Pepys — The Network is a File System

Sape Mullender

Pascal Wolkotte

Bell Laboratories
2018 Antwerp, Belgium

Francisco Ballesteros

Enrique Soriano

Gorka Guardiola

Rey Juan Carlos University

TR RoSaC-2011-4

1. Introduction

This document describes the design of Pepys, a large-scale distributed file system for the the Internet. It will be maintained as a working document that tracks our current thoughts on and motivation for its design. As such, it'll be a document subject to continuous change.

Issues that require further discussion will be put in footnotes (we don't need them for other purposes, they don't get in the way of the main argument and they can be easily found. Identify them by name of the note maker, like so[1])

In the last section, we'll track the changes.

1.1. The Network is a File System

A network, generally, is there to move data from *there* to *here*. A file system, in contrast, can be thought of as moving data from *then* to *now*.

A *distributed* file system can be thought of as one that provides both space and time travel for data. It was a popular research theme in the eighties and nineties of the previous century to make these distributed file systems fault tolerant as well as fast. It is now time to revisit some of those ideas.

Recently, companies such as Akamai have started using massive storage servers as tools for accelerating web access. By replicating content on many servers, users accessing such content can be served by one of many servers, for example, the nearest, or the least loaded one.

[1] Example note. Once agreement is reached, they should be removed. [sape]

Using disks to speed up the network is, of course, a useful thing. But it can be turned around as well. One can use the network to speed up a storage service. We'd like to do both: Use the the *combination of network and storage* to provide better service to the user community.

In contrast to the Akamai approach — making the network faster — we want to *view the network as a file system*.

Here's why.

1. Browsers spend a lot of their time fetching files. A typical URL already names a server and a file on that server. Removing the distinction between fetching a file locally and fetching one over the internet can only make life simpler.
2. File systems come with authentication and access control. This is badly missing in HTTP (although HTTPS tends to authenticate the server — *but not the client*) and it is done very *ad hoc* inside web applications.
3. Web caching is necessarily primitive as a consequence of the poverty of the HTTP protocol and it only concerns web pages. A distributed file system has caching as one of its important design considerations and we may expect it'll do a much better job at doing it right and maintaining consistency as required.
4. Information sharing in the internet almost requires the help of third-party services such as YouTube (for video), Flickr, Picasa (for photos), Google Sharing (for everything legal), or BitTorrent (for most things illegal), etc. Information shared must be protected with user names and passwords and people collect hundreds of them in the process of accessing friends' web sites. Direct sharing becomes simple in a distributed file system and its natural access control mechanisms can be used to dictate precisely who gets to access the data.
5. As we have moved from one computer serving many to many computers serving one, file synchronization and backup becomes a serious issue. A distributed file system that maintains a consistent view of one's files will become Quite Important. Additionally, backup in a server away from home or the office is important as a matter of disaster recovery (fire, flood, earthquake).
6. A single protocol (e.g. HTTP) may not work well in scenarios that differ radically in network and computing resources. What might work well in a LAN may perform poorly overseas. A uniform file system interface may be able to group together different protocols and caching strategies, each one tailored to specific storage, computing, and network conditions. The result might be close to use an optimal protocol for the scenario relevant to a particular user location and time.

> We envision the creation of a set of protocols that unifies a federation of otherwise autonomous file servers to create a distributed file system that can span the globe while providing a single interface that is easy to understand.

Each file will live on just a few servers chosen and/or approved of by the file's owner. File system owners can enter into bilateral agreements to trust one another for the purposes of specific file operations. A regular web server would allow almost anyone to read and cache its files. A home file server might allow his Telco's server to store backups of its files and the servers of friends and relations to read the directories containing vacation snapshots. A work file server might allow employees and their home file servers to cache and update files related to certain projects.

> Furthermore, we envision a file system that can help applications provide sophisticated and interactive *file sharing*.

(But only if the semantics that do so are generic and not specific to particular services.)

For example, we want to be able to allow users to be able to know that they are editing the same file at the same time, so they can (1) avoid doing that, (2) wait for the other to finish, or (3) embark on a joint edit session.

1.2. Media Data

Isochronous data (audio and video) already consumes about half the bandwidth in the internet and this fraction will increase further. The combined storage and communication infrastructure will have to be adapted to being able to deliver isochronous data with a negligible probability of deadline miss.

Home users are shifting their attention from broadcast TV to offerings such as Catch-up TV (CuTV), Video on Demand (VoD), YouTube, etc.. As a consequence, backbone traffic will increase dramatically, unless caching in the Central Office is used to reduce the number of downloads over the backbone.

Telcos and Internet Service Providers (ISPs) typically operate at the boundary of high-speed internet trunks and low-speed, often dedicated, connections to end users. By integrating storage with the DSLAM[2], in the Central Office, content can be delivered to end users with latencies less than than their reaction times.

In Service-Level Agreements (SLAs) with movie and video content providers, Telcos and ISPs can exploit their caches to serve home users directly out of local caches for most content and download from the providers only for uncommon content.

Regular access control can be used to give end users access to digital content. Pay-per-view mechanisms can be used as the trigger to add users to the access-control list (to remove them again after a specified period has elapsed, or the content has been viewed).

The challenge for isochronous data will be to achieve just-in-time delivery of the data to the end user. Late delivery is worse than early delivery: late data is useless data. But very early delivery isn't good either, because it needs to be stored and the storage space may not be available.

Caching servers on the last mile to the end user make just-in-time data delivery much easier: The last mile is often a dedicated link and can be scheduled for isochronous data delivery. A caching server in the last mile can be made to buffer a significant amount of the media data relieving the rest of the network from strenuous real-time demands.

The only notable exception is delivery of live broadcast TV for which special real-time delivery arrangements will always be necessary. But a cache at the Central Office can still be used to allow pausing the TV at home and time-shift viewing.

---

[2] Digital Subscriber Line Access Multiplexer

1.3. Dynamic Content

Unix pioneered the notion of dynamic files. These are files that do not have an on-disk representation; instead, their contents are generated on the fly, whenever these files are read. Device files are just one example of such files: writing to `/dev/tty` causes text to appear on the terminal and reading from it delivers the characters typed on the keyboard.

But the concept is much broader. The files in `/proc` in Unix, Plan 9 and Linux represent the state of the processes running on the system. In Plan 9, they also represent the interface for managing, monitoring and debugging processes.

Dynamic content is quite common in the web also and has become the standard way to present data to specific users (a user's bank balance viewed via a web interface is a quintessential example).

Obviously such dynamic content must be supported and it is clear that files provide at least as good a mechanism for doing so as dynamic HTTP.

We argue that files are a better way because:

1. The reader is always authenticated, so no obtuse steps are required (log in, cookies, etc.) before dynamic content can be delivered.
2. The file system may remain aware of the set of users for a given file at a given time (UNIX and Plan 9 use *file descriptors* for this purpose). This may be used to arbitrate access and also to customize synthesized contents for the file on a per-usage basis.

1.4. Access Control

In a far-flung file system as proposed here, authentication and access control are key elements. The system will have to be able to support a variety of authentication methods and will need to have a rich set of access-control mechanisms.

The entities authenticating — *principals* — will be people, systems and services, but also principals acting in *roles*, *groups* of principals and principals *delegated* to *speak for* other principals.

Examples are people in the role of manager of a service; groups of family members or a group formed by the members of a project team; a program acting on behalf of a user (e.g., outlook managing a mailbox on behalf of a user — one usually doesn't want the user to manipulate the mailbox directly).

A variety of authentication mechanisms will have to be supported, for example via passwords, RFID cards, smart cards, mobile phones, iris scans. Another reason for multiple authentication mechanisms is that frequent use of one method weakens its strength. For that reason it is not appropriate to use the authentication mechanism used to access one's savings account also to open one's car door.

A key challenge will be to make the access-control mechanisms understandable and manageable to lay users.

1.5. Replication, Backup, Consistency and Archiving

Many users now have more than one computer: PC, laptop, home and work computers, PDA, eBook, phone. Maintaining consistency between the file replicas stored on these machines is becoming quite a challenge.

Most people do not regularly make backups (many don't do it at all) and, even if they do, the backups are often stored in the same room as the computer from which they are taken.

Distributed systems support for maintaining consistency, making backups and keeping archival records is sorely needed. Using an outside-the-home service is useful on several fronts: (1) backups aren't kept in the home, so, in case of fire, quake or flood, the data is safe; (2) when all computers in the home are switched off, the data is still accessible; (3) data sharing need not use a home's uplink to the network which tends to be slow.

We will assume that most files are replicated across multiple platforms, either for data safety or because they are used on more than one host. We are investigating a model in which files are updated by replacing their current version by a new one. Versions themselves are immutable and can, therefore, be cached safely without concern for obsolescence.

The label "current" that is associated with precisely one version of each file must be maintained via a fault-tolerant distributed algorithm. A host that is temporarily off-line has no way of knowing which version of a file is current, or even whether it has the current version on board. We allow every host to update the last-known current version by creating a new version based on it, but race conditions and being off-line may prevent the new version from making it to currency.

A separate mechanism is needed to consolidate the changes made to versions that could not become current into the current version. If any off-line updates are to be allowed, the possibility of such inconsistency cannot be avoided. In practice, it is not a great deal [Satya:Coda], but mechanisms must be provided to consolidate changes in parallel versions.

An archive can be used to store the history of a file by storing every one of its versions[3]. Similarly, backups can be made by pointing to one or several servers and designating them at all times to store the current version[4] of the files under consideration.

Files can be designated as *transient* to prevent their history from being stored or even from being backed up. Other files, with dynamic content (i.e., content generated only when the file is read), can be marked as uncacheable.

What has been said is semantics, the actual protocol may perform actions (relative to version numbering and file ownership) on a per-tree basis, perhaps operating on subtrees (and defining them as operations are performed on different files from different places. For example, new versions may be created in practice for entire sub-trees, actually affecting directories from the root of the subtree down to the file that changed. Sub-trees may be defined at run time according to their actual usage. A benefit of doing so is that a client might create new versions locally within a locally owned subtree.

1.6. Latency

Latency is an important issue and not just for media data. Media data mostly suffers from jitter, the variation in latency, but this can be solved, for all but interactive media, by increasing buffer sizes.

---

[3] Experience with the Plan 9 archive suggests that the amount of storage consumed by an archive is not excessive. The twenty years of Plan 9 history consume less than half of the total storage in the system.

[4] And perhaps also archived versions.

In all case, we strive for minimal latency between the client's *open*, or *read* and the arrival of the result. This implies that, in most cases, a single message round trip should suffice to open a file and read data. To this end, we are experimenting with combining operations in a single message and volunteering information where that information is likely to be requested.

Most files are read whole and from the beginning. When a file is opened for reading, it, therefore, stands to reason that the next request will be a read operation. A local file server can anticipate by issuing a combined *open+read* request. Similarly, for files open for reading, close could be implied by reaching end-of-file, especially if the reader is itself a cache.

For example, directories are usually small (or at least not too large) and in all but a few cases are fully read. A single request will transfer entire directory contents, including metadata for contained files, to the client.

Devices remain undisturbed by this design, because they are marked as uncacheable and all requests proceed directly to the device server.

1.7. Scalability

Protocols in a storage system as described above could be used to replace HTTP as the primary mechanism to download and upload browser content in the web. Unlike HTTP, they facilitate caching and sharing and they make it possible that applications, using just one interface, can access objects locally and remotely.

Third-party file sharing services (Picasa, Flickr, YouTube) are no longer needed; neither are special peer-to-peer data sharing protocols such as BitTorrent.

Potentially, a file service as described above could become very large indeed. We envision several ways to keep the complexity resulting from sheer size at bay:

1. The service is intended to be a federation of otherwise autonomous file servers, separately managed. The intent is that mismanagement of one server does not adversely affect the performance of others.
2. Servers only interact with a limited number of peer servers. Such interactions are based on (limited) mutual trust resulting from access-control agreements between the owners and the users of the files stored on them. In other words, file servers only interact as a result of explicit file sharing operations which need to be granted by attendant access rights.
3. Caching servers are intended to be placed on the path between a user and the data sought by that user. We assume users will use a caching file server on the computer from which they operate, another one at the gateway between their site (home or office) and the network, another one again at the ISP's or Telco's DSLAM/Central Office, and one or more on the data provider's premises.

   Thus, file access typically passes through relatively few management domains, approximately: (a) the user, (b) the user's office, (c) the Telco and (d) the content provider and there are SLA between the participants on the path.
4. All caching is on the basis of immutable versions. Cached entries, therefore, may become obsolete, but never incorrect. Genuine distributed control is only needed for (a) maintaining the administration of which version is deemed *current*, and (b) maintaining the required amount of redundancy.

2. Design Principles

2.1. File systems and thin clients

We need two things:

1. A protocol to share resources
2. Client software to achieve a “single system” feeling, no matter what we use.

The next sections discuss each point. But both things are different issues.

2.2. File system

There are many distributed file systems that can be used. In practice none of them work well for different usage scenarios, because scenarios may go from disconnected operation to sharing a single (central) file server.

We think we need **a system that can integrate different file system protocols**, but we should probably strive to design **a single protocol that can serve all scenarios**. We should at least design protocols that work well in the local-area case, using the system overseas, and disconnected operation.

To do so, we need a general purpose (but abstract) protocol that can integrate different concrete protocols underneath yet maintain the same semantics to the user. The protocol provides a minimum set of requests but permits other requests to be added, without disruption to existing software, to address specific scenarios where the core protocol may not be enough (e.g., video streaming). A negotiation phase before speaking the actual protocol establishes which extensions are supported by the parties.

We propose what follows:

1. Files are versioned: once *committed*, versions are immutable. Updates start from one immutable version and produce a new version based on that.
2. Versions are identified by a signed 64-bit *timestamp* that represents the number of nanoseconds elapsed since the start of the third millennium (negative numbers represent timestamps from the second millennium).
3. Given a file and a 64-bit timestamp, the file system allows finding the version with the highest timestamp less than or equal to the given one (which usually is the version that was current at the time).
4. A directory changes only when entries are created, renamed, or deleted. Changes, therefore, do not percolate to the root of the file system.
5. A *change log* is associated with each directory. This reports version updates in the subtree of that directory. The *change log* for the root, therefore, reports all version updates for the file system.
6. Each *file system* implements an autonomous file tree. A cache may hold files from multiple file systems. The end user builds a global name space by *grafting* (mounting) trees or subtrees from sundry servers onto a local trunk. For now, we assume the use of a *prefix mount table*.
7. The interface, as provided to the client software on machines using the system, is loosely based 9P. The working name for the new protocol is **ΠP**. This is the protocol used between the local client file system and a file system cache as well as between a cache and a file server. This does not specify anything regarding the protocol used to get to the server(s).

8. End-user file servers provide to the applications and the user the usual file system interface of the operating system being used. On Windows, the file system will look like a *Z*: drive, on Linux it'll fit into the Virtual File System, on Plan 9 it'll present a 9P interface, etc.
9. Each file must have precisely one *current version*. When version management is distributed, it should use some flavour of distributed atomic update, such as two-phase commit to go from one version to the next. Before a version can become current, it must be in stable storage.
10. Disconnected operation is explicitly supported. When authoritative information about what version of a file is current is missing, the latest known current version is used and, although users are warned, they can proceed normally. Updates are stored and, when the system reconnects, an attempt is made to make the latest updates current.
11. On systems that can be expected to disconnect (e.g., laptops), the persistent cache will be aggressive in acquiring latest versions of essential files and files that have recently been used. A concern here is finding mechanisms that keep sets of files mutually consistent; for example, if a system-call interface is changed, one needs all the new versions of kernel, binaries, `/sys/include` and libc or none, but not some.

In the remainder of this document all these points will be worked out in detail.

2.3. Thin client software

Regarding the software for thin-clients, a candidate is a new version of o/mero and o/live. This is actually independent of the file system software (may exploit caches and file system features but is otherwise independent as long as the edition process is involved).

Some features like grouping trees for browsing, a change log of editions, etc. are already implemented in *o/mero* and *o/live*. A replacement could be implemented taking into account the new protocol and also the desired set of potential users.

A place for improvement is that *o/live* tries too hard to maintain a synchronized edition process. This is used to keep Zerox'd buffers synchronized, for example. In *o/live* a Zerox works also across machine boundaries and not just within the same terminal. But experience indicates that cooperative editing is rare if present at all.

Currently, mouse operations synchronize the edition. We propose to move more of the edition loop into the viewer for the user interface, perhaps including selection handling and maybe command execution for certain commands.

Users know when they want to synchronize changes, therefore, exploring an explicit ''synchronize'' key might be interesting.

This requires experimentation and building actual prototypes, because in general (for user interfaces at least) you never know until you try them. We can make a start by modifying what the Octopus has to adapt to the new system.

2.4. Versioning

Pepys will be a *versioning* file system. A file changes by the creation of a new version that becomes the new *current* version.

Files can be opened in two ways: *publicly* and *privately.*

In both cases, the result of opening a file for modification results in a new *version*; the difference is that, in the case of *public* opens, other users of the file can observe the changes as they happen, whereas, in the case of a *private* open, the changes do not become visible until the file is closed.

When a file is *privately opened*, the current version is cloned into an unnamed and invisible new version that can be read and written by the opening client. When it is closed, this version will be timestamped (i.e., acquire a name) and — normally — become *current*; the previously current version becomes *archived*.

The best way to describe *public* updates to files is by describing them in a way that does not actually reflect the way they are implemented:

When a current version is *publicly opened*, (for updating) for the first time, an *unnamed archival* copy is created. The original version can then be modified by the opening client and other clients that also open the file and the changes can be observed by clients that already had the version publicly opened beforehand. When the last (updating) client closes the version, the version is timestamped with the current time and made *current*; the unnamed archival copy is archived under the timestamp of the previous version.

The default behaviour for append-only files is to open *publicly*. For all other files, the default behaviour is to open *privately*. The default choices are encoded in a file's meta-data so that synthesized files and device files are opened in the intended way by default. Flags supplied to *open* can be used to override the default choice.

Versions have states and Figure 1 gives the state-transition diagram for publicly or privately opened versions.

The meaning of the states is the following:

Current
: The version representing the current (public) state of a file

Nascent
: Modifiable unnamed version visible only to the creating client.

Pre Current
: A version on its way to become current. Not yet current on servers that cannot communicate with the file's controlling server.

Dud
: An immutable version that failed to become the current version because of a race condition.

Archival
: An immutable version that is no longer current or active.

Pre Archival
: An immutable version that is not visible and on its way to becoming *archival*.

Figure 1 shows some states with "Pre" in parentheses. This is to indicate that, during disconnected operations, there may not be a current version to base a next current version on. If there is indeed a race condition and the pre-current version at the basis of an update was no longer current (or was never current at all), then all versions, including the resulting pre-archival version will become "merely" dud.

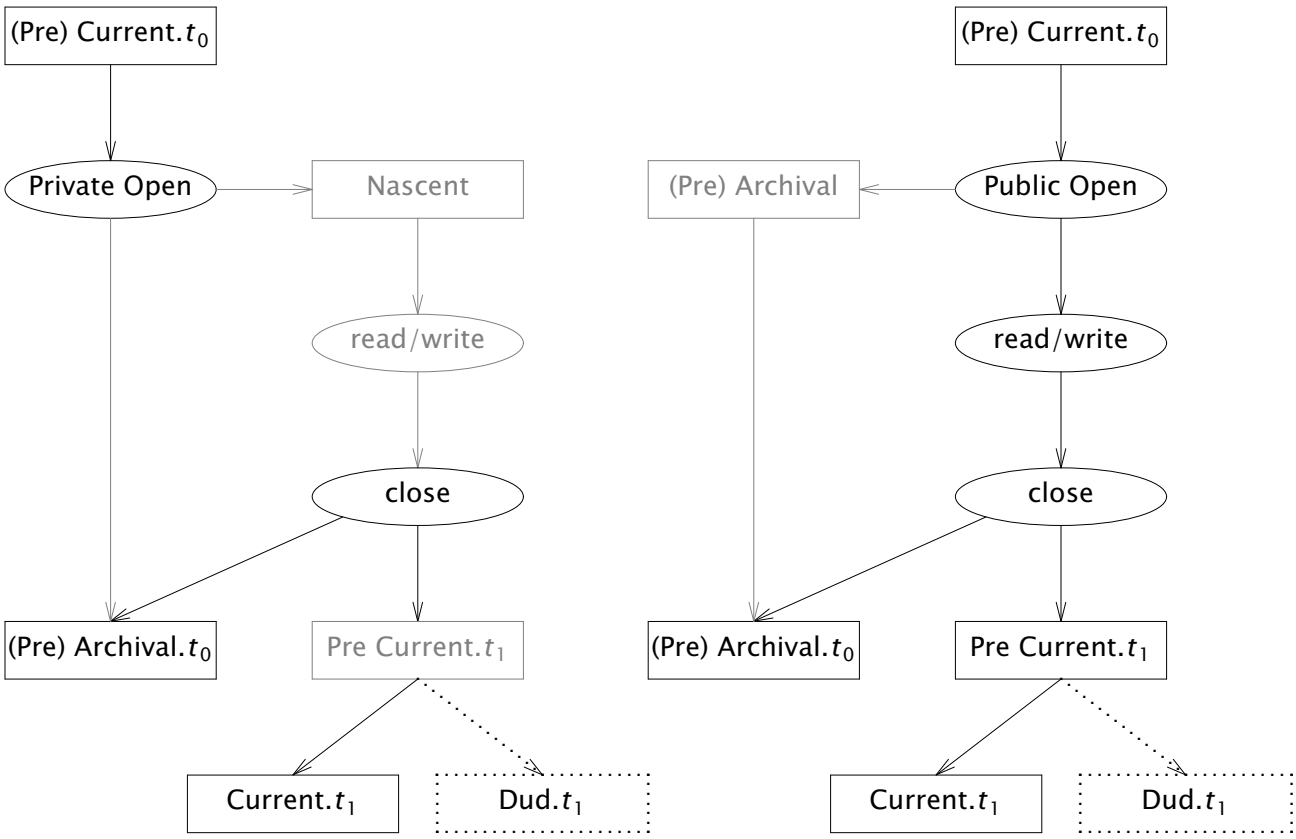

Figure 1 State-transition diagram Illustrating update procedures using private and public opens. Timestamps t_0 and t_1 ($t_0 < t_1$) represent version identifiers. Dotted transitions are those expected to be rare. Grey states and transitions are not publicly visible (note that they don't have timestamps associated with them either).

The operations on files and versions are discussed in the following sections. *Read* and *write* are very much like their conventional Plan 9 counterparts. *Open* and *close* have considerable additional semantics, but the defaults have been carefully chosen to make existing applications work normally.

2.4.1. Create

Create opens the first, empty, version of a new file or directory. Depending on the type of *open*, it creates a *nascent* or *current* version. No *archival* version will be created on *close*.

Directories cannot be written; they are always created *current*.

2.4.2. Open

All visible versions of files or directories, (i.e., *current pre-current* and *dud* ones), can be opened *read-only* with appropriate permissions.

Opening a file for reading does not affect the version of the file at all. The version that is opened is the one that gets read. There is no distinction in *public* or *private*.

For writing, only the *current*, or *pre-current* version of a file can be opened. Directories cannot be opened for writing at all.

If a file is opened *privately*, a *nascent* version is created that behaves as a mutable copy of the *current* (or *pre-current*) version it was based on.

If a file is opened *publicly*, and the *current* version is not already publicly open, then the *current* version is copied (virtually) into an invisible *pre-archival* version and the *current* version is made available for reading and writing by the client.

If a file is publicly opened that was already publicly opened by another client, then the current version is shared and reads and writes by both clients are intermixed.

The sharing behaviour for a version that is publicly open by multiple clients is left unspecified, although *writes* by one client must eventually become visible to *reads* from another client.

To assist the reading client, the server will return an EOF indication in case the client reads for the first time beyond the length of the version that is returned to the client on open. However, the client can try to continue beyond this offset without a close.

Append-only files are *publicly* opened by default, all other files *privately*.

Two properties, *force-public* and *force-private* can be used to override the defaults.

2.4.3. Close

Closing a version opened read-only is a non-event as is closing a file that was opened *publicly* while other clients still have it open.

When a *publicly* opened version is closed, and the client closing is the last one to have it publicly open, then the following actions are taken:

i. The version is flushed to stable storage,

ii. The controlling server, in an atomic operation, assigns the current timestamp to the current version, sets the state of the newly created version to *pre-current*, and makes the saved *pre-archival* version visible as *archival* with the timestamp of the previous version.

iii. The controlling server then attempts to make the *pre-current* version *current*.

This last step will fail if the authoritative server is incommunicado, in which case the version remains *pre-current* until the authoritative server can be reached again. It can also fail if, while the updates were going on, the current version was replaced by another (presumably *private*) update. In that case, the version becomes *dud*.

When a *privately* opened version is closed, the actions are:

I. The *nascent* version is written to stable storage.

II. The *nascent* version becomes *pre-current* with the current timestamp as version identifier.

III. The authoritative server is then asked to make the new *pre-current* version *current* and to make the previously *current* version *archival*.

This can fail for the same reasons as in the *public* case.

2.4.4. Versioning for Directories

Version changes do *not* propagate to the root of the tree. Applications that have to locate quickly changes in a file tree (e.g., indexing tools) may rely on the per-directory change log file described before.

In general, directories are managed like files, considering that file creation, removal, and metadata updates are the ''write'' operations for directories.

Automatic reconciliation for directories is provided for simple cases (creation of files with different names, non-conflicting removals, metadata updates for different files). In other cases a *dud* for the involved file(s) may be produced.

2.5. End of File

In all files, when the end-of-file is read, a flag notes the fact in the response to the read request. Thus, there is no need for the extra "null read" we have come to love so much in Unix systems.

This ''extra'' read may still be used for most files, but note the difference described in the next section.

2.6. Append-Only Files

Append-only files are handled like they are on Plan 9, considering what was said before about versioning. There is one interesting difference.

For append-only files, the read following the read containing the end-of-file flag, at the position of that end-of-file, *will block* until data is appended and becomes available to be read. For append-only files, *tail -f* is built in!

2.7. Metadata

If this file system is going to be useful across different platforms, the metadata should be rich enough to support them all. We'll never know precisely what metadata will be needed so it is important to make it flexible.

Apart from certain mandatory metadata (which may be stored in special ways for efficiency) we propose a mechanism whereby the user may specify any number of (*attribute*, *value*) pairs, where attributes can be arbitrary strings and values consists of *non-zero* size and data. Only mandatory attributes will have a restricted value space.

Since metadata is variable length and fairly free-format, it has to be accessed through regular *read* and *write* calls that follow an *open* operation with a *metadata* mode bit set.

For convenience, all metadata is presented in the (*attribute*, *value*) format, but mandatory metadata particularly is unlikely to be stored in this way.

To remove a metadata element, one writes it with zero size. (Conventionally, boolean attributes, such as *cacheable* use a size of one and a value of 0, i.e., the null string.)

The mandatory metadata we have identified to date is shown in Table 1.[5] Access control lists (`Acl`) are explained in the next section.

---

[5] This table is updated to reflect wild changes made in the next section. See the next footnote. [esoriano].

Attribute	*Type*	*Description*
Mandatory attributes		
vid†	Vid	unique file & version identifier
prev†	vlong	previous version (or ~0LL for first version)
length	vlong	length of the file/directory
acl	Acl	access control list
muid†	char*	name of principal to create (this version)
Predefined attributes		
d†	char	directory
c	char	cacheable
a	char	append-only
p	char	public-open is default
t	char	temporary file (no archival copies)
l	char	exclusive use
n	char	non-delegable

Table 1 Mandatory and predefined metadata elements. Fields marked with † are read-only and cannot be added, changed or removed.

3. Authentication and Access Control[6]

Caches operate on behalf of the users they represent. This implies that a user must delegate *some* authority to the cache server so that the cache can **speak for** the users it represents.[7]

Traditional UNIX permissions are defined for the owner of the file, an unique group, and the rest of the world. This scheme does not provide enough flexibility and several problems arise when different classes of users need to share data.

On the other hand, modern file systems, such as NFSv4 or MacOSX HFS+, support baroque Access Control Lists. These schemes provide a lot of flexibility, but they are very complex and are hard to configure.

One thing we must address is this: file and cache servers are allowed to manipulate users' files as a matter of course. Sometimes, file servers in different management domains can all create, read, write and delete files on behalf of a particular user. We do, hoewever, need to put users firmly in control of which servers can do what to their files. File servers may act *on behalf* of a user, but they must still be *distinguishable from* that user. For example, they will not be able to authenticate as that user.

It turns out that the *group* concept can help out here. A *group* is a collection of principals with the property that each principal *speaks for* the group. If the group can write a file, any mamber of the group can write the file. Members of the group authenticate themselves as themselves, not as members of the group.

---

[6] This section has been dramatically modified. This is an alternative proposal for the FS access control scheme. If we do not agree on this, I will restore the old section. [esoriano]
I agree to some extent and I've changed it again to reflect this. [sape]

[7] Do we really need fine grain (per-file) delegation support? Is per-tree delegation enough? I don't know, but we must think about this before opening the pandora's box. [esoriano]
Yes, we do. Source files and object files often share a directory but their protection may require different attributes. [sape]

Groups, therefore, can be represented as lists of principals and there is no reason not to store that list as a file. The name of the group is the name of the file (e.g., group "pepys" is stored in — say — `adm/pepys`. Note that groups are concepts local to a file tree.

Principals are members of groups. Names of principals can easily be distinguished from those of groups: the name of a principal will typically be something like `sape@plan9.bell-labs.com`, while the name of a group will be a simple identifier. This allows us to name groups inside of groups, providing for an easy way to add groups "family", "friends" and "relations" to the group "vacation-snapshots".

Users can simply be groups with just one member. And there is no reason for files not to be owned by a group.

The Access Control List is reprented by an `Acl` structure that simply consists of a count $n > 0$, followed by n *name/permission* pairs. The first name is the name of the owner of the file (which is a group, often, but not necessarily consisting of one member). The remaining names are names of additional groups. These are UTF8 strings. The permissions are bit maps as shown in the next section.

The primary file server, by its role, has all rights to every file it manages (a right it cannot be denied). If there are zero additional groups on the ACL, Unix and Plan 9 can list the files as having identical owner and group with the owner's rights and no rights for others.

Speaking of others, we may introduce a default group called *everybody* or *none* that everybody is always a member of and use that in the Acl.

3.1. New Permission Bits

- `r`: Permission to read the data of the file or to list the directory
- `w`: Permission to write the file's data or to create files in the directory
- `x`: Permission to execute the file or to search the directory
- `d`: Permission to delete the file
- `R`: Permission to read the metadata
- `W`: Permission to write the metadata
- `l`: Any operation over the file is logged (audited). An external tool can be used to watch the log and raise alarms depending on rules[8].

4. Current Version Management

One of the biggest issues for the design of Pepys is maintaining consistency in the face of caching, replication and failures. Making versions immutable is not simply avoiding the problem. The problem merely shifts to consistently keeping track of the *current* version.

We'll call the collection of algorithms and mechanisms for doing this *currency management*.

Information about which version of a given file is current must be replicated for fault tolerance and *agreement* is necessary to change it. The system for *currency management* is a distributed system.

[8] We should keep it out of the file server. [esoriano]

If we intend to allow some form of *exclusive update* meaning that a particular process/server is given the right to replace the current version with a new one (provided there are no failures), then the mechanism to agree on such exclusivity becomes part of *currency management*.

The locus of activity for a file sometimes moves from one location to another. A document may be edited at work during the day and at home in the evening. If the link between home and work is slow, then coordination in only one place would make work in the other place inefficient. We'd like to have a mechanism to move the power to make decisions about currency to where the activity takes place.

The Cambridge University Envoy[9] system and Ceph File System[10] both introduces a file management paradigm in which caching servers can be made responsible for managing *subtrees* of the main file tree. Those caching servers can then delegate the management of *subsubtrees* to yet other servers (or even back to the main server).

We want to adopt this principle for *currency management*, because

- Projects tend to be organized by directory tree, so access tends to follow that organization,
- If control is transferred on the basis of individual files, the risk of very fragmented control is great,
- Envoy investigated a mechanism on the lines of a *weak force* that pulls control of a file tree to a central place and a *strong force* that pulls control of a subtree to where it is used. The combination of such forces causes control over subtrees to move where they are used, but to move that control back to a central location when activity ceases for a sufficient amount of time.

A complicating factor with the adoption of this model is that, before a server can accept responsibility for currency management of a subtree, it must have up-to-date currency information for every single file in that tree.

Here's a Aunt Sally proposal for doing currency management.

4.1. Currency Update Algorithm

To explain the basic protocol for maintaining the current version, it is simplest to consider a single file. The basic idea is this:

1. Information about what version of a file is current can be cached under timeout (i.e., such information in the cache can be relied upon only until the timeout expires).
2. There is a group of servers (and, here, a client using ΠΡ counts as a server) that can cache information about what version is current.
3. These servers are expected to respond to requests to allow their timeouts to expire immediately.
4. Timeouts are chosen such that, if a server does not respond to such a request, and the system has to wait. the wait time is acceptable (and the server may be given

---

[9] Russell Glen Ross, *Cluster storage for commodity computation*, Computer Laboratory Technical Report N° 690, (UCAM-CL-TR-690) Cambridge University, June 2007, ISSN 1476-2986 (http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-690.pdf)

[10] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn, *Ceph: A Scalable, High-Performance Distributed File System*, Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06), November 2006 (http://www.usenix.org/events/osdi06/tech/weil.html)

much shorter timeouts in the future).

5. One member of the server group acts as *currency coordinator*, and manages currency changes.
6. For fault tolerance, it may also create a small nucleus of servers from among the group mentioned in Point 1 above, that helps *manage* the current version. The servers in this group replicate the currency information for reliability.
7. The coordinator exchanges keep-alive messages with the other members of the nucleus. If one of the members of the nucleus becomes unreachable, it may be replaced by another. If the coordinator gives the ghost, one of the remaining members in the nucleus becomes the new coordinator.
8. When the coordinator is active, it can appoint its replacement coordinator; it can also change the membership of the nucleus.
9. A currency change happens via the following steps:
   a. The coordinator sends a *prepare-to-update* message to the members of the nucleus.
   b. The members of the nucleus write the new version to stable storage (if that hadn't been done already) and they recall from the server group any outstanding cached copies of the current-version information.
   c. The members of the nucleus wait until new version is written and, for each server group member, until either an acknowledgement to the cache recall has arrived, or the caching timer has expired.
   d. They then send a *ready-to-update* message back to the coordinator.
   e. The coordinator performs the current-version update and sends a *version-update-commit* message to the nucleus members.
   f. When the *version-update-commit* message arrives, the nucleus members may once again allow current-version information to be cached by servers in the cache group.
10. If a nucleus member crashes during an update, the update will be completed by the remaining members. If the coordinator crashes, a new coordinator will be appointed first. Then that coordinator will finish the update in consultation with the remaining nucleus members.

Some observations are in order. If the nucleus consists of a single server, then that server obviously becomes the currency coordinator and it will not have to consult with any other machines to do currency updates (but it *will* have to recall any outstanding cached currency information). If the nucleus consists of two servers, then one is the currency coordinator and the other more or less acts as a hot standby.

4.2. Currency Management for File Trees

A file tree T is managed by a server S_T. The servers in the group Γ_T are authorized by S_T to participate in caching and currency management. Membership of Γ_T is controlled by S_T and all members of Γ_T are informed of any membership changes. We expect membership changes to be relatively rare.

S_T assigns, for any subtree $T_1 \subseteq T$, a currency coordinator M_{T_1} to coordinate changes to the currency of the member files of T_1.

The currency coordinator M_{T_1} selects a nucleus $\Theta_{T_1} \subseteq \Gamma_T$, such that $M_{T_1} \in \Theta_{T_1}$, to act as backup currency coordinators. It is possible that $\Theta_{T_1} = \{T_1\}$; that is, it is possible that Θ_{T_1} consists only of M_{T_1}

For each file $f \in T_1$, the currency coordinator M_{T_1} determines what version of f is current. Additionally, M_{T_1} can delegate control of any subtree $T_{1.1} \subseteq T_1$ to another currency coordinator $M_{T_{1.1}}$.

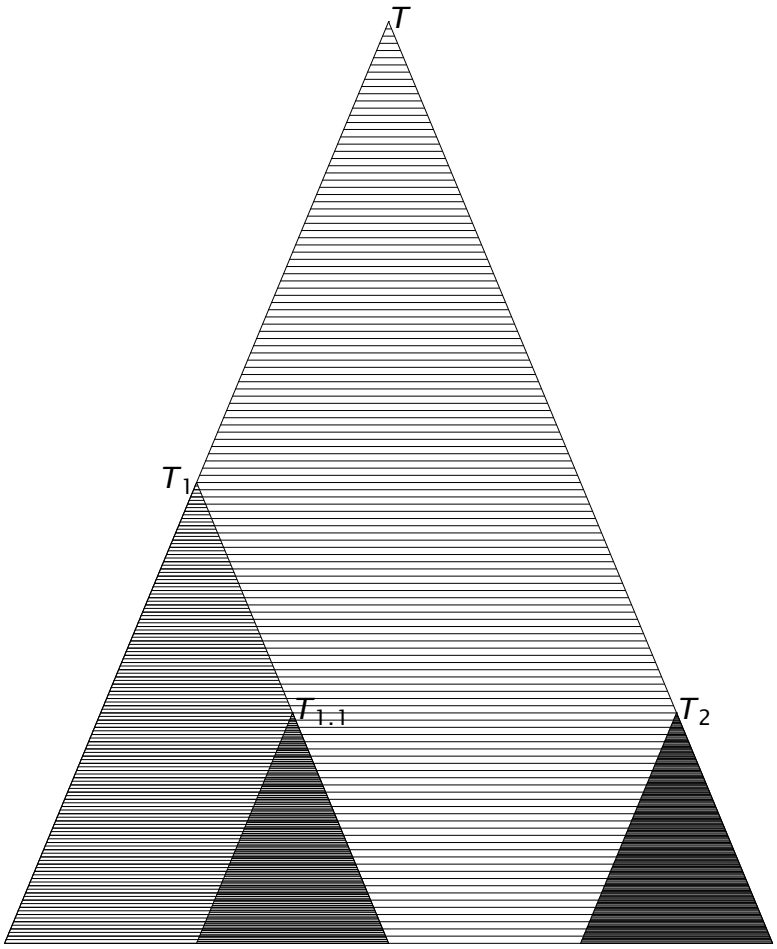

Figure 2 Example of a file tree and subtrees. Each shade is managed by a different nucleus of servers.

Figure 2 illustrates how different currency coordinators M_T, M_{T_1}, etc., manage different subtrees. Each shade in the figure represents a set of files in the file tree — whose root is at the top of the figure — managed by a different currency coordinator.

The layout of the figure shows how S_T must have delegated coordination of T to M_T, how, subsequently, M_T delegated coordination of its subtrees T_1 and T_2, respectively, to M_{T_1} and M_{T_2} and how, finally, M_{T_1} delegated coordination of subsubtree $T_{1.1}$ to $M_{T_{1.1}}$.

For currency updates, M_{T_1} and members of Θ_{T_1} use the protocol as described in the previous section. The currency manager uses a similar protocol to delegate currency control for one of its subtrees to another nucleus.[11]

Experimentation will be needed to establish rules for handing off coordination for a subtree to another nucleus. The general principle will be that, once the rate of requests for updates in a subtree exceeds a certain threshold, a hand over of the coordination can take place to a set of servers nearest the location of the updates. When the rate drops below another threshold, the subtree can rejoin its parent.

Such coordination changes must be broadcast to all members of the server group, Γ_T, for the tree. Note that the broadcast isn't merely to the server group of the subtree (we expressly did not define this, so there is no such thing), because servers need to know where to go for information about any file anywhere in the tree.

This would typically be done by notifying S_T of all coordination changes and then having S_T inform Γ_T. Details of how this is done are still murky; it could, for example, be done by having members of Γ_T tracking an append-only log of management information.

5. ΠP Protocol

In this section we propose the new ΠP protocol that tries to optimize the number of round-trips. This protocol is based on a few facts and assumptions:

- Reducing round-trips is always reduced by grouping operations.
- Most files have a direct relation with others and their relation is known by the client. For example, all files in a directory or a web-page and all their images. The client may want to update the version of all related files or read multiple related small files in one RPC.
- The order of file operations (open, close, read, write, create, remove) on a set of files is not always fixed. Therefore, the client should be able to order the commands arbitrarily without increasing the number of RPCs.
- A walk in 9P only walks to a file's parents or children. In a versioned file system, the relation between file versions are known. Each file has both parents and children similar to a file's parent and children in the name space. Using the (previous) version number in the metadata of the file, it is easy for a client to build a path and walk to the previous version of a file.
- A server in the distributed file system has to act on behalf of other principals in the system. To limit the number of connections between a client and server this connection has to be shared amongst different principals. We assume that the number of principals per connection is limited.

The protocol has a number of distinct phases similar to 9P. The phases and corresponding messages are explained in the next sections.

All messages carry an initial *size* for the entire message, to help programs ignore unknown messages. Furthermore each message has a *unique tag*, such that multiple messages can be outstanding on a connection at any moment in time.

5.1. Negotiation phase and protocol extensions

The aim is to provide a general purpose protocol yet permit it to operate well on concrete, but different, scenarios: local area networks, [A]DSL lines, and wide area. It should work well in general, however, certain scenarios may require specific requests besides the ones used in most cases. To provide for expansion, an initial negotiation may take place to select a protocol and agree on supported features:

```
size Tproto tag msize nmsgs protocol1+option1+option2  protocol2+...
size Rproto tag msize nmsgs protocol1+option1...
```

The request proposes the maximum message size (msize), the maximum number of requests in a single message (nmsgs), and one or more protocols plus a set of known options. An example could be:

```
size Tproto tag 8192 128  ΠP+lease+stream 9p2000
```

to ask for ΠP plus leasing requests. The reply must include a maximum message size and maximum number of requests in a single message, both must be equal to or smaller than requested, and a single protocol of those mentioned in the request (the first one understood by the server) plus the list of options implemented by the server. For example:

```
size Rproto tag 8192 1 ΠP+lease
```

Both parties agree now to use the protocol and extensions indicated, which implies a particular set of requests.

Only the extended client and server need to know about extension semantics. Other parties must only be careful to handle unknown requests as appropriate. There are two requirements regarding extensions:

1. Any non-extended client or server *must* still work no matter what will be added in the future (it might work worse, but should still work). That is, extensions may not invalidate existing clients or servers.
2. Intermediate programs that act as gateways for ΠP messages must still forward (verbatim) all unknown requests. They might correspond to unknown extensions and they should still be valid end-to-end in the ΠP connection.

5.2. Communication phase

After the negotiation phase, there are two request messages in the protocol: *Tgroup* and *Tflush*.

The *Tgroup* message packages one or more elementary file operations, which are described later. The client determines the order of operations in the list and the server has to handle the list in order. In case one of the operations fails, the successive operations are not handled by the server.

```
size Tgroup tag mresp nops ops[nops]
size Rgroup tag nrepl repl[nrepl]
        ...
size Rgroup tag nrepl repl[nrepl]
```

Each file operation is limited to a single file and user, which is referenced by the client's chosen *fid*. The fid can be used in multiple group messages until the client informs the server that it can forget this fid. Multiple fids can be used in a single group message, which enables batching of different users and files within a single request.

The server can respond with a sequence messages for each group request, because not all responses to the list of file operations fit in a single message. A single file operation reply can not be split across multiple response messages. The client limits the maximum number of server response messages (*mresp*) in its request. The client can use $mresp \leqq nops$. This enables the client to control its incoming bandwidth and server response time. A larger mresp may result in a faster response time of earlier operations

in Tgroup, because the server knows it can fit the remaining replies in future responses. The group RPC is done when:

- A specific operation failed. (An *Rerror* must be issued for it).
- The sum of replied operations ($\sum nrepl_i$) in the sequence of responses is equal to the number of operations (nops) in the client's request.
- The server responded with the maximum number of response messages indicated by the client (nresp).

After the last response, the client can reuse the tag for a next request.

The *flush* message can be used to discard a specific outstanding *Tgroup* or *Tflush* message.

```
size Tflush tag oldtag
size Rflush tag oldtag
```

It does not matter if the message flushed is a group of one or more requests. It would attempt to flush the entire message.

The *error* message can be used as a server's response when a client's request or file operation fails.

```
size Rerror tag errorstring
```

The tag refers to the message that failed at the server. However, an error message can also refer to a particular operation within a batched request (one sent in a group of requests). In the latter case the reply uses the format of a elementary reply and omits the tag field (see the next section), and is sent within the group of replies. The group of replies to the file operations is order preserving, thus the position of the error message in the Rgroup message determines the operation that failed at the server.

A response to the following message (in case the open fails):

```
Tgroup tag=3 mresp=1 nops=4 {
        Twalk fid=6 nwfid=6 nameelem="dira"
        Twalk fid=6 nwfid=7 nameelem="b.file"
        Topen fid=7 omode= OREAD|OCERR
        Tread fid=7 offset=0 read=512
}
```

will be:

```
Rgroup tag=3 nrepl=3 {
        Rwalk vid=2340823232
        Rwalk vid=2340823482
        Rerror "User does not have read permission"
}
```

5.3. Elementary File Operations

The file operations are slightly adjusted from those defined in 9P2000. For now we only specify the differences in what follows. Each file operation does not have a tag as it is part of a group message. Instead, it is identified by its position in the *Tgroup* message. Their position in the (multiple) Rgroup response(s) is preserved as the file operations are handled in order. An *Rerror* reply can be packed in a Rgroup response. It will always be the last response, as successive operations are not executed. But see below.

a) Elementary request format:

```
size Ttype fid args
```

b) Elementary reply format:

```
size Rtype [vid] args
```

Each operation that is able to change the assignment of a fid to a particular vid will also return this vid[11].

An operation starts with a *fid*, determined by the client, that refers to a specific file and authenticated principal. (The authentication operation is shown later). Multiple fids of a single client can refer to the same file or principal. For example, to open both data and metadata using different fids. Operations that link a *fid* to a specific file return the *vid*, i.e., its global unique identifier. The *vid* is informative for the client and can not be used directly to gain access to a specific file.

5.3.1. Authentication and initiation phase

Usually, the first requests made by a client (using a *Tgroup* request) perform authentication and attachment to a file tree in the server.

The following elementary operations made by the client are to *authenticate* a user to a specific (sub)tree shared by the server. The authentication request starts this procedure by introducing a *user name* and *file tree* and linking this with a *fid* chosen by the client. The fid has to be used by the client in successive file operations either to authenticate

---

[11] Can a read or write change the vid too? Or only if you have multiple readers/writers in public mode? [ptw]
I think that it is required only if the file is public. [nemo]
Some current Plan9 implementation always increment the version number after a write and not on the close. In that case all file operations should return a vid by default, such that a client is always as up-to-date as possible with what happens at a server? However, in section 2.4 a version can only change on the opens and close. Thus in that case read/write operations itself do not change the vid. I added a scenario (In .ms file and not in footnote, because it kills the footnote's format), where I am wondering what should happen in case it is a public mode open. [ptw]

If it's a public open, it should be almost like unix. In your scenario, the version is the last known (if it was public). IIRC. [nemo].
True, but it has obtained data from a newer version. So how does client B (cache) handle its version management, i.e. act on requests for this file by its clients? Maybe it is a case that will never happen, so I am discussing an unrealistic problem.[ptw]
We could include an update number along with the version number. It would reflect how many updates do we know that are made to a current version. Just as a hint for caching. However, I don't like this. I prefer to redefine what *vid* means for us, so that we may have multiple vids for what we call the *current* version. Just that such version numbers are not archived. What do you think? If you agree (discuss with Sape in case he does not notice this footnote) you could remove these footnotes and go back to edit the discussion on version ids. [nemo]
We don't want to change the meaning of vid. However, we added a proposal in section 2.4.2 and 5.3.7. A client always reads the version represented by the vid that is returned on open. After the EOF of this version is reached, a client may read beyond this version's offset. However, it will not know the appropriate version for the data read. Maybe a hint on clunk can help. [ptw]
I think this will lead to surprises (EOF will no longer be EOF). Why don't we just say that (because of the public open) the client may read changes made by the other even though the version is the same, i.e., the current one? I'm propossing to remove the change regarding EOF in 2.4.2 and leave things as they were. That is, if someone added more bytes and it was a public open, then read would return more bytes, as expected. So, clunk, in your scenario would return the version according to 2.4.2 as of several days ago. [nemo]

or update/retrieve regular files. A fid has an implicit auth (and version) status.

```
size Tauth afid user aname
size Rauth avid
```

After a successful authentication, an *Attach* request links a new *fid* to the root of the tree. The client should clunk the authentication fid, *afid,* when it is no longer needed.

```
size Tattach fid afid user aname
size Rattach vid
```

In both cases the server responds with a globally unique identifier (*vid*). This identifier is a concatenation of a file identifier, *xid*, and version identifier (see §5.1) A *vid* identifies a version globally and can, therefore, be used on different servers to refer to the same version. Servers in a distributed setting arrange among themselves how to divide the identifier space to prevent clashes.

Versions are practically[12] immutable, so clients and servers can use the *vid* of the current version to check for changes: if the *vid* changes, the contents of the file change.

A *xid* (or *vid*) cannot be used by a client to gain direct access to a (version of a) file. Access is only granted via the file system's name space. It *is* possible to navigate the name space at a point in the past, however.

A client operating on behalf of multiple principals must authenticate separately for each of these and attach separately for each. The unique *fids* in those requests are used by the server to track the authenticated principal in subsequent operations.

An attach refers to the current version of a file tree by default. Archival versions are accessed by specifying the name `archive` in the *attach* message. The root of the archive tree gives access to the file tree at a time in the past via paths such as */yyyy/mmdd/*, */yyyy/mmdd/hhmm/*, `/today/`*hhmm/*, or `/v`*nnnnn...n/* to refer to the root at specific times.

5.3.2. Walk

The walk operation is used establish a reference between a specific file and a *fid*. It either initiates a new *fid* or changes the current *fid* to the new file. The specific file can be named by a client's known specific filename. The walk handles one path element, i.e. filename, per request, because you can group multiple walks in a group message. The server responds with a *vid* that is linked to the new *fid*.

A walk to a particular archived version of a file is only possible by an attach to a specific named tree and walking the hierarchy of this tree. This tree will show the last archived version of a file before a given timestamp. A link in the file's metadata to its previous version can be used to attach to the correct tree.

```
size Twalk fid newfid nameelem
size Rwalk newvid
```

[12] See the discussion of public opens in §2.

5.3.3. Open

The open operation has some additional flags in the omode:

- OPRIV - open the file in private mode (see section 2.4)
- OPUB - open the file in public mode (see section 2.4)
- OCTL - that opens the metadata of a file (in private mode by default). The metadata can be read/updated using the read/write operations. Specifying also OTRUNC only clears the non-mandatory attributes.
- OCERR - that enables to server to close the file on an error. The error should occur in the group message on an arbitrary fid or in future group message when the fid is used by another file operation that triggers an error response.
- OCEOF - closes the file and forgets the fid as soon as the server has replied with an end-of-file indication (0 bytes left).

```
size Topen fid omode
size Ropen vid iounit
```

5.3.4. Clunk

The clunk operation removes the client's fid as a reference to a file. If the file is open, this means a *close* on the file (fid). Because the close of an open fid for writing will generate a new version the server will reply with the newly generated version or the old version (see section 2.4 on versioning). The vid is included for homogeneity even if the fid was not open (or was open read-only in private mode).

```
size Tclunk fid
size Rclunk vid
```

5.3.5. Create

Create introduces a new file in a directory. A particular version of a file can also be created in the version tree (it may be specified as ~0 otherwise). This is required to support reconnection after disconnected operation. The permissions will be all the ACEs[13] from the parent's ACL, where the permissions are masked by the *mask*, field.

```
size Tcreate fid omode filename mask versionname|~0
size Rcreate vid iounit
```

5.3.6. Remove

It removes the file name from a directory in case it is the current or quasi-current version. The version itself becomes an archive. In case the file is an archive or dud it removes this particular version of a file. A file server may refuse to remove an archived (not dud) version of a file.

```
size Tremove fid
size Rremove
```

5.3.7. Read

The read reply includes an indication of the number of bytes still to be read in the file at the time of the reply. Zero means EOF, -1 means unknown. The first time an EOF is returned it reflects the version's length that is returned on open[13].

[13] Of course, the client has to start at an offset <= length and read with a count >= (length-offset). Please, remove if you agree [ptw]

```
size Tread fid offset count
size Rread left count data[count]
```

5.3.8. Write

The write response does also include the offset of the data written. This informs the client where in the file the server has written his piece. In case the server omits the client's offset, e.g. append-only files, it enables the client to quickly find concurrent additions by others.

```
size Twrite fid offset count data[count]
size Rwrite offset count
```

5.3.9. Metadata

Instead of separate messages for metadata (stat and wstat), we use the same four operations as for the file's regular data, i.e. open, close, read, and write.

The open and close to access the file's metadata is mandatory, such that metadata operations can be split over multiple Tgroup messages. This enables the client read the file's metadata and update it in a successive request without the possibility of another client modifying the file in between. It also enables a simple decoupling of the maximum message size and metadata size, although we do imply big metadata as the default. Changes in metadata will create a new version identical to a regular open/close.

The read and write operations remain identical to regular data with the exception that the contents of the messages requires parsing by client and server. The data in the message has to be formatted as a series of mandatory metadata records, plus some optional ones, in the format of

```
<key[s], count value[count]>
```

tuples. A write operation has to add a modifier (write key and value, append value, remove key) per tuple. The server may limit the size and number of non-mandatory keys.

6. Data Structures and Algorithms

6.1. Identifiers

Servers, files and versions all need to be uniquely identifiable. A server contains files and a file contains versions. A version is, therefore, identified by the composition of a server identifier, a file identifier (within the server) and a version identifier (within the file).

The size of each of these identifiers has to be such that servers, files and versions can be uniquely named without running out of bits. We propose to make each of these 64 bits in size.

14

I think we are over-engineering this. Shouldn't we leave things as they were? See the reply on the previous footnote, section 5.3. [nemo]

If ΠP truly catches on as the new way to manipulate objects in the internet, one must count on every machine attaching to the network running a server. This makes 32 bits insufficient. The next sensible choice is 64 bits (because modern processors can manipulate — move, compare — 64-bit entities in a single instruction) and 64 bits should be enough for the foreseeable future.

The same is true for files in a (distributed) file system. Some file systems are large enough that 32 bits is not enough to name them all. In addition, some servers will want to add some internal structure to file identifiers to aid locating them quickly. Again, 64 bits provides a large enough name space.

Finally, we are proposing to use nanosecond time stamps of 64 bits to identify versions. Using midnight, January 1st, 2000 as the anchor, a 64-bit nanosecond counter allows the expression of times from September 11, 1707 to April 10, 2292.

Using timestamps as version identifiers allows finding specific versions of a file by date. Nanosecond accuracy is usually a lot more than we need, but it also gives us a way to separate multiple versions created at (roughly) the same time.

This results in the following types:

```
typedef uvlong  ServerID;
typedef uvlong  FileID; /* Not to be confused with Fid */
typedef vlong   VersionID;

typedef struct Xid {
        ServerID        s;
        FileID  f;
};

typedef struct Vid {
        Xid;
        VersionID       v;
};
```

This makes a `Xid` a 128-bit data structure that identifies a file (but not a version). A `Vid` (Version-ID) is a `Xid` plus a 64-bit timestamp that identifies a version.

A *vid* is a bit like the Plan 9 *qid*, but, for now, it lacks a *type* field. That could probably do with a little discussion: what would a type be useful for? Clients get the *vid* of a file when they've walked to it which is before they get its properties, so, for that reason, it could be useful. On the other hand, operation grouping allows the client to get the properties in the same RPC as the walk anyway, so, for now, we won't include a *type* field in the *vid*.

6.2. Directories

Directories contain references to files and directories. The main elements of a directory, therefore, are *names* and *xids*. Directories do not name specific versions, because that would force updates of directories when the files they refer to change.

Names are arbitrarily long UTF8 strings that may not contain slashes or the *null* character.

6.3. Finding Files and Versions

A file server or cache server must be able to find files efficiently on either disk or in memory. Such searches are often accompanied by a time stamp: "find the file as it was on time *t*", or "find the current version."

We propose to use a balanced tree to keep *vids* sorted, most significantly on *xid* and, least significantly, on timestamp. Our Antwerp prototype (if it deserves than name already) uses an AVL tree. This allows searching for a specific *vid*, but, when that cannot be found, the neighbouring nodes are found and, if there are earlier or later versions of the file represented by the *vid*, those will be the neighbours in the AVL tree. A small test shows that inserting one million random *vids* into the tree takes less than 200 ns per node (2.7 GHz 2-core Intel 64).[15].

When the file system starts, the server reads the index from disk sorts the entries into an AVL tree. The index on disk is a sequence of `Indexelem` structs as shown in Figure 3. We're planning on storing data in a log-structured file system, so, typically, we have to write data before metadata. Most of the time, data and metadata will be written in one go, so file plus metadata can also be read back in one go. To facilitate this, we add an *offset* to the index element that can be used to find out where the actual metadata starts.[16]

```
typedef uvlong Diskaddr;
typedef struct Indexelem Indexelem;
struct Indexelem {
        Vid;
        Diskaddr daddr; /* where to start reading */
        int     doff;   /* where the interesting bit starts */
        int     dlen;   /* disk metadata size — if 0, not on disk */
};
```

Figure 3 Disk Index element

These index elements are read into the server's memory when it starts up. The memory data structure, including the overhead for the AVL tree is now 60 bytes (call it 64). That means we can store 16 per kilobyte or 16K per megabyte. We measured the Plan 9 file size to be 500K on average, which is probably small compared to most systems. Thus, we'll be using 60 bytes per 500 KB, or an overhead of roughly an MB per 10 GB.

That's quite doable to keep in memory for a server holding less than a TB, but a sizeable server will perhaps serve as much as a PB or more and then an in-memory index would take up more than a GB, and that is memory space large wasted because the

---

[15] We have the modified libavl, it's available if you want it. [sape]
Yes please. [nemo]
I think it may be a good idea to use a log-based file system, for both speed and reliability. But I'm not sure the proposed structure, specially the AVL, would be fine. It would be O(log n)and we could probably do O(1). A request has a fid and we must have a hash from fid to a memory file info (Indexelem), perhaps. For requests on the file (all but walk) we are done. In this case it would help to have a link to an in-memory structure for the previous version. For requests from that file (walk) we are almost done. We can be done if we have the tree structure in memory (perhaps only in part, but full if we can). Then we may directly walk to the next component in O(1), without reading. I have changed my alternate proposal in the next section to try to get closer to this one. [nemo].

[16] This may be fine for files that fully rewritten. But then there are logs, and video files, and other ones that are big enough that we do not want to use contiguous assignation (as suggested by *Indexelem*, or I'm missing something). [nemo].

overwhelming majority of files (and versions) will not be accessed in any reasonable timeframe.

We need a disk structure for the index, one that, given a *xid* and a *time stamp* finds the version current at the time of the time stamp in just one or two disk accesses.

If one assumes that versions are somewhat uniformly distributed over the lifetime of a file, then, one can do a two-stage lookup: The first stage finds information about the file: location of the current version, timestamps of current, t_c, and oldest, t_o, versions and the location of a contiguous "file" on disk containing in-order information about the location of each of the versions with their timestamps.

In the second stage, one can then find the version with the timestamp t sought, by seeking a fraction $(t-t_o)/(t_c-t_o)$ into the file and reading a chunk around that position of sufficient size to warrant a reasonable expectation of finding the version sought there.

This allows finding most current versions with one disk access (using a hash on *xid*) and finding many versions with just one more.

It probably makes sense, when older versions are accessed, to read the version information for all versions at once, because users are often looking for a particular versions by examining many.

The AVL tree in memory can be used as a cache for location information of files deemed to be active. SInce the space required for tree elements is quite small (64 bytes), caching of this information can be done fairly aggressively.

6.3.1. Discussion on data structures [nemo]

There will be locality in all of server-ids, file ids, and version-ids (A file server will have a single server-id, but a cache might handle multiple servers). Also, file operations are likely to follow the tree structure provided to the client, if we consider the tree as the *current* file tree. Therefore, locating the files of interest is a matter of traversing the file tree as presented to the user (and navigated by *walk* requests). Time travel (i.e., walking the archived portion of the tree) is similar, but may require location of particular (archived) versions instead of those present in the tree.

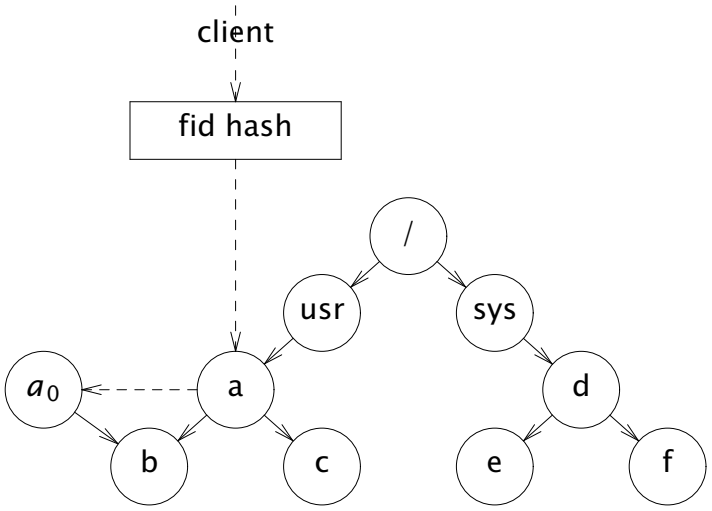

Figure 4 In memory structures after creation of file `/usr/a/c`.

In memory the file server structure might be as shown in the figure. We may keep:

1) A hash table to go from fids to Memfile (in-memory information for a file).

2) A tree of files in use (only their Memfiles), not the actual files.

The idea is that we build (probably on demand) a tree of files in use, perhaps anticipating further requests. For most requests but for *Twalk* we are done by using the hash table on fids. Then it is a matter of keeping in Memfile what we need to avoid much disk I/O.

For *Twalk*, if the memory structure keeps the (sub)tree in use, it is likely that we also have the Memfile for the destination at hand, by following just one pointer from the original Memfile. This is O(1) again, and we are in the case of an operation made to the Memfile we have at hand.

To aid in time walk, we can link to each Memfile the previous version of it. The client can only start a time shift at attach time. However, because we do not update directories to refer to newer versions of files, we must maintain a tree (implicit or explicit) and navigate on directories to the past. An alternative, perhaps worth reconsidering, is modifying the directory to also refer to the version of the file. Or perhaps doing so just within the file server[17].

On disk the file server may use a log of updates[18]. required to reach the current root for files. This index may contain a table to obtain the current version for a file and, then, accept further indexing to travel back in time. Thus, we could include:

1) A table to go from xid (note, not vid) to file.

2) A file is now a table of versions for the file. For each version, the file knows its list of (data and metadata) blocks, somehow.

I suggest we use a real table for (1) but a linked list (with skip lists) for (2).

The disk may be organized as depicted in the figure. All areas mentioned in the figure are logical, which means that they are defined as the nested concatenation, mirrowing, stripping, or partitioning of existing disk space. The top-level view for each area is always a concatenation (of perhaps just one device). Extra space may be added to a concatenation at run time.

---

[17] This would only be required in case we have to support versions side-walks, but we agreed on not supporting this due to security issues. However, if we are able to support this in the data-structure, why not in the protocol? [ptw]

It was scary to accept it on the protocol. But it may be a good thing to keep in the implementation. A version change does not change the directory, just the file (IIRC). Thus, we may have to navigate through old directories to reach recent files and vice-versa. Having the link to the previous version in the implementation may help. But the version id in the metadata might just be enough if we use a different implementation. I'm not yet sure that it's good not to change the directory to refer to new versions. All this is a consequence of that. Still thinking on it all. [nemo].

OK, within the server I would agree, but outside, may generate a lot of additional network traffic. [ptw]

I think that this link is just to be considered a local (within the same server) cache, to avoid the need to walk /vers/whateverpath when we have the file at hand. Accross the nextwork we can use Tgroup to walk to the previous version in O(1). Perhaps we should forget about this back-time-link even as an optimization and go back to this if an actual implementation requires so. [nemo]

[18] The disk file structure looks quite like ext2/3 and I have the feel that in the linux-world they don't want to move forward in this direction. I will have a look to Btrfs as they claim to reduce the number of indirections of file blocks. [ptw]

B trees scare me as a FS implementation tool. I heard from people working for BeOS that their file system was hard to get right mostly because it was using them. But I'm viassed here. [nemo].

For easy of administration, the file system warns when any area gets 50% full, 75%, and 90% full.

Figure 5 The disk uses a temporary work area for current versions and temporary files. The archive keeps a sequential log of inmutable versions. An index is the the entry point to find both working files and archived files.

Files are created and modified in a single working area. This area consists on a log of updates for a file. When a file becomes inmutable, it is archived by appending its contents to an archive log. File blocks are archived separately, trying to reuse the same block to archive the same contents.

On a periodic basis (or upon exhaustion of the working log), a cleaning phase starts. Upon entering the cleaning phase, new files stored in a second working area and temporary files (not archived) still alive are copied into the second working area. At this point, the first working area is declared empty and the roles of the first and second working area are switched.

The structure for a file in the log may be given by a data structure mentioning some inmediate data, some direct references to blocks, a simple indirect, double, triple, and perhaps quad indirect reference. For metadata, the file may contain some inmediate data plus some direct references to blocks. That is:

```
typedef struct File File;
typedef struct Adresss Address;
struct File {
        Stat;           /* mandatory metadata; including vid */
        Address previous;       /* address in disk for previous version */
        Address self;   /* in disk */
        uvlong doffset; /* offset for optional metadata after data */
        Address direct[N];      /* direct blocks (after inmediate) */
        Address single;
        Address double;
        Address triple;
        Address quad;
        Address ddirect[N];     /* direct blocks for metadata */
        uchar embedded[];       /* inmediate data and metadata */
};
```

The File structure must be placed on a block boundary. Thus Address refers to a block (when in disc) or to an archived block id. The rest of the block may be used by inmediate data/metadata. Thus, small files are processed using a single block.

The index data structure provides a map for obtaining the address for the current version of all files in the tree and for any version of the root of the tree.

7. Protocol extensions

7.1. Version handling and tree delegation[19]

Usually, a client contacts directly a server and mounts a file tree. Of course, this may be a cache instead of a server. For high availability, a server (or a cache) may be actually a collection of machines providing a more reliable service, but they operate as a single server to the client. In general, there is a hierarchy as shown in the figure.

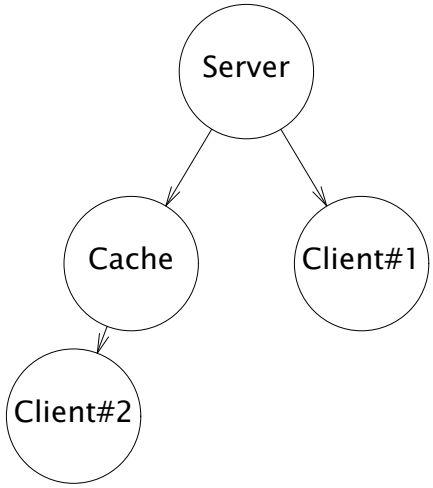

Figure 6 A hierarchy of caches or clients rooted at a server.

Perhaps, *Server* or *Cache* are more than one machine. To the rest of the systems, they behave as a single program.

The idea is that each cache behaves as a client to the server and as a server to the client below (as it could be expected). This means that when *Client#2* in the figure needs to obtain a new version to make a file current, the request must proceed up to *Server* in the figure, and the reply must go down to the client again. But that is to be avoided.

Instead, a client (and therefore a cache) may issue a

```
Twant fid try
```

to the server to request ownership of the file. The usage of *try* (a boolean field) is discussed later. The server must reclaim ownership if delegated to other clients (or wait until it expires) and then issue a

```
Rwant vid seconds
```

to the client. Clients implementing this extension are expected to issue a

```
Twant fid
```

as soon as the *Rwant* has been received. This is used by the server to invalidate the lease explicitly, before the time expires. In this case, the server replies with an

[19] This must be fully rewritten to match the currency management algorithms mentioned in section 4. In any case, we can perhaps manage the required message exchanges as a protocol extension. [nemo].

```
Rerror  reclaimed
```

and considers the client aware of the reclamation as soon as another

```
Twant   fid
```

has been issued. A null fid may be used to state that the client does not currently want anything.

To avoid too many requests, the client can set *try* to true in *Twant* to ask for ownership if available and learn that it cannot be gained otherwise. A server that receives a request with *try* set does not reclaim ownership from other clients.

Using this mechanism, the client tries to acquire ownership for the largest subtree where actual ownership requests are wanted. If the entire subtree is not available, the client becomes more shy and asks for subtrees instead. In worst case, actual ownership requests (with *try* set to false) are sent for the few files that really require it.

With this mechanism, caches sitting in the middle points of the tree may retain ownership for subtrees used by their clients. Ownership requests may then be handled by the middle points without disturbing other subtrees not below them in the path from the root server.

Should this scheme be used, a single server (or cache or client) would have file ownership at a time. Therefore, it may assign a new version atomically to a file as needed.

7.2. Extension for multiple file operations.[20]

This extension is intended to let a client:

1. Select all files in a single directory
2. Filter them on attributes, for example, their name, type (DIR or FILE), length, permission etc.
3. Perform certain operations on the (filtered) set files.

The new (elementary) operations added by this extension are *Tforall* and *Tfilter*, that both create a reference to a set of files. Files in a set are ordered, which determines the order in which they are processed. The reference changes the meaning of following elementary file operations within the *Tgroup* as explained below. An error on an operation made to a file cancels processing of following operations just for that file (other files in the same set do not care). The server continues with the first operation for the next file in the whole set. The scheme would be:

---

[20] I changed the text and protocol definition. Both Tforall and Tfilter create a set of files, which can be refered to with a single temporal fid. Primarily to decouple Tskip->Tfilter and the order in the operations. If you do not agree, please change it back to the original proposal. [ptw]
The current description may also enable conditional execution for a particular file (using Tfilter on a regular single file fid), but I did not elaborate on this. [ptw]
Nemo suggested to force ffid=tfid, for each filter and file operation. If a file is filtered, no further filter and file operations are made from the group (on that file). Let's have the initial implementation like this, and explore the more flexible approach when we have more experience with this Tforall implementation. However, I kept the description to reflect this more flexible approach [ptw].
Let's do that. [nemo]

```
Tgroup {
        Tforall fid tfid
        Tfilter fid ffid attrname cmpop attrvalue
        Top1 fid=tfid
        Top2 fid=ffid
        ...
        Topn fid=tfid
        [Tclunk tfid]
        [Tclunk ffid]
}

Rgroup {
        Rforall mfiles
        Rop1[0]
        Rop2[0]
        ..
        Ropn[0]
        Rop1[1]
        Rfilter nops=3
        Rop5[1]
        ...
        Ropn[mfiles-1]
}
```

Tforall creates the set of files set contained in the directory implied by *fid*. The fid cannot be a set of directories. The order of this set is the same as when retrieved by reading the directory. Each file in this set is refered to by a single temporary fid *tfid* and the following operations can use this tfid to perform the operation on each file in the set. After all operations have been applied, the temporary fid is clunked without client intervention (Thus, the *Tclunk* in the example may be included by the client but must be enforced by the server after the last reply). The server returns in the reply message *Rforall* the number of files that are part of tfid set, such that the client can interpret the following replies.

A *Tfilter* request is used to reduce the set of files identified by the fid, but maintain the order of the original set. It compares a file attribute to a given value (only == and != operators are accepted for most attributes but numeric mandatory attributes also understand < and > operators)[21]. The filtered set of files is refered to by a new temporary fid *ffid*. When used after a *Tforall*, it may be used to filter in (and out) some files depending on their attributes. The following operations can also use this ffid to perform operations on the filtered set. The server does not reply for this message.

The Tforall has resulted in an ordered set *M* of *m* files, which are used to execute the succesive (elementary) operations. Each operation that is necessary on this set of files shall be listed in the same group message as the introduced temporary fid only valid within this group message. The operation refers with its fid to either the whole set M identified by tfid, or a filtered set f(M) identified by ffid. This enables selective execution on particular files. In case one or more successive elementary file operation cannot be excecuted, because Tfilter has filtered this particular file out of the set referenced by the operation's fid, the server returns an *Rfilter* with the number of operations that are skipped. This enables the client to determine the responses for individual files. The maximum total number of replies in Rgroup equals 1 + nops*mfiles. In case an error occurs with a particular file and operation, the server returns an Rerror and continues

[21] Other operators may be examined, if we have more experience with the implemented current proposal.

with the next file in the ordered set M.