Synchronization in Distributed Systems

Clock Synchronization

Midterm comments

Readings: Lecture on the web **Bibliography:**

Class notes **[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.

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

Distributed Algorithm

Token Ring Algorithm

<u>A Centralized Algorithm (election algorithm)</u>

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.

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.

<u>DISTRIBUTED MUTUAL EXCLUSION ALGORITHM</u>

Requires total ordering of the events in the system.

A collection of threads that do not share memory, are connected to a local area network (LAN). The threads need mutual exclusion over some shared resources. In order to avoid a bottleneck a central server will not be used, rather the Distributed Algorithm fore Mutual Exclusion will be used.
There will be N nodes connected by the LAN and we assume that:

- there are no lost or garbled messages.
- Messages sometimes arrive in a different order than they were sent.
- Nodes do not fail or halt.

When a process wants to enter the CS:

1) It builds a message:

Message = name of the CS + its process ID + the current time

2) It sends the message to all processes in the system including itself.

When a process receives a request message from another process (there are three cases):

If the receiver is not in the CS and doesn't intend to enter the CS, it sends back an "OK" message.

If the receiver is in the CS, it doesn't reply and queues the request.

If the receiver wants to enter the CS, it compares the timestamp of the incoming message with the one contained in the message sent by itself. If the incoming message is lower, the receiver sends back an "OK" message, otherwise the receiver queues the incoming request and sends nothing.

A process can enter the CS **after receiving the OK** messages from **all** of the other processes.

-) In practice there are now *n* points of failure.

If one process goes down, the system goes down.

Not so easy to implement, slower and more expensive. Need to keep track of a membership list.

Client – Server model
Socket Communication

http://java.sun.com/docs/books/tutorial/networking/overview/networking.html

Interprocess Communication in distributed systems can be done through different methods:

- Message passing (high level)
- Pipes (low level)
- Sockets (low level)
- Remote Procedure Calls (high level)

Client/Server model

Client-server systems are based on a model for interactions between cooperating processes named **Server**s and **Client**s. One process, the *server*, is a process that provides a specified service to any other process, a *client*. The client-server model is *asymmetric*.

The server always exists on the network, passively waiting for requests. The server may become a client when it issues requests to another server.

Clients and Servers are communicating through **messages** or **remote procedure calls** (**RPC**).

Clients and Servers:
- are independent parallel processes
- may run on the same machine or on separate machines
- may run on dedicated machines (LAN services)

Ports: a 16 bits number that identifies an application. (0-1023 are restricted numbers)

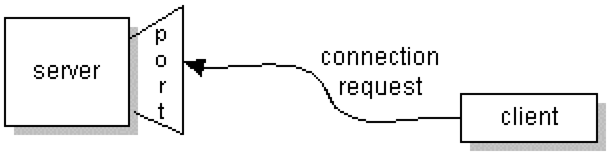
IP Address: 4 byte number (each number is in range from 0 to 255) – 32bits
It identifies the computer.

DNS: Domain Name System

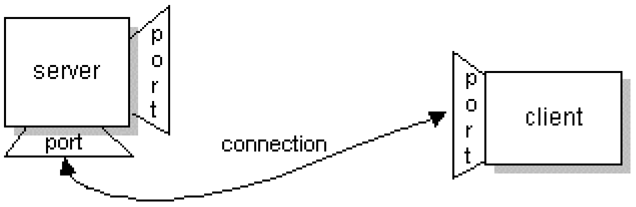
Socket: connection between two hosts, one end point of a two-way communication link between two threads on a network.
Sockets are an innovation of Berkeley Unix that allow the programmer to treat a network connection as another stream onto which bytes can be written and from which bytes can be read.
Sockets shield the programmer from low-level details of the network, such as media types, packet sizes, packet retransmission, network addresses and more.

What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.
On the client-side: The client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. Note that the socket on the client side is not bound to the port number used to rendezvous with the server. Rather, the client is assigned a port number local to the machine on which the client is running.
The client and server can now communicate by writing to or reading from their sockets.

Definition: A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, java.net includes the ServerSocket class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the Socket and ServerSocket classes.
Two classes can be used: **Socket** and **ServerSocket.**
The **java.net.Socket** class is Java's fundamental class for performing client-side TCP operations.