Personal Storage Clouds from Portable Components

Stanislas Polu and David Mazières
Stanford University

Abstract

A number of applications would seem to benefit from storing data "in the cloud"—that is at application storage providers such as Amazon S3. The cloud model is appealing because it relieves end users from the need to administer storage systems and worry about backups. Furthermore it avoids inconsistencies; for example users who store mail "in the cloud" using services such as Gmail can get an up-to-date consistent view of their mailbox from any PC with Internet connectivity.

However, the cloud model has a dark side. Large storage providers present a massively central point of failure: if everyone depended on a handful of services such as Amazon S3, then an outage could cause untold damage. Furthermore, storage providers must monetize their services, which potentially drives them to act against users' interests by charging fees or, in the case of unencrypted services such as email, inserting advertising or even violating privacy.

We advocate a different model called "personal clouds," in which a user stores data in a reliable cloud of sorts, but the user's cloud is built entirely out that user's own storage devices. We present a network file system, pFS, that realizes this personal cloud model. pFS exploits the fact that with the proliferation of devices, users have more and more storage capacity and redundancy. pFS is designed to keep a loosely consistent file system replicated across a sufficient number of devices that data loss due to hardware failure is highly improbable. Because of its weak consistency model, pFS achieves good performance by updating files locally and propagating the changes to other replicas asynchronously.

1 Introduction

Users have an increasing number of devices with sizable data storage. PDAs, digital audio players, cell phones, and cameras will all soon provide storage capacities comparable to today's laptops. The ability to carry around more storage opens up new opportunities, but also creates complications. First, more devices mean more failures. People are more likely to lose or damage a cell phone or PDA than a desktop machine; yet when cell phones have 60 GB of storage, they will likely contain data the owner cannot afford to lose. Second, more devices mean there are more places any given file may live. Whatever file a user wants is then less likely to be on the device he or she is currently using. Users will thus somehow need to manage their data by transferring files between devices.

One solution to these problems is to store all data "in the cloud"—i.e., to keep the primary copy of data at an external service provider such as Amazon or Google, rather than in a user's own devices. This trend is particularly evident with web mail, which many find preferable to running their own mail servers. Web mail solves the problem of synchronizing saved mail accessed from multiple clients. It also saves end users from dealing with server failures. Web sites such as Google Docs promise a similar experience for a broader range of applications.

Unfortunately, the cloud model has a number of limitations. In many cases, users need Internet connectivity to access their data. Yet devices—even cell phones—do not always have network connectivity. Moreover, the cloud model unnecessarily consumes precious wide-area network bandwidth even when two devices on the same network share files. Worse yet, widespread dependence on a small number of storage providers creates a dangerously large central point of failure. Someone could easily take down Amazon S3 just as a Pakistani ISP recently caused a world-wide blackout of YouTube. Finally, the cloud model requires service providers to recover costs. Some applications, such as email, can achieve this through advertising. Others (such as those based on Amazon S3) require fees to the storage provider. In these cases an alternative that does not require fees would be preferable for users.

Fortunately, two trends suggest an alternative model

that may be preferable for many users. First, the raw storage owned by users is plentiful and growing faster than Internet connection speeds. From 1990 to 2005, the time required to transmit a typical home user's hard disk contents over the wide-area network increased from 0.6 to 120 days [21]. Usable storage capacity of "the cloud" is therefore limited by bandwidth to something much smaller than local storage. Second, as users get more devices, they have access to more failure independence. Though we often lose servers, desktop machines, laptops, cell phones, or PDAs, any single person is unlikely to lose all of their devices simultaneously.

These trends enable a data management model we term *personal clouds*. A personal cloud is a file system in which multiple devices each store a local copy of every file. Modifications propagate between devices opportunistically as network conditions permit. For instance, modifications made on a user's home machine might automatically propagate to her laptop over the local network; at work the next day the laptop could automatically push the changes out to a local server. Though changes also propagate over the wide area network, personal clouds exploit the higher bandwidth of physically transported devices to synchronize far more data than would otherwise be practical. Moreover, should any device fail, its replacement can re-initialize itself from any existing copy.

This paper presents *pFS*, a network file system that implements personal clouds. Unlike traditional file systems, pFS makes all updates locally and propagates them asynchronously to other devices sharing the same file system, ensuring current network conditions do not slow down applications. Moreover, clients of a particular pFS file system store every file. There is no "master" or primary copy of the data. All clients are equal, and any client–client communication patterns are permissible. When concurrent updates occur on different devices, pFS reconciles the changes, and exposes any conflicts for applications to resolve.

pFS is far from the first network file system to support disconnected operation. Indeed, we rely heavily on ideas introduced by previous file systems such as Ficus [17] and CODA [11]. Our contributions are two-fold. First, by eliminating the distinction between clients and servers, pFS provides a new usage model that provides a more viable alternative to the "cloud" model. In particular, pFS eliminates the need for any high-reliability components, and also for any system administrative support. pFS works fine even in situations where no replica has a publicly routable IP address.

Second, pFS capitalizes on hardware trends to gain a large amount of simplicity. The fact that storage is becoming plentiful allows devices to *replicate* file systems rather than *cache* parts of them. Thus, server replication and disconnected operation are one and the same mechanism with pFS, and there is no need for complex server-side logic for such tasks as backup and volume migration. Unlike some previous file systems, pFS handles reconciliation in terms of directory entries, rather than files, which at least in pFS's context seems reduce the number of reconciliation corner cases. pFS is also structured around a single internal function, `pfs_set_entry`, in terms of which all the file system modifications can be straight-forwardly implemented. Thus, while previous file systems required a large amount of engineering effort, pFS is only 10,000 lines of C code and was implemented by a single student in just over three months. Nonetheless, pFS is a useful system providing better performance than NFS (though of course it also provides weaker consistency than NFS).

The next section discusses related work. Section 3 further motivates the "personal cloud" paradigm. Section 4 describes the data structures around which pFS is organized and discusses conflict resolution. Section 5 gives the details of the implementation. Section 6 evaluates the pFS implementation, after which we briefly discuss future work and conclude.

2 Related Work

pFS was inspired by a number of recent commercial products that follow the cloud model. DropBox [8] creates a remote file system stored on Amazon S3 and keeps a copy of it locally on every device involved. Disconnected operation is permitted, but conflict detection must take place on DropBox's servers. Moreover, update propagation must also go through a remote, centralized infrastructure, preventing devices from leveraging fast local-area network bandwidth or other communication links such as Bluetooth or USB. The same approach is used by .Mac [9]. Live Mesh [10] also appears to have a similar architecture to these other products, though unfortunately we don't have enough details on its implementation. However, the fact that it limits the space available on a "mesh" suggests that Microsoft's central storage infrastructure or client network bandwidth may be a limiting factor.

The goal of pFS is to provide a similar user experience to these services, yet without relying on a centralized infrastructure. Thus, users can make full use of the storage capacity and network bandwidth supported by their devices, without paying any fees, viewing any advertising, losing any privacy, or unnecessarily subjecting themselves to storage quotas. Our thesis is that not only can many of the well-known concepts in distributed file systems be readily adapted to disintermediate these central services, but also that hardware trends and the

specific usage model allow for significant design simplifications compared to previous work.

The most relevant previous file systems are Ficus [17] and CODA [11], both of which support replication of servers and disconnected operation. Compared to these systems, pFS achieves simplicity by eliminating the distinction between clients and servers, eliminating the need for caching and hoarding (using only replication), assuming only a modest number of clients (on the order of a dozen), eliminating a synchronous/consistent mode of operation, exposing conflicts in such a way that they can be resolved by software entirely outside of the file system, and assuming that reconciliation can happen over high-bandwidth links by transporting devices (thereby eliminating the need for techniques such as trickle reintegration). pFS also organizes its representation of the file system around directory entries rather than files, which avoids the need to handle update/update conflicts and name conflicts separately [20].

BlueFS [16] and a follow-on project EnsemBlue [16] are both file systems targeted at mobile devices. These projects also address the problems of cache and power management, performance optimization in the face of different storage media, and namespace management, none of which pFS specifically addresses. Like pFS, BlueFS takes advantage of users transporting storage devices to move data directly between pairs of clients. Unlike pFS, BlueFS still requires a centralized server.

Other peer-to-peer file systems have been proposed with no central server, notably Ivy [15]. However, a big difference between pFS and Ivy is that pFS replicates a file system only on devices belonging to a particular user, whereas Ivy spreads the data over potentially huge numbers of machines owned by different people.

A closer analog to pFS's peer-to-peer architecture is distributed revision control systems such as BitKeeper and git [13]. These systems have an egalitarian structure that eliminates the concept of a "master" repository—all repositories are equal, and can synchronize with each other pairwise. However, while repositories store complete modification histories, pFS only retains enough recent versions of each file to ensure the "no lost updates" property. In this sense, pFS is closer to directory mirroring tools such as Unison [1], Rumor [6], and Tra [4]. However, as a file system, pFS has the advantage of knowing exactly what files are being are being modified, and can queue changes up for transmission immediately rather than needing to scan the whole file system periodically.

pFS is related to several other distributed systems that are not file systems. Bayou [19] was designed to handle concurrent updates to objects in a disconnected fashion, but was explicitly intended not to be a file system so as to force applications to provide conflict resolution code. PRACTI [2] provides topology independence like pFS, but also has such features as partial replication which pFS doesn't need. Finally peer-to-peer backup systems such as Pastiche [3] [24] address the backup problem in a different way from pFS, by allowing people to use each others' spare disk space for backup.

3 Personal Clouds

Our thesis is that mobile devices (PDAs, smartphones, digital audio players, etc.) with connectivity (WIFI, BlueTooth, USB, etc.), carried by users wherever they go, can be used as a substitute to an always available central infrastructure for update propagation. A key observation is that there is more connectivity *among* a user's devices, particularly her mobile devices, than there is *between* a user and any plausible central infrastructure. This connectivity is both in terms of frequency (many users are often out of reach of the "global Internet" but are not often out of reach of their cell phone) and in terms of magnitude by taking advantage of the physically transported devices, but also relying on local connectivity instead of the WAN when possible.

Based on this observation, one of our main technical contributions is the design of a flexible versioning system in which a central server is absent. There is no authoritative copy in this system. Instead, any device can propagate updates to any other, as network conditions permit, and conflicts can be resolved on any participating device.

We then use this versioning system to build pFS, a distributed file system in which data is stored on every device and propagated asynchronously and seamlessly. pFS yields two benefits. First, as with other systems, it addresses the "synchronization problem": it provides a user a consistent view of her data, which she can modify from any of her devices. Second, because of the sheer multiplicity of personal devices, pFS can provide many of the benefits of "cloud computing" (including durability and availability, both achievable through replication on multiple personal devices), without involving a central server.

3.1 Usage Case

We now walk through a typical case of the "personal cloud" infrastructure as we envison it. User *X* finishes editing his holiday video at home. The video is stored on his desktop computer, but also pushed to his digital player via USB. The next day, in the train to work, without connectivity, he makes a few changes on his laptop to his quarterly report. The update is propagated via bluetooth to his cell phone. Arriving at work, he plugs in his media player, and enjoys the video on a widescreen with his colleagues, while the updates he made on the train are

seamlessly uploaded via bluetooth from his cell phone to his computer at work. He can then work locally on the latest copy of his report without even booting his laptop. While his son is using the desktop computer at home to play a game after school, the updates he made during the day are automatically fetched from his computer at work. When *X* gets back home, he is able to keep working on an up-to-date copy of his data even if he forgot his laptop.

3.2 Caveats to this Vision

First, as regards remote computing, personal clouds *complement* this function. The principles we espouse are compatible with the integration of remote servers as nodes in one's personal cloud for providing services that are not available on conventional devices, such as web-based access and modification of data.

Second, we must attend to confidentiality. The reason is that, under pFS, a user's data is not only more available (by definition, people carry around mobile devices), it also more likely to fall into someone else's hands (people drop and lose their mobile devices). If pFS is used to transport highly sensitive data on cell phones, the cell phone should really stored the data in encrypted format, which we have not yet implemented and could complicate synchronization.

3.3 Model

In the following sections, we assume that mobile devices as smartphones and digital players have enough storage capacity for holding all of a user's data. This might become true in a near future but is certainly not today. In the mean time, the software easily supports creating multiple file systems. Thus, one might create a large file system that is replicated on a home machine, work machine, and laptop, and a smaller file system that is replicated on a user's cell phone as well as these three machines.

4 Versioning

This section describes pFS's model for reconciling different versions of files when receiving updates asynchronously.

4.1 Terminology

We use the term *device* to mean any hardware running the pFS software. Though pFS currently only runs on Unix machines, we envision porting it to other types of hardware such as PDAs, cell phones, and digital audio players. Users must supply a unique symbolic name for each device on which they use pFS (e.g., `market.scs.stanford.edu`, or `stan-cell-phone`).

A pFS *file system* is a directory tree replicated on one or more devices. Each file system has a unique ID, generated randomly at creation time. We use the term *replica* to denote a copy of a particular file system stored by a device. Note that one device may store replicas of multiple file systems.

Each replica is also owned by a particular *user*. Though all replicas on a portable device would typically be owned by the same user, this need not be the case on a Unix machine where multiple users each have accounts they use to access different pFS file systems. Every replica has a unique name, given by the tuple $\langle \text{file system ID}, \text{device}, \text{user} \rangle$.

Each file system has metadata consisting of an access control list and a set of replicas known to be storing the file system. The access control list specifies the users allowed to create and synchronize replicas of the file system. Each user specified can have read-only or read-write access to the file system. Though currently users are named by opaque random IDs, eventually we envision naming users by public keys so as to simplify authentication of device owners.

4.2 Disconnected Operation

pFS makes no assumption concerning network partitioning and connectivity. We model connectivity as connections happening opportunistically between two replicas of the same file system. When such connections occur, the two replicas send each other all data and metadata necessary to synchronize their replicas of the file system. With the current prototype, connections occur when two nodes can reach each other over the Internet, but the software is structured such that we can easily add other types of connection, such as USB and Bluetooth connections.

Because pFS is essentially permanently in disconnected operation mode, it must detect and resolve conflicts when multiple replicas of the same file system are modified between synchronization events. As with previous file systems, pFS enforces the "no lost updates" guarantee [17]—in other words, when reconciling replicas, pFS will only discard a version of a file if that version's modification history is a prefix of another version it is not discarding.

Rather than version files and directories, pFS versions directory entries. This has two benefits. First, it simplifies the handling of cases where the same file name has been concurrently used for a file on one replica and a directory on another. Second, it allows us to simplify the implementation, so that at its core pFS only really has one complex function, `pfs_set_entry` (discussed in Subsection 4.6), and the rest of the sys-

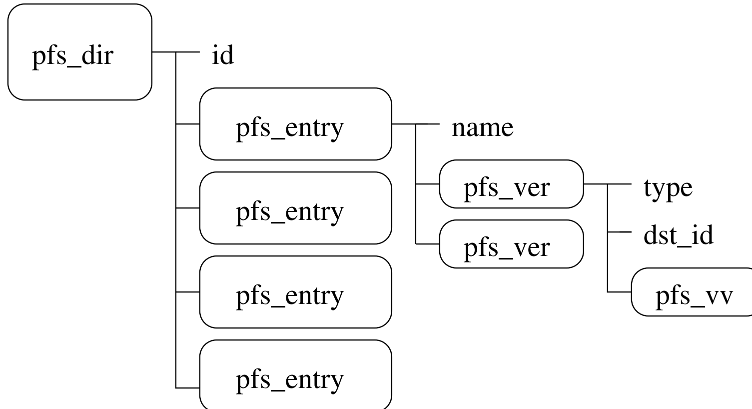

Figure 1: Schematic view of PFS data structures

tem simply involves translating file system calls into `pfs_set_entry` calls.

4.3 Data structures

pFS assigns each file and directory a globally unique ID generated upon creation by hashing the device ID and a counter. File IDs change whenever a file changes. Essentially a file ID identifies an immutable file version. By contrast, directories keep the same ID even when modified. Otherwise, modifying any file would require changing all directories up to the root. Intuitively, the choice of mutable directory entries also makes sense because pFS reconciles the meaning of individual file names with a directories, whereas concurrent updates to the same file are always considered a conflict.

Figure 1 shows the basic data structures making up a directory tree, and Figure reffig:memstruct shows the corresponding data structures. Each directory is represented by a `pfs_dir` data structure, containing the directory's unique ID and a set of `pfs_entry` structures. Each `pfs_entry` maps a file name onto a set of `pfs_ver` structures. When no conflicting updates have been made to a file, the directory entry has only one `pfs_ver` structure. However, when reconciling two replicas with different versions of the file, if one does not supersede (i.e., reflect the entire modification history of) the other, the `pfs_entry` structure includes a list of all versions required to preserve the "no lost update" guarantee.

Each `pfs_ver` contains the file type (regular, directory, or symbolic link), the "destination" ID of the file contents, and finally a version vector [18]. When the entry type is a regular file or symbolic link, the destination ID points to an immutable blob of data. When the entry type is a directory, the destination is the ID of another `pfs_dir` structure. A version vector contains a monotonically increasing version number for each replica that has modified the directory entry.

If v is a version vector and r is a replica, we write $v[r]$ to designate the version number assigned by r the last time r modified the resource pointed to by this directory entry. We say that a version vector v_1 dominates v_2 (written $v_1 \geq v_2$) iff $\forall r, v_1[r] \geq v_2[r]$. Update conflicts are detected when an entry contains two versions with unordered version vectors—i.e., $v_1 \not\geq v_2$ and $v_2 \not\geq v_1$. Conversely, when one version's vector dominates another, it is safe to discard the one whose version vector is dominated. These rules are fairly standard for systems employing version vectors. However, we note that some file systems, such as Coda, require more complex version vectors because of the distinction between replicas and clients [23].

4.4 Conflicts

When a conflict is detected, pFS does not attempt to resolve it automatically. Rather, it attempts to expose the conflict to users or applications by generating two different file names for the two different versions of the entry. When a conflict is detected for an entry `filename`, two (or more) names pointing to two different ressources are listed:

$$\text{replica}_1\texttt{:filename}$$
$$\text{replica}_2\texttt{:filename}$$

Where replica_1 and replica_2 are the names of the two replicas that created the conflicting directory entries. pFS currently reserves the character "`:`" to avoid confusion between normal and conflicted files.

There has been much work on automatically merging concurrent changes to the same file. For instance, Coda has a notion of application-specific resolvers (ASRs) [12] that are fired off automatically based

```
struct pfs_dir
{
  char id [PFS_ID_LEN];
  uint32_t entry_cnt;
  struct pfs_entry ** entry;
};
struct pfs_entry
{
  char name [PFS_NAME_LEN];
  uint32_t main_idx;
  uint32_t ver_cnt;
  struct pfs_ver ** ver;
};
enum pfs_entry_type {
  PFS_DEL = 0,
  PFS_FIL = 1,
  PFS_DIR = 2,
  PFS_SML = 3
};
struct pfs_ver
{
  uint8_t type;
  char dst_id [PFS_ID_LEN];
  struct pfs_vv * vv;
};
struct pfs_vv
{
  char last_updt [PFS_ID_LEN];
  uint32_t len;
  char ** sd_id;
  uint32_t * value;
};
```

Figure 2: In-memory data structures for directories, directory entries, and version vectors.

on configuration files and the types of files with conflicts. ASRs require support from the file system in part because conflicts may be detected at servers, yet must be resolved at clients, and in part because ASRs run with a special view of the file system allowing them to see conflicted files as directories containing multiple versions. Both of these problems are alleviated in pFS by eliminating the distinction between clients and servers, and exposing all versions of conflicted files to all users (by reserving a special character for file names).

When a user (or an application) has resolved a conflict, it must signal to pFS that a new version of the file now supersedes one of the other `pfs_ver` structures corresponding to that directory entry. Users can do this with a utility:

`resolve` *old-version file*

`resolve` tells pFS that *file* now contains all the information needed from *old-version*, so that it is possible to garbage collect *old-version* without violating the "no lost updates" guarantee. pFS handles this request internally by setting the version vector of file v_f to the element-wise maximum of its old value and *old-file*'s version vector, v_o:

$$\textbf{foreach}\ r : v_f[r] \leftarrow \max(v_f[r], v_o[r])$$

The mechanism for requesting that pFS update a version vector is simply to make the `link` system call on file names containing the special reserved "`:`" character. The actual merging of data into *file* is also done with ordinary file system calls—namely reading *old-version* and overwriting *file*. Thus, no special interfaces are required to resolve conflicts. In fact, `resolve` is nothing but a tiny shell script calling `ln`. This also means that applications aware of pFS can do their own conflict resolution. In fact, a mechanism similar to Coda's ASRs could be implemented entirely outside of the file system (except users would have to launched it manually on a directory tree after a synchronization event).

4.5 Main Version

When multiple conflicting versions of a file exist, pFS running on any given replica must choose one as the *main version* to show under the regular file name, while all other versions appear with a "replica`:`" prefix. The replica included in the file name is the one that last updated the file version, as determined by the `last_updt` field in the version vector. Note that only the main version of a file is writable; all other versions are read only.

pFS chooses the main entry by defining a total order on version vectors, $>_r$, which is specific to each replica r. The main version of a file is the one whose version vector is maximal under $>_r$. We say $v_1 >_r v_2$ iff:

1. $v_1 \geq v_2$, or
2. $v_1[r] > v_2[r]$, or
3. $v_1 \not\geq v_2$ and $v_2 \not\geq v_1$ and $v_1[r] = v_2[r]$ and the sum of all entries in v_1 is greater than the sum of all entries in v_2, or the sums are equal but $h(v_1) > h(v_2)$ for some function h that normalizes the version vectors and outputs an integer.

Consider two versions of a directory entry, f_1 and f_2, with associated version vectors v_1 and v_2, respectively. Condition 1 simply states that if f_1 supersedes f_2 (as evidenced by the fact that v_1 dominates v_2 in the partial order), then f_1 will be preferred as the main version. In fact, f_2 can safely be garbage collected without violating the "no lost updates" rule.

Condition 2 says that f_1 contains at least one modification made by replica r that is not reflected in the update history of f_2. The goal in choosing f_1 is to ensure

op#	*replica*	*operation*	r_1		r_2		r_3	
1	r_1	**create** f	$\langle 1,0,0\rangle$					
2	$r_1 \leftrightarrow r_2$	**sync**	$\langle 1,0,0\rangle$		$\langle 1,0,0\rangle$			
3	r_2	**write** f			$\langle 1,1,0\rangle$			
4	r_1	**write** f	$\langle 2,0,0\rangle$					
5	$r_2 \leftrightarrow r_3$	**sync**			$\langle 1,1,0\rangle$		$\langle 1,1,0\rangle$	
6	$r_1 \leftrightarrow r_3$	**sync**	$\langle 2,0,0\rangle$	$\langle 1,1,0\rangle$			$\langle 1,1,0\rangle$	$\langle 2,0,0\rangle$
7	r_3	**write** f					$\langle 1,1,1\rangle$	$\langle 2,0,0\rangle$
8	$r_2 \leftrightarrow r_3$	**sync**			$\langle 1,1,1\rangle$	$\langle 2,0,0\rangle$	$\langle 1,1,1\rangle$	$\langle 2,0,0\rangle$
9	r_2	**resolve** $f_{\langle 2,0,0\rangle} \rightarrow f$			$\langle 2,2,1\rangle$			
10	$r_1 \leftrightarrow r_2$	**sync**	$\langle 2,2,1\rangle$		$\langle 2,2,1\rangle$			
11	$r_1 \leftrightarrow r_3$	**sync**	$\langle 2,2,1\rangle$				$\langle 2,2,1\rangle$	

Figure 3: Evolution of version vectors for a file f modified at three different replicas. When a replica has multiple versions of the file, we list the main version first. Any versions not listed can be garbage collected. Note that h arbitrarily orders $h(\langle 1,1,0\rangle) > h(\langle 2,0,0\rangle)$, which determines the main version chosen by r_3 n step 6.

that the main version of a file seen on replica r always reflects all modifications made by r. This guarantee also rests on the fact that only the main version of a file is ever writable, so that a replica can never create a version of a file that does not include all of its own previous updates to that file. Condition 3 is simply arbitrary to make $>_r$ a total order.

Since only the main version of a file is writable, users must reconcile conflicting updates pairwise between individual divergent versions (containing "`:`") and the main version. Once users have incorporated changes from a "`:`" file, they call `resolve` on the old version, which issues the system call

$$\textbf{link}\ (\mathrm{replica}_j : f, f).$$

As previously discussed, this updates the version vector of the main version to supersede that of $\mathrm{replica}_j$'s last version, so the "`:`" file can be garbage collected.

Note that even though only the main version of a file is writable, "`:`" directories are always writable. This stems from the fact that directories do not change ID when modified. Section 4.7 discusses directory conflicts in more detail.

Figure 3 illustrates the evolution and ordering of version vectors with an example. In step 6, r_3 has two conflicting version of file f. Since r_3 has made no modifications to either version, it arbitrarily (with h) picks $\langle 1,1,0\rangle$ as the primary version. The user on r_3 then writes to this version, but without resolving it, so that both versions still exist when it syncs with r_2. The user on r_2 then invokes `resolve` to indicate its main version $\langle 1,1,1\rangle$ should supersede the other version in step 9. When this propagates to the other two replicas the delete the superseded versions of f.

4.6 `pfs_set_entry`

A nice property of this versionning system is that all modifications made through the traditional Unix file system interface can be straight-forwardly translated into invocations of a single function, `pfs_set_entry`. `pfs_set_entry` is the atomic update operation that will be propagated among the replicas of a file system. pFS logs these operations, and replaying them sequentially reconstructs the associated file system structures. When synchronizing with another replica, the updates are applied and can be simply appended to a log for further propagation to other replicas. However, entries with whose version vectors are dominated by other (later) versions never need to be propagated to other replicas.

The function's prototype is as follows:

```
int pfs_set_entry
        (struct pfs_instance *pfs,
         const char *file_system,
         const char *dir_id,
         const char *name,
         const struct pfs_ver *ver,
         const bool_t reclaim);
```

`pfs_set_entry` adds version `ver` (see Figure 2 for a declaration of `pfs_ver` structure) to the entry `name` in the directory designated by `dir_id` in the file system designated by `file_system`. Note that the directory structure does not have a name, but is simply designated by its `dir_id`. (The directory's name is mapped to `dir_id` in the parent directory, and the root directory ID is stored in the file system metadata.)

Recall that a `pfs_ver` structure has a `dst_id` field that references a blob of data (for regular files and symbolic links), or another directory. The `reclaim` argument simply specifies whether or not the object referenced by the previous value of `dst_id` can be garbage

collected (which it usually can, except with rename where the same `dst_id` field will be inserted into the destination directory entry).

Let us show a few examples of how file system operations map to `pfs_set_entry` operations. For simplicity, we ignore the `pfs` and `file_system` arguments, which are identical for all calls within a file system. We also show `pfs_ver` structures within braces, and omit the version vector.

- **open (*f*, O_CREAT, 0666)**:
  generate a new back-end storage file (blob) and *id*
  `pfs_set_entry` $(dir_id, f,$ {`REG`, *id*}, `1`)
- **close (*f*)** when *f* is dirty:
  regenerate back-end storage id'
  `pfs_set_entry` $(dir_id, f,$ {`REG`, id'}, `1`)
- **unlink (*f*)**:
  `pfs_set_entry` $(dir_id, f,$ {`DEL, 0`}, `1`)
- **rmdir (*d*)**:
  `pfs_set_entry` $(dir_id, d,$ {`DEL, 0`}, `1`)
- **rename (f_1, f_2)**:
  `pfs_set_entry` $(dir_id_2, f_2,$ {`REG`, *id*}, `1`)
  `pfs_set_entry` $(dir_id_1, f_1,$ {`DEL, 0`}, `0`)

We have not shown the version vectors used here. They are generated or incremented according to the rules we defined previously. Note that a deleted entry persists in the system with a `pfs_ver` structure of type `DEL`. This conservative technique is needed to handle cases where an entry might have been deleted on one replica and updated on another. The `DEL` type has to be treated as any other type of entry by the versionning system. In the future, we may consider vector time pairs [4] to alleviate the need for tracking deleted files.

4.7 Directory conflicts

One tricky scenario arises when a directory entry of type directory conflicts with another entry (which can be of any type—`DEL`, `REG`, or even `DIR` if a directory has been deleted and re-created). Consider the case in which a file system exists with the file `d/f`. Suppose replica r_1 deletes file `d/f`, then deletes directory `d`, while replica r_2 simultaneously modifies file `d/f`. When the two sync, what the user will see is that the deleted directory will re-appear on r_1 under the name r_2:`d`, at which point the user can re-delete r_2:`d/f` and resolve the directory. How this happens internally is a bit more subtle, and relies on two behaviors we have not yet explained.

Suppose a replica is receiving updates from another replica. If the sender provides an update that deletes a directory that is not empty on the receiver, then the receiver immediately re-creates the directory, with the same directory ID and using a version vector that supersedes the delete operation. If, conversely, the receiver gets a `pfs_set_entry` update whose `dir_id` it does not know, it immediately generates a new, dangling `pfs_dir` structure which is not reachable from the local root directory, but at least allows the synchronization operation to continue.

Revisiting the previous example, when r_1 sends its updates to r_2, upon receiving the `pfs_set_entry` that removes directory `d`, r_2 will re-create the directory by appending a new `pfs_set_entry` to its own log. When r_2 pushes its updates to r_1, r_1 will receive an update to file `f` in directory `d`, but will not know that this is directory `d` because it will not recognize the `dir_id`. Therefore r_1 simply creates a dangling directory into which to place the new version of `f`. Later on, r_2 will push the directory re-creation to r_1, and r_1 will attach the dangling directory back into the appropriate parent directory under the name `d`.

After deleting `d` and before synchronizing with r_2, r_1 might instead have created a regular file, symbolic link, or new directory with the same name but a different `dir_id`. All these cases are handled the same, and will result in a directory r_2:`d` appearing on r_1.

We note that a directory must be empty to be resolved. Thus, the user on r_1 must delete all the files within r_2:`d` or move them to a new directory before r_2:`d` can be superseded and garbage-collected.

5 Implementation

Our implementation of pFS relies on Fuse [7] to expose a mounted file system to users. All the code runs in user-space and stores the state of *replicas* in a directory on the device native file system. We implemented pFS as a two level architecture (Figure 4 depicts these two levels on top of the back-end storage). The main components of our implementation are as follows:

- **Back-end storage**: a directory on the local file system containing all data and metadata used and generated by pFS.
- **libpfs**: implements the core functionality of pFS; this module exports a POSIX-like file system interface along with extra functionality specific to pFS.
- **Fuse stub**: exposes pFS to the user as a file system using Fuse. It directly translates the calls received from Fuse to the libpfs interface.
- **pfsd**: a daemon that is responsible for propagating updates using any communication channel available and applying updates received from other devices.

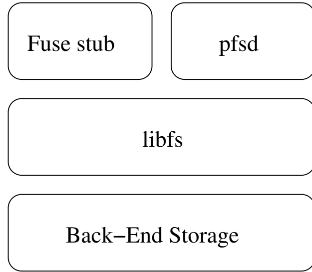

Figure 4: pFS Implementation as a two level architecture based on the device local file system, which is labelled "Back-End Storage" in the figure.

> pfsd receives callbacks from libpfs whenever an update is made locally, and uses the libpfs API to commit updates received from remote devices on the local *replica*.

We designed the core component of pFS as a library in order to provide an easy-to-use API for taking advantage of our versionning system, that is independent of the communication channels used to propagate the updates between the different devices. pFS has been developped in C and its implementation is only 10,000 lines of code.

5.1 libpfs

The pFS file system exposed to the user through Fuse is layed out as follows: the root directory contains a list of directories representing each *file system* in which the device is involved.

Since libpfs is structured as a library, all functions get a `pfs_instance` structure as argument. This structure contains the path of the back-end storage and the list of currently opened files. libpfs exports a POSIX-like file system API: `pfs_open`, `pfs_close`, `pfs_pwrite`, `pfs_unlink`... These calls get absolute paths as argument for files and directories, mainly for compability with the Fuse API which uses absolute paths from the root of the mounted file system.

Therefore, we need to walk down the chain of `pfs_dir` structures anytime an operation has to be made on a file that has no associated file handler. Since `pfs_dir` structures are stored in files contained in the back-end storage, we have implemented a cache to walk theses structures in memory. The cache is a simple LRU cache implemented with a hash table, mapping `pfs_dir` IDs to their actual in-memory data structures. It does not need to be large since modifications made inside a directory result in many requests to the same `pfs_dir` structures. In our implementation we are caching 64 `pfs_dir` structures, a pointer to the cache is stored in the `pfs_instance` structure. libpfs provides a `pfs_sync_cache` function for periodically writing back the cache to disk (every 10s in our implementation of pfsd). The cache has incurred an important improvement over the performances of libpfs.

Local file operations: When a file is created, a new destination ID is generated. A file named after this ID is created on the back-end storage and a `pfs_set_entry` call is issued to insert the new `pfs_ver` into its parent `pfs_dir` structure present or fetched in the cache. Opening a file for reading does not need to update any versioning information and is directly mapped to an `open` call on the back-end storage file.

Opening a file for writing incurs more overhead since it necessitates the generation of a new destination ID. This new mapping is needed as soon as the `pfs_open` call is issued since the actual ID for the file might be revoked if an update containing a dominating version is received while the file is edited. The `pfs_ver` associated with the newly generated ID is inserted into its parent `pfs_dir` structure when the file is subsequently closed. Therefore opening a file for writing incurs a non-negligible overhead and maps to two operations on the back-end storage: a `link` and an `open` call.

Merging a version into the local main version, or deleting a file, is translated into a call to `pfs_set_entry`, and an `unlink` call on the back-end storage to revoke the version that has been superseded.

Finally, an `fsync` call flushes the cache to disk and maps to the associated `fsync` call on the back-end storage file.

Local directory operations: Directory creations are mapped to the creation of a file on the back-end storage, the insertion of a `pfs_dir` structure in the cache and the update of the parent `pfs_dir`. Directory removals are mapped to an `unlink` call on the back-end storage, the removal of an entry in the cache and the update of the parent `pfs_dir`. Therefore, directory operations incur a slight overhead by mapping directory operations to back-end storage files operations.

libpfs also provides interfaces used for two-way communication of updates with pfsd. Functions can be registered by pfsd to be called any time a `pfs_set_entry` is issued. The argument passed to those callbacks is a `pfs_updt` structure that contains all the arguments needed to issue the same `pfs_set_entry` on another device; namely, the ID of the *file system*, the ID of the parent `pfs_dir` structure, the name of the file and the new `pfs_ver` structure. The updates received by pfsd

from the network can be applied to the local copy of pFS by calling pfs_set_entry directly.

Finally, libpfs exports a pfs_create_fs function for creating a new *file system*. The function creates a new directory /new_fs on the root of the pFS tree, and generates a new pfs_dir structure representing the root of new_fs. libpfs looks for the list of *replicas* participating in any given *file system* in the file /new_fs/.pfs_replica. Adding and removing *replicas* is delegated to pfsd.

5.2 pfsd

We have implemented a prototype of pfsd using IP as a transport layer. We used a simple protocol allowing the system to send updates from one device to another and retransmit associated files contents. To allow the devices to discover each other and retrieve their respective IPs, we designed a central naming service with which they register when connected to the network.

pfsd still lacks some functionality; it does not optimize back-end storage file transfers: the entire content of a modified file is retransmitted as its ID is regenerated. This is clearly a mechanism that can be optimized as stated in section 7. pfsd also does not currently provide automated management of *file systems*' *replicas*. The devices participating in a *file system* are specified at creation: dynamic addition of *replicas* is not yet supported. Adding a replica is still possible by archiving a back-end storage copy of an existing replica, copying it to the device that has to be added, and modifying the local configuration files.

pfsd keeps a log of the updates it receives from the local instance of libpfs and from other devices. The log entries are pfs_updt structures augmented with a propagation vector. This vector is a list of booleans representing wether or not the given update has been propagated to a *replica*. Updates to this propagation vector are retransmitted between the devices until it is equal to vector 1. When this is the case, the log entry is deleted to avoid wasting storage capacity. Log entries are also elided when their associated versions are superseded by updates received from the local *replica* or remote devices. pfsd opportunistically attempts to propagate any unpropagated updates between two devices when permitted by the network.

pfsd has been designed to be resilient to brutal disconnections that frequently occur on laptop computers. The flexibility provided by the structure of the updates helped a lot in this case, since update transmissions are not dependent on each others in the context of libpfs. After two instances of pfsd get disconnected, when connectivity is revived, pfsd simply restarts its sequential propagation of the updates stored in its log.

6 Evaluation

6.1 libpfs

This section presents an evaluation of pFS local performance. Our goal is to show that, locally, pFS provides a seamless user experience, updates being propagated aynchronously. We compare the performances of pFS, ext3, Fuse only (We implemented a trivial Fuse based file system replicating all calls directly to the local file system), and NFS (Protocol v.3 over LAN with default settings) over a set of microbenchmarks. The machine used is a 2.4 GHz Dual-Core Intel Xeon, with 2GB of RAM and a Gibabit ethernet controller, running Ubuntu Server 8.04 distribution. The microbenchmarks we used are the ones used for the evaluation of LFS [22]. The small file benchmark (LFS_S, Figure 5) consists of the creation of 10,000 small (4096 bytes) files (LFS_SC) in 100 different directories, the access for reading of those files (LFS_SR), and finally their deletion (LFS_SD). We augmented the small file benchmark with a small file write benchmark (LFS_SW), which opens every file for writing, truncates it, rewrites the 4096 bytes and calls fsync. We added this benchmark to illustrate the case where libpfs incurs the most overhead (extra link operation for versioning, and cache flushing when syncing to disk), as described in section 5. The large file benchmark (LFS_L, Firgure 6) consists of writing sequentially 30,000 blocks of 4096 bytes (LFS_LSW), reading them sequentially (LFS_LSR), re-writing them randomly (LFS_LRW) and finally re-reading them randomly (LFS_LRR).

The benchmarks are synthetic, and do not represent realistic workloads. The goal is to show the strengths and weaknesses of libpfs. Figure 6 shows, as expected, that versioning maintenance does not incur any visible overhead compared to Fuse for large files. pFS is substantially faster than NFS on every benchmark except the small write benchmark (Figure 5). This benchmark emphasizes the operation that incurs the most overhead due to versionning maintenance: pFS performs comparatively to NFS on this benchmark where it is three times slower than native ext3.

We stress that pFS does not provide the same semantics as NFS, since NFS has close-to-open consistency. pFS is not intended as replacement for NFS, which solves a very different need, and these results should not be interpreted as claiming that pFS is "better" than NFS. However, NFS is a popular file system in daily use by many people, and pFS's ability to beat NFS's performance indicates that performance should not present a barrier to pFS's adoption.

Figure 7 shows the performance of pFS (in the same conditions as previously) on a macrobenchmark consisting of untaring, configuring, compiling and cleaning

	untar	*configure*	*compile*	*clean*	*end-to-end*
ext3	11.0	333.5	562.8	4.6	911.8
pFS	33.2	351.9	581.1	10.6	976.9
NFS	167.8	662.1	826.0	66.1	1722.1

Figure 7: Macrobenchmarks. untar, configure, compile and clean of gzip-1.2.4. All values are in ms.

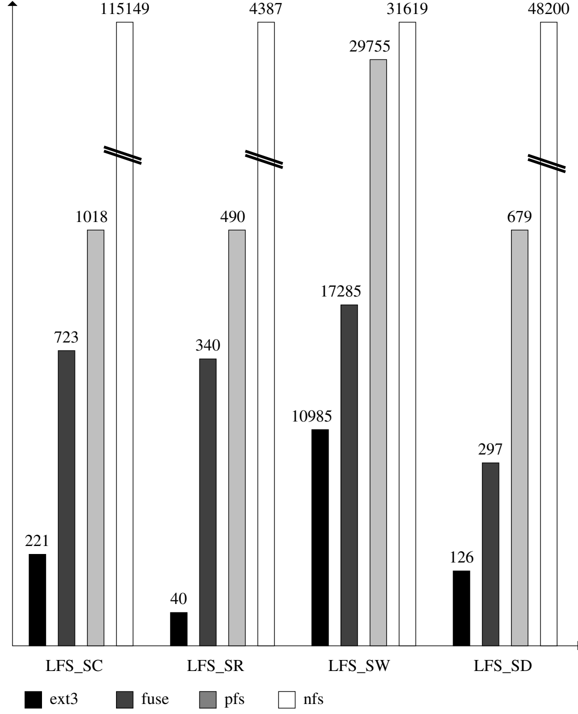

Figure 5: LFS_S benchmark : creation (LFS_SC), reading (LFS_SR), writing (LFS_SW) and deletion (LFS_SD) of 10,000 small files in 100 directories. All values are in ms.

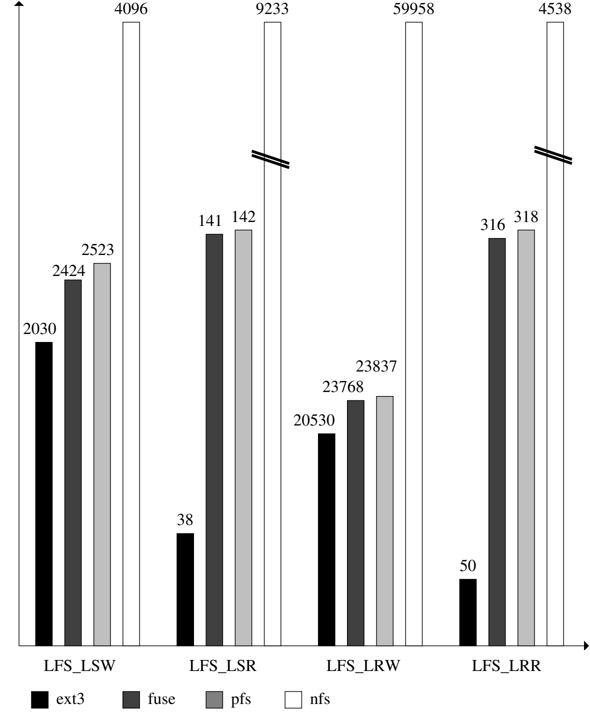

Figure 6: LFS_L benchmark : senquential writing (LFS_LSW), sequential reading (LFS_LSR), random writing (LFS_LRW), random reading (LFS_LRR) of a large file. All values are in ms.

gzip-1.2.4. pFS incurs a 7% overhead over ext3 (while NFS incurs a 88% overhead). This suggests that end-to-end application performance on pFS should be perfectly acceptable for day-to-day computing.

6.2 pfsd

In this section we evaluate the bandwidth consumption of pFS and the overhead due to the propagation of the versioning information. We sequentially runned the LFS_SC, LFS_SR, LFS_SW, and LFS_SD microbenchmarks and mesured the bandwidth usage they incurred on an NFS client, and a pFS *replica* participating in a *file system* with only one other device. [1]

Figure 8 shows that the simplicity of our prototype's protocol incurs a limited overhead to the actual size of the 10.000 files (cumulating a total 39 MB). pfsd retransmits the whole content of every file during the LFS_SW benchmark, which could be drastically optimized, especially since LFS_SW rewrites the same content as LFS_SC. The total bandwidth usage is slightly higher for the LFS_SC benchmark, due to the 100 directories creation wich totalizes aproximatively 50 KB of bandwidth consumption. Obviously, if the updates where to be propagated to more than one device, those number would be accordinlgy demultiplied. This experiment illustrates the fact that the versioning information generated by pFS does not incur a large overhead compared to

[1] Disclaimer : we used the kernel driver byte count to evaluate the bandwidth consumption. The results may contain noise that we tried to limit as much as possible (results issued from a more controlled experiment will be available in a near future).

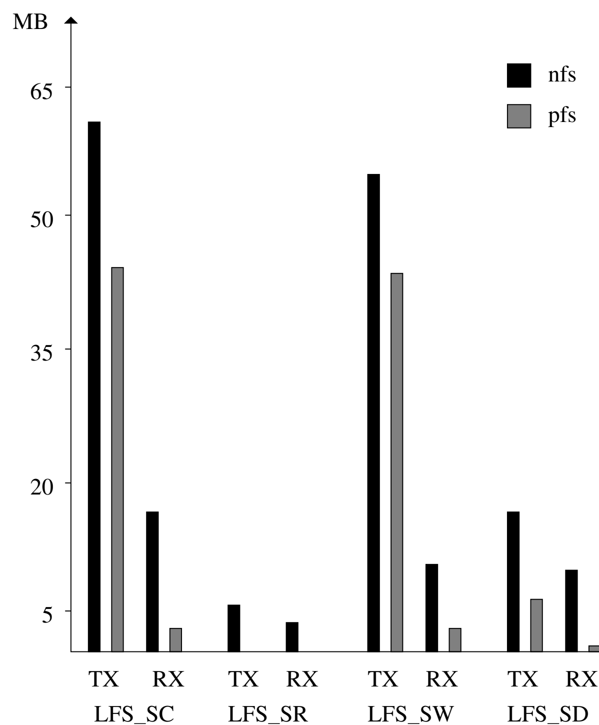

Figure 8: Bandwidth usage incurred by the LFS_SC, LFS_SR, LFS_SW, and LFS_SD microbenchmarks.

NFS, even when updating a large number of small files.

Even if our prototype of pfsd is not yet optimized for bandwitdth consumption, it already allowed us to experience the benefits of our serverless personal cloud model and the ease of use of libpfs' versioning system. WAN and LAN communications over IP being achieved at virtually no cost, pfsd opportunistically communicates updates resulting in an efficient propagation of the state among the devices. pFS is transparent to use, does not require any maintenance once the devices are configured, and conflicts rarely occur on a relatively well connected set of devices. We believe that it would be really difficult to create conflicts if cell phones and media players were to be used as simple relays, even with a small capacity and a simple LRU cache for updates.

6.3 Hidden Costs

There are a few hidden costs associated with the use of pFS. First, the update log used by pfsd : even if they are deleted once propagated to every devices or superseded, this log may grow considerably should a device remain unused for a long period of time without being removed from the system. "Rumor Mongering" technique as described in [5] could be used to alleviate this issue.

Second, libpfs keeps track of deleted files to avoid ambiguity in face of resources creation and removal. Such deleted entries are only kept for directories that are still accessible on a replica which limits their number. Nevertheless, such entries can be garbage collected. A solution to this issue has already been provided in [17] under the section : "Insert/Delete ambiguity".

7 Discussion and Future Work

There is no final design for pfsd since it is highly dependent on which type of device and which type of communication channel it relies on. This section is a survey of lessons we have learned from designing pfsd and directives we have identified to iterate on our implementation.

Naming service In the context of IP a naming service is necessary to enable the devices to discover each others' address from their unique names. We used a central server, but DNS could also be used as a naming service if public IPs are available. A naming service is not always needed. As an example, a daemon using Bluetooth would rely on the built-in mechanisms to attach devices such that they recognize each other when placed at proximity.

Update Logging: pfsd needs to maintain a log of every update it sees in order to retransmit them to other devices. Depending on the context, updates should be augmented by pfsd with a propagation vector to keep track of which devices have already receive any given update. Updates whose version is dominated by more recent updates have to be elided. Moreover, any update that has been propagated to all devices can and should be removed from the log to avoid wasting local storage capacity. This prevents simple retransmission of the whole update log when a new device joins a *file system*. Nevertheless such initial synchronization can be achieved by transmitting a back-end storage archive from any other *replica*.

File transmission: Along with the updates, back-end storage files containing the immutable content of file verisons handled by pFS have to be transmitted between the *replicas*. Files associated with different versions of a same file are very likely to have similarities. rsync[25] principles as well as concepts introduced in LBFS[14] should definitely be considered for file transmission in order to avoid wasting bandwidth.

Choice of communication channel: If different communication channels are available (IP (LAN / WAN), Bluetooth, USB), it is essential to define a policy to decide which one to use given the updates to be propagated. It would be inefficient to propagate GB of music over

WAN from a desktop computer at home to a desktop computer at work if this data can be efficiently pushed to a mobile device such as a digital player that will be able to quickly retransmit it when connected to the desktop computer at work.

There is a need for the formalization and the automatic acquisition of statistics concerning file access patterns, connectivity patterns, and propagations dynamics in the context of pFS. such statistics could be exploited to make efficient choice of the communication chanels to use given the context in which the updates have to be propagated.

8 Conclusion

We have presented pFS, a network file system that implements the "personal clouds" model we advocate in which each device stores a local copy of the data. There is no primary replica. files are accessed locally, which yields good performances compared to classical networked file systems, and updates are propagated asynchronously among the different *replicas*. pFS takes advantage of the physically transported devices and local connectivity to propagate updates, resulting in the same user experience as if everything was stored in the "cloud", without having to rely on a centralized infrastructure.

References

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

[2] N Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[3] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 285–298, December 2002.

[4] Russ Cox and William Josephson. File synchronization with vector time pairs. Technical Report TR-2005-014, MIT CSAIL, February 2005.

[5] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *6th annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[6] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Proceedings of the 17th International Conference on Conceptual Modeling: Workshop on Mobile Data Access*, 1998.

[7] Csaba Henk and Miklos Szeredi. Filesystem in userspace. `http://fuse.sourceforge.net`.

[8] Drew Houston and Arash Ferdowsi. Dropbox. `http://www.getdropbox.com`.

[9] Apple Inc. .mac. `http://www.mac.com`.

[10] Microsoft Inc. Live mesh. `http://www.mesh.com`.

[11] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

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

[13] Linus Torvalds et al. GIT – fast version control system. `http://git.or.cz/`.

[14] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[15] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 31–44, December 2002.

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

[17] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated

peer-to-peer filing. *Software – Practice and Experience*, 11(1), 1997.

[18] D. Stott 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.

[19] Karin Petersen, Mike J. Spreitzer, and Douglas B. Terry. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint-Malo, France, October 1997. ACM.

[20] Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the ficus file system. In *USENIX Summer*, pages 183–195, 1994.

[21] R. Rodrigues and C. Blake. When multi-hop peer-to-peer routing matters. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, La Jolla, CA, 2004.

[22] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[23] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[24] Dinh Nguyen Tran, Frank Chiang, and Jinyang Li. Friendstore: cooperative online backup using trusted nodes. In *Proceedings of the First International Workshop on Social network systems*, April 2008.

[25] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1997.