Lecture 10:
Clocks and Time
Overview

• **Time service**
  – requirements and problems
  – sources of time

• **Clock synchronisation algorithms**
  – clock skew & drift
  – Cristian algorithm
  – Berkeley algorithm
  – Network Time Protocol

• **Logical clocks**
  – Lamport’s timestamps
Time service

• Why needed?
  – to measure delays between distributed components
  – to synchronise streams, e.g. sound and video
  – to establish event ordering
    • causal ordering (did A happen before B?)
    • concurrent/overlapping execution (no causal relationship)
  – for accurate timestamps to identify/authenticate
    • business transactions
    • serializability in distributed databases
    • security protocols
Clocks

• **Internal hardware clock**
  – built-in electronic device
  – counts *oscillations* occurring in a quartz crystal at a definite frequency
  – store the result in a *counter register*
  – *interrupt* generated at regular intervals
  – interrupt handler reads the counter register, scales it to convert to time units (seconds, nanoseconds) and updates *software clock*
    • e.g. seconds elapsed since 1/01/1970
Problems with internal clocks

• Frequency of oscillations
  – varies with temperature
  – different rate on different computers

• Accuracy
  – typically 1 sec in 11.6 days

• Centralised time service?
  – impractical due to variable message delays
Clock skew and drift

- **Clock skew**
  - difference between the readings of two clocks

- **Clock drift**
  - difference in reading between a clock and a nominal perfect reference clock *per unit of time of the reference clock*
    - typically $10^{-6}$ seconds/second = 1 sec in 11.6 days
Sources of time

• **Universal Coordinated Time (UTC, from French)**
  – based on **atomic** time but leap seconds inserted to keep in phase with astronomical time (Earth’s orbit)
  – UTC signals broadcast every second from **radio** and **satellite** stations
  • land station accuracy 0.1-10ms due to atmospheric conditions

• **Global Positioning System (GPS)**
  – broadcasts UTC

• **Receivers for UTC and GPS**
  – available commercially
  – used to synchronise local clocks
Clock synchronisation

- **External**: synchronise with authoritative source of time
  - the absolute value of difference *between the clock and the source* is bounded above by $D$ at every point in the synchronisation interval
  - time *accurate* to within $D$

- **Internal**: synchronise clocks with each other
  - the absolute value of difference *between the clocks* is bounded above by $D$ at every point in the synchronisation interval
  - clocks *agree* to within $D$ (not necessarily accurate time)
Clock compensation

• Assume 2 clocks can each drift at rate $R$ msecs/sec
  – maximum difference $2R$ msecs/sec
  – must resynchronise every $D/2R$ to agree within $D$

• Clock correction
  – get UTC and correct software clock

• Problems!
  – what happens if local clock is 5 secs fast and it is set right?
  – timestamped versions of files get confused
  – time must never run backwards!
  – better to scale the value of internal clock in software without changing the clock rate
Synchronisation methods

• Synchronous systems
  – simpler, relies on known time bounds on system actions

• Asynchronous systems
  – intranets
    • Cristian’s algorithm
    • Berkeley algorithm
  – Internet
    • The Network Time Protocol
Synchronous systems case

• **Internal synchronisation** between two processes
  – know *bounds* MIN, MAX on message delay
  – also on clock drift, execution rate

• **Assume One** sends message to Two with time \( t \)
  – **Two** can set its clock to \( t + \frac{(MAX+MIN)}{2} \) (estimate of time taken to send message)
  – then the skew is at most \( (MAX-MIN)/2 \)
  – why not \( t + MIN \) or \( t + MAX \)?
    • maximum skew is larger, could be \( MAX-MIN \)
Cristian’s algorithm

- Estimate message propagation time by \( p = \frac{(T_1 - T_0 - h)}{2} \) (=half of round-trip of request-reply)
- Set clock to UTC + \( p \)
- Make multiple requests, at spaced out intervals, measure \( T_1 - T_0 \)
  - but discard any that are over a threshold (could be congestion)
  - or take minimum values as the most accurate

Time Server with UTC receiver gives accurate current time
Cristian’s algorithm

• Probabilistic behaviour
  – achieves synchronisation only if round-trip short compared to required accuracy
  – high accuracy only for message transmission time close to minimum

• Problems
  – single point of failure and bottleneck
  – could multicast to a group of servers, each with UTC
  – an impostor or faulty server can wreak havoc
    • use authentication
    • agreement protocol for $N > 3f$ clocks, $f$ number of faulty clocks
The Berkeley algorithm

- Choose master co-ordinator which periodically polls slaves
- Master estimates slaves’ local time based on round-trip
- Calculates average time of all, ignoring readings with exceptionally large propagation delay or clocks out of synch
- Sends message to each slave indicating clock adjustment

Synchronisation feasible to within 20-25 msec for 15 computers, with drift rate of $2 \times 10^{-5}$ and max round trip propagation time of 10 msec.
The Berkeley algorithm

• **Accuracy**
  – depends on the round-trip time

• **Fault-tolerant average:**
  – eliminates readings of faulty clocks - **probabilistically**
  – average over the **subset** of clocks that differ by **up to** a specified amount

• **What if master fails?**
  – elect another leader
Network Time Protocol (NTP)

- **Multiple** time servers across the Internet
- **Primary** servers: directly connected to UTC receivers
- **Secondary** servers: synchronise with primaries
- **Tertiary** servers: synchronise with secondary, etc
- Scales up to large numbers of servers and clients

Copes with failures of servers – e.g. if primary’s UTC source fails it becomes a secondary, or if a secondary cannot reach a primary it finds another one.

Authentication used to check that time comes from trusted sources
NTP Synchronisation Modes

- **Multicast**
  - one or more servers periodically multicast to other servers on high speed LAN
  - they set clocks assuming small delay

- **Procedure Call Mode**
  - similar to Cristian’s algorithm: client requests time from a few other servers
  - used for higher accuracy or where no multicast

- **Symmetric protocol**
  - used by master servers on LANs and layers closest to primaries
  - highest accuracy, based on pairwise synchronisation
NTP Symmetric Protocol

- $t$ = transmission delay (e.g. 5ms)
- $o$ = clock offset of B relative to A (e.g. 3ms)
- Record local times $T_1 = 10$, $T_2 = 18$, $T_3 = 20$, $T_4 = 22$

Let $a = T_2 - T_1 = t + o$, $b = T_4 - T_3 = t' - o$, and assume $t \approx t'$

Round trip delay = $t + t' = a + b = (T_2 - T_1) + (T_4 - T_3) = 10$

Calculate estimate of clock offset $o = (a - b)/2 = 3$
NTP Symmetric Protocol

- T4 = current message receive time determined at receiver
- Every message contains
  - T3 = current message send time
  - T2 = previous receive message receive time
  - T1 = previous receive message send time
- Data filtering (obtain average values of clock offset from values of t corresponding to minimum t)
- Peer selection (exchange messages with several peers favouring those closer to primaries)
- How good is it? 20-30 primaries and 2000 secondaries can synchronise to within 30 ms
Logical time

- For many purposes it is sufficient to agree on the same time (e.g. internal consistency) which need not be UTC time
- Can deduce causal event ordering $a \rightarrow b$ (a occurs before b)
- Logical time denotes causal relationships
- but the $\rightarrow$ relationship may not reflect real causality, only accidental
Event ordering

Define $a \rightarrow b$ (a occurs before b) if

- $a$ and $b$ are events in the same process and $a$ occurs before $b$, or
- $a$ is the event of message sent from process A and $B$ is the event of message receipt by process B

If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

$\rightarrow$ is partial order.

For events such that neither $a \rightarrow b$ nor $b \rightarrow a$ we say $a, b$ are concurrent, denoted $a \parallel b$. 
Example of causal ordering

- a → b, c → d
- b → c, d → f
- a \parallel e
Logical clocks [Lamport]

- **Logical clock** = monotonically increasing software counter (not real time!)
  - one for each process P, used for timestamping

- **How it works**
  - $L_P$ incremented before assigning a timestamp to an event
  - when P sends message m, P timestamps it with current value t of $L_P$ (after incrementing it), **piggybacking** t with m
  - on receiving message (m,t), Q sets its own clock $L_Q$ to maximum of $L_Q$ and t, then increments $L_Q$ before timestamping the message receive event

- **Note** a → b implies $T(a) < T(b)$
Totally ordered logical clocks

- Problem: $T(a) = T(e)$, and yet $a$, $e$ distinct.
- Create **total** order by taking account of process ids.
- Then $(T(a), \text{pid}) < (T(b), \text{qid})$ iff $T(a) < T(b)$ or $T(a) = T(b)$ and $\text{pid} < \text{qid}$.

11 February, 2002
Vector clocks

- Totally ordered logical clocks
  - arbitrary event order, depends on order of process ids
  - i.e. (T(a),pid) < (T(b),qid) does not imply a → b, see a, e

- Vector clocks
  - array of N logical clocks in each process, if N processes
  - vector timestamps piggybacked on the messages
  - rules for incrementing similar to Lamport’s, except
    - processes own component in array modified
    - componentwise maximum and comparison

- Problems
  - storage requirements
Vector timestamps

- VT(b) < VT(c), hence b → c
- neither VT(b) < VT(e), nor VT(b) < VT(e), hence b || e
Summary

• Local clocks
  – drift!
  – but needed for timestamping

• Synchronisation algorithms
  – must handle variable message delays

• Clock compensation estimate average delays
  – adjust clocks
  – can deal with faulty clocks

• Logical clocks
  – sufficient for causal ordering