Today's Objectives

- Wrap up Distributed File Systems
- Timing

Sakai Poll Exam Replacement Day Results

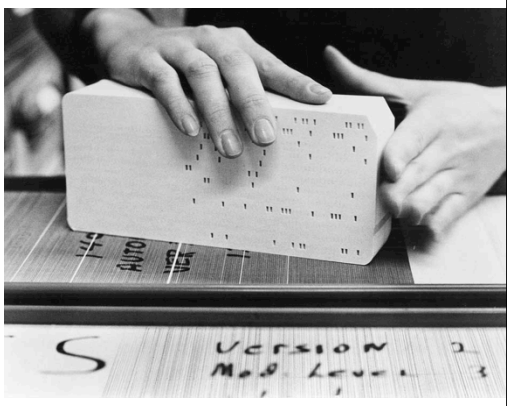
- Wednesday, November 15 - 2 - 14%
- Friday, November 17 - 12 - 86%

- Last class before break: Wednesday

- Exam will go out tomorrow
- Can start Wednesday at midnight

Inverted Index Project

- Due tonight
- Like old-timey programming
  - Want to make sure your program is really good before running
  - Takes a long time to get feedback



http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/punchcard/breakthroughs/

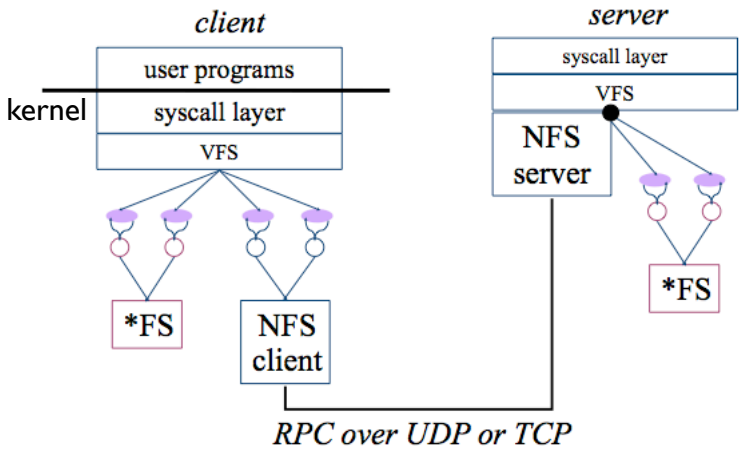
Review

- What is the motivation for a distributed file system (DFS)?
- How does a DFS make remote files look the same as local files?
- What are some policies that DFS can use when managing file caches?
  - Consider: what happens when a client updates a file?
- What is NFS?
  - What is its protocol built on?

Review: Sun NFS

- Sun Microsystem's Network File System
  - Widely adopted in industry and academia since 1985
  - (we use it)
- All NFS implementations support NFS protocol
  - Currently on version 4
  - Protocol is a set of **RPCs** that provide mechanisms for clients to perform operations on remote files
  - OS-independent but originally designed for UNIX

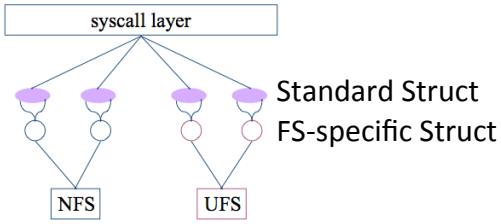
Network File System (NFS)



VFS=Virtual File System

VFS: Vnodes

- Every file or directory in active use is represented by a virtual node or ***vnode*** object in memory
  - Each file system maintains a cache of its vnodes
  - Each vnode has a standard file attribute struct
  - Each standard struct points at file-system-specific file attribute struct



Stateless NFS

- NFS server maintains no in-memory hard state
  - Only hard state is stable file system image on disk
  - No record of clients or open files
  - No implicit arguments to requests (no server-maintained file offsets)
  - No write-back caching on server
  - No record of recently processed requests
- Why?

Stateless NFS

- NFS server maintains no in-memory hard state
  - Only hard state is stable file system image on disk
  - No record of clients or open files
  - No implicit arguments to requests (no server-maintained file offsets)
  - No write-back caching on server
  - No record of recently processed requests
- Why? **Simple recovery after server failure!**

Recovery in NFS

- If server fails and restarts, no need to rebuild in-state memory state on server
  - Client reestablishes contact
  - Client retransmits pending requests
- Classical NFS used UDP
  - Server failure is transparent to client since there is no "connection"
  - Sun RPC masks network errors by retransmitting requests after an adaptive timeout
    - Dropped packets are indistinguishable from crashed server to client

NFS Server Caching

- Cache read results, writes, directory operations
- Write-through cache vs. write-back cache?
  - **Write through**: Each update written to disk immediately
  - When write operation returns, client is guaranteed stable update
- Pros:
  - Stateless (easy to implement), no data lost on crash
- Cons:
  - Slow: client must wait for disk write

Drawbacks

- Stateless nature has obvious advantages but also some drawbacks
  - Recovery by retransmission constrains server interface
    - "Execute mostly once" semantics = send and pray
    - Executions usually only happen once, but not guaranteed
  - Update operations are disk-limited (write-through cache)
  - Server cannot help in client cache consistency

NFS Client Caching

- Clients cache read, writes, and directory ops
  - What if multiple people updating the same file at the same time? Consistency problems!
- NFS approach:
  - Server maintains last modification time/per file
  - Client remembers time it initially retrieved data
  - On file access, client checks timestamp against server (every 3-30 seconds)
    - Unnecessary timestamp checking
    - How long to set the timeout? What is the tradeoff?

TIME AND GLOBAL STATE

Time

- Time is an important practical issue in distributed systems
  - Example: often require computers to timestamp electronic commerce transactions

Why is that problematic?

Time

- Time in an important practical issue in distributed systems
  - Example: often require computers to timestamp electronic commerce transactions
- But time can be problematic
  - Physical clocks in computers are not all synchronized
  - There is no global clock in distributed systems
- Need a way to order events and approximate time synchronization in distributed systems

Process States

- How can we order and timestamp the events that occur across all distributed processes?
- Assume a distributed system consists of N processes
  - Each process executes on a single processor
    - Memory is not shared
  - Each process *p* has state *s*
    - Includes values of all variables and objects in *p*
  - Processes can only communicate via sockets

Events

- An ***event*** is an occurrence of a single action that a process carries out as it executes
  - Either a communication action or state-changing action
- **Happens-before relationship: →**
  - Order events within a single process so that $e \rightarrow e'$ iff e occurs before e'
- Define the history of process p_i to be the series of events within it, ordered by relation →
  - history(p_i) = $h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

Time Design Questions

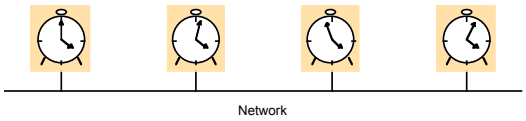
- How accurate does time need to be?
- How is time used in a distributed system?
- What does "A happened before B" mean in a distributed system?

Clocks

- Ordering events in a process is not the same as assigning a timestamp to them
- Timestamps require date and time of day
- Computers have *hardware* clocks
- OS reads hardware clock and adds some offset to produce *software* clock
- Thus we can timestamp events using software clocks ***only if*** the *clock resolution* is smaller than interval between events
- Works for one process but will it work for N distributed processes?

Problems with Clocks in Distributed Systems

- Clock skew
  - Instantaneous difference between readings of any 2 clocks



- Clock drift
  - Problem that occurs when two or more clocks count time at different rates

Research Question: Can we synchronize physical clocks across computers to provide global event ordering across processes?

Synchronizing Physical Clocks

- External synchronization
  - Synchronize physical clocks with some external source of time
  - UTC = Coordinated Universal Time
- Internal synchronization
  - Synchronize using the time between events that occur on different computers ("logical clocks")
  - For clocks C_i and C_j, if we know $C_i - C_j < D$, then we know the clocks agree within the bound D
- Internal synchronization *does not* imply external synchronization!
  - But external synchronization does imply internal synchronization

Synchronous Systems

- Simplest possible synchronization case: internal synchronization in synchronous systems
  - Sync systems usually use blocking `send` and `recv` calls
- In a synchronous system, we know:
  - Max drift rate of clocks
  - Max transmission delay
  - Time to execute each step of the process
- Synchronization
  - One process sends time *t* to other process in message *m*
  - Receiving process sets clock to be *t + transmission_time* of *m*

Problems?

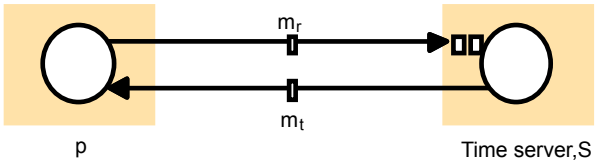
Synchronous Systems

- Transmission time is subject to variation!
- But we know the *min* and *max* transmission time
- Uncertainty in transmission time = *max - min*
- Set clock halfway between: *t + (max-min)/2*
- Skew is at most *(max-min)/2*
- In general, for *N* clocks, optimum bound on clock skew is *(max-min)(1-1/N)*

But, most systems are asynchronous…

Cristian's Method

- Most distributed systems are *asynchronous* → unbounded transmission delay
- Round trip times (RTTs) are often reasonably short (in LANs)
- Cristian suggested a probabilistic algorithm using a time server for external synchronization in asynchronous systems
  - Process requests time in m_r and gets response in m_t
  - t is time according to S (the time server)
  - T_{round} is time between sending m_r and receiving m_t
  - Process sets clock to be $t + T_{round}/2$



Problems?

Problems

- Time server is single point of failure!
  - But can replicate…
  - …as long as the replicas stay synchronized
- Faulty time server could wreak havoc on distributed system using Cristian's method

Berkeley Algorithm

- How can we deal with faulty clocks?
- Gusella and Zatti developed algorithm for internal synchronization in LANs
- One computer is chosen as producer, other computers (who want to be synchronized) are the consumers
  - Producer polls consumers for local clock values
  - Producer estimates RTTs between consumers
  - Producer takes a "fault-tolerant" average of all values obtained to determine "global" clock value
    - Eliminates readings from faulty clocks
  - Producer sends back individual "skews" (+/-) to each consumer

Network Time Protocol (NTP)

- Cristian's method and the Berkeley algorithm are intended primarily for use within intranets
  - Rely on relatively low latency measurements between participants
- Need a method for distributing time information and external synchronization over the wide-area (like the Internet)
  - Must be able to deal with variations in latency

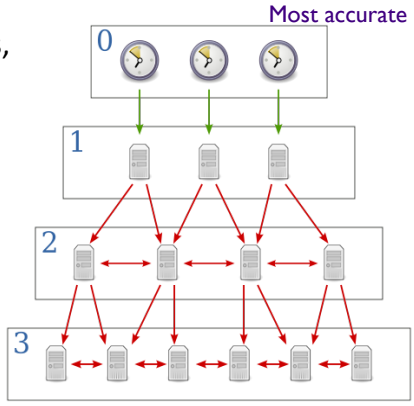
Solution: NTP

NTP

- Developed by Dave Mills at University of Delaware
- Initially developed in early 1980s
- Runs over UDP on port 123
- Specifically designed to handle effects of variable latency measurements (often called *jitter*)
- Goals: reliability, scalability
- Synchronizes clocks to UTC

NTP Clock Strata

- Stratum 0: atomic clocks, GPS clocks, radio clocks w/ UTC
- Stratum 1: Time servers (primary), attached directly to Stratum 0 devices
- Stratum 2: Send requests to one or more Stratum 1 time servers
- Stratum 3: Send requests to one or more Stratum 2 computers
- And so on…
- Up to 256(!) strata levels supported in current version of NTP

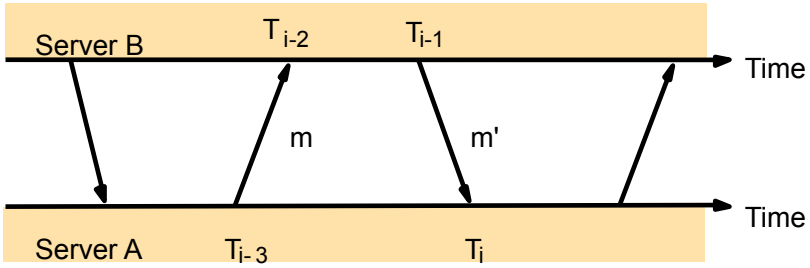


Lowest leaf: users' workstations Reconfigurable in response to failures

https://en.wikipedia.org/wiki/Network_Time_Protocol#/media/File:Network_Time_Protocol_servers_and_clients.svg

Synchronizing Servers

- All messages sent using UDP
- Each message bears timestamps of recent events:
  - Local times of *Send* and *Receive* of previous message
  - Local times of *Send* of current message
- Recipient notes the time of receipt T_i
  - Have T_{i-3}, T_{i-2}, T_{i-1}, T_i



Accuracy of NTP

- For each pair of messages between two servers, NTP estimates an offset o between the two clocks and a delay d_i (total time for the two messages, which take t and t')

  $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$
- This gives us (by adding the equations) :

  $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
- Also (by subtracting the equations)

  $o = o_i + (t' - t)/2$ where $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$
- Using the fact that t, t'>0 it can be shown that

  $o_i - d_i/2 \le o \le o_i + d_i/2$.
  - Thus o_i is an estimate of the offset and d_i is a measure of the accuracy

NTP Statistics

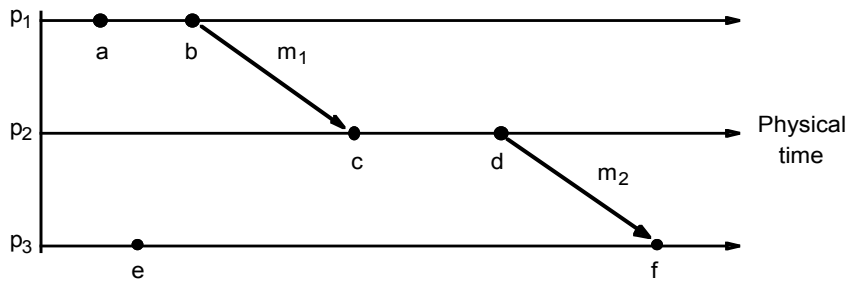
- In 1999 there were 175,000 hosts running NTP in the Internet
- Among these there were:
  - Over 300 valid Stratum 1 servers
    - Never contacted directly, except by Stratum 2
  - Over 20,000 servers at Stratum 2
  - Over 80,000 servers at Stratum 3
- Accuracy of 10s of milliseconds over Internet paths (even more accurate on LANs)

Source: http://www.ntp.org/ntpfaq/NTP-s-def.htm

LOGICAL CLOCKS

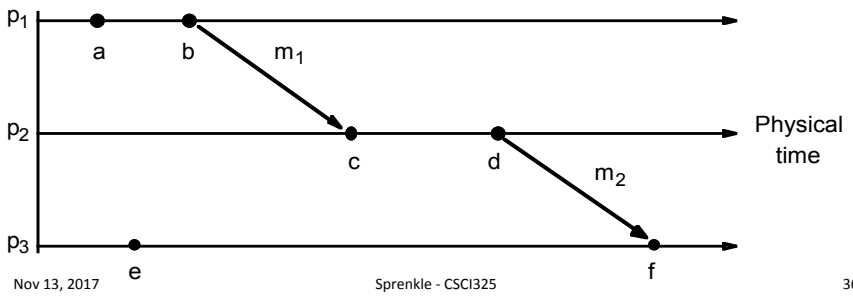
Logical Time and Logical Clocks

- Instead of synchronizing clocks, event ordering can be used
- Rules:
  1. If two events occurred at the same process p_i (i = 1, 2, … N) then they occurred in the order observed by p_i, that is $\rightarrow_i$
  2. When a message m is sent between two processes, *send(m)* happened before *receive(m)*
  3. The happened-before relation is transitive



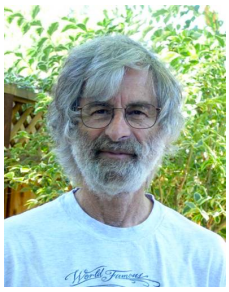
Happened Before Relation

- What do we know about events a, b, c, d, f?
  - Rule 1: a → b (at p_1), c → d (at p_2)
  - Rule 2: b → c (by m_1), d → f (by of m_2)
  - Rule 3: a → b → c → d → f = a → f
- What do we know about a and e?
  - No relation → they are concurrent: a || e



Lamport's Logical Clocks

- A logical clock is a monotonically increasing software counter
  - Need not relate to a physical clock

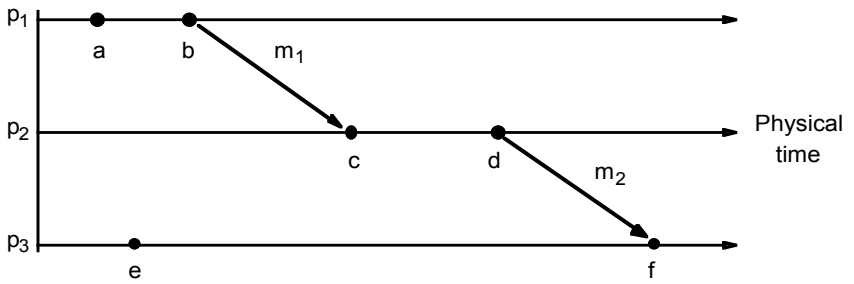


Leslie Lamport

LaTeX

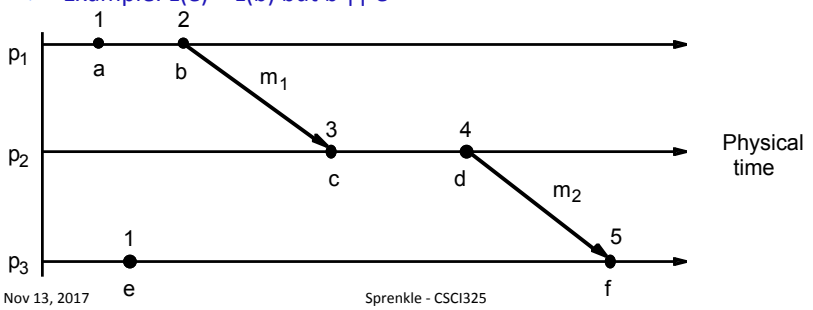
Lamport's Logical Clocks

- Each process p_i has a logical clock, L_i
  - Can be used to apply *logical timestamps* to events using rules:
    - LC1: L_i is incremented by 1 before each event at process p_i, $L_i = L_i + 1$
    - LC2:
      a) when process p_i sends message m, it piggybacks on m the value $t = L_i$
      b) when p_j receives (m,t) it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (m)



Lamport's Logical Clocks

- Each of p_1, p_2, p_3 has its logical clock initialized to zero
- The clock values on events are those immediately *after* the event
  - e.g., 1 for a, 2 for b.
- For m_1, t = 2 is piggybacked and c gets L_2 = max(0,2)+1 = 3
- Note that e → e' implies L(e) < L(e')
- Does L(e) < L(e') imply e → e' ?
  - No! The converse is not true: L(e) < L(e') does not imply e → e'
  - Example: L(e) < L(b) but b || e

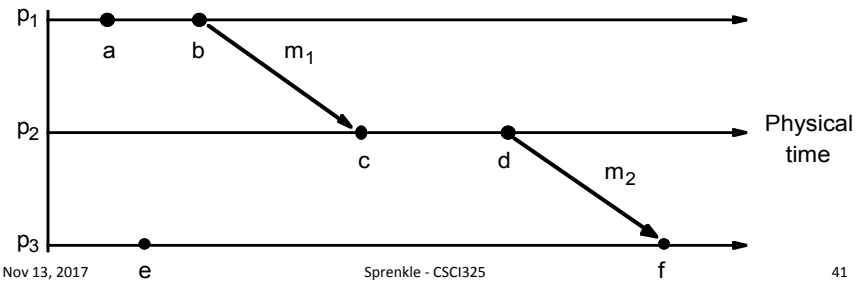


Lamport Clocks → Vector Clocks

- Limitation of Lamport clocks:
  - L(e) < L(e') does not imply e happened before e'
  - If L(e) < L(e'), we want to know *for sure* that e happened before e'
- How can we overcome the limitation?
- Solution: **Vector clocks**
  - ***Vector*** timestamps (rather than a single number) are used to timestamp local events
  - Vector clock $V_i[i]$ is the number of events that p_i has timestamped
  - $V_i[j]$ ($j \neq i$) is the number of events at p_j that p_i has been affected by
- Vector clocks are used in many schemes for replication of data to ensure consistency

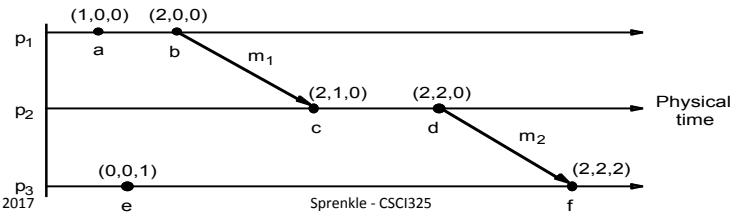
Vector Clocks

- Vector clock V_i at process p_i is an array of N integers
- Rules for determining vector clocks:
  - VC1: Initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
  - VC2: Before p_i timestamps an event, it sets $V_i[i] = V_i[i] + 1$
  - VC3: p_i piggybacks $t = V_i$ on every message it sends
  - VC4: **Merge**: When p_i receives (m,t) it sets $V_i[j] := \max(V_i[j], t[j])$ $j = 1, 2, \ldots N$



Vector Clocks

- At p_1: a(1,0,0), b(2,0,0), piggyback (2,0,0) on m_1
- At p_2: On receipt of m_1 get max ((0,0,0), (2,0,0)) = (2,0,0), and add 1 to own element in clock = (2,1,0) for event c
- At p_3: On receipt of m_2 get max ((0,0,1), (2,2,0)) = (2,2,1) and add 1 to own element in clock
- Vector timestamp operations: =, <=, max, etc.
  - Compare elements pairwise
- Note that e → e' still implies L(e) < L(e')
- And now the converse is also true (L(e) < L(e') implies e → e')
- Can you see a pair of parallel events?
  - c || e because neither V(c) <= V(e) nor V(e) <= V(c)



Summary: Time and Clocks in Distributed Systems

- Accurate timekeeping is important for distributed systems
- Algorithms (e.g., Cristian's and NTP) synchronize clocks in spite of their drift and the variability of message delays
- For ordering an arbitrary pair of events at different computers, clock synchronization is not always practical
- The *happened-before relation* is a partial order on events that reflects a flow of information between them
- **Lamport clocks** are counters that are updated according to happened-before relationship between events
- **Vector clocks** are an improvement on Lamport clocks
  - By comparing vector timestamps, can tell whether two events are ordered by happened-before or are concurrent