Time and Global States

Prof. Dave Bakken

Cpt. S 464/564 Lecture

November 29 & December 4, 2000

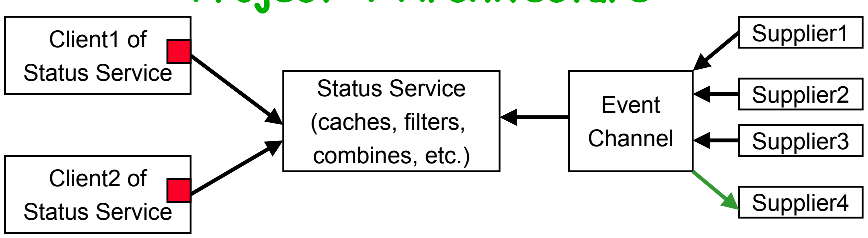
Administrative Items

- Handouts
  - Paper "Practical Uses of Synchronized Clocks in Distributed Systems" by Barbara Liskov (required for 564 only)
  - Paper "Time, Clocks, and the Ordering of Events in a Distributed System" by Leslie Lamport. (required for 564 only)
  - Homework #4
- Slightly updated grading weights:

Component	464	564
Exams (2):	45%	30%
Homeworks (5) and Surprise Quizzes :	15%	20%
Projects (5):	40%	40%
Particiation	0%	10%

- Project #4 discussion, then Chapter 10…

Project 4 Architecture



Grading Criteria

- 40% Baseline with "no frills"
- 10% demo (runs OK without crashing, only 5% if no GUI)
- 20% 2 extra features (564 only; 20% max, i.e. no extra credit)
  - 10% use event model other than canonical push
  - 10% use object wrapper at client to cache a value
  - 10% different kind of client to status service
  - 10% client of status service gets callback
  - …. (make up your own … even better)
- 20% originality/realism of exact status service, suppliers, clients, and discussion of this in your writeup. I will provide a baseline example worth 0%..
- 10% Rest of writeup

Time in Distributed Systems

- Time is a very useful concept!
- Computers can only be synchronized by network messages, but the latency can vary…
- We can not synchronize closely enough to be able to, in general, tell the ordering of two arbitrary events at different computers
- We can, however, establish an ordering on some events, and this can be used in many situations

Outline

- **Clocks, events, and process states (10.2)**
- Synchronizing physical clocks (10.3)
- Logical time and logical clocks (10.4)
- Global states (10.5)

Notations for Reasoning about Time

- Model
  - A DS consists of a collection *P* of *N* processes $\{P_1, \ldots, P_N\}$
  - Each process executes on a single processor (no migration)
  - Processors do not share memory
  - A process P_i has state S_i which it transforms as it executes
  - Processes communicate only by message passing
- Actions a process can take
  - Send a message
  - Receive a message
  - Transform its state
- Event: occurrence of a single action above
- The sequence of events P_i can take can be placed in a single total ordering, $\rightarrow_i$, between the events
  - I.e. $e \rightarrow_i e'$ iff event *e* occurs before *e'* at P_i
  - Note: this is well-defined, even with multiple threads, because single processor

Notations for Reasoning about Time (cont.)

- History: the series of events of a process that take place within it
- $history(p_i) = h_i = <e_i^0, e_i^1, e_i^2, \ldots>$

Clocks

- We now know how to order events at a process, but how to timestamp them?
- Operating system
  - Reads in computer's hardware clock value, $H_i(t)$
  - Adds an offset to produce a software clock: $C_i(t) = A^* H_i(t) + B$
- Problem #1: physical clocks on different computers will have skew: differences at a given instance
- Problem #2: clocks will drift: they will increment $H_i(t)$ at slightly different rates
  - Drift rate: change in the offset (difference in reading) between $H_i(t)$ and a theoretical perfect clock
  - Typical drift rates are a few seconds a month
  - High precision clocks drift only a few seconds to a few dozen seconds a year

UTC and GPS

- International Atomic Time
  - Based on Cs^{133} (one second =~ 9 billion transitions).
  - Since the earth's rotation is slowing, this diverges from astronomical time.
- Coordinated Universal Time (UTC)
  - Based on atomic time
  - But with an accasional leap second thrown in to keep it close to astronomical time.
  - Broadcast on shortwave radio stations.
- GPS units can also provide time, accurate to 1 microsecond or so
- US NIST lets you dial up on a phone and get accuracy to a few milliseconds

Outline

- Clocks, events, and process states (10.2)
- **Synchronizing physical clocks (10.3)**
- Logical time and logical clocks (10.4)
- Global states (10.5)

Synchronizing Physical Clocks

- To know what time things happen at, with any degree of precision, we need to synchronize our clocks
- External synchronization: synchronizing with an authoritative time source
  - Clocks C_i are accurate to within bound $D>0$ after this
  - I.e., for authoritative source S, $| S_i(t) - C_i(t) | < D$, for all i,t
- Internal synchronization: clocks agree with each other
  - Clocks C_i agree with each other within bound D
  - I.e., $| C_i(t) - C_j(t) | < D$, for all i,j,t
- Clocks that are internally synchronized are not necessarily externally synchronized!
  - Why?
  - How?
- A clock $H_j(t)$ is correct if it meets its specs (often in terms of drift rate)
  - If incorrect, it has failed
  - Crash failure: does not return any time
  - Arbitrary failure: anything else… (what? effects?)
- Note: a clock does not have to be accurate to be correct
  - Why? Useful in some situations?

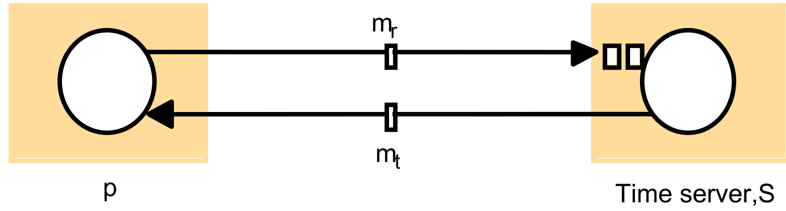
Simple Clock Synchronization

- Simplest possible case: two processes synchronize
  - Time server S sends sends message m to process p, including its current time t
- How can p set its clock?
  - $H_p(t) = t + T_{transmission}$
  - S and p are now internally synchronized
- Problem: cannot know $T_{transmission}$
- Observations
  - Can always find T_{min}
  - Can generally find T_{max} with high statistical confidence
    - 100% in a synchronous system, *ipso facto*
  - Uncertainty of message transmission: $u = (T_{max} - T_{min})$
  - Can potentially even derive a pdf of $T_{transmission}$ between and T_{min} and T_{max}
- Workaround
  - Set $H_i(t) = t + X$, where X is T_{min} or T_{max} or $(T_{max} + T_{min})/2$
  - What are worst case clock skews for each (in terms of u)?

Cristian's Clock Synchronization

- Scenario: two processes synchronize
  - Process *p* sends message m_r to authoritative time server *S*
  - *S* sends back message m_t including its current time *t*
  - *P* uses *t* in m_t to update its clock
  - Figure 10.2, next slide…
- How can *P* set its clock???
- Observation:
  - Uncertainty *u* is often small in practice
  - So provide a *probabalistic* synchronization, which depends on *u* at the time

Figure 10.2 Clock synchronization using a time server



Cristian's Clock Synchronization (cont.)

- Process *p* records round trip time
  - T_{round} = (time m_t received) – (time m_r sent)
- Naïve estimate: assume both latencies are same (reasonable)
  - $H_p(t) = t + (T_{round}/2)$
  - $u = T_{round}/2$
- Observation: can often derive T_{min}, so
  - Earliest time that *S* could have sent m_t is T_{min} after m_r was sent by *p*
  - Latest time that *S* could have sent m_t is T_{min} before m_t was received at *p*
  - Cuts worst case clock skew to $((T_{round}/2) - T_{min})$

Berkeley Clock Synchronization Algorithm

- Master time server which does not get requests from clients, but polls its slaves which are to be synchronized
- Slaves send back their clock values
- Master estimates their local clock times by observing round-trip times
- Master then averages the values to derive a new one
  - Tends to cancel out inaccuracies
- Master does not send back the new time to update to
  - Because transmission back introduces another element of uncertainty.
- Rather, it sends back the amount (+/-) by which the given slave's clock should be updated by.
- Note: Average is a *fault-tolerant average*
  - It chooses subset of clocks whose times do not differ from one another by a specified amount
  - Then takes average from these.

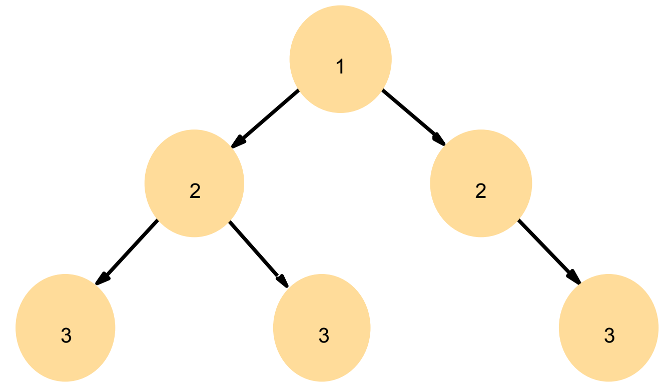
Network Time Protocol (NTP)

The standard for the Internet; design features:

- *Accurate to UTC*, despite large and varying delays
  - Discriminates between quality of timing data from different sources
- *Reliable service*, despite lengthy delays of connectivity of given links
  - Redundant servers and paths to servers
- *Allow frequent resynch* to offset drift
  - Scales to lots of clients and servers
- *Provide security against interference*
  - Uses authentication techniques, validates return addresses of messages it gets, etc.
- Architecture (Fig 10.3, next slide)
  - Logical hierarchy called a synchronization subnet
  - Primary servers (Strata 1) are directly connected to UTC
  - Secondary servers (Strata 2) synch with them
  - Etc. down the tree
- Algorithms take into account
  - Strata (lower # is better accuracy)
  - Roundtrip delays

  when assessing quality of time to assess for a given server
- Can reconfigure tree for various reasons…

Figure 10.3: An example synchronization subnet in an NTP implementation



Note:
• Arrows denote synchronization control
• Numbers denote strata

NTP (cont.)

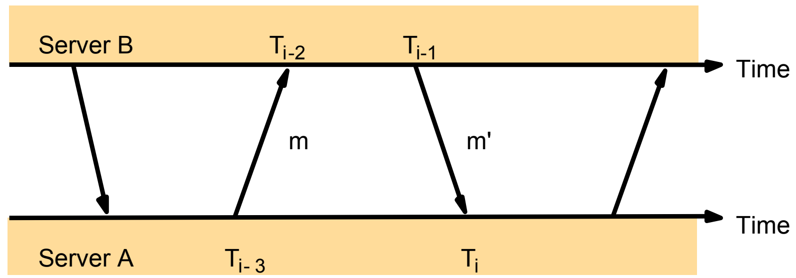
- Three modes of operation
  1. Multicast mode: for LAN
     - server(s) multicast time
     - others set to it, assuming very small delay

     Efficient, but not great accuracy
  2. Procedure-call mode
     - similar to Cristian's, server accepts time queries from other computers

     Useful where multicast not supported, or higher accuracy required

     Lots of messages, though
  3. Symmetric mode
     - Pair of servers exchange timing data
     - Meant for higher levels (lower strata) for highest accuracies
- UDP used for all modes
- Even if messages lost, the timestamps in messages which arrive are valid…
- Each message keeps timestamps of *a number of recent events*

Figure 10.4: Messages exchanged between a pair of NTP peers



- Recent messages between processes are tracked
- For each pair of messages sent, calculated *offset* o_i and *delay* d_i
- NTP servers apply data filtering to most recent 8 < o_i, d_i > values: *filter dispersion*
- More details in the book…
- Not only used for setting clock values, but may choose another server to synch with (another kind of reconfiguration)!

Outline

- Clocks, events, and process states (10.2)
- Synchronizing physical clocks (10.3)
- **Logical time and logical clocks (10.4)**
- Global states (10.5)

Logical Time

- Time in Distributed Systems
  - Computers can only be synchronized by network messages, but the latency can vary
  - We can not synchronize enough to be able to, in general, tell the ordering of two arbitrary events at different computers.
  - We can, however, establish an ordering on some of the events, and this can be used in many situations.
- Logical Time
  - Builds up a notion of what we can reason about w.r.t. the order of events
  - Defines the "Happened-before" relation
  - Source: Lamport, Leslie. "Time, Clocks and the Ordering of Event in a Distributed System", *Communications of the ACM*, Vol. 21, July 1978, pp. 558-565.
    - One of the seminal works in distributed systems…

Happened-Before Relation

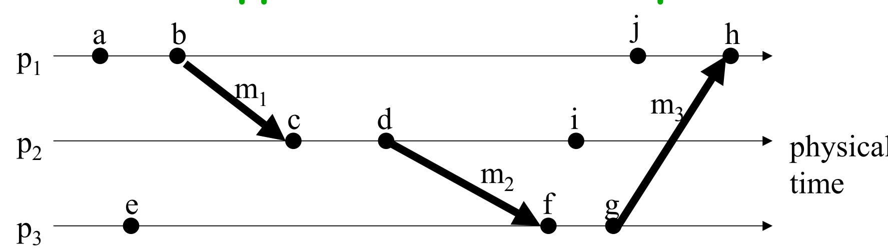
- Happened-Before relation, →, based on observations:
  1. If two events occur in the same process, then they occurred in the order in which that process observes them.
  2. The receipt of a message happens after its being sent.
  3. "Happened-before" is transitive
- Corresponding Rules for events *x*, *y*, *z*, process *p*, and message *m*

  HB1: $x \rightarrow_p y$, then $x \rightarrow y$

  HB2: send(m) → recv(m)

  Transitivity: x → y and y → z, then x → z
- Concurrency: If a ~→ b and b ~→ a, then a||b ("a is concurrent with b")'
- Note: if *x* → *y* ("*x* happened before *y*") then *y* ← *x* ("*y* happened after *x*"), notationally

Happened-Before Example

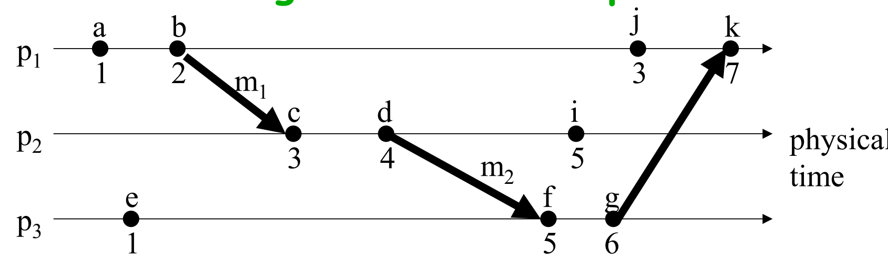


- Example table of →, ←, ||
- Limitations of Happened-Before
  - Covert channels
  - Too pessimistic: some things *a*→*b* did not have *a* causing *b*!
- Happened-before also called
  - Causal ordering
  - Potential causality
  - Lamport ordering
  - (irreflexive) partial ordering

Logical Clocks

- How to implement “Happened Before”??
- Logical Clock, a monotonically increasing counter.
- Let
  - Each process *p* keeps its own logical clock, $C_{p,}$ which it uses to timestamp events
  - C_p (*a*) is the logical time at process *p* at which event *a* occurred
  - *C(a)* is the logical time at which event *a* occurred at the process it occurred at
- Processes keep their own logical clocks, initialized to 0. Updated by rules:
  - LC1: Before each event occurs, increment C_p
  - LC2:
    - When a process *p* sends a message *m*, it piggybacks on *m* value t= C_p
    - When process *q* receives <*m,t*>, *q* computes $C_q = \max(C_q, t) + 1$ then timestamps *m*

Logical Clock Example



- Note if *a* → *b* then *LC(a)* < *LC(b)*
- However, *LC(a)* < *LC(b)* does <u>not</u> imply *a* → *b*
  - Above, C(e) < C(b) yet *b* || *e*
  - Also note that concurrency is not transitive: *a*||*e* and *e*||*b* yet *a*→*b*

Partial Orderings

- Logical clocks impose a <u>partial ordering</u> on set of all events. “A partial ordering over a set S is a function PO such that, for all s, t in S, either
- 1. PO(s) < PO(t)
- 2. PO(s) > PO(t)
- 3. PO(s) == PO(t)”

  (Note that PO is defined for all members of S.)”
- Examples:
- S == students in the class
- PO1 == number of coins in the student’s pockets
- PO2 == student’s grade on project #2
- PO3 == number of teeth in student’s mouth

Total Orderings and Logical Clocks

- Total order is more strict and sometimes more useful. “A total ordering over a set S is a function TO such that, for all s, t in S, either
- 1. PO(s) < PO(t)
- 2. PO(s) > PO(t)
- 3. s == t

  (i.e., it is defined for all members of s, and the function’s value is unique for all elements of the set)”
- How to create a total ordering out of the LC’s partial one???
  - Just break “ties” among logical clocks by using any total ordering over the processes involved
  - e.g., looking at host ID (unique, virtually always comparable))
- (If time, do example from 565 exam, or at end of lecture)

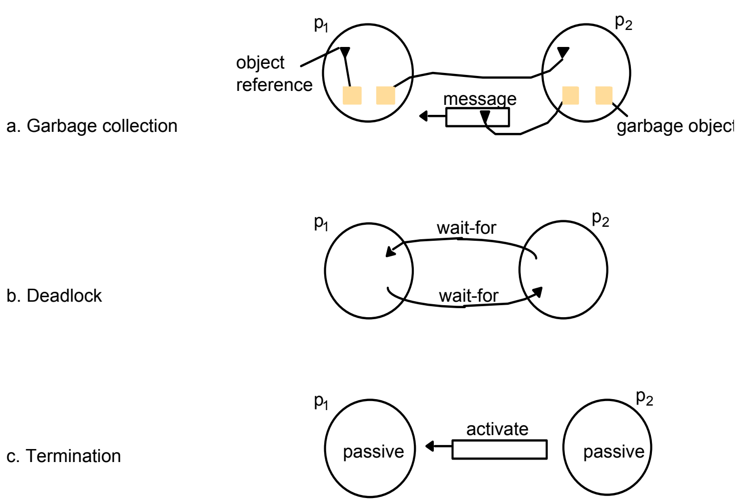
Outline

- Clocks, events, and process states (10.2)
- Synchronizing physical clocks (10.3)
- Logical time and logical clocks (10.4)
- **Global states (10.5)**

Global States

- General problem: is a given property true of a DS as it executes?
  - Huge number of applications to these general concepts
- Example 1: distributed garbage collection (Fig 10.8a)
  - Garbage object: one that has no references to it anywhere in a DS
  - Property to prove: no references anywhere to a given object
  - Must verify that there are no references to it *anywhere* in the DS
  - But could be one in transit in a message…
- Example 2: distributed deadlock (Fig 10.8b)
  - Detecting that a DS is deadlocked and cannot make progress (without help)
  - Property to prove: a cycle in the “waits for” relationship exists
- Example 3: distributed termination detection (Fig 10.8c)
  - Detecting that a distributed algorithm has terminated
  - Active process: one still doing work
  - Passive process: one not active, but that will respond to a message
  - Property to prove: all processes are passive, and no messages are in transit
- Example 4: distributed debugging (Sec 10.6)
  - Property to prove: value for a given variable everywhere in a system is x
  - Another property: each p_i has variable x_i, and constraint $|x_i - x_j| < \delta$, $\forall i$ holds

Figure 10.8 Detecting global properties



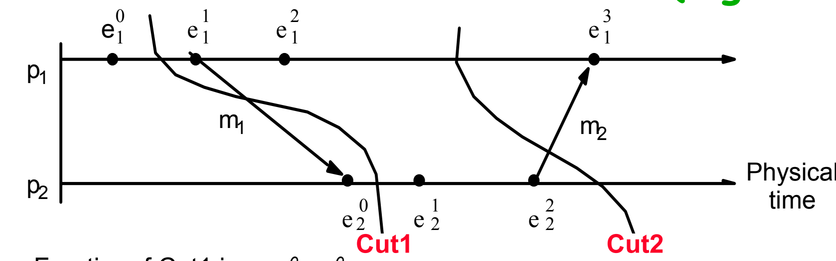
Notations for Global States

- Context
  - We might be able to observe succession of states in an individual process
  - But how can we construct a valid global state?
  - Problem: lack of global time
  - Q: how might you construct a valid global state with perfect global clocks?
- Consistent global states can still be done with imperfect clocks, sort of…
- Notation and definitions
  - $history(p_i) = h_i = <e_i^0, e_i^1, e_i^2, \ldots>$
  - Prefix of a process’s history: $h_i^k = <e_i^0, e_i^1, \ldots, e_i^k>$
  - State of a process: s_i^k c is the state of p_i right before e_i^k occurs; s_i^0 is init state
- Notational problem: how to deal with messages in transit from p_i to p_j?
  - Record state of the (logical) channel from p_i to p_j
  - How: check the two processes events
    - recall message sends and receives are events (that plus modifying state)
  - If p_i has “send m” as e_i^m and p_j has “receive m” as e_j^n then is in the channel

Notations for Global States (cont.)

- Global history: union of the individual … histories: H = $h_0 \cup h_1 \cup \ldots \cup h_{N-1}$
- Forming a global states
  - Mathematically, we could take any set of states of the individual processes to form a global state S = $\{s_1, s_2, \ldots s_N\}$
  - But what states are meaningful: what *could* have happened at the same time? ….
  - Recall a process state corresponds to the initial prefix of its history
  - So a global state corresponds to initial prefixes of the individual processes' histories
- A cut of the system's execution: a subset of its global history that is a union of prefixes of process histories: C = $h_1^{C1} \cup h_2^{C2} \cup \ldots \cup h_N^{CN}$
  - Q: What is state s_i of p_i in global state S corresponding to the cut C?
  - A: The state of p_i right after the last event processed by p_i in C: e_i^{Ci}
  - Frontier of cut C: set of events $\{e_1^{C1}, e_2^{C2} \ldots e_N^{CN}\}$

Consistent and Inconsistent Cuts (Fig 10.9)



- Frontier of Cut1 is < e_1^0, e_2^0 >
- Frontier of Cut2 is < e_1^2, e_2^2 >
- Cut1 is inconsistent:
  - Includes the receipt of message m_1 : e_2^0
  - Excludes the receipt of message m_1 : e_1^1
  - I.e., cut reflects "effect" but not "cause" …. could not have happened!
  - We can tell this by examining the → "happens before" relation
- Cut2 is consistent:
  - Both sending and receiving of m_1 is included
  - Sending of m_1 is included, but not its receipt
    - Consistent with its actual execution: message delivery took nonzero time

Consistent and Inconsistent Cuts (cont.)

- A cut C is consistent if, for each event it contains, it also contains all the events that happened-before that event:
  - $\forall$ events $e \in$ C: $f \rightarrow e \Rightarrow f \in$ C
- Consistent global state: one that corresponds to a consistent cut
- A run: a total ordering of all the events in a global history that is consistent with each local history's ordering, $\rightarrow_i$ (i =1,2,…,N)
- A linearization run (a.k.a. consistent run): an ordering of the events in a global history H that is consistent with this happened-before relation on H
- Questions:
  - Do all runs pass through any or all consistent global states?
  - Do all linearization runs pass through any or all consistent global states?
- State S' is reachable from state S if there is a linearization that passes through S and then S'
  - Does not guarantee it *will* be reached, only its possible

Global state predicates, stability, safety, and liveness

- Global state predicate: a function that maps from the set of global states to true or false
  - Detecting deadlock or termination amounts to evaluating a predicate
- Stable global predicate: one that, if it becomes true, stays true
  - Examples: object is garbage, deadlock, termination
  - Unstable example: anything with distributed debugging
- Safety w.r.t. α: (undesirable property) α evaluates to **false** for all states S reachable from S_0
- Liveness w.r.t. β: for any linearization L starting at S_0, (desirable property) β evaluates to **true** for some state S_L reachable from S_0
- Safety and liveness are categories of properties discussed a lot in practice
  - Safety properties of form "nothing bad ever happens"
  - Liveness properties of form "something good eventually happens"
- Note: skipping "snapshot" algorithm of Sec 10.5.3, and its not testable…