Imperial College London

Computer Networks and Distributed Systems

Part 2.4 – Time Services

Course 527 – Spring Term 2016-2017

Emil Lupu

e.c.lupu@imperial.ac.uk

Contents

Requirements

Measure delays between distributed components

Synchronise streams e.g. sound and vision

Detect event ordering for causal analysis

Utilities use modification timestamps e.g. archive, make

Local Time

Quartz crystal oscillates and decrements counter.

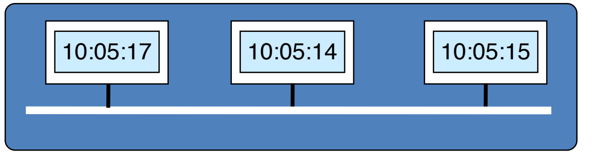
On zero, counter is reset to the value in clock register and causes an interrupt. Interrupt rate controlled by value in register.

Interrupt handler updates software clock e.g. secs since 1/1/1970

Problems

A clock's frequency varies with temperature

Clocks on different computers drift due to differing oscillation period



- Typical accuracy is 1 in 10^{-6} = 1 sec in 11.6 days
- Centralised time service? Impractical due to variable message delays

Time Sources

Universal Coordinated Time (UTC)
Based on atomic clocks but leap seconds inserted to keep in phase with astronomical time - earth's orbit round sun.

Radio stations broadcast UTC & provide a short pulse every second. Random atmospheric delays make accuracy ±10 msec

Geostationary Environment Operation Satellite (GEOS) or Global Positioning Systems (GPS) provide UTC to ±0.5msec

Require (GPS or UTC) receivers on servers to support a clock synchronisation service. *What about everything else?*

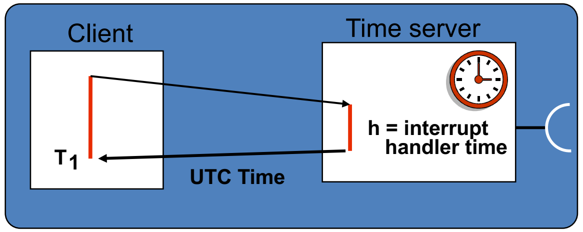
Clock Compensation

Assume 2 clocks can each drift at rate of r msec/s.
Max difference = 2r msec/s
To guarantee accuracy between 2 clocks to within d msecs requires resynch every d/2r secs.

Get UTC and correct software clocks. What happens if local clock is 5 secs fast and you set it right?
Time must never run backward! Rather slow clock down.

Clock register normally set to generate interrupts every 10msec and interrupt handler adds 10msec to software clock. Instead add 9 until correction is made or add 11 to advance clock.

Cristian's Algorithm



Time Server with UTC receiver gives accurate current time

Estimate of message propagation time $p = (T1 - T0 - h)/2$. Set clock to UTC + p

Measure $T1 - T0$ over a number of transactions but remove outliers and/or take minimum values as being most accurate

Single server would be point of failure & bottleneck

An impostor or faulty server sending incorrect times can wreak havoc

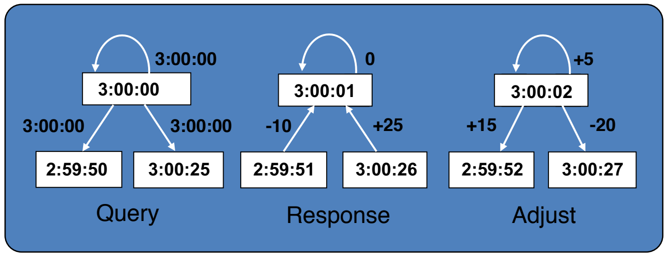
Berkley Algorithm

Co-ordinator chosen as master & periodically polls slaves to query clocks.

Master estimates local times with compensation for propagation delay

Calculate average time, but ignore occasional readings with propagation delay greater than a cut-off value or whose current clock is badly out of synch.

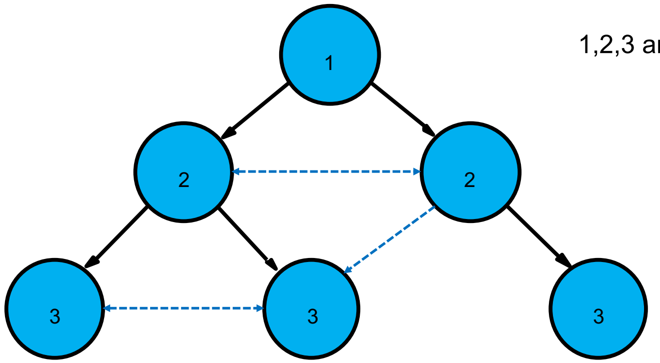
Sends message to each slave indicating clock adjustment



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

Network Time Protocol (NTP)

Multiple servers across the Internet connected to UTC receivers.
Secondary servers synchronise with primaries
Tertiary Servers synchronise with secondary servers etc.
Scales to large numbers of servers and clients



1,2,3 are strata

NTP Synchronisation Modes

Multicast

- one or more servers periodically multicast to other servers on high speed LAN. They set clocks assuming some small delay.

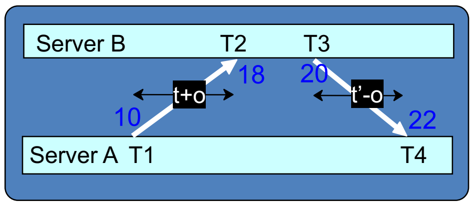
Procedure Call Mode

- Similar to Cristian's algorithm. A client requests time from a few other servers.
- Used where there is no multicast or higher accuracy is needed e.g. a group of file servers on a LAN

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. 5 ms)
o = clock offset of B relative to A (e.g. 3 ms)
Let a = T2 - T1 = **t** + **o**,   Let b = T4 - T3 = **t'** - **o**
RTT = **t** + **t'** = a + b = (T2 - T1) + (T4 - T3)
If T1 = 10, T2 = 18, T3 = 20 and T4 = 22 then RTT = 10
2**o** = a - b = (T2-T1) - (T4 - T3) + (**t** - **t'**) = (T2-T1) - (T4 - T3) = 8-2 = 6
Clock offset **o** = (a-b)/2 = ((T2-T1) - (T4 - T3))/2 = 3
(assuming t ≈ t)

NTP Symmetric Protocol

T4 = current message receive time is determined at receiver. Every message contains:

- T3 = current message send time
- T2 = previous received message receive time
- T1 = previous received message send time

Data filtering: values of o which correspond to minimum values of t are used to get average values of actual clock offset.

Peer selection: exchange messages with several peers looking for most reliable values favouring lower level ones (e.g. primaries)

20-30 primaries and over 2000 secondaries can synchronise to within 30ms.

Logical Time

For many purposes it is sufficient that processes *agree* on the same time (i.e. internal consistency) which need not be real or UTC time.

Event Ordering

$a \rightarrow b$ = a happens before b

1. If a and b are events in the same process and a occurs before b then $a \rightarrow b$ is true
2. If is a is the event of message sent from process A and b is the event of message receipt by process B then $a \rightarrow b$ is true
3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
4. If x and y happen in different processes which do not exchange messages then $x \rightarrow y$ is not true and $y \rightarrow x$ is not true ie x and y are said to be **concurrent** and nothing can be said about their order.

Logical time denotes causal relationship but the $\rightarrow$ relationship may not reflect real causality e.g. a process may receive message x and then send message y so $x \rightarrow y$ even though it would have sent y if x had not been received.

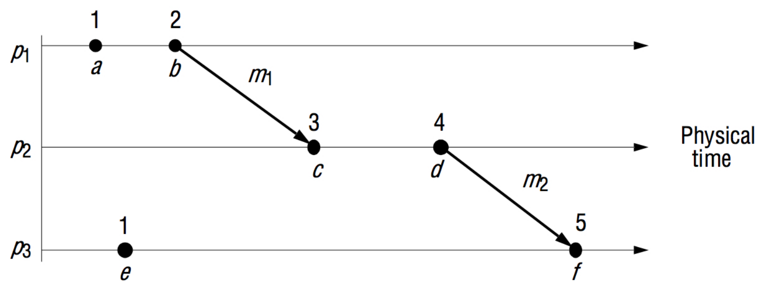
Lamport's Logical Clocks

A monotonic software counter can be used to implement logical clocks. Each process p keeps its own logical clock C_p which it uses to timestamp events

1. C_p is incremented before assigning a timestamp to an event at process *p*
2. When a process p sends a message m, it timestamps it by including the value $t = C_p$ (after incrementing Cp)
3. When a process q receives a message *(m, t)* it sets $C_q := \max(C_q, t)$ then C_q is incremented and assigned as a timestamp to the message received event.

Note: $a \rightarrow b$ implies $T_a < T_b$ but $T_a < T_b$ does not imply $a \rightarrow b$

Logical Clocks - Total Ordering



Logical Clocks give a partial order on the set of all events as distinct events can have the same identifier.

A total ordering can be imposed by including the process identifier with the event identifier

$(T_a, P_a) < (T_b, P_b)$ if and only if $T_a < T_b$, or $T_a = T_b$ and $P_a < P_b$

Vector clocks [Mattern and Fidge]

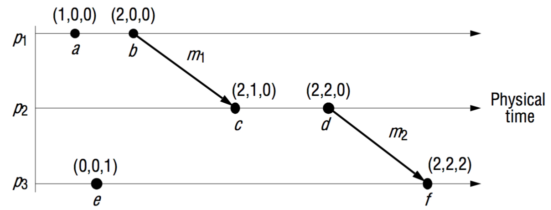
A vector clock for N processes is an array of N integers. Each process keeps its own vector clock, Vi , which it uses to timestamp local events.

When *pi* receives a timestamp *t* in a message, it sets

$V_i[j] := \max(V_i[j], t[j])$, for $j = 1, 2,.., N$.

$V_i[i]$ is the number of events that p_i has timestamped

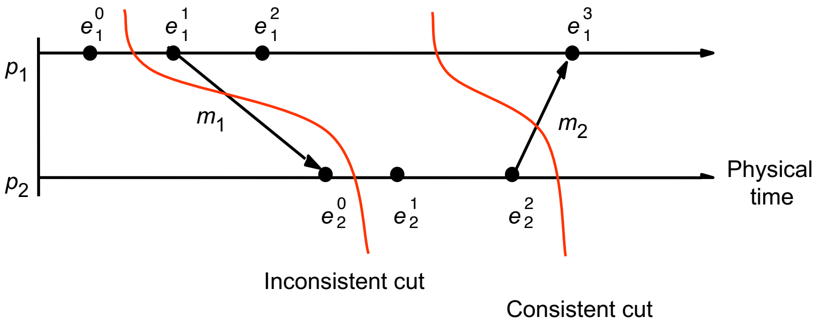
$V_i[j]$, (j *different than* i) is the number of events that have occurred at p_j that have potentially affected p_i .



Global state

A consistent cut is one where the causes are present for all the effects.

A consistent global state state corresponds to a consistent cut.



Summary

Local clock drift results in non-synchronised clocks

Synchronisation algorithms have to cope with variable message delays between nodes

Clock compensation algorithms send local readings, and estimate average delays to derive clock adjustments eg

- Cristian
- Berkley
- NTP

Logical clocks are sufficient for causal ordering e.g. event dependencies – based on incrementing counters