Distributed Systems

CS425/ECE428

Feb 15 2023

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta and Nikita Borisov
Today’s agenda

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

• **Mutual Exclusion**
  • Chapter 15.2
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 $m$. 
ISIS algorithm: failures

• What happens if sender fails while multicasting a message?

• What happens if sender fails while multicasting the final priority of a message?

• What happens if a process fails before sending the proposed priority for a message?

• What happens if a process fails after sending the proposed priority for a message?
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 causal order multicast

• Similar to FIFO Multicast
  • What you send with a message differs.
  • Updating rules differ.

• Each receiver maintains a vector of per-sender sequence numbers (integers)
  • Processes P1 through PN.
  • Pi maintains a vector of sequence numbers Pi[1…N] (initially all zeroes).
  • Pi[j] is the latest sequence number Pi has received from Pj.
Implementing causal order multicast

• **CO-multicast**\( (g,m) \) at **P**\(_j\):
  
  - set **P**\(_j\)[\(j\)] = **P**\(_j\)[\(j\)] + 1
  
  - piggyback entire vector **P**\(_j\)\([1\ldots N]\) with m.

  **B-multicast**\( (g,\{m, **P**\(_j\)\([1\ldots N]\)\}) \)

• **On B-deliver**\( (\{m, V[1\ldots N]\}) \) at **P**\(_i\) from **P**\(_j\):** If **P**\(_i\) receives a multicast from **P**\(_j\) with sequence vector \( V[1\ldots N] \), buffer it until both:

  1. This message is the next one **P**\(_i\) is expecting from **P**\(_j\), i.e.,
     
     \[ V[j] = **P**i[j] + 1 \]

  2. All multicasts, anywhere in the group, which happened-before m have been received at **P**\(_i\), i.e.,

     For all \( k \neq j \):
     
     \[ V[k] \leq **P**i[k] \]

When above two conditions satisfied,

  **CO-deliver**\( (m) \) and set **P**\(_i\)[\(j\)] = \( V[j] \)
Causal order multicast execution

Self-deliveries omitted for simplicity.
Causal order multicast execution

Deliver!

Deliver!

Self-deliveries omitted for simplicity.
Causal order multicast execution

[Diagram showing causal order multicast execution with nodes labeled P1 to P4 and time progression.]

Self-deliveries omitted for simplicity.
Causal order multicast execution

Self-deliveries omitted for simplicity.
Causal order multicast execution

Self-deliveries omitted for simplicity.
Causal order multicast execution

P1
[0,0,0,0]
[1,0,0,0] Deliver!
[1,1,0,0] Deliver!
[1,1,0,1] Deliver!
Missing 1 from P1 Buffer!

P2
[0,0,0,0]
[1,0,0,0] Deliver!
[1,1,0,0] Deliver!
[1,1,0,1] Deliver!

P3
[0,0,0,0]
[1,0,0,0] Deliver!

P4
[0,0,0,0]
[1,0,0,0] Deliver!
[1,0,0,1] Deliver!

Causality condition true for buffered multicasts
Deliver P2’s buffered multicast, [1,1,0,0]
Deliver P4’s buffered multicast, [1,1,0,1]
Causal order multicast implementation

- Only looks at multicast messages delivered to the application.
- Ignores causality created due to other network messages.
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
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

Sender

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 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.
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:

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();
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.

```plaintext
wait(S) (or P(S) or down(S)):
  while(1) { // each execution of the while loop is atomic
    if ($S > 0$) {
      $S--$; enter()
      break;
    }
  }

signal(S) (or V(S) or up(s)):
  $S++$; // atomic exit()
```

Atomic 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 Pi
    if (leader has token)
      Send token to Pi
    else
      Add Pi to queue
  • On receiving a token from process Pi
    if (queue is not empty)
      Dequeue head of queue (say Pj), 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
Ring-based Mutual Exclusion

- $N$ Processes organized in a virtual ring
- Each process can send message to its successor in ring
- Exactly 1 token
  - **enter()**
    - Wait until you get token
  - **exit()** // already have token
    - Pass on token to ring successor
- If receive token, and not currently in 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?
  - Next class!