Lectures on distributed systems

Process Synchronization and Election Algorithms

Paul Krzyzanowski

Process Synchronization: mutual exclusion

Process synchronization is the set of techniques that are used to coordinate execution amongst processes. For example, a process may wish to run only to a certain point, at which it will stop and wait for another process to finish certain actions. A common resource (such as a device or a location in memory) may require exclusive access and processes have to coordinate amongst themselves to ensure that access is fair and exclusive. In centralized systems, it was common enforce exclusive access to shared code. Mutual exclusion was accomplished through mechanisms such as test and set locks in hardware and semaphores, messages, and condition variables in software. We will now revisit the topic of mutual exclusion in distributed systems. We assume that there is group agreement on how a critical section (or exclusive resource) is identified (e.g. name, number) and that this identifier is passed as a parameter with any requests.

Central server algorithm

The central server algorithm simulates a single processor system. One process in the distributed system is elected as the coordinator (Figure 1). When a process wants to enter a critical section, it sends a ***request*** message (identifying the critical section, if there are more than one) to the coordinator.

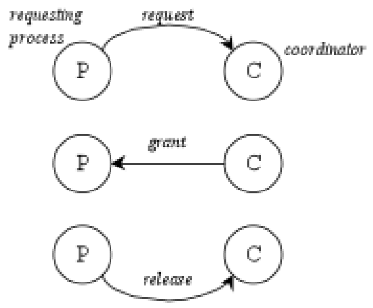

Figure 1. Centralized mutual exclusion

If nobody is currently in the section, the coordinator sends back a ***grant*** message and marks that process as using the critical section. If, however, another process has previously claimed the critical section, the server simply does not reply, so the requesting process is blocked.

When a process is done with its critical section, it sends a ***release*** message to the coordinator. The coordinator then can send a *grant* message to the next process in its queue of processes requesting a critical section (if any).

This algorithm is easy to implement and verify. It's fair in that all requests are processed in order. Unfortunately, it suffers from having a single point of failure. A process cannot distinguish between being blocked (not receiving a grant because someone else is in the critical section) and not getting a response because the coordinator is down. Moreover, a centralized server can be a bottleneck in large systems.

Ricart & Agrawala's Distributed mutual exclusion

Ricart & Agrawala came up with a distributed mutual exclusion algorithm in 1981. It requires the following:

- total ordering of all events in a system (e.g. Lamport's algorithm or others).
- messages are reliable (every message is acknowledged).

When a process wants to enter a critical section, it:

1. composes a message containing {mesage identifier(machine, proc#), name of critical section, timestamp).
2. sends a *request* message to all other processes in the group (may use reliable group communication).
3. wait until *everyone* in the group has given permission.
4. enter the critical section.

When a process receives a *request* message, it may be in one of three states:

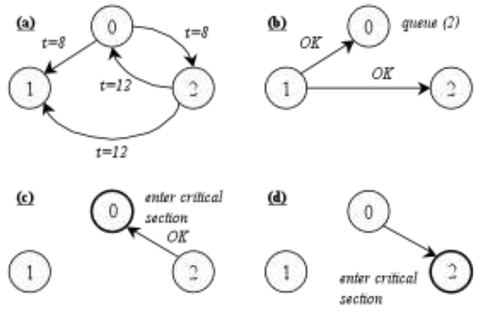

Figure 2. Distributed mutual exclusion

Case 1: The receiver is not interested in the critical section, send reply (*OK*) to sender.

Case 2: The receiver is in the critical section; do not reply and add the request to a local queue of requests.

Case 3: The receiver also wants to enter the critical section and has sent its request. In this case, the receiver compares the timestamp in the received message with the one that it has sent out. The earliest timestamp wins. If the receiver is the loser, it sends a reply (*OK*) to sender. If the receiver has the earlier timestamp, then it is the winner and does not reply. Instead, it adds the request to its queue.

When the process is done with its critical section, it sends a reply (*OK*) to everyone on its queue and deletes the processes from the queue.

As an example of dealing with contention, consider Figure 2. Here, two processes, 0 and 2, request access to the same resource (critical section). Process 0 sends its request with a timestamp of 8 and process 2 sends its request with a timestamp of 12 (Figure 2a).

Since process 1 is not interested in the critical section, it immediately sends back permission to both 0 and 1. Process 0, however, *is* interested in the critical section. It sees that process 2's timestamp was later than its own (12>8), so process 0 wins. It does not send a response to process 2 but instead queues the request from 2 (Figure 2b).

Process 2 is also interested in the critical section. When it compares its message's timestamp with that of the message it received from process 0, it sees that it lost (process 1's timestamp was earlier), so it replies with a permission message to process 0 and continues to wait for all other group members to give it permission to enter the critical section (Figure 2c).

Process 0 received a permission message from process 2 as well as other group members. Hence, process 0 received permission from the entire group and can enter the critical section. When process 0 is done with the critical section, it examines its queue of pending permissions, finds process 2 in that queue, and sends it permission to enter the critical section (Figure 2d). Now process 2 has received permission from everyone and can enter the critical section.

One problem with this algorithm is that a single point of failure has now been replaced with *n* points of failure. A poor algorithm has been replaced with one that is essentially *n* times worse. All is not lost. We can patch this omission up by having the sender always send a reply to a message... either an *OK* or a *NO*. When the request or the reply is lost, the sender will time out and retry. Still, it is not a great algorithm and involves quite a bit of message traffic but it demonstrates that a distributed algorithm is at least possible.

Token Ring algorithm

For this algorithm, we assume that there is a group of processes with no inherent ordering of processes, but that some ordering can be imposed on the group. For example, we can identify each process by its machine address and process ID to obtain an ordering. Using this imposed ordering, a logical ring is constructed in software. Each process is assigned a position in the ring and each process must know who is next to it in the ring (Figure 3).

- The ring is initialized by giving a **token** to process 0. The token circulates around the ring (process *n* passes it to (*n*+1)mod *ringsize*.
- When a process acquires the token, it checks to see if it is attempting to enter the critical section. If so, it enters and does its work. On exit, it passes the token to its neighbor.
- If a process isn't interested in entering a critical section, it simply passes the token along.

Only one process has the token at a time and it must have the token to work on a critical section, so mutual exclusion is guaranteed. Order is also well-defined, so starvation cannot occur. The biggest drawback of this algorithm is that if a token is lost, it will have to be generated. Determining that a token is lost can be difficult.

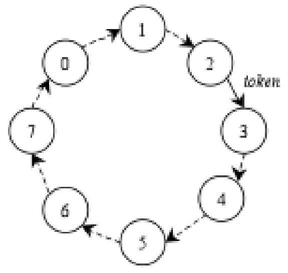

Figure 3. Token ring algorithm

Election algorithms

We often need one process to act as a coordinator. It may not matter which process does this, but there should be group agreement on only one. An assumption in election algorithms is that all processes are exactly the same with no distinguishing characteristics. Each process can obtain a unique identifier (for example, a machine address and process ID) and each process knows of every other process but does not know which is up and which is down.

Bully algorithm

The bully algorithm selects the process with the largest identifier as the coordinator. It works as follows:

1. When a process *p* detects that the coordinator is not responding to requests, it initiates an election:

    a. *p* sends an *election* message to all processes with higher numbers.

    b. If nobody responds, then *p* wins and takes over.

    c. If one of the processes answers, then *p*'s job is done.

2. If a process receives an *election* message from a lower-numbered process at any time, it:

    a. sends an OK message back.

    b. holds an election (unless its already holding one).

3. A process announces its victory by sending all processes a message telling them that it is the new coordinator.

4. If a process that has been down recovers, it holds an election.

Ring algorithm

The ring algorithm uses the same ring arrangement as in the token ring mutual exclusion algorithm, but does not employ a token. Processes are physically or logically ordered so that each knows its successor.

- If any process detects failure, it constructs an election message with its process I.D. (e.g. network address and local process I.D.) and sends it to its successor.

- If the successor is down, it skips over it and sends the message to the next party. This process is repeated until a running process is located.

- At each step, the process adds its own process I.D. to the list in the message.

Eventually, the message comes back to the process that started it:

1. The process sees its ID in the list.

2. It changes the message type to *coordinator*.

3. The list is circulated again, with each process selecting the highest numbered ID in the list to act as coordinator.

4. When the *coordinator* message has circulated fully, it is deleted.

Multiple messages may circulate if multiple processes detected failure. This creates a bit of overhead but produces the same results.

References

Distributed Systems: Concepts and Design, G. Coulouris, J. Dollimore, T. Kindberg, (c)1996 Addison Wesley Longman, Ltd., pp. 300-309 (section 10.4)

Distributed Operating Systems, Andrew Tanenbaum, ©; 1995 Prentice Hall.