Synchronizing Physical Time

Observations:

- In some systems (e.g., real-time systems) actual time are important, and we typically equip every computer host with one physical clock.
- Computer clocks are unlikely to tick at the same rate, whether or not they are of the 'same' physical construction.
  - E.g., a quartz crystal clock has a drift rate of 10^{-6} (ordinary), or 10^{-7} to 10^{-8} (high precision).
  - C.f. an atomic clock has a drift rate of 10^{-13}.

Questions:

1. How do we synchronize computer clocks with real-world clocks?
2. How do we synchronize computer clocks themselves?

Some Stories

- Since Jan. 1, 1958, the standard *second* has been defined as 9,192,631,770 transitions of Cs^{133}. The *time* defined by this is called ***International Atomic Time* (*TAI*).** TAI time has been worldwide accepted since 1967.
- The *solar second* equals $1/(24x3600)_{th}$ of the *solar day*, which is obtained by measuring the interval between the two points where the sun reaches its highest position (i.e., at noon).
- However, the solar time gets longer and longer (about 30 TAI seconds in the past 40 years).

Some Stories (cont.)

- ***Coordinated universal time (UTC)*** is an international standard (replacing the astronomical-based Greenwich Mean Time) that is based on TAI, but stays in phase with the apparent motion of the sun (by inserting *leap seconds* to TAI time).
- UTC signals are broadcast regularly from land-based radio stations (with accuracy 0.1-10 ms) and satellites (with accuracy 0.1-1 ms).
- UTC receivers are commercially available, but is expensive.

Compensation for clock drift

- A computer clock usually can be adjusted forward but not backward.
  - Typical example: Y2K problem.
- Common terminology:
  - ***Skew*:** the instantaneous difference between (the readings of) two clocks.
  - ***Drift rate*:** the difference between the clock and a nominal perfect reference clock per unit of time.
- Linear adjustment:
  - Let *C* be the software reading of a hardware clock *H*. Then the operating system usually produces C in terms of H by the following: $C(t) = \alpha H(t) + \beta$

Cristian's Algorithm

When *P* receives the message, it should set its time to $t+T_{trans}$, where T_{trans} is the time to transmit the message.

$T_{trans} \approx T_{round}/2$, where T_{round} is the round-trip time

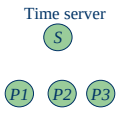
Accuracy.

Let *min* be the minimum time to transmit a message one-way.
Then *P* could receive *S*'s message any time between

$$[t+min, t+ T_{round} - min]$$

So accuracy is $\pm(T_{round}/2 - min)$

The Berkeley algorithm



1. Server polls each client.
2. Each client responds to the server its local time.
3. The server estimates the clients' local time (similar to Cristian's technique), and averages the time (including the server's own reading, but excluding those that may have drifted badly). It then tells each client their **offset.** (*Why not the actual clock value?*)

The Network Time Protocol (NTP)

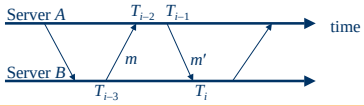
- Provide a service enabling clients across the Internet to be synchronized accurately to UTC, despite the large and variable message delays encountered in Internet communication.
- Provide a reliable service that can survive lengthy losses of connectivity there are redundant servers and paths between servers.
- Enable clients to synchronize sufficiently frequently; to offset the rates of drift found in most computers.
- Provide protection against interference with the time service.

NTP – Basic Concepts

- The NTP servers are connected in a logical hierarchy, where servers in level *n* are synchronized directly to those in level *n–1* (which have a higher accuracy). The logical hierarchy can be reconfigured as servers become unreachable or failed.
- NTP servers synchronize with one another in one of three modes (in the order of increasing accuracy):
  - Multicast on high speed local LANs
  - Procedure call mode (*a la* Cristian's algorithm)
  - Symmetric mode (for achieving highest accuracy).

Symmetric Mode

A pair of servers exchange timing information



Assume: m takes t to transfer, m' takes t' to transfer
Offset between A's clock and B's clock is o; i.e., $A(t) = B(t) + o$

Then, $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} - o + t'$

Assuming that $t \approx t'$, then the offset o can be estimated as follows: $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i) / 2$

Since $T_{i-2} - T_{i-3} + T_{i-1} - T_i = t + t'$ (let's say, $t + t'$ equal to d_i)

Then $o = o_i + (t' - t)/2$

Given that $t', t \geq 0$, the accuracy of the estimate of o_i is:

$$o_i - d_i/2 \leq o \leq o_i + d_i/2$$

Symmetric Mode (cont.)

- The eight most recent pairs $<o_i, d_i>$ are retained; the value of o_i that corresponds to the minimum d_i s chosen to estimate o.
- Timing messages are delivered using UDP.

Synchronizing Logical Clocks

Observations.

- If two events occurred at the same process, then they occurred in the order in which it observes them.
- Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

Causal ordering (*happened-before* relation)

1. If process p execute x before y, then $x \rightarrow y$.
2. For any message m, $send(m) \rightarrow rcv(m)$.
3. If $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$.

Two events a and b are said **concurrent** if neither $a \rightarrow b$ nor $b \rightarrow a$.

Logical Clocks

A **logical clock** C_p of a process p is a software counter that is used to timestamp events executed by p so that the *happened-before* relation is respected by the timestamps.

The rule for increasing the counter is as follows:

- LC1: C_p is incremented before each event issued at process p.
- LC2: When a process q sends a message m to p, it piggybacks on m the current value t of C_q; on receiving m, p advances its C_p to max(t, C_p).

Illustration of Timestamps

timestamp	*P1*:	*P2*:	timestamp
1	x:=1;	y:=2;	1
2	y:=0;		
3	send (x) to *P2*;		
		receive(x) from *P1*;	4
		x:=4;	5
		send (x+y) to *P2*;	6
7	receive(y) from *P2*;	x:= x+y;	7
8	x:=x+y;		

Reasoning about timestamps

Consequence: if $a \rightarrow b$, then $C(a) < C(b)$

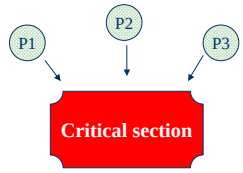
The partial ordering can be made total by additionally considering process ids.

Suppose event a is issued by process p, and event b by process q. Then the total ordering $\rightarrow_t$ can be defined as follows:

$a \rightarrow_t b$ iff $C(a) < C(b)$ or $C(a) = C(b)$ and ID(p) < ID(q).

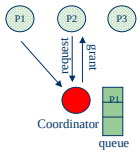
Does $C(a) < C(b)$ imply $a \rightarrow b$?

Example: Mutual Exclusion



Attempt: Using Physical Timestamps

- Use a centralized coordinator to main a queue of requests, which are ordered by physical timestamps.
- A process wishing to enter CS sends a request to the coordinator, and enters the CS when the coordinator grants its request.



Problems:

1. The coordinator becomes a bottleneck.
2. How many other requests may overtake a process request?
3. No fault tolerance.

Lamport's Solution

Each process maintains its own request queue, in which requests (to CS) are ordered by timestamps

- Each process broadcasts a request message when it wishes to enter CS. (Note: all messages are timestamped.)
- A process enters CS if
  (i) Its request is on top of its request queue, and
  (ii) It has received Ack to its request from every other process.
- After exiting CS, a process P broadcast an Exit message to tell all processes to remove P's request from their queues.

Does this solution addresses all the three problems in the previous slide?

Vector Timestamps

Each process P_i maintains a vector of clocks VT_i such that $VT_i[k]$ represents a count of events that have occurred at P_k and that are known at and that are known at P_i.

The vector is updated as follows:

1. All processes P_i initializes its VT_i to zeros.
2. When P_i generates a new event, it increments $VT_i[i]$ by 1; VT_i is assigned as the timestamp of the event. Message-sending events are timestamped.
3. When P_j receives a message with timestamp vt, its updates its vector clock as follows:

$$VT_i[k] := \max(VT_i[k], vt[k])$$

Illustration of Vector Timestamps

timestamp	*P1*:	*P2*:	timestamp
⟨1,0⟩	*x*:=1;	*y*:=2;	⟨0,1⟩
⟨2,0⟩	*y*:=0;		
⟨3,0⟩	send (*x*) to *P2*;		
		receive(*x*) from *P1*;	⟨3,2⟩
		x:=4;	⟨3,3⟩
		send (*x*+*y*) to *P2*;	⟨3,4⟩
⟨4,4⟩	receive(*y*) from *P2*;	*x*:= *x*+*y*;	⟨3,5⟩
⟨5,4⟩	*x*:=*x*+*y*;		

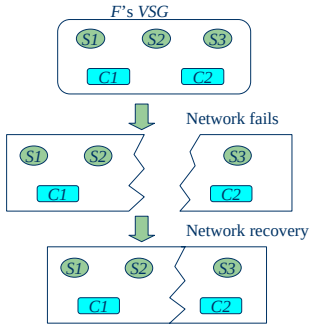
Reasoning about vector timestamps

Partial orders '≤' and '<' on two vector timestamps u, v are defined as follows:

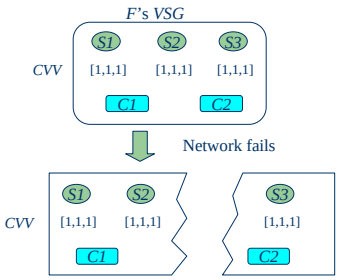
$u \le v$ iff $u[k] \le v[k]$ for all k's, and $u < v$ iff $u \le v$ and $u \ne v$.

Property: e happened-before f if, and only if, $vt(e) < vt(f)$.

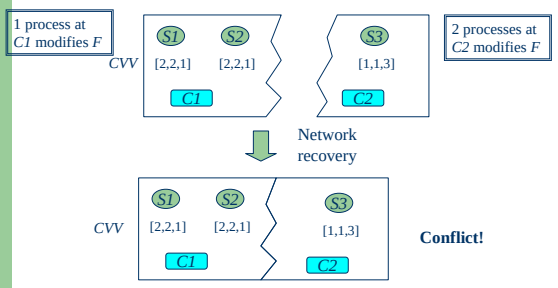
Example: Replication in Coda



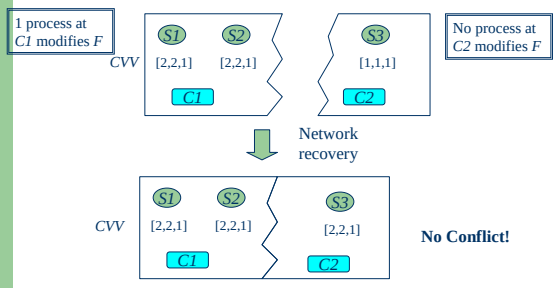
Example: Replication in Coda (cont.)



Example: Replication in Coda (cont.)



Example: Replication in Coda (cont.)



Problems about vector clocks

- Drawback: scalability
- However, if we are to be able to tell whether or not two events are concurrent by inspecting their timestamps, then the size of a timestamp in proportion to *N* is unavoidable [Charron-Bost, 1991].
- Some techniques can be used to store and transmitting smaller amount of data, at the expense of the processing required to reconstruct complete vectors [Raynal and Singhal, 1996].