CIS 630 - Fall 2010
Distributed Systems

Lecture 6
Coordination and Agreement

University of Oregon
Department of Computer and Information Science
Business and Logistics

☐ Project teams
  ☐ Meet and discuss project ideas
  ☐ Bounce ideas off Prof. Malony
    (5:15 – 6:30pm, Room 100)

☐ First problem set assigned

☐ Dr. Kate Keahey’s, *Cloud Computing for Science*
  ☐ CIS colloquium talk, 3:30pm Thursday, Deschutes
  ☐ You are required to be there! Good job!

☐ Programming exercise due midnight Friday
  (Check functionality next week?)
Acknowledgements

- Some material taken from author’s teaching slides based on Distributed Systems: Concepts and Design book
- Some figures taken from Distributed Systems: Concepts and Design book
Lecture Objectives

- Problems of distributed coordination and agreement
- Intuition and algorithmic techniques
- Understanding of theoretical and practical solutions
  - Limits due to the possibility of failure
- Algorithms
  - Distributed mutual exclusion and election algorithms
  - Multicast communication, consensus, and related problems
- Appreciate the impact of whether we use a synchronous or asynchronous system model on the algorithms we construct
Coordination and Agreement

- Coordination
  - Given a set of processes, we want them to agree on some value

- Instances of coordination problems occur all over the place in distributed systems
  - What is the balance of the bank account?
  - Who has control over modifying the database?
  - Which version of the data is up to date?
Failure in Distributed System

- Network separation due to communication link failure
  - Divides processes into communication partitions
  - Intra-partition works
  - Inter-partition delayed
  - May be intransitive
  - Assume temporary

- Node failures

- Failure detections tries to determine if processes are still active
Failure Detection

☐ If we want to say whether or not a process has failed, we need a detection mechanism that determines the state that the process in question is in

☐ Unreliable mechanism
  ☐ Unsuspected: “We recently heard from it.”
  ☐ Suspected: “Process may have failed – we haven’t heard from it in a while.” (Timeout)

☐ Reliable mechanism
  ☐ Unsuspected: “We recently heard from it.”
  ☐ Failed: It is dead.
Failure Detection Problem

- Obviously failure detection is a hard problem that is somewhat circular.
- One mechanism is to place a watcher in a node that has the sole purpose of monitoring the processes.
- What is wrong with this picture?
- If the watcher watching a process dies, but the process itself lives, how do we conclude that it has failed or not?
Distributed Mutual Exclusion

- Processes need to coordinate their activities
  - Fundamental concurrency problem
  - If share resources, mutual exclusion required to prevent interference and ensure consistency
  - Critical section problem (similar to OS problem)
- How do we control access to code and/or data that, for correctness reasons, only one process can work with at any given time?
  - Have the same problem in distributed systems
  - Require a distributed mutual exclusion solution
  - Solely based on message passing
Mutual Exclusion

☐ What is an example of this?

☐ Say we have a bank account with primitive operators like “withdraw” and “deposit”

☐ Now, say we write a transfer operation like this:

\[\text{Transfer}(\text{amt}, \text{src}, \text{dest}) = \text{src\_balance} = \text{src.getBalance()};\]
\[\text{src\_balance} -= \text{amt}; \text{dest\_balance} = \text{dest.getBalance()};\]
\[\text{dest\_balance} += \text{amt}; \text{dest.setBalance(dest\_balance)};\]
\[\text{src.setBalance(src\_balance)};\]

☐ What could possibly go wrong?
Mutual Exclusion (continued)

- Easy. We could have two processes read the balance before one of them has written the modified balance back to the store.
- We want to ensure that only one process is allowed to be in the transfer operation at any given time.
- We’d call the transfer code to be a critical section
- **How do we do this in a distributed system?**
  - *In a sequential system, we typically rely upon locks and atomic operations to protect critical sections*
  - *We don’t have those in a distributed system where we assume that all we have are message
Distributed Mutual Exclusion Problem

- Fixed number of processes sharing a resource
  - Only one process can use resource at a time
  - Need to synchronize to avoid conflict
  - No shared variables

- Assumptions
  - System is asynchronous
  - Processes do not fail
  - Messages are delivered reliably

- Use distributed critical section (CS) to coordinate activities

- Application-level protocol for CS execution
  - Three operations: enter(), resourceAccesses(), exit()
Basic Distributed Mutual Exclusion

Consider the following to be the abstract structure of a critical section that we want to enforce:

- Enter()
- DoSomethingDangerous()
- Exit()

So, what we need to implement is a distributed mechanism with which a process can determine if it is safe for it to enter the region, and notify others when it is out.
Essential Requirements

- **ME1: (safety)**
  - At most one process may execute in the CS at a time

- **ME2: (liveness)**
  - Requests to enter/exit the CS eventually succeed
  - Implies freedom from *deadlock* and *starvation*
  - Absence of starvation is a *fairness* condition

- **ME3: (→ ordering)**
  - If one request to enter the CS happened-before another, then entry to the CS is granted in that order
  - Prevents a process from entering more than once while another process is waiting
Conditions

Find algorithm satisfying three conditions:

- **C1**: a process holding the resource must release it before it is granted to another process
- **C2**: a different request for the resource must be granted in order in which requests were made
- **C3**: if every process granted the resource eventually releases it, every request is eventually granted

Safety involves C1 and liveness involves C2 and C3
Important Assumption

☐ An underlying assumption here is that the processes who wish to enter the critical section obey the entry and exit protocol

☐ If a client sees that the critical section is occupied, and goes ahead anyways into it, there is little these protocols can do to prevent it

☐ We assume that the processes are correct – in other words, they obey the protocol
Evaluation of Mutual Exclusion Algorithms

- **Bandwidth**
  - How much is consumed
  - Proportional to the number of messages sent
  - Entry and exit operations

- **Client delay**
  - How much delay is incurred at each entry and exit

- **Throughput**
  - How is access to the critical section as a whole affected
  - *Synchronization delay* between processes

- Do not consider the time within critical section
Central Scheduler: Need for Total Order

- Non-trivial problem due to ordering requirement
- Consider a central resource scheduler (next slide)
- Central scheduler grants requests in order received
  - Let P0 be the scheduling process
  - Suppose P1 sends a request to P0
  - P1 then sends a message to P2
  - P2 then sends a request to P0
  - P2’s request is received first at P0 (How?)
  - C2 is violated if P2’s request is granted first
- Must implement a system of logical clocks and create a total ordering (maintain causal ordering)
Server Managing Mutual Exclusion Token

- Use a server token granting approach
- Central server controls an access token
- When a process wants to enter critical section, it asks the server for the token
- If the server has it, it gives it to the process requesting it
- The process returns the token when it exits the critical section
- Any process that arrives when the token is not on the server wait in a FIFO queue
- Maintain an ordered queue of waiting requests
- Are requirements satisfied? Performance evaluation?
Distributed Algorithms

- Now consider the distributed case
- Each process independently follows rules
- No central synchronizing process or central store
- Mutual exclusion approach can be generalized to implement any multiprocess synchronization
- Synchronization specified by a state machine
  - Commands (all processes), states, and transitions
  - Each process independently simulates SM using commands from all processes in time order
  - Total time ordering ensures that each process uses the same sequence of commands
Distributed Algorithms (continued)

- All processes must actively participate
- Failure of one process will halt the system
- Addressing the failure problem is more difficult
- Concept of failure is only meaningful in the context of physical time
  - Otherwise no way to distinguish a failed process from one which is pausing between events
  - Response “timeouts” are only indication of a “crashed” system
Distributed Algorithm Timing Models

- Synchronous model
  - Distributed components take steps simultaneously
  - Unrealistic
  - Useful as intermediate step to general solution

- Asynchronous model
  - Distributed components take steps in arbitrary order at arbitrary speeds
  - Extra uncertainty about the order of events
  - Does allow timing considerations to be ignored
  - General and portable, but can be inefficient
Distributed Algorithm Timing Models (continued)

- Partially synchronous (timing-based) model
  - Assume some restrictions on relative timing of events
  - But execution is not completely lock-step
  - Most realistic, but most difficult to program
  - Can lead to efficient implementations
  - But implementations can be fragile and break if timing assumptions are violated
Ring-Based Algorithm

- Consider a topology-based distributed algorithm
- Multicast algorithms takes $2(n-1)$ messages to obtain mutual exclusion – expensive (see next algorithm)
- Consider token mechanism, as in central scheduler
  - Obtaining token grants access to resource
  - Returning token signifies release of resource
- Use a logical ring of processes for token passing
  - Token is passed from process to process (neighbor)
  - A process request (implicit) is granted when it receives the token and it holds the token until done
- What about happened-before order?
Ring Transmission of Mutual Exclusion Token

- Critical section access is granted on token receipt
- Critical section exit occurs on token forwarding
- Is the token obtained in a happened-before order?
- Continuous network resources required
- Performance? What is the best/worst case delay?
Distributed Mutual Exclusion Algorithm

- Assume messages from $P_i$ to $P_j$ are received in order
  - Can ensure this with messages #'s and ACKs
- Each process maintains a request queue
  - Each initially contains $T_0:P_0:req$ ($T_0<T_i$ for all $i$)
- Algorithm is defined by five rules:
  - **R1**: to request the resource, $P_i$ sends the message $T_m:P_i:req$ to every other process and puts that message on its request queue ($T_m$ is the LC timestamp)
  - **R2**: when a process $P_j$ receives $T_m:P_i:req$, it places it on its request queue and sends an ACK to $P_i$
    (* $P_i$ should defer processing requests until its own request has been sent and timestamp is known)
Distributed Mutual Exclusion Alg. (continued)

- **R3**: to release the resource, $P_i$ removes any $T_m:P_i:req$ from its request queue and sends $T_n:P_i:rel$ to all.
- **R4**: when process $P_j$ receives $T_n:P_i:rel$, it removes any $T_m:P_i:req$ from its request queue.
- **R5**: process $P_i$ is granted the resource when the following two conditions are satisfied:
  - **C1**: there exists $T_m:P_i:req$ ordered before any other request in its queue by the total clock ordering relation.
  - **C2**: $P_i$ has received a message from every other process timestamp later than $T_m$.

Does this work? Can you find any problem?
Ricart and Agrawala’s Algorithm

Mutual exclusion with multicast and logical clocks

On initialization
state := RELEASED;

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

On receipt of a request \(<T_i, p_i>\) at \(p_j (i \neq j)\)
if \((\text{state} = \text{HELD or (state} = \text{WANTED and } (T, p_j) < (T_i, p_i))))\
then
queue request from \(p_i\) without replying;
else
reply immediately to \(p_i\);
end if

To exit the critical section
state := RELEASED;
reply to any queued requests;

| RELEASED: | outside CS |
| WANTED:   | desired CS |
| HELD:     | holds CS   |
Ricart and Agrawala Performance

☐ How many messages are sent to request entry?
☐ How many messages are received to grant entry?
☐ What is the minimum and maximum delay?
Multicast Synchronization

- Used in Ricart and Agrawala distributed algorithm

[Diagram showing network communication with timestamps 41, 34, and 2.]

- Ordering using Lamport clocks
- Not interested in entering the critical section and replies immediately to requests
- No reply
- Lower timestamp
Maekawa’s Voting Algorithm

- Based on the idea that clever partitioning of the processes into “voting sets” allows granting of access to critical section from only a subset of processes
  - For a process to enter a CS, it is not necessary for all peers to grant it access, just a subset
  - Every subset has a non-empty intersection with every other
- Think of processes voting for one another to enter CS
  - A “candidate” must collect sufficient votes to enter
  - Processes in the intersection of two cast for just one
- Voting set $V_i \subseteq \{p_1, p_2, \ldots, p_n\}$
  - $p_i \in V_i$, $V_i \cap V_j \neq \emptyset$, $|V_i| = K$
  - Each process $p_j$ is in $M$ voting sets
  - Optimal: $K \sim \sqrt{N}$ and $M = K$
Maekawa’s Voting Algorithm – 1

☐ Initialization:
   - Everyone sets their state to released
   - Everyone sets their VOTED flag to false

☐ For a process $p$ to enter the critical section:
   - Set state to WANTED
   - Multicast request to all processes in its voting set
   - Wait until it hears back from everyone in its voting set
     - set state to HELD when this occurs
     - proceed to critical section (CS)
Maekawa’s Voting Algorithm – 2

☐ On receipt of message from $p$ on $q$:
  ☐ If $q$ is in a HELD state or has voted
    ➢ queue the request from $p$, do not reply
  ☐ Otherwise
    ➢ reply to $p$ and set VOTED flag to true

☐ When $p$ exits CS:
  ☐ Set state to RELEASED
  ☐ Multicast release to all members of voting set

☐ On receipt of release from $p$ on $q$:
  ☐ If queued requests exist
    ➢ Pop the queue, send a reply to it
    ➢ Set VOTED to false
Maekawa’s Algorithm – 1 (pseudo code)

Voting set $V_i \subseteq \{p_1, p_2, \ldots, p_n\}$

- $p_i \in V_i$, $V_i \cap V_j \neq \emptyset$, $|V_i| = K$

---

**On initialization**

- $state := \text{RELEASED}$;
- $voted := \text{FALSE}$;

**For $p_i$ to enter the critical section**

- $state := \text{WANTED}$;
  - Multicast request to all processes in $V_i - \{p_i\}$;
  - Wait until (number of replies received = $(K - 1)$);
- $state := \text{HELD}$;

**On receipt of a request from $p_i$ at $p_j$ ($i \neq j$)**

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

---

It is not necessary for all peers to grant access. Form subsets of peers and “vote” for candidates locally who then compete globally.

$V_i$ : voting set for each process $p_i$ is a subset of $\{p_1, p_2, \ldots, p_n\}$

Hold off request if already in candidate selection

Voting sets must overlap

continued on next slide
Maekawa’s Algorithm – 2 (pseudo code)

For $p_i$ to exit the critical section

state := RELEASED;
Multicast release to all processes in $V_i - \{p_i\}$;

On receipt of a release from $p_i$ at $p_j$ ($i \neq j$)
if (queue of requests is non-empty)
then
  remove head of queue – from $p_k$, say;
  send reply to $p_k$;
  voted := TRUE;
else
  voted := FALSE;
end if
Maekawa’s Algorithm Analysis

- Each process \( p_j \) is in \( M \) voting sets
- The benefit of this algorithm is that Maekawa characterized the optimal size of the voting sets:
  - \( K \sim \sqrt{N} \)
  - Number of messages is reduced to \( O(\sqrt{N}) \)
- Delay is again related to the round trip time
- Safety achieved since processes in the overlap \( V_i \cap V_k \) could not have voted for both \( p_i \) and \( p_k \) to enter
  - Only one vote can be cast between successive releases
- Hmm, something is fishy here …
- What about other requirements?
Maekawa’s Algorithm Problems

- Unfortunately, the algorithm is deadlock prone!
- Consider the following:
  - $V_1 = \{p_1, p_2\}$ $V_2 = \{p_2, p_3\}$ $V_3 = \{p_1, p_3\}$
  - Suppose all three processes concurrently ask for entry to the critical section
- What can happen? <thinking goes here>
- How do you fix the algorithm? Where?
Elections

- Procedure to choose (elect) a process from a group
  - Main requirement is uniqueness (elect a unique process)
    - processes must agree on choice
  - Process plays some particular role
    - “server” in Berkeley time algorithm
  - Must work in the presence of failure
  - Work in the presence of multiple calls for election

- Builds on mutual exclusion problem
  - Example: choosing server in central-server algorithm
  - Different because of failure assumption

- Bully algorithm [Silberschatz et al., 1993]
- Ring-based algorithm [Chang and Roberts, 1979]
Definitions and Requirements

- Each process is unique
- A process *calls the election*
  - Takes an action that initiates a run of election algorithm
  - Does not call more than one election at a time
  - $N$ processes could call $N$ concurrent elections
    - Because of uniqueness, one will end up the winner
- A process is a *participant* or a *non-participant*
  - *Participant*: engaged in some run of election algorithm
  - *Non-participant*: not currently engaged
- Choice of elected process must be unique
- Elected process must have largest *identifier* (WLG)
  - Identifiers must be unique and totally ordered
 Definitions and Requirements (continued)

☐ Each process $p_i$ ($i=1,2,...,N$) has a variable $elected_i$
  ☐ Contains identifier of elected process
  ☐ Initially set to undefined ($\perp$)

☐ During any particular run of the algorithm:
  ☐ E1: (safety)
    ➢ a participant process $p_i$ has $elected_i = \perp$ or $elected_i = P$
    ➢ $P$ is chosen as the non-crashed process at the end of the run with the largest identifier
  ☐ E2: (liveness)
    ➢ all processes $p_i$ participate and eventually set $elected_i \neq \perp$ or crash
Definitions and Requirements (continued)

There may be processes $p_j$ that are not yet participants, which record in $elected_j$ the identifier of the previous elected process.

Measure the performance of an election algorithm

- Total network bandwidth utilization
  - proportional to total number of messages sent

- Turnaround time
  - number of serialized message transmission times between initiation and termination of a single run
  - how fast the election algorithm runs from begin to end
Ring-Based Election (Chang and Roberts [1979])

- Logical ring arrangement and asynchronous system
- Neighbor communication (clockwise)
  \[ p_i \rightarrow p_{(i+1)\mod N} \]
- Do not know other process ID’s a priori
  - Elect process with highest ID (elect coordinator)
- All processes remain functional (assume no failures)
- Election participant and non-participant
  - Process makes itself a participant to start election
  - Sends election message with ID to neighbor
- Receipt of election message
  - If message ID >, forward message (become participant)
Ring-Based Election (continued)

- If message ID < & non-participant, substitute own ID in message and forward it (become participant)
- If message ID < & participant, don’t forward (Why?)
- If message ID =, process becomes coordinator
  - mark itself as non-participant
  - send elected message to neighbor with it’s ID

☐ Receipt of elected message
  - Mark as non-participant (extinguishes new elections)
  - Forward elected message (elected\_i = coordinator ID)

☐ Are E1 and E2 met?

☐ Worst case requires 3N-1 messages (improvements?)
Ring-Based Election in Progress

Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown darkened.

Issues:
1. Multiple elections
2. Failures (assume none)

Anyone can start by sending “election” message

Elect a single process called the “coordinator” (process with largest identified)

Forward largest identifier if not currently a participant

Coordinator is the process that receives its own id, then sends “elected” message
Bully Algorithm (Garcia-Molina [1982])

- What happens if there are crashes during election?
  - Bully algorithm allows processes to crash
    - Ring algorithm does not (Why?)

- Assumptions
  - Message delivery is reliable
  - System is synchronous
    - uses timeouts to detect a process failure
  - Processes know about other processes
    - know their ID (< order), can communicate with them

- Goal: selects surviving member with largest ID
  - Each process knows which processes have higher IDs
Bully Algorithm – Basic Idea

- A process who wants to start an election sends an election message to all processes above it
  - If it receives an answer from one, it is not the coordinator
- A process that receives the election message answers and attempts to contact those above it
- This proceeds until a process receives no answers
  - Either due to timeouts and/or process failures …
  - … or it being the maximum
- Allows a single process to decide if coordinator
- Issues?
  - Slow process and safety
Bully Algorithm – More Formal (1)

- Three types of messages:
  - *election*: announces election
  - *answer*: response to election message (apply max delay)
  - *coordinator*: announce identity of new coordinator

- Elections begin on *coordinator* failure
  - Noticed through timeouts
  - Assume some form of “keep alive” messages

- Election message sent to processes with ↑ID’s
  - Wait for answer message (determine if they are alive)

- Build reliable failure detector
  - $T = 2T_{\text{trans}} + T_{\text{process}}$ (upper bound on receiving response)
Bully Algorithm – More Formal (2)

- No *answer* messages received in certain time
  - Process considers itself the *coordinator*
  - Sends *coordinator* messages to processes with ↓ID’s
- Otherwise, wait for *coordinator* message
  - If none received in time (delay \(T’\)), begin new election
  - If received, record *coordinator* ID in \(elected_i\)
- Receipt of *election* message
  - Send back an *answer* message
  - Begin another election (unless already begun)
- Restarted process begins another election and may bully its way to become the *coordinator*
Example of Bully Algorithm

The election of coordinator $p_2$, after the failure of $p_4$ and then $p_3$

Liveness met by assumption of reliable message delivery

Timeouts should be set high enough to ensure reliable failure detection
Multicast

- Multicast is a generic group communication operation
  - IP multicast is simply one instance of it
- The basic primitives are:
  - \( \text{Multicast}(m, g) \) where \( m \) is a message and \( g \) is a group of participating processes
  - \( \text{Deliver}(m) \) delivers the message on the receiving process
- Messages in a multicast system are annotated with the sender and group that they are intended for
Basic Multicast

- This is the easiest multicast scheme
- For all processes in the group, the sender sends a point-to-point message
- Delivery is achieved with the basic receive operation
- Unlike IP multicast, we assume the communications is reliable
  - One can use IP multicast to implement this if a layer is placed over the UDP messages to do retries, duplicate handling, and order enforcement
  - Why not just use TCP?
- Not efficient in its basic form because of sender bottleneck
Reliable Multicast

- In basic multicast, we can see the following occur:
  - Sender starts its sequence of sends
  - Part of the way through, it dies
  - Some set of group members never get the message

- Reliable multicast adds what is known as an "Agreement proper"
  - If any member of the group delivers $m$, then all of the correct (not failed) processes in the group will eventually deliver $m$.

- We can achieve this on top of basic multicast
Reliable Multicast Approach

- At start, everyone initializes their received set to empty
- The originator of the message uses basic multicast to send to the entire group, including itself
- On the basic deliver occurring on a process \( q \):
  - If the message is not in the received set on \( q \):
    - add it to the set
    - if \( q \) was not the originator of the message, basic multicast it to the group
    - successfully deliver the message reliably
- Good property: processes can die, but if any of them successfully deliver it, then everyone will eventually see it
- Bad property: the message gets sent by every member of the group to everyone else (flooding)
  - Not efficient
Ordered Multicast

- Ordering requirements on when messages delivered

- **FIFO ordering**
  - If a correct process says `multicast(g,m)` and then `multicast(g,m')`, then every correct process that delivers `m'` will deliver `m` before `m'`

- **Causal ordering**
  - If `multicast(g,m)` happens before `multicast(g,m')`, then any correct process that delivers `m'` will deliver `m` before `m'`.
    - happened-before relation hold on the group!

- **Total ordering**
  - If a correct process delivers `m` before `m'`, then any other correct process that delivers `m'` will deliver `m` before `m'`
Ordered Multicast Consequences

- Causal ordering implies FIFO ordering
  - Due to happened-before relation in a single process
- FIFO and causal orderings are only partial orders
- For time reasons, we will not go into the algorithms
  - Look in book
  - See that the use of logical clocks (either Lamport or Vector) makes it possible to negotiate the necessary ordering by attaching timestamps to messages as they are moved around and ordered
- Ordering is expensive in delivery latency and bandwidth consumption
IP multicast – an implementation of group communication

- Built on top of IP (IP packets addressed to computers)
- Allows sender to transmit a single IP packet to a set of computers that form a multicast group
- Dynamic membership of groups
  - can send to a group with or without joining it
- To multicast, send a UDP datagram with a multicast address
- To join, make a socket join a group enabling it to receive messages to the group
Revision of IP Multicast (continued)

- **Multicast routers**
  - Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.

- **Failure model**
  - Omission failures if some but not all members may receive a message.
    - example: a recipient may drop message
    - example: a multicast router may fail
  - IP packets may not arrive in sender order, group members can receive messages in different orders
## Introduction to Multicast

- Multicast communication requires coordination and agreement
- **Aim:** Group members receive copies of messages sent to group
- Many different delivery guarantees are possible
  - Agree on the set of messages received or on delivery ordering
- Process can multicast using a single operation
  - **Efficiency**
    - send once on each link
    - using hardware multicast when available
  - **Delivery guarantees**
    - can not make a guarantee if multicast is implemented as multiple sends and the sender fails
    - can also do ordering
- Many projects (Amoeba, Isis, Transis, Horus)
System Model

- The system consists of a collection of processes which can communicate *reliably* over 1-1 channels
- Processes fail only by crashing (no arbitrary failures)
- Processes are members of groups
  - Destinations of multicast messages
- In general, process $p$ can belong to more than one group
- Operations
  - $multicast(g, m)$:
    - sends message $m$ to all members of process group $g$
  - $deliver(m)$:
    - get a multicast message delivered
    - may be delayed to allow for ordering or reliability
System Model (continued)

☐ Multicast message $m$ carries the id of the sending process $sender(m)$ and the id of the destination group $group(m)$

☐ We assume there is no falsification of the origin and destination of messages
Open and Closed Groups

- **Closed groups**
  - Only members can send to group
  - Useful for coordination of cooperating server groups

- **Open**
  - Event notification to groups of interested processes
Reliability of One-to-One Communication

- The term reliable 1-1 communication is defined in terms of validity and integrity

- **Validity**
  - Any message in the outgoing message buffer is eventually delivered to the incoming message buffer
  - Implemented by use of acknowledgements and retries

- **Integrity**
  - The message received is identical to one sent, and no messages are delivered twice
  - Implemented by checksums, rejection of duplicates
  - If allowing for malicious users, use security techniques
Basic Multicast

☐ A correct process will eventually deliver the message provided the multicaster does not crash
  - Note that IP multicast does not give this guarantee
☐ The primitives are called \textit{B-multicast} and \textit{B-deliver}
☐ A straightforward but ineffective method of implementation
  - Use a reliable one-to-one \textit{send}
    - with integrity and validity as above
  - \textit{B-multicast}(g,m):
    - for each process \( p \in g \), \textit{send}(p, m)
  - On \textit{receive} \( (m) \) at \( p \), \textit{B-deliver} \( (m) \) at \( p \)
☐ Problem
  - If large \# processes, the protocol will suffer from \textit{ack-implosion}
Implementation of Basic Multicast over IP

- Each process $p$ maintains:
  - Sequence number, $S^p_g$ for each group it belongs to
  - Sequence number, $R^q_g$, of the latest message it has delivered from process $q$

- For process $p$ to $B$-multicast a message $m$ to group $g$
  - Piggybacks $S^p_g$ on message $m$, using IP multicast to send
  - Piggybacked sequence numbers allow recipients to learn about messages they have not received

- On receive $(g, m, S)$ at $p$:
  - If $S = R^q_g + 1$, $B$-deliver ($m$) and increment $R^q_g$ by 1
  - If $S < R^q_g + 1$, reject message because already delivered
  - If $S > R^q_g + 1$, a message is missing, request from sender

- If the sender crashes, then a message may be delivered to some members of the group but not others
Reliable Multicast

- Protocol is correct even if the multicaster crashes
- It satisfies criteria for validity, integrity and agreement
- It provides operations R-multicast and R-deliver
- **Integrity** - a correct process, $p$ delivers $m$ at most once
  - Also $p \in \text{group}(m)$ and $m$ was supplied to a multicast operation by $\text{sender}(m)$
- **Validity** - if a correct process multicasts $m$, it will eventually deliver $m$
- **Agreement** - if a correct process delivers $m$ then all correct processes in $\text{group}(m)$ will eventually deliver $m$
  - **Atomicity** - all or nothing, even if multicaster crashes
Reliable Multicast Algorithm w/ Basic Multicast

- Processes can belong to several closed groups
- To \textit{R-multicast} a message, process \textit{B-multicasts} to all processes in the group, including itself

\textbf{On initialization}

\texttt{Received := \{};\\

\textbf{For process } p \texttt{ to } R\text{-multicast message } m \texttt{ to group } g \\
\quad \texttt{B-multicast}(g, m); \quad \texttt{// } p \in g \texttt{ is included as a destination}\\

\textbf{On B\text{-deliver}(m) at process } q \texttt{ with } g = \text{group}(m) \\
\quad \textbf{if } (m \notin \texttt{Received}) \\
\quad \quad \texttt{then} \\
\quad \quad \quad \texttt{Received := Received \cup \{} m \}; \\
\quad \quad \quad \texttt{if } (q \neq p) \texttt{ then B-multicast}(g, m); \texttt{ end if} \\
\quad \quad \texttt{R-deliver } m; \\
\quad \texttt{end if}
Reliable Multicast Algorithm (continued)

☐ Validity
  ☑ A correct process will B-deliver to itself

☐ Integrity
  ☑ Because the reliable one-to-one channels used for B-multicast guarantee integrity

☐ Agreement
  ☑ Every correct process B-multicasts the message to the others

  ☑ If p does not R-deliver then this is because it didn’t B-deliver - because no others did either
Reliable Multicast over IP Multicast

- This protocol assumes groups are closed
  - Uses piggybacked acknowledgement messages
  - Uses negative acknowledgements for missed messages

- Process \( p \) maintains:
  - \( S^p_g \): message sequence number for each of its group
  - \( R^q_g \): sequence number of latest message received from process \( q \) to \( g \)

- For process \( p \) to \( R\)-multicast message \( m \) to group \( g \)
  - Piggyback \( S^p_g \) and +ve acks for messages received in the form \( <q, R^q_g> \)
  - IP multicasts the message to \( g \), increments \( S^p_g \) by 1
Reliable Multicast over IP Multicast (continued)

- On receipt by of a message to g with S from p
  - Iff $S = R^p_g + 1$ R-deliver message and increment $R^p_g$ by 1
  - If $S \leq R^p_g$ discard the message
  - If $S > R^p_g + 1$ or if $R < R^q_g$ (for enclosed ack $<q,R>$)
    - Then it has missed messages and requests them with negative acknowledgements
  - Puts new message in hold-back queue for later delivery

- Piggybacked values in a message allow recipients to learn about messages they have not yet received
Hold-back Queue for Arriving Multicast Messages

- Not necessary for reliability (as in using IP multicast)
- Simplifies the protocol, allowing sequence numbers to represent sets of messages
- Hold-back queues are also used for ordering protocols.
Properties of Reliable Multicast over IP

- **Integrity**
  - Duplicate messages detected and rejected (checksums)

- **Validity**
  - Due to IP multicast in which sender delivers to itself

- **Agreement**
  - Processes can detect missing messages
  - Keep copies of messages for retransmission

- **Discarding of message copies no longer needed**
  - Note which processes have received messages
  - When all processes in g have the message, discard it
  - Use ‘heartbeat’ messages to check if processes alive
Ordered Multicast

- A variety of delivery orderings may be implemented
- FIFO ordering
  - \(\text{multicast}(g, m)\) followed by \(\text{multicast}(g,m')\)
  - Every correct process that delivers \(m'\) will deliver \(m\) before \(m'\)
- Causal ordering
  - \(\text{multicast}(g, m) \rightarrow \text{multicast}(g,m')\)
  - Where \(\rightarrow\) is the happened-before relation between \(g\) messages
  - Any correct process that delivers \(m'\) will deliver \(m\) before \(m'\)
- Total ordering
  - If a correct process delivers message \(m\) before it delivers \(m'\)
  - Any other correct process delivering \(m'\), delivers \(m\) before \(m'\)
- Ordering is expensive in delivery latency and bandwidth consumption
Total, FIFO, and Causal Multicast Ordering

- Notice the consistent ordering of totally ordered messages $T_1$ and $T_2$. They are opposite to real time. The order can be arbitrary it need not be FIFO or causal.

- FIFO related messages $F_1$, $F_2$

- Causally related messages $C_1$, $C_3$

- These definitions do not imply reliability, but we can define atomic multicast - reliable and totally ordered.

- Ordered multicast delivery is expensive in bandwidth and latency. There, the less expensive orderings (FIFO or causal) are chosen for applications for which they are suitable.
Display from a Bulletin Board Program

- Bulletin board applications which multicast messages
- One multicast group per topic (e.g., `os.interesting`)
- Require reliable multicast and ordering

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>A.Hanlon</td>
<td>Mach</td>
</tr>
<tr>
<td>24</td>
<td>G.Joseph</td>
<td>Microkernels</td>
</tr>
<tr>
<td>25</td>
<td>A.Hanlon</td>
<td>Re: Microkernels</td>
</tr>
<tr>
<td>26</td>
<td>T.L’Heureux</td>
<td>RPC performance</td>
</tr>
<tr>
<td>27</td>
<td>M.Walker</td>
<td>Re: Mach</td>
</tr>
</tbody>
</table>

- total (makes the numbers the same at all sites)
- causal (makes replies come after original message)
- FIFO (gives sender order)
Implement FIFO Ordering over Basic Multicast

- We discuss FIFO ordered multicast with operations FO-multicast and FO-deliver for non-overlapping groups
  - Can be implemented on top of any basic multicast
- Each process $p$ holds:
  - $S^p_g$: a count of messages sent by $p$ to $g$ and
  - $R^q_g$: sequence number of latest message to $g$ that $p$ delivered from $q$
- For $p$ to FO-multicast a message to $g$
  - Piggybacks $S^p_g$ on the message
  - B-multicasts it and increments $S^p_g$ by 1
- On receipt of a message from $q$ with sequence number $S$
  - $p$ checks whether $S = R^q_g + 1$; if so, it FO-delivers it
- If $S > R^q_g + 1$
  - $p$ places message in hold-back queue until intervening messages have been delivered
  - B-multicast does eventually deliver messages unless sender crashes
Implementation of Totally Ordered Multicast

- The general approach is to attach totally ordered identifiers to multicast messages
  - Each receiving process makes ordering decisions based on the identifiers
  - Similar to the FIFO algorithm, but processes keep group specific sequence numbers
  - Operations TO-multicast and TO-deliver

- We present two approaches to implementing total ordered multicast over basic multicast
  - Using a sequencer (only for non-overlapping groups)
  - The processes in a group collectively agree on a sequence number for each message
Total Ordering using a Sequencer

1. Algorithm for group member \( p \)

   **On initialization:** \( r_g := 0; \)

   **To TO-multicast message \( m \) to group \( g \)**
   
   \( B\)-multicast(\( g \cup \{ \text{sequencer}(g) \} \), \( <m, i> \));

   **On B-deliver(\( <m, i> \)) with \( g = \text{group}(m) \)**
   
   Place \( <m, i> \) in hold-back queue;

   **On B-deliver(\( m_{\text{order}} = \langle \text{“order”}, i, S \rangle \)) with \( g = \text{group}(m_{\text{order}}) \)**
   
   wait until \( <m, i> \) in hold-back queue and \( S = r_g \);
   
   **TO-deliver \( m \);** // (after deleting it from the hold-back queue)
   
   \( r_g = S + 1; \)

2. Algorithm for sequencer of \( g \)

   **On initialization:** \( s_g := 0; \)

   **On B-deliver(\( <m, i> \)) with \( g = \text{group}(m) \)**
   
   \( B\)-multicast(\( g \), \( \langle \text{“order”}, i, s_g \rangle \));
   
   \( s_g := s_g + 1; \)

A process wishing to TO-multicast \( m \) to \( g \) attaches a unique id, \( id(m) \) and sends it to the sequencer and the members.

Other processes: **B-deliver**

\( <m, i> \) put \( <m, i> \) in hold-back queue

**B-deliver** order message, get \( g \) and \( S \) and \( i \) from order message

wait till \( <m, i> \) in queue and \( S = r_g \),

**TO-deliver** \( m \) and set \( r_g \) to \( S+1 \)

The **sequencer** keeps sequence number \( s_g \) for group \( g \)

When it **B-delivers** the message it multicasts an ‘order’ message to members of \( g \) and increments \( s_g \).
Discussion of Sequencer Protocol

- Since sequence numbers are defined by a sequencer, we have total ordering.
- Like B-multicast, if the sender does not crash, all members receive the message.
- What are potential problems with a single sequencer?
- What happens when some members stop multicasting?
- What can the sequencer do about its history buffer becoming full?
- Members piggyback on their messages the latest sequence numbers they have seen.
- Members that do not multicast send heartbeat message.
Kaashoek’s Protocol uses Hardware Multicast

- Sender transmits one message to sequencer
- Sequencer multicasts sequence number and message
- But IP multicast is not as reliable as B-multicast
  - Sequencer stores messages in its history buffer
  - Retransmission on request
- Members notice messages are missing by inspecting sequence numbers
The ISIS algorithm for total ordering

- This protocol is for open or closed groups

1. The process $P_1$ *B-multicasts* a message to members of the group
2. The receiving processes propose numbers and return them to the sender
3. The sender uses the proposed numbers to generate an agreed number

The ISIS algorithm for total ordering

This protocol is for open or closed groups
ISIS Total Ordering – Sequence # Agreement

Each process, q keeps:
- $A^q_g$ the largest agreed sequence number it has seen and
- $P^q_g$ its own largest proposed sequence number

1. Process p $B$-multicasts $<m, i>$ to g, where $i$ is a unique identifier for m.

2. Each process q replies to the sender p with a proposal for the message’s agreed sequence number of
   - $P^q_g := \text{Max}(A^q_g, P^q_g) + 1$.
   - Assigns the proposed sequence number to the message and places it in its hold-back queue

3. p collects all the proposed sequence numbers and selects the largest as the next agreed sequence number, a.
   It $B$-multicasts $<i, a>$ to g. Recipients set $A^q_g := \text{Max}(A^q_g, a)$, attach a to the message and re-order hold-back queue.
Discussion of Ordering in ISIS Protocol

- Hold-back queue
- Ordered with smallest sequence numbered message at front
- When agreed number is added, the queue is re-ordered
- When the message at front has an agreed id
  - It is transferred to delivery queue
  - Those not at front of the queue with agree id are not transferred
- Every process agrees on order and delivers messages in order
  - Therefore we have total ordering
- Latency
  - 3 messages are sent in sequence
  - Therefore it has a higher latency than sequencer method
  - This ordering may not be causal or FIFO
Causally Ordered Multicast

- We present an algorithm of Birman [1991] for causally ordered multicast in non-overlapping, closed groups
  - It uses the \textit{happened before} relation (on multicast messages only)
    - ordering imposed by one-to-one messages is not taken into account
- It uses vector timestamps
  - Count the number of multicast messages from each process that happened before the next message to be multicast
Causal Ordering using Vector Timestamps

Algorithm for group member $p_i$ ($i = 1, 2\ldots, N$)

On initialization

$$V_i^g[j] := 0 \quad (j = 1, 2\ldots, N);$$

To CO-multicast message $m$ to group $g$

$$V_i^g[i] := V_i^g[i] + 1;$$

$B$-multicast$(g, <V_i^g, m>);$  

On $B$-deliver$(<V_j^g, m>)$ from $p_j$, with $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;  

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);  

$CO$-deliver $m;$  

then it CO-delivers the message and updates its timestamp

Note: a process can immediately $CO$-deliver to itself its own messages (not shown)

To $CO$-multicast $m$ to $g$, a process adds 1 to its entry in the vector timestamp and $B$-multicasts $m$ and the vector timestamp

When a process $B$-delivers $m$, it places it in a hold-back queue until messages earlier in the causal ordering have been delivered:

a) earlier messages from same sender have been delivered  
b) any messages that the sender had delivered when it sent the multicast message have been delivered
Comments

- After delivering a message from \( p_j \), process \( p_i \) updates its vector timestamp
  - By adding 1 to the \( j \)th element of its timestamp
- Compare the vector clock rule where
  \[ V_i[j] := \max(V_i[j], t[j]) \] for \( j=1, 2, \ldots N \)
  - In this algorithm we know that only the \( j \)th element will increase
- For an outline of the proof see page 449
- If we use \( R \)-multicast instead of \( B \)-multicast then the protocol is reliable as well as causally ordered.
- If we combine it with the sequencer algorithm we get total and causal ordering
Comments on Multicast Protocols

- We need to have protocols for overlapping groups because applications do need to subscribe to several groups.
- Definitions of “global FIFO ordering” in book and some references to papers on them.
- Multicast in synchronous and asynchronous systems.
  - All of our algorithms do work in both.
- Reliable and totally ordered multicast.
  - Can be implemented in a synchronous system.
  - But is impossible in an asynchronous system (reasons discussed in consensus section - paper by Fischer et al.)
Multicast Summary

- Multicast communication can specify requirements for reliability and ordering
  - In terms of integrity, validity and agreement

- B-multicast
  - Correct process will eventually deliver a message provided the multicaster does not crash

- Reliable multicast
  - Correct processes agree on the set of messages to be delivered
  - Two implementations: over B-multicast and IP multicast
Multicast Summary (continued)

- Delivery ordering
  - FIFO, total and causal delivery ordering.
  - FIFO ordering by means of senders’ sequence numbers
  - Total ordering
    - by means of a sequencer
    - by agreement of sequence numbers between processes in a group
  - Causal ordering by means of vector timestamps

- The hold-back queue is a useful component in implementing multicast protocols
Consensus and Problems of Agreement

- Get processes to agree on a value after one or more of the processes has proposed what the value should be
  - Example: bank credit/debit transactions

- Distributed consensus is hard
  - In fact, situations can exist in which consensus can not be achieved
  - These situations are actually not rare or unexpected

- Our standard assumption is of a set of processes communicating by messages

- Here is the wrinkle: consensus must be reachable in the presence of faults
  - Assume some number of faults
Consensus Approach

- Define consensus precisely and look at three problems
- Systems model
  - Collection of processes $p_i \ (i=1,2,...,N)$
  - Message passing
    - Processes can fail, but communications reliable
- Each process starts undecided
- Processes propose a value
- Communication occurs and a decision value is reached
- Processes reach the decided state when they receive this final decision message
Consensus Model and Requirements

☐ Agree on a value after one or more processes have proposed a value $v_i$
  ☐ Process $p_i$ each have a decision variable $d_i$
  ☐ Move process state from undecided to decided
  ☐ Cannot change once decided

☐ Requirements
  ☐ Termination: each process sets $d_i$ eventually
  ☐ Agreement: $d_i = d_j \ (i,j = 1,\ldots,N)$ if $p_i, p_j$ in decided state
  ☐ Integrity (stronger): if same value proposed by all correct processes, any decided process chose that value
  ☐ Validity: if all processes propose the same value, then every correct process chooses that value
Consensus Cases

- **Simple case**
  - Processes do not fail
  - Each process reliably multicasts proposed value
    - guarantees integrity
  - Collect all $N$ values
  - Evaluate $\text{majority}(v_1, v_2, ..., v_N) = \text{majority}$ or $\bot$
    - have some known majority function (could
    - guarantees agreement and integrity

- **Process failures**
  - Introduces complication of detecting failures
  - Not clear whether there is termination
Consensus Cases (continued)

- Interactive consistency
  - Every process proposes a single value
  - Agree on vector of values ("decision vector")

- Arbitrary (byzantine) failures
  - Processes may communicate random values
  - Must compare what have received with what other processes claim to have received
  - Leads to the Byzantine Generals problem

- Consider synchronous and asynchronous approaches
  - What is the difference between synchronous and asynchronous assumptions?
Consensus Problem Relations

- The types of consensus problems are related
  - IC from BG: run BG N times, once with each process acting as a commander (i.e., run BG per vector entry)
  - C from IC: run IC, and then compute the majority function on each element of the vector on all processes
  - BG from C: commander sends proposed value out and to itself, and then all processes run consensus

- In systems with potential crashes, consensus is equivalent to totally ordered multicast
  - All processes multicast their value
  - Each process chooses the first value that it delivers
Consensus in a Synchronous System (Dolev [1983])

- Assume $f$ failures tolerated

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

**On initialization**

$Values_i^1 := \{ v_i \}; \quad Values_i^0 = \{ \}$;

**In round $r$ ($1 \leq r \leq f + 1$)**

- $B$-multicast($g$, $Values_i^r - Values_i^{r-1}$); // Send only values that have not been sent
- $Values_i^{r+1} := Values_i^r$;
- while (in round $r$)

  
  
  **On $B$-deliver($V_j$) from some $p_j$**

  $Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

**After $(f + 1)$ rounds**

Assign $d_i = \text{minimum}(Values_i^{f+1})$;

- $B$-multicast: values are eventually delivered
- Up to $f$ of the $N$ processes exhibit crash failures
- In each round, processes $B$-multicast values
- Record any new values
- Must show each process arrives at the same set of values at the end of the final round
  - Assume otherwise at end
  - Work backwards round by round
  - Can only go backwards $f$ rounds
Byzantine Generals Problem (Lamport et al. [1982])

- Classic consensus problem proposed by Lamport
- Three or more generals are to agree to attack or retreat
  - Commander: issues the order
  - Lieutenants: decide to attack or retreat
  - One or more generals may be treacherous (i.e., faulty)
    - general tells one peer to attack and the other to retreat
  - Differs from “pure” consensus because one special process initiates the orders
- Requirements
  - Termination: eventually each correct process sets $d_i$
  - Agreement: decision value is same for correct processes
  - Integrity: if commander correct, then all correct processes decide on the value that commander proposed
Byzantine Generals Problem

- What is the main idea behind this problem?
- How to detect a system that is faulty
  - Either by malfunction or lying
- Book refers to proofs of impossibility of detection
  - For $N \leq 3f$
    - $f$ is the number of failure
    - $N$ is the number of participants
- Three generals (X,Y,Z)
  - X tells Y “attack!” and X tells Z “retreat!”
  - Y tells Z “attack!” and Z tells Y “retreat!”
  - Y and Z can not tell who is lying
Three Byzantine Generals

Three processes send unsigned messages

Assume processes exhibit arbitrary failures. May send any message with any value at any time. Up to $f$ of $N$ processes may be faulty.

Faulty processes are shown shaded

Must choose commander’s value

If one process is allowed to fail, Lamport et al. showed that no solution exists if $N \leq 3f$, but they gave an algorithm for $N \geq 3f+1$ for synchronous system for unsigned messages

No algorithm can distinguish between the two scenarios if the commander does not sign its messages
Four Byzantine Generals

Faulty processes are shown shaded

In the general case ($f \geq l$), the Lamport et al. algorithm for unsigned messages operates over $f+1$ rounds.
Asynchronous Consensus

- Asynchronous systems have no known bounds on times
  - A very long message wait period can look like a failure
- Methods exist to work around this, through fault detection, fault masking, and so on
- In any case, consensus in an arbitrary asynchronous system is impossible