Chapter 1: Communication in Distributed Systems

Chapter 2: Basic Principles in Distributed Systems

Chapter 3: Coordination
  • Time and Synchronization
  • Coordination Algorithms
  • Distributed Transactions

Chapter 4: Fault Tolerance and Performance Improvements

Chapter 5: Middleware

3.2: Coordination Algorithms
  • Mutual Exclusion
  • Voting
  • Election of Coordinators
  • Consensus Problems
Mutual Exclusion

- Multiple processes sharing data need to be protected against parallel accesses

> Critical regions

- In single-processor systems: semaphores, mutexes, monitors, ...
- Other concepts are needed in distributed systems

assumption: message delivery is reliable and processes will not fail, for simplicity there is only one critical region

- Conditions for evaluation:
  - Fairness
  - Starvation
  - Deadlocks
  - Robustness
  - Performance: bandwidth, delay, ...
A Centralised Algorithm

Straightforward: *simulating a single-processor system*

- One process becomes a *coordinator* which controls access operations and keeps a queue for processes which want to enter the critical region.
- A process which wants to enter a critical region, asks the coordinator for permission.
- If there is no more process in the critical region, the requestor gets an OK from the coordinator.
- Otherwise, the request is queued. When the critical region is left by the holding process, the coordinator takes the oldest request from its queue and sends back an OK to the sender of the request.

![Diagram of the algorithm](image)
A Centralised Algorithm

**Advantages:**

+ Guaranteed exclusive access by centralised control
+ Fair algorithm guaranteeing order of requests
+ No starvation of single processes
+ Easy to implement
+ Only three messages per entry in the critical region

**Disadvantages:**

- Coordinator becomes a single point of failure and a performance bottleneck
- It is hard to see, if the coordinator is blocked or crashed (in this case, a new coordinator has to be determined)
A Distributed Algorithm

Ricart/Agrawala: assume that there is a total ordering of all events in a system (e.g. using Lamport's algorithm)

• Process $P$ wants to enter a critical region; it sends a message containing its process number and timestamp to all processes

• A process $Q$ receiving the message of $P$ distinguishes
  - If $Q$ is not in the critical region and wants not to enter, it answers OK
  - If $Q$ is in the critical region, it queues the request until it leaves the region
  - If $Q$ wants to enter the critical region, it compares $P$'s timestamp with its own (sent in an own message to all processes). The lower timestamp wins, $Q$ answers with OK resp. queues the request

• A process receiving OK from all processes can enter the critical region

• When leaving the critical region, an OK is sent to all processes with requests in its queue
Example for Distributed Algorithm

- No deadlocks, no starvation, but $2(n-1)$ messages for $n$ processes
- And: multiple-point-of-failure: not responding means denying permission...
- And: if no group communication is available, each process must manage groups...
- And: overloaded processes are performance bottlenecks for the whole mechanism...
- At the end: slower, more complex, less robust than the centralised algorithm...
Ricart and Agrawala’s algorithm

On initialization

\[ state := \text{RELEASED}; \]

To enter the section

\[ state := \text{WANTED}; \]
Multicast request to all processes;
\[ T := \text{request's timestamp}; \]
\[ \text{Wait until (number of replies received} = (N - 1)); \]
\[ state := \text{HELD}; \]

On receipt of a request \(<T_i, p_i>\) at \(p_j\) \((i \neq j)\)

if \((state = \text{HELD or (state} = \text{WANTED and (} T, p_j) < (T_i, p_i))\)
then
\[ \text{queue request from } p_i \text{ without replying}; \]
else
\[ \text{reply immediately to } p_i; \]
end if

To exit the critical section

\[ state := \text{RELEASED}; \]
reply to any queued requests;
A Token Ring Algorithm

Completely different algorithm:

- Construct a logical ring from all processes, with each process knows the successor
- On initializing, process 0 gets a token, which circulates around the ring
- Having the token, a process is allowed once to enter a critical region. After leaving it, the token is passed on

- Advantages: no synchronization delay, no starvation, no deadlocks
- Problems: no real-world ordering of entry requests, token loss, token duplication, process crashes, maintenance of the current ring configuration
Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>2 ((n - 1))</td>
<td>2 ((n - 1))</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to (∞)</td>
<td>0 to (n - 1)</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

The centralised algorithm seems to be best...
Maekawa's Voting Algorithm

It is not necessary to have a permission from all processes before entering a critical region; permissions are only needed from subsets (which have to have overlaps)

Associate a voting set $V_i$ with each process $p_i$, where $V_i \subseteq \{p_1, p_2, ..., p_N\}$ so that for all $i, j = 1, 2, ..., N$

- $p_i \in V_i$
- $V_i \cap V_j \neq \emptyset$ (no disjunctive sets)
- $|V_i| = K$ (each set has same size)
- each $p_j$ is contained in $M$ of the voting sets

(optimal solution with minimal $K$: $K \sim N^{-1/2}$, $M = K$)

[non-trivial problem: how to calculate the optimal sets?]

It works, because by $V_i \cap V_j \neq \emptyset$ there is one process in the intersection only voting for a process in one of the subsets.

... but: deadlocks are possible! (but adaptation of the algorithm is possible)

Only $3N^{-1/2}$ messages, which is smaller than $2(N-1)$ for $N > 4$
Maekawa’s Algorithm – Part 1

On initialization
state := RELEASED;
voted := FALSE;

For \( p_i \) to enter the critical section
state := WANTED;
Multicast request to all processes in \( V_i \);
Wait until (number of replies received = \( K \));
state := HELD;

On receipt of a request from \( p_i \) at \( p_j \)
if \( (\text{state} = \text{HELД} \text{or voted} = \text{TRUE}) \)
then
queue request from \( p_i \) without replying;
else
send reply to \( p_i \);
\text{voted} := \text{TRUE};
end if

All processes in \( V_i \) have answered
\( p_j \) holds token or has granted access to another process
\( p_j \) votes for \( p_i \) to enter the critical region
Maekawa’s Algorithm – Part 2

For $p_i$ to exit the critical section
state := RELEASED;
Multicast release to all processes in $V_i$;

On receipt of a release from $p_i$ at $p_j$
if (queue of requests is non-empty)
then
  remove head of queue - from $p_k$, say;
  send reply to $p_k$;
  voted := TRUE;
else
  voted := FALSE;
end if

On leaving the critical region, all other processes in the set are informed
Now critical region is empty; give permission to next process in the queue
Raymond’s Token-based Approach

Alternative to Maekawa’s Algorithm:

- Processes are organized as an un-rooted $n$-ary tree.
- Each process has a variable $\text{HOLDER}$, which indicates the location of the access privilege relative to the node itself.

$$\begin{align*}
\text{HOLDER}_A &= D \\
\text{HOLDER}_B &= A \\
\text{HOLDER}_C &= A \\
\text{HOLDER}_D &= E \\
\text{HOLDER}_E &= \text{self} \\
\text{HOLDER}_F &= D
\end{align*}$$

- Each process keeps a $\text{REQUEST}_Q$ that holds the names of neighbors or itself that have sent a REQUEST, but have not yet been sent the privilege message in reply.
Raymond’s Token-based Approach

- **To enter the critical region (CR):**
  - Enqueue self. If a request has not been sent to HOLDER, send a request
- **Upon receipt of a REQUEST message from neighbor x:**
  - If x is not in queue, enqueue x
  - If self is a HOLDER and still in the CR, it does nothing further
  - If self is a HOLDER but exits the CR, then it gets the oldest requester from REQUEST_Q, sets it to be the new HOLDER, and sends PRIVILEGE to it
- **Upon receipt of a PRIVILEGE message:**
  - Dequeue REQUEST_Q and set the oldest requester to be HOLDER
  - If HOLDER = self, then hold the PRIVILEGE and enter the CR
  - If HOLDER = some other process, send PRIVILEGE to HOLDER. In addition, if the REQUEST_Q is non-empty, send REQUEST to HOLDER as well
- **On exiting the CR:**
  - If REQUEST_Q is empty, continue to hold PRIVILEGE
  - If REQUEST_Q is non-empty, then dequeue REQUEST_Q and set the oldest requester to HOLDER, and send PRIVILEGE to HOLDER. In addition, if the (remaining) REQUEST_Q is non-empty, send REQUEST to HOLDER as well
Example: Raymond’s Token-based Approach

1. Initial State

2. Req. by P8

3. Req. by P2

4. Req. by P6

5. P1 passes T to P3

6. P3 passes T to P8

7. P8 passes T to P3, to P6

8. Req. by P7

9. P6 passes T to P3, to P1
Election Algorithms

- Many distributed algorithms need a *coordinator/initiator/*...
- It does not matter which process takes over the special role
- Thus: *electing* a coordinator
- In general:
  - Locating the process with the best election value, very often the highest process number (network address, ...); it will become the coordinator
  - Any process can call for an election. The result of an election does not depend on the initiating process
  - There could be concurrent calls for the same election

- **Goal of election algorithm**

  to ensure that after an election started, all processes agree on the same process to be the coordinator

- Assumption of algorithms: each process knows the election values (in the following: the process numbers) of all processes, but it does not know which of the processes are up and running
Voting vs. Election

- Maekawa’s algorithm is an example of voting
  - Processors are not aware of the result of their vote
  - Failure does not figure in the voting

- Elections are initiated to select a coordinator or to grant special privileges to a process
  - All processes are informed of the result
  - Election is usually initiated when processor failure occurs
The Bully Algorithm

**Bully Algorithm**: when any process $P$ notices that the current coordinator does not respond, it initiates the following steps:
1. $P$ sends an ELECTION message to all processes with higher numbers
2. If no one responds, $P$ wins and becomes the new coordinator
3. If one of the higher-number processes answers, it takes over the election

Example:
- Process 4 mentions that Coordinator 7 is not responding. It sends an election message to all higher-numbered processes.
- The living processes 5 and 6 respond, 4 can stop his job. Someone higher-numbered is responsible.
- process 0 is crossed out indicating it is a crashed process.
The Bully Algorithm

- Process 5 and 6 continue by sending election messages to all higher-numbered processes.

- Process 5 receives an answer and can stop the election. Process 6 gets no answer (usage of timeouts for considering process failures) and knows that it is the highest-numbered living process.

- Process 6 becomes the new coordinator and pushes this information to all processes.

Note: if a process comes back which previously has been down, it starts an election because it does not know about the current coordinator.
A Ring Algorithm

Alternative approach: seeing all processes as a ring (without token!)

**Assumption**: each process knows its successor

1. A process noticing that the coordinator is down sends an ELECTION message containing its process id to its successor
2. If the successor is down, the sender skips this process and looks for the next working process in the ring
3. Each process receiving the message adds its own process id to the message
4. A process receiving a message containing its own number, stops the election. The message type is changed to COORDINATOR and circulates ones more. The process with the highest number contained in the message is the new coordinator. All other processes are the new ring members.

Notice: it does not violate the election process, if there are two or more elections in parallel
Consensus Problems

*Needed in some situations:* Agreement of several processes on the 'correct' value of some data after one or more processes have proposed what the value should be (e.g. ‘go’ or ‘abort’)

*Problems:*
- A communication system never is completely reliable: loss/distort of messages
- Processes can be faulty or even malicious

Even in such cases, an agreement should be possible.

Usual example for explanation is the problem of **Byzantine Generals**:

Several byzantine generals surround a foreign camp and think about an attack. An attack can only be successful if all forces attack jointly. The generals exchange messages by (unreliable) riders. The riders can be caught (message loss), additionally some generals could be traitors (faulty processes).
Definition of Consensus Problems

- Every process $p_i$ begins in an *undecided* state and proposes a value $v_i$ from a set $D$
- The processes communicate with each other by exchanging their values
- Then each process sets a *decision variable* $d_i$. It enters the *decided* state in which values do no more change.

Requirements on a consensus algorithm:

- **Termination**: each correct process sets the decision variable
- **Agreement**: the decision values of all correct processes are the same
- **Integrity**: if all correct processes propose the same value, this value is chosen in the *decided* state

If there are no faulty processes and message losses, the solution is simple: choose the value which was proposed by most processes.
Different Consensus Problems

Consensus problem

Agree on a value fulfilling the requirements given above. *Notice: solving consensus is equivalent to solving reliable and totally ordered multicast.*

Byzantine generals

Three or more generals have to agree to attack or to retreat. One commander issues the order. The other generals decide to attack or to retreat. Each of the generals (including the commander) can be 'faulty'. In this problem, the *integrity* requirement differs from the general formulation: if the commander is correct, then all correct generals agree on the value he had proposed.

Interactive consistency

Agreement on a *vector* of values, one for each participating process ('decision vector'). Now the *integrity* requirement is: if $p_i$ is correct, then all correct processes agree on $v_i$ as the $i$th component of the vector.
Consensus in a Synchronous System

Assumption: up to $f$ of the $N$ processes can be faulty (crash failures)

- Each process collects proposed values from the other processes
- Algorithm proceeds in $f+1$ rounds

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

\textit{On initialization}
\begin{align*}
\text{Values}_i^1 & := \{ v_i \}; & \text{Values}_i^0 & := \{ \};
\end{align*}

\textit{In round } $r$ ($1 \leq r \leq f + 1$)
\begin{align*}
\text{B-multicast}(g, \text{Values}_i^r \setminus \text{Values}_i^{r-1}); & \text{ // Send only values that have not been sent} \\
\text{Values}_i^{r+1} & := \text{Values}_i^r;
\end{align*}

\textbf{while} (in round $r$)
\begin{align*}
\{ & \text{On B-deliver}(V_j) \text{ from some } p_j \\
\text{Values}_i^{r+1} & := \text{Values}_i^{r+1} \cup V_j;
\}
\end{align*}

\textit{After } $(f + 1)$ rounds
\begin{align*}
\text{Assign } d_i = \text{minimum}(\text{Values}_i^{f+1});
\end{align*}
More complicated: Byzantine Generals

*Given*: 
- \( N \) processes (generals) with at most \( f \) faulty processes
- Timeout to detect absence of messages (but it is not clear if the sender is crashed or simply not answering, i.e. faulty)
- Private communication channels between pairs of processes

*Goal*: each correct process \( p_i \) computes a vector \( x_i = (x_{i1}, x_{i2}, ..., x_{in}) \) with
- \( x_{ir} = v_r \) if \( p_r \) is correct
- \( x_{ir} = x_{ik} \) if \( p_r \) and \( p_k \) are correct

All known solutions have the following characteristics:
1. An algorithm only terminates correctly if \( N \geq 3f+1 \)
2. The worst case for agreeing is \( f+1 \) message delivery times
3. Large number of exchanged messages: each process has to collect all messages and execute the algorithm (it can not trust other processes)
Impossibility for $N < 3f+1$

Example: $N = 3$, $f = 1$

Faulty processes are shown shaded

Two cases in which $p_2$ can not decide which value is correct
Example: \( N = 4 \)

*Simplest example* for byzantine generals: \( N=4, f=1 \):

- Algorithm consists of two rounds
  - round 1: commander sends own value to the other three processes
  - round 2: other processes send values collected in the first round to the other processes
- Information of the faulty process can be wrong or even not send (then it can be set to a random value)
- Compute vector \( x_i = (x_{i1}, x_{i2}, x_{i3}, x_{i4}) \) [for correct process \( p_i \)]
  - \( x_{ii} = v_i \)
  - \( x_{ir} (r \neq i) \): at least two of the three incoming values are equal; set \( x_{ir} \) to this value (*majority decision*).
Four Byzantine Generals

Faulty processes are shown shaded
Example: Byzantine Generals for Interactive Consistency

Step 1

Step 2

Step 3

<table>
<thead>
<tr>
<th>Step 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>&lt;1,?,3,4&gt;</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>&lt;1,?,3,4&gt;</td>
</tr>
</tbody>
</table>
Conclusion

When implementing distributed software, several services can be helpful to support the cooperation between the software components:

- Controlling access to shared resources – how to achieve mutual exclusion for shared resources
- Sometimes, one component in needed to become a coordinator – how to determine which component it should be
- How to come to an agreement between redundant components if fault tolerance is needed

→ Such services should be implemented by a middleware to give a basis for software development