Lecture N1: Networks and Distributed systems

****

Review -- 1 min

****

RAID

Rethink the sync (guest lecture)

- Performance v. durability
- Example

o T1 begin		
o W1		
o W2	barrier	barrier
o T1 end	Barrier + block	
o T2 begin		
o W3		
o W4	barrier	barrier
o T2 end	Barrier + block	
o T3 begin		
o W5		
o W6	barrier	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

****

Distributed systems intro
Basic NW communication
-- send/recv packet

-- routing
-- DNS
-- reliability
-- sharing
-- performance
-- RPC

II Distributed systems
3 problems

- performance
- consistency
- 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 cloud	$0.1K $1K	10B ???	1/100 1/1K-1/10K (bursty)

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; professional management of system

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

Physical reality v. desired abstractions

Desired abstraction: Programming/using distributed system looks like programming/using centralized system

- Location independence
- Performance
- consistency
- Failures, reliability
- security

Location independence – step 1 – how to assemble distributed system…

Message transmission/delivery

From the point of view of operating system, network is just another I/O device

In particular, NIC -- network interface controller on bus

Send/receive messages by DMA or PIO/Memory mapped I/O -- transfer message from memory to NIC or vice versa

[[picture]]

Routing

Routing -- need to get message to right process on right machine

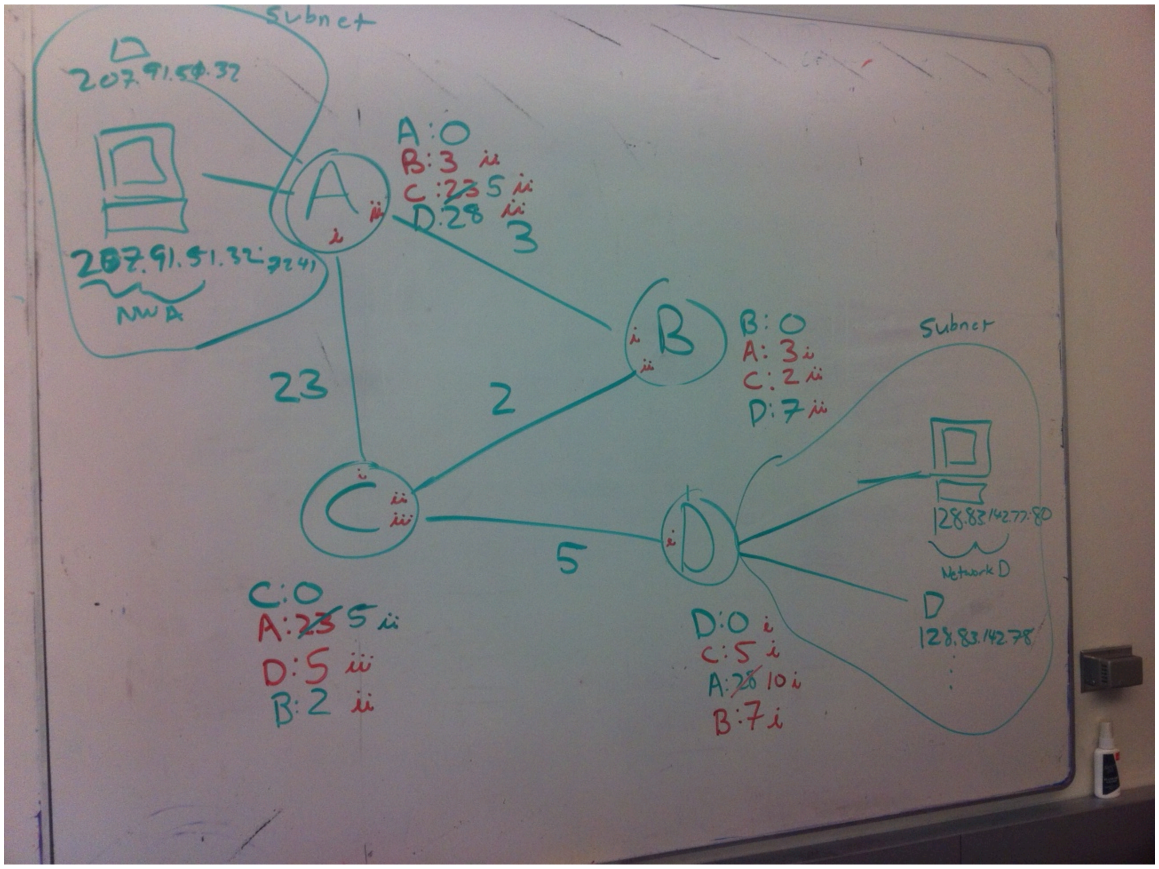
Each machine has an ID (e.g., IP address 128.83.141.37)
A process on a machine can create a port
--> e.g., utcs web server is 128.83.120.139:80

So, task is to get packet from a port on one machine to a port on another machine

Example: RIP routing
(old/simplified version of Internet routing; can be used within an organization; not sufficient across organizations -- security, policy issues; BGP there...)

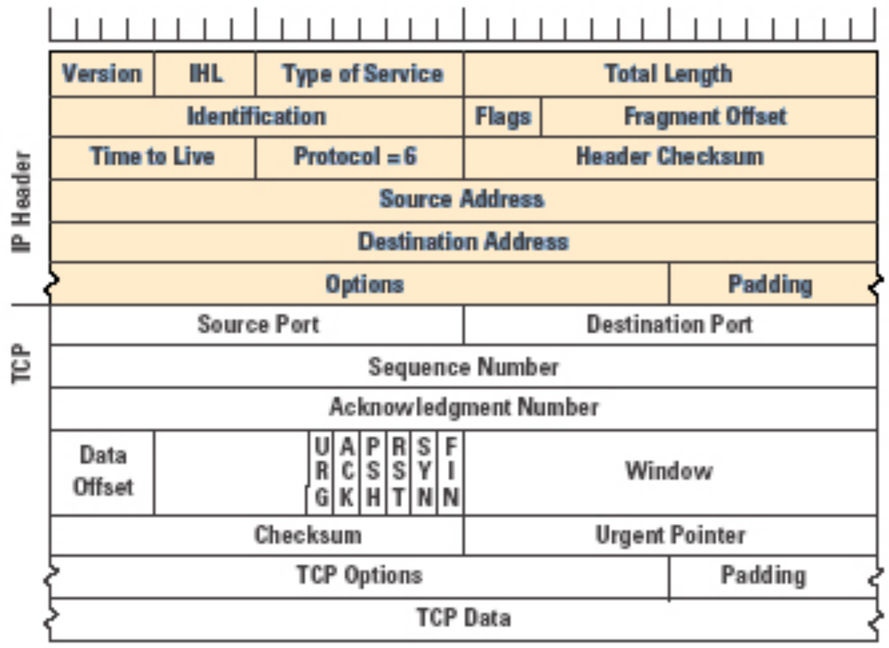
For Internet IP routing, machine IP address is <network><host> --> route to right network, then switch(es) send packet to right hpst on network

(1) Learning routes -- RIP
RIP protocol builds shortest path tables in router e.g., (simplified -- just to get intuition that this all plausibly can be done...)



(2) sending packet
-- device driver puts packet in NIC. Needs to specify destination

layered address:
IP header: source/destination IP addresses
TCP header: source/destintation port numbers



typically -- add one more layer: source/destination ethernet addresses
Sender needs to send ethernet packet to IP router --> learn IP router's IP address (from DHCP or manual configuration); then learn IP router's ethernet address (from ARP -- broadcast "who has this IP address" and back comes needed ethernet address)

This may happen at each hop

ARP cache --> don't have to keep asking via network

--> OK. So now we can get packets from here to there (and back)

so outbound data path is [cover of Comer's book]

application
TCP queue of sent packets
TCP output <--- TCP timer OR UDP
IP
ARP

DRIVER

DNS

How know IP address of www.cs.utexas.edu

Domain name system (DNS)
-- client knows IP address of DNS server
-- client can ask "give me IP address for <name>"
-- DNS, itself, is a distributed protocol (different servers cooperate to provide service) -- skipping details

Message loss

Problem: packets can be lost
-- interference (especially wireless network), overflowing buffers at routers or receivers

example: 2 nodes sending at full speed to 1 node [picture]

Solution 1: Request/reply or receive/acknowledge

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)

--> "At least once" semantics -- receiver receives each packet at least once (but maybe multiple times) (assuming neither sender nor receiver crashes or gives up)

+ Simple, good match with request/reply communications patterns
- Low throughput for large requests (1 packet per round trip latency. e.g., 1KB per 10ms --> 100 KB/s)

Solution 2: Pipeline solution 1 -- multiple packets in flight; resend unacked packets after timeout

Optimizations:

(1) cumulative acks -- ack of packet i means that all packets up to i have been received

(1a) Combine acks -- don't send ack for each packet; send for every other packet, etc.

(2) immediate resend on nack -- when receiver recieves packet i, ack i; then receives packet i+2 (missing i+1). Can't ack i+2 (b/c cumulative ack); instead resend ack i; sender receives "ack i; ack i" and knows that i+1 was not received --> resend it immediately

(3) [often bad] Delayed acks -- for bidirectional communication, application layer at receiver will likely send data back to sender; so, don't ack the packet at network level; instead, count the reply as the ack. (In TCP, each data packet I send also carries acks for all that I've received on stream)

(4) [often bad] Nagle's algorithm -- combine small packets to reduce overheads ("as long as there is a sent packet for which sender has not received ack, buffer output until packet is full")
--> Made sense for telnet on modem; probably not useful for real time video game on LAN...
--> HORRIBLE interaction with delayed acks (optimizations were introduced by different groups at about the same time -- early 1980s...)

Sharing the network

Network is shared resource with no global "root"
How do we keep a malicious user or faulty program from hogging the network

ANSWER: We can't (DDOS attacks)

OK. How do we get normal users and programs from hogging network/how do we divide network resources fairly?

IP level: Overloaded switches drop packets

PICTURE

TCP level: Adaptive send rate
-- start slow
-- if no losses, increase rate
-- if loss, reduce rate
--> Overloaded router causes TCP to slow down

Details
Van Jacobson and Michael Karels "Congestion avoidance and control"
(Classic paper)

" In October of '86, the Internet had the first of what became
a series of 'congestion collapses'. During this period, the
data throughput from LBL to UC Berkeley (sites separated
by 400 yards and two IMP hops) dropped from 32 Kbps to 40 bps."

Key idea: Conservation of packets -- when running near capacity, don't put a new packet in until an old packet leaves network

5 fixes to previous TCP to get conservation of packets:

(i) slow-start
-- congestion window -- cnwd -- max # of packets in flight
-- on start or cnwd = 1
-- on ack, increase cwnd by 1
(not so slow -- doubles cnwd on each round trip)

[[aside -- sender or receiver may have max cwnd. This may limit bandwidth for long paths -- if RTT is high, need deep pipeline to fill it.]]

(ii) round-trip-time variance estimation
-- TCP uses resend on timeout
-- Problem: variance rises rapidly with load
-- e.g., at 75% load, round trip times can vary by a factor of 16
-- old timer caused many unnecessary retransmissions under load
--> throwing gasoline on a fire

-- new timer much better

(ii) exponential retransmit timer backoff
-- you provably need this for stability
-- this is why your web browser stalls for 5 seconds then 30 then...
(hint: hit "reload" if page not there in 5 seconds...)

(iv) more aggressive receiver ack policy

(v) dynamic window sizing on congestion
-- additive increase, multiplicative decrease
-- halve cwnd on loss
-- increment by 1/cwnd packets on each successful ack (1 packet per round trip; much slower than "slow start")
-- "traffic jam effect" -- easier to get into congestion than to get out of it...

--> reasonably fair sharing

bandwidth = k(B/R sqrt(p))
-- B packet size
-- R round trip time
-- p packet drop probability

--> flows at congested link with same packet size and same round trip time will get same fraction of bandwidth (since they have same drop probability)
--> if different round trip times, then "closer" one can do much better

"TCP Friendly" -- protocols expected to be TCP friendly -- whatever congestion avoidance algorithm they use, it should not send more than k(B/R sqrt(p))

-- easy to write code that is not tcp friendly; don't do this.

(k = 1.2247)

Bonus observation -- small loss rates kill you for long-haul links

Using messaging to build services

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 direcction (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

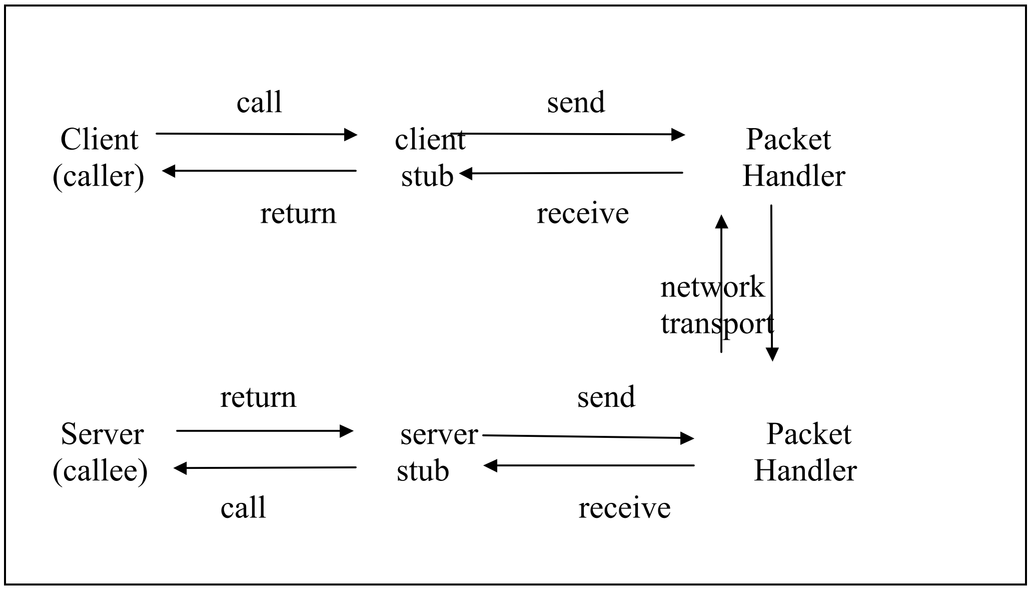
```
client
    remoteFileSys->Read("rutabaga");
```

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

Problem solved?

RPC provides location transparency – except

Performance
Failures -- message loss, machine crash
Consistency/replication
Security

- All hard problems.
- Fundamental limits (e.g., you can't atomically update an object replicated at multiple machines)
- Diffcult trade-offs among goals -- e.g., consistency v. availability CAP

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:~~
~~4) Sender sends message (msg, msgId) and sets timer~~
~~5) Receiver receives message and sends (ack, msgId)~~
~~6) Sender receives (ack, msgId) and clears timer~~

~~If timer goes off, goto (1)~~

~~How does this work? Local procedure call guarantes *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 (but what if receiver crashes and loses seq number info?)~~

~~in general -- request may be executed 0, 1, 2, or more times.~~

~~(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~~

Network (I/O) performance: LogP model.

--> If you want good network performance, need to **pipeline** requests
[picture]

Pipeline more complex than for CPU because (a) not fixed number of stages, (b) not balanced stages... essentially, CPU designed around pipeline --> simpler to think about it's pipline. In distributed systems, need a more complete model.

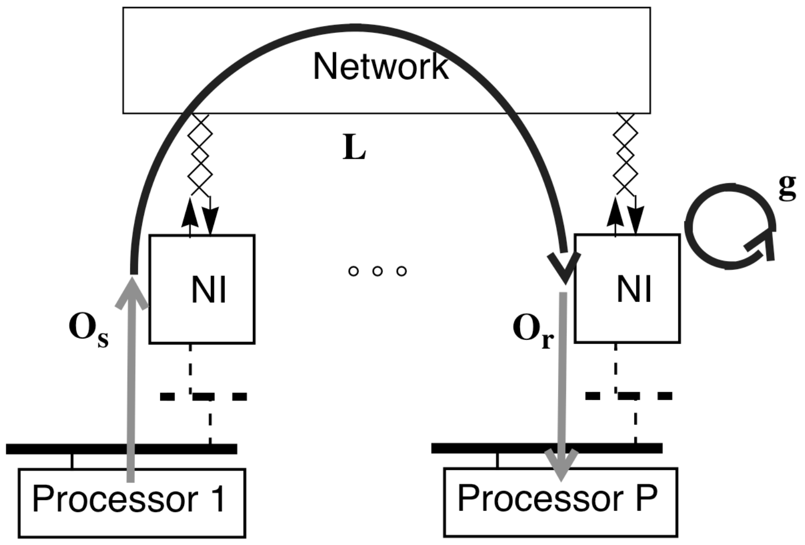


FIGURE 1. LogP parameters in a generic platform

Latency – Elapsed wallclock time from X to Y.
note: "latency" alone is ambiguous; need to say latency from what to what; "latency from x to y". E.g., 1-way latency v. round-trip latency v. ...

e.g., how long from when one byte packet sent to when it is received

NOTE: can hide latency with **pipelining**; latency does not imply resource is busy. Latency tells you how deep pipeline needs to be.

Overhead – Time for first pipeline stage.
Bottleneck time to initiate operations. (Can't be overlapped.)
e.g., Cpu time to put packet on wire.

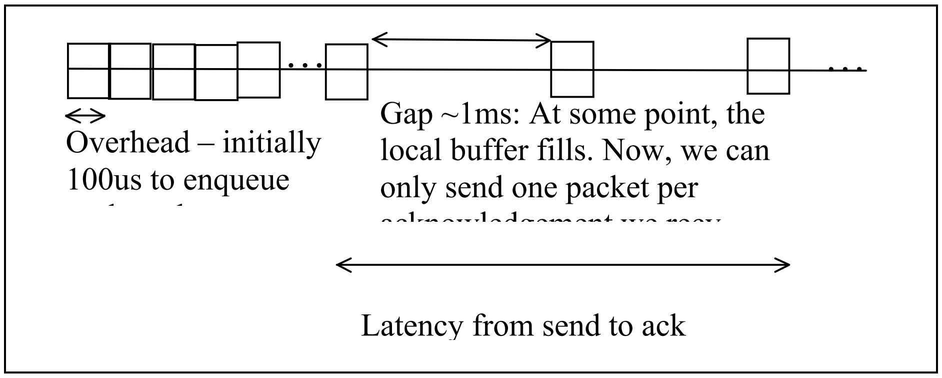
Throughput, bandwidth – (1/gap) -- time for bottleneck stage. Maximum steady state rate.
Time consumed by slowest pipeline stage.
e.g., maximum bytes per second

How does overhead differ from latency?
    Overhead: resource usage
    Latency: real-time end-to-end delay
How would you measure latency of a network request?
How does overhead differ from latency?

How does overhead differ from bandwidth?
How would you measure overhead of sending a packet?
How would you measure bandwidth of a network?

Example

E.g., suppose you open a TCP connection and start sending 1KB messages to another node on a 10Mbit/s Ethernet


Overhead – initially 100us to enqueue
Gap ~1ms: At some point, the local buffer fills. Now, we can only send one packet per
Latency from send to ack

1) What is "bottleneck rate"? (for overhead, BW)

The only tricky thing about this is that you have complex pipelining models (e.g., a disk request "occupies" CPU, bus, scsi controller, scsi bus, disk arm)
Which one is the bottleneck depends on configuration (how many disks? How many SCSI busses? How fast CPU?)

Which one is the bottleneck depends on how question is asked:
E.g., "For a Seagate Barracuda 5100 disk, what is the average overhead per 1-sector disk request?" v. "For a Dell Dimension 5100, what is the overhead per 1-sector disk request?" The first is asking how long a disk seek and rotation take; the second is asking how long the CPU is busy to set up a request.
Need to consider: What bottleneck is the question asking about?
For throughput, steady state bottleneck is the same in both cases.
For overhead, first stage overhead differs.

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

Overhead (network send/receive)-- 10's-100's us to send/receive TCP/IP packet; 1-10us for streamlined protocols

Bandwidth -- 1Mbit/s 3G phone, 1-10 Mbit/s home internet connection, 10-50Mbit/s WiFi, 100-1000Mbit/s desktop, 1Gbit/s-10Gbit/s data center network

	Throughput	Overhead	100 byte	4KB	Remote 4kB read
TCP/IP Wireless	10 Mbit/s	0.1 ms	.1ms + .08ms	.1ms + 3ms	3.3ms
TCP/IP Ethernet	100 Mbit/s	0.1 ms	.1ms + .008ms	.1ms + .3ms	0.5 ms
TCP/IP Gigabit Ethernet	1000 Mbit/s	0.1 ms	.1ms + .0008ms	.1ms + .03ms	0.2 ms

AM/ Myrinet	1200Mbit/s	.007ms	.007ms + .001ms	.007ms + .03ms	.04ms

Example
Create 1MB file using NFSv3. Close --> client needs to write back 1MB to server.
How long?

Assume client sends one 4KB block at a time, waits for server to get block safely to disk.

100Mbit/s network; o_send = o_recv = 100us
each block:

> 100us (o_send) + 4KB/100Mbit/s + 1us (latency from last byte off NIC to last byte arrives) + 100us (o_recv) + 10ms (disk) + 100us (o_send) + .5KB/100Mbit/s + 1us + 100us (o_recv)
>
> = 100us + 300us + 1us + 100us + 10000us + 100us + 35us + 1us + 100us = 10702us

1MB/4KB = 256 blocks

> 256 * 10702us = 2.75s

1 Gbit/s network
each block:

> 100us + 4KB/1Gbit/s + 1us + 100us + 10ms + 100us + .5KB/1Gbit/s + 1us + 100us
> = 100us + 30us + 1us + 100us + 10000us + 100us + 3.5us + 1us + 100us = 10436us
>
> 256 * 10436 = 2.74s

MORAL: fast network doesn't buy you much if you haven't paid attention to latency and overhead.

Example (cont)
---> Instead of sending one block at time, send all blocks. Then wait for server to send "Ack" saying all on disk

1 Gbit/s network:
-- bottleneck is o_send and o_recv
-- picture
-- Time: 256*100us (now last packet starting to go on wire)
    + 4KB/1000Mbit/s (now last packet entirely on wire)
    + 1us (now last byte of last packet arriving at receiver)
    + 100us (now last packet received at receiver)
    + 4KB/100MB/s (now last packet on disk; assuming end of streaming, sequential write)
    + 100us (now ack is on wire)
    + 256B/1000Mbit/s (now ack on wire)
    + 1us (now last byte of ack is at receiver)
    + 100us (now ack received at receiver)

    = 25600us
    + 30us
    + 1us
    + 100us
    + 40us
    + 100us
    + 1.5us
    + 1us
    + 100us
    = 26236us
    = 26ms

MORAL: Need to pipeline to get good performance from IO systems

Notes on example
-- NFS v3 kind of works like first example (except 5-10 outstanding requests at a time instead of 1); NFS v4 adds support for something like second approach

-- Send/receiver overheads in example are clearly too high. Any modern machine can keep a 100Mbit/s or even 1Gbit/s network "full" of 4KB messages.
(What does o_send need to be?)

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:

- (1) in LAN – minimize overhead
- (2) in WAN – keep pipeline full

Example: LogP network benchmark

LogP Performance Assessment of Fast Network Interfaces
www.cs.rutgers.edu/~rmartin/papers/papers/micropaper.ps

```
Sender: (uniprocessor)
Thread1:
Start timer
  repeat M times
      send msg
      spin for delta
Stop timer

Thread2:
while (1) receive msg


Receiver:
while(1)
   receive msg
   send reply
```

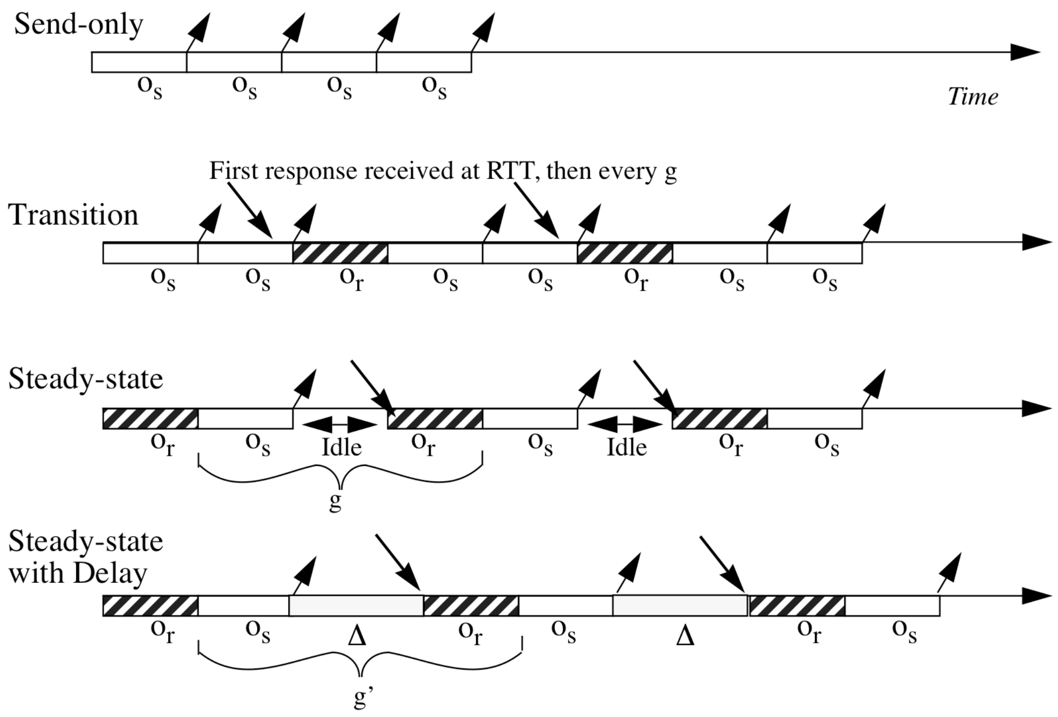


FIGURE 3. Request issue time-line

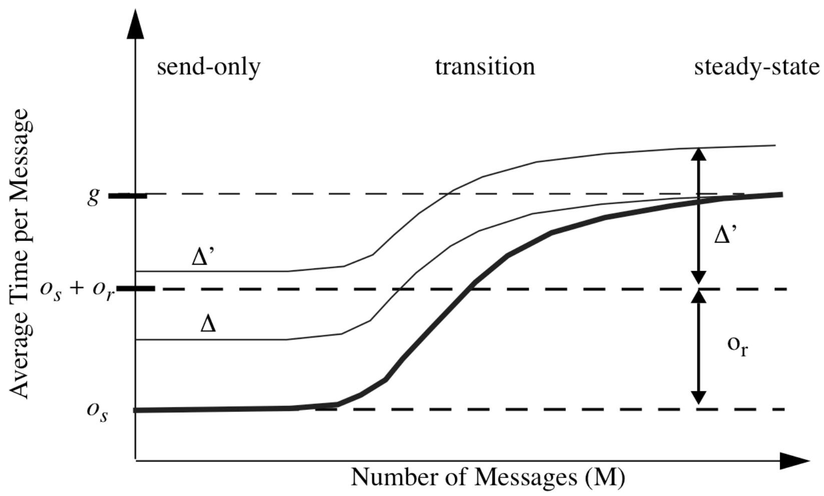


FIGURE 4. Expected microbenchmark signature

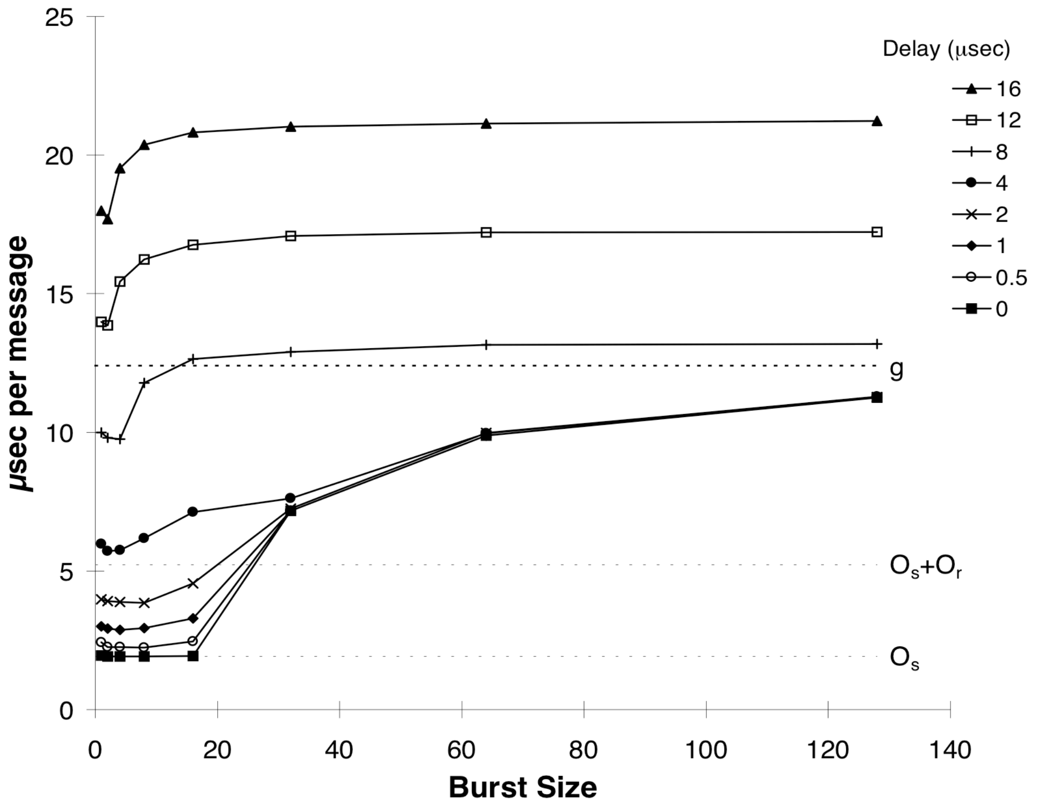


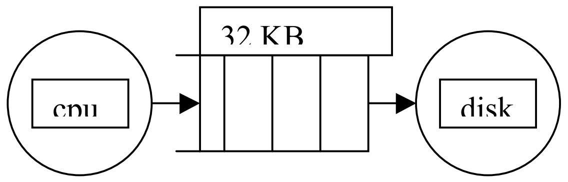
FIGURE 6. **Myrinet microbenchmark signature**

Example: Batching
General rule of thumb: OS provides abstraction of byte transfers, but batch into block I/O for efficiency (pro-rates overhead and latency over larger unit)

Example

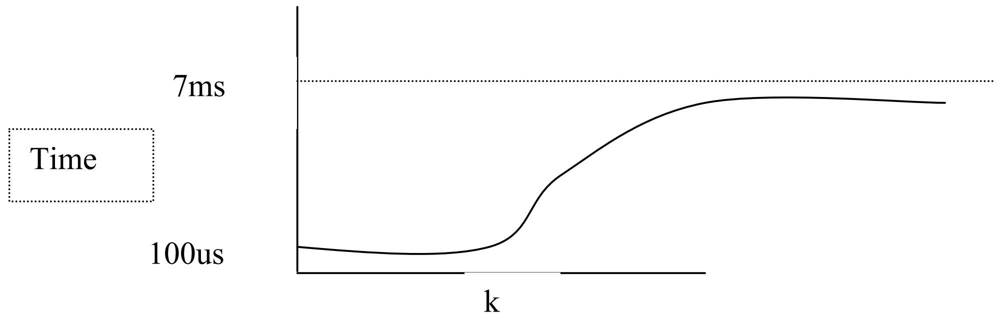
- Suppose CPU takes 100us of processing to issue one 512 byte write request
- Each request is to a random sector on disk
- Disk has parameters as above (4ms avg seek, 3ms ½ rot, transfer .02ms)
- 32KB write buffer on disk (producer/consumer bounded buffer)

- Writes are issued asynchronously (CPU can issue k+1 as soon as k is in write buffer)



o: 100us | 0 |
7ms
1 100us | 0-64*7=448ms |

(1) Suppose CPU issues k back-to-back requests, when does CPU complete?



(2) When does first write to disk complete at the disk?

(e.g., latency from when first write starts at CPU until done at disk?)

7.1ms

(3) Suppose there are 500 writes in a burst, when does the last write complete at the disk?

100us + 500 * 7ms

****
Admin - 3 min
****

****
Lecture - 23 min
****

Distributed file system

Outline
Distributed File Systems
2 Case studies: NFS, AFS
Crosscutting issues
Performance
Failures
Cache coherence/consistency
Distributed commit

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

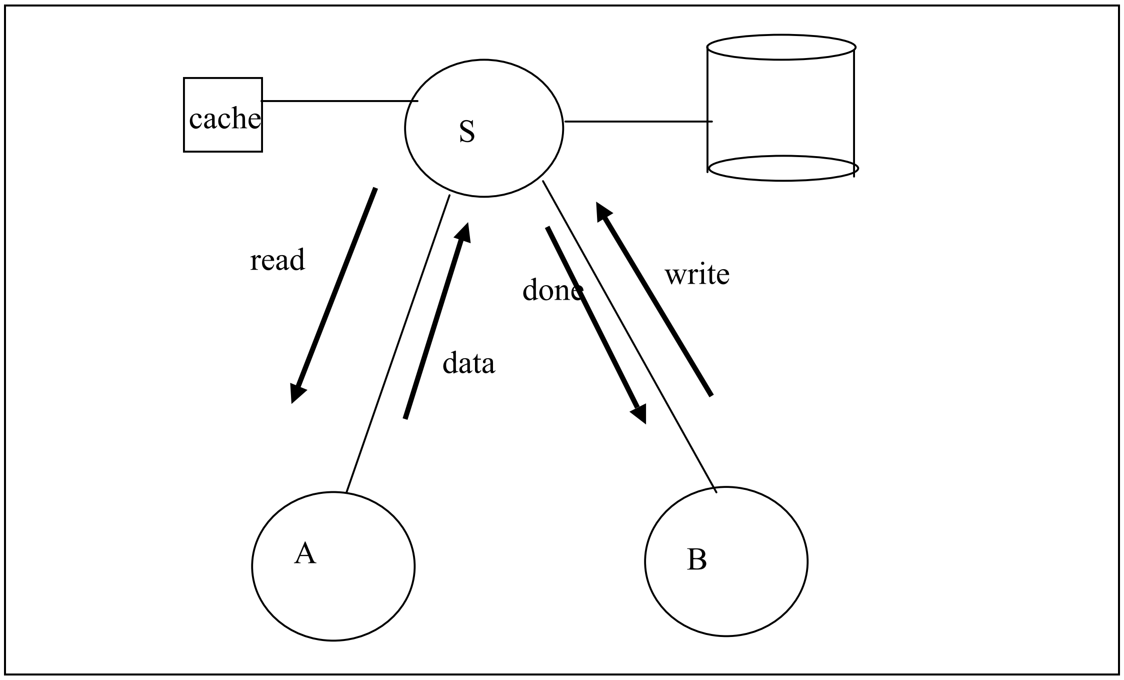
These issues and strategies we will discuss are much more general than file system – arise in many distributed systems.

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

issues: Failures, performance
Failures – see NFS (below)

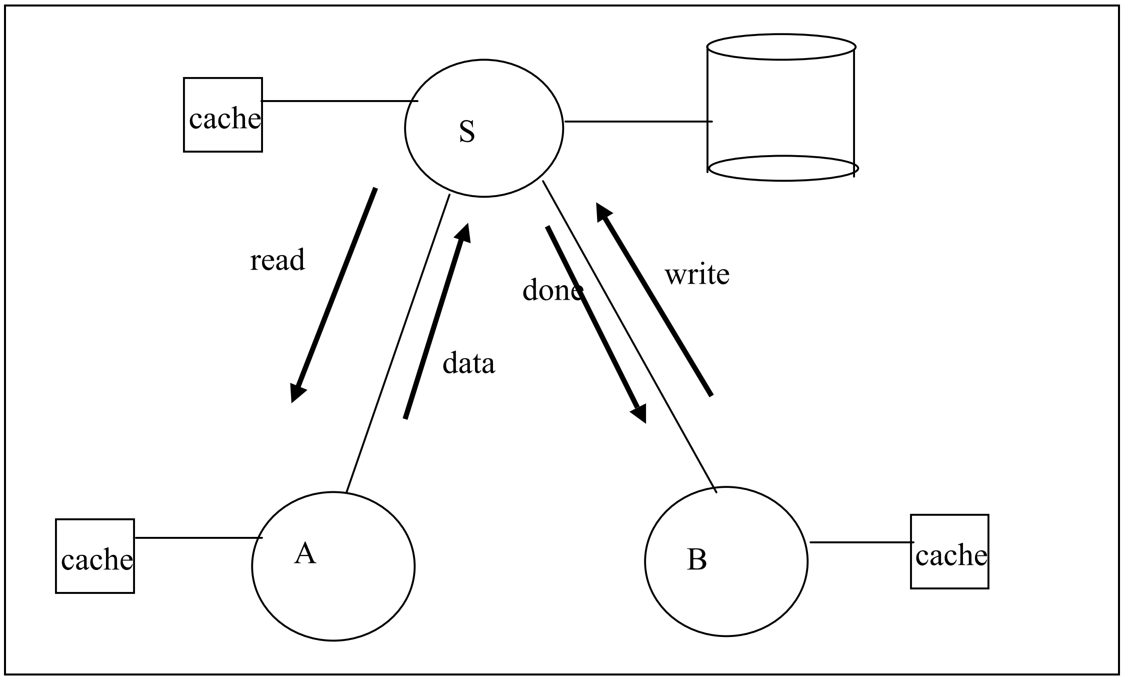
Performance can be lousy:
- going over network is slower than going to local memory!
- lots of network traffic
- server can be a bottleneck – what if lots of clients?

NFS (Sun Network File System)

(I'll talk about "NFS v.3" to illustrate issues in a simple system; NFS v. 4 makes significant changes, including some of the state-of the art techniques I'll talk about later this week...)

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) helps performance; challenges 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

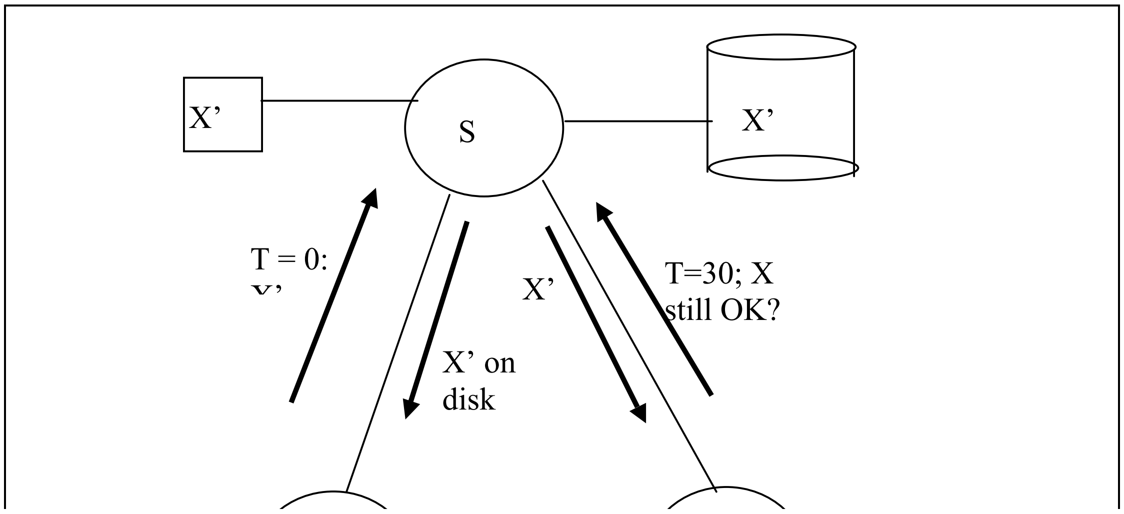
- write through within 30 seconds or immediately when file closed

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

If rule #1 in CS is "any problem can be solved with an additional level of indirection", Dahlin's rule #2 is "I can make it go as fast as you want, as long as you don't need the right answer"

We'll talk about better ways to enforce consistency next week.

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): solutions

Key idea: Server is **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 server disk.

Client caches dirty data until close. Client failure --> data loss. Network write (to server) -- block until data safely 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) Timeout and repeat requests to mask lost messages

Standard RPC technique.
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 guarantes *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

3)

4) 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

5) 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!

Next 2 weeks -- address fundamental problems in distributed systems. You see them in NFS
-- performance
-- consistency
-- distributed commit
-- security
After we talk about (some of) these, we'll revisit in context of a scalable cluster file system: The Google File System

****
****
****

Performance

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

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

Caching can help, but

- generally gets rid of "transparent stub generation" advantage of RPC
- makes failure handling more complex, raises consistency issues
- Not work for all worlkloads, all cases. (E.g., web caching -- data changes, zipf distribution --> client caches have 20-50% hit rate --> network performance dominates (amdhal's law)

--> *Caching alone can't fully mask slow network.*

NFS Example:

File close needs to write back all dirty sectors from client cache to server disk.

Network performance

"How fast is your network?"

Bandwidth isn't whole story. Bandwidth is the MIPS of I/O
In architecture, MIPS is one of three factors (cycles per instruction, instruction count, instructions per second) -- only looking at one is misleading
Similar issues for IO

Example

Suppose I have a 100Mbps and 1000Mbps network. Is second network 10x faster?

Not if I use it to do a "remote read" (50 byte request, 50 byte response)

Graph: (lab) 510us (100Mbps), 501us (1000 Mbps)

(Graph: fixed portion + variable portion…)

Cross-country: 50.5ms (10Mbps), 50.5ms (100Mbps)

What's going on?

Example

e.g., Suppose I replace load/store from local memory with load/store from remote machine via network.

Bandwidth not *that* different -- maybe 10-100 GB/s v. 10 Gbit/s (2011) --> 10-100x

But slowdown would probably be many times that (1000x-100,000x)

Other factors

-- Latency. Speed of light to get across building (~us)/campus(100us)/country(10's of ms) (v. 100ns to memory)

-- Overhead. Thousands of instructions to send/receive a packet (100us to send/recv a packet)

Result: Even if network bandwidth is 10Gbit/s, if I only access one remote word at a time, I'll probably see an effective bandwidth of 1 word per 100us or 1ms (100-1000x slower)

So, if bandwidth alone can get you off by a factor of 1000x, how do you reason about performance?

****

****

****

Cache consistency

Today: cache consistency – callbacks, leases

Wednesday: reliability

Recall -- NFS caching sometimes gives wrong answer

-- client caching data checks with server to see if still valid if it has been more than 30s since last check

--> Window of vulnerability
--> My compiles occasionally fail and I tear my hair out

"I can make any system run fast as long as you don't insist on the right answer."

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
    - o Caching, write buffering, multipath routing through network
    - o → 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)
    - o 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?"
    - o 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++){      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?
  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:                                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 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:                                 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."

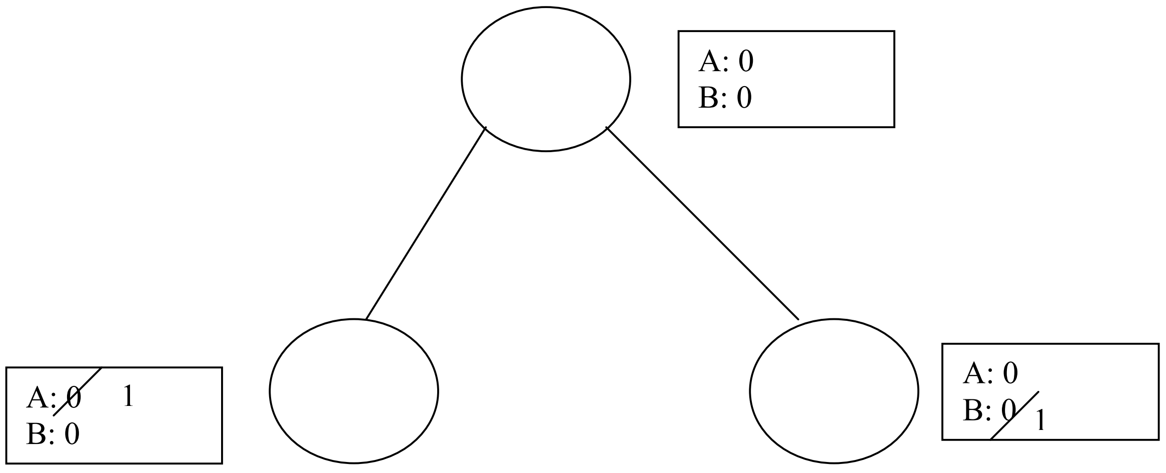
– Why might a system exhibit inconsistency?

– Notice

In first example, order between writes must be maintained...fairly obvious notion of causality

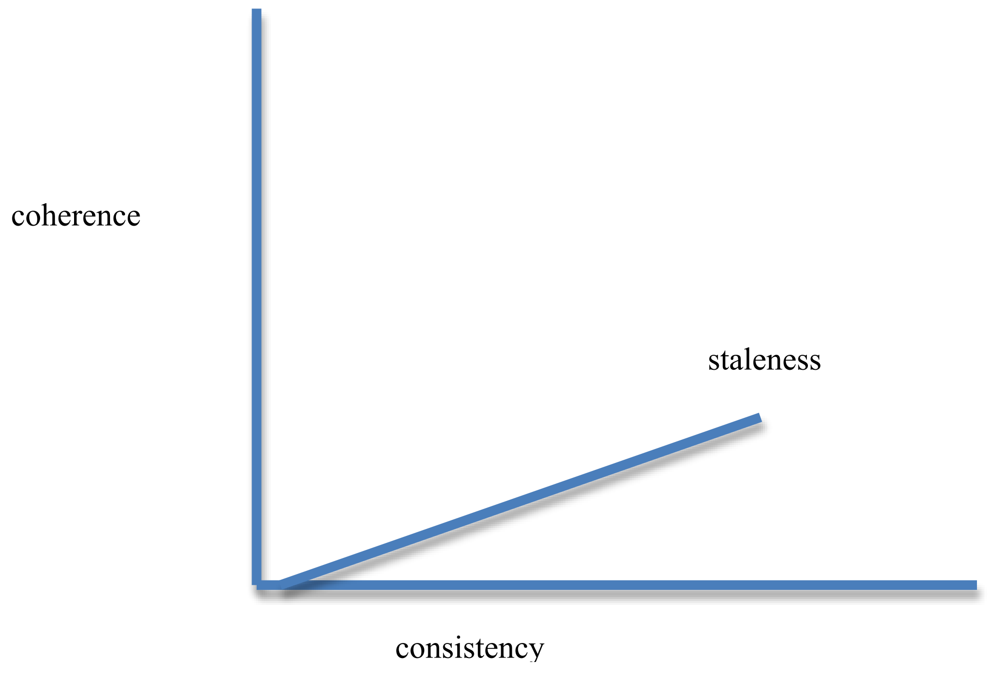
In second example, order between writes and reads must be maintained. (Less obvious?)

Consistency involves ordering both writes and reads.



Semantics for non-atomic memory

Above defines 3 axes/dimensions:



Now we can start talking about particular design points in this space.

One option: Insist that distributed memory look "just like" local memory

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

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

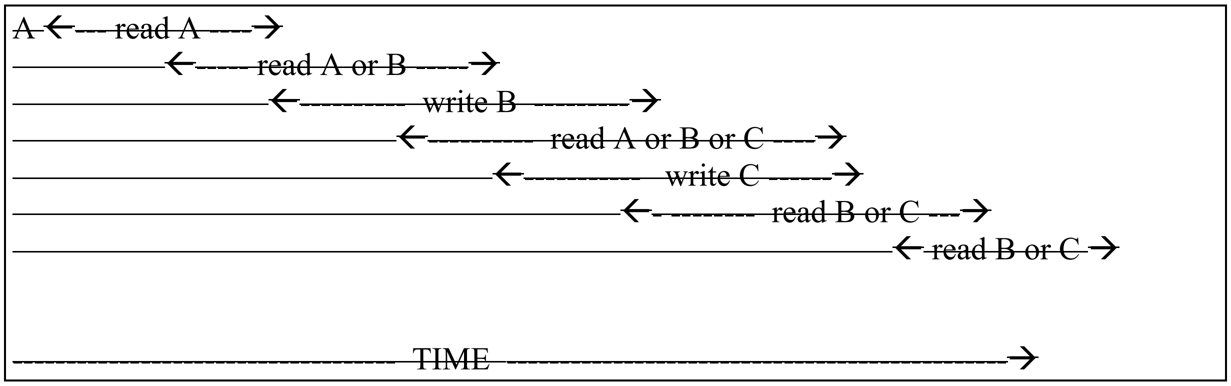
strict coherence = delta coherence, delta = 0

linearizability = sequential consistency + delta coherence + delta = 0
(formal definition is essentialy -- sequential consistency + the global sequence is consistent with real time)

--> linearizability is essentially the origin in the design space -- the strongest consistency we typically ask for

~~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~~


~~A ←--- read A ----→~~
~~←----- read A or B -----→~~
~~←---------- write B ---------→~~
~~←---------- read A or B or C ----→~~
~~←----------- write C ------→~~
~~←- -------- read B or C ---→~~
~~←read B or C→~~
~~TIME →~~

~~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 -- indeterminant -- 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. Othewise, 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~~

Limitations of strong consistency

So, we can define "perfect" consistency/coherence/staleness.

Are we done? "Distribibuted systems should implement linearizability"???

Unfortunately, no. Implementing these semantics has costs. Some of these costs are fundamental (and sometimes they are unacceptable.)

– Sequential consistency has fundamental performance cost: fast reads or fast writes but not both
r + w >= t (where r is read time, $w is the write time, and t is the minimal packet transfer time between nodes.) [Lipton and Sandberg]

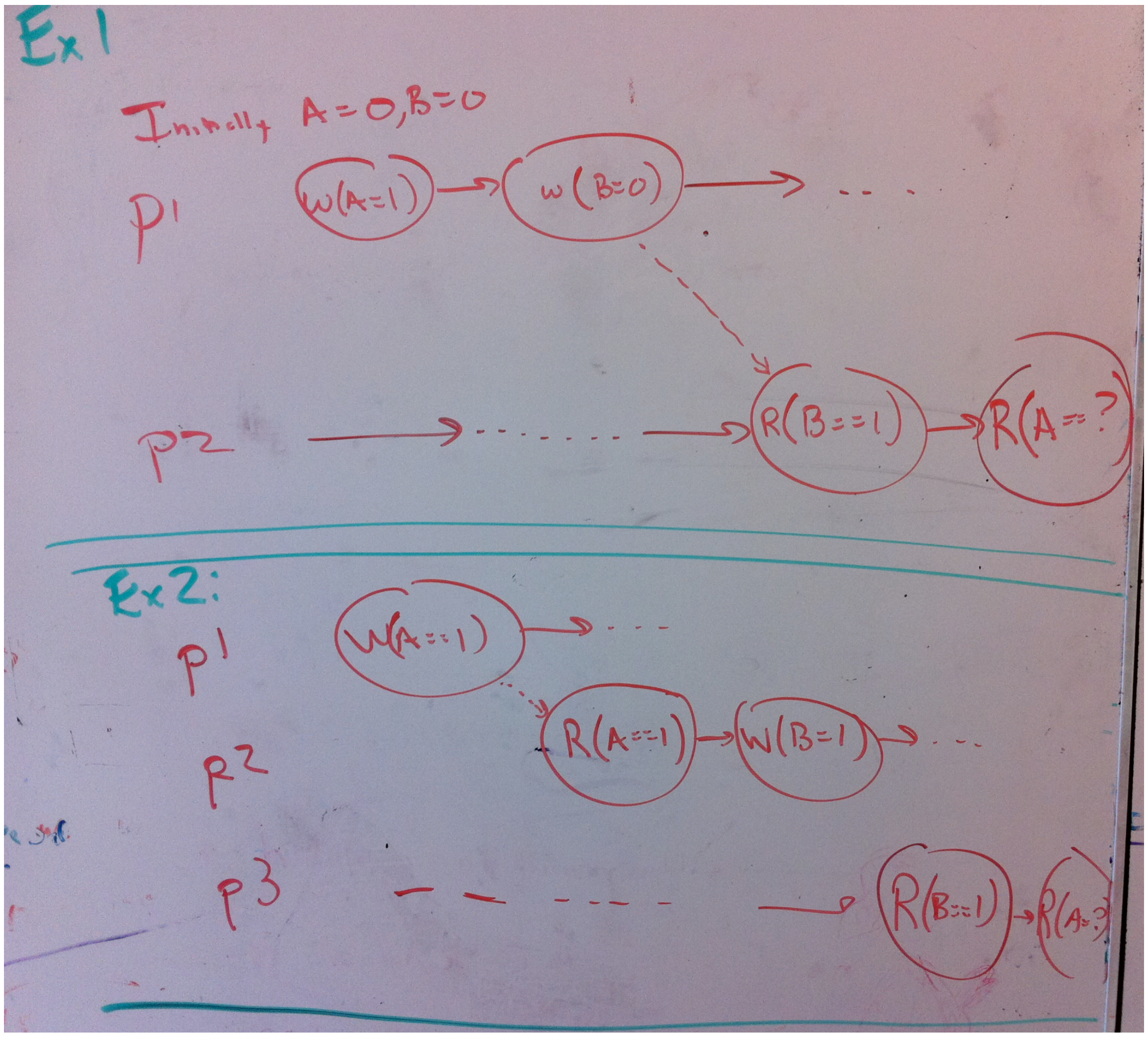
– Sequential consistency has a fundamental **CAP dilemma** (Brewer): A system can not have sequential **C**onsistency and maintain 100% **A**vailability in the presence of **P**artitions.

→ 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.

Basic idea -- if I see a write that you issued, then I can also see (at least) all writes you could have seen when you issued the write

- 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

Theorem: Causally consistent is the strongest consistency you can provide without giving up availability. (Dahlin, Alvisi, Mahajan -- April 2011)

There are also weaker options

– ***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.)

Why would you ever want this? Requires no coordination at all. E.g., 2 web servers....

How to provide improved consistency across clients?

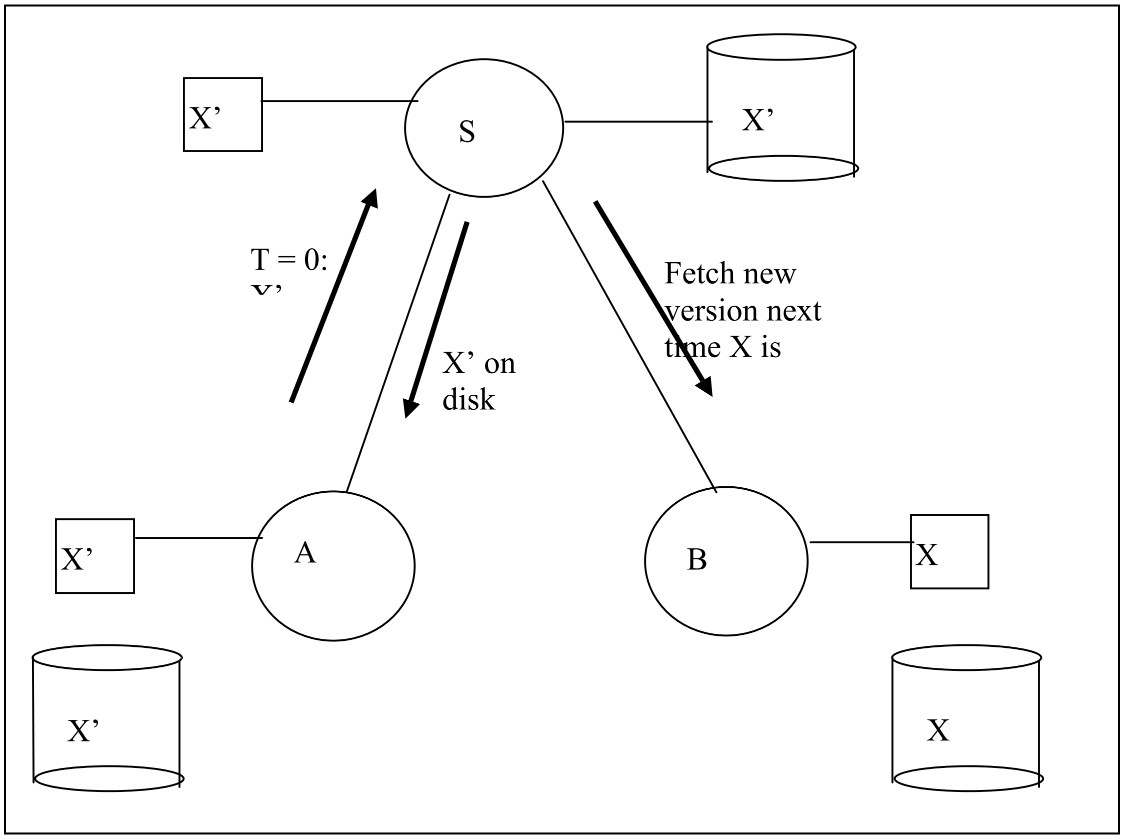
(1) Poll each read – send every read to central server → get centralized semantics (e.g., can get sequential consistency this way – see the global order?)

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?

- linearizability if client issues one operation at a time and blocks until completion
- (weaker if I can, say, read from cache while my write is pending)

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: CAP -- consistency v. availability during partitions

Key idea: Leases

CAP says sequential consistency must give up availability during partitions

How does this manifest in AFS?

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 linearizability, you have to wait]

Naïve solution: client blocks indefinitely if any client crashes

- How does this scale as we increase # clients?

Solution: **lease --** 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

Leases do a pretty good engineering job on CAP dilemma. If I can talk to server, I can access data. Clients disconnected from server are stuck. (Notice -- they are stuck even if they have the data they want to read in their cache.)

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) Reads -- prefetch "hoard" data into local cache
Want to make sure you have everything in cache you need. What should you do? (Hoard list)

(2) Writes -- write updates to local log; send log to server when reconnect
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?)

Problem: 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)

Avoiding central server

Coda lets me write when disconnected, but all updates go through server

What if you don't want to have to synchronize through a server

Basic idea

- Each node's writes are a log [picture]
- Version vector – index of highest known write from each node
- Log exchange – you send me your VV, I send you all updates you have not yet seen
- → Eventual consistency
- Lamport clock – my *accept stamp* = max(VV) + 1
- Send elements from my log sorted by accept stamp
- → Causal consistency
- Still need to deal with conflicting concurrent writes (how can you detect?)

Google file system [see gradOS notes]

Consistency in memcached

memcached: reading from a database is slow
--> have another set of machines act as a cache (distributed hash table)
[picture]

basic idea:

```
read(x)
    data = memcached->read(x)
    if(data) return data
    else
        data = db->read(x)
        memcached->set(x, data)
        return x
```

```
write(x, data)
    db->write(x, data)
    memcached->set(x, data)
```

What incoherence might you observe?
How long can incoherence last (how much staleness)
What if writer crashes after setting DB but before setting memcached?

Obvious fix (?)

```
write(x, data)
     memcached->clear(x);
     db->write(x, data);
     memcached->set(x, data);
```

Does this solve the problem?

****

Summary - 1 min

****

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