Recap: Consensus

• On a synchronous system
  – There’s an algorithm that works.
• On an asynchronous system
  – It’s been shown (FLP) that it’s impossible to guarantee.
• Getting around the result
  – Masking faults
  – Using failure detectors
  – Still not perfect
• Impossibility Result
  – Lemma 1: schedules are commutative
  – Lemma 2: some initial configuration is bivalent
  – Lemma 3: from a bivalent configuration, there is always another bivalent configuration that is reachable.

Why Mutual Exclusion?

• Bank’s Servers in the Cloud: Think of two simultaneous deposits of $10,000 into your bank account, each from one ATM.
  – Both ATMs read initial amount of $1000 concurrently from the bank’s cloud server
  – Both ATMs add $10,000 to this amount (locally at the ATM)
  – Both write the final amount to the server
  – What’s wrong?

• The ATMs need mutually exclusive access to your account entry at the server (or, to executing the code that modifies the account entry)

Mutual Exclusion

• Critical section problem
  – Piece of code (at all clients) for which we need to ensure there is at most one client executing it at any point of time.
• Solutions:
  – Semaphores, mutexes, etc. in single-node OS
  – Message-passing-based protocols in distributed systems:
    » enter() the critical section
    » AccessResource() in the critical section
    » exit() the critical section
• Distributed mutual exclusion requirements:
  – Safety – At most one process may execute in CS at any time
  – Liveness – Every request for a CS is eventually granted
  – Ordering (desirable) – Requests are granted in the order they were made

Mutexes

• To synchronize access of multiple threads to common data structures
  Allows two operations:
  ```java
  lock()
  ```
  ```java
  while true:    // each iteration atomic
  if lock not in use:
    label lock in use
    break
  unlock()    // label lock not in use
  ```
Semaphores

• To synchronize access of multiple threads to common data structures
• Semaphore S=1:
  – Allows two operations
  – wait(S) (or P(S)):
    while(1){ // each execution of the while loop is atomic
      if (S > 0)
        S--;
      break;
    }
  – signal(S) (or V(S)):
    S++;
  – Each while loop execution and S++ are each atomic operations

How Are Mutexes Used?

mutex L= UNLOCKED;
extern mutex L;

ATM1:
lock(L); // enter
// critical section
obtain bank amount;
add in deposit;
update bank amount;
unlock(L); // exit

ATM2
lock(L); // enter
// critical section
obtain bank amount;
add in deposit;
update bank amount;
unlock(L); // exit

Distributed Mutual Exclusion Performance Criteria

• Bandwidth: the total number of messages sent in each entry and exit operation.
• Client delay: the delay incurred by a process at each entry and exit operation (when no other process is in, or waiting)
  – (We will prefer mostly the entry operation.)
• Synchronization delay: the time interval between one process exiting the critical section and the next process entering it (when there is only one process waiting)
• These translate into throughput — the rate at which the processes can access the critical section, i.e., x processes per second.
• (these definitions more correct than the ones in the textbook)

Assumptions/System Model

• For all the algorithms studied, we make the following assumptions:
  – Each pair of processes is connected by reliable channels (such as TCP).
  – Messages are eventually delivered to reliable channels (such as TCP).
  – Processes do not fail (why?)
• Four algorithms
  – Centralized control
  – Token ring
  – Ricart and Agrawala
  – Maekawa

1. Centralized Control

• A central coordinator (master or leader)
  – Is elected (next lecture)
  – Grants permission to enter CS & keeps a queue of requests to enter the CS.
  – Ensures only one process at a time can access the CS
  – Has a special token per CS
• Operations (token gives access to CS)
  – To enter a CS Send a request to the coord & wait for token.
  – On exiting the CS Send a message to the coord to release the token.
  – Upon receipt of a request, if no other process has the token, the coord replies with the token; otherwise, the coord queues the request.
  – Upon receipt of a release message, the coord removes the oldest entry in the queue (if any) and replies with a token.

1. Centralized Control

• Safety, liveness, ordering?
• Bandwidth?
  – Requires 3 messages per entry + exit operation.
• Client delay:
  – one round trip time (request + grant)
• Synchronization delay
  – one round trip time (release + grant)
• The coordinator becomes performance bottleneck and single point of failure.
2. Token Ring Approach

- Processes are organized in a logical ring: \( p_i \) has a communication channel to \( p_{(i+1) \mod (n)} \).
- Operations:
  - Only the process holding the token can enter the CS.
  - To enter the critical section, wait passively for the token. When in CS, hold on to the token.
  - To exit the CS, the process sends the token onto its neighbor.
  - If a process does not want to enter the CS when it receives the token, it forwards the token to the next neighbor.

Features:

- Safety & liveness, ordering?
- Bandwidth: 1 message per exit
- Client delay: 0 to \( N \) message transmissions.
- Synchronization delay between one process’s exit from the CS and the next process’s entry is between 1 and \( N-1 \) message transmissions.

3. Ricart & Agrawala’s Algorithm

- Processes requiring entry to critical section multicast a request, and can enter it only when all other processes have replied positively.
- Messages requesting entry are of the form \( <T, p_i> \), where \( T \) is the sender’s timestamp (Lamport clock) and \( p_i \) the sender’s identity (used to break ties in \( T \)).

On initialization:

- \( \text{state} \Rightarrow \text{RELEASED} \); 
- To enter the section, \( \text{state} \Rightarrow \text{WANTED} \); 
- Multicast request to all processes; 
- \( T \Rightarrow \text{request’s timestamp} \); 
- Wait until \( \text{number of replies received} = (N-1) \); 
- \( \text{state} \Rightarrow \text{HELD} \); 

On receipt of a request \( <T_i, p_i> \) at \( p_j \):

- if \( \text{state} = \text{HELD} \) or \( \text{state} = \text{WANTED} \) and \( (T_j, p_j) < (T_i, p_i) \) 
  - then queue request from \( p_i \) without replying; 
  - else 
    - reply immediately to \( p_i \); 
    - and if 
      - To exit the critical section, \( \text{state} \Rightarrow \text{RELEASED} \); 
      - reply to any queued requests; 

CSE 486/586 Administrivia

- PA2 due this Friday. 
- Midterm on Wednesday (3/12)
Analysis: Ricart & Agrawala

- Safety, liveness, and ordering?
- Bandwidth:
  - \(2(N-1)\) messages per entry operation
  - \(N-1\) unicasts for the multicast request + \(N-1\) replies
  - \(N-1\) unicast messages per exit operation
- Client delay
  - One round-trip time
- Synchronization delay
  - One message transmission time

4. Maekawa's Algorithm

- Simple example

4. Maekawa's Algorithm

- Observation: no need to have all peers reply
- Only need to have a subset of peers as long as all subsets overlap.
- Voting set: a subset of processes that grant permission to enter a CS
- Voting sets are chosen so that for any two processes, \(p_i\) and \(p_j\), their corresponding voting sets have at least one common process.
  - Each process \(p_i\) is associated with a voting set \(v_i\) (of processes)
  - Each process belongs to its own voting set
  - The intersection of any two voting sets is non-empty
  - Each voting set is of size \(K\)
  - Each process belongs to \(M\) other voting sets

Maekawa’s Algorithm – Part 1

On initialization

\[\text{state} = \text{RELEASED};\]
\[\text{voted} = \text{FALSE};\]

For \(p_i\) to enter the critical section

\[\text{state} = \text{WANTED};\]
Multicast request to all processes in \(V_i\);
Wait until (number of replies received = \(K\));
\[\text{state} = \text{HELD};\]

On receipt of a request from \(p_i\) at \(p_j\)

\[\text{if (state = HELD or voted = TRUE)}\]
\[\text{then}
\]
\[\text{queue request from p_i, without replying;}
\]
\[\text{else}
\]
\[\text{send reply to p_i;}
\]
\[\text{voted = TRUE;}
\]
\[\text{end if}
\]

Continues on next slide

Maekawa’s Algorithm – Part 2

For \(p_i\) to exit the critical section

\[\text{state} = \text{RELEASED};\]
Multicast release to all processes in \(V_i\);
On receipt of a release from \(p_i\) at \(p_j\)
\[\text{if (queue of requests is non-empty)}\]
then

\[\text{remove head of queue – from p_j, say;}
\]
\[\text{send reply to p_j;}
\]
\[\text{voted = TRUE;}
\]
[else}
\[\text{voted = FALSE;}
\]
[\text{end if}
Maekawa’s Algorithm – Analysis

- Bandwidth: $2/N$ messages per entry, $\sqrt{N}$ messages per exit
  - Better than Ricart and Agrawala’s $(2(N-1)$ and $N-1$ messages)
- Client delay: One round trip time
  - Same as Ricart and Agrawala
- Synchronization delay: One round-trip time
  - Worse than Ricart and Agrawala
- May not guarantee liveness (may deadlock)
  - How?

Summary

- Mutual exclusion
  - Coordinator-based token
  - Token ring
  - Ricart and Agrawala’s timestamp algorithm
  - Maekawa’s algorithm

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).