CptS 464/564
Lecture 18

13th March 2003

Not handing back exams today: I couldn't get them all graded and cross-checked for consistent grading by 10AM yesterday, so they will now have to wait 'til after break.

Instead we will complete the discussion of Chapter 4 by talking about distributed file systems. I'm going to skip the sections in Ch. 4 concerning DCE, WWW and groupware – they are nearly content-free. If we need them as examples later, we will talk about them then.

Finish the last page of the Lecture 17 notes

Distributed file systems

Review of application file system operations: open, read, write, close. OS provides a *file pointer* that is an implicit parameter in each read or write operation. In addition, there is a notion of file names, and directories (note different use of term *directory* here – in terms of discussion last time we would call it a *naming context)*. Directories may be stored in the same data space as files, but need not be.

To use a file, an application first must *open* it, providing a filename and access mode (read/write/append/etc.). The OS uses the name to find the file ID and then checks permissions. If successful a *file descriptor (*more generally a *file handle)* is returned to the application. The file handle is passed to subsequent read/write operations along with a buffer. When finished using a file, an application must *close* it, which allows the OS to release resources associated with the open file (buffers, file pointer, etc.).

Recall that disks are much slower than processors: typical access times (according to Seagate's website describing their products) are 4-10ms. In 4 ms a 1GIPS processor executes 4 *million* instructions. It thus is very worthwhile to cache (buffer) data going to and from disk, making some portion of disk operations unnecessary, thus improving performance of the system.

Mounting: way back when, a file system corresponded to a physical device – usually a disk drive of some kind. What happens if you want more than one disk on your system?

A:, B:, C:, D: solution was adopted in the PC world, but the problem had previously been solved better, using a notion of mounting. Beginning from a single global root namespace (directory) conventionally called /, we choose *mount points* at which additional filesystems are *mounted* or attached. This allows us to have a single, hierarchical namespace that reaches all the available files through the open/close/read/write suite. (With the advent of Windows NT, the mount point philosophy was adopted there as well; drive letters are now points in a namespace onto which filesystems are mounted.)

VFS: what happens when different file systems are layed out differently, thus requiring different code to interpret the bits stored on the disk? A layer of indirection was introduced called the Virtual File System (VFS) to solve this problem. The code that runs in response to, say, a *read* system call is chosen based on the *type* of the filesystem on which the file resides.

With mounting and VFS concepts, adding the ability to use remote files is quite easy, at least conceptually. We just need to implement the VFS operations in terms of the remote file protocol we wish to use. Separating *mount* from *open* has the further desirable effect of taking the expensive operation of figuring out *where* a file resides from the frequent operation of opening files. (This is an example of amortization of expense: mount is expensive, but since it makes many files available in one operation we hope to spread that cost over many opens).

We can immediately anticipate a failing of this approach: recall the discussion (and exam question) about interfaces and contexts in which objects are used: the open/close/read/write suite was designed without taking into consideration the possibility that the files themselves might be remote. As a result, the interface does not take into account problems of partial failure, concurrency, and latency that arise in distributed systems.

Design choices:

- how does the client discover/look up the right server for its file?
- is the directory service distributed or centralized?
- how much data should be sent between client and server with each read/write request?
- Cache at client? at server? both? How can multiple client caches be kept consistent with one another? What kind of consistency is provided?
- Should servers keep track of clients?

We turn now to looking at some distributed file systems and how they have addressed (or not) these problems.

Extreme design choices (Fig. 4.8)

Sun Network File System (NFS)

I disagree with the claim in the book that "it is a striking example of good distributed system design." It might be a striking example of "good enough" distributed system design, but of even that I am doubtful.

Basically, NFS is a relatively straightforward mapping of the Unix file system-related system calls into a distributed setting, with deviations from the system-calls mostly following from one key decision: file servers were to retain no state associated with clients (stateless server model). The goal was to make reliable operation of the server independent of reliable operation of the clients, by making the server's behavior entirely independent of the clients. Servers can be rebooted at will, *but* servers cannot provide any synchronization between clients that might be using the same file.

Consequently, there is no *open* operation in the NFS interface. Instead, clients obtain file-handles that identify files by filesystem and inode on the server. Care is taken to invalidate the handles if an inode is re-used using a generation-number. *read* and *write* operations pass a file offset as well as a file handle along with the data for the operation.

Read and *write* operations take place in chunks of several kilobytes (to amortize the expensive RPC over as much data as possible). Data is cached in the buffer manager of the client OS. Client is responsible for verifying validity of cached data. A messy model, much like normal Unix consistency without locking: files may change at any time and application programs will not find out about the change. Application writes are handled similarly – application writes into OS-managed buffer; some time later the buffer is written to the file server. Interestingly, the NFS protocol says that by default these write RPCs are synchronous with delivery of the data to storage that will persist even if the file server crashes (so-called stable storage). This is a stronger guarantee than is made to Unix application programs, even for local filesystems. It is not clear to me why this provision was considered important. However, it has been the source of many performance problems over the years, and also the source of much revenue for vendors of stable storage hardware that operates at memory speed.

One example of nasty NFS behavior: suppose an application writes a file, closes it, exits. The client OS now issues write operations for the file blocks. OOPS - file system is full. Blocks cannot be written and there is no program running that can possibly compensate for the problem!

Other problems with NFS: client computers that lock up because a file server is down; because a filesystem has changed on a fileserver (the dreaded stale NFS file handle) problem, ...

Decisions in NFS's design mean that it really works well only on local area networks.

Andrew File System (AFS)

Why Andrew? Developed at Carnegie-Mellon University; named after founder Andrew Carnegie. It is now a product of Transarc Corp. which is owned by IBM.

Principle goal: file servers that can support thousands of client workstations. Requires very good caching strategy. Not only must servers not be bothered frequently to read/write data they must not even be asked for cache validation very frequently.

The design was motivated by data gathered from a study of files and file use in an academic environment (the "Andrew benchmark"):

- most files are small
- reads are much more common than writes
- reads are typically sequential
- most updating happens by one user (client) at a time
- once a file is accessed, it is likely to accessed again soon

These observations led to a design that used:

- whole file caching on client disks
- reads and writes happen only from local disk cache
- files remain cached even after close
- updates are written back to the server only after file close
- volume-level (think filesystem level) replication for read-mostly volumes – highly beneficial for read-only volumes. Master volume is updated and then manually *released* to update the replicas.

Considerations: AFS is not nearly so popular as NFS. Partly due to its proprietary nature – NFS was freely available with most Unix systems by the early 1990's. A server could be set up with a single machine that served other duty as well. AFS required an expensive license, and a substantial commitment of hardware – a minimum of 3 machines, often many more. At least as important, AFS clients were heavyweight – because they used disk caching they were hard to integrate with client OSs, and thus using AFS would constrain what versions of an OS you could use to those for which Transarc provided support. Also, though the benchmark reveals the ways that files are *mostly* used, it doesn't reveal how important the minority uses are, and how disruptive to the AFS model – large files, shared files, frequently updated files are used frequently enough that they need to be solidly designed as well as the most common cases.

There is now an OpenAFS project a SourceForge. I don't know anything about it but exploring there would probably reveal interesting details.

Server-side considerations

Mostly looked at client issues so far. On the server side, NFS can be and is implemented over almost any OS and filesystem. AFS defined its own OS and file system layers for the server as well as for the client and was thus heavyweight on the server as well. An NFS server can be implemented as a user-level process without much difficulty, though typically the function is built into the OS for performance reasons.

We saw how the VFS layer allowed an OS to use many different file system types. We might also ask whether a single file system might be viewed by different distributed file system protocols. This turns out to be remarkably difficult. As an example, our very own file server "zeus" serves both NFS and Windows SMB protocol from the same underlying files. Issues

- case sensitive in Unix/NFS; case insensitive for Windows/SMB
- file locking
- different access control models (Unix user/group/other; Windows ACL)

In my opinion, Network Appliance (the creator of zeus) has done remarkably well at providing a useful and practical service in the face of many issues and requirements that are conflicting and essentially unresolvable.

Stateful vs. Stateless servers: in my opinion the NFS stateless model is not ideal – there is too much need for coordinated access to files, *but* the NFS motivation for statelessness is right on the mark. Servers must not be dependent on the proper behavior of clients. MSWindows servers have had problems with this in the past. AFS seems to have made a good set of choices about limiting the impact of misbehaving clients by time-limiting the effects of misbehaving clients. Resulting suggestion: *the not-very-stateful* server.