Eyo: Device-Transparent Personal Storage

Jacob Strauss[*]   Justin Mazzola Paluska

Chris Lesniewski-Laas   Bryan Ford[†]   Robert Morris   Frans Kaashoek

Massachusetts Institute of Technology   [†]*Yale University*
[*]*Quanta Research Cambridge*

Abstract

Users increasingly store data collections such as digital photographs on multiple personal devices, each of which typically offers a storage management interface oblivious to the contents of the user's other devices. As a result, collections become disorganized and drift out of sync.

This paper 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* synchronizes these collections across any network topology, including direct peer-to-peer links. *Eyo* provides applications with a storage API with first-class access to object version history in order to resolve update conflicts automatically.

Experiments with several applications using *Eyo*—media players, a photo editor, a podcast manager, and an email interface—show that device transparency requires only minor application changes, and matches the storage and bandwidth capabilities of typical portable devices.

1 Introduction

Users often own many devices that combine storage, networking, and 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 objects from whichever device they are using, in addition to the device where they first created or added the object to their collection. Currently, users must manually decide to shuttle all objects to a single master device that holds the canonical copy of a user's object collection. This hub-and-spoke organization leads to 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 determine whether a

and b are different objects, copies of the same object, or different versions of the same object.

A better approach to storing personal data would provide *device transparency*: the principle that users should see the same view of their data regardless of which of their devices they use. Device transparency allows users to think about their unified data collection in its entirety regardless of which device a particular object may reside on, rather than as the union of independent copies of objects scattered across their devices.

Traditional distributed file systems provide *location transparency* whereby a file's name is independent of its network location. This property alone is insufficient for use with disconnected, storage-limited devices. A device-transparent storage system, however, would provide the same abstraction regardless of connectivity.

One attempt at providing device transparency is to store all data on a centralized cloud server, and request objects on demand over the network. In the presence of poor or disconnected networks, however, this approach fails to provide device-transparency: disconnected devices cannot access new objects or old objects not cached locally. In addition, two devices on the same fast local network cannot directly exchange updates without communicating with the central hub.

Beyond the challenge of transferring data between devices, either by direct network connections or via centralized servers, providing device-transparent access to a data collection faces two additional challenges: (1) concurrent updates from disconnected devices result in conflicting changes to objects, and (2) mobile devices may not have enough space to store an entire data collection.

This paper presents *Eyo*, a new personal storage system that provides device transparency in the face of disconnected operation. *Eyo* synchronizes updates between devices over any network topology. Updates made on one device propagate to other reachable devices, and users see a single coherent view of their data collection from any of their devices. Since these updates may cause

conflicts, *Eyo* supports *automated conflict resolution*.

The key design decision behind *Eyo* is to use object metadata (e.g., author, title, classification tags, play count, etc.) as a proxy for objects themselves. This decision creates two requirements. First, *Eyo* requires applications to *separate object metadata from content*, so that *Eyo* knows what is metadata and what is content. Second, *Eyo* must replicate metadata on *every* device, so that applications can manage any object from any device as though that device held the master copy of that object.

To meet these requirements, *Eyo* provides a new storage API to applications. This API separates metadata from content, and *presents object version histories as first-class entities*, so that applications can automatically resolve most common divergent version histories without user intervention, while incorporating the presentation and resolution of other conflicts as a part of ordinary operation. In return, applications delegate inter-device synchronization to *Eyo*.

Experiments using *Eyo* in existing applications—media players, a photo editor, a podcast manager, and an email interface—show that *Eyo* transforms these standalone applications into distributed systems providing device-transparent access to their data collections, takes advantage of local peer-to-peer communication channels, permits automatic conflict resolution, and imposes only modest storage and bandwidth costs on devices.

Eyo's main contribution is a design for device transparency for disconnected storage-limited devices, building on our earlier proposal for device transparency [44]. The design adopts techniques pioneered by existing systems (e.g., disconnected operation in Coda [22], application-aware conflict resolution in Bayou [46], placement rules in Cimbiosys [36] and Perspective [40], version histories in source control systems [16], update notifications in EnsemBlue [33]). Eyo wraps these techniques in a new storage interface that supports efficient, continuous, peer-to-peer synchronization, and avoids most user involvement in conflict resolution.

The remainder of this paper is organized as follows: Section 2 describes *Eyo*'s API and its use, followed by *Eyo*'s synchronization protocols in Section 3 and implementation in Section 4. Section 5 evaluates *Eyo* with existing data collections and applications. Section 6 describes related systems, and Section 7 concludes.

2 *Eyo*

Eyo enables a traditionally architected, single-device application to work as a distributed application whose state is scattered among many devices. For example, suppose we have a traditional photo management application that copies photos from a camera into a local database of photo albums. After modifying the application to use *Eyo*, the album database becomes replicated automati-

cally across all devices, permitting the user to manage her photo collection from whichever device is most convenient. *Eyo* maintains these properties for the photo application on all devices, even if a given device isn't large enough to hold the entire collection of photos.

Eyo sits between applications, local storage, and remote network devices. *Eyo* uses an overlay network [13] 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.

In order to bring these features to applications, *Eyo* provides a new storage API. *Eyo*'s API design makes the following assumptions about applications:

- Users identify objects by metadata, not filesystem path. For example, headers and read/replied flags of emails or the labels and dates of photos.
- The application provides user interfaces that make sense when the device stores only object metadata; for example, songs listings or genre searches.
- Modification of metadata and insertion/deletion of objects are common, but modification of object content is rare. For example, a user is more likely to change which folder a stored email message resides in, and less likely to change the message itself.
- Metadata is small enough to replicate on every device. In our application study (Section 5.1), we find that metadata is less than 0.04% of the size of typical music and photo objects.
- Application developers agree on the semantics of a basic set of metadata for common data types, in order to permit multiple applications to share the same data objects: e.g., standard email headers, ID3 tags in MP3 audio files, or EXIF tags in photos. Applications can still attach arbitrary data to metadata in addition to the commonly agreed upon portions.

We believe that these assumptions match the characteristics of common personal data management applications. The following sections describe how *Eyo*'s techniques can transform such applications into peer-to-peer distributed applications operating on a device-transparent data collection, using the photo album application as a running example.

2.1 Objects, metadata, and content

Eyo stores a user's object collection as a flat set of versioned objects. Figure 1 shows an example of an object representing one photo, with multiple versions from adding and removing organizing tags. Each *Eyo* version consists of a directed acyclic graph of collections of metadata and content. Edges in the version graph denote parent-child relationships. Newly created object versions include explicit predecessor pointers to their parent ver-
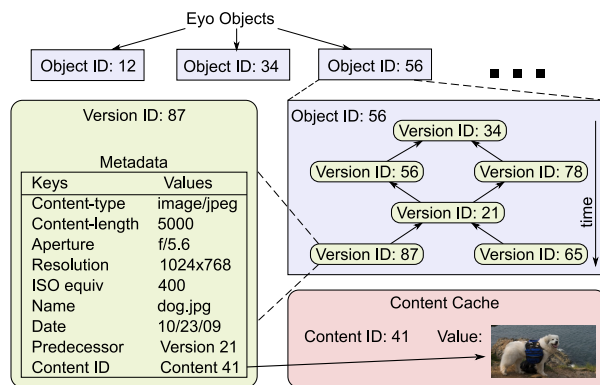
Figure 1: *Eyo* object store.

object creation and manipulation:

create(*ID hint*) → (*objectID*, *versionID*)

lookup(*query*) → list<(*objectID*, *versionID*)>

getVersions(*objectID*) → list<*versionID*>

getMetadata(*objectID*, *versionID*) → list<(*key,value*)>

open(*objectID*, *versionID*) → *contentID*

read(*contentID*, *offset*, *length*) → *contents*

newVersion(*objectID*, list<*versionID*>,

             metadata, *contents*) →*versionID*

deleteObject(*objectID*) → *versionID*

placement rules:

addRule(*name*, *query*, *devices*, *priority*) → *ruleID*

getRule(*name*) → (*ruleID, query, devices, priority*)

getAllRules() → list<*ruleID, query, devices, priority*>

removeRule(*ruleID*)

event notifications:

addWatch(*query*, *watchFlags*, *callback*) → *watchID*

removeWatch(*watchID*)

callback(*watchID*, *event*)

Figure 2: *Eyo* API summary. Event notifications are discussed in Section 2.2, and placement rules in Section 2.3.

sions, represented as a *parent* version attribute. An object version's metadata consists of a set of *Eyo-* and application-defined key/value pairs. The metadata also contains a content identifier; the associated content might or might not be present on any particular device.

Applications retrieve objects via queries on metadata. *Eyo* expects applications to maintain rich enough metadata to display to the user meaningful information about an object, even on devices not storing the content. In our photo album example, the metadata may include rating, album, and location to help the user sort photos.

Eyo's API contains elements similar to databases for searching, reading, and editing object metadata, along with elements similar to filesystems for reading and writing object content. This distinction is deliberate, as it matches common uses of media applications which often use both elements internally. In addition, the API

provides mechanisms to learn about and repair conflicts, to specify content placement rules, and to receive notices about changes to the object store. Figure 2 lists commonly used *Eyo* methods. The figure omits alternate iterator-based versions of these methods for constructing or viewing large collections as well as library functions combining these base operations. 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 an object's content, but the content is not present on the device, *Eyo* signals an error. A user can still perform useful operations on metadata, such as classifying and reorganizing objects (e.g., updating the rating of a photo), from a device that does not store content. 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 2.3). Section 3 shows how metadata replication supports efficient synchronization.

2.2 Queries

While *Eyo* does not provide human-readable object identifiers, it provides queries with which applications can implement their own naming and grouping schemes. For example, the photo application may tag photos with their associated albums. Queries return IDs for all objects that have metadata attributes matching the query. As in Perspective [40], users never see *Eyo* queries; applications create queries on their behalf.

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 inter-device synchronization, fulfilling a purpose similar to filesystem notification schemes such as inotify [28].

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-viewed photos, but does allow searching for photos viewed more than 100 times. *Eyo* limits queries to these restricted forms to assure efficiency for query 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.

2.3 Placement Rules

Eyo allows applications to specify *placement rules* controlling which objects' content has highest priority for storage on storage-limited devices. For example, the placement rules for our photo album application may

specify that a user's laptop should hold only recent albums, but that a backup device should hold every photo. Applications are expected to generate placement rules based on user input. Experience suggests that users are not very good at predicting what objects they will need or at describing those objects with rules [40]. *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.

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, the photo album application might present a UI allowing the user to indicate which devices should hold a complete photo album. 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 stores high-priority content in preference to low-priority. When space permits, *Eyo* provides *eventual filter consistency* [36] 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 3.4, ensures that at least one copy of content persists even if no placement rule matches.

To ensure that all devices know all placement rules, *Eyo* stores each rule as an object with no content, but whose metadata contain the query, priority, and device set. Any device can modify a placement rule. If a conflict arises between rule versions, *Eyo* conservatively applies the union of all current versions' requirements. Similarly, if an object has multiple current versions and any current version matches a placement query, *Eyo* acts as if the query had matched *all* versions back to the common ancestor. This behavior ensures that any device that may be responsible for the object's content has all versions required to recognize and resolve conflicts.

Because placement rules operate at object granularity, applications that maintain related variations of content should store these variations as separate objects linked via metadata, so that different placement rules can apply to each variation. For example, our photo application stores both a full size and a thumbnail size image of the same base photo, assigning a high priority placement rule to replicate the thumbnail objects widely, but placing the full-size versions only on high-capacity devices.

2.4   Object Version Histories

Much of *Eyo*'s API design and storage model is motivated by potentially disconnected devices. Devices carry replicas of the *Eyo* object store and might make independent modifications to their local replicas. Devices must therefore 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, as detailed in Section 3. Synchronization replaces each device's set of object versions and metadata attributes with the union of the devices' sets.

For example, in Figure 1, suppose device A uses *Eyo* to 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 its content. If an application on B then modifies $O56$'s metadata or content, the application calls newVersion($O56$, [O56:34], *metadata*, *content*), indicating that the new version ($O56{:}78$), 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 $O56{:}34$. Since the version history is linear, *Eyo* applications will use the unique most recent version.

2.5   Continuous Synchronization

To propagate updates to other devices as promptly as possible, *Eyo* provides *continuous synchronization*. Continuous synchronization helps reduce concurrency conflicts by propagating changes as quickly as the network allows, essentially serializing changes. Continuous synchronization also improves the user experience by showing changes from other devices "instantly". If two devices on the same network run the photo album application, for example, rating changes in one application will be immediately reflected in other application. Section 3 details continuous synchronization.

2.6   Automated Conflict Management

A primary goal of *Eyo*'s API is to enable applications to offer the user automated conflict management. To manage conflicts, applications need access to history information, notifications when conflicts arise, and a way to resolve those conflicts permanently.

Eyo uses per-object version histories and update notifications to provide a distributed metadata database that describes objects at the same granularity as user-visible objects. Applications thus need to examine only the *Eyo*-provided history of changes to a single object at a time in order to resolve changes. Cloud synchronization services that use existing filesystem APIs would instead require

applications to examine two (or more) complete copies of a metadata database and write a resolution procedure to operate on the entire collection at once.

Continuing with the example from Figure 1, consider a case where 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. When it detects concurrent updates, *Eyo* presents to the application each of the head versions along with their common ancestors.

Eyo's version graphs with explicit multiple parent versions are inspired by version control systems [16, 45]. Where version control systems keep history primarily for users to examine, *Eyo* instead 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 common 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. 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 [34].

Applications hold responsibility for handling concurrent updates of the same object on different devices, and should therefore structure the representation of objects in a way that makes concurrent updates either unlikely or easy to merge automatically. Applications must notice when concurrent updates arise, via *Eyo* queries or watch notifications. When they do occur, applications should either resolve conflicts transparently to the user, or provide ways for users to resolve them. This division allows *Eyo* to take advantage of fleeting network connectivity to transfer all new updates. Users avoid interruptions about irrelevant objects, and can wait until some more convenient time to merge conflicts, or perhaps ignore unimportant conflicts forever.

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 photo between albums on one device, while changing the rating for the same photo on another device. Applications can arrange for these pairs of operations to be *composable*, e.g., ensuring that album tags and ratings can be set independently in the metadata. *Eyo* identifies these conflicting modifications, but the applications themselves merge the changes since applications know the uses of these attribute types, and so can determine the correct final state for these classes of concurrent changes.

Some concurrent updates, however, require user intervention to merge them into a single version. For example, a user might change the caption of a photo different ways on different devices. In such cases it is sufficient for *Eyo* to detect and preserve the changes for the user, either to 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. All of *Eyo*'s API calls work regardless of whether an object contains an unresolved conflict, so it is up to applications as to whether they wish to operate on conflicted objects.

Applications may not intentionally create conflicts: when calling newVersion(), applications may list only 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 or newly learned version. *Eyo* can thus safely delete these older versions without affecting later conflict resolution. For example, in Figure 1, if all devices knew about version $O56:21$, that version is a *unique ancestor* for the object $O56$, and *Eyo* may *prune* the older versions $O56:34$, $O56:56$, and $O56:78$. Section 5.4 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 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 Continuous Synchronization

Eyo needs to synchronize two classes of data between devices, metadata and content, and faces 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, in contrast, can consist of large objects that change infrequently and 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 goal of content synchronization is to move objects to locations that best match placement policies.

Given the different needs for these two classes of data, *Eyo* uses different protocols for each class. Both run over UIA [13], an overlay network supporting direct peer-to-peer links as well as centralized cloud server topologies.

3.1 Metadata Synchronization

The primary goal of *Eyo*'s metadata synchronization protocol is to maintain identical copies of the entire metadata collection. This process must be efficient enough to run continuously: updates should flow immediately to devices connected to the same network. 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 and to push notifications to reachable devices whenever a local application writes a new version of an object. *Eyo* identifies and organizes changes as they occur rather than iterating over the entire collection, allowing *Eyo* to quickly find the set of changed objects (among a much larger set of unchanged objects) at every synchronization opportunity.

Eyo groups multiple metadata updates into a permanent collection called a *generation*. Each generation is uniquely named by the device that created it and includes an id field indicating how many generations that device has created. A generation includes complete metadata updates, but only identifiers and new status bits for content updates. All synchronization occurs at the granularity of individual generations; each device that holds 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 [32], but named differently to avoid confusion with the versions of individual objects. For a personal group with n devices, each *Eyo* device updates a single n-element vector of $(device, id)$ tuples indicating the newest generation authored by $device$ that it holds.

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 its local generation vector, or the next generation vector it will write if it has new changes pending. When a device receives a reply, it incorporates the newly learned changes into its local data store, updates its generation vector accordingly, notifies applications about newly learned changes, and updates and applies placement rules to the newly learned changes.

When a device receives an incoming getGenerations request, it first gathers all pending changes, if any, into a new generation. 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, the device queues a getGenerations request in
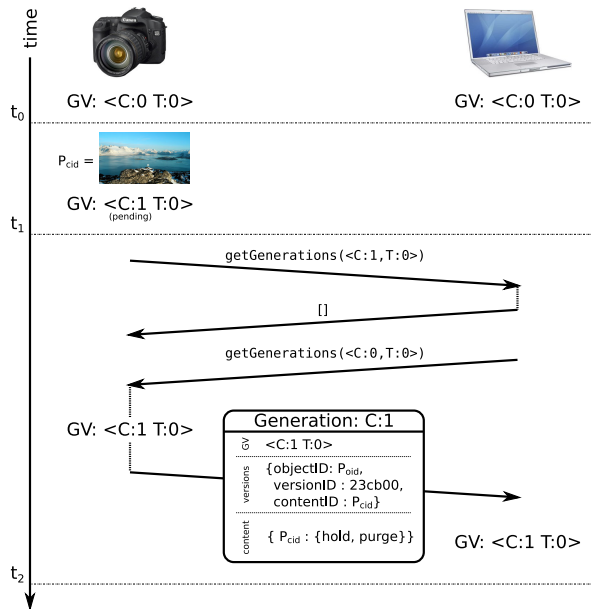

Figure 3: Metadata Synchronization: Messages sent between two devices for one new object

the reverse direction to update itself, either immediately, or when next reachable if the request fails.

Figure 3 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. Initially, at t_0 in Figure 3, 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 binds 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.

3.2 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 three separate steps: determining when generation objects have been seen by all devices in a group, 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. 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 G, it may lazily move G's contents into its *archive generation*, which groups together updates made by different devices and from different original generations, and does not retain those origins. *Eyo* preserves at least one generation for each device separate from the combined archive, 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.

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 then proceeds as described in Section 2.4.

3.3 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. Exist-
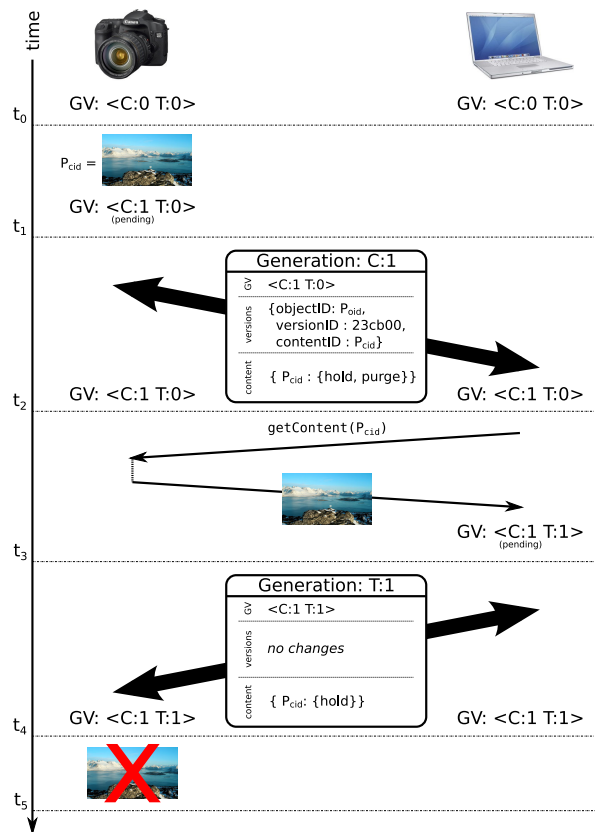

Figure 4: Content Synchronization. The thick double arrows represent a metadata sync from Figure 3.

ing 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 is not an undue burden.

Users remove devices from an *Eyo* group by deleting them from the underlying overlay network. 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. Removing an inactive or uncommunicative device from an *Eyo* group allows the surviving devices to make progress truncating history.

3.4 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 transit between devices.

Eyo uses placement rules to solve the first of these

challenges, as described in Section 2.3. 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 3). Since all devices learn about all metadata updates, all devices thus learn which devices should hold 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 [11] 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 issues a new metadata update removing its promise to keep the object in the future. 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 holding the last two copies of an object each simultaneously announce their desire to remove it, then the generations that contain these modifications cannot be totally ordered. Neither device will be able to delete the object, as neither can identify another device that has accepted responsibility for storing the object.

This protocol ensures that, as long as no devices are lost, stolen, or broken, 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 shows an example content sync that continues where the metadata sync of Figure 3 leaves off. 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. 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 $\texttt{C:1}$ 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, $\texttt{T:1}$.

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 if it needs to reclaim that space for new items, placing the system in a stable state matching the user's preferences.

4 Implementation

Eyo's prototype implementation consists of a per-user daemon that runs on each participating device and handles all external communication, and a client library that implements the *Eyo* storage API. The daemon is written in Python, runs on Linux and Mac OSX, 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. It uses SQLite [43] 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. The *Eyo* implementation uses XML-RPC for serializing and calling remote procedures to fetch metadata updates, and 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 [47] or BitTorrent [6], although we have not yet implemented swarming transfers. We implemented *Eyo* API modules for Python and a library for C applications. The client libraries fulfill most application requests directly from the metadata store via SQLite methods, though they receive watch notifications from the daemon via D-Bus [8] method calls.

5 Evaluation

We explore the performance of *Eyo* by examining the following questions:

- Is *Eyo*'s storage model useful for applications and users?
- Can *Eyo* resolve conflicts without user input?
- 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?

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. We focus on two types of applications: (1) media 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 while connected 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 also built an IMAP-to-*Eyo* gateway to enable existing email clients to access messages stored in *Eyo*.

We evaluate *Eyo*'s storage and bandwidth costs using three data collections: 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 compare *Eyo*'s synchronization protocols to existing synchronization tools, Unison and MobileMe. Although neither tool aims 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, and demonstrates the need for direct peer-to-peer updates.

5.1 *Eyo* Application Experiences

Adapting existing applications to use *Eyo* is straightforward. Table 1 summarizes the changes made to each application. In each case, we needed to modify only a small portion of each application, indicating that adopting the *Eyo* API does not require cascading changes through the entire application. The required changes were limited to modules composing less than 11% of the total project size for the C-based applications, and significantly less for the Python applications. The C-based application changes were spread over a few months; the python applications needed only a few days of work.

***Eyo* provides device-transparency.** *Eyo* transforms 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

Size (lines)	Rawstudio	Rhythmbox	QuodLibet	gPodder	Email
total project	59,767	102,000	16,089	8,168	3,476
module size	6,426	9,467	428	426	312
lines added	1,851	2,102	76	295	778
lines deleted	1,596	14	2	2	N/A
language	C	C	python	python	python
content	←—— individual files ——→				N/A
metadata	central DB, sidecar files	←—— central DB ——→			N/A

Table 1: Comparisons of applications adapted to *Eyo*, including lines of code changed along with descriptions of the application's original organization for storing metadata and content. For email, the "total project" size only includes Twisted's IMAP module and server example code, and "lines added" includes all of our newly written code.

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.

Device transparency brings new features to the applications. For example, Rhythmbox and QuodLibet can show the user's entire media collection, even when content is not present, allowing users to search for items and modify playlists from any device. In Rawstudio, users can search for or organize photos in the entire collection, even when the content is missing. Surprisingly few changes were necessary to support missing content. This is because applications already have code paths for missing files or unreachable network services. Content that is not on the current device triggers these same code paths.

Users rarely encounter metadata conflicts. As a consequence of device transparency, users may encounter conflicts from concurrent changes on multiple devices. These concurrent changes result in multiple head versions of these objects when connectivity resumes. For changes to distinct pieces of metadata, the version history *Eyo* provides permits applications to resolve concurrent changes simply by applying the union of all user changes; *Eyo*'s client library makes this straightforward.

For concurrent changes to the same piece of metadata, the application must manually resolve the conflict because the correct policy depends on the application and the metadata item. In most cases, users are never aware when concurrent updates occur, as the applications perform these operations automatically. For example, if one device changes an email message status to "read" while another device changes the status to "replied", *Eyo* will signal a conflict to the application. However, the IMAP gateway knows that these updates are composable and resolves the conflict without user intervention.

Application	Type	User-Visible Conflicts Possible?	Why?
IMAP	Email Gateway	No	Boolean 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 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.

	Email	Music	Photos
number of objects	724230	5299	72380
total content size	4.3 GB	26.0 GB	122.8 GB
native metadata size	169.3 MB	2.6 MB	22.6 MB
Eyo metadata size	529.6 MB	5.8 MB	52.9 MB
metadata/content overhead	12%	0.02%	0.04%
metadata store per object	766 bytes	1153 bytes	767 bytes

Table 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 *Eyo* metadata size is the on-disk size after adding all objects.

As shown in Table 2, it is possible to cause end-user visible effects. For example, Rhythmbox and QuodLibet allow users to modify metadata directly in the UI, which may require manual intervention to resolve. However, these kinds of user-visible conflicts only arise due to manual, concurrent changes and are rare in practice.

In Rawstudio, during the course of editing on two devices, users may create conflicting versions of a photo. Rather than hiding the change or requiring immediate conflict resolution, *Eyo* exposes each version as a "development version" of the photo. While this feature is typically used to let the user test different exposure and color settings, *Eyo* uses the feature to show concurrent branches of the photo.

In the other applications, gPodder and email, user-visible conflicts are impossible, as users cannot 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.

5.2 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 3 shows the resulting metadata store sizes.

The table shows that for each of the 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 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, not the content type nor the size of content objects. The number of objects, along with the amount of metadata per object, thus provides a lower bound on the necessary 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. Including video content would further reduce the relative amount of overhead *Eyo* devotes to storing object metadata.

5.3 Bandwidth Costs

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

To measure bandwidth costs, we placed a pair of object-generating devices on the same network and a remote device on a different network with a slow link to the object-generating devices. The 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 5.2). The remote object has no placement rules matching the new objects, so it does not fetch any of the associated content. As such, all of the bandwidth used by the remote device is *Eyo* metadata and protocol overhead.

The bandwidth consumed over the slow link, as expected, relates 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 remote 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.
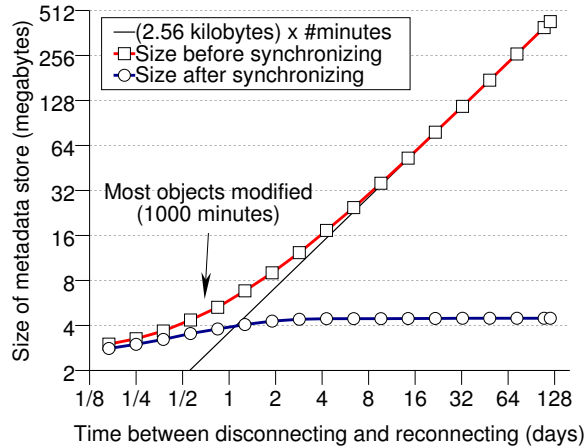
Figure 5: Storage consumed by metadata versions queued for a disconnected device (Log-Log plot).

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 after 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 4: Synchronization Delay Comparison: Time to propagate one new update to an existing data collection between two devices on the same local network.

5.4 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.

In this measurement, we place an initial set of 1000 non-conflicting 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 5 shows the before (square markers) and after (circle markers) 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 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 stor-

age 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 5.2) would show a much smaller fraction of recoverable storage than this example. The absolute amount of recoverable space would be the identical given the same sequence of updates.

All of the object types shown in Table 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 5 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 3.3, does not require a consensus vote of the remaining devices. Temporarily evicting a device allows the remaining devices to truncate history and preserve 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. None of the applications we have examined generate changes anywhere near the frequency that this experiment assumes.

5.5 Synchronization Comparison

This section compares the latency of exchanging a single small update between two physically adjacent devices using *Eyo* to two existing classes of synchronization tools: a point-to-point file synchronizer, Unison [3], and a cloud service, MobileMe [29]. In this experiment, 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. Table 4 summarizes the results. 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. After starting, Unison first iterates over the local data collection to determine which files have changed. 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.

MobileMe and *Eyo* both run continuously and continually track object changes that need propagation to other devices as applications edit data. Neither suffers a startup delay, and delays are independent of the number of objects in the collection. Although both systems send similar amounts of data (less than 10 kilobytes), MobileMe updates take between several seconds to several minutes to propagate, which is long enough for a person to notice the delay. *Eyo*'s delays in this topology fall between 5 and 15 milliseconds.

MobileMe's star topology requires that all updates pass through a distributed cloud system, even if the two devices are physically adjacent on the same local network, as in this example. *Eyo*, in contrast, discovers local network paths, and uses those to send updates directly to the local device.

Eyo can share types of data for which the other two are unsuited. Neither could store a music collection or a photo catalog shared between devices. If two devices read the catalog at startup and each later write some changes, the last write would win, and the other device's version of the file would be preserved separately but marked as a conflict. The user would have to choose one or the other versions, as the other synchronization tools provide no help for the application to resolve the concurrent changes automatically. *Eyo*, in contrast, naturally shares metadata about for these types of collections without requiring a user to routinely manage conflicts.

6 Related Work

The two systems most closely related to *Eyo* are Cimbiosys [36] and Perspective [40]. Though neither attempts to provide device transparency, *Eyo* shares ideas, like placement rules, with both. Cimbiosys provides an efficient synchronization protocol to minimize communication overhead while partially replicating objects across large groups of devices, but provides no way for a device to learn of all objects without storing all such objects. Perspective allows users to see their entire collection spanning several devices, but disconnected devices cannot continue to see the complete collection. Neither

of these systems preserve object history to help applications deal with concurrent updates. Polygraph [26] discusses extensions to Cimbiosys to guard against compromised devices. *Eyo* could apply these approaches for the same purposes.

Optimistic Replication Coda [22], Ficus [20], Ivy [30], and Pangaea [39] provide optimistic replication and consistency algorithms for file systems. Coda uses a centralized set of servers with disconnected clients. Ficus and Ivy allow updates between clients, but do not support partial replicas. Pangaea handles disconnected servers, but not disconnected clients. An extension to Ficus [37] adds support for partial replicas, but removes support for arbitrary network topologies.

BlueFS [31] and EnsemBlue [33] expand on approaches explored by Coda to include per-device affinity for directory subtrees, support for removable devices, and some consideration of energy efficiency. *Eyo*'s lookup and watch notifications provide applications with similar flexibility as EnsemBlue's persistent queries without requiring that a central server know about and process queries.

Podbase [35] 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 [46] 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 requires that updates be *eventually-serializable* [12]. *Eyo* instead tracks derivation history for each individual object, forming a partial order of happened-before relationships [24].

PersonalRAID [42] 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.

Systems like TierStore [9], WinFS [27], PRACTI [4], Pheme [49], and Mammoth [5] each support partial replicas, but limit the subsets to subtrees of a traditional hierarchical filesystems rather than the more flexible schemes in Cimbiosys, Perspective, and *Eyo*. TierStore targets Delay-Tolerant-Networking scenarios. WinFS aims to support large numbers of replicas and, like *Eyo*, limits update messages to the number of actual changes rather than the total number of objects. PRACTI 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.

Several of these systems make use of application-specific resolvers [38, 23], which require developers to

construct stand-alone mechanisms to interpret and resolve conflicts separately from the applications that access that data. *Eyo*'s approach instead embeds resolution logic directly into the applications, which avoids the need to recreate application context in separate resolvers. Presenting version history directly to the applications, instead of just the final state of each conflicting replica, permits applications using *Eyo*'s API to identify the changes made in each branch.

Star Topologies Many cloud-based storage systems provide a traditional filesystem API to devices (e.g., Dropbox [10], MobileMe's iDisk [21], and ZumoDrive [50]) or an application-specific front end atop one of the former systems (e.g., Amazon's Cloud Player [1]). These systems require that the central cloud service store all content in the system, and provides a filesystem API for devices. While these systems provide a centralized location for storing content, they do not enable disconnected updates between devices, or handle metadata about the objects that changes on multiple devices. Other systems such as Amazon's S3 [2], use a lower-level put-get interface, and leave all concurrency choices to the application using it. *Eyo* could use a system like S3 as one repository for object content or for metadata collection snapshots for added durability.

A number of systems build synchronization operations directly into applications so that multiple clients receive updates quickly, such as one.world [19], MobileMe [29], Live Mesh [25] , and Google Gears [17]. In these systems a centralized set of servers hold complete copies of the data collections. Applications, either running on the servers themselves or on individual clients, retrieve a subset of the content. Clients can neither share updates directly nor view complete data collections while disconnected from the central hub.

Point to point synchronization: Point-to-point synchronization protocols such as rsync [48], tra [7], and Unison [3] provide on-demand and efficient replication of directory hierarchies. None of these systems easily extend to a cluster of peer devices, handle partial replicas without extensive hand-written rules, or proactively pass updates when connectivity permits.

Attribute Naming Storage system organization based on queries or attributes rather than strict hierarchical names have been studied in several single-device settings (e.g., Semantic File Systems [15], HAC [18], and hFAD [41]) and multi-device settings (e.g., HomeViews [14]), in addition to optimistic replication systems.

7 Summary

Eyo implements the *device transparency* abstraction, which unifies collections of objects on multiple devices into a single logical collection. To do so, *Eyo* uses a novel storage API that (1) splits application-defined metadata from object content and (2) allows applications to define placement rules. In return for using the new API, *Eyo* provides applications with efficient synchronization of metadata and content over peer-to-peer links. Evaluation of several applications suggests that adopting *Eyo*'s API requires only modest changes, that most conflicting updates can be handled automatically by the applications without user intervention, and that *Eyo*'s storage and bandwidth costs are within the capabilities of typical personal devices.

Acknowledgments

References

[1] CloudPlayer. http://amazon.com/cloudplayer/.

[2] Amazon S3. http://aws.amazon.com/s3/.

[3] S. Balasubramanian and Benjamin C. Pierce. What is a File Synchronizer? In *Proceedings of MobiCom*, 1998.

[4] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proceedings of NSDI*, 2006.

[5] 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 Dept of Computer Science, 2002.

[6] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

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

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

[9] Michael Demmer, Bowei Du, and Eric Brewer. TierStore: A Distributed File-System for Challenged Networks. In *Proceedings of FAST*, 2008.

[10] Dropbox. http://dropbox.com/.

[11] 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.

[12] Alan Fekete, David Gupta, Victor Luchangco, Nancy A. Lynch, and Alexander A. Shvartsman. Eventually-Serializable Data Services. In *Proceedings of PODC*, 1996.

[13] 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 OSDI*, 2006.

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

[15] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic File Systems. In *Proceedings of SOSP*, 1991.

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

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

[18] Burra Gopal and Udi Manber. Integrating Content-based Access Mechanisms with Hierarchical File Systems. In *Proceedings of OSDI*, 1999.

[19] 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.

[20] 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 USENIX Summer*, 1990.

[21] iDisk. http://apple.com/idisk.

[22] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of SOSP*, 1991.

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

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

[25] Live Mesh. http://www.livemesh.com.

[26] 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 EuroSys*, 2009.

[27] Dahlia Malkhi, Lev Novik, and Chris Purcell. P2P Replica Synchronization with Vector Sets. *SIGOPS Oper. Syst. Rev.*, 41(2):68–74, 2007.

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

[29] MobileMe. http://www.apple.com/mobileme/.

[30] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of OSDI*, 2002.

[31] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and Storage Flexibility in the Blue File System. In *Proceedings of OSDI*, 2004.

[32] 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.

[33] Daniel Peek and Jason Flinn. EnsemBlue: Integrating Distributed Storage and Consumer Electronics. In *Proceedings of OSDI*, 2006.

[34] 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 SOSP*, 1997.

[35] Ansley Post, Juan Navarro, Petr Kuznetsov, and Peter Druschel. Autonomous Storage Management for Personal Devices with PodBase. In *Proceedings of Usenix ATC*, 2011.

[36] 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 NSDI*, 2009.

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

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

[39] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of OSDI*, 2002.

[40] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: Semantic Data Management for the Home. In *Proceedings of FAST*, 2009.

[41] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems are Dead. In *Proceedings of HotOS*, 2009.

[42] 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 FAST*, 2002.

[43] http://www.sqlite.org/.

[44] 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 HotStorage*, October 2009.

[45] http://subversion.tigris.org.

[46] 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 SOSP*, 1995.

[47] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An Architecture for Internet Data Transfer. In *Proceedings of NSDI*, 2006.

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

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

[50] ZumoDrive. http://zumodrive.com/.