past changes are no longer needed and reclaim space taken up by those updates. This process involves two separate steps: determining when generation objects have been seen by all devices in a group and combining the contents of those generation objects into a single archive, and truncating the version history of individual objects.

Eyo learns that each other device has seen a given generation G by checking that every other device has written some other generation G' that includes G in its generation vector, meaning that G' *covers* G. At this point, no other existing device can correctly send a synchronization request that would include G in the reply, so it can remove G from its generation log. Once a device learns that all other devices have received a given generation

```
function ARCHIVEGENERATIONS
    for all $(d, i) \in generationVector$ do
        //$i$ is the newest generation written by device $d$ that we've received.
        $minid \leftarrow i$
        for all $g \in generationVector, g \neq d$ do
            $minid \leftarrow \min(minid, generations[g][-1].gv[d])$
        for $j \leftarrow [archive.gv[d] + 1, \min(minid, i - 1)]$ do
            //All other devices have seen device $d$'s $j$th generation
            $archive.c \leftarrow archive.c \cup generations[d][j].c$
            $archive.gv[d] \leftarrow j$
            delete $generations[d][j]$
```

Figure 4-5: Pseudocode to archive generations

G, it may lazily move G's contents into its *archive generation*, as shown in pseudocode in Figure 4-5. *Eyo* preserves at least one non-archived generation for each device, even if that generation is fully known to all other devices. This ensures that *Eyo* knows the latest generation each other device has reported as received (used by, e.g., ARCHIVEGENERATIONS).

Object versions in the archive generation are known by all the user's devices, and are thus candidates for pruning, which is the second phase of history truncation. Version pruning proceeds as described in section 3.2. To enable garbage collection as soon as possible, devices acknowledge receipt of metadata updates by creating an acknowledgment generation: a generation with no contents except for the newly learned generation vector, which is not shown in the example in figure 4-4, but is detailed in figure 4-3. Devices do not acknowledge receipt of these otherwise-empty generations. Devices do not need to publish acknowledgment generations to achieve device-transparency: their only purpose is to reclaim space sooner.

Eyo nominates versions for truncation by searching for common ancestors back from each head version. Figure 4-6 contains the details. These common ancestors are articulation points (also known as cut vertices) in the version graph for a single object: a single version that, if deleted, splits the version graph into two connected components, one descended from the common ancestor, and one from which preceded the common ancestor. The search follows the traditional depth-first-search method of identifying articulation points in a graph [57]. *Eyo* repeats this search considering the subgraph that each head version derived from, and only includes versions that qualify as common ancestors in all of the subgraphs. These articulation points represent the oldest version of a single object that applications might need in order to resolve conflicts. Any versions older than these are candidates for pruning, if the device can be sure that no other device will write some new version based on an older version than the common ancestor.

Figure 4-7 details the solution to this requirement, which is that pruning versions may only proceed if the common ancestor is in the archive. In this case, no later version can conflict with that ancestor, since the other device knew about the common ancestor: any newer version must derive from that one or a younger descendant, as *Eyo* does not permit

```
function commonAncestors($(objectID)$)
    $articulationPoints \leftarrow []$
    $g \leftarrow$ versionGraph$(objectID)$
    for all $hv \in g.headversions$ do
        $articulationPoints[hv] \leftarrow \emptyset$
        $\forall n \in g, n.visited \leftarrow False$
        $t \leftarrow 1, arrive \leftarrow [], low \leftarrow [], pred \leftarrow []$
        $hv.visited \leftarrow True$
        $pred[hv] \leftarrow \emptyset$
        $arrive[hv] \leftarrow low[hv] \leftarrow 0$
        $stack \leftarrow [hv]$
        while $stack \neq \emptyset$ do
            $v \leftarrow stack.top()$
            $adj \leftarrow \{v.parent\} \cup \{v.children$ reachable via $parent$ pointers from $hv\}$
            $found \leftarrow False$
            for all $n \in adj$ do
                if $\neg n.visited$ then
                    $found \leftarrow True$
                    $n.visited \leftarrow True$
                    $arrive[n] \leftarrow low[n] \leftarrow t$
                    $pred[n] \leftarrow v$
                    $stack.push(n)$
                    $t{+}{+}$
            if $\neg found$ then
                $stack.pop()$
                for all $n \in adj$ do
                    if $n \neq pred[v] \wedge arrive[n] < arrive[v]$ then
                        $low[v] \leftarrow \min(low[v], arrive[n])$
                    else if $v = pred[n]$ then
                        $low[v] \leftarrow \min(low[v], low[n])$
                        if $low[n] \geq arrive[v] \wedge v \neq hv$ then
                            $articulationPoints[hv].add(v)$
    $r \leftarrow articulationPoints[g.headversions[0]]$
    for all $hv \in g.headversions[1:-1]$ do
        $r \leftarrow r \cap articulationPoints[hv]$
    return $r$
```

Figure 4-6: Pseudocode to identify common ancestors of head versions, where some version still exists which is older than the common ancestor.

```
function PRUNEOBJECT($objectID$)
    $todel \leftarrow \emptyset$
    for all $ca \in$ COMMONANCESTORS$(objectID)$ do
        if $ca \in archive$ then
            for all $p \in ca.parents$ do
                $todel.push(p)$
    while $todel \neq \emptyset$ do
        $d \leftarrow todel.pop()$
        if $d$ not already deleted then
            for all $p \in d.parents$ do
                $todel.push(p)$
            delete $p$ from $archive$
```

Figure 4-7: Pseudocode to prune object version graphs of all versions not needed for conflict resolution.

applications to intentionally create conflicts, meaning that all newly written versions must derive from a currently-known head version. *Eyo* lazily searches for such candidate metadata versions to delete, but does not normally carry out deletions until pressed for storage space.

Figure 4-1 shows an example of the state stored on one device, A, with two other devices. In this example, generations `A:1` through `A:4` and `C:1` through `C:2` were universally known. Their contents were moved to the archive generation, and hence they no longer appear in the generation log. Device B in this example has not written any objects, but has written acknowledgments for other generations from A and C. At the time of this example, four generations are eligible to be truncated, as they are universally known by all three devices: `A:5`, `A:6`, `C:3`, and `C:4`. *Eyo* can then move the following three versions to the generation archive: `a:74`, `f:84`, and `t:29`. *Eyo* can then check whether any of these versions were unique ancestors, and if so, could prune older versions of those objects.

Devices may delete object contents when no local application is currently using that content object, and one of the following cases applies: (1) no content identifier in the metadata lists that content identifier, meaning that the associated object versions were pruned or deleted elsewhere without contention, (2) a local application issues a directive to remove all versions of an object permanently, or (3) the device successfully passed responsibility for the object, as in the camera example. As in the case of metadata, *Eyo* detects and notes content objects eligible for deletion, but normally does not reclaim space until under pressure to reuse it.

These truncations mechanisms ensure that when devices communicate updates freely, devices only need to keep a very shallow and linear version history for each object, and similarly only need to keep a few content versions for each object.

4.5 Adding and Removing Devices

When a user adds a new device to their personal group, and that new device first synchronizes with an existing device, *Eyo* sees a GETGENERATIONS() request with missing elements in the incoming generation vector. Existing devices reply with a complete copy of all generations plus the archive generation. This copy cannot easily be broken down into smaller units, as the archive generation differs between devices due to pruning. Users expect new devices to require some setup, however, so this one-time step should not be an undue burden. Single devices do not normally contain any archive, and so do not impose any burden on the existing devices in the group.

This procedure is not limited to adding a single device to an existing group. Two existing device groups can merge, though in this case each device in the group needs to fetch a complete archive when it first learns about the merge. Merging two existing groups of devices should be rare in practice, and so cascading archive exchanges should as well. If such use were common, it could be handled either by deferring archiving generations until after the merge, or by a more extensive change whereby devices keep multiple separate archives partitioned by the original group that created it.

Users remove devices from an *Eyo* group by deleting them from the underlying UIA group. Unless the user explicitly resets an expelled device entirely, it does not then delete any objects or content, and behaves thereafter as group with only one device. The surviving group also does not delete objects the expelled device created, but neither queries the expelled device for new updates nor considers the expelled device to determine whether all devices know about a given generation. Removing an inactive or uncommunicative device from an *Eyo* group allows the surviving devices to make progress truncating history. An expelled device can rejoin the group later, as long as the device uses the same underlying UIA permanent device identifier. This re-introduction maintains all old history by exchanging complete archive generations, just as when adding a new device.

4.6 Content Synchronization

The challenges in moving content to its correct location on multiple devices are (1) determining which objects a particular device should hold, (2) locating a source for each missing data object on some other device, and (3) ensuring that no objects are lost in the process of moving them between devices.

Eyo uses placement rules to solve the first of these challenges, as described in section 3.5. Each device keeps a sorted list of content objects to fetch, and updates this list when it learns about new object versions, or when changes to placement rules affect the placement of many objects.

Eyo uses the global distribution of metadata through a user's personal group to track the locations of content objects. In addition to the version information, devices publish notifications about which content object they hold (as shown in Figure 4-4). Since all devices learn about all metadata updates, all devices thus learn which devices should hold

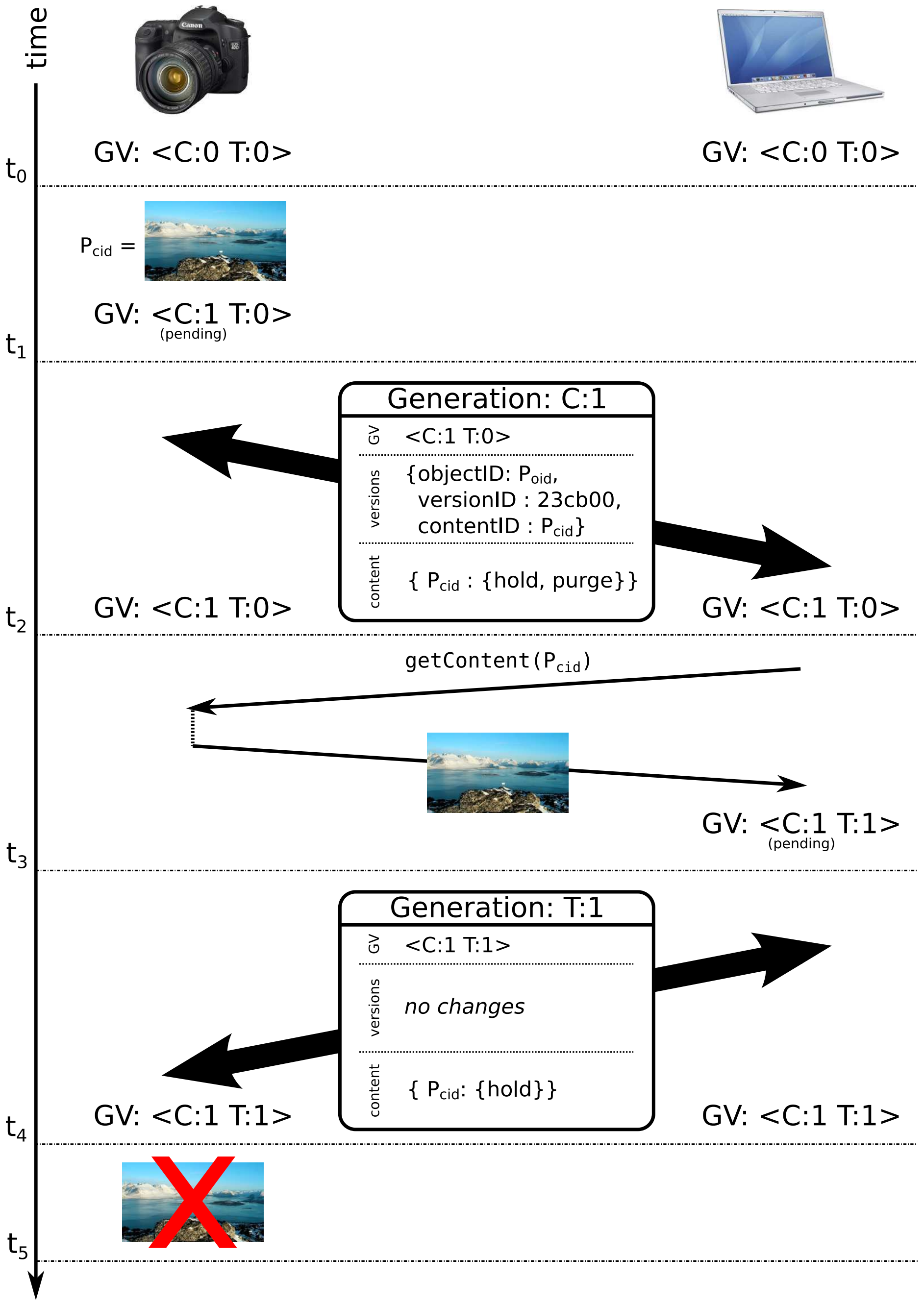

Figure 4-8: Content Synchronization. The thick double arrows represent a metadata sync from Figure 4-4.

content as part of the same process. When *Eyo* learns that another device is reachable, it can look at the list of content to fetch, and determine which objects to request from the reachable device.

To ensure that content objects are not deleted prematurely, *Eyo* employs a form of custodial transfer [12] whereby devices promise to hold copies of given objects until they can pass that responsibility on to some other device. When a device adds content to its local data store as a result of a matching placement rule, it signals its intent to hold the object via a flag in the metadata.

If placement rules later change, or the device learns of newer higher-priority data that it would prefer to hold, it signals a request to delete the object as a metadata update. At this point, however, the promise to hold still applies to the original data holder. Its responsibility continues to apply until some other device authors a generation that falls strictly later than the one which removed the promise, and includes a new or existing promise to hold that same data item. If two different devices each holding the last copy of an object announce their desire to remove that item concurrently, so that the generations that contain these modifications cannot be totally ordered, then neither device will be able to delete the object, as neither will be able to identify another device that has accepted responsibility for storing the object.

This protocol ensures that, as long as no devices are lost or stolen, each non-deleted item will have at least one live replica in the device collection. This property does not depend on the existence or correctness of placement rules: applications may delete or modify placement rules without needing to ensure that some other rule continues to apply to that object.

Figure 4-8 shows an example content sync that continues where the metadata sync of Figure 4-4 leaves off. When the target device receives the camera's metadata update at time t_2, it evaluates its own placement rules, and adds P_{cid} to its list of content it desires. The generation `C:1` that T received included P_{cid}, so T knows that C has a copy (the *hold* bit is set) of P_{cid} that it wants to delete (the *purge* bit). At t_3, T sends a getContent(P_{cid}) request to C, which replies with the new photo. Because T intends to keep P, it adds a *hold* bit to P_{cid} in the next generation it publishes, `T:1`.

At t_4, the devices synchronize again and the camera and target again contain identical state. But the camera now knows an important fact: the target (as of last contact) contained a copy of P, knew that C did not promise to keep P via the *purge* bit, and hence the target has accepted responsibility (*hold* but not *purge*) for storing P. Thus, at t_5, the camera can safely delete P, placing the system in a stable state matching the user's preferences.

This content synchronization mechanism allows content to safely move between devices, while requiring each device to implement only a simple fetch operation to move objects.

Chapter 5

Implementation

Eyo's prototype implementation consists of a per-user daemon, *eeyore*, that runs on each participating device and handles all external communication, and a client library that implements the *Eyo* storage API.

5.1 *eeyore*

eeyore, the per-device server process that implements the *Eyo* API and protocols as described in the previous chapters, is written in Python, and runs on Linux and Mac OSX. *eeyore* keeps open connections (via UIA) to each peer device whenever possible, and otherwise attempts to reestablish connections when UIA informs *Eyo* that new devices are reachable. *eeyore* uses SQLite [54] to hold the device's metadata store, and to implement *Eyo* queries. The daemon uses separate files in the device's local filesystem to store content, though it does not expose the location of those files to applications. *eeyore* uses XML-RPC for serializing and calling remote procedures to fetch metadata updates. *eeyore* uses separate HTTP channels to request content objects. This distinction ensures that large content fetches do not block further metadata updates. Larger content objects can be fetched as a sequence of smaller blocks, which should permit swarming transfers as in DOT [59] or BitTorrent [7], although *eeyore* does not yet implement swarming transfers.

5.2 Application Client Libraries

Two client libraries accompany *eeyore*. A Python module, and a C library each provide the *Eyo* API for applications, though the two versions differ on many of the details of the API. For example, the Python module provides a high-level object API that uses the `file` object interface for accessing content objects, whereas the C library provides applications with standard file descriptors for reading content. The C library provides many more low-level functions for manipulating metadata tags. The Python module implements metadata

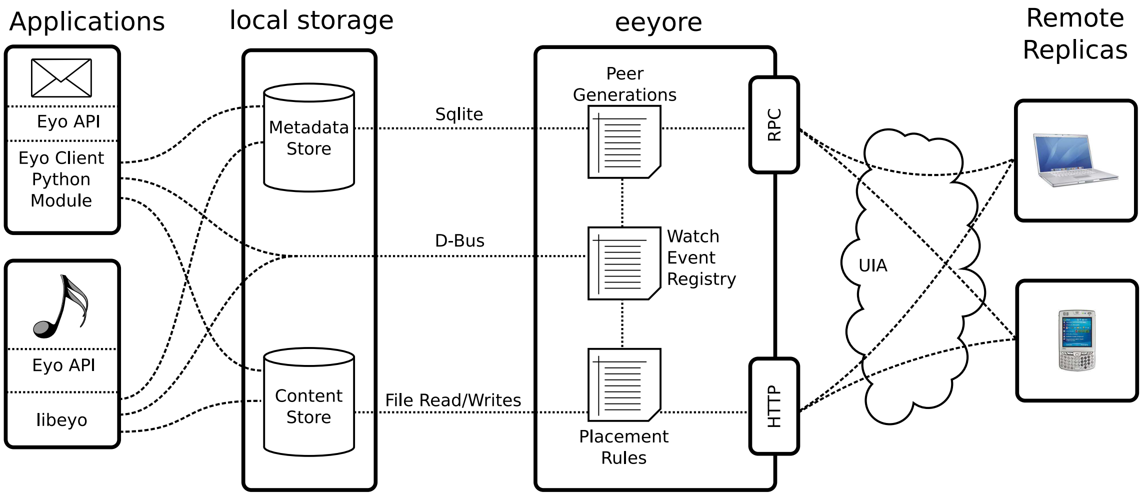

Figure 5-1: Internal *eeyore* components.

collections via dictionaries and so does not need any *Eyo*-specific operations to manipulate such collections.

The two client libraries present *Eyo* queries to applications in different forms, but both represent queries as lists of operators and values. For example, when using the C *Eyo* library, an application could construct a simple query as follows, which matches objects that both have a the metadata attribute “filename”, and where the associated value is the string “foo.jpg”:

```
eyo_build_query(&q, EYO_OP_EQUAL, EYO_KEY, "filename",
                EYO_VAL_STRING, "foo.jpg", EYO_OP_NULL);
```

eeyore does not provide any integrity guarantees to protect local metadata and content state from application bugs which could incorrectly modify or delete objects. Although *eeyore* should be robust toward applications calling *Eyo* API functions incorrectly, it does not protect against applications opening and modifying *eeyore*’s data structures directly without using the API methods, as the data stores are accessible on disk to the user’s applications.

The *Eyo* client libraries are both optimized for read performance, on the assumption that applications will frequently use LOOKUP() queries to identify data objects and view groups of objects. Responses to these queries will populate user interfaces, and so must return quickly. Write performance, on the other hand, is not as important. All local writes eventually result in writing most of the same data over network links, so the network eventually limits write performance rather than local storage. Client modules read from *eeyore*’s metadata and content stores directly, rather than requiring inter-process communication channels for most read accesses. SQLite does not provide a notification mechanism for applications, so *Eyo* uses D-Bus [10] to send watch notifications to client applications.

eeyore cannot depend on client-side libraries to send watch notifications directly to interested applications. The API requires that interested applications receive exactly one watch notification for each matching update notifications, so authoring a new version and sending the associated watch notification must be an atomic operation. *eeyore* cannot send a notification as part of a database transaction to record a write, as the client application could fail between the metadata write and the D-Bus method call. Instead, *Eyo* client libraries append their updates to a write-only table in the metadata store and then send a notification to *eeyore*. *eeyore* validates the write, and copies metadata to the correct destination, sends related watch notifications, and then replies to the client application to indicate a successful write. This process adds latency to application writes, but these extra delays are not present in application reads and queries.

5.3 Limitations

SQLite does not implement any of the fine-grained locking schemes commonly found in stand-along databases. Instead, client applications that write the metadata database must lock the entire database for each transaction. The *Eyo* prototype inherits this limitation: many applications can read data concurrently, but any write operation blocks all other application reads, even for unrelated metadata objects.

None of the *Eyo* design, however, depends directly on SQLite. The client libraries do not expose any of the internal table structure to applications. For example, while SQLite might be appropriate for clients with very limited hardware resources, an alternate implementation could present the same application API but use a different internal database to improve performance. Because the *Eyo* query interface is quite limited and only uses a small portion of the SQL query language, one of the NoSQL systems (e.g., MongoDB [35] or CouchDB [8]) could serve as a viable alternative to SQLite.

Chapter 6

Evaluation

This thesis proposes a new storage API for applications, and APIs are notoriously difficult to evaluate. We would like to establish that the *Eyo* API provides substantive benefits to real applications and that the costs to developers and end users of moving to this new API are worthwhile. In addition to the API's suitability, we also evaluate *Eyo*'s design and implementation, both in terms of the bandwidth and space overheads of device-transparent storage, and the performance of *Eyo*'s continuous synchronization protocols.

We explore these issues by examining the following questions:

- Is *Eyo*'s storage model useful for applications and users?
- Is it necessary to involve applications in automatic conflict resolution?
- Do *Eyo*'s design choices, such as splitting metadata from content, unduly burden devices' storage capacity and network bandwidth?
- Are *Eyo*'s continuous synchronization protocols efficient in terms of the bandwidth consumed, and the delay needed to propagate updates?

The following sections describe methods for answering these questions followed by results for each investigation.

6.1 Method

We employ three methods to evaluate *Eyo*: (1) adapting existing applications to use *Eyo*'s storage API instead of their native file-based storage to examine the modification difficulty and describe the new features of the modified versions, (2) storing example personal data collections to examine storage costs, and (3) measuring *Eyo*'s synchronization protocol bandwidth and delays to compare against existing synchronization tools.

The purpose for adapting existing applications to use *Eyo* as their primary storage interface is to examine whether *Eyo*'s API is a good match for those uses, describe how those applications use the *Eyo* API, and how difficult those changes were. While it would certainly be possible to design entirely new applications around the *Eyo* API, those applications might turn out to use different internal structures than existing applications.

We evaluated *Eyo*'s storage and bandwidth costs using three data collections storage in *Eyo*: email, music, and photos. These collections served as a basis for a synthetic workload used to measure bandwidth costs and storage costs due to disconnected devices.

We compared *Eyo*'s synchronization protocols to two existing synchronization tools. While neither of our comparisons aim to provide device-transparent access to a data collection, the comparison does verify that the performance of *Eyo*'s metadata synchronization protocol is independent of the number of objects in the collection.

The following sections describe the results of these efforts and relates them back to the earlier questions: Is *Eyo* useful, is its model appropriate, and are its costs reasonable?

6.2 Applications Overview

This section briefly describes the existing applications that we modified. We chose applications with a wide range of types of interactions between users and their data. We focus on two areas: (1) audio and photo applications, where users do not currently see a device-transparent data collection, and (2) email, where users already expect a device-transparent view, but typically only get one today when successfully networked to a central server. We modified two media players, Rhythmbox and Quod Libet, the Rawstudio photo manager, and the gPodder podcast manager, to use *Eyo* instead of the local filesystem. We built an IMAP-to-*Eyo* gateway to enable arbitrary email clients to access messages stored in *Eyo*.

The descriptions in this section refer to the original, unmodified versions of each application. All of these applications are open source; several have popular commercial alternatives that inspired our choices, but we did not investigate those close-sourced applications.

Rawstudio Rawstudio is a photo editor, written mostly in C and C++, meant for organizing and process RAW format digital photographs. Users import these raw files, which usually consists the exact bits recorded by a camera's sensor along with image settings (e.g., white balance, color space, contrast) that the camera normally uses internally to produce a compressed jpeg-format file. In Rawstudio, users can losslessly change the development settings and apply additional effects such as exposure compensation, to produce JPEG-format versions of the unmodified originals. Rawstudio keeps a central database of image metadata, allows users to add short textual tags to individual or groups of images, and then locate those images either by tag or by location in the local filesystem.

Rhythmbox Rhythmbox is a music manager and player, written in C, built with several GNOME libraries and frameworks. It permits users to add music (usually in MP3 or similar form) to a logical library, where Rhythmbox keeps a central database of song metadata, and keeps each song in a separate file on disk. Users can view and play collections though

Size (lines)	Rawstudio	Rhythmbox	QuodLibet	gPodder	Email
original project size	59,767	102,000	16,089	8,168	3,476
affected module size	6,426	9,467	428	426	312
lines added	1,851	2,102	76	295	778
lines removed	1,596	14	2	2	N/A

Table 6.1: Source lines of code [51] comparisons of applications adapted to *Eyo*. In each case, only a small, self-contained module needed to be modified. The project sizes do not include libraries. For email, the “original project size” only includes Twisted’s IMAP module and server example code, and ‘lines added’ includes all of our newly written code. The ‘lines added’ and ‘lines removed’ counts are from diffstat, and so do not match total line definitions exactly.

several types of groups, such as playlists, album, or user-supplied searches that Rhythmbox carries out against its central metadata database.

QuodLibet QuodLibet is also a music player and manager. It is written in Python, and consequently has a significantly smaller codebase than Rhythmbox. QuodLibet keeps a centralized database of song metadata, but it allows users to add arbitrary tags as metadata rather than relying on a predefined schema.

gPodder gPodder is a simple podcast manager, written in Python. It allows users to subscribe to RSS podcast feeds. It periodically checks those feeds for updates, and when it finds that new episodes are available, downloads and caches those files locally on disk until the user listens to and deletes those objects.

IMAP server Instead of modifying an existing email application, we built an IMAP server so that existing IMAP client applications could access email messages stored in *Eyo*. Our server differs from traditional folder-based IMAP servers in that our server permits messages to appear in multiple folders at the same time, much as GMail permits tagging messages with multiple tags.

6.3 Results: *Eyo* API Experiences

Adapting existing applications to use *Eyo* is straightforward. Table 6.1 summarizes the changes made to each application. In each case, we only needed to modify a small portion of each application, indicating that adopting the *Eyo* API does not require cascading changes through the entire application. In all cases the required changes were limited to modules composing less than 11% of the total project size. Rhythmbox required more changes than the other applications primarily because we added support for storing and accessing data from *Eyo* but did not remove the ability to use the existing filesystem data stores. In the other applications we entirely replaced the existing storage uses with *Eyo*. Our version of gPodder is slightly smaller, as we omitted functions to create and manage a metadata database, and required no additional code to handle multiple versions of objects beyond one-line merge calls.

To show that *Eyo* is a good fit for applications, we consider the following points in addition to simply looking at the magnitude of code changes.

***Eyo* provides device-transparency.** The simple changes transformed the existing media applications from stand-alone applications with no concept of sharing between devices into a distributed system that presents the same collection over multiple devices. The changes do not require any user interface modifications to support device transparency; users simply see a complete set of application objects rather than the local subset. However, some user interface changes are necessary to expose placement rules and conflict resolution to the user. It is no accident that these new features needed few changes, and indicates that *Eyo* is a good match for application-level objects. While the new features for the email system were less dramatic—clients automatically share new messages and status changes without a server—these new features required no changes at all in the user-facing email clients, only in the IMAP server.

Metadata is useful alone even without the related content. The modified media applications can show the user's entire music collection. Even when content isn't present, users can search for items, modify playlists, see where objects do reside, and, if reachable, fetch remote objects transparently. In Rawstudio, users can search for photos by tag through the entire collection even when the content is missing, organize those photos into new groups, and show which devices hold the associated content. Surprisingly few changes were necessary to support objects with missing content. Although *Eyo* does provide a metadata field indicating whether the associated content is available locally, the applications generally functioned correctly even without additional logic to examine this field, e.g., by continuing on to the next item in a playlist, for two reasons. (1) Applications need to fail gracefully when given files they cannot interpret, such as unsupported image or music file types, and (2) applications that keep a central metadata database may hold pointers to files in traditional network filesystems that become unreachable. Handling missing content in *Eyo* triggers these same code paths.

Applications automatically compose concurrent updates. Concurrent updates occur as a part of normal application operations, for example every time users play the same song or read the same mail message from disconnected devices. These actions result in multiple head versions of these objects when connectivity resumes. In most cases, the version history *Eyo* provides permits applications to resolve concurrent changes simply by applying the union of all user changes; the *Eyo* client library makes this straightforward. A few cases require application-specific involvement, e.g. the applications that track playcounts use a custom merge procedure to sum up the count increments.

Application	Type	User-Visible Conflicts Possible?	Why?
IMAP	Email Gateway	No	Boolean metadata flag changes only
gPodder	Podcast Manager	No	User cannot edit metadata directly
Rhythmbox	Media Player	Yes	Edit Song title directly
QuodLibet	Media Player	Yes	Edit Song title directly
Rawstudio	Photo Editor	Yes	Edit settings: contrast, exposure . . .

Table 6.2: Description of whether applications can handle all version conflicts internally, or must show the presence of multiple versions as a result of some concurrent events, along with an explanation or example of why that result holds for each application.

The experience with these applications led us to conclude that applications must be involved in automatic conflict resolution because the correct policy depends on the application and the metadata item. For example, different playcount histories could equally validly be resolved by taking the maximum count instead of summing the increments. Only the application designer has sufficient information to choose the appropriate policy for each metadata item.

As another example, in our IMAP application if one device updates the “unread” message flag and another device updates the “replied” flag, then *Eyo* will flag a conflict to the application. However, the IMAP gateway knows that these updates are composable and resolves the conflict without user intervention.

An alternate type of concurrent update arises when importing external data objects into *Eyo*. Our gPodder version, for example, downloads podcasts and stores episode metadata and content in *Eyo*. It includes the RSS feed’s <`GUID`> element in the hints to `create()` to ensure that multiple clients that independently download the same episode while disconnected automatically merge the objects once connectivity is restored. In addition to the client-to-*Eyo* IMAP server, we also implemented an *Eyo*-to-server gateway which, acting as an IMAP client, pulls new messages from a user’s external IMAP inbox into the *Eyo* store. Like gPodder, it uses create hints based on message ID’s to avoid inserting duplicate messages.

Users rarely encounter version histories. Applications use version histories internally to merge divergent version histories back into a single head version, but in most cases users are never aware when concurrent updates occur, as the applications perform these operations automatically. A few cases however, do result in end-user visible effects. Table 6.2 summarizes the results.

Because Rhythmbox and Quod Libet allow users to modify metadata directly in the UI, it is possible for users to make conflicting changes requiring manual intervention on two devices. These kinds of user-visible conflicts only arise due to direct, concurrent changes. As outlined above, normal operations, such as playing a song, or editing a playlist, never result in user-visible conflicts.

Rawstudio does permit user-visible conflicts, but this does not normally cause a problem. Rawstudio allows users to save a set of several versions of the ‘development settings’ for each photo. If a user concurrently changes the settings on two devices, Rawstudio can show both branches of this conflicted object as a different set of settings. The user-supplied image tags cannot cause conflicts, and all of the other metadata fields are read-only to the user.

The other applications, gPodder and email, prohibit user-visible conflicts entirely, as users don’t edit individual metadata tags directly. These two applications *never* show multiple versions to end users, even though the underlying system-maintained version histories exhibit forks and merges. The ability to hide these events demonstrates the usefulness of keeping system-maintained version histories so that applications face no ambiguity about the correct actions to take.

Summary. In summary, we found that modifying applications to use the *Eyo* storage model was not difficult. In most cases, applications use objects in the same patterns as they did before, except that end users experience a coherent collection rather than a disjoint set of objects on different devices. *Eyo* provides applications with the necessary information to hide many concurrent changes from users.

6.4 Results: Metadata Storage Costs

To determine the expected size of metadata stores in *Eyo*, we inserted three modest personal data sets into *Eyo*: the email, music, and photo collections a single user gathered over the past decade. We included a collection of email messages as a worst-case test; this collection includes a large number of very small objects, so the metadata overhead will be much larger than for other data types. Table 6.3 shows the resulting metadata store sizes. To extract metadata, we parsed email messages to extract useful headers, imported the user’s media attribute database, and used `exiftags` or `dcraw` to extract attributes from photos. This example considers only the static metadata store size—*Eyo* stores a single version of each object —the next sections examine the costs of multiple versions.

The table shows that for each of the three data types, *Eyo*’s metadata store size is approximately 3 times as large as the object attributes alone. The overhead comes from database indexes and implementation-specific data structures.

The most important feature this data set illustrates is that the size of the metadata store is roughly (within a small constant factor) dependent only on the number of individual objects, but not the content type, and not the size of content objects. The number of objects,

Email	
number of messages	724230
total content size	4.3 GB
median message size	4188 bytes
native metadata size	169.3 MB
Eyo metadata store size	529.6 MB
metadata/content overhead	12%
metadata store size per message	766 bytes
Music	
number of tracks	5278
number of playlists	21
total content size	26.0 GB
mean track size	5.1 MB
native metadata size	2.6 MB
Eyo metadata store size	5.8 MB
metadata/content overhead	0.02%
metadata store size per object	1153 bytes
Photos	
number of JPEG/RAW objects	61740/10640
total number of objects	72380
JPEG/RAW content size	32.7/90.1 GB
total content size	122.8 GB
native metadata size	22.6 MB
Eyo metadata store size	52.9 MB
metadata/content overhead	0.04%
metadata store size per object	767 bytes

Table 6.3: Metadata store sizes for example datasets. The native metadata size is the size of the attribute key/value pairs before storing in *Eyo*. The metadata store size is the on-disk size after adding all objects.

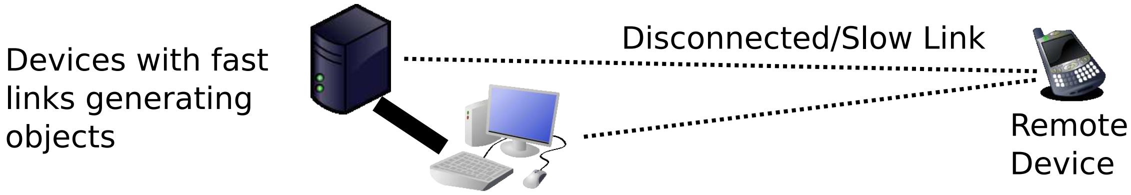

Figure 6-1: Topology for the scenarios in sections 6.5 and 6.6

along with the amount of metadata per object, thus provides a lower bound on the storage capacity of each device.

The total metadata size in this example (less than 600 MB) is reasonable for today's current portable devices, but the total content size (153 GB) would not fit on a laptop only a few years old nor on many current portable devices. Adding video would only increase the disparity between metadata and content store sizes, and reduce relative amount of overhead *Eyo* devotes to storing object metadata.

6.5 Bandwidth Costs

In addition to storage costs, the metadata-everywhere model places bandwidth costs on other devices in the system, even when those devices do not store the newly created objects.

Figure 6-1 shows the simplest possible network topology to examine bandwidth and storage costs for disinterested devices that lack any placement rule matching newly changed objects (this section), and for absent devices (next section).

In this scenario, a pair of object-generating devices create new objects at exponentially distributed times at a variable average rate, attaching four kilobytes of attributes to each new object (larger than the median email message headers considered in section 6.4). The disinterested device ("remote" in the topology) has only a slow link to the other replicas, and we measure the synchronization bandwidth passed over this slow link, averaged over a month of simulated time. The disinterested device does not fetch any of the associated content objects, hence all of the bandwidth in this case is metadata and protocol overhead.

Figure 6-2 shows that the bandwidth consumed over the slow link, as expected, increases linearly with the update rate. If the slow link had a usable capacity of 56 kbps, and new updates arrive once per minute on average, the disinterested device must spend approximately 1.5% of total time connected to the network in order to stay current with metadata updates. This low overhead is expected intuitively: small portable devices routinely fetch all new email messages over slow links, so the metadata bandwidth for comparable content will be similar.

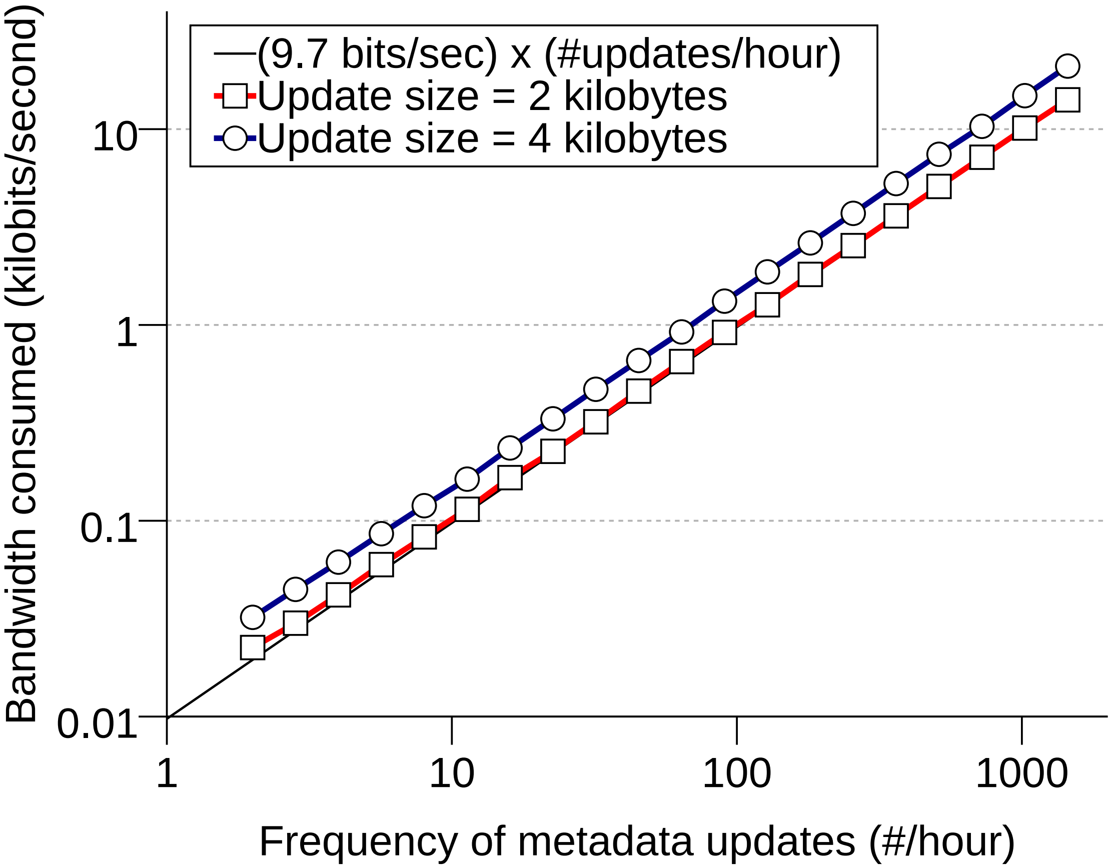

Figure 6-2: Average connection bandwidth required to continuously synchronize metadata changes.

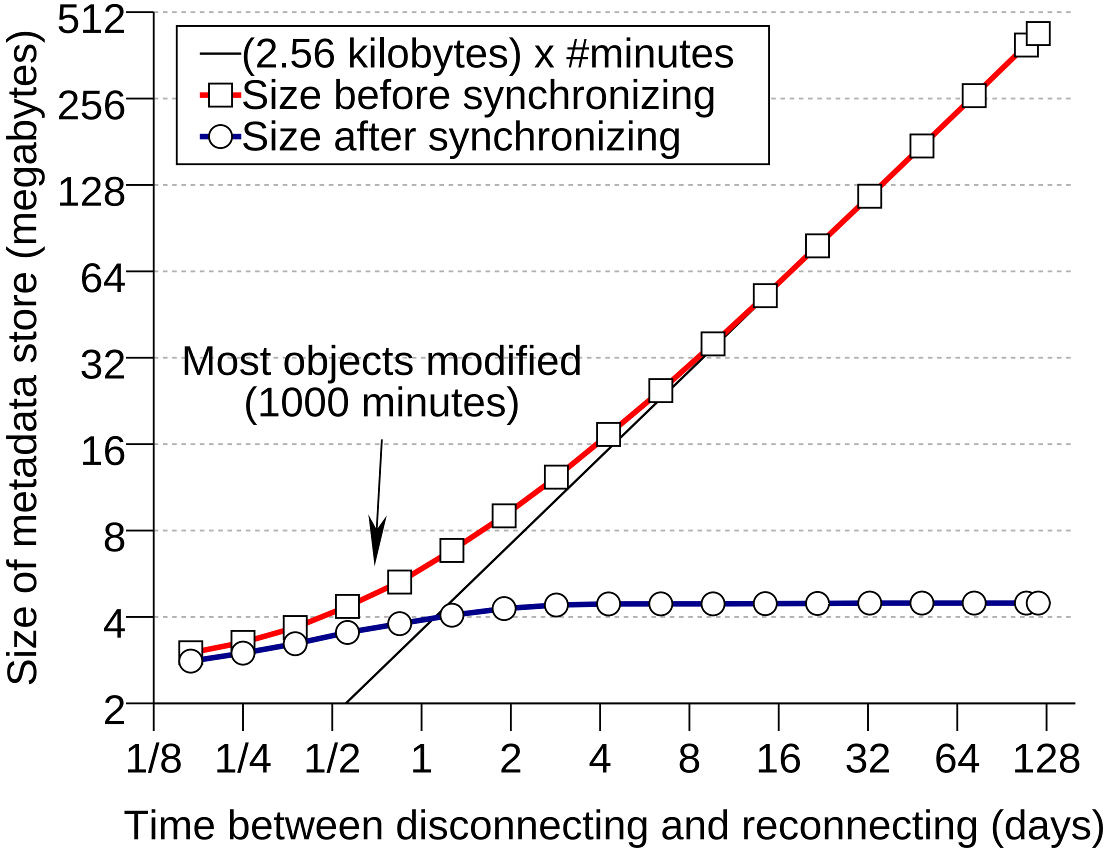

Figure 6-3: Storage consumed by metadata versions queued for a disconnected device.

6.6 Disconnected Devices

When an *Eyo* device, R, is disconnected from the rest of the group due to network partitions, or because the device in question is turned off, the other devices will keep extra metadata object versions, which might prove necessary to construct causally ordered version graphs once R returns.

Using the topology in Figure 6-1, we place an initial set of 1000 unconflicted objects synchronized across the three devices. The remote device R then disconnects from the network, and stays disconnected for a single period of time Δt ranging from four hours to four months. Starting after R is out of communication, the other replicas generate new versions to one of the existing objects at an average rate of once per minute, attaching 2 kilobytes of unique metadata, so the devices save no space by storing only changed attributes.

After the interval Δt, we measure the size of the *Eyo* metadata store on the generating devices, allow R to reconnect and synchronize, let each device prune its metadata, and then measure the metadata store again. Figure 6-3 shows the before and after sizes as a function of the disconnect interval Δt. The figure shows two regions, for Δt before and after 1000 minutes, the point at which most objects have been modified. For $\Delta t \gg 1000$ minutes, the system reaches a steady state where the size of the metadata store is proportional to the amount of time passed, but after returning and synchronizing shrinks to a constant size independent of the amount of time spent disconnected. The amount of recoverable storage is the difference between the two curves. The current *eeyore* implementation stores exactly one version beyond those strictly necessary to go back to the newest unique ancestor for each object, which is why this steady state size is larger than the initial storage size, and why the post-synchronization size changes during the initial non-steady state region.

A collection with more objects (for example, the one shown in section 6.4) would show a much smaller fraction of recoverable storage than this example, though the absolute amount of recoverable space would be the identical under the same update pattern.

All of the object types shown in Table 6.3 contain immutable contents, so disconnected devices using those data types cause overhead in *Eyo*'s metadata store, but not the content store. If updates change content as well, then the storage costs would be proportionally larger.

Figure 6-3 shows that a long-term uncommunicating device can cause unbounded growth of the metadata store on other devices. If this absence persists long enough that a device runs out of space, *Eyo* can present the user with two options: turn on and synchronize the missing device, or evict it from the system. Evicting the missing device, as discussed in section 4.5, does not require a consensus vote of the remaining devices. Temporarily evicting a device allows the remaining devices to truncate history, and preserves data until re-adding the missing device later.

These results show that users are unlikely to encounter problems due to accumulating metadata in practice, as large collections and infrequently used devices alone cannot cause problems. It is instead the rate of individual edits that consumes excess space, and none of

System	Description
Unison	Delays of at least 1 second for small collections. Large collections take significantly longer: 23 seconds for an existing collection of 500K objects, 87 seconds for 1M objects
MobileMe	Most updates arrive with delays of between 5 and 15 seconds. Occasionally as long as 4 minutes. Delay does not depend on collection size.
Eyo	All delays fall between 5 and 15 milliseconds. Delay does not depend on collection size.

Table 6.4: Synchronization Delay Comparison: Time to propagate one new update to an existing data collection between two devices on the same local network.

the applications we have examined generate changes anywhere near the frequency that this experiment assumes.

6.7 Synchronization Comparison

This section compares the performance of *Eyo*'s synchronization protocol to two existing alternatives: Unison [4], a stand-alone file-level synchronization tool, and MobileMe [3], a cloud-based storage subscription service.

These experiments aim to measure the time it takes for a minimal metadata change to propagate between two physically adjacent devices. In this setting, to provide a device-transparent view of data and show the same data view to users, frequent updates must pass between devices as quickly as possible. In each case, two devices initially hold a synchronized data collection with some number of existing small or metadata-only objects. One device then makes a single minimal change, and we measure the time it takes for that update to appear on the second device. For MobileMe, the single change took the form of editing an existing calendar entry to fall instead on the next or previous day. For Unison, the change was a one-byte edit to an existing single block-sized file, and for *Eyo* the change was a new metadata version of an existing metadata object.

Table 6.4 summarizes the results of measuring the update propagation delay for each of these systems. Since Unison is a stand-alone synchronizer, the measurement time includes the time to start up the program to send an update, which results in delays of around one second even for very small data collections. *Eyo* (and MobileMe) run continuously, so do not suffer such a startup cost. When started, Unison must first iterate over the local data collection to determine which files have changed, and for large data collections, this time dominates the end-to-end delay, resulting in delays of tens of seconds for collections of a few hundred thousand individual objects. *Eyo* never needs to iterate over the local metadata collection to identify which objects need updates, as *Eyo* continually tracks object changes that need propagation to other devices.

MobileMe and *Eyo* both track updates as applications edit data, so the delays are independent of the number of objects in the collection. Although both systems in this comparison send similar amounts of data (less than 10 kilobytes), MobileMe updates take between several seconds to several minutes to propagate, whereas *Eyo*'s delays fall between 5 and 15 milliseconds. MobileMe's star topology requires that all updates pass through the central cloud system, even if the two devices are physically adjacent on the same local network, as they are in this example. MobileMe's delays are not due to client polling, as clients appear to learn of new updates via an asynchronous notification via a persistent TCP connection, but are longer than can be attributed solely to network propagation delays. *Eyo*, in contrast, discovers local network paths, and uses those to send updates directly to the local device.

We expect that systems that are designed with the same performance goal as *Eyo*, namely to ensure that synchronization processing time and communication size is independent of the total collection size, (e.g., Cimbiosys [44], WinFS [31]), would show results very similar to *Eyo*'s in this type of setting.

The results of these measurements demonstrate that passing updates quickly between peer devices requires a synchronization protocol that efficiently identifies missing updates to send without scanning the data collection, and taking advantage of local networks to send updates directly whenever possible.

Chapter 7

Related Work

Many of the underlying mechanisms in *Eyo* derive from mechanisms in other systems. Cimbiosys & Perspective are the two most closely related systems, which we discuss next, followed by other optimistic replication schemes, and other systems such as version control systems and attribute-based file systems.

7.1 Cimbiosys & Perspective

Cimbiosys [44] and Perspective [48], are the two systems most closely related to *Eyo*. Though neither attempts to provide device transparency, *Eyo* shares ideas with each. For example, *Eyo* adopts placement rules from existing mechanisms in both systems.

Cimbiosys is a replication platform for applications to use content-based filtering rules with efficient synchronization protocols to minimize communication overhead. Cimbiosys does not provide a device-transparent view: devices learn about objects that match their local filter, and must store all of those objects, but do not learn about the rest of the objects in the data collection. Cimbiosys supports large groups of devices, and unlike *Eyo*, does not require that the devices know *a priori* of the identities of the other peers. In order to achieve efficient communication (dependent on the number of changes, rather than the number of total objects), Cimbiosys requires that the devices organize into a tree structure based on their content filters, and that devices periodically exchange updates with their parent and child devices in this tree. The device that sits at the root of this tree must hold a universal filter, meaning that it collects and then holds a copy of all content in the data collection. *Eyo* does not require that devices organize into a tree structure, or that any one device in the collection hold a complete copy. Cimbiosys requires that applications manage communication with peer devices, unlike *Eyo*, which manages all communication itself.

Perspective allows users to specify *views* over a data collection, which map content queries to devices which should hold replicas of those objects. Perspective does not provide communication protocols as efficient as in Cimbiosys or *Eyo*: a single synchronization event takes $\mathrm{O}(min(n_1, n_2))$ time, where n_i is the number of files stored on device i. Perspective does not provide disconnected device-transparent access to the data collection, as disconnected devices only know about files in their matching view. Perspective exports it's views via a traditional filesystem API, so does not require any application changes, unlike *Eyo* and Cimbiosys.

Neither Cimbiosys nor Perspective retains object's version history, or provides an API to applications that helps them manage and resolve conflicts simply, though both detect concurrent changes to objects.

7.2 Optimistic Replication Schemes

In addition to Cimbiosys and Perspective, *Eyo* incorporated ideas found in several other optimistic replication schemes. Coda [27], Ficus [23], Ivy [36], and Pangaea [47] provide optimistic replication and consistency algorithms for file systems. Coda uses a centralized set of servers with disconnected clients. Ficus and Ivy allow for updates between clients, but do not provide for partial replicas, and Pangaea handled disconnected servers, but not disconnected clients. An extension to Ficus [45] adds support for partial replicas, at the cost of no longer supporting arbitrary network topologies.

Several of these systems make use of Application-specific resolvers [28, 46], which require developers to construct stand-alone mechanisms to interpret and resolve conflicts separately from the applications that normally access that data. While *Eyo*'s approach does require direct changes to applications, embedding resolution logic directly in the applications avoids the need to recreate application context in separate resolvers, and permits multiple applications to edit, and subsequently resolve, changes to the same data objects. Presenting version history directly to the applications, instead of just the final state of each conflicting replica, permits applications using *Eyo*'s API to precisely identify the changes made in each branch.

BlueFS [37] and EnsemBlue [39] extend Coda to permit a limited degree of decentralized updates along with more flexible placement rules. *Eyo*'s lookup and watch notifications provide applications with similar flexibility as EnsemBlue's persistent query interface without requiring that a central server know about and process queries.

Podbase [42] replicates files between personal devices automatically whenever network conditions permit, but does not provide a way to specify placement rules or merge or track concurrent updates.

Bayou [58] provides a device transparent view across multiple devices, but does not support partial replicas, and requires all applications to provide merge procedures to resolve all conflicts. Bayou, like most optimistic replication schemes, requires that updates be *eventually-serializable* [13]. *Eyo* instead tracks derivation history for each individual object, forming a partial order of happened-before relationships [29].

PersonalRAID [53] tries to provide device transparency along with partial replicas. The approach taken, however, requires users to move a single portable storage token physically between devices. Only one device can thus use the data collection at a given time.

TierStore [11], WinFS [31], PRACTI [5], Pheme [25], and Mammoth [6] each support partial replicas, but limit the subsets to subtrees of a traditional hierarchical filesystems rather than the more flexible schemes in Cimbiosys [44], Perspective [48], and *Eyo*. TierStore targets Delay-Tolerant-Networking scenarios. WinFS aims to support large numbers of replicas and, like *Eyo*, limits updates messages to the number of actual changes rather than the total number of objects. PRACTI also provides consistency guarantees between different objects in the collection. *Eyo* does not provide any such consistency guarantees, but *Eyo* does allow applications to coherently name groups of objects through the exposed persistent object version and content identifiers. None of these systems provide device transparency over a complete collection.

7.3 Star Topologies

A number of systems build synchronization operations directly into applications so that multiple clients receive updates quickly, such as one.world [22], MobileMe [3], Google Gears [20], and Live Mesh [34]. Each of these systems follows the cloud model described in section 1.1.2, where a centralized set of servers hold complete copies of the data collections, and applications, either running on the cloud servers themselves, or on individual clients, retrieves some subset of the content. Disconnected clients cannot share updates directly, nor view complete data collections while disconnected.

7.4 Point to point synchronization:

Point-to-point synchronization protocols such as rsync [61], tra [9], and Unison [4] provide on-demand and efficient replication of directory hierarchies. Unison compares directory hierarchies on two machines and updates both copies to include changes made on the other. Tra keeps additional state on synchronization events to avoid detecting false conflicts when synchronizing groups of more than two devices. Rsync (which unison uses internally) efficiently compares large files to only send the changed portions at synchronization times. None of these systems easily extend to a cluster of peer devices, handle partial replicas without extensive hand-written rules, or proactively pass updates whenever connectivity permit without user intervention. Since all of these systems user the standard file system interface, none require application changes.

7.5 Version Control Systems

Software version control systems such as Git [19], Subversion [56], and Mercurial [33] provide many algorithms and models for reasoning about version histories, allowing developers to time-shift working sets back to arbitrary points. Version control systems normally store the complete history for each object, to permit developers to examine the entire lifetime of an individual object. Subversion keeps the complete data collection in a single centralized repository, so users can only resolve conflicts (or exchange updates) when they can communicate with the repository. Distributed version control systems such as git and Mercurial store complete collections of the entire project history on each client, so that operations such as committing or merging can occur between any two clients. *Eyo* keeps only a limited history needed to describe events leading to a potential conflict. Some version control systems (like CVS or Subversion) permit partial replicas, where some clients check out subdirectories of an overall project. Others, like git, require that clients hold a complete copy of a data collection. In this respect, git provides device transparent access to a repository, though it is not suitable for storage-limited devices that cannot store the collection's entire history.

7.6 Attribute Naming

Storage system organization based on queries or attributes rather than strict hierarchical names have been studied in several single-device (e.g., Semantic File Systems [18], HAC [21], hFAD [50], LISFS [38]) and multi-device settings (e.g., HomeViews [17]), in addition to the contemporary optimistic replication systems. Several of these systems observe that strict hierarchies found in traditional filesystems pose unnecessary restrictions on data organization and concurrency, that users frequently ignore the folders and use searches to locate their files instead, and requires that separate machines agree on a single organizational structure. *Eyo* uses attribute-based queries for applications to identify objects for the same reasons as in each of these systems.

Chapter 8

Discussion and Future Work

This chapter covers two topics: (1) it describes several extensions that could incorporate additional features into *Eyo*'s current design, and (2) it considers alternative designs that would follow from different assumptions about *Eyo*'s use and goals.

8.1 Extensions

This section describes several possible additions to *Eyo*'s base design to provide additional features.

8.1.1 Security considerations

The *Eyo* design as presented so far assumes (1) that each device within a group of devices faithfully carries out the synchronization protocol and stores the data it promises to store, and (2) that *Eyo* only receives valid instructions from end users. *Eyo* already partially addresses the first issue by using any storage space to replicate data beyond the copies required by placement rules, which provides some benefits in case of device failures. *Eyo* could adopt the strategies of Polygraph [30] to address the second type of attack (e.g., an attacker breaks into a device and issues commands to delete all items). In fact, *Eyo* already contains the necessary infrastructure to implement Polygraph's rollback mechanism, lacking only an interface to specify when an attack occurred.

8.1.2 Extension to multiple users

The discussion of *Eyo* thus far considered only a single user's devices. However, it may often be useful to share data collections between a few different people, for example if they live in a single household. For this purpose *Eyo* builds on UIA's shared groups [16], which provide a way to name the union of all devices controlled by several users. *Eyo* maintains separate metadata stores on each device, one for personal data, and one for shared data. All devices in a shared group can create, modify, or delete objects in the shared store, but only devices in the personal group can see or modify objects in the personal store. *Eyo*'s support for shared collections does not currently scale to large numbers of users and devices, but it should be adequate for family-sized groups, each member having a few devices.

8.1.3 Extension for storage-limited devices

If an *Eyo* group contains devices that are limited enough that they expect to be unable to hold even the full metadata collection, they can instead act as limited edge devices. This mode of operation would not present a device-transparent view, but may be useful for devices such as photo-frames that have limited storage space and user interfaces. These devices would gather all metadata updates, but only retain metadata objects matching their own placement rules. They would therefore be unable to forward updates further.

8.1.4 User Study

Although we have used *Eyo* ourselves and found it useful for our own purposes, a broader user study could provide additional support for our conclusions. In addition to our own investigations, a group at Nokia research is using *Eyo* in a system to present and propagate collections of social networking data and has found *Eyo*'s API to be very useful for this purpose.

8.2 Alternative Designs

This section describes alternative designs in cases where the desired properties differ from *Eyo*'s in several ways.

8.2.1 Implementing *Eyo* without UIA

It would be possible to implement almost all of *Eyo*'s design without relying on UIA for communicating between devices. If users manually constructed a list of their devices, and limited communication to times when those devices could directly communicate through secure channels, such as a local USB connection, or via a HTTPS server, the same metadata and content exchange protocols could work over such a system, although it would be harder for end users to describe the group initially.

The resulting system would still provide device-transparency, though it would miss opportunities for passing updates between devices that could nominally communicate via an internet relay. Furthermore, without a way to authenticate those links, devices could not ensure that the updates they received were authentic. As such, end users would be more likely to experience version conflicts when modifying the same objects on multiple devices.

8.2.2 Mutable Content

Eyo would not need extensive changes to handle frequently mutated content. The current design handles mutable content by replacing it entirely, which is simple but inefficient. On the storage side of the design, small changes to content objects could be stored in a Merkle tree [43] to avoid storing complete copies. The protocol for fetching content from other devices would need to be augmented to take this blocking into account to avoid transferring the same sub-portion of an object more than once, as existing tools already do (e.g, rsync, git, and many others). Applications reading objects would need to read content objects after combining the multiple blocks, which could be done by providing custom functions to read objects rather than returning file descriptors, or by implementing a FUSE [14] user-level filesystem and maintaining the existing file descriptors.

Even with these changes, applications would need additional changes to evaluate and merge content changes. These changes would likely be very specific to the individual data types, and hard to generalize across different applications and data types.

8.2.3 No Disconnected Operations

If *Eyo* devices were only ever used in situations where they could communicate with a single large centralized server, devices could provide device-transparent access to a data collection without requiring that each device store a complete metadata copy. Much of *Eyo*'s design would still be useful, however, because there would still be long delays for devices not physically near the central server. To limit user-visible delays while evaluating queries and displaying results, it would still make sense to cache frequently accessed metadata on devices. Transmission delays for large content objects would still necessitate playing content on individual devices, meaning that *Eyo*'s placement rules would remain as designed. If applications checked with the central server on each data write, and aborted or rolled back any concurrent writes, *Eyo* could avoid keeping version histories, as the central server could decree which was the newest version of any single object. Applications could optimize the caching of object metadata by notifying *Eyo* (perhaps at install time) which attributes they use in order to identify and display objects to users. *Eyo* could then cache those values locally, while ignoring attributes or entire objects that lack an appropriate local application to view those objects.

8.2.4 Without placing all metadata everywhere

If *Eyo* did not place metadata for every object on each device, but still required disconnected operation, it could not provide device-transparent access to the data collection. This change would provide a different experience to end users, and would be more similar to Cimbiosys [44] in operation. The metadata synchronization protocol would need to incorporate placement rules that operate on metadata in addition to rules that operate on object content. The content synchronization protocol would need some different mechanism to locate objects, and to ensure their persistence, without gossiping this information via the metadata synchronization protocol. One way to ensure this would be to adopt, as Cimbiosys did, the requirement that some device serve as the root of a filter tree, and promise to hold a complete copy of all metadata and all content. This central point would thus serve as the fallback device to fetch content from.

Chapter 9

Summary

Growing storage and network capabilities of mobile devices, combined with personal data collections that do not fit on some of the devices, leads to confusion caused by the object-on-a-device abstraction that traditional storage systems provide. This thesis describes an alternative abstraction, *device transparency*, that unifies the collections of objects on multiple devices into a single logical collection. It proposes a novel storage API that provides explicit version histories, application-defined metadata that is stored separately from object content, and placement rules.

An implementation of this API in the *Eyo* storage system includes effecient synchronization protocols for object metadata and content through direct peer-to-peer links. The metadata protocol communicates updates continuously and automatically whenever network connectivity permits.

An evaluation with several applications suggests that adopting *Eyo*'s API to achieve device transparency for these application is modest, most cases of concurrent updates can be handled automatically by the applications wihtout user intervention, and that the storage and bandwidth costs are within the capabilities of typical personal devices.

The main ideas explored in *Eyo* can hopefully be adopted into future mobile platforms. Doing so would enhance their user experiences, and provide users with better control over their personal data. Users would manage a single unified data collection, rather than combinations of independent device-sized partitions. The source code for the *Eyo* prototype implementation will be available publicly from `http://pdos.csail.mit.edu/eyo/`.

Bibliography

[1] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[2] Apple, Inc. iTunes. http://www.apple.com/itunes/.

[3] Apple Inc. MobileMe. http://www.apple.com/mobileme/.

[4] S. Balasubramanian and Benjamin C. Pierce. What is a file synchronizer? In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998.

[5] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[6] Dmitry Brodsky, Jody Pomkoski, Shihao Gong, Alex Brodsky, Michael J. Feeley, and Norman C. Hutchinson. Mammoth: A Peer-to-Peer File System. Technical Report TR-2003-11, University of British Columbia, Department of Computer Science, 2002.

[7] Bram Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[8] Apache CouchDB. http://couchdb.apache.org/.

[9] Russ Cox and William Josephson. File Synchronization with Vector Time Pairs. Technical Report MIT-CSAIL-TR-2005-014, MIT, 2005.

[10] D-Bus. http://dbus.freedesktop.org/.

[11] Michael Demmer, Bowei Du, and Eric Brewer. TierStore: A Distributed File-System for Challenged Networks. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, 2008.

[12] Kevin Fall, Wei Hong, and Samuel Madden. Custody Transfer for Reliable Delivery in Delay Tolerant Networks. Technical Report IRB-TR-03-030, Intel Research Berkeley, 2003.

[13] Alan Fekete, David Gupta, Victor Luchangco, Nancy A. Lynch, and Alexander A. Shvartsman. Eventually-Serializable Data Services. In *Symposium on Principles of Distributed Computing*, 1996.

[14] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net.

[15] Flickr. http://www.flickr.com/.

[16] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent Personal Names for Globally Connected Mobile Devices. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[17] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy. Homeviews: peer-to-peer middleware for personal data sharing applications. In *Proceedings of the 2007 ACM International Conference on Management of Data (SIGMOD'07)*, 2007.

[18] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic File Systems. In *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[19] Git. http://git.or.cz/.

[20] Google Gears. http://gears.google.com.

[21] Burra Gopal and Udi Manber. Integrating Content-based Access Mechanisms with Hierarchical File Systems. In *OSDI '99: Proceedings of the third Symposium on Operating Systems Design and Implementation*, 1999.

[22] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[23] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the USENIX Summer Conference*, June 1990.

[24] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[25] Jiandan Zheng, Nalini Belaramani, and Mike Dahlin. Pheme: Synchronizing Replicas in Diverse Environments. Technical Report TR-09-07, University of Texas at Austin, February 2009.

[26] D. Scott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[27] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[28] Puneet Kumar and M. Satyanarayanan. Flexible and Safe Resolution of File Conflicts. In *Proceedings of the USENIX 1995 Technical Conference*, 1995.

[29] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[30] Prince Mahajan, Ramakrishna Kotla, Catherine Marshall, Venugopalan Ramasubramanian, Thomas Rodeheffer, Douglas Terry, and Ted Wobber. Effective and Efficient Compromise Recovery for Weakly Consistent Replication. In *Proceedings of the Fourth ACM European Conference on Computer Systems (Eurosys'09)*, 2009.

[31] Dahlia Malkhi, Lev Novik, and Chris Purcell. P2P replica synchronization with vector sets. *SIGOPS Oper. Syst. Rev.*, 41(2):68–74, 2007.

[32] John McCutchan. inotify. http://inotify.aiken.cz/.

[33] Mercurial source control management. http://mercurial.selenic.com/.

[34] Microsoft. Live Mesh. http://www.livemesh.com.

[35] MongoDB. http://www.mongodb.org/.

[36] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[37] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation (OSDI'04)*, 2004.

[38] Yoann Padioleau and Olivier Ridoux. A Logic File System. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[39] Daniel Peek and Jason Flinn. EnsemBlue: Integrating Distributed Storage and Consumer Electronics. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[40] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, 1997.

[41] Picasa web albums. http://picasaweb.google.com/.

[42] Ansley Post, Petr Kuznetsov, and Peter Druschel. PodBase: Transparent Storage Management for Personal Devices. In *Proceedings of the 7th International Workshop on Peer-to-Peer Systems (IPTPS'08)*, February 2008.

[43] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, 2002.

[44] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A Platform for Content-based Partial Replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

[45] David Ratner, Peter L. Reiher, Gerald J. Popek, and Richard G. Guy. Peer Replication with Selective Control. In *Proceedings of the First International Conference on Mobile Data Access*, 1999.

[46] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the 1994 USENIX Summer Conference*, 1994.

[47] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.

[48] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: Semantic Data Management for the Home. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, 2009.

[49] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implemenration of the Sun Network Filesystem. In *Proceedings of the Usenix Summer Conference*, pages 119–130, Portland, Oregon, 1985.

[50] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems are Dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII*, 2009.

[51] SLOCCount. http://www.dwheeler.com/sloccount/.

[52] Smugmug. http://www.smugmug.com/.

[53] Sumeet Sobti, Nitin Garg, Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Y. Wang. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *Proceedings of the first USENIX Conference on File and Storage Technologies (FAST'02)*, 2002.

[54] SQLite. http://www.sqlite.org/.

[55] Jacob Strauss, Chris Lesniewski-Laas, Justin Mazzola Paluska, Bryan Ford, Robert Morris, and Frans Kaashoek. Device Transparency: a New Model for Mobile Storage. In *Proceedings of the First Workshop on Hot Topics in Storage and File Systems (HotStorage'09)*, October 2009.

[56] Subversion. http://subversion.tigris.org.

[57] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[58] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, and Alan J. Demers. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.

[59] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An Architecture for Internet Data Transfer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[60] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7(3):323–342, 1982.

[61] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.