Distributed Systems

CS425/ECE428

Feb 15 2022

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 \(m\).
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 Pj:
  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..N]}) at Pi from Pj: If Pi receives a multicast from Pj with sequence vector $V[1 \ldots N]$, buffer it until both:
  1. This message is the next one Pi is expecting from Pj, i.e.,
     \[ V[j] = P_i[j] + 1 \]
  2. All multicasts, anywhere in the group, which happened-before m have been received at Pi, 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

Self-deliveries omitted for simplicity.
Causal order multicast execution

Time

P1
[0,0,0,0]

[1,0,0,0]
Deliver!

[1,1,0,0]

P2
[0,0,0,0]

[1,0,0,0]
Deliver!

[1,1,0,0]

P3
[0,0,0,0]

Missing 1 from P1
Buffer!

P4
[0,0,0,0]

[1,0,0,0]
Deliver!

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!

- **P2**
  - [0,0,0,0]
  - \([1,0,0,0]\) Deliver!

- **P3**
  - [0,0,0,0]
  - Missing 1 from P1 Buffer!

- **P4**
  - [0,0,0,0]
  - \([1,0,0,0]\) Deliver!

- **Buffering and Delivery**
  - Deliver P1’s multicast, \([1,0,0,0]\)
  - Deliver P2’s buffered multicast, \([1,1,0,0]\)
  - Deliver P4’s buffered multicast, \([1,1,0,1]\)

- **Causality Condition**
  - Causality condition true for buffered multicasts
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

Sender
B-Multicast using unicast sends

Sender

TCP/UDP packets
B-Multicast using unicast sends

Sender

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.

Sender

TCP/UDP packets
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!
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 $1,000 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.

```c
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
```

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 $P_i$
    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

To be continued in next class.