CSCI 344/715
Lecturer: Dr. Simina Fluture

Synchronization in Distributed Systems
Clock Synchronization

Readings: Lecture on the web
Class notes

Bibliography:
[AT] ch. 3.1, 3.2

Clock Synchronization

In a centralized system, time is unambiguous. When a process needs to know the time it makes a system call to the kernel. If ***A*** asks for the time, and shortly after that ***B*** asks for the time, the value that ***B*** gets should be higher than the value that ***A*** got.
In a distributed system, achieving agreement on time is not trivial.

Example: implications of the lack of global time on the Unix *make* program.

Make examines the times at which all the source and object files were last modified.
If the source file *input.c* has time 2151 and the corresponding object file *input.o* has time 2150, make knows that *input.c* has been changed since *input.o* was created, and thus *input.c* must be recompiled. *Make* goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile them.

Suppose that *output.o* has time 2144 and shortly thereafter *output.c* is modified but is assigned time 2143 because the clock on its machine is slightly slow. *Make* will not call the compiler.

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

Logical clocks

In practice when a system has *n* computers, the clocks might get out of synchronization and give different values when read out. Lamport pointed out that clock synchronization need not be absolute. What matters is that processes agree on the order in which events occur. In this case it is conventional to speak of the clocks as **logical clocks**.

The "happens before" relation:

$a \rightarrow b$ means "a happens before b"

1. If *a* and *b* are events in the same process (thread) and event *a* occurs before *b* than $\boldsymbol{a \rightarrow b}$
2. if *a* is the event of a message being sent by one process (thread or machine) and *b* is the event of a message being received by another process (thread, machine) then $\boldsymbol{a \rightarrow b}$ is true.

Note: if x and y are events in different processes and no messages are exchanged then $x \rightarrow y$ is not true, nor is $y \rightarrow x$. We say x and y are concurrent.

We assign a time c(a) to each event such that:.

If $a \rightarrow b$ then $C(a) < C(b)$

If a is the sending of a message by one process and b is the reception of that message by another process, then C(a) and C(b) must be assigned in such a way that $C(a) < C(b)$.

The clock time, C, must always go forward (increasing), never backward (decreasing).

Consider three processes that are running on different machines, each with its own clock, running at its own speed at a constant rate but not unique.

Each message carries the sending time according to the sender's clock.

Topics: Mutual Exclusion implementation
Centralized Algorithm
Token Ring Algorithm

A Centralized Algorithm (election algorithm)

Process P, wants to enter a critical section: sends a *request.*

Coordinator: "no reply" or replies "permission denied". (P blocked on queue)
replies "OK"

The Process leaving the Critical Section: sends a "release" message.

Fig. Centralized algorithm for Mutual Exclusion

3 messages / use of Critical Section: request, OK, release

+) easy to implement
can be used for more general resource allocation
acts like a monitor (messages are like entries, replies are like notify( ).

-) if the coordinator goes down, the system goes down
the single coordinator can become a bottleneck.
there is no distinction between "coordinator down" and "permission denied by no reply"

Token Ring algorithm

In software a logical ring is constructed by which each process has a position in the ring.

Fig. Token Ring algorithm for Mutual Exclusion

When the ring is initialized, process 0 gets a token. The token circulates around the ring.

When a process acquires the token, it checks if it is attempting to enter the CS:

If Yes, the process enters the CS. After it has exited, it passes the token along the ring. It is not permitted to enter a second CS using the same token.
If NO, the process passes the token.

When no process wants to enter the CS, the token circulates at high speed around the ring.

-) token might be lost.
A process might be down. This can be detected by requiring a process receiving the token to acknowledge the receipt.