Distributed Systems
[Fall 2013]

Lec 7: Time and Synchronization

Slide acks: Dave Andersen, Randy Bryant

(http://www.cs.cmu.edu/~dga/15-440/F11/lectures/09-time+synch.pdf)

Any Questions for HW 2?

- Deadline is tomorrow before midnight!

- Poll: Where are you on HW 2?
  a) Largely done with both parts (maybe some testing left)
  b) Largely done with first part
  c) Done with neither part

Today’s outline

- Distributed time
  - A baseball example
- Synchronizing real clocks
  - Cristian’s algorithm
  - The Berkeley Algorithm
  - Network Time Protocol (NTP)
- Logical time
  - Lamport logical clocks
  - Vector clocks

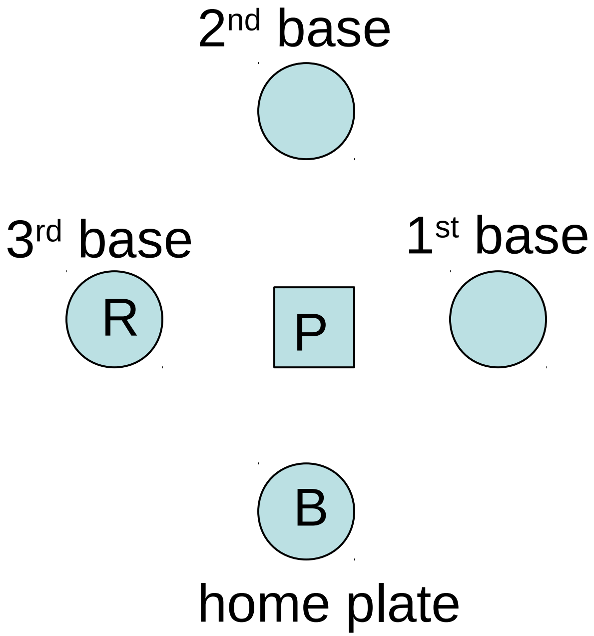
Distributed Time

- The notion of time is well-defined (and measurable) at each single location
- But the relationship between time at different locations is unclear
  - Can minimize discrepancies, but never eliminate them
- Examples:
  - If two file servers get different update requests to same file, what should be the order of those requests?
  - Did the runner get to home base before the pitcher was eliminated?

A Baseball Example

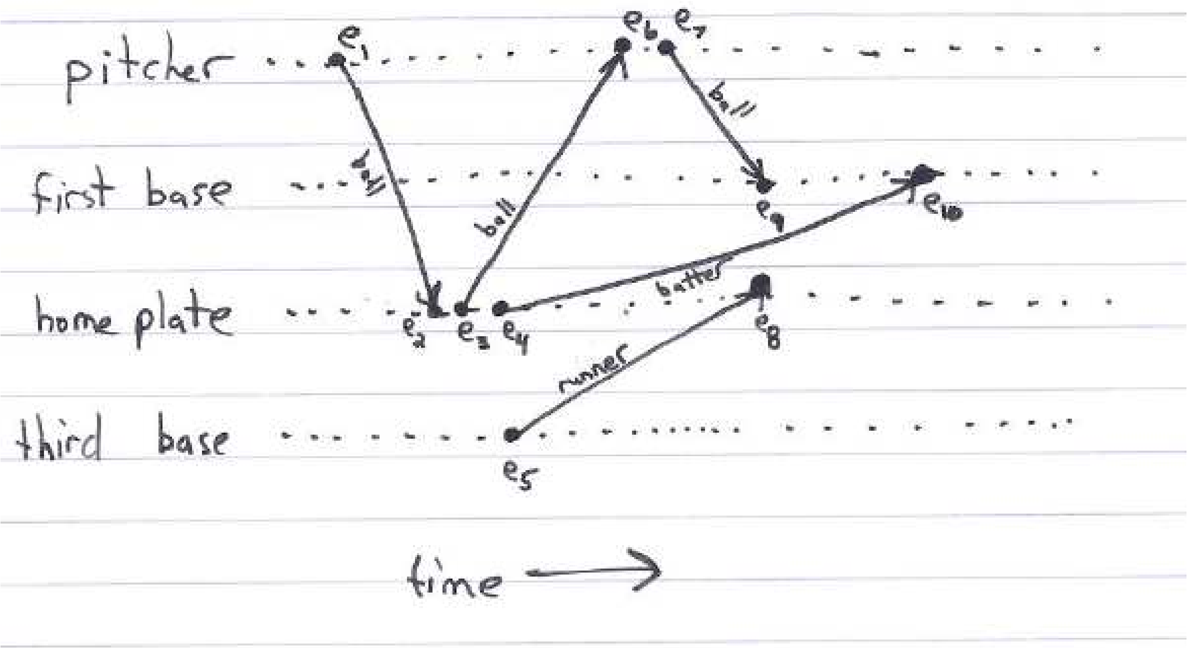
- Four locations: pitcher’s mound (P), home plate, first base, and third base

- Ten events:
  - e_1: pitcher (P) throws ball toward home
  - e_2: ball arrives at home
  - e_3: batter (B) hits ball toward pitcher
  - e_4: batter runs toward first base
  - e_5: runner runs toward home
  - e_6: ball arrives at pitcher
  - e_7: pitcher throws ball toward first base
  - e_8: runner arrives at home
  - e_9: ball arrives at first base
  - e_{10}: batter arrives at first base



A Baseball Example

- Pitcher knows e_1 happens before e_6, which happens before e_7
- Home plate umpire knows e_2 is before e_3, which is before e_4, which is before e_8, …
- Relationship between e_8 and e_9 is unclear



Ways to Synchronize

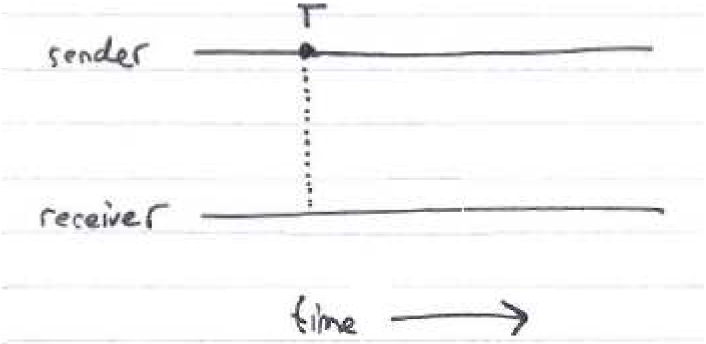
- Send message from first base to home when ball arrives?
  - Or both home and first base send messages to a central timekeeper when runner/ball arrives
  - But: How long does this message take to arrive?

- Synchronize clocks before the game?
  - Clocks drift
    - One-in-a-million drifting => 1 second in 11 days

- Synchronize clocks continuously during the game?
  - E.g.: NTP, GPS, etc.
  - But how do these work?

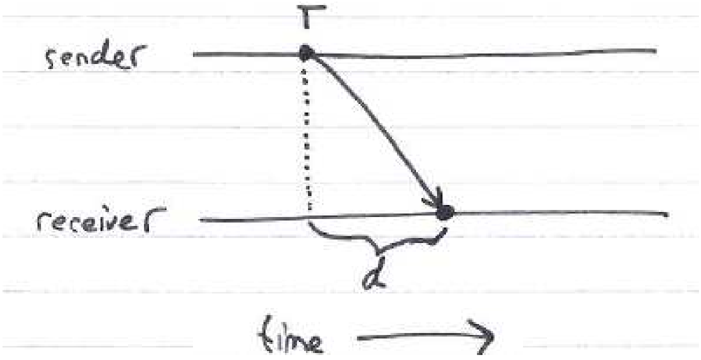
Real-Clock Synchronization

- Suppose I want to synchronize the clocks on two machines (M1 and M2)
- One solution:
  - M1 (sender) sends its own time T in message to M2
  - M2 (receiver) sets its time according to the message
  - But what time should M2 set?



Perfect Networks

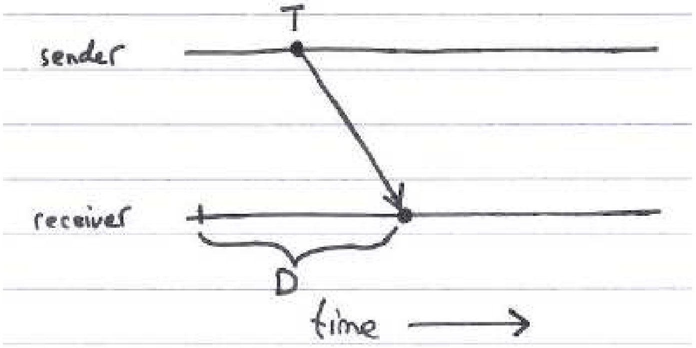
- Messages always arrive, with propagation delay exactly *d*



- Sender sends time *T* in a message
- Receiver sets clock to *T+d*
  - Synchronization is exact

Synchronous Networks

- Messages always arrive, with propagation delay *at most D*



- Sender sends time *T* in a message
- Receiver sets clock to *T* + *D/2*
  - Synchronization error is at most *D/2*

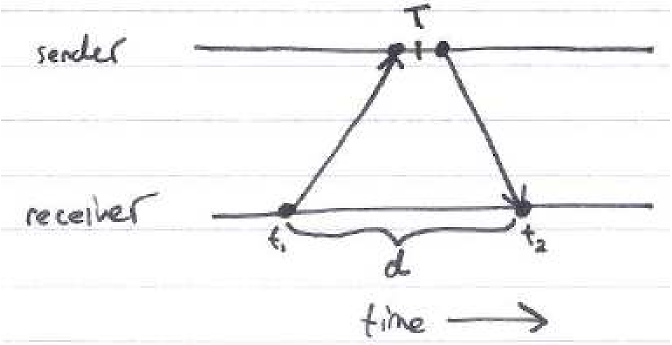
Synchronization in the Real World

- Real networks are asynchronous
  - Propagation delays are arbitrary

- Real networks are unreliable
  - Messages don’t always arrive

Cristian's Algorithm

- Request time, get reply
  - Measure actual round-trip time *d*



- Sender's time was *T* between t_1 and t_2
- Receiver sets time to $T + d/2$
  - Synchronization error is at most *d/2*
- Can retry until we get a relatively small *d*

The Berkeley Algorithm

- In Cristian’s algorithm, how does sender know the “right” time?
- Master uses Cristian’s algorithm to gather time from many clients
  - Computes average time
  - Discards outliers
- Sends time adjustments back to all clients

The Network Time Protocol (NTP)

- Uses a hierarchy of time servers
  - Class 1 servers have accurate (and expensive) clocks
    - connected directly to atomic clocks or GPS receivers
  - Class 2 servers get time from Class 1 and Class 2 servers
  - Class 3 servers get time from any server
  - Client machines (e.g., your smartphones, laptops, desktops, or server machines) synchronize w/ time servers
- Synchronization similar to Cristian’s alg.
- Accuracy: Local ~1ms, Global ~10ms

Real Synchronization Is Imperfect

- Clocks are never *exactly* synchronized
- Often inadequate for distributed systems
  - Might need totally-ordered events

- But, more often than not, distributed systems do not need **real** time, but **some** time that every machine in a protocol agrees upon!
  - E.g.: suppose file servers S1 and S2 receive two update requests, W1 and W2, for file F
  - They need to apply W1 and W2 in the same order, but they may not really care precisely which order…

Logical Time

- Capture just the “happens before” relationship between events
  - Discard the infinitesimal granularity of time
  - Corresponds roughly to causality

- Time at each process is well-defined
  - Definition ($\rightarrow_i$): We say $e \rightarrow_i e'$ if e happens before e' at process i

Global Logical Time

- Definition (→): We define e → e’ using the following rules:
  - Local ordering: $e \rightarrow e'$ if $e \rightarrow_i e'$ for any process i
  - Messages: send(m) → receive(m) for any message m
  - Transitivity: $e \rightarrow e''$ if $e \rightarrow e'$ and $e' \rightarrow e''$

- We say e “happens before” e' if $e \rightarrow e'$

Concurrency

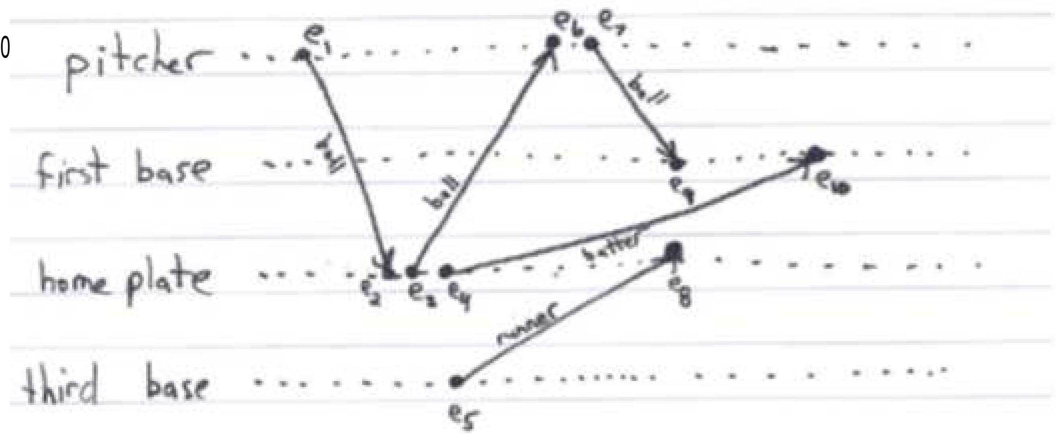
- → is only a partial-order
  - Some events are unrelated

- Definition (concurrency): We say *e* is concurrent with *e'* (written $e \| e'$) if neither *e* → *e'* nor *e'* → *e*

Back to Baseball

Events:

e_1: pitcher (P) throws ball toward home
e_2: ball arrives at home
e_3: batter (B) hits ball toward pitcher
e_4: batter runs toward first base
e_5: runner runs toward home
e_6: ball arrives at pitcher
e_7: pitcher throws ball toward first base
e_8: runner arrives at home
e_9: ball arrives at first base
e_{10}



The Baseball Example Revisited

- $e_1 \rightarrow e_2$
  - by the message rule

- $e_1 \rightarrow e_{10}$, because
  - $e_1 \rightarrow e_2$, by the message rule
  - $e_2 \rightarrow e_4$, by local ordering at home plate
  - $e_4 \rightarrow e_{10}$, by the message rule
  - Repeated transitivity of the above relations

- $e_8 \parallel e_9$, because
  - No application of the → rules yields either $e_8 \rightarrow e_9$ or $e_9 \rightarrow e_8$

Lamport Logical Clocks

- Lamport clock L assigns logical timestamps to events consistent with "happens before" ordering
  - If $e \rightarrow e'$, then $L(e) < L(e')$

- But not the converse
  - $L(e) < L(e')$ does not imply $e \rightarrow e'$

- Similar rules for concurrency
  - $L(e) = L(e')$ implies $e \parallel e'$ (for distinct e,e')
  - $e \parallel e'$ does not imply $L(e) = L(e')$

- I.e., Lamport clocks arbitrarily order some concurrent events

Lamport's Algorithm

- Each process *i* keeps a local clock, L_i

- Three rules:
  1. At process *i*, increment L_i before each event
  2. To send a message *m* at process *i*, apply rule 1 and then include the current local time in the message: i.e., $send(m,L_i)$
  3. When receiving a message (m,t) at process *j*, set $L_j = max(L_j,t)$ and then apply rule 1 before time-stamping the receive event

- The global time $L(e)$ of an event *e* is just its local time
  - For an event *e* at process *i*, $L(e) = L_i(e)$

Lamport on the baseball example

- Initializing each local clock to 0, we get
    - $L(e_1) = 1$ (pitcher throws ball to home)
    - $L(e_2) = 2$ (ball arrives at home)
    - $L(e_3) = 3$ (batter hits ball to pitcher)
    - $L(e_4) = 4$ (batter runs to first base)
    - $L(e_5) = 1$ (runner runs to home)
    - $L(e_6) = 4$ (ball arrives at pitcher)
    - $L(e_7) = 5$ (pitcher throws ball to first base)
    - $L(e_8) = 5$ (runner arrives at home)
    - $L(e_9) = 6$ (ball arrives at first base)
    - $L(e_{10}) = 7$ (batter arrives at first base)
- For our example, Lamport’s algorithm says that the run scores!

Total-order Lamport Clocks

- Many systems require a total-ordering of events, not a partial-ordering
- Use Lamport's algorithm, but break ties using the process ID
  - $L(e) = M * L_i(e) + i$
    - M = maximum number of processes

Important Points

- Physical Clocks
  - Can keep closely synchronized, but never perfect

- Logical Clocks
  - Encode causality relationship
  - Lamport clocks provide only one-way encoding