Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for some of materials: Indy Gupta and Nikita Borisov
Logistics

• MP1 has been released.
  • Due on March 4th, 11:59pm.

• HW1 is due next week Monday.
  • You should be able to solve all questions by now (except 7c).
  • You should be able to solve 7c after the end of today's class.

• Academic Integrity:
  • Violation: copying from peers, from previous years’, from the web, etc.
Today’s agenda

- **Wrap up Multicast**
  - Chapter 15.4
  - Tree-based multicast and Gossip

- **Mutual Exclusion**
  - Chapter 15.2
Recap: Multicast

- Useful communication mode in distributed systems:
  - Writing an object across replica servers.
  - Group messaging.
  - ...

- Basic multicast (B-multicast): unicast send to each process in the group.
  - Does not guarantee consistent message delivery if sender fails.

- Reliable multicast (R-multicast):
  - Defined by three properties: integrity, validity, agreement.
  - If some correct process multicasts a message m, then all other correct processes deliver m (exactly once).
  - When a process receives a message ‘m’ for the first time, it re-multicasts it again to other processes in the group.
Recap: Ordered Multicast

- **FIFO ordering**: If a correct process issues multicast\((g,m)\) and then multicast\((g,m')\), then every correct process that delivers \(m'\) will have already delivered \(m\).

- **Causal ordering**: If multicast\((g,m) \rightarrow multicast(g,m')\) then any correct process that delivers \(m'\) will have already delivered \(m\).
  - Note that \(\rightarrow\) counts multicast messages delivered to the application, rather than all network messages.

- **Total ordering**: If a correct process delivers message \(m\) before \(m'\), then any other correct process that delivers \(m'\) will have already delivered \(m\).
Ordered Multicast

- **FIFO ordering**: If a correct process issues `multicast(g,m)` and then `multicast(g,m')`, then every correct process that delivers `m'` will have already delivered `m`.

- **Causal ordering**: If `multicast(g,m) \rightarrow multicast(g,m')` then any correct process that delivers `m'` will have already delivered `m`.
  - Note that \( \rightarrow \) counts multicast messages *delivered* to the application, rather than all network messages.

- **Total ordering**: If a correct process delivers message `m` before `m'` then any other correct process that delivers `m'` will have already delivered `m`.
Implementing total order multicast

• Basic idea:
  • Same sequence number counter across different processes.
  • Instead of different sequence number counter for each process.

• Two types of approach
  • Using a centralized sequencer
  • A decentralized mechanism (ISIS)
Implementing total order multicast

• Basic idea:
  • Same sequence number counter across different processes.
  • Instead of different sequence number counter for each process.

• Two types of approach
  • Using a centralized sequencer
  • A decentralized mechanism (ISIS)
ISIS algorithm for total ordering

1 Message
2 Proposed Seq
3 Agreed Seq
**ISIS algorithm for total ordering**

- Sender multicasts message to everyone.
- Receiving processes:
  - reply with *proposed* priority (sequence no.)
    - larger than all observed *agreed* priorities
    - larger than any previously proposed (by self) priority
  - store message in *priority queue*
    - ordered by priority (proposed or agreed)
  - mark message as undeliverable
- Sender chooses *agreed* priority, re-multicasts message id with agreed priority
  - maximum of all proposed priorities
- Upon receiving agreed (final) priority for a message ‘m’
  - Update m’s priority to final, and accordingly reorder messages in queue.
  - mark the message m as deliverable.
  - deliver any deliverable messages at front of priority queue.
Example: ISIS algorithm
How do we break ties?

- Problem: priority queue requires unique priorities.

- Solution: add process # to suggested priority.
  - priority.\(\text{id of the process that proposed the priority}\)
  - i.e., \(3.2 = \text{process 2 proposed priority 3}\)

- Compare on priority first, use process # to break ties.
  - \(2.1 > 1.3\)
  - \(3.2 > 3.1\)
Example: ISIS algorithm

[see .pptx file for animations]
Proof of total order with ISIS

- Consider two messages, \( m_1 \) and \( m_2 \), and two processes, \( p \) and \( p' \).
- Suppose that \( p \) delivers \( m_1 \) before \( m_2 \).
- When \( p \) delivers \( m_1 \), it is at the head of the queue. \( m_2 \) is either:
  - Already in \( p \)’s queue, and deliverable, so
    - \( \text{finalpriority}(m_1) < \text{finalpriority}(m_2) \)
  - Already in \( p \)’s queue, and not deliverable, so
    - \( \text{finalpriority}(m_1) < \text{proposedpriority}(m_2) \leq \text{finalpriority}(m_2) \)
  - Not yet in \( p \)’s queue:
    - same as above, since proposed priority > priority of any delivered message
- Suppose \( p' \) delivers \( m_2 \) before \( m_1 \), by the same argument:
  - \( \text{finalpriority}(m_2) < \text{finalpriority}(m_1) \)
  - Contradiction!
Ordered Multicast

• FIFO ordering
  • If a correct process issues multicast($g,m$) and then multicast($g,m'$), then every correct process that delivers $m'$ will have already delivered $m$.

• Causal ordering
  • If multicast($g,m$) $\rightarrow$ multicast($g,m'$) then any correct process that delivers $m'$ will have already delivered $m$.
  • Note that $\rightarrow$ counts multicast messages delivered to the application, rather than all network messages.

• Total ordering
  • If a correct process delivers message $m$ before $m'$, then any other correct process that delivers $m'$ will have already delivered $m$. 
More efficient multicast mechanisms

• Our focus so far has been on the application-level semantics of multicast.

• What are some of the more efficient underlying mechanisms for a B-multicast?
B-Multicast

Sender
B-Multicast using unicast sends

Sender

TCP/UDP packets
B-Multicast using unicast sends

Closer look at physical network paths.
B-Multicast using unicast sends

Sender

Redundant packets!
B-Multicast using unicast sends

Similar redundancy when individual nodes also act as routers (e.g. wireless sensor networks).

How do we reduce the overhead?
Tree-based multicast

Instead of sending a unicast to all nodes, construct a minimum spanning tree and unicast along that.
Tree-based multicast

A process does not directly send messages to \textit{all} other processes in the group.

It sends a message to only a subset of processes.
Tree-based multicast

A process does not directly send messages to all other processes in the group. It sends a message to only a subset of processes.

Closer look at the physical network.
Tree-based multicast

Also possible to construct a tree that includes network routers. **IP multicast!**
Tree-based multicast

What happens if a node fails?
Overhead of tree construction and repair.

Sender

TCP/UDP packets
Third approach: Gossip

Transmit to $b$ random targets.
Third approach: Gossip

Transmit to b random targets.

Other nodes do the same when they receive a message.
Third approach: Gossip

Transmit to b random targets.

Other nodes do the same when they receive a message.
Third approach: Gossip

No “tree-construction” overhead.
More efficient than unicasting to all receivers.
Also known as “epidemic multicast”.
Probabilistic in nature – no hard guarantees.
Good enough for many applications.
Third approach: Gossip

Used in many real-world systems:
• Facebook’s distributed datastore uses it to determine group membership and failures.
• Bitcoin uses it to exchange transaction information between nodes.
Multicast Summary

- Multicast is an important communication mode in distributed systems.

- Applications may have different requirements:
  - Basic
  - Reliable
  - Ordered: FIFO, Causal, Total
  - Combinations of the above.

- Underlying mechanisms to spread the information:
  - Unicast to all receivers.
  - Tree-based multicast, and gossip: sender unicasts messages to only a subset of other processes, and they spread the message further.
  - Gossip is more scalable and more robust to process failures.
Today’s agenda

• **Wrap up Multicast**
  - Chapter 15.4
  - Tree-based multicast and Gossip

• **Mutual Exclusion**
  - Chapter 15.2

• Goal: reason about ways in which different processes in a distributed system can safely manipulate shared resources.
Why Mutual Exclusion?

• **Bank’s Servers in the Cloud**: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate 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?
Why Mutual Exclusion?

• Bank’s Servers in the Cloud: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate 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
  • You lost $10,000!

• The ATMs need mutually exclusive access to your account entry at the server
  • or, mutually exclusive access to executing the code that modifies the account entry.
More uses of mutual exclusion

• Distributed file systems
  • Locking of files and directories

• Accessing objects in a safe and consistent way
  • Ensure at most one server has access to object at any point of time

• In industry
  • Chubby is Google’s locking service
Problem Statement for mutual exclusion

• **Critical Section Problem:**
  • Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.

• Each process can call three functions
  • `enter()` to enter the critical section (CS)
  • `AccessResource()` to run the critical section code
  • `exit()` to exit the critical section
Our bank example

ATM1:

```cpp
enter();
// AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
exit();
```

ATM2:

```cpp
enter();
// AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
exit();
```
Mutual exclusion for a single OS

• If all processes are running in one OS on a machine (or VM):
  • Semaphores
  • Mutexes
  • Condition variables
  • Monitors
  • …
Processes Sharing an OS: Semaphores

- Semaphore == an integer that can only be accessed via two special functions
- Semaphore \( S = 1; \) // Max number of allowed accessors.

\[
\text{wait}(S) \text{ (or } P(S) \text{ or } \text{down}(S)):
\begin{align*}
\text{while}(1) \{ & \quad // \text{each execution of the while loop is atomic} \\
\text{if } (S > 0) \{ & \\
\quad S--; & \quad \text{enter()} \\
\quad \text{break;} & \\
\} \\
\}
\]

\[
\text{signal}(S) \text{ (or } V(S) \text{ or } \text{up}(s)):\n\quad S++; \quad // \text{atomic}
\]

\text{Atomic} \text{ operations are supported via hardware instructions such as compare-and-swap, test-and-set, etc.}
Our bank example

ATM1:

enter();

// AccessResource()
obtain bank amount;
add in deposit;
update bank amount;

// AccessResource() end
exit();

ATM2:

enter();

// AccessResource()
obtain bank amount;
add in deposit;
update bank amount;

// AccessResource() end
exit();
Our bank example

Semaphore S=1; // shared

ATM1:

wait(S); //enter

// AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
signal(S); // exit

ATM2:

wait(S); //enter

// AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
signal(S); // exit
Mutual exclusion in distributed systems

• Processes communicating by passing messages.

• Cannot share variables like semaphores!

• How do we support mutual exclusion in a distributed system?
Mutual exclusion in distributed systems

• Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  • Central server algorithm
  • Ring-based algorithm
  • Ricart-Agrawala Algorithm
  • Maekawa Algorithm
Mutual Exclusion Requirements

• Need to guarantee 3 properties:
  • **Safety** (essential):
    • At most one process executes in CS (Critical Section) at any time.
  • **Liveness** (essential):
    • Every request for a CS is granted eventually.
  • **Ordering** (desirable):
    • Requests are granted in the order they were made.
System Model

• Each pair of processes is connected by reliable channels (such as TCP).

• Messages sent on a channel are eventually delivered to recipient, and in FIFO (First In First Out) order.

• Processes do not fail.
  • Fault-tolerant variants exist in literature.
Mutual exclusion in distributed systems

• Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  • Central server algorithm
  • Ring-based algorithm
  • Ricart-Agrawala Algorithm
  • Maekawa Algorithm
Central Server Algorithm

• Elect a central server (or leader)
• Leader keeps
  • A queue of waiting requests from processes who wish to access the CS
  • A special token which allows its holder to access CS

• Actions of any process in group:
  • enter()
    • Send a request to leader
    • Wait for token from leader
  • exit()
    • Send back token to leader
Central Server Algorithm

• Leader Actions:
  • On receiving a request from process $P_i$
    
    if (leader has token)
    
    Send token to $P_i$

    else
    
    Add $P_i$ to queue

  • On receiving a token from process $P_i$
    
    if (queue is not empty)
    
    Dequeue head of queue (say $P_j$), send that process the token

    else
    
    Retain token
Analysis of Central Algorithm

- Safety – at most one process in CS
  - Exactly one token
- Liveness – every request for CS granted eventually
  - With $N$ processes in system, queue has at most $N$ processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- Ordering:
  - FIFO ordering guaranteed in order of requests received at leader
  - Not in the order in which requests were sent or the order in which processes enter CS!
Analysis of Central Algorithm

- **Safety** – at most one process in CS
  - Exactly one token
- **Liveness** – every request for CS granted eventually
  - With $N$ processes in system, queue has at most $N$ processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- **Ordering:**
  - FIFO ordering guaranteed in order of requests received at leader
  - Not in the order in which requests were sent or the order in which processes call ‘enter’!
Analyzing Performance

Three metrics:

• **Bandwidth**: the total number of messages sent in each `enter` and `exit` operation.

• **Client delay**: the delay incurred by a process at each `enter` and `exit` operation (when no other process is in CS, or waiting)
  - *We will focus on the client delay for the enter 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). Measures of the *throughput* of the system.
Analysis of Central Algorithm

- **Bandwidth**: the total number of messages sent in each `enter` and `exit` operation.
  - 2 messages for enter
  - 1 message for exit

- **Client delay**: the delay incurred by a process at each `enter` and `exit` operation (when no other process is in CS, or waiting)
  - 2 message latencies or 1 round-trip (request + grant) on enter.

- **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)
  - 2 message latencies (release + grant)
Limitations of Central Algorithm

- The leader is the performance bottleneck and single point of failure.
Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm
Ring-based Mutual Exclusion

Currently holds token, can access CS

Token: ●
Ring-based Mutual Exclusion

Cannot access CS anymore

Here’s the token!

Token: ●
Ring-based Mutual Exclusion

Currently holds token, can access CS

Token: ⬤
Ring-based Mutual Exclusion

• $N$ Processes organized in a virtual ring
• Each process can send message to its successor in ring
• Exactly 1 token

• \texttt{enter()}
  • Wait until you get token
• \texttt{exit()} // already have token
  • Pass on token to ring successor

• If receive token, and not currently in \texttt{enter()}, just pass on token to ring successor
Analysis of Ring-based algorithm

• Safety
  • Exactly one token

• Liveness
  • Token eventually loops around ring and reaches requesting process (no failures)

• Ordering
  • Token not always obtained in order of enter events.
Analysis of Ring-based algorithm

• **Safety**
  • Exactly one token

• **Liveness**
  • Token eventually loops around ring and reaches requesting process (no failures)

• **Ordering**
  • Token not always obtained in order of enter events.
Analysis of Ring-based algorithm

• Bandwidth
  • Per enter, 1 message at requesting process but up to \( N \) messages throughout system.
  • 1 message sent per exit.
  • *Constantly consumes bandwidth even when no process requires entry to the critical section (except when a process is executing critical section).*
Analysis of Ring-based algorithm

• Client delay:
  • Best case: just received token
  • Worst case: just sent token to neighbor
  • 0 to \( N \) message transmissions after entering enter()

• Synchronization delay between one process’ exit() from the CS and the next process’ enter():
  • Best case: process in enter() is successor of process in exit()
  • Worst case: process in enter() is predecessor of process in exit()
  • Between 1 and \((N-1)\) message transmissions.

• Can we improve upon this \(O(n)\) client and synchronization delays?
Mutual exclusion in distributed systems

• Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  • Central server algorithm
  • Ring-based algorithm
  • Ricart-Agrawala Algorithm (next class)
  • Maekawa Algorithm
MPI: Event Ordering

- [https://courses.grainger.illinois.edu/ece428/sp2024/mps/mp1.html](https://courses.grainger.illinois.edu/ece428/sp2024/mps/mp1.html)
- Lead TA: Siddharth Lal

**Task:**
- Collect **transaction** events on distributed **nodes**.
- **Multicast** transactions to all nodes while maintaining **total order**.
- Ensure transaction **validity**.
- Handle **failure** of arbitrary nodes.

**Objective:**
- Build a decentralized multicast protocol to ensure total ordering and handle node failures.
MPI Architecture Setup

- Example input arguments for first node:
  ```
  ./mp1_node node1 config.txt
  ```
- config.txt looks like this:

```
3
node1 sp21-cs425-g01-01.cs.illinois.edu 1234
node2 sp21-cs425-g01-02.cs.illinois.edu 1234
node3 sp21-cs425-g01-03.cs.illinois.edu 1234
```
MPI Architecture Setup

node ID
config_file

node 1

node ID
config_file

node 2

node ID
config_file

node 3

node 1

node 2

node 3
MPI Architecture

TX A; TX B

node 1

TX C; TX D

node 2

TX E; TX F

node 3

Multicast protocol

TX A; TX C; TX B; TX E; TX F; TX D

Total ordering
Transaction Validity

- **DEPOSIT abc 100**: Adds 100 to account abc (or creates a new abc account)

- **TRANSFER abc -> def 75**: Transfers 75 from account abc to account def (creating if needed)

- **TRANSFER abc -> ghi 30**: Invalid transaction, since abc only has 25 left
Transaction Validity: ordering matters

DEPOSIT xyz 50
TRANSFER xyz -> wqr 40
TRANSFER xyz -> hjk 30
[invalid TX]

BALANCES xyz:10 wqr:40

DEPOSIT xyz 50
TRANSFER xyz -> hjk 30
TRANSFER xyz -> wqr 40
[invalid TX]

BALANCES xyz:20 hjk:30
Graph

• Compute the “processing time” for each transaction:
  • Time difference between when it was generated (read) at a node, and when it was processed by the last (alive) node.

• Plot the CDF (cumulative distribution function) of the transaction processing time for each evaluation scenario.
MP1: Logistics

• Due on March 4th.
  • Late policy: Can use part of your 168 hours of grace period accounted per student over the entire semester.

• You are allowed to reuse code from MP0.
  • Note: MP1 requires all nodes to connect to each other, as opposed to each node connecting to a central logger.

• Read the specification carefully. Start early!!