# AFS and distributed file systems

CS380L: Mike Dahlin

November 5, 2003

Plan to throw one away. You will anyway.

*Fred Brooks. Mythical Man Month.*

## 1 Preliminaries

### 1.1 Review

### 1.2 Outline

- Scalable performance
  - Design space
  - A bit of queuing theory
  - Myth: centralization v. scalability
- Consistency
- Admin/security

### 1.3 Preview

## 2 Scalable performance

### 2.1 Design space/trade-offs

4 questions:

1. What AFS optimizations improve performance?
2. How important is the optimization?
3. What is the cost of the optimization?
4. Which optimizations are good trade-offs?

- Clients cache data at local disk
  1. How important is the optimization?
     - + Bigger cache $\rightarrow$ better hit rate $\rightarrow$ more requests local, lower server load
     - - Modern fast networks: Local disk latency $>$ server memory lateny
     - + Modern disks: high bandwidth
  2. What is the cost of the optimization?

  - Harder to deploy to diskless clients?
  - Need to recover consistency state after reboot (is this hard?)

3. Good trade-off?

- Whole file caching

  1. How important is the optimization?
     - Simplifies design
     - Arguably: good to cache in clusters (how useful is it to cache half a file?)
     - Reduces number of requests to server
     - "Prefetch" entire file when first block read (more BW, optimize latency)
  2. What is the cost of the optimization?
     - First read of large, uncached file is slow!
     - Cannot access files larger than local disk (significant for desktop workstation? Signficant for heterogeneous devices/pervasive computing?)
     - Cannot run a distributed database (more generally: relies heavily on strong assumption about workload – "Ousterhout found that most Unix files opened and read in their entirety")
  3. Good trade-off?
     - Later systems went to 64KB as unit of caching

- Open/close consistency (v. BSD semantics)

  1. How important is the optimization?
     - Performance: reduces server callback state and number of messages (1 callback per file instead of per block)
  2. What is the cost of the optimization?
     - See above discussion of whole file caching; many points apply here
     - non-transparancy of location – 2 processes on same machine see updates immediately; 2 processes on different machines see updates at open/close boundaries
     - What should two users on same machine see?
  3. Good trade-off?
     - Can simulate these semantics with explicit locking, but what if you want some other semantics?

- Callbacks (v. contact server on open)

  1. How important is the optimization?
     - Very important, paper argues
     - Note strange intuition: give server more responsibility (deliver callbacks) *improves* scalability! (viz www)
       * Polling systems almost always settle for weak consistency (e.g., 30-seconds for NFS; get-if-modified-since in HTTP) because poll-each-time is expensive (server load, latency)
       * Better performance, lower load, better semantics – what's not to love?
  2. What is the cost of the optimization?
     - More complex – need to keep server and client state synchronized (more on this below)
  3. Good trade-off?

- Medium-grained volumes (e.g. per-user)

1. How important is the optimization?
   - Split users across machines more easily
2. What is the cost of the optimization?
3. Good trade-off?

- Server optimizations:
  - New iopen() interface (to avoid repeated name resolution)
  - New RPC implementation
  - Process structure

1. How important is the optimization?
   - Argued – very important at server
   - Included optimizations at client too
2. What is the cost of the optimization?
   - Much harder to deploy system (more on this below)
3. Good trade-off?

### 2.2 A bit of queuing theory

Paper does a good job of presenting results: x-axis = load, y-axis = response time. These examples reveal some general principles of which you should be aware.

Question: when should you buy a faster computer?

One approach - buy when it will pay for itself in improved response time

Queuing theory allows you to predict how response time will change as a function of hypothetical chnges in # users, speed of CPU, speed of disk, etc

Might think you shouldn't buy a faster X when X has spare capacity (utilization of X ¡ 100%), but for most systems, response time goes to infinity as utilization goes to 100%

How does response time vary with # users?

GRAPH

Worst case: all users submit jobs at same time. Thus response time gets linearly worse as add extra users, linearly better as computer gets faster

Best case: each user submits job after previous one completes. As increase #users, no impact on response time (until system completely utilized)

What if we assume users submit jobs randomly and they take random amounts of time. Possible to show mathematically: $responsetime = \frac{servicetime}{1-utilization}$

*fine print - exponential distribution*

Lessons

- Experimental design
  - Don't compare systems' response time under same load and declare one is bettter than the other
  - Make sure you are testing system in the load regime you intend

- System design
  - Be careful about indulging in clever scheduling
    * It may only matter when load is heavy
    * May be better to increase system capacity by 10% and eliminate queues

## 2.3 "Distributed" = "scalable" myth (aka "Centralized" = "not scalable" myth)

- You will often hear people state that their motivation for designing a distributed system was to improve scalability.
  - Sometimes this is shorthand for a sound argument. Often, however, it is confused thinking
  - "This system needs to handle X requests for second. We must distribute it or we will have a centralized bottleneck.
  - Example: "peer to peer file systems are more scalable because they don't have a centralized bottleneck"
  - Truth: Given X CPUs and Y disks, I can build a file system with much higher max throughput if you let me put the CPUs in the same room than if you make me distribute them across locations.
  - NFS v. AFS: AFS architecturally more scalable; NFS still tenable b/c building big fast servers is simpler than building AFS (?)
  - Example in AFS paper – easier to add new disks to existing server than to move a user's partition to a different server.
- Aspects of scalability (or: don't forget brute force)
  - Tight coordination helps maximize throughput (requests/second) of a server
    - If all you care about is maximizing the number of requests per second your server can handle, centralize it.
    - If requests depend on other requests, easier to build fast server if communication is fast
    - x-processor SMP > x-processor cluster in a room > x-processor set of machines spread across the country
    - Counter-argument 1: client-server network costs
      - If request+response > internal coordination bandwidth to process request, may be good to move part of server near clients
    - Counter-argument 2: Servers are expensive
      - Hardware companies charge more for multi-processor machines
      - → cluster of machines in a room can be cheapest way to get scalability for services that don't need too much coordination
  - Availability and latency
    - Possible to improve latency or availability by handling requests locally (requires careful protocol/system design)
    - In practice challenging to design highly-available distributed protocols (Lamport's definition of distributed system)
    - Brute force approach: carefully engineered system in a well-protected room works pretty well in many cases.
  - Shift costs to administrative units/users (and/or use idle resources)
    - Even if system uses more total resources, can be cheaper than a more efficient system that uses dedicated server resources
    - AFS example: use client disks and processing to avoid bothering servers
    - Is total system processing reduced? Maybe not.
    - P2P example: use spare capacity of clients rather than dedicate server resources
  - Danger of distribution: security
    - Often limits how aggressively a system can be distributed beyond a machine room
  - Disaster recovery – geographic distribution can be essential

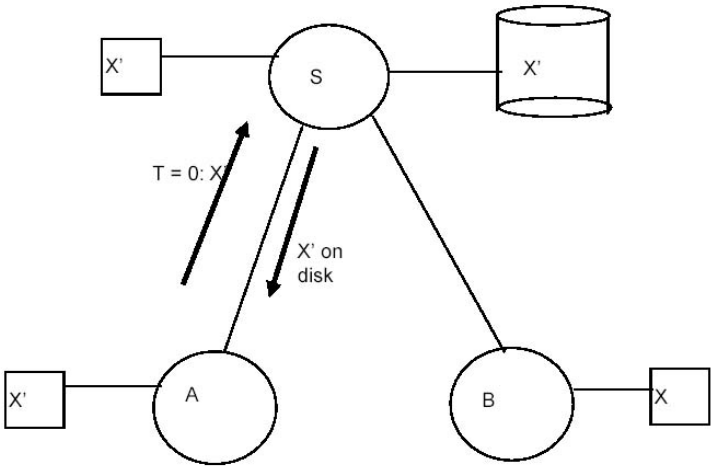
# 3 Admin

# 4 Consistency

- Consider updates in NFS or AFS (recall: cache files at clients)
- Suppose A and B both read object X, then A updates X to X'. What will B read?

## 4.1 Sequential ordering constraints

- Cache coherence - what should happen? What if one Cpu changes file and before it's done, another CPU reads file?
- strict coherence - any read on a data item x returns a value corresponding to the most recent write on x [Tannenbaum[1]]




- Seems simple,
  - But in a distributes system
    - Caching
    - Write buffer
    - Multi-threading
    - Out of order messages, peer-to-peer
    - Delayed messages

[1]This is actually Tannenbaum's defintion of "strict consistency"; I disagree with the way he organizes things. He says that "strict consistency" is "impossible to implement in a distributed system" and that sequential consistency "is a slightly weaker model than strict consistency." Both statements are, in my view, misleading.

Strict coherence can be implemented in a distributed system by having a lock manager hand out read and write tokens to client caches and having it ensure that when a cache holds a write token on an object, no other caches hold a read or write token on that object (e.g., via invalidations.)

Sequential consistency is a weaker condition than strict coherence in some sense: strict coherence relies on a global notion of time while sequential consistency allows the final "sequential order" to differ from the "real-time order" of requests. For example Amol Nayate recently built a system that enforces sequential consistency and enforces a separate maximum staleness.

But in another sense, sequential consistency is *stronger* than strict coherence. In particular, sequential consistency also requires that all operations complete in program order even if they are to different memory locations. Thus sequential consistency constrains operations on different objects while strict coherence only constrains operations on the same object.

Tannenbaum's definition of "strict consistency" appears to differ by assuming that all operations are instantaneous and non-blocking; I think his assertions may be true under these assumptions, but I find these assumptions strange.

* Disconnected operation
* ...

– but in distributed system reads and writes take time: actual read could occur anytime between when system call is started, and when system call returns

* if read finishes before write starts, then get old copy
* if read starts after write finishes, then get new copy
* if reads and writes overlap, get ??? (either old or new)
* if writes overlap, get ??? (either old or new)

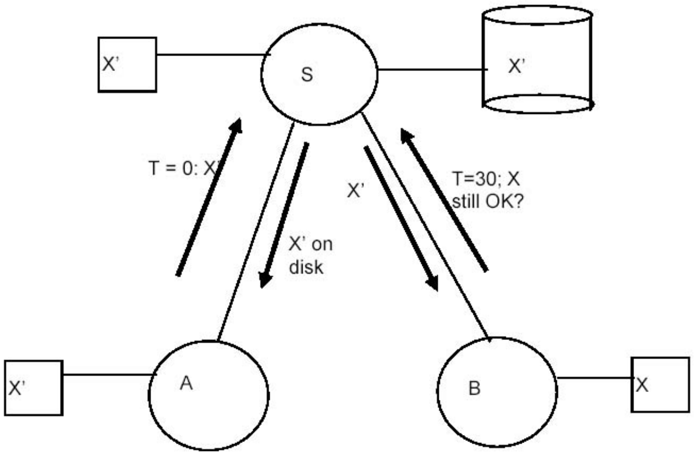
```
A
 <--- read A ---->
                   <----- read A or B ----->
                                <----------  write B  --------->
<----------  read A or B or C ---->
<-----------  write C ------>
    <- --------  read B or C --->
                                                                        < read B or

<-------------------------------------  TIME  ------------------------------------
```

- in above diagram, non-deterministic as to which write - C or B - ends up winning
- once I read B, all later reads must return B or C

## 4.2 NFS: weak cache consistency

What if multiple clients are sharing same files? Easy if they are both reading - each gets a copy of the file
What if one writing? How do updates happen?
At writer - NFS has hybrid delayed write/write through policy

- write through within 30 seconds or immediately when file closed

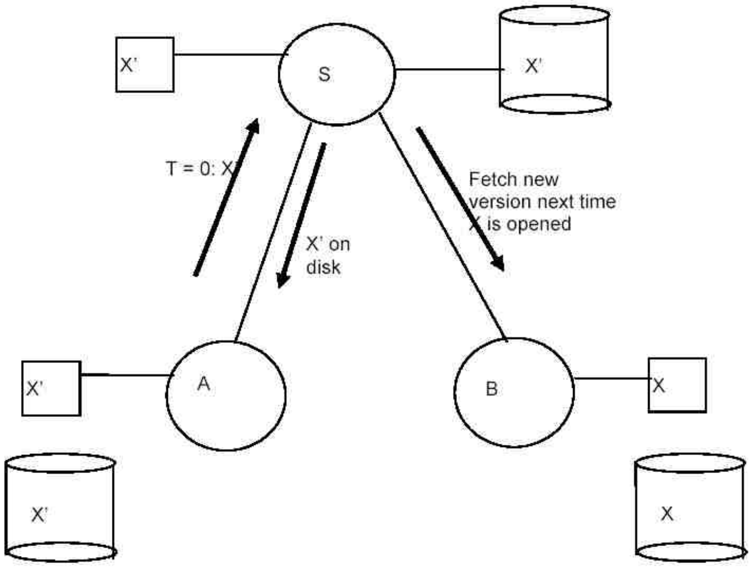



- How does other client find out about change (it has cached copy, so doesn't see any reason to talk to the server)
- In NFS, client polls server periodically, to check if file has changed.

- Poll server if data hasn't been checked in last 3-30 seconds (exact timeout is tunable parameter)
- Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout. They then check server, and get new version.

### 4.3 AFS

More precise consistency model




1. callbacks
   - server records who has copy of file
   - send "callback" on each update
2. write-through on close
   - If file changes, server is updated (on close)
   - Server then immediately tells those with old copy
3. session semantics - updates visible only on close
   - In UNIX (single machine) updates visible immediately to other programs who have file open
   - In AFS, if on same machine see updates as they happen
   - In AFS on different machine, everyone who has file open sees old version; anyone who opens file again will see new version

In AFS:

- on open and cache miss - get file from server; set up callback
- on write close: send copy to server; tells all clients with copies to fetch new version on next open
- Challenge: improved caching + consistency increases failure handling complexity:
- What if server crashes? Lose all callback state
- Reconstruct callback information from clients - go ask everyone "who has which files cached?"
  – Client state means we *must* make client and server consistent
  – Client state means we *can* make client and server consistent

## 4.4 Leases

- Suppose A is caching X, but connection between S and A fails and B writes X
- What happens?
- Option 1: delay write completion until acks come back → arbitrary delay, poor availability
- Option 2: lease on A's callback
  - When A reads X, it gets a lease – a callback of limited duration
    * E.g., "if X updated in next 10 seconds, S will notify A"
    * Lease automatically expires in 10 seconds
    * On disconnection, B's write can complete when lease expires
    * what should happen on A's read during disconnection
      · Option a: block (reduce availability for disconnected nodes rather than whole system)
      · Option b: return possibly stale value – relax consistency
    * What about A's writes? Next time: Coda
  - → Leases combine callbacks with polling
  - Variation: volume lease
    * Problem with short leases: high renewal overhead
    * Problem with long leases: poor availability
    * Solution: volume leases
      · To read must hold lease on both object and volume
      · Short volume leases (e.g., 5 seconds); renewal overhead amortized across many objects in volume
      · Long object leases
      · Note: invalidations still sent only for individual objects (no need for "volume invalidate")

## 4.5 consistency v. coherence v. staleness

- Coherence restricts *order* of reads and writes to *one location*
  - Can you tell memory system is playing tricks on you by looking at one location?
  - Example

```
P1:                                   P2:
for(ii = 0; ii < 100; ii++){          while(1){
   write(A, ii);                         printf(``%d ``,
}                                          read(A));
                                         }
```

  - Where is incoherence?
    * 1 2 3 3 3 4 9 10 9 11 12 13 ...
  - Why might a system exhibit incoherence?
- *Staleness* bounds the maximum (real-time) *delay* between writes and reads to one location.
  - Can you tell memory system is playing tricks on you by looking at clock?

– Example (think stock prices)

```
P1:                                                  P2:
while(1){                                             while(1){
   sleep(1000ms);                                       sleep(1000ms);
   write(A, ``At %t price is %d\n'');                   printf(``%s``,
}                                                          read(A));
                                                     }
```

– Where is staleness (assuming real time OS)

  * At 1:00:00 price is 10.50
  * At 1:00:01 price is 10.55
  * At 1:00:02 price is 10.65
  * At 1:00:02 price is 10.65
  * At 1:00:02 price is 10.65
  * At 1:00:05 price is 13.18
  * ...

– Why might a system exhibit incoherence?

- *Consistency* restricts order of reads and writes *across locations*

  – Can you tell memory system is playing tricks on you by looking at multiple locations?

  – Example 1

```
P1:                                       P2:
  for(ii = 0; ii < 100; ii++){            while(1){
    write(A, ii);                           printf(``(%d, %d), ``,
    write(B, ii);                              read(A), read(B));
  }                                       }
```

  – Where is inconsistency?:

    * (0,0), (0,1), (1,2), (4,3), (4,8), (8, 9), (9,9), (9,10), (9,10), (10,10), (11,10), (11,11), (12,12), ...

  – Is there also incoherence?

  – Example 2 (a classic)

```
P1:                            P2:
   write(A, 0);                   write(B, 0);
   ...                            ...
   write(A, 1);                   write(B, 1);
   if(read(B) == 0){              if(read(A) == 0){
      printf(``P1.'');               printf(``P2.'');
   }                              }
```

  – Which outputs are legal under strict coherence? Under sequential consistency?

    * "P1."
    * "P2."
    * ""

    * "P1.P2."
    * "P2.P1."
  - Why might a system exhibit inconsistency?
  - Notice
    * In first example, order between writes must be maintained...fairly obvious notion of causality
    * In second example, order between writes and reads must be maintained. (Less obvious?)
    * $\rightarrow$ Consistency involves ordering both writes and reads.

## 4.6 Consistency and coherence semantics

- Definitions (from Tannenbaum *Distributed Systems*)
  - strict coherence - any read on a data item x returns a value corresponding to the most recent write on x[2]
  - delta coherence - the maximum real-time delay between when a write completes and when a subsequent read begins such that the read must return a value at least as new as that write. (Note: this one is not from Tannenbaum)
    * Strict coherence = delta coherence with delta = 0
  - sequential consistency – The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in the sequence in the order specified by its program
  - linearizable – sequential consistency + strict coherence. E.g., If $Timestamp_{op1}(x) < Timestamp_{op2}(y)$ then operation op1(x) should precede op2(y) in this sequence.
  - Linearizable is a stronger condition than sequential consistency
- Limitations of strong consistency
  - Sequential consistency has fundamental performance cost: fast reads or fast writes but not both
    * $r + w \geq t$ (where $r$ is read time, $w$ is the write time, and $t$ is the minimal packet transfer time between nodes.) [Lipton and Sandberg]
  - Sequential consistency has a fundamental CAP dillemma (Brewer): A system can not have sequential **C**onsistency and maintain 100% **A**vailability in the presence of **P**artitions.
  - $\rightarrow$ develop weaker models
  - *causal consistency* – writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.
    * Esentially – if $P_1$ reads a write $A$ before issuing a write $B$, then any process that sees $B$ cannot subsequently see the old value of $A$
    * Hard to see why this is useful if you assume a centralized consistency server. Think about a world where machines can send writes to one another. If $A$ reads a bunch of writes from $B$ and then creates some writes of its own. Then $C$ synchronizes with $A$, $A$ must send $B$'s writes to $C$ before sending its own.
  - *FIFO consistency* (aka *PRAM consistency*) – writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in different orders by different processes
  - Is FIFO stronger or weaker than causal?

[2]See discussion above

### 4.7 Key ideas

- key idea 1: pick time when operation "completes"
  - use blocking to enforce coherent/consistent order
  - generalization – *weak consistency*, *release consistency*, *lazy release consistency* – locks define range of time when reads and writes can complete
- key idea 2: for strict coherence make sure that only one copy of data during a write
  - Need a central location ("lock manager")
  - Or a broadcast medium (e.g., snooping)
- key idea 3: for sequential consistency no out-of-order reads or writes
  - cannot service cache reads until a read miss or write completes
  - in single-threaded process with blocking reads and writes, strict coherence = sequential consistency $\rightarrow$ OS folk tend to use terms interchangably (bad idea).
- Claim: AFS provides sequential consistency (and strict coherence) under two caveats
  1. Ignoring the case of multiple processes sharing a file on same machine
  2. All reads and writes during an open/close session "begin" when the open begins and "end" when the close ends
  3. Why: server is a central point that serializes all concurrent reads and writes;

## 5 Administration and scalability

### 5.1 Volumes + Mount points

Volume

- Administrative unit
- Location independent – can move from one machine to another
- Copy-on-write semantics possible (cheap checkpoint; simplifies backup – very nice)
- Global slowly-replicated mapping of volume to server
  - v. NFS mount table – needs to be manually replicated per machine (a pain)

### 5.2 Security

- Scale $\rightarrow$ not all machines trust one another
- Most contemporary file systems (NFS, Sprite) designed to share files across a "workgroup"
  - Trust (mostly) clients – assume that all incoming requests come from a trusted kernel
  - User logs in $\rightarrow$ kernel checks password and associates UID with processes
  - User accesses file via system call $\rightarrow$ kernel attaches UID to requests to server
  - Server trusts UID in request (after all, password was checked)
  - NFS: small help – don't allow remote requests from "root" UID

        * I can still act as any user other than root

- Problems (Bad anyhow; especially bad for scalable system)
  - Assumes machine is trusted – doesn't work for laptops, win95/98, dorm room machines
  - Assume network is secure – doesn't work for wireless, dorm room machines
- AFS improvements

### 5.3 Portability

- Minimize kernel mods – most of AFS is in user-level daemons
  - Simplify building system – put most AFS code into user space
    * Structure: upcall threads + user-level handler + use existing file system for persistent storage
    * Additional advantage: improve scalability by avoiding limited kernel resources
  - Changes:
    * iopen, etc. interface – avoid path parsing on requests
    * Rationale: path parsing a big overhead
  - Result: Portability a big problem (IMHO)
    * Need to patch kernel $\rightarrow$ 6 month delay to new revision of kernel
  - lessons
    * Measure before optimizing (yes)
    * Still, no such thing as small kernel change
    * Perhaps open source/linux world fixes this
    * Perhaps non-invasive ways to get similar benefits (e.g., keep old kernel interface but add cache of path-¿inumbers... run more slowly on unmodified kernel, but still run...)

## 6 Conclusions

- Don't forget brute force as a way to improve scalability.
- There is a good "second system effect" – many of the best system building projects have happened when researhers build 2 versions of the system, learning lessons from the first.
- Goal: support 10K clients – did they make it?
  - Increased number of clients per server by 2-4x at significant cost to compatibility – not decisive
  - Improved security – this seems more important (in "niche" market – college dorm rooms; not as decisive within closed network of a company)