Distributed Systems

- *48 - Communication Basics*
- *49 - NFS*
- 50 - AFS
- GFS
- Review

Andrew File System *AFS v1*

- primary motivation was *scale*
  - how many clients could a single server accommodate?
  - user-visible behavior well-defined
  - *whole-file caching* (not block)
    - all reads and writes are to copy on local disk

```
TestAuth      Test whether a file has changed
              (used to validate cached entries)
GetFileStat   Get the stat info for a file
Fetch         Fetch the contents of file
Store         Store this file on the server
SetFileStat   Set the stat info for a file
ListDir       List the contents of a directory
```

Figure 50.1: **AFSv1 Protocol Highlights**

Andrew File System *AFS v2*

- Problems w/ v1:
  - full path traversal costs (on the server!)
    - "/home/keleher/412/exams/s25/exam3s25.tex"
  - client issues too many TestAuth msgs (*sound familiar?*)
  - load not balanced across servers (fix by reapportioning files across volumes on different servers)
  - server has a process per client (fix using threads)
- Improving the protocol:
  - *client callbacks*:
    - promise from the server to notify client if cached file changed
  - file identifier (FID) (like NFS's file handle)
    - volume id
    - file id
    - "uniquifier" (usually called epochs elsewhere)

AFS

example

Client (C_1)	Server
fd = open("/home/remzi/notes.txt", ...);	
Send Fetch (home FID, "remzi")	
	Receive Fetch request look for remzi in home dir establish callback(C_1) on remzi return remzi's content and FID
Receive Fetch reply write remzi to local disk cache record callback status of remzi Send Fetch (remzi FID, "notes.txt")	
	Receive Fetch request look for notes.txt in remzi dir establish callback(C_1) on notes.txt return notes.txt's content and FID
Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local `open()` of cached notes.txt return file descriptor to application	
read(fd, buffer, MAX); perform local `read()` on cached copy	
close(fd); do local `close()` on cached copy if file has changed, flush to server	

fd = open("/home/remzi/notes.txt", ...);
Foreach dir (home, remzi)
if (callback(dir) == VALID)
use local copy for lookup(dir)
else
Fetch (as above)
if (callback(notes.txt) == VALID)
open local cached copy
return file descriptor to it
else
Fetch (as above) then open and return fd	

Andrew File System *cache consistency*

Mentioned two issues w/ NFS:

- update visibility
  - when will server be updated w/ client write?
- cache staleness:
  - when will clients be informed their versions are out of date?

- AFS procedure:
  - client writes, possibly many times
  - *closes*
    - writes complete file back to server, becomes visible
    - server *breaks callback*
      - contact each server w/ a callback and invalidate its copy

all apps on single machine see same copy

Andrew File System *cache consistency*

Client$_1$			Client$_2$		Server	*Comments*
P_1	P_2	Cache	P_3	Cache	Disk	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → A	A	A	do not see writes...
		B	close()	A	A	
close()		B		~~A~~	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		~~C~~	D	
		D	open(F)	D	D	Unfortunately for P_3
		D	read() → D	D	D	the last writer wins
		D	close()	D	D	

Andrew File System *cache consistency*

- AFS provides also *close-to-open* consistency
  - whole-file caching and updating
    - never see concurrent writes diff clients in same version of a file
  - "last writer wins" (really last *closer* wins)
- Crash recovery complicated
  - crashing client might miss callback (*client treats cache as suspect after crash*)
  - crashing server loses callbacks table
    - server might inform all clients after recovery
    - or clients constantly check for server liveness w/ heartbeats
  - *there is a cost to building a more sensible and scalable caching model*

NFS vs AFS

- primarily differ in caching
  - What to cache?
    - NFS caches blocks
    - AFS entire files (on disk)
  - When to push writes to server?
    - Loosely defined for NFS:
      - any time from *right away*, to *when file is closed*
      - (only modified blocks)
    - If any part modified, AFS pushes entire file at `close()`
  - Final contents after concurrent merges by different clients:
    - NFS: writes by the different clients might be intermingled
    - AFS: final version reflects *the last write*; other write is lost

AFS vs NFS

(12 pts) AFS and NFS

Assume:

- We have five unique blocks of data: "A", "B", "C", "D", and "Z", each 4k in length.
- "`write(F, C, 8k)`" means "write the entire block C starting at offset 8k of file F."
- File "F" is initialized to "Z,Z,Z,Z", i.e. it's a 16k file w/ four copies of Z.

C1	C2
open(F)	
write(F, A, 0)	
	open(F)
write(F, B, 4k)	
	write(F, D, 4k)
write(F, C, 8k)	
close(F)	
	close(F)

Which of NFS and AFS could result in the following final contents of "F"?

- A, B, C, Z: NFS only
- A, D, C, Z: NFS only
- A, D, Z, Z: neither
- Z, D, Z, Z: AFS only

Google File System *v1*

- Needs
  - need to handle *massive* files
  - most mutations are appends
  - co-design w/ applications (*also an advantage*)
- Assumptions
  - built from hundreds, or thousands, of cheap machines
  - failures are the common case
- Features
  - relaxed consistency (*also an advantage*)
  - atomic record append (without locking)
  - no data caches
    - append-only model means re-use not common
    - host operating system does limited caching anyway

GFS *two types of nodes*

- multiple *chunk servers*
  - hold fixed size *chunks*
  - immutable once written
  - identified by a globally unique 64-bit ID
- coordinator (GFS *master*)
  - single machine holds all *metadata* in memory
    - persistent
      - file and chunk namespaces (think directories)
      - mappings from files to chunks
      - persistent by flushing *operations log* locally, remotely before visible
    - soft state
      - locations of chunk replicas
      - on startup or recovery restore by asking chunkservers
    - total state is 64 bytes for each 64MB chunk
    - background garbage collection, replica reassignment and balancing