

# Lecture #24: Networks and Distributed systems

\*\*\*\*\*

## Review -- 1 min

\*\*\*\*\*

### In-kernel data structures

- n open file table
  - o book has confusing terminology. They say “2 open file tables, a per-process and a global”
    - per-process: current position + pointer to global
    - global: reference count + file ID + cached header stuff...
- n caching
- o mmap

### RAID

### FS Scheduling

### Rethink the sync (guest lecture)

- Performance v. durability
- Example
  - o T1 begin
  - o W1
  - o W2 barrier

---

  - o T1 end Barrier + block
  - o T2 begin
  - o W3
  - o W4 barrier

---

  - o T2 end Barrier + block
  - o T3 begin
  - o W5
  - o W6 barrier

---

  - o T3 end Barrier + block
  - o Print/send message “done” Block

Barriers (write scheduler), block (sync)

→ Better performance

→ Better reliability (current disks “cheat” because otherwise performance is too horrible)

\*\*\*\*\*

## Outline - 1 min

\*\*\*\*\*

Motivation

Basic NW communication

3 problems

- performance
- reliability
- security

Case study: Distributed file systems

\*\*\*\*\*

## Preview - 1 min

\*\*\*\*\*

Today: motivation, basics, file system example, performance

Monday/Wednesday: Reliability:

Network failures:

- Retransmission, idempotent requests

Machine failures

- Careful protocol construction (e.g., ad hoc solutions)
- 2 phase commit
- Reliable asynchronous messaging

if time: security

\*\*\*\*\*

## Lecture - 20 min

\*\*\*\*\*

## Motivation

Technology trends:

Decade	Tech	\$/ machine	Sales Volume	Users/ machine
50's	custom	\$10M	100	1000's
60's	mainframe	\$1M	10K	100's
70's	mini-computers	\$100K	1M	10's
80's	PCs	\$10K	100M	1
90's	PCs, portables, PDAs	\$1K	1B	1/10
00's	appliances	\$0.1K	10B	1/100

### **Centralized v. Distributed systems**

**Distributed system:** physically separate computers working together

Why do we need distributed systems?

- Cheaper to build lots of simple computers
  - Mfg rule of thumb: 2x increase in quantity → 10% reduction in cost per unit
- Easier to add power incrementally (v. design whole new machine)

### **Promise of distributed systems**

- Higher availability – one machine goes down, use another
- Better reliability – store data in multiple locations
- More security – easier to make each (small) piece secure

If we're not careful, reality will be disappointing

- Worse availability – depend on every machine being up
- Lamport: “A distributed system is a system where I can't get any work done if a machine I've never heard of crashes.”
- Worse reliability – can lose data if any machine crashes
  - Worse security – anyone in the world can break into my systems

Key idea: coordination is more difficult b/c can only use network for coordination and because of *partial failures* – part of the system (a connection, a machine) fails while the rest keeps running

## Send/Receive

How do you program a distributed application?

Need to synchronize multiple threads, but they are on multiple machines (no test&set)

Atomic send/receive – doesn't require shared memory for synchronizing cooperating threads

Note that send and receive are atomic  
never get portion of a message (all or nothing)  
two receivers can't get same message

Mailbox – temporary holding area for messages (ports)

Looks like producer/consumer queue

Receive(buffer, mbox)

→ Wait until mbox has message in it, then copy message into buffer, and return

when packet arrives, OS puts message into mbox, wakes up one of the writers

Send(buffer, mbox)

When can Send return?

- when receive gets message?
- when message is safely buffered on destination node?
- Right away, if message is buffered on source node?

## Message styles

1-way – messages flow in one direction (UNIX pipes, TCP)

2-way – request-response (remote procedure call)

1-way communication

Producer:

```
int msg1[1000];
while(1){
    prepare message; // add coke to mach.
    Send(msg1, mbox);
}
```

Consumer

```
int msg2[1000];

while(1){
    receive(msg2, mbox);
    process message; // drink coke
}
```

no need for producer/consumer to keep track of space in mailbox –  
handled by send/receive

## 2-way communication

What about 2-way communication? Request/response – e.g. “read a file” stored on a remote machine

Also called – client-server

Client = requestor

server = responder

Server provides “service” to client

**request/response:**

```
client:
    char response[1000];
```

```
send("read rutabaga", mbox1);
receive(response, mbox2);
```

server:

```
char command[1000], answer[1000];

receive(command, mbox1);
decode command;
read file into answer;
send(answer, mbox2);
```

## Remote procedure call

Call a procedure on a remote machine

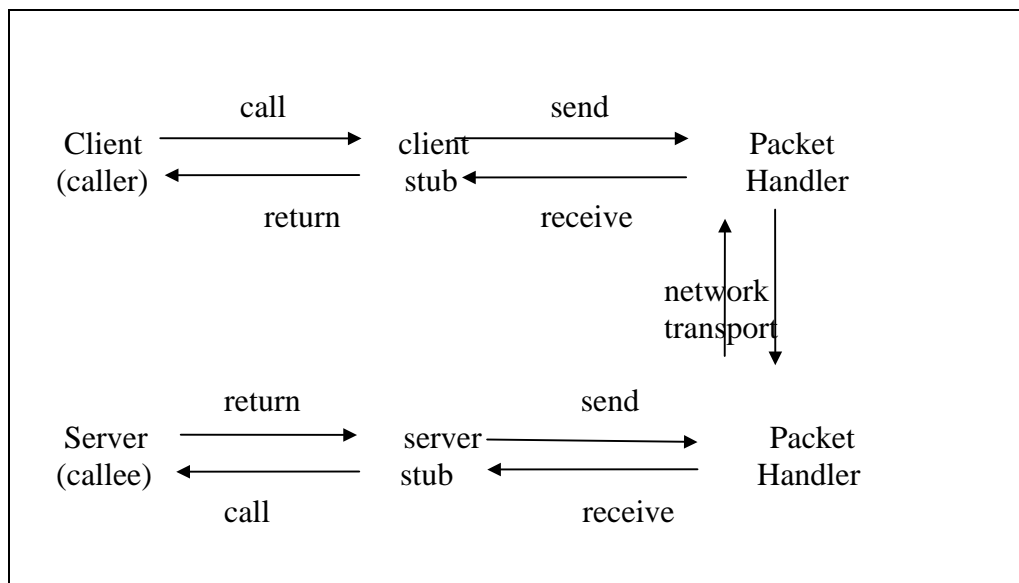
client

```
remoteFileSys->Read("rutabaga");
```

translated into call on server:

```
fileSys->Read("rutabaga");
```

Implementat on top of request-response message passing  
"stub" provides glue



```
client stub:
  build message
  send message
  wait for response
  unpack reply
  return result
```

```
server stub:
Create N threads to wait for work to do
  loop:
    wait for command
    decode and unpack request parameters
    call procedure
    build reply message with results
    send reply
```

### ***Comparison between RPC and procedure call***

What's equivalent

Parameters – request message

Result – reply message

Name of procedure – passed in request message

return address – mbox2

### ***Implementation issues***

Stub generator – implements stubs automatically

for this, only need procedure signature – types of arguments,  
return value

generate code on client to pack message, send it off, on server  
to unpack message, call procedure

How does client know which mbox to send to? Binding

static – fixed at compile time (e.g. C)

dynamic – fixed at runtime (e.g. Lisp, RPC)

In most RPC systems, dynamic binding via name service.

Name service provides dynamic translation of service → mbox

Why runtime binding?

Access control – check who is permitted to access service  
fail-over – if server fails, use another

## Problems with RPC

RPC provides location transparency – except

### Failures

Different failure modes in distributed system than on single machine

Several kinds of failure

(1) communication interruption

- lost message
- lost reply
- cut wire
- ...

Simple solution:

Request/acknowledge protocol

Common case:

- 1) Sender sends message (msg, msgId) and sets timer
- 2) Receiver receives message and sends (ack, msgId)
- 3) Sender receives (ack, msgId) and clears timer

If timer goes off, goto (1)

How does this work? Local procedure call guarantees *exactly once* semantics. What does retransmission guarantee?

- What if msg 1 lost?
- What if ack lost?

Guarantees *at least once* semantics ***assuming no machines crash or otherwise discontinue protocol***

- Receiver guaranteed to recv message at least once

- Receiver may recv message multiple times. Receiver MAY use sequence number to filter repeated transmissions so that each is acted upon just once

## (2) Machine fails

Several variations:

- ◆ user level bug causes address space to crash
- ◆ machine failure, kernel bug causes all AS on same machine to fail
- ◆ power outage causes all machines to fail

Before, whole system would crash. Now: one machine can crash, while others stay up.

Now, one machine can crash, while others stay up. If file server goes down, what do the other machines do?

Example: simple send/ack protocol above -- Difficult to deal with machine crashes

- If sender crashes (or if sender gives up because it has tried 100 times in a row) what is the post condition?
  - Receiver may or may not have received message
- If receiver crashes, filtering repeated messages to act on them exactly once is tricky → carefully design protocol to either (a) tolerate *at least once* semantics or (b) detect/avoid replication even across sender/receiver failures

Tricky – processing a message can have arbitrary side effects. Want *exactly once* semantics or protocol may have strange behaviors

Tomorrow: strategies for dealing with machine failures in distributed protocols

Ad-hoc strategies (file systems)

Two-phase commit

Persistent message queues

## **Performance**

Cost of a procedure call << same machine RPC << network RPC

means programmer must be aware that RPC is cheap, but not free

## Network performance

Overhead – Cpu time to put packet on wire

Latency – how long from when one byte packet sent to when it is received

Throughput – maximum bytes per second

Latency – significant fraction of the speed of light (1 foot/ns) → <1us anywhere in building

	Throughput	Overhead	100 byte	4KB	Remote 4kB read
TCP/IP Wireless	10 Mbit/s	0.5-1ms	.5ms + .08ms	.5ms + 3ms	4ms
TCP/IP Ethernet	100 Mbit/s	0.5-1ms	.5ms + .008ms	.5ms + .3ms	1.3ms
TCP/IP Gigabit Ethernet	1000 Mbit/s	0.5-1ms	.5ms + .0008ms	.5ms + .03ms	1.03ms
AM/ Myrinet	1200Mbit/s	.007ms	.007ms + .001ms	.007ms + .03ms	.04ms

Moral 1: Don't just look at bandwidth

What is latency if go cross-country?

3000 miles \* 5000 ft/mile → 15ms

now 4KB read dominated by latency for all networks

Key to good performance:

- in LAN – minimize overhead
- in WAN – keep pipeline full

Caching can help, but

- generally gets rid of “transparent stub generation” advantage of RPC
- makes failure handling more complex

In distributed system, you have to face up to these two problems;  
illustrate with network file system

\*\*\*\*\*

Admin - 3 min

\*\*\*\*\*

\*\*\*\*\*

Lecture - 23 min

\*\*\*\*\*

## Distributed file system

Outline

Distributed File Systems

2 Case studies: NFS, AFS

Crosscutting issues

Failures

Performance

Cache coherence

A **distributed file system** provides transparent access to files stored on a remote disk

Themes:

**failures**: what happens when a server crashes, but a client doesn't? Or vice versa?

**Performance** → caching; use caching at both clients and server to improve performance

**cache coherence** – how do we make sure each client sees most up-to-date copy?

**Atomic update** – how to update state at two or more machines

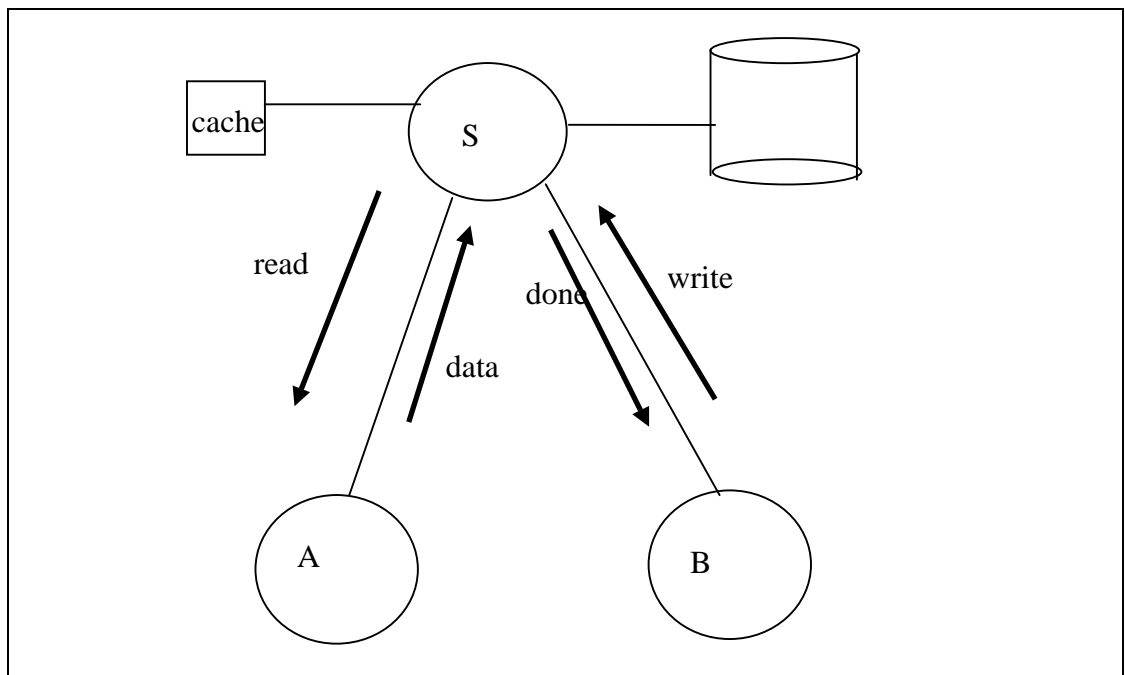
These issues and strategies we will discuss are much more general than file system – arise in many distributed systems.

## Simple: no caching

use RPC to forward every file system request to remote server (e.g. Novell Netware)

Example operations: open, seek, read, write, close

Server implements each operation as it would for a local request and sends back result to client  
*straightforward utilization of RPC*



Advantage: server provides consistent view of file system to both A and B

2 issues: Failures and performance  
Failures – see NFS (below)

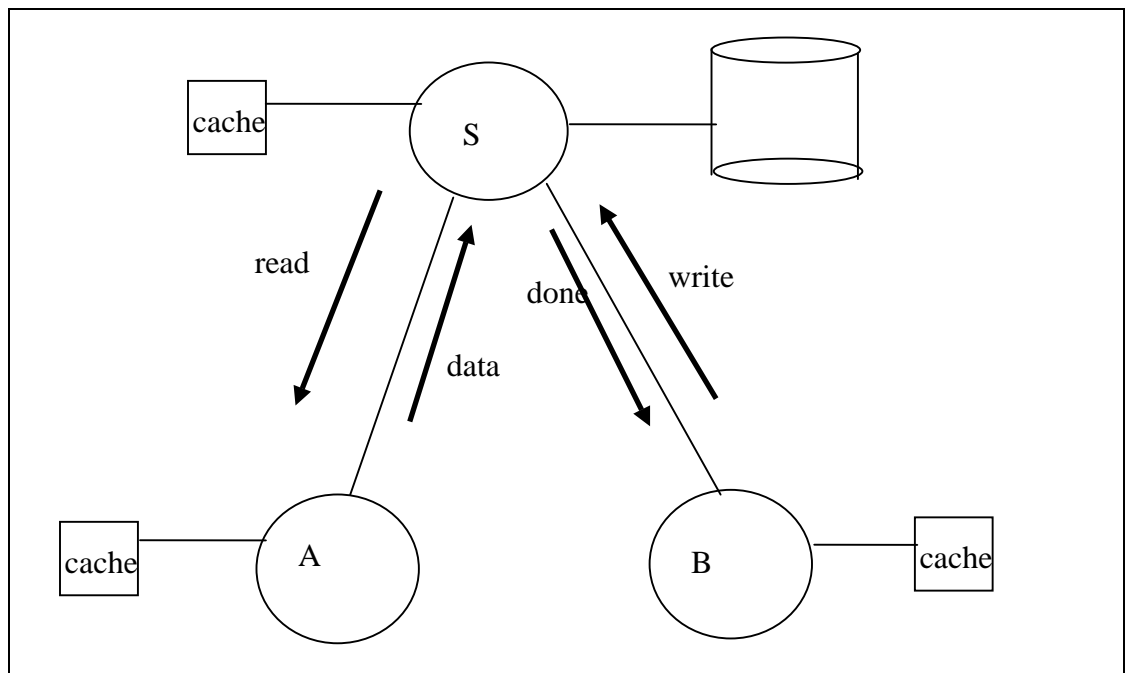
Performance can be lousy:  
going over network is slower than going to local memory!  
lots of network traffic

server can be a bottleneck – what if lots of clients?

## NFS (Sun Network File System)

Idea: use caching to reduce network load

Cache file blocks, file headers, etc at both clients and servers



Advantage: if open/read/write/close can be done locally, no network traffic

Issues:

- (1) no longer have automatic stub generation → lose one advantage of “RPC” over message passing
- (2) failures and cache consistency

### ***Issues: part 1: 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

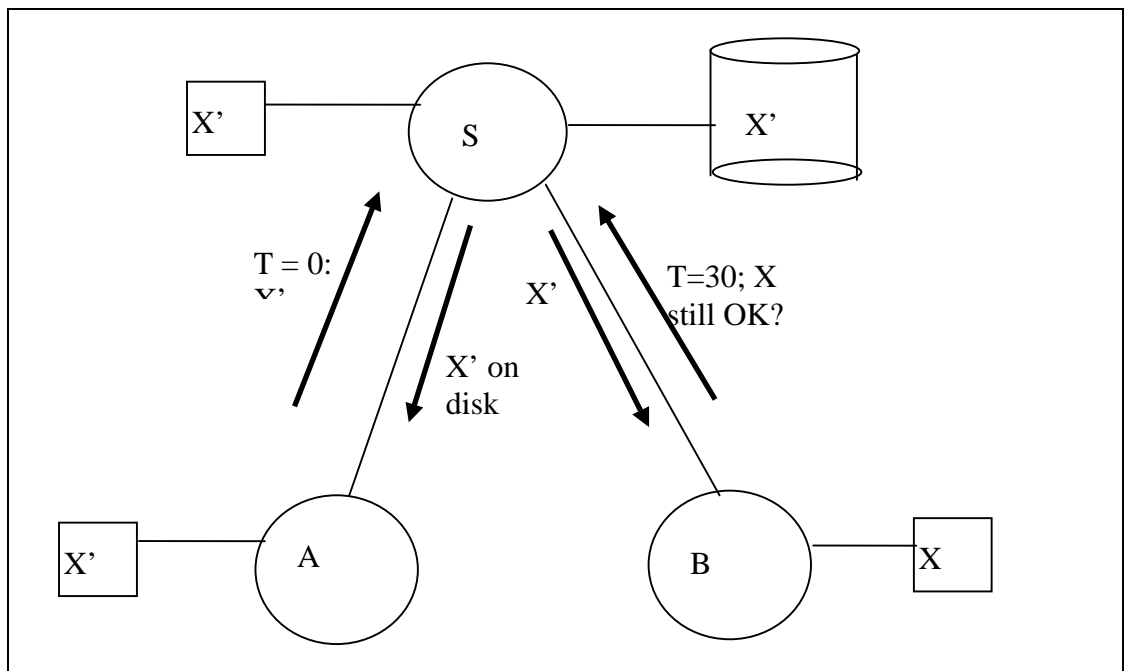
- write through within 30 seconds or immediately when file closed

How does other client find out about change (it has cached copy, so doesn't see any reason to talk to the server)

### ***NFS protocol, part 1: weak consistency***

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.



What if multiple clients write the same file? In NFS, can get either version (or parts of both). Completely arbitrary!

HTTP uses essentially same protocol

### **Issues, part 2: Failures**

What if server crashes? Can client wait until server comes back up, and continue as before?

- 1) any data in server memory but not yet on disk can be lost
- 2) shared state across RPCs. Ex: open, seek, read. What if server crashes after seek? Then when client does “read”, it will fail.
- 3) Message retries: suppose server crashes after it does UNIX “rm foo”, but before acknowledgement?  
Message system will retry – send it again. How does it know not to delete it again? (Could solve this with two-phase commit protocol, but NFS takes a more ad hoc approach – sound familiar?)

What if client crashes?

- 1) Might lose modified data in client cache

NFS: Solve problems in protocol (ad hoc?)

### **NFS Protocol (part 2): stateless**

Client not allowed to rely on any server state

- 1) write through caching – when a file is closed, all modified blocks are sent immediately to server disk. To the client “close” doesn’t return until all bytes are stored on disk.
- 2) Stateless protocol – server keeps no state about client (except as hints to help improve performance; e.g. a cache)
  - each read request gives enough information to do entire operation – ReadAt(inumber, position) not Read(openFile)
  - when server crashes and restarts, can start processing requests immediately, as if nothing happened
- 3) Operations are “idempotent”: all requests are OK to repeat (all requests are done *at least once*). So, if server crashes between disk

I/O and message send, client can resend message, server just does operation all over again

- read and write file block are easy – just re-read or re-write file block; no side effects
- What about “remove”? NFS just ignores this problem – does the remove twice; second time returns an error if file not found

4) Failures are transparent to client system

Is this a good idea? What should happen if server crashes? Suppose you are an application, in middle of reading a file, and server crashes?

Options;

a) hang until server comes back up (next week)?

b) return an error? Problem is: most applications don't know they are talking over the network – we're transparent, right?

Many UNIX apps simply ignore errors! Crash if there is a problem. (Network → many more errors than before)

NFS does both options – can select which one. Usually, hang and only return error if really must – if see “NFS stale file handle” that's why

## ***NFS Summary***

NFS pros & cons

+ simple

+ highly portable

■ sometimes inconsistent

■ doesn't scale up to large # of clients

Might think NFS is really stupid, but Netscape/WWW does something similar: cache recently seen pages, and refetch them if they are too old. Nothing in WWW to help with cache coherence

**Notice:** what happened to “RPC → transparent distributed system”?

■ performance → add caching

■ failures → change all public methods to (mostly) idempotent

- performance v. failures → write through cache
  - performance v. failures → weak consistency
- Basically ended up rearchitecting and rewriting everything!

## Cache consistency

Today: cache consistency – callbacks, leases  
Wednesday: reliability

### ***Sequential ordering constraints***

Cache coherence – what should happen? What if one Cpu changes file and before it's done, another CPU reads file?

“right answer” turns out to be more subtle than one might hope...

- Essentially same problem as reasoning about synchronization of multi-threaded programs – we have several programs running on (potentially) multiple processors (at arbitrary speeds), what can they see as they read and write memory (or files)?
- But now even load/store may not be atomic operations
  - Caching, write buffering, multipath routing through network
  - → write by one thread may not immediately be seen by another!
- Consistency/coherence/staleness semantics define how “non-atomic” memory can be (and give you a basis for reasoning about distributed programs)
  - Essentially ask “can a distributed program tell that memory is ‘playing tricks on it’ compared to case where all threads run on uniprocessor with single memory?”
  - Memory system semantics restrict/define which “new” behaviors a memory system (or file system) can expose to a program

### **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++){
write(A, ii);
}
while(1){
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?

e.g., 2 nodes, writer sends updates via Internet; updates get reordered en route...

e.g., cooperative caching -- read cached value from two different peers, could get out-of-order answer

e.g., client switching between two servers (e.g., on Internet, get redirected to different Akamai node)

**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:
while(1){
sleep(1000ms);
write(A, "At %t price is %d\n");
}
P2:
while(1){
sleep(1000ms);
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 staleness?

e.g., NFS polling interval

e.g., network delay prevents update/invalidation from reaching cache for a while...

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

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

“P1.”

“P2.”

“”

“P1.P2.”

“P2.P1.”

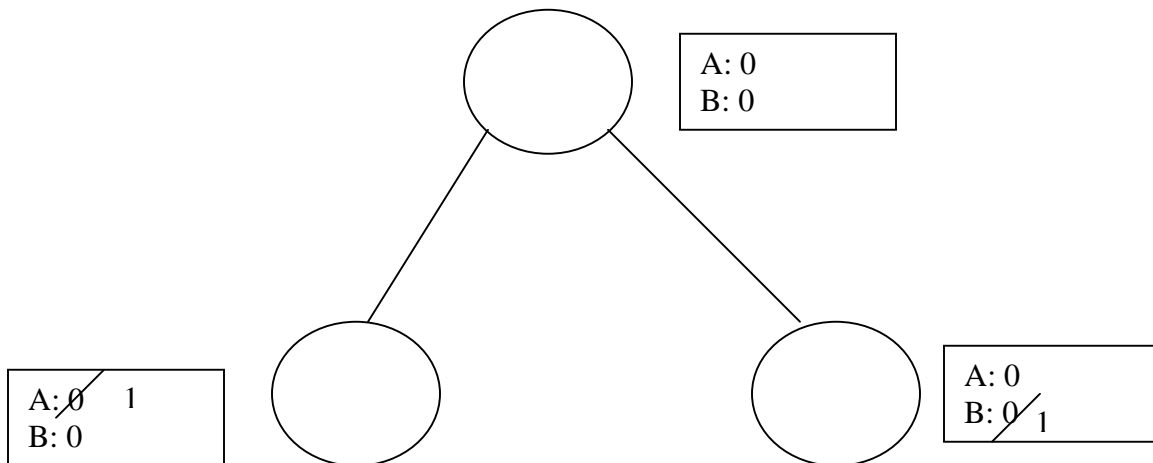
- Why might a system exhibit inconsistency?

- 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?)

Consistency involves ordering both writes and reads.

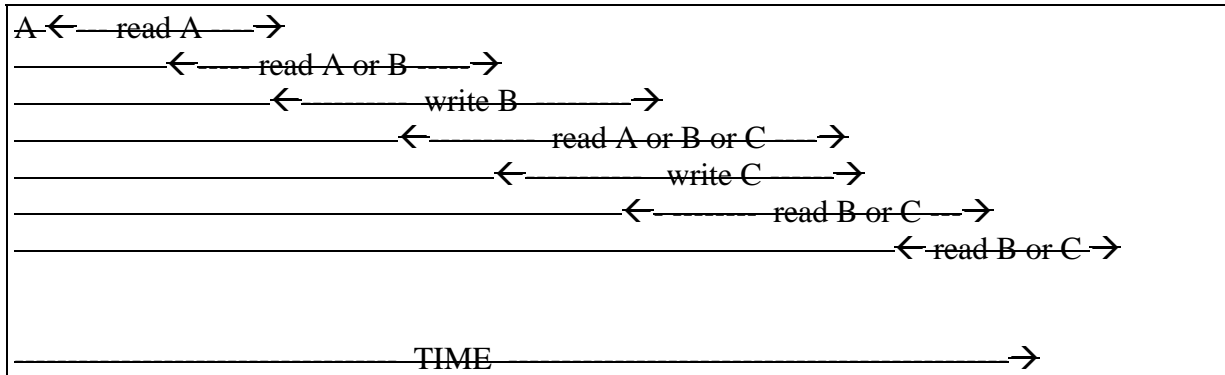


**Definitions** (from Tannenbaum *Distributed Systems (with slight modifications and additions)*)

- **strict coherence** - any read on a data item x returns a value corresponding to the most recent write on x

Even this is a bit less tight than you might hope... Simple to see what strict coherence means if reads and writes are instantaneous. But they are not!

Note that every operation takes time: actual read could occur anytime between when system call is started, and when system call returns



Assume what we want is distributed system to behave exactly the same as if all processes are running on a single UNIX system  
 if read finishes before write starts, then get old copy  
 if read starts after write finishes, then get new copy

Otherwise — indeterminate — can get either new or old copy

Similarly, if write starts before another write finishes, may get either old or new version. (Hence, in above diagram, non-deterministic as to which you end up with!)

In NFS, if read starts more than 30 seconds after write finishes, get new copy. Otherwise, who knows? Could get partial update.

Regular v. atomic semantics

**Regular semantics** — return either the value of the last completed write or that of one of the writes which are concurrent with the read.

**Atomic semantics** — guarantee that the read and write operations to the variable behave exactly as if they happened instantaneously in some point in time which is within the actual time where the operation took place. (Usually this is what is assumed for strict coherence)

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

Sounds pretty strong (and it is). But it is not “perfect” – e.g.,

```
A           B
// Initially A, B are 0
write(A, 1)
write(B, 1)   sleep(1 year)
              read(A), read(B)
              printf("A=%d B=%d", A, B)
```

output “A=0 B=0” is legal under sequential consistency!

→ Expect certain staleness guarantees

– **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

**strict coherence** = delta coherence, delta = 0

**linearizability** = sequential consistency + delta coherence + delta = 0

## 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 dilemma** (Brewer): A system can not have sequential Consistency and maintain 100% Availability in the presence of Partitions.

→ 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.

- 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.

e.g.,			
while(1)	while(1)	while(1)	while(1)
write(A, I++)	write(B, j++)	print(A, B)	print(A, B)
		(1,1)	(1,1)
		(1,2)	(2,1)
		(1,3)	(3,1)
		(1,4)	(4,1)
		(2,5)	(5,3)

This is causally consistent, but not sequentially consistent.

This would be useful if A and B were on different nodes on internet – I might see the closer node’s updates before the more distant nodes, and you might see a different order...

e.g.,		
while(1)	while(1)	while(1)
write(A, I++)	write(B, readA)	print(A, B)
		(1,1)
		(1,2)
		(1,3)
		(1,4)
		(2,5)

No longer causally consistent – if I see new value of B, then I need to see new value of A

– **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?

(A weaker semantic allows more legal orderings than a stronger semantic. Consistency semantic A is stronger than consistency semantic B if any sequence of read and write results that are legal in A are also legal in B but there is at least one sequence that is legal in B but that is not legal in A.)

## How to provide improved consistency across clients?

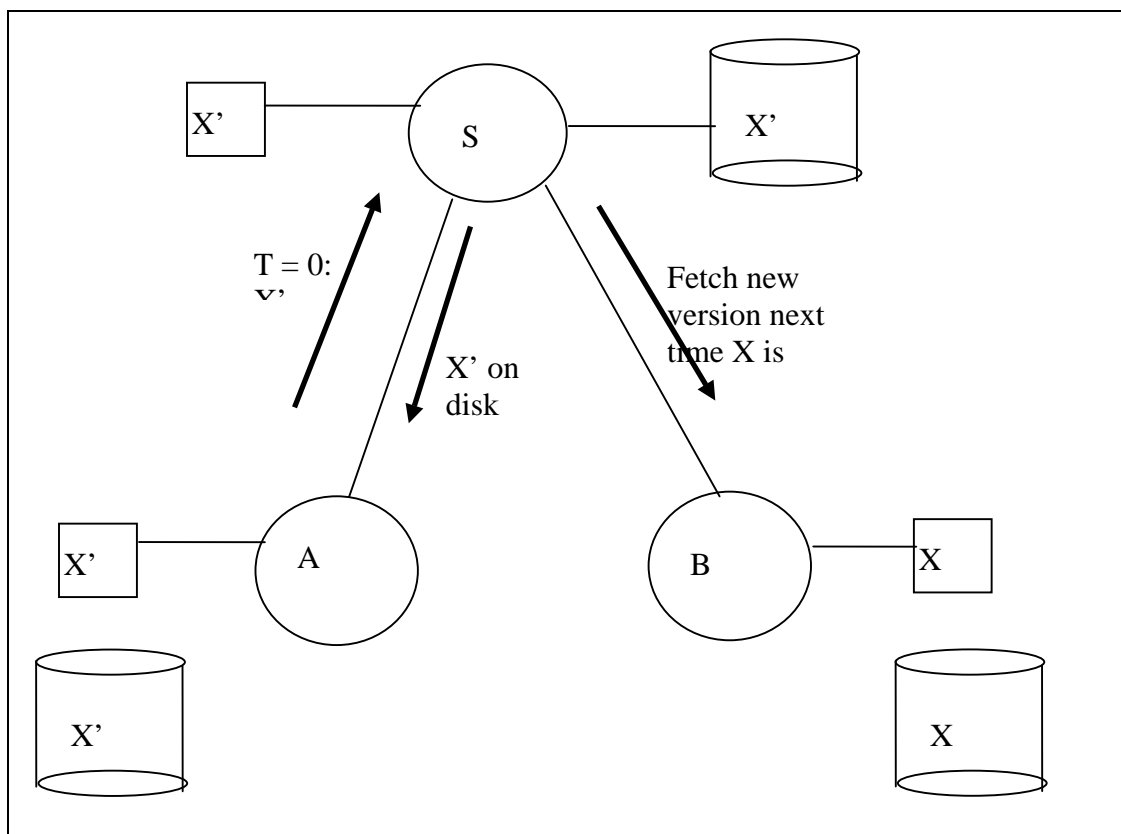
- (1) Poll each read – send every read to central server → get centralized semantics (e.g., can get sequential consistency this way – see the global order?)

We can optimize this with getattr(), but still slow...

## Callbacks (e.g., Sprite, Andrew File System)

AFS (CMU late 80's) → DCE DFS (commercial product)

Notify client if data they are caching is no longer valid



Callbacks:

- (1) When a client reads data from server, server remembers that client has data
- (2) When client writes data, server notifies all other clients (that are caching the data) that they must contact server on next read

Write begins  
Tell server “I want to write foo”  
Server tells all clients “discard current copy of foo”  
All clients acknowledge  
Server tells writer “ok to write foo”  
Write completes

What semantics does this provide?

- sequential coherence if cache blocks on local miss (what happens if read of X is allowed to pass read of Y in the cache?)
- sequential consistency if write blocks until all invalidations acknowledged (when does a write “complete”?)

### ***Fault tolerance 1: Recovery of callback state***

AFS approach: protocol level design (e.g., ad-hoc)

Challenge: improved caching + consistency increases failure handling complexity:

What if server crashes? Lose all callback state

QUESTION: Why is this a problem?

QUESTION: What can you do?

Reconstruct callback information from clients – go ask everyone  
“who has which files cached?”

QUESTION: What if client crashes?

### ***Fault tolerance 2: Leases***

Write completes when all caching clients have acknowledged

QUESTION: why do I have to wait?

[answer – you can return early if you are willing to weaken semantics... but if you want sequential, you have to wait]

Naïve solution: client blocks indefinitely if any client crashes

- How does this scale as we increase # clients?

Solution: combine polling and callbacks

Lease: cache has the right to access cached object X for Y seconds; after Y seconds, must renew lease before accessing cached object

Server does callbacks for X seconds after lease

New solution:

- (1) Write waits until all caching nodes acknowledge or leases expire (sequential coherence)
- (2) Write returns immediately (delta coherence)

Enhancement: Volume lease...

## Other AFS features

1) files cached on local disk

NFS caches only in memory

→ reduce server load

2) 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, everyone who has file open sees old version; anyone who opens file again will see new version

In AFS slight variation: session semantics

- a) on open and cache miss – get file from server; set up callback
- b) on write close: send copy to server; tells all clients with copies to fetch new version on next open

Essentially – think of all reads happening when file opened and all writes happening when file closed...

### **AFS pros & cons**

Relative to NFS, less server load:

- + disk as cache → more files can be cached locally
- + callbacks → server not involved if file is read-only

- more complex recovery

### ***Fault tolerance 3: Disconnected operation***

AFS stores data on local disk

Suppose server crashes – can client keep going?

- almost – except renewing callbacks on open/close; writing though on close

Support *disconnected operation* – allow client to access cached data even when it cannot contact server.

- Improve availability
- Support mobility

Coda (and NTFS)

- (1) Want to make sure you have everything in cache you need. What should you do? (Hoard list)
- (2) Need to make sure that updates you did when disconnected make it back to server. What should you do? (Log writes + reconciliation)

CAP dilemma: Cannot provide sequential consistency and 100% availability in a system that can be partitioned.

- What consistency does this provide? (causal?)

Conflicting writes...

What happens if two disconnected nodes both write same file? Is this OK?

Coda solution:

- (1) Detect
- (2) Regular files: manual selection of “right” version to keep
- (3) Directories: automatically correct most cases (manual for the rest)

\*\*\*\*\*

Summary - 1 min

\*\*\*\*\*

Next time: improve consistency, 2 phase commit → atomic distributed updates