Time and synchronization

(“There’s never enough time…”)

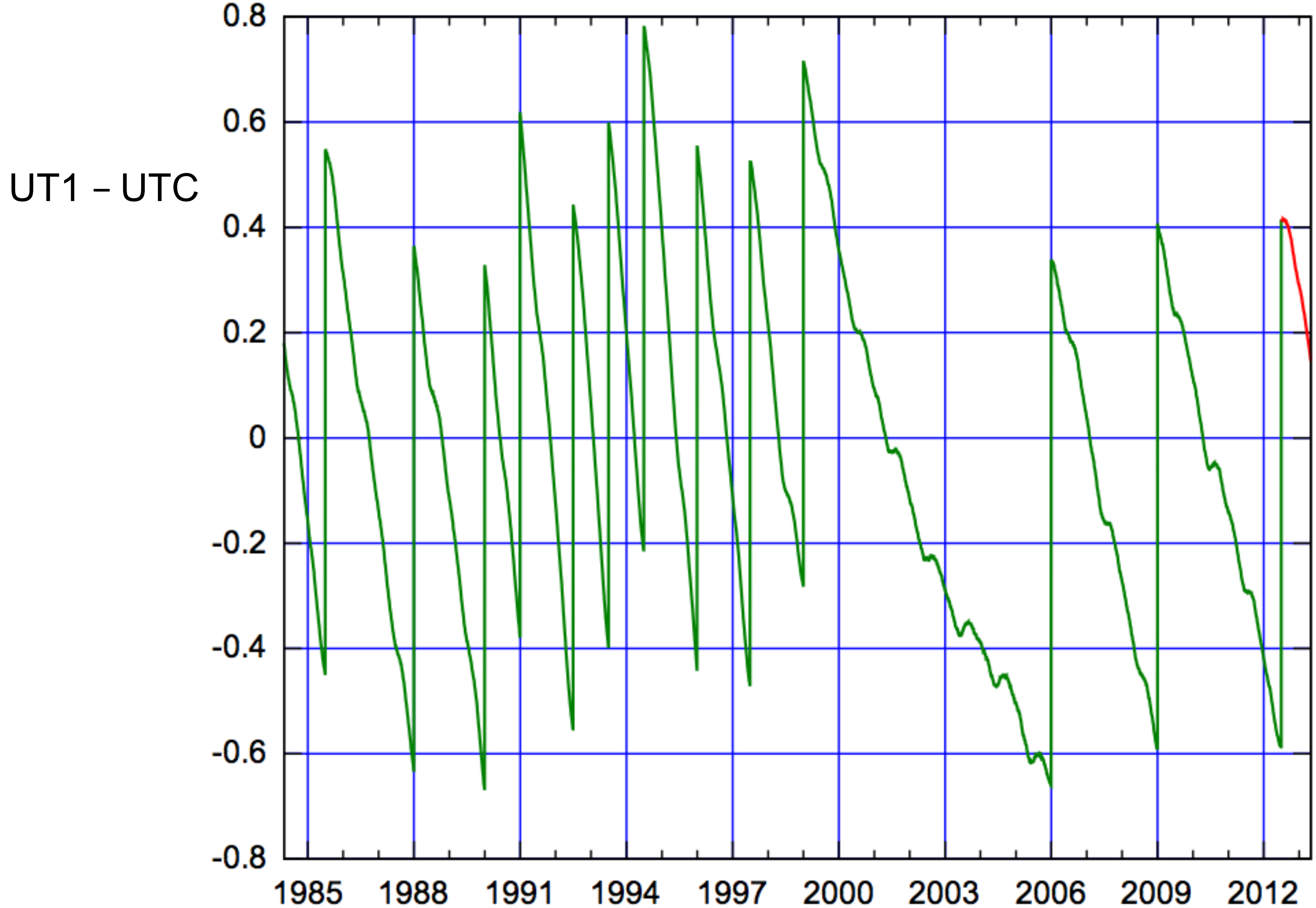
Why Global Timing?

- Suppose there were a globally consistent time standard
- Would be handy
  - Who got last seat on airplane?
  - Who submitted final auction bid before deadline?
  - Did defense move before snap?

Time Standards

- UT1
  - Based on astronomical observations
  - “Greenwich Mean Time”
- TAI
  - Started Jan 1, 1958
  - Each second is 9,192,631,770 cycles of radiation emitted by Cesium atom
  - Has diverged from UT1 due to slowing of earth’s rotation
- UTC
  - TAI + leap seconds to be within 800ms of UT1
  - Currently 35
  - Most recent: June 30, 2012

Comparing Time Standards



Clocks

- Piezoelectric effect:
  - Squeeze a quartz crystal: generates electric field
  - Apply electric field: crystal bends
- Quartz crystal clock:
  - Resonation like a tuning fork
  - Accurate to parts per million
  - Gain/lose ½ second per day

Challenges

- Two clocks do not agree perfectly
- **Skew:** The time difference between two clocks
- Quartz oscillators vibrate at different rates
- **Drift:** The difference in rates of two clocks
- If we had two perfect clocks:
  - Skew = 0
  - Drift = 0

When we detect a clock has a skew

- Eg: it is 5 seconds behind
- Or 5 seconds ahead

- What can we do?

When we detect a clock has a skew

- e.g. it is 5 seconds behind
  - We can advance it 5 seconds to correct
  - Might skip over event scheduled in-between
- Or 5 seconds ahead
  - Pushing back 5 seconds is a bad idea
    - Message was received before it was sent
    - Document closed before it was saved etc..

  - We want **monotonicity**: time always increases
  - We want **continuity**: time doesn't make jumps

When we detect a clock has a skew

- e.g. it is behind
  - Run it faster until it catches up
- It is ahead
  - Run it slower until it catches up

- This does not guarantee correct clock in future
  - Need to check and adjust periodically

Distributed time

- Premise
  - 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
- Reality
  - Stationary GPS receivers can get global time with < 1μs error
  - Few systems designed to use this

A football example

- Five locations: Goal Keeper K1, Player P1, Player P2, Player P3 and Goal Keeper K2
- Ten events:
  - e_1: keeper K1 kicks the ball
  - e_2: ball arrives to Player P1
  - e_3: P1 kicks the ball to Player P2
  - e_4: P2 runs with the ball
  - e_5: P2 kicks the ball towards Player P3
  - e_6: P3 kicks the ball towards other keeper K2
  - e_7: The ball arrives near the keeper
  - e_8: But another Player P4 touches the ball
  - e_9: Ball also touches the keeper K2
  - e_{10}: Ball went inside the goal

A football example

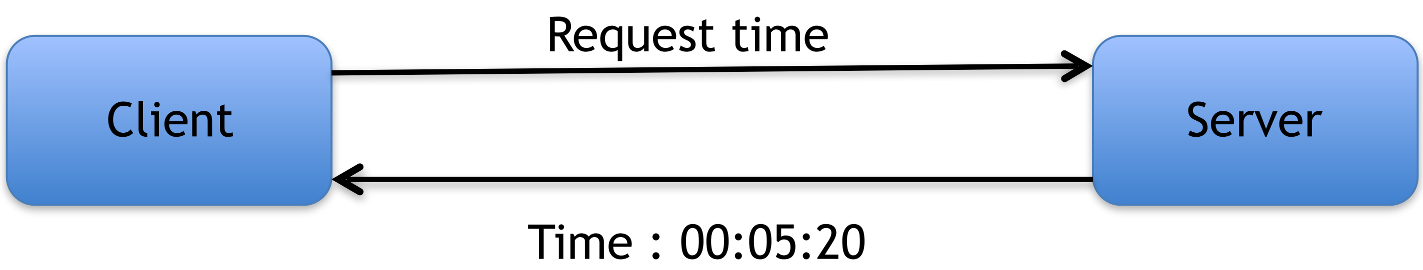
- Players and umpire knows e_1 happens before e_6, which happens before e_7
- Umpire knows e_2 is before e_3, which is before e_4, which is before e_8, …
- Relationship between e_8 and e_9 is unclear

Ways to synchronize

- Send message from a player to another?
  - Or to a central timekeeper
  - How long does this message take to arrive?
- Synchronize clocks before the game?
  - Clocks drift
    - million to one => 1 second in 11 days
- Synchronize continuously during the game?
  - GPS, pulsars, etc

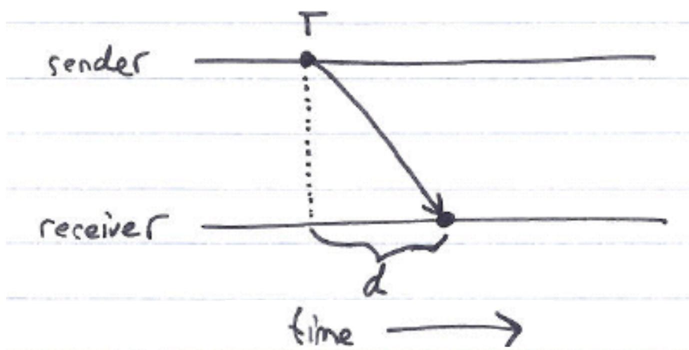
How clocks synchronise

- Obtain time from time server:



Perfect networks

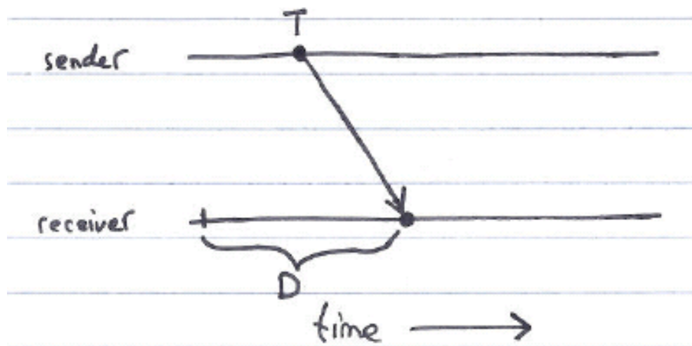
- Messages always arrive, with propagation delay exactly *d*



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

Synchronous networks

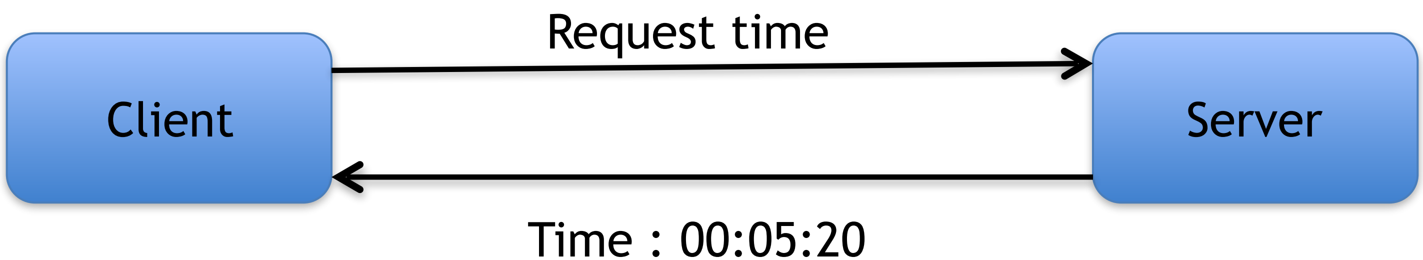
- Messages always arrive, with propagation delay *at most D*



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

How clocks synchronise

- Obtain time from time server:



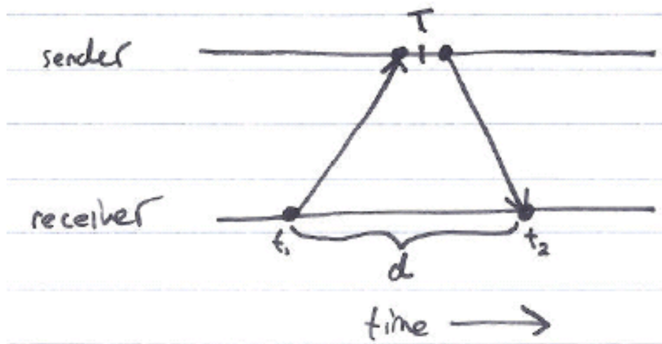
- Time is inaccurate
  – Delays in message transmission
  – Delays due to processing time
  – Server’s time may be inaccurate

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

- Master uses Cristian's algorithm to get time from many clients
  - Computes average time
  - Can discard outliers
- Sends time adjustments back to all clients

The Network Time Protocol (NTP)

- Uses a hierarchy of time servers
  - Class 1 servers have highly-accurate clocks
    - connected directly to atomic clocks, etc.
  - Class 2 servers get time from only Class 1 and Class 2 servers
  - Class 3 servers get time from any server
- Synchronization similar to Cristian's alg.
  - Modified to use multiple one-way messages instead of immediate round-trip
- Accuracy: Local ~1ms, Global ~10ms

Real synchronization is imperfect

- Clocks never exactly synchronized
- Often inadequate for distributed systems
  - might need totally-ordered events
  - might need millionth-of-a-second precision

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 ($\rightarrow$): We define e $\rightarrow$ e' using the following rules:
  - Local ordering: $e \rightarrow e'$ if $e \rightarrow_i e'$ for any process i
  - Messages: send(m) $\rightarrow$ 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

- $\rightarrow$ is only a partial-order
  – Some events are unrelated
- Definition (concurrency): We say e is concurrent with e' (written $e \| e'$) if neither $e \rightarrow e'$ nor $e' \rightarrow e$

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 $\rightarrow$ rules yields either $e_8 \rightarrow e_9$ or $e_9 \rightarrow e_8$

Lamport logical clocks

- Lamport clock *L* orders events consistent with logical "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 $e \| e'$ (for distinct e,e')
  - $e \| e'$ 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. To receive 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

Vector Clocks

- Goal
  - Want ordering that matches causality
  - $V(e) < V(e')$ if and only if $e \rightarrow e'$
- Method
  - Label each event by vector $V(e)$ $[c_1, c_2 \ldots, c_n]$
    - c_i = # events in process i that causally precede e

Vector Clock Algorithm

- Initially, all vectors [0,0,…,0]
- For event on process i, increment own c_i
- Label message sent with local vector
- When process j receives message with vector $[d_1, d_2, \ldots, d_n]$:
  - Set local each local entry k to $\max(c_k, d_k)$
  - Increment value of c_j

Vector clocks on the baseball example

Event	Vector	Action
e_1	[1,0,0,0]	pitcher throws ball to home
e_2	[1,0,1,0]	ball arrives at home
e_3	[1,0,2,0]	batter hits ball to pitcher
e_4	[1,0,3,0]	batter runs to first base)
e_5	[0,0,0,1]	runner runs to home
e_6	[2,0,2,0]	ball arrives at pitcher
e_7	[3,0,2,0]	pitcher throws ball to 1st base
e_8	[1,0,4,1]	runner arrives at home
e_9	[3,1,2,0]	ball arrives at first base
e_{10}	[3,2,3,0]	batter arrives at first base

- Vector: [p,f,h,t]

Important Points

- Physical Clocks
  - Can keep closely synchronized, but never perfect
- Logical Clocks
  - Encode causality relationship
  - Lamport clocks provide only one-way encoding
  - Vector clocks provide exact causality information