Distributed File Systems
This material is partially covered in Chapter 12 of Coulouris, Dollimore, Kindberg, and Blair.
Synchronous vs. Asynchronous

- Execution speed
  - synchronous: bounded
  - asynchronous: unbounded
- Message transmission delays
  - synchronous: bounded
  - asynchronous: unbounded
- Local clock drift rate:
  - synchronous: bounded
  - asynchronous: unbounded

These definitions are abstracted from the textbook, page 64.
Failures

- **Omission failures**
  - something doesn't happen
    - process crashes
    - data lost in transmission
    - etc.

- **Byzantine (arbitrary) failures**
  - something bad happens
    - message is modified
    - message received twice
    - etc.

- **Timing failures**
  - something takes too long

This material is from Section 2.4.2 of the text.
Detecting Crashes

- Synchronous systems
  - timeouts
- Asynchronous systems
  - ?
- Fail-stop
  - an oracle lets us know
Note that in the typical distributed file system, servers are not replicated. However, the data might be. In fact, one server, the “file server”, might handle meta data, and data might be handled on other servers.
DFS Components

- Data state
  - file contents
- Attribute state
  - size, access-control info, modification time, etc.
- Open-file state
  - which files are in use (open)
  - lock state
Possible Locations

Client
- data cache
- attr cache
- open-file state

Server
- data cache
- attr cache
- open-file state
- local file system

---

CS 138  XV–9  Copyright © 2017 Thomas W. Doeppner. All rights reserved.
In Practice …

- Data state
  - NFS
    - weakly consistent
    - less weak if program uses locks
  - CIFS and DFS
    - strictly consistent
- Lock state
  - must be strictly consistent
Thursday morning, November 17th
At 7:00 a.m.
Maytag, the department’s central file server, will be taken
down to kick off a filesystem consistency check.
Linux machines will hang.
All Windows users should log off.
Normal operation will resume by 8:30 a.m. if all goes well.
All windows users should log off before this time.
Questions/concerns to problem@cs.brown.edu

(Note that the November 17 in question was in 2005.)
What we’re accustomed to with local file systems is that, in the event of a crash, everything goes down. This is simple to deal with.
In a distributed system, if the server crashes, there is no inherent reason for clients to crash as well. Assuming there was no damage done to the on-disk file system, client processes might experience a delay while the server is down, but should be able to continue execution once the server comes back up, as if nothing had happened. The crash of a client computer is bad news for the processes running on that computer, but should have no adverse effect on the server or on other client computers. We'd like the effect to be as if the client processes on the crashed computer had suddenly closed all their files and terminated.
In Practice …

- NFS version 2
  - relaxed approach to consistency
  - handles failures well
- CIFS
  - strictly consistent
  - intolerant of failures
- DCE DFS
  - strictly consistent
  - sort of tolerant of failures
- NFS version 4
  - either relaxed or strictly-consistent
  - handles failures very well
NFS consists of three component protocols: a mount protocol for making collections of files stored on servers available to client nodes, the NFS file protocol for accessing and manipulating files and directories, and a network lock manager (NLM) for locking files over the network and recovering lock state after failures.

An NFS server gives its clients access to one or more of its local file systems, providing them with opaque identifiers called file handles to refer to files and directories. Clients use a separate protocol, the mount protocol, to get a file handle for the root of a server file system, then use the NFS protocol to follow paths within the file system and to access its files, placing simple remote procedure calls to read them and write them. A third protocol, the network lock manager protocol (NLM), was added later and may be used, if desired, to synchronize access to files. All communication between client and server is with ONC RPC.
Note that the server contains no state information about the clients – such information is kept solely on the individual clients.
Consistency in Basic NFSv2

Data cache
- file x block 1
- file x block 5
- file y block 2
- file y block 17

Attribute cache
- file x attrs
  - validity period
- file y attrs
  - validity period
More …

- All write RPC requests must be handled synchronously on the server
- Close-to-Open consistency
  - client writes back all changes on close
  - flushes cached file info on open
Since NFS servers are stateless, the crash and restart of a client has no effect on them. Thus client crash recovery involves no actions on NFS’s part.
Recovery from a server crash is easy for the server — it merely resumes processing requests. The client is generally delighted when the server recovers; its concern is what to do while the server is down.

Client machines may choose to “mount” NFS file systems in one of two ways: soft mounts and hard mounts. With a soft mount, if the server crashes, then attempts to access the down file system fail — a “timed-out” error code is returned. This is often not terribly useful, since most applications are not equipped to deal with such problems. In the other approach (the hard mount) if the server crashes, then clients repeatedly re-try operations until the server recovers. This can try the patience of users, but it doesn’t break applications.
File Locking

- State is required on the server!
  - recovery must take place in the event of client and server crashes
- Locking Protocol is independent of the File Protocol
  - locking is advisory
  - one can lock a file and ask if a file is locked
  - not required to honor locks
    - may read/write a file locked by others!
The stateless approach clearly doesn’t work if one is concerned about locking files. For this, NFS employs a separate lock protocol. Each participating machine runs two special processes: a lock manager, typically known as the lock daemon, and a status monitor, typically known as the status daemon. The lock daemons manage the locking and unlocking of files; the status daemons help cope with crashes. When an application on a client machine attempts to lock a remote file, it contacts the local lock manager which places an RPC to the lock manager on the server, which locks it there.

If a client crashes, the server's status daemon unlocks all of the files that the client had locked. The only difficulty here is determining whether the client has crashed — perhaps it is merely slow. The approach used in NFS is that the client, when it comes back to life, announces to the server that it has been down. Of course, human intervention may be required if the client is down for a long period of time.

If a server crashes, it loses all knowledge of which files are locked and by whom. So, when it comes back up, it must ask the clients to tell it which files they had locked. Upon receipt of this information, it restores its original state and resumes normal operation.
NFS Version 3

- In use at Brown and in most of the rest of the world
- Basically the same as NFSv2
  - improved handling of attributes
  - commit operation for writes
  - various other things
CIFS

- Common Internet File System
  - Microsoft’s distributed file system
- Features
  - strictly consistent
- Not featured …
  - depends on reliability of transport protocol
  - loss of connection == loss of session
History

- Originally a simple means for sharing files
  - developed by IBM and called server message block protocol (SMB)
  - ran on top of NetBIOS
- Microsoft took over
  - renamed CIFS in late 1990s
  - uses SMB as RPC-like communication protocol
    - runs on NetBIOS
    - usually layered on TCP
    - sometimes no NetBIOS, just TCP
Consistency vs. Performance

- Strict consistency is easy …
  - … if all operations take place on server
  - no client caching
- Performance is good …
  - … if all operations take place on client
  - everything is cached on client
- Put the two together …

∅
- or you can do opportunistic locking
Opportunistic locks are used by CIFS.
Back to NFS

- File system name space
  - how is distributed file system perceived on clients?
- Cross-computer links
  - how are files on other computers referred to?
Servers divide their files up into disjoint collections called *file systems*, each of which contains a rooted directory tree naming all of its members. Servers, as specified in local configuration files, specify which file systems, or subtrees within a file system, are available to which clients. A client can then *mount* a remote file system. This entails superimposing the root of the remote file system on top of a directory in the client’s current naming tree. The root of the remote file system (the mounted file system) effectively replaces the mounted-on directory. Thus the remote file system is attached to the client’s naming tree at the mounted-on directory (and the previous contents of that directory are invisible as long as the remote file system remains mounted). The mount protocol provides some security (by restricting which clients are allowed to mount a file system) and gives the client node the means for fitting remote file systems into its file-system name space.
File Handles

- Servers provide opaque file handles to clients to refer to files
  - contents mean nothing to clients
  - identify files on server
- Clients contact server via mount protocol to obtain file handles of roots of exported file systems
The file handle is used to identify the file to the server for all subsequent operations, including reading and writing. It consists of three items: the remote file-system ID, the file ID (relative to the file system — for example, this might be the inode number), and a generation number. The first two items might seem sufficient to identify a file, but the third component is necessary: Suppose that you have opened a file, but, while you have it open, someone else deletes the file. Furthermore suppose that a new file is created that reuses the inode of the original file. If the file handle consisted only of the remote file-system ID and file ID, there would be no way for it to distinguish between the old file (which no longer exists) and the new file (which has the same file ID (or inode) as the old file). The generation number is an integer that is stored with the inode and that is incremented when the file associated with the inode is deleted. Thus each use of the same inode (and hence file ID) has a different generation number, and thus, in our example, the server can determine that the file referred to by your file handle doesn’t exist any more. NFS’s straightforward way of telling this to you is to print the message “stale file handle” on your console.
This slide shows a stylized directory hierarchy on a server, split into its component file systems. From the point of view of applications running directly on the server, there is just one integrated tree. However, the server exports the file systems independently to its clients.
Here a client has mounted the server file system rooted at /B onto its directory /C2. Thus, from the client’s perspective, the previous contents of directory C2 is replaced with the server’s /B. On the server, one could follow the path /B/F/K/S, but this path crosses a mount point on the server. The corresponding path on the client would start: /C2/F/K. However, the client would not see a file system mounted at K — the mount point exists only on the server.
Now the client mounts the server’s file system /B/F/K on the client’s directory /C2/F/K; the client can now follow the path /C2/F/K/S.
Local vs. Global Namespace

- Local namespace
  - each host configures its own file-system namespace
  - NFS clients each mount the appropriate remote file systems
- Global namespace
  - all hosts share the same namespace
  - not done in early NFS
Mount Protocol Problems

• Local namespaces don’t work
• Achieve global name space by having each client mount everything consistently
  • giving each client a table listing all possible mounts is administratively difficult
  • performing all possible mounts is time consuming
    • mounting is a “heavyweight” operation
Rather than this …

```
  dev  etc  home
```

```
max  louisa  twd  carlos  rohil  rodrigo  atty  haris  ishan
```
... this

dev  etc  home

automount

database

Autofs
To provide something resembling a global namespace, the *network information system* (NIS, originally known as “Yellow Pages,” until Sun discovered the name was trademarked by someone else) is used to hold information (in the form of an NIS map) describing the layout of the namespace, in particular which servers provide which file systems and where they should be mounted. NIS provides this information in the form of a completely replicated distributed database.

Automounting is the notion of having clients delay mounting remote file systems until they’re needed, then using information from NIS to determine what should be mounted.

**Automounting: 2000**

- Maintain description of global namespace in global database: NIS
- Do mounts only when needed
- Automount times out after period of unused
Automounting: 2017

- Global namespace maintained in LDAP database
  - lightweight directory access protocol
    - vendor neutral
  - everything mounted at boot time
    - fewer, but larger, file systems
  - no timeout
DFS (DCE’s distributed file system) provides a cell-wide file system (a cell is a potentially large collection of machines under the same administration), organized into a cell-wide directory hierarchy. This hierarchy is logically partitioned into pieces called filesets. These filesets are stored on the DFS servers, using the server’s local file systems (e.g. FFS). This mapping of fileset to server is stored in a distributed database known as the fileset location database (FLDB).

Clients contact the FLDB to determine the locations of filesets. To minimize network use, server load, and average client access time, the clients buffer portions of files on the clients’ local disks.
Filesets are named in a space that is independent of the directory hierarchy. This fileset name space is mapped via a distributed database called the fileset location database into the servers that manage them.

DFS mount points are implemented as symbolic links, where the value of the link is the name of the fileset mounted in the directory. A client traversing such a mount point looks up the fileset name in the fileset location database, which tells it which server hosts the fileset. The client then contacts that server to follow the rest of the path.

The effect is that DFS provides a single global name space that is shared by all clients. Unlike NFS, the name space isn’t assembled on each client, but is built into the directory hierarchy.
DFS maintains a cache of recent file requests on disk (or, for diskless workstations, in memory). DFS file servers hand to a client an object known as a token (the dark ovals in the slide), which informs a client that its cached copy of the file (within a range of bytes) is (still) valid. There are a number of different types of tokens, e.g. read tokens and write tokens, depending upon what the client is doing with the file. As long as the client’s cache manager holds that token, operations on the file can be satisfied locally without any network requests for data. Tokens are revoked when a request comes into the file server that is incompatible with the tokens held by clients, e.g. a write request that changes data on the server while clients hold read tokens for the same file and same byte range.

Files are copied between DFS file servers and DFS clients in units called chunks. The default chunk size for clients with caches stored on disk is 64K, but it can range from 8K to 256K.
DFS uses tokens to ensure Unix single-system semantics across a distributed environment.

A token can be thought of as a certificate, manufactured by the file server and delivered to the client, that gives the client a set of rights to perform certain specified operations in its cache. Tokens contain a type field that specifies the particular operation (DATA_READ, DATA_WRITE, STATUS_READ, STATUS_WRITE, LOCK_READ, LOCK_WRITE, OPEN_EXCLUSIVE, OPEN_READ, OPEN_WRITE, OPEN_SHARED, OPEN_DELETE, OPEN_PRESERVE, SPOT_HERE, SPOT_THERE). In addition, DATA and LOCK tokens contain a byte range and apply only to data within that range.

Two tokens are said to be compatible if they can be held by different clients simultaneously, for example, two DATA_READ tokens for the same range of bytes in the same file. Tokens are said to be incompatible if the semantics of the two tokens conflicts. For example, if a client requests a DATA_WRITE token for a particular range of bytes in a file, any outstanding DATA_READ or DATA_WRITE tokens must first be revoked and returned to the file server.
In the next few slides we go through an example in which two clients are sharing a file. At the moment, both clients have read tokens for a portion of the file in their caches and can read without interference. However, one of the clients desires to modify the file. In response to the user’s *write* request, the DFS code on the client sends a request to the server asking for a write token.
If one client is to be modifying the file, then, for consistency, no other clients should be reading the file. Thus the server sends a *revoke* request to the other client, demanding that it give back its read token for the file.
The server now grants write access to the original client by giving it a write token for the desired portion of the file.
Now the other client decides to read from the file again.
The server must now send a revoke request to the first client, asking for both the write token and the modified portion of the file.
Finally, a new read token is granted to the second client.
Since DFS must maintain a lot of state information (e.g., which tokens are out), its crash recovery is much more complicated than Basic NFS’s. Three independent things could go wrong:

- A client could crash: thus the server will need to reclaim all the tokens that were held by the client.
- A server could crash: token information is not held in non-volatile storage. It must somehow be recreated when the server comes back up.
- The network could fail: though both client and server remain up, neither can communicate with the other.

There are two features that we would like DFS to provide:

- The client should be able to use its cache even if the server is down or not accessible.
- The server should be able to revoke tokens from a client if the client is down or not accessible.

If either the server or the client has crashed, then providing these two features is not difficult. But if a network outage occurs and the server and client become separated from each other but both continue to run, then the two features conflict with each other.

DFS is forced to use a compromise approach (there is no other alternative). If the client cannot contact the server, then it will continue to use its cache until its tokens expire — they are typically good for two hours, though they are normally refreshed every minute or so. However, if the server is actually up and running, but is somehow disconnected from the client, then the server might want to revoke the tokens. If the server has no need to revoke tokens, then it does nothing. But if some other client that is communicating with the server attempts an operation that conflicts with the unresponsive client’s tokens, then the server is forced to take action.
If the server hasn’t heard from the client for a few minutes, then it can unilaterally revoke the client’s tokens. This means that when the client does resume communication with the server, it may discover that not only are some of its tokens no longer good, but some of its modifications to files may be rejected.

To protect client applications from such unexpected bad news, the client-side DFS code will cause attempts to modify a file to fail if it has discovered that the server is not responding. A client program can take measures to deal with this problem by repeatedly retrying operations until the server comes back to life.
Another way of stating the last bullet is that it’s the weak semantics of (stateless) NFSv2 and v3 that makes the hard mount possible.
NFS Version 4

• Better than …
  – NFS version 2
  – NFS version 3
  – CIFS
  – DFS
  – (why aren’t we running it?)
NFSv4: Why?

- Problems with NFSv3
  - doesn’t provide exact Unix file semantics
  - doesn’t support mandatory locks
  - doesn’t cope with byzantine failures

There are other problems, but these are the ones we talk about in this course.
For exact Unix semantics, the server must know how many file descriptors, on all clients, refer to each file. This has to do with how files are deleted. The “rm” command in the shell removes a link to a file from a directory. Once there are no directories referring to a file, it is deleted, except if at least one process (perhaps on a client) has it open. A file won’t be deleted until it’s the case that both there are no directories that refer to the file and no process has the file open.
State Recovery

- Server crash recovery
  - clients reclaim state on server
    - grace period after crash during which no new state may be established
- Client crash recovery
  - server detects crash and nullifies client state information on server
Coping with Non-Responsiveness

- Leases
  - locks are granted for a fixed period of time
    - server-specified lease
  - if lease not renewed before expiration, server may (unilaterally) revoke locks and share reservations
    - most client RPCs renew leases
  - clients must contact server periodically
    - if clientid is rejected as stale, then server has restarted
    - server's grace period is equal to lease period
As far as client 1 is concerned, the server crashed at step 2 (since the client couldn’t contact it) and didn’t restart until step 5. If the server, at step 6, has no information about its lock state prior to the crash, it cannot recognize that client 1 should not be allowed to reclaim its lock in step 6.
Coping …

• Possibilities
  1) server keeps all client state in non-volatile storage
  2) server keeps all client state in volatile storage and refuses all reclaim requests (effectively emulating CIFS)
  3) something in between …
Compromise

- Keep enough client state in non-volatile memory to know which clients were active at time of crash
  - will honor reclaim requests from these clients
  - will refuse reclaim requests from others

- What to keep:
  - client ID
  - the time of the client’s first acquisition of a share reservation or lock after a server reboot or client lease expiration
  - a flag indicating whether the client’s most recent state was revoked because of a lease expiration
DFS ≠ LFS

• Servers might give up on non-crashed clients
  – clients may lose locks
  – clients may lose files
  – NFSv4 attempts to make such things unlikely