Distributed File Systems  Part 2
CIFS

• Common Internet File System
  – Microsoft’s distributed file system
• Features
  – batched requests and responses
  – 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
CIFS Example

```c
char buffer[100];
HANDLE h = CreateFile(
    "Z:\dir\file",
    GENERIC_READ|GENERIC_WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    NULL
);

ReadFile(h, buffer, 100, NULL, NULL);
...
SetFilePointer(h, 0, NULL, FILE_BEGIN);
WriteFile(h, buffer, 100, NULL, NULL);
CloseHandle(h);
```
Share Mode

• When opening a file
  – specify intended use of file (desired access)
    - read, write, or both
  – specify restrictions on how others may use the file (controlled sharing)
    - read, write, both, or none
### SMB Example

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMB_COM_NEGOTIATE → server</td>
<td>Client gives server list of SMB versions it is willing to use.</td>
</tr>
<tr>
<td>client ← SMB_COM_NEGOTIATE</td>
<td>Server chooses one and informs client. It also sends the client a security challenge.</td>
</tr>
<tr>
<td>SMB_COM_SESSION_SETUP_ANDX → server</td>
<td>Client sends encrypted challenge to server, along with user’s identity.</td>
</tr>
<tr>
<td>client ← SMB_COM_SESSION_SETUP_ANDX</td>
<td>Server verifies identity and fills in UID field.</td>
</tr>
<tr>
<td>SMB_COM_TREE_CONNECT_ANDX → server</td>
<td>Client identifies the share it wants to use.</td>
</tr>
<tr>
<td>client ← SMB_COM_TREE_CONNECT_ANDX</td>
<td>Server fills in security fields and TID field.</td>
</tr>
<tr>
<td>SMB_COM_OPEN_ANDX → server</td>
<td>Client sends path name of file.</td>
</tr>
<tr>
<td>client ← SMB_COM_OPEN_ANDX</td>
<td>Server performs access checks (based on UID) and if ok, responds by filling in FID field.</td>
</tr>
<tr>
<td>SMB_COM_READ → server</td>
<td>Client requests data from file.</td>
</tr>
<tr>
<td>client ← SMB_COM_READ</td>
<td>Server returns data.</td>
</tr>
<tr>
<td>SMB_COM_CLOSE → server</td>
<td>Client closes file.</td>
</tr>
<tr>
<td>client ← SMB_COM_CLOSE</td>
<td>Server returns success.</td>
</tr>
<tr>
<td>SMB_COM_TREE_DISCONNECT → server</td>
<td>Client disconnects from share.</td>
</tr>
<tr>
<td>client ← SMB_COM_TREE_DISCONNECT</td>
<td>Server returns success.</td>
</tr>
</tbody>
</table>
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 …
  Ø
  – or you can do opportunistic locking
Opportunistic locks are used by CIFS.
Refinements

- Two types of op locks
  - level I
    - client may modify file
  - level II
    - client may not modify file

- Client must request op lock
  - may request only level-I
- Server may deny request
  - may give level-II instead
- Requests done only on open
Read-Only Access

• Why request a level-1 op lock if file is to be open read-only?
  – typical Windows application always opens read-write
  – may never get around to writing …
Exclusive oplocks are used only on older clients that can’t handle batch oplocks.
Suppose a client has an op lock on a file. It has made changes to the file that it hasn’t sent to the server. The server crashes and restarts. The client gets a new op lock. Does it make sense for the client to now send its cached changes to the server (so that the client application is oblivious of the crash)?

a) No, this would never make sense
b) Yes, it would make sense only if the client had exclusive access to the file (via a share mode)
c) Yes, it would make sense only if the client didn’t have exclusive access to the file
d) Yes, it would always make sense

By "make sense", we mean that it would make sense to modify the protocol so this is done automatically.
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. 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.
Opening a File Read-Only

1) 
\[ \text{Client} \rightarrow \text{open}(A, \text{read\_only}) \rightarrow \text{Server} \]

2) 
\[ \text{Client} \rightarrow \text{File A:}\ \text{open\_read} \rightarrow \text{granted} \rightarrow \text{Server} \]
Reading Data

3) Client

File A: open_read

read(A)

Server

4) Client

File A: open_read

granted; data

File A: data_read

Server
Another Client Opens Read-Write

5) Client opens file A for read-write.

6) Client grants write access to file A.
**Second Client Reads**

7) Client opens file A with `open_read`, reads data, closes file A with `open_write`. Then, client sends `read(A)` to the server.

8) Server grants data to the client, allowing it to open file A again with `open_read`, read data, close file A with `open_write`.
Second Client Attempts Write

9) Client
   File A: open_read
   File A: data_read
   write(A)
   Server
   File A: open_write
   File A: data_read

10) Client
    File A: open_read
    File A: data_read
    revoke(data_read, A)
    Server
    File A: open_write
    File A: data_read
Permission Granted

11) Client

File A: open_read

granted

File A: open_write

File A: data_read

File A: data_write

Server
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 3: What’s Wrong?

- Works poorly through firewalls
- Doesn’t do mandatory locking
- Susceptible to network problems over the Internet
- Doesn’t handle Windows clients
NFS Version 4

- Better than ...
  - NFS version 2
  - NFS version 3
  - CIFS
  - DFS
  - (why aren’t we running it?)
Firewalls present a challenge to the earlier versions of NFS: as explained in the slide, NFS relies on dynamically chosen port numbers, which are likely to be blocked by firewalls. (Note that this doesn’t apply to the NFS protocol itself, which uses the well known port number 2049, which firewalls can be configured to accept.) Not only must client requests to the server use dynamic port numbers, but the NLM_GRANTED callback procedure (used in the network lock manager protocol) requires that the server contact the client via a dynamic port.
NFSv4 provides a new approach for handling exported file systems. As in earlier versions, however, the server has an exports list indicating which file systems are exported to whom.
What’s new is that clients see a single hierarchy on the server, made up of the exported file systems along with the minimal set of directories required to link these file systems to the server’s root. Thus, in our example, clients see the exported file systems along with the directories /, /vol, and /backup. Since the directories /admin and /vol2 aren’t necessary for linking the exported file systems to the root, they are not visible to the clients. Similarly, no non-directory files in /, /vol, and /backup are visible to the clients. This collection of directories made visible to clients solely to link the exported file systems to the root is known as a *pseudo file system*. It behaves, from the client’s perspective like any other file system, though no part of it is visible except that which is necessary for linking the exported file systems.
Server State

• It’s required!
  – share reservations
  – mandatory locks
• Hierarchy of state
  – client information
  – open file information
  – lock information
Note that there’s a hierarch of state information. Thus if the client crashes, everything related to that client can be nullified. If an open owner terminates, then just what’s related is nullified. Note that an “open owner” represents all processes sharing the opening of a file.
Locking

- Requires additional state on server
  - must be reestablished if server crashes
  - must be removed if client crashes
- For mandatory locking, read/write calls require holding of appropriate locks
  - client must supply "lock owner" with lock requests and read/write requests
  - server must verify that read/write caller owns lock
- Blocking Locks
  - client application must be notified when lock is available
Implementing Locks Right …

• Handle both Unix and Windows
• Get the semantics right
  – both advisory and mandatory
  – who owns a lock?
• Handle failures sanely
• Make it efficient
  – client-side caching where possible
• Make it doable
  – if lock not currently available
    – server might not be able to send callback
      • client polls server
Mandatory Locks (1)

- Just like advisory locks, but you can’t ignore them
  - require state on server
  - state is recovered after a server crash
- Nothing more to say …
  - (wrong …)
Note that NFSv4 is to work with both Unix and Windows clients.
The client determines who the lock owner is, based on Unix or Windows semantics (or perhaps a different OS’s).
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
Note that, for Unix clients, it’s rare to have lock state. Thus for most applications, NFSv4 behaves like NFSv3.
Pathological Network Problems

1) Client 1 obtains a lock on a portion of a file
2) There’s a network partition such that client 1 and server can no longer communicate
3) The server crashes and restarts
4) Client 2 obtains a lock on the same portion of the same file, modifies the file, and then releases the lock
5) The server crashes and restarts and the network partition is repaired
6) Client 1 recontacts the server and reclaims its lock

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
  - time of last two server reboots
Quiz 2

Our system employs the compromise of the previous slide. Consider the scenario previously discussed:
1) Client 1 obtains a lock on a portion of a file
2) There’s a network partition such that client 1 and server can no longer communicate
3) The server crashes and restarts
4) Client 2 obtains a lock on the same portion of the same file, modifies the file, and then releases the lock
5) The server crashes and restarts and the network partition is repaired
6) Client 1 recontacts the server and reclaims its lock

a) In step 4, client 2 is refused the lock
b) In step 6, client 1 is refused the reclaim request
c) The complete scenario can still occur
Quiz 3

Again assume our NSFv4 system uses the compromise approach. A client obtains a lock on a file. The server crashes (the client does not). The server comes back up.

a) the client will be able to reclaim its lock, just as it could in NFSv3

b) the client will be able to reclaim its lock only if the server wasn't down too long

c) the client won’t be able to reclaim its lock
Additional Issues

- **Authentication**
  - poorly supported in NFSv3
  - extensible (and well supported) in NFSv4
- **Authorization**
  - Windows clients require ACLs
  - NFSv4 supports Windows-like ACLs
- **Parallel I/O**
  - pNFS
Genesis of GPFS
Genesis of GPFS

In GPFS, data servers are directly connected to large disk arrays and store both data and meta-data (directories, inodes, etc.). The managers serve tokens to clients, in a manner similar to DFS, allowing synchronized access to data and meta-data. Individual files might be spread across disks of a number of data servers; clients can have simultaneous data transfers from a number of data servers. The managers and clients collectively implement the file system.
At the Brown CS department, GPFS clients, Managers, and data servers are interconnected with 10 Gbps links. Co-residing on some GPFS client machines are NFS servers, on others are CIFS servers. All other department computers are either NFS or CIFS clients, and are generally connected via 1 Gbps links.
GPFS Tokens

- Data tokens
  - if no tokens outstanding, requestor gets token for entire file
    - no further token traffic if no contention
  - if another node has token for file
    - revoked if node has closed file
    - otherwise file range is split between the two
Non-monotonic updates, such as truncating a file, require exclusive access to metadata.
Fault Tolerance

- Lots of redundancy
  - data: RAID
  - data paths: dual-ported RAID arrays
  - multiple managers
  - multiple client nodes (NFS/CIFS servers)
    - client nodes may be managers and vice versa
      - managers are elected
- Network partition
  - quorum is required