Coordination

CMPS 4760/6760: Distributed Systems
Overview

- Distributed Mutual Exclusion (15.2)
- Leader Election (15.3)
- Group communication (6.2, 15.4)
- Consensus (15.5)
Distributed Mutual Exclusion

- **Critical Section**: a resource shared by multiple processes
  - Operations performed on the replicated table should appear atomic
  - Medium Access Control in Ethernet

- **Distributed Mutual Exclusion**: message passing only
Problem Specification

- **Safety**: At most one process can execute in the critical section (CS) at a time
  - Safety – nothing “bad” will happen

- **Liveness**: Every request for the critical section is eventually granted
  - Liveness – something “good” will eventually happen

- **Fairness**: Different requests must be granted in the order they are made
A simple centralized solution

- A server serves as the coordinator for the CS
- Any process that needs to access the CS sends a request to the coordinator
- The coordinator puts requests in a queue in the order it receives them and grants permission to the process that is at the head of the queue
- When a process exits the CS, it sends a release message to the coordinator
A simple centralized solution

- Assuming no faults, safety and liveness satisfied, but not fairness (why?)

- Performance
  - Entering the CS takes 2 messages
  - Exiting the CS takes 1 message
  - Synchronization delay between one-process exiting the CS and the other process entering it: a round-trip time
A ring-based algorithm

- Arrange processes into a logical ring
- A token passes through the processes in a single direction
- A process can access the CS when it receives the token. It forwards the token to its neighbor when it exits the CS
- If a process receives the token and does not need to access the CS, it immediately forwards the token to its neighbor
A ring-based algorithm

- Assuming no faults, safety and liveness satisfied, but not fairness (*why?*)

- Performance
  - Processes send and receive messages around the ring even when no one requires entry to the CS
  - Delay to enter the CS: $0 \sim N$ messages
  - Synchronization delay: $1 \sim N$ messages
Assumptions

- No faults in the system: both processes and communication links are reliable
- A process that is granted access to the critical section eventually releases it (**cooperation**)
- A single critical section (CS)
- A completely connected graph, so that every process can directly communicate with every other process in the system
Some formal definitions

\( \text{req}(s) = \text{true} \) iff the process has requested the CS and has not yet released it.

\( \text{cs}(s) = \text{true} \) iff the process has the permission to enter the CS.

**Cooperation:** \( \text{cs}(s) \Rightarrow (\exists t: s < t \land \neg \text{req}(t)) \)

**Safety:** \( (s || t) \land (s \neq t) \Rightarrow \neg (\text{cs}(s) \land (\text{cs}(t))) \)

**Liveness:** \( \text{req}(s) \Rightarrow (\exists t: s \leq t \land \text{cs}(t)) \)
Some formal definitions

\[\text{next}_\text{cs}(s) \triangleq \min\{t | s \leq t \land cs(t)\}\]

\[\text{req}\_\text{start}(s) \triangleq \text{req}(s) \land \neg\text{req}(\text{prev}(s))\]

**Fairness:** \((\text{req}\_\text{start}(s) \land \text{req}\_\text{start}(t) \land s \rightarrow t) \Rightarrow \text{next}_\text{cs}(s) \rightarrow \text{next}_\text{cs}(t)\)
Ricart and Agrawala’s Algorithm

- To request a resource, $P_i$ sends a **timestamped** message to all processes.
- On receiving a request from $P_j$:
  - $P_i$ sends an **okay** message if either $P_i$ is not interested in the CS or its own request has a higher timestamp value.
  - Otherwise, $P_j$ is kept in a pending queue.
- $P_i$ is granted the resource when it has requested the resource and it has received the **okay** message from every other process in response to its request message.
- To release a resource, $P_i$ sends **okay** to all the processes in the pending queue.
Ricart and Agrawala’s Algorithm

\( P_i :: \)

\( \text{var} \)

\( pendingQ: \) list of process ids initially \( \text{null} \);  
\( myts: \) integer initially \( \infty \);  
\( numOkay: \) integer initially 0;

\( \text{request:} \)

\( myts = \text{logical\_clock}; \)

send \( request \) with \( myts \) to all other processes

\( numOkay = 0; \)

Updated whenever an event happens, not shown in the algorithm
Ricart and Agrawala’s Algorithm (continued)

receive \( (u, \text{request}) \):
if \( u.\text{myts} < \text{myts} \) send \text{okay} to process \( u.\text{p} \);
else append(\text{pendingQ, p});

receive \( (u, \text{okay}) \):
\text{numOkay} = \text{numOkay} + 1;
if \( \text{numOkay} == N - 1 \) enter\_\text{critical\_section};

release:
\text{myts} = \infty;
send \text{okay} to all the processes in \text{pendingQ};
\text{pendingQ} = \text{null};
Ricart and Agrawala’s Algorithm (continued)

- $2(N - 1)$ messages per invocation of CS
  - $N - 1$ requests
  - $N - 1$ okay
- Or $N$ messages if there is hardware support for multicast
- Satisfies safety, liveness, and fairness

![Diagram of message communication among processes $p_1$, $p_2$, and $p_3$. Messages are labeled with 41 and 34.]
Maekawa’s algorithm

- First solution with a sublinear $O(\sqrt{N})$ message complexity.
- “Close to” Ricart-Agrawala’s solution, but each process is required to obtain permission from only a subset of peers.
Maekawa’s algorithm

- Each process \( i \) is associated with a subset \( V_i \). Divide the set of processes into subsets that satisfy the following conditions:
  
  a) \( i \in V_i \)
  
  b) \( V_i \cap V_j \neq \emptyset, \forall i,j \)

- Main idea: Each process \( i \) is required to receive permission from \( V_i \) only. Correctness requires that multiple processes will never receive permission from all members of their respective subsets.
Maekawa’s algorithm

- Each process $i$ is associated with a subset $V_i$. Divide the set of processes into subsets that satisfy the following conditions:
  
  a) $i \in V_i$
  b) $V_i \cap V_j \neq \emptyset$, $\forall i, j$
  c) $|V_i| = K$, $\forall i$
  d) Any $i$ is contained in $M V_i$’s
Maekawa’s algorithm

**Example.** Let there be seven processes $0, 1, 2, 3, 4, 5, 6$

- $V_0 = \{0, 1, 2\}$
- $V_1 = \{1, 3, 5\}$
- $V_2 = \{2, 4, 5\}$
- $V_3 = \{0, 3, 4\}$
- $V_4 = \{1, 4, 6\}$
- $V_5 = \{0, 5, 6\}$
- $V_6 = \{2, 3, 6\}$
Maekawa’s algorithm

\( P_i :: \)

**On initialization**
\[
\text{state} := \text{RELEASED}; \ \text{voted} := \text{FALSE};
\]

**To enter the critical section**
\[
\text{state} := \text{WANTED};
\]
Multicast timestamped request to all processes in \( V_i \);
Wait until (number of replies received = K);
\[
\text{state} := \text{HELD};
\]

**On receipt of a request from \( P_j \)**
\[
\text{if} (\text{state} = \text{HELD} \text{ or} \ \text{voted} = \text{TRUE})
\]
queue request from \( P_j \) without replying;
\[
\text{else}
\]
\[
\text{send reply to} \ P_j; \ \text{voted} := \text{TRUE};
\]

**To exit the critical section**
\[
\text{state} := \text{RELEASED};
\]
Multicast release to all processes in \( V_i \);

**On receipt of a release from \( P_j \)**
\[
\text{if} (\text{queue of requests is non-empty})
\]
remove the request with the lowest timestamp from the queue – from \( P_k \), say;
\[
\text{send reply to} \ P_k;
\]
\[
\text{voted} := \text{TRUE};
\]
\[
\text{else}
\]
\[
\text{voted} := \text{FALSE};
\]
Maekawa’s algorithm

- Safety is easily satisfied
  - Let \( i \) and \( j \) attempt to enter their Critical Sections. \( V_i \cap V_j \neq \emptyset \) implies there is a process \( k \in V_i \cap V_j \). Process \( k \) will never send reply to both. So it will act as the arbitrator and establishes safety.

- However, liveness and fairness are not
  - Assume \( 0, 1, 2 \) want to enter their critical sections.
    - From \( V_0 = \{0,1,2\} \), 0,2 send reply to 0, but 1 sends reply to 1;
    - From \( V_1 = \{1,3,5\} \), 1,3 send reply to 1, but 5 sends reply to 2;
    - From \( V_2 = \{2,4,5\} \), 4,5 send reply to 2, but 2 sends reply to 0;
    - Now, 0 waits for 1 (to send a release), 1 waits for 2 (to send a release), and 2 waits for 0 (to send a release). So deadlock is possible!
Maekawa’s algorithm

- To ensure liveness
  - Assume that the system is **Global FIFO**: Every process receives incoming messages in strictly increasing order of logical timestamps
  - Modified Maekawa’s algorithm

- Performance
  - $2K$ messages per entry to the CS
  - $K$ messages to exit CS

- Ideally $K \sim \sqrt{N}$
  - Let $N = (M - 1)K + 1$ and $M = K$, then $N = K(K - 1) + 1$
  - A set of $V_i$’s exits if $K - 1$ is a power $p^m$ for some prime $p$ (equivalent to finding a finite projective plane of $N$ points). E.g., $K = 3, 4, 5$

- A simple approximation using $\sqrt{N} \times \sqrt{N}$ grid: $K = 2\sqrt{N} - 1$
Overview

- Distributed Mutual Exclusion (15.2)
- Leader Election (15.3)
- Group communication (6.2,15.4)
- Consensus (15.5)
Lead Election

- Electing a Coordinator
  - E.g., centralized mutual exclusion

- Breaking symmetry
  - E.g., remove one of the nodes in the cycle to remove the deadlock
Problem Specification

- **Correctness**: assume each process has a unique identifier and a local variable \textit{leader} that defines the leader.
  
  - Safety: for all non-faulty and participant processes, \(\text{leader} = (P: \text{a particular non-faulty process with the highest id}) \text{ or null}\)
  
  - Liveness: all non-faulty processes eventually participate and \(\text{leader} \neq \text{null}\)

- **Performance**
  
  - Network bandwidth utilization: total number messages sent
  
  - Turnaround time: number of serialized message transmission times between the initiation and termination of a single run
Calling for an election

- Any process can call for an election (e.g., when it detects the leader has failed)
- A process can call for at most one election at a time
- Multiple processes are allowed to call an election simultaneously.
  - All of them together must yield only a single leader
- The result of an election should not depend on which process calls for it
Lead Election vs. Mutual Exclusion

- **Similarity:** whichever process enters the critical section becomes the leader

- **Differences**
  - Starvation/fairness is irrelevant in leader election
  - Exit from CS is unnecessary for leader election.
  - Leader needs to inform every active process about its identity
Chang-Roberts Algorithm

- Assumptions
  - Processes arranged in a logical ring
  - No failures
  - Asynchronous systems

- Main idea: the process with the maximum identifier gets elected as the leader
Chang-Roberts Algorithm

\[ P_i : \]

\[
\text{var} \\
\quad \text{myid: integer;}
\]
\[
\text{participant: boolean initially } false; \\
\text{leaderid: integer initially } null;
\]

To initiate election:
\[
\text{send } (\text{election, myid}) \text{ to } P_{i+1}; \\
\text{participant } = \text{true};
\]
Chang-Roberts Algorithm

Upon receiving a message (election, j):
   if (j > myid)
       send (election, j) to P_{i+1};
   else if (j == myid)
       send (leader, myid) to P_{i+1};
   else if (j < myid) ∧ ¬participant
       send (election, myid) to P_{i+1};
       participant = true;

Upon receiving a message (leader, j):
   leaderid = j;
   if (j ≠ myid) send (leader, j) to P_{i+1};
Chang-Roberts Algorithm

- Safety and liveness satisfied
- Message complexity
  - Worst case:
    - Every process is an initiator
    - *Election* message from $P_i$ travels $i$ processes
    - $N$ leader messages
    - $\sum_{i=1}^{N} i + N = O(N^2)$
  - Best case: $N - 1 + N + N = 3N - 1$
  - Average case: $O(N\log N)$ (see homework 2)
Chang-Roberts Algorithm

- Turnaround time
  - Assume only $P_1$ starts an election
  - $N - 1$ election messages to reach $P_N$
  - Another $N$ election message before $P_N$ announces its election
  - $N$ leader messages
  - turnaround time: $3N - 1$
Bully Algorithm

**Assumptions**

- Processes can crash, channels are reliable
- Synchronized systems: can detect process failures via timeouts
  - Timeout: $T = 2T_{\text{trans}} + T_{\text{process}}$
- A completely connected graph
- Each process knows other processes and their identifiers.
Bully Algorithm

Stage 1

Stage 2

Stage 3

Eventually.....

Stage 4
Bully Algorithm

- Any process $P$ can initiate an election (when it notices the leader has failed)
- $P$ sends election messages to all processes with higher IDs and awaits answers
  - If no answer messages, $P$ becomes leader and sends leader messages to all processes with lower IDs
  - If it receives an answer, it drops out and waits for a leader message (if no leader message, restart election after $T'$)
- If $P$ receives an election message
  - Immediately broadcast a leader message if it is the process with highest ID
  - Otherwise, returns an answer message and starts an election (unless it has begun one)
- If $P$ receives a leader message, it treats sender as the leader
Bully Algorithm

- Meets the liveness requirement (in synchronous systems)
- Meets the safety requirement if no process is replaced
- Performance – best case
  - the process with the second highest id notices the failure of the coordinator and elects itself.
  - $N - 2$ leader messages sent.
  - Turnaround time is one message transmission time.
Bully Algorithm

- Performance – worst case
  - the process with the lowest id detects the failure.
  - $N - 1$ processes altogether begin elections
  - Message complexity is $O(N^2)$
  - Turnaround time: see Homework 2