Introduction

- We need to measure time accurately
  - to know the time an event occurred at a computer
  - to do this we need to synchronize its clock with an authoritative external clock
- Algorithms for clock synchronization useful for
  - concurrency control based on timestamp ordering
  - authenticity of requests e.g. in Kerberos
- There is no global clock in a distributed system
  - Issues of clock accuracy and synchronization
- Logical time is an alternative for
  - ordering of events - also useful for consistency of replicated data

In the NFS client cache scheme, what properties do we require from the clocks?
Computer Clocks and Timing Events

- Each computer in a DS has its own internal clock
  - used by local processes to obtain the value of the current time
  - processes on different computers can timestamp their events
  - but clocks on different computers may give different times
  - computer clocks drift from perfect time and their drift rates differ from one another.
  - clock drift rate: the relative amount that a computer clock differs from a perfect clock

Even if clocks on all computers in a DS are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied

Hardware & Software Clocks

We have seen how to order events at a process
To timestamp events, use the computer’s clock
At real time, $t$, the OS reads the time on the computer’s hardware clock $H(t)$
It calculates the time on its software clock $C(t) = \alpha H(t) + \beta$
- e.g. a 64 bit number giving nanoseconds since some base time
- in general, the clock is not completely accurate
- but if $C_i$ behaves well enough, it can be used to timestamp events at $p_i$

What is clock resolution? How accurate should it be?
Skew between Computer Clocks in a Distributed System

Computer clocks are not generally in perfect agreement

**Skew**: the difference between the times on two clocks (at any instant)

Computer clocks are subject to **clock drift** (they count time at different rates)

**Clock drift rate**: the difference per unit of time from some ideal reference clock

Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).

High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

What happens to clocks when their batteries become very low?

---

Coordinated Universal Time (UTC)

International Atomic Time is based on very accurate physical clocks (drift rate 10^{-13})

UTC is an international standard for time keeping

It is based on atomic time, but occasionally adjusted to astronomical time

It is broadcast from radio stations on land and satellite (e.g. GPS)

Computers with receivers can synchronize their clocks with these timing signals

Signals from land-based stations are accurate to about 0.1-10 millisecond

Signals from GPS are accurate to about 1 microsecond

Why can't we put GPS receivers on all our computers?
Synchronizing Physical Clocks: External and Internal

External synchronization
- A computer’s clock $C_i$ is synchronized with an external authoritative time source $S$, so that:
  - $|S(t) - C_i(t)| < D$ for $i = 1, 2, \ldots N$ and for all real time $t$ in $I$.
- The clocks $C_i$ are accurate to within the bound $D$.

Internal synchronization
- The clocks of a pair of computers are synchronized with one another so that:
  - $|C_i(t) - C_j(t)| < D$ for $i, j = 1, 2, \ldots N$, and for all real time $t$ in $I$.
- The clocks $C_i$ and $C_j$ agree within the bound $D$.

Internal synchronized $\Rightarrow$ external synchronized?

External synchronized $\Rightarrow$ internal synchronized?

Clock Correctness

A hardware clock, $H$ is said to be correct if its drift rate is within a bound $\rho > 0$. (e.g. $10^{-6}$ secs/sec)
- This means that the error in measuring the interval between real times $t$ and $t'$
  is bounded:
  - $(1 - \rho) (t' - t) \leq H(t') - H(t) \leq (1 + \rho) (t' - t)$ (where $t' > t$)
- Which forbids jumps in time readings of hardware clocks

Weaker condition of monotonicity
- $t' > t \Rightarrow C(t') > C(t)$
- e.g. required by Unix make
- can achieve monotonicity with a hardware clock that runs fast by adjusting the values of $\alpha$ and $\beta$, $C(t) = \alpha H(t) + \beta$

A faulty clock is one that does not obey its correctness condition
- crash failure - a clock stops ticking
- arbitrary failure - any other failure e.g. jumps in time

Consider the 'Y2K bug' - what sort of clock failure would that be?
**Synchronization in a Synchronous System**

A synchronous distributed system is one in which the following bounds are defined:
- the time to execute each step of a process has known lower and upper bounds
- each message transmitted over a channel is received within a known bounded time
- each process has a local clock whose drift rate from real time has a known bound

**Internal synchronization in a synchronous system**
- One process $p_1$ sends its local time $t$ to process $p_2$ in a message $m$,
- $p_2$ could set its clock to $t + T_{trans}$ where $T_{trans}$ is the time to transmit $m$
- $T_{trans}$ is unknown but $\min \leq T_{trans} \leq \max$
- **uncertainty** $u = \max - \min$. Set clock to $t + (\max - \min)/2$ then skew $\leq u/2$

- What is the optimum bound when synchronizing $N$ clocks?

Is the Internet a synchronous system? Any bound on transmission?

---

**Cristian’s Method (1989) for an Asynchronous System**

A time server $S$ receives signals from a UTC source
- Process $p$ requests time in $m_r$ and receives $t$ in $m_t$ from $S$
- $p$ sets its clock to $t + T_{round}/2$
- **Accuracy $\pm (T_{round}/2 - \min)$**
  - because the earliest time $S$ puts $t$ in message $m_t$ is $\min$ after $p$ sent $m_r$
  - the latest time was $\min$ before $m_t$ arrived at $p$
  - the time by $S$’s clock when $m_t$ arrives is in the range $[\min, t + T_{round} - \min]$

$T_{round}$ is the round trip time recorded by $p$
$\min$ is an estimated minimum single trip time

What is the potential problem in using a single time server?
Berkeley Algorithm

Cristian’s algorithm -
- a single time server might fail, so they suggest the use of a group of synchronized servers
- it does not deal with faulty servers

Berkeley algorithm (also 1989)
- An algorithm for internal synchronization of a group of computers
- A master polls to collect clock values from the others (slaves)
- The master uses round trip times to estimate the slaves’ clock values
- It takes an fault-tolerant average (eliminating any above some (maximum) average round trip time, or with faulty clocks – using a subset of close clocks)
- It sends the required relative adjustment to the slaves (better than sending the absolute time which depends on the round trip time)
- Measurements
  - 15 computers, clock synchronization 20-25 millisecs drift rate < 2x10^-5
  - If master fails, can elect a new master to take over (not in bounded time)

Network Time Protocol (NTP)

A time service for the Internet - synchronizes clients to UTC
- Reliability from redundant paths, scalable, authenticates time sources
- Primary servers are connected to UTC sources
- Secondary servers are synchronized to primary servers
- Synchronization subnet - lowest level servers in users’ computers

A synchronization subset
NTP - Synchronisation of Servers

The synchronization subnet can reconfigure if failures occur, e.g.
- a primary that loses its UTC source can become a secondary
- a secondary that loses its primary can use another primary

Modes of synchronization (UDP):
- Multicast
  - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
- Procedure call
  - A server accepts requests from other computers (like Cristiain’s algorithm). Higher accuracy; also useful if no hardware multicast.
- Symmetric
  - Pairs of servers exchange messages containing time information
  - Used where very high accuracies are needed (e.g. for higher levels)

Messages Exchanged between a Pair of NTP Peers

What is the key problem in internal synchronization?
- What is the bound of the clock offset/skew between two servers?

Each message bears (3) timestamps of recent events:
- Local times of Send and Receive of previous message
- Local times of Send of current message

Recipient notes the time of receipt $T_i$ (we have $T_{i-3}$, $T_{i-2}$, $T_{i-1}$, $T_i$)

In symmetric mode there can be a non-negligible delay between arrival of one message and the dispatch of the next
Accuracy of NTP – bound of clock offset/skew

For each pair of messages between two servers, NTP estimates an offset \( o \), between the two clocks and a delay \( d_i \) (total time for the two messages, which take \( t \) and \( t' \))

\[ T_{i-2} = T_{i-3} + t + o \text{ and } T_i = T_{i-1} + t' - o \]

This gives us (by adding the equations):

\[ d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1} \]

Also (by subtracting the equations)

\[ o = o_i + (t' - t)/2 \text{ where } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 \]

Using the fact that \( t, t' > 0 \) it can be shown that

\[ o_i - d_i/2 \leq o \leq o_i + d_i/2 . \]

– Thus \( o_i \) is an estimate of the offset and \( d_i \) is a measure of the accuracy

NTP servers filter pairs \( \langle o_i, d_i \rangle \), estimating reliability from variation (a. synchronization dispersion), allowing them to select peers (b)

Accuracy of 10s of ms over Internet paths (1 on LANs)

Clocks, Events and Process States

A distributed system is defined as a collection \( P \) of \( N \) processes \( p_i \), \( i = 1, 2, \ldots N; \)

Each process \( p_i \) has a state \( s_i \), consisting of its variables (which it transforms as it executes)

Processes communicate only by messages (via a network)

Actions of processes:

– Send, Receive, change own state

Event: the occurrence of a single action that a process carries out as it executes e.g. Send, Receive, change state

Events at a single process \( p_i \), can be placed in a total ordering denoted by the relation \( \rightarrow_i \) between the events. i.e.

\( e \rightarrow e' \) if and only if \( e \) occurs before \( e' \) at \( p_i \)

A history of process \( p_i \) is a series of events ordered by \( \rightarrow_i \)

\( \text{history}(p_i) = h_i = <e^0_i, e^1_i, e^2_i, \ldots> \)

The 'happened before' relation is essential to understanding logical clocks
Logical Time and Logical Clocks (Lamport 1978)

Instead of synchronizing clocks, event ordering can be used

Event ordering in a single process is easy
– Event ordering in a distributed system is more complex

1. If two events occurred at the same process $p_i$ ($i = 1, 2, \ldots N$) then they occurred in the order observed by $p_i$, that is $\rightarrow_i$
2. When a message, $m$ is sent between two processes, $send(m)$ happened before $receive(m)$
3. The happened before relation is transitive
   $e \rightarrow e' \text{ and } e' \rightarrow e''$, then $e \rightarrow e''$

HB1, HB2 and HB3 (page 397) are formal statements of these 3 points

the happened before relation is the relation of causal ordering

Events Occurring at Three Processes

$a \rightarrow b$ (at $p_1$) $c \rightarrow d$ (at $p_2$) $b \rightarrow c$ because of $m_1$ also $d \rightarrow f$ because of $m_2$

Not all events are related by $\rightarrow$
consider $a$ and $e$ (different processes and no chain of messages to relate them)
they are not related by $\rightarrow$; they are said to be concurrent; write as $a \parallel e$
Lamport’s Logical Clocks

A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.
- To numerically capture happened-before ordering

Each process $p_i$ has a logical clock, $L_i$ which can be used to apply logical timestamps to events
- $L_i(e)$: the timestamp of event $e$ at $p_i$
- $L(e)$: the timestamp of event $e$ at any process
- $LC1$: $L_i$ is incremented by 1 before each event is issued at process $p_i$
- $LC2$:
  - (a) when process $p_i$ sends message $m$, it piggybacks on $m \equiv L_i$
  - (b) when $p_j$ receives $(m,t)$ it sets $L_j := \max(L_j, t)$ and applies $LC1$ before timestamping the event receive $(m)$

$e \rightarrow e' \Rightarrow L(e) < L(e')$ \hspace{1cm} $L(e) < L(e') \Rightarrow e \rightarrow e'$?

An Example of Lamport’s logical clocks

Each of $p1, p2, p3$ has its logical clock initialised to zero, the clock values are those immediately after the event. e.g. 1 for a, 2 for b.

For $m1$, 2 is piggybacked and c gets $\max(0,2)+1 = 3$

Give an example to show the converse is not true

$e \rightarrow e'$ implies $L(e)<L(e')$ The converse is not true, that is $L(e)<L(e')$ does not imply $e \rightarrow e'$. e.g. $L(b) > L(e)$ but $b \parallel e$
Vector Clocks

The shortcoming of Lamport logical clocks:
- \( L(e) < L(e') \) does not imply \( e \) happened before \( e' \)

Vector clock \( V_j \) at process \( p_i \) is an array of \( N \) integers, which the process use to timestamp its local events
- VC1: initially \( V[j] = 0 \) for \( i, j = 1, 2, \ldots N \)
- VC2: before \( p_i \) timestamps an event it sets \( V[i] := V[i] + 1 \)
- VC3: \( p_i \) piggybacks \( t = V_i \) on every message it sends
- VC4: when \( p_i \) receives \((m, t)\) it sets \( V[j] := \max(V[j], t[j]) \) \( j = 1, 2, \ldots N \).

\( V[i] \) is the number of events that \( p_i \) has timestamped
\( V[j] \ (j \neq i) \) is the number of events at \( p_j \) that \( p_i \) has been affected by

---

**An Example of Vector clocks**

At \( p_1 \) \((1,0,0)\) \( b \) \((2,0,0)\) piggyback \((2,0,0)\) on \( m_1 \)
At \( p_2 \) on receipt of \( m_1 \) get \( \max ((0,0,0), (2,0,0)) = (2, 0, 0) \) add 1 to own element \( = (2,1,0) \)
Meaning of =, <=, max etc for vector timestamps - compare elements pairwise
Note that \( e \rightarrow e' \) implies \( L(e) \leq L(e') \). The converse is also true. Exercise 10.3 for a proof.

Can you see a pair of parallel events?
Detecting Global States

a. Garbage collection

b. Deadlock

c. Termination

Summary

accurate timekeeping is important for distributed systems. algorithms (e.g. Cristian’s and NTP) synchronize clocks in spite of their drift and the variability of message delays. for ordering of an arbitrary pair of events at different computers, clock synchronization is not always practical. the happened-before relation is a partial order on events that reflects a flow of information between them. Lamport clocks are counters that are updated according to the happened-before relationship between events. vector clocks are an improvement on Lamport clocks, – we can tell whether two events are ordered by happened-before or are concurrent by comparing their vector timestamps
Clock Synchronization

When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

How *make* works?

Example