Clock Synchronization

Figure 5-1 as motivation

Lamport's paper

Section 17.1 SGG

"Time, Clocks, and the Ordering of Events in a Distributed System" by Leslie Lamport. Communications of the ACM. July 1978.

A program is a sequence of events. Partial ordering "happened before" relationship.

For events a, b, $a \to b$ means:

1. a comes before b in the same process
2. sending of message at a to b in another process
3. transitive. $a \to b$ and $b \to c$ implies $a \to c$

Two events are concurrent if it is not known if one happens before the other.

$a \to b$ means it is possible for event a to *causally* affect event b.

See picture from paper.

Logical Clocks

Can we set up a logical clock to guarantee a partial ordering of events?

Assigning numbers to events:
$C_i(a)$ number of event a in process i (one processor's clock)
$C(b) = C_j(b)$ for process j containing event b (set of all clocks)

Clock Condition:
$a \rightarrow b$ implies $C(a) < C(b)$

Note that $a \not\rightarrow b$ and $b \not\rightarrow a$ does not imply $C(a) = C(b)$.

C1: $a \rightarrow b$ in process i implies $C_i(a) < C_i(b)$
C2: passing message from i, event a to j, event b implies $C_i(a) < C_j(b)$.

Implementation Rules:

1. Each process P_i increments C_i between any two successive events (statements).
2. Messages contain a timestamp $T_m = C_i(a)$.

   Upon receiving a message P_j sets $C_j = \max(C_j + 1, T_m + 1)$

Provides a partial ordering of causally related events.

Can extend to a *total ordering* of events by assigning processors a priority to break ties with the clock.

Can use total ordering to ensure mutual exclusion in a distributed environment.

WWV clock

Fort Collins, CO has atomic clock built on vibratiions of Cesium 133 atom.

Universal Coordinated Time (introduction of leap seconds). Can have a receiver on the net tuned to WWV.

Physical Clock Synchronization

Want to define a constant ρ such that the *maximum drift rate* of the clock (C) is bounded.

$$1 - \rho \leq dC/dt \leq 1 + \rho$$

Want maximum drift between two clocks to be δ, must be resampled every $\delta/2\rho$ (each clock can drift ρ amount in opposite directions)

Cristian's algorithm. Send a message to the time server and get back a reply. Adjust clock to the time. Fig. 11-6. Considerations:

- Cannot adjust clock backwards (rather must move the clock gradually backwards)
- Propagation delay of message is at least $(T_1 - T_0)/2$. Could also add in processing time if known.
- Problem if central time server fails. Temporarily lose service.

Berkeley algorithm. Time server periodically computes a network time and sends it out to everyone else. Does not synchronize to an external time.

Averaging algorithms. Broadcast time—decentralized. Could also pick a random node to send to, less overhead, but slower convergence.

Multiple External Time Sources. Intersect them, average, and throw out any outlyers. Fig. 11-8 for example of OSF's DCE approach. Universal Coordinated Time (UTC).

Network Time Protocol (NTP). Standardized time protocol. Use a hierarchy of servers with those at the top receiving UTC directly (Fig 10.3) Three modes:

1. multicast—on a LAN
2. procedure call—like Cristian's algorithm
3. symmetric—servers communicate and maintain a timing association

Protocol uses symmetric mode to calculate offset o and delay d between two clocks (Fig 10.4).

Mutual Exclusion

Can use Lamport's algorithm.

Look at centralized algorithm. Fig. 11-9.

Distributed algorithms. Ricart-Agrawala have an updated version of Lamport's algorithm.

Token Based Algorithms

Create a ring (logical or physical) and pass a token between the nodes. If the node needs the critical section then it grabs the token.

Suzuki-Kasami's Broadcast algorithm—keep a vector of current state and broadcast requests. Each machine broadcasts a request for the token when it is needed.

Singhal has a heuristic improvement to only send to the sites it thinks may have the token.

Comparison

Look at Fig 11-12.

Election Algorithms

Used when a unique process needs to be distinguished to play a particular role.

One process may *call an election* when the need arises.

Elections must work in the face of multiple processes calling an election.

Bully Algorithm

Assumes that a process knows about other processes with a higher identifier.

A process holds an election to elect a leader. The election is held as follows:

1. P sends an *election* message to all processes (sites) with a higher number.
2. If no one responds with an *answer* message, P wins and becomes the leader.
3. If a higher-up answers then it takes over and starts an election. P is done.
4. Winner sends a *coordinator* message to all nodes indicating it has won.

Ring-Based

Similar to bully algorithm in trying to elect the highest numbered node.

Any node can start and marks itself as a *participant* (vs. *non-participant*) in an election. It puts its identifier in the *election* message. Successive nodes mark themselves as participants and if they have a greater value then they substitute their id in the elected message, otherwise they just pass the value on.

When the receiver gets its id back then it is the *coordinator* and sends an *elected* message. The other nodes use this message to mark themselves as *non-participants*.

Notion of participation is used to quelch other elections. At worst the protocol could take $3n-1$ messages ($n-1$ to find leader, n before leader sees its id again and n to send around elected message).

Show a picture with 3-9-15-6-12-18 (start with 3).

Termination Detection

How to know when a distributed computation (election, deadlock detection, distributed query) terminates??

Huang, 1989.

Use a weighting algorithm with a controlling agent. Initially the controlling agent has a weight of one and all other nodes have a weight of zero. `Show picture`.

Invariant: sum of weights in the system is always one.

Algorithm

1. When a computation is sent by a process with weight W to another process then divide W into $W_1 + W_2$. Reassign $W = W_1$ and send W_2 to P.
2. On receipt of message, the process P adds the weight to its current weight.
3. If a process is no longer active then it sends its entire weight W to the controlling agent.
4. On receiving a message the controlling agent adds the weight to its current value. When the weight returns to one the computation is terminated.