Distributed Systems

Week 6, Lecture 2

Logistics

- Any questions regarding programming assignment 2?

- Important things coming up:
  - Next Tuesday: Book assignment #2 assigned.
  - 11/18/08: No Class – I will be in Austin at SC08.
  - 11/25/08: Term exam!

  - **Post-Thanksgiving**: Term paper presentations.
    - 15 Students * 2 Days = ~10 min. each.
    - We'll talk more about this more in the next two weeks.
  - **Finals week**: Project due, project presentation. Your chance to show off what you did.

Today

- Distributed file systems.
- Last time, we talked about replication.

- There is quite a bit of conceptual overlap with the issues that arise in file systems and replicated systems.

Basic file systems

- A basic file system (not distributed) provides an abstraction layer between user applications and data stored on a device (disk, memory, etc…).

- In a distributed file system, a similar abstraction layer is provided.
  - Transparent access to files on storage medium, potentially accessed via a network.
  - Provide an interaction mechanism that closely resembles that of local file systems.

Why?

- What is useful about a distributed file system?
- We saw last time that one can address performance, reliability, and fault-tolerance through replication.
- Even without replication, a remote file system is a good choice for management.
  - Imagine an organization with 100 machines that all work with files you need to back up.
  - You can either run around with 100 external drives or tapes, or you can centralize the storage on one machine that you backup and protect.
    - E.g.: RAID – a basic replication-like service at the hardware level.

Parts of a file system

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

Contents

- The FS contains data and metadata.
  - Metadata: Owners, permissions, associations, etc…
- FS translates names to actual physical files.
- Naming is our most common experience with FS's.
  - /usr/local/bin/foo
  - URLs are a generalization of the same hierarchical file naming concept, except they roll into the naming scheme the host of the data, and the protocol required to access it.

Transparencies

- Distributed file systems, where a network sits between the client and the holder of the file, seek to provide certain transparencies:
  - **Access**: Familiar operations make sense. (e.g.: open())
  - **Location**: It shouldn't matter to the user where the files are.
  - **Mobility**: Physical files can be moved on the server without the client being aware of it. (e.g.: Splitting a partition)
  - **Performace**: The file service should attempt to hide performance variability on the file server.
  - **Scaling**: Expansion of the service to handle greater capacities (both w.r.t. storage and client counts) should be invisible to the client.

Critical issue

- How do we deal with concurrency control?
- Potential instance of two users accessing a single file.
  - Maybe updating it.
- How do we deal with this conflict?
- Typically, file systems impose or provide a locking mechanism.

Locking schemes

- Advisory locking
  - Application developers responsible for acquiring file locks.
  - If they don't, data corruption is possible.
- Mandatory locking
  - If a file is locked, the file system forces other programs that try to open the file to try to acquire the lock.
  - FS tries to protect against careless app. writers.

- Rule of thumb:
  - Windows = Mandatory locking
  - UNIX systems = Advisory locking

File locking

- `FileLock` class in Java

- C: (As usual, a bit more tedious)

```
/* open file */
int fd = open("file", O_RDWR);

/* acquire lock */
flock lock = { F_WRLCK, SEEK_SET, 0, 0, 0 };
fcntl(fd, F_SETLK, &lock);
```

Heterogeneity

- Distributed file systems can help with heterogeneous systems.
  - Different hardware architectures.
  - Different operating systems.

- NFS is an example of a distributed file system that provides this.
  - In fact, NFS was one of the main users of SunRPC and used the external data representation we talked about many weeks ago to deal with the issues of different data representations.

Security

- Distributed file systems also must pay careful attention to security.
- On a single machine with a non-shared file system, user permissions and access control lists are typically sufficient.
- Things get a bit trickier when the client and server are two different machines.
  - Authentication of requests from the client.
  - Signatures on requests and responses (to defeat impersonation attacks).
  - Potential encryption (e.g.: transport over SSL).

NFS

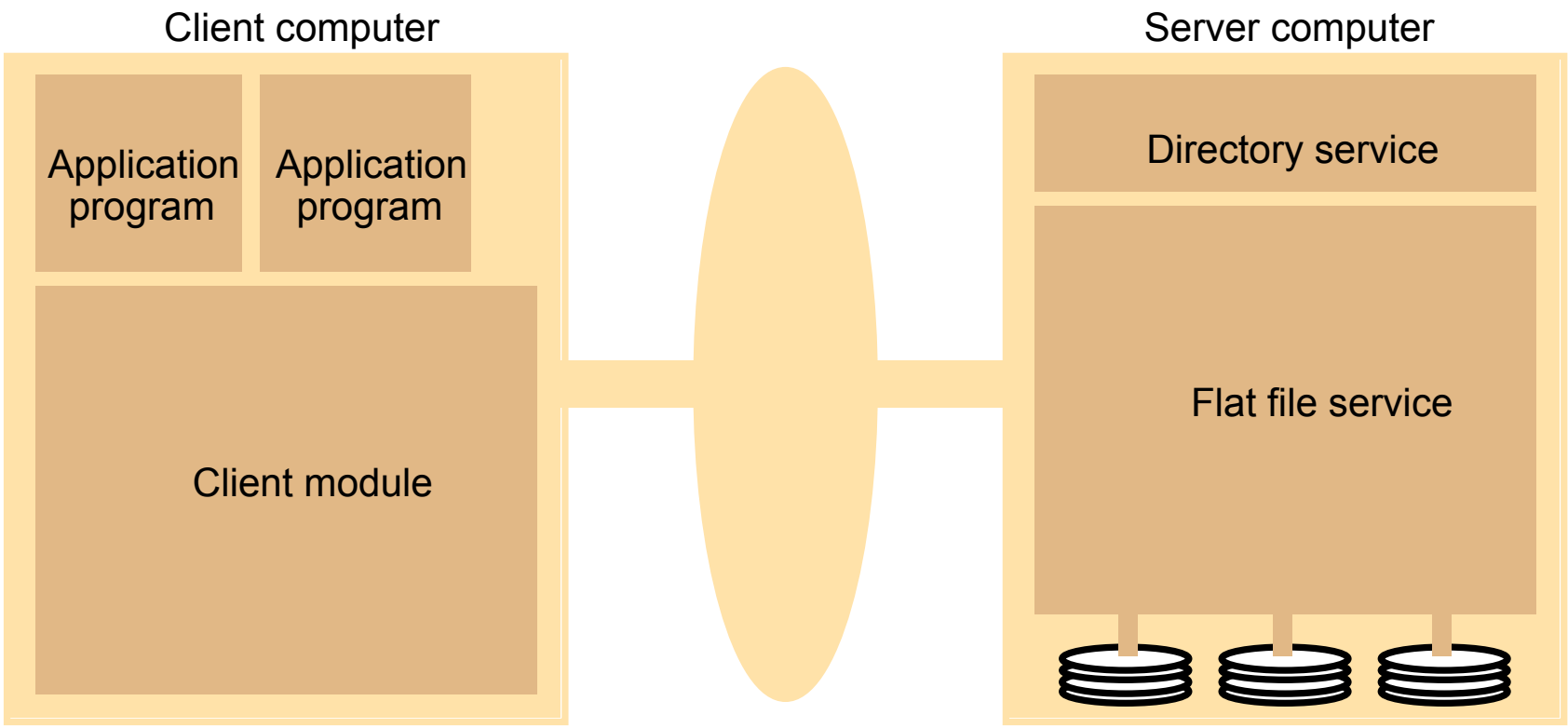
- NFS is one of the oldest distributed file systems.
- Originated in the mid-1980s at Sun Microsystems.
- Released as an open standard early on to encourage third party products adopting it.
- Based on SunRPC with XDR.

- Designed with client/server symmetry – a client can become a server to share it's files.
- Big emphasis on OS and hardware neutrality.

AFS

- Andrew File System
- Developed at CMU in the mid/late 1980s.
- Focus on reducing network overhead.
  - Based on whole-file caching.
- Originally implemented on top of BSD and Mach, but has made it's way into other operating systems since.

Components of a File Service



Components

- **Flat file service**: Map Unique file IDs to concrete files.
- **Directory service**: Map names to UFIDs. Also maintain logical structure (e.g.: directories).
- **Client module**: Interface provided to user applications. E.g.: UNIX-style file operations for NFS.

- Interface between client module and file service uses some RPC layer to make functions like read(), write(), create(), delete(), get/setAttribute() available to clients.

Security mechanisms

- Server side authentication required.
- Want to avoid stateful servers.
- So, two approaches:
  - 1. Authenticate with server, get a token back representing successful authentication and pass that with subsequent interactions.
  - 2. Pass user information with each call and authenticate for each operation.
- Clearly #2 is easier at the cost of some low overhead. NFS and AFS choose #2.

Directory services

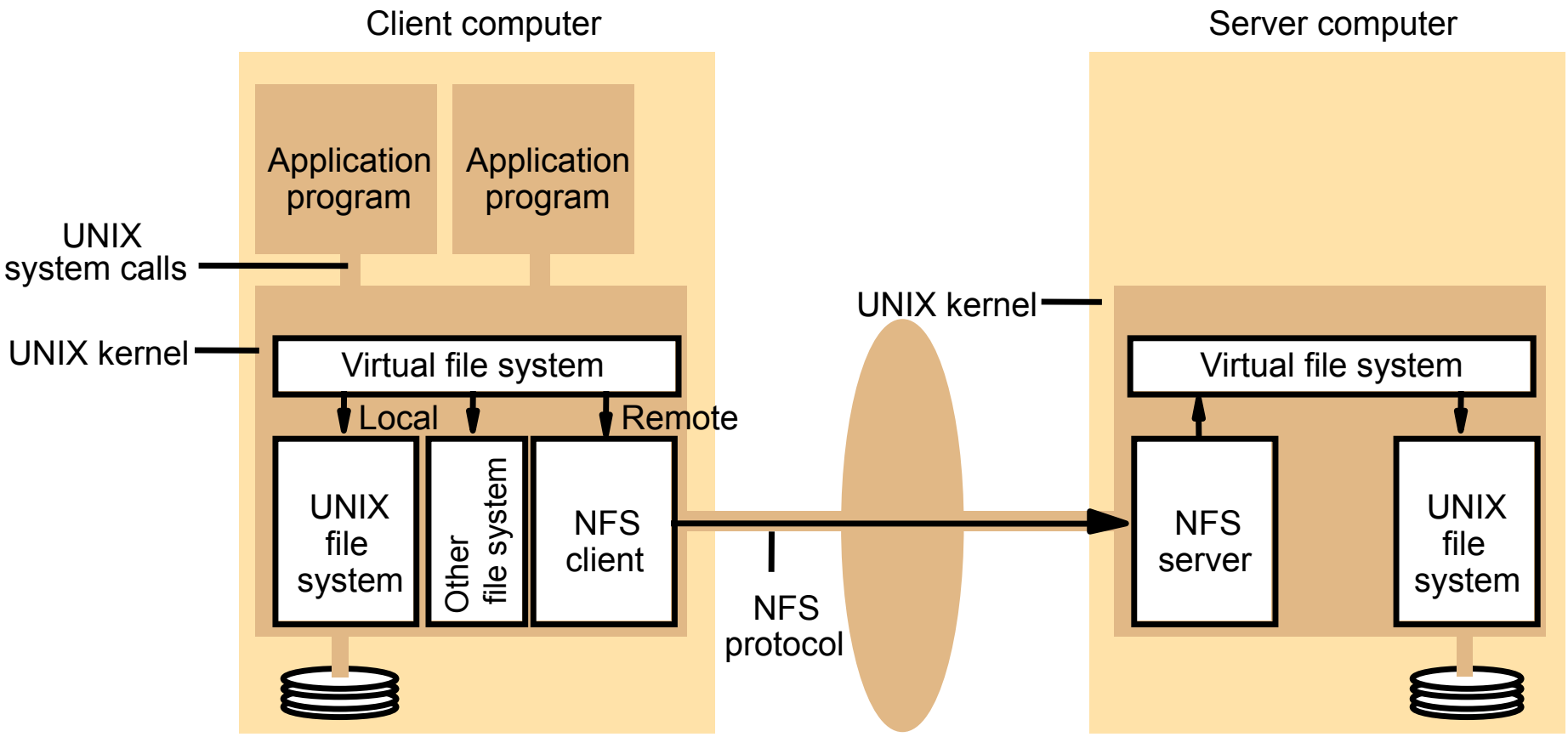
- Directory service has a simple RPC interface exposed to the client.
  - Lookup, AddName, UnName, GetNames.

- Since interface with flat file service is based on fileIDs, clients must interact with the directory service to resolve names into fileIDs.
  - Also to discover set of possible names (e.g.: ls).

Concrete example: NFS

- Based on SunRPC over either TCP or UDP.
- Server accepts connections from any client, responds only if they authenticate validly.
- Uses a virtual filesystem module to hide the implementation of the file system from the client.
  - Other file systems use the VFS module too, not just NFS.
  - These other file systems are not necessarily distributed.

NFS Architecture



NFS mounting services

- Servers export a set of local file systems to clients.
- Clients mount these, just like any other file system, although with additional information in the mount command to indicate where the file server is.
- Supports hard mounting and soft mounting.
  - **Hard mounting**: Clients block until request is fulfilled, retrying until it finishes.
  - **Soft mounting**: No blocking if requests not fulfilled after a few retries, FS indicated failure.
- Hard mounting more common. Why? Sloppy application developers don't always check error conditions when accessing files, so many apps run unpredictably when failures occur.
- Hard mounting causes familiar NFS-related hangs. Next time an NFS server wedges up on you, blame sloppy programmers.

Automounting

- When a remote file system is accessed before it is mounted, the automounter will automatically issue the request to the server to perform the mount.
- After a period of inactivity, the automounter can unmount the filesystem.
- Implementation of automounting may vary, either through the use of user-space processes and symbolic link tricks, or actual mounting via kernel-level automounters.

Caching

- One of the most important parts of NFS: caching.
  - This makes NFS a viable distributed file system from a performance perspective.
    - NFS isn't perfect though. More on that later.
- File systems typically cache data in memory.
  - Exploit locality of operations to avoid costly accesses to slow disk.
- NFS provides this too.
- Server side caching is implemented just like regular file system caching, to avoid costly disk access.

Caching

- Complications arise with write operations though.
- Clients who write to the filesystem need some assurance that their writes actually made it to nonvolatile store.
- Two methods:
  - **Write-through**: When a client writes, the data is stored in the server cache AND written to disk before a response is issued.
  - **Write/commit**: Write operations go to cache only. When file closed, or explicit commit command is issued, then data is pushed to nonvolatile store.
- Commit added in NFSv3.
- Write through addresses independent failures. If server crashes, clients are allowed to continue assuming the data actually was safely stored.

Caching

- Client side caching is possible too.
- Complication is that clients must be able to determine if their cached copies of files are out of date.
- Timestamps are used. Cached data has two timestamps:
  - Tc : Time when cache entry was last validated.
  - Tm : Time when entry was last modified on the server.
- At any given time T, an entry is considered valid if T-Tc is less than some "freshness interval" t or the modification time at the server matches that recorded in the cache.
- So what does this imply?

Caching

- In the limit, smaller freshness intervals cause NFS to provide one-copy consistency.
  - Lower t = higher network and server load.
- Whenever a cached entry is used, the client first checks to see if the freshness constraint holds.
- If so, it then contacts the server to check the modification timestamps.

- Does not guarantee consistency. This is ok though, as most applications will rarely suffer from accessing inconsistent data.
- This is a hard lesson to learn sometimes, when people use the filesystem assuming consistency.
  - Beware when developing apps to not assume filesystems provide consistency!

Caching

- Writes are kept in the local cache until they are flushed out to the server.
  - Explicit close
  - sync operation
  - Daemons to help implement read-ahead and delayed-write will also cause flushes to occur.

NFS locking

- NFS does not guarantee that locking is available via the server. Older servers sometimes don't support it.
  - Mounting requires you to specify "nolock" option.
- Specialized daemon (lockd) is used to forward fcntl() lock requests to the server.
  - lockd is actually part of the RPC layer. (rpc.lockd)
- Only support for advisory locking.
- Client and server lockd architecture has provisions to recover from crashes.

NFS performance

- For most general use cases, NFS performance is acceptable for users.
- Observations of real workloads show higher incidence of reads versus writes, so write-through overhead isn't that bad.
- Frequent network traffic to check timestamps (getattr) due to caching.
- Large number of "lookup" calls to the directory service expected.
- Some tuning can be performed for caching to find freshness timeout values that fit a given workload and client environment.

AFS

- AFS built for scalability, potentially in wide-area network environments.
  - Whole-file operations are the basis of AFS.
    - Serving of whole files.
    - Caching of whole files.
- Shared, infrequently updated files will be cached in clients.
- Very large caches will let users acquire most of the files they will need in cache.

Design motivations

- AFS designed based on observed, measured workloads.
- Observations:
  - Files typically tiny.
  - Reads far more common than writes.
  - Sequential access common, random access rare.
  - Reads/write most commonly originate from only one user.
    - Shared files typically only modified by one user.
  - Files are referenced in bursts.
    - If a file has recently been referenced, it is likely to be referenced again soon.

More recent interesting FS developments

- FUSE: Filesystem in Userspace
- Allows for new file systems to be installed by users without requiring direct kernel manipulation.
  - No recompilation, no kernel modules.

- Some of the more interesting FUSE-based filesystems out there are distributed file systems.
- My favorite: SSH-fs.
  - Extremely useful.
  - What does it do? Allows me to bring a remote file system into my locally addressable file-space using SSH only.
  - So remote side doesn't have to be running anything special.

Plan9

- Developed at Bell labs by the UNIX designers.
- Goal of plan9 was to distribute tasks amongst sets of machines instead of assuming single big machines.
- Based on name-space concept.
  - Filesystem is a hierarchical namespace.
- Unlike UNIX, everything is a file.
  - Devices, networks, graphics, etc…
- A program that provides services is a file-server, and it introduces files into the namespace related to operations it provides or performs.
- Based on a protocol called 9P.
  - Implementation of 9P available in Linux called v9fs.