Today’s Topics - Coordination and Agreement

Chapter 12.

- Distributed Mutual Exclusion. 12.2
- Elections 12.3
- Consensus 12.5

All figures are in the book “Distributed Systems Concepts and Design” by Coulouris, Dollimore and Kindberg
Distributed Mutual Exclusion

- As in operating systems we some want mutual exclusion: that is we don’t want more than one process accessing a resource at a time.

- In operating systems we solved the problem by using either using semaphores (P and V or by using atomic actions.)

- In a distributed system we want to be sure that we can access a resource and that nobody else is accessing the resource.

- A critical section is a piece of code where we want mutual exclusion.
Requirements for Mutex Algorithms

Safety  At most one process may execute in the critical section at a time.

Liveness  Requests to enter and exit the critical section eventually succeed.

The liveness condition implies freedom from both deadlock and starvation. A deadlock would involve two or more of the processes becoming stuck indefinitely. Starvation is indefinite postponement of entry for a process that has requested it.
Further Requirements

→ ordering  Access is granted in the happened before relation.

This is desirable so that process can coordinate their actions. A process might be waiting for access and communicating with other processes.
Performance criteria

**bandwidth** The bandwidth consumed, which is proportional to the number of messages sent in each entry or exit to the C.S.

**client delay** The client delay incurred by a process entering or exiting the C.S.

**throughput** The algorithm’s effect upon the *throughput* of the system.
The Central server Algorithm

The central server algorithm can be seen as a *token* based algorithm. The system maintains one token and make sure that only one process at a time gets that token. A process has to not enter the C.S. if it doesn’t have a token.

- A process request to enter the C.S. by asking the central server.
- The central server only grants access to one person at a time.
- The central server maintains a queue of requests and grants them in the order that people sent them.
- Main disadvantage: Single point of failure.
- Main advantage: Lower overhead in communication.
Server managing a mutual exclusion token for a set of processes
Token ring based algorithm

- Arrange them in a logical ring.
- Pass the token around.
- If you get the token and you don’t want to enter the C.S. pass the token on otherwise keep hold of it.
- Disadvantage: Maximum message delay equal size of the ring minus 1.
A ring of processes transferring a mutual exclusion token
Lamport Timestamps

- Remember that Lamport timestamps give a total order that everybody can agree on.
- Basic idea of Ricart and Agrawala’s algorithm. When ever somebody wants to enter the C.S. it multicasts to everybody a message saying I want to enter.
- When it receives Ok messages from everybody else then it enters.
- If the process wants to enter the C.S. and receives a request it compares timestamps and waits if the received message has an earlier timestamp. (If timestamps are identical then use process i.d. as a tiebreaker). Main disadvantage large communication overhead.
Multicast synchronization

Diagram showing a network of three processes labeled $P_1, P_2, P_3$, with messages labeled 34 and 41 exchanged and replies indicated.
Maekawa’s Voting Algorithm

- Maekawa’s algorithm makes it possible for a process to enter the critical section by obtaining permission from only a subset of the other processes.

- This is done by associating a voting set $V_i$ with each process $p_i$. The sets $V_i$ are chosen such that:
  - $p_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$
  - $|V_i| = K$ each voting set is of the same size.
  - Each process $p_j$ is contained in $M$ of the voting set $V_i$. 
Maekawa’s Voting Algorithm

- To obtain entry into the critical section a process sends a request message to all $K$ members of $V_i$.
- It can enter when it has received all $K$ replies.
- When it has finished it sends a release message.
- A process reply unless either its state is HELD or it has already replied since it last received a release message.
- Otherwise, it queues the request message. When it receives a release message, it removes the head of its queue and sends a reply.
Leader Election Algorithms

Some of the algorithms for clock synchronisation required a leader process. We want an algorithm that always gives one leader to a group of processes. There are two on the market:

- The Bully Algorithm
- The Ring Algorithm
The Bully Algorithm

- Each process has a number.
- If a process, $P$ notices that there is no leader or it receives an ELECTION message then it sends an ELECTION to all higher number processes.
- If one of the higher-up answers, it takes over then $P$ can stop otherwise $P$ takes over.
The Bully Algorithm

(a) Previous coordinator has crashed
(b) Coordinator
(c) Election

(d) OK
(e) Coordinator
The Ring Algorithm

We organize the processes in a logical ring. The process with the highest identifier will become leader.

- Initially every process is marked as a non-participant.
- Any process begins an election by marking itself as participant and sending on its id in an election message.
- On forwarding an election message the process marks itself as a participant.
- When a process receives is own id, it becomes the coordinator and sends an elected message to its neighbour.
A ring-based election in progress
Consensus

- How do a group of processors come to a decision?
- Example suppose a number of generals want to attack a target. They know that they will only succeed if they all attack. If anybody backs out then it is going to be a defeat.
- The example becomes more complicated if one of the generals becomes a traitor and starts to try and confuse the other generals. By saying yes I’m going to attack to one and no I’m not to another.
- How do we reach consensus when there are Byzantine failures? It depends on if the communication is synchronous or asynchronous.
Byzantine Generals in a synchronous system

Problem:

- A number of processes.
- Private synchronous channels between them.
- Process try and reach consensus or agree on a value.
- Goal given a number of faulty processes is it possible to distinguish between correct communication and faulty communication?
Byzantine Generals - Requirements

Byzantine generals problem differs from consensus.

- Byzantine: A process supplies a value that the others are to agree upon. Consensus: Each process proposes a value.

The requirements are:

- **Termination**: Eventually each correct process sets its decision variable.

- **Agreement**: The decision value of all correct processes is the same.

- **Integrity**: If the commander is correct, then all correct processes decide on the commanders value.
Impossibility with three processes

Situation a commander with two lieutenants. The commander is trying to send an attack order to both. If one of the lieutenants is treacherous then:

```
Commander
   /   \
/     \ \
| attack | attack |
|        |
Lieutenant 1           Traitor
   /   \
/     \  \
| retreat | 
```
Impossibility with three processes

If the commander is treacherous then:

Lieutenant 1 does not have anyway of distinguishing between the first and second situations. So there is no solution to the Byzantine general problem with 2 ok processes and 1 traitor.
Extending the result

- You can show that there is no solution to the problem if $N$ the number of processes and $f$ the number of faulty processes satisfies

$$N \leq 3f$$

- You do this by taking a supposed solution with a more than a third of the processes faulty and then turn this into a solution for 1 faulty and 2 correctly working generals, by getting the three generals to simulate the solution for then $N \leq 3f$ situation, by passing more messages.
Solution where $N > 3f$

The general solution for $N \geq 3f + 1$ is too large. Instead we present the solution with 1 faulty process and 3 correct processes.

- The correct generals reach agreement in two rounds of messages:
  - In the first round, the commander sends a value to each of the lieutenants.
  - In the second round, each of the lieutenants sends the value it received to its peers.
- A lieutenant receives a value from the commander, plus $N - 2$ values from its peers.
Solution with one Faulty process

- The correct lieutenants apply a simple majority function to the set of values they receive to get the correct value.
- If there is no majority the majority function will return $\perp$.
- Also handles faulty processes that omit to send a message with $\perp$.
- In the general case ($f \geq 1$) operates over $f + 1$ rounds. Costly algorithm in terms of number of messages.
Four Byzantine generals

Faulty processes are shown in blue tint.
Impossibility in asynchronous system

- The discussion so far has relied on message passing being synchronous.
- Messages pass in rounds.
- In an asynchronous system you can’t distinguish between a late message and a faulty process.
- It is impossible to solve the Byzantine general problem in an asynchronous system.