Learning Objectives

• To learn algorithms of achieving mutual exclusion among a collection of processes, so as to coordinate their accesses to shared resources
• To learn how an election can be implemented in a distributed system
• To learn multicast communication algorithms
• To understand the major challenges: asynchronous message passing and failures
Failure Assumption and Failure Detectors

- Each pair of processes is connected by reliable channels
- Unreliable failure detectors
  - Two values: unsuspected or suspected, both unreliable
  - Failure-detection algorithm based on \( T + D \) criteria (\( T \) is the period of sending query and \( D \) is the estimated maximum message transmission)
  - False positive and false negative
- Reliable failure detectors
  - Feasible in synchronous distributed systems

Re: Mutual Exclusion – Race Conditions

- Race conditions: when two or more processes/threads are reading or writing some shared data and the final results depends on who runs precisely
- Critical region/section: the part of the program where the shared memory is accessed

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read balance; $1000</td>
<td>Read balance; $1000</td>
<td>$1000</td>
</tr>
<tr>
<td></td>
<td>Deposit $200</td>
<td>Deposit $200</td>
<td>$1000</td>
</tr>
<tr>
<td></td>
<td>Update balance $1000 + $200</td>
<td>Update balance $1000 + $200</td>
<td>$1200</td>
</tr>
</tbody>
</table>

Two threads want to deposit an account; overwriting issue
Distributed Mutual Exclusion

- Master/slave model and P2P decentralized model

- Requirements for mutual exclusion
  - ME1 (safety): at most one process may execute in the critical section at a time
  - ME 2 (liveness): requests to enter and exit the critical section eventually succeed (no deadlock or starvation)
  - ME 3 (ordering for more fairness): if one request to enter the critical section happened-before another, then entry is granted in that order

- Application-level protocol for executing a critical section
  - enter() + resourceAccesses() + exit()
  - Assumes each pair of processes is connected by reliable channels

Algorithms for Distributed Mutual Exclusion

- Master/slave model and P2P decentralized model

- Performance Metrics
  - The bandwidth consumed, which is proportional to the number of messages sent in each entry and exit operation
  - The client delay incurred by a process at each entry and exit operation
  - The throughput of the system (the collection of processes), measured by the effect using the synchronization delay between one process exiting the critical section and the next process entering it (the shorter it is, the greater the throughput is)

- Fault tolerance
  - What happens when messages are lost?
  - What happens when a process crashes?
Master/Slave Model – The Central Server Algorithm

- A central server grants permission (a token) to enter the critical section
- Discuss the ME1, ME2, and ME3 requirements (Why ME3 not met)?
- Discuss the performance of the algorithm

A Ring-based Algorithm

- The exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction clockwise.
- Discuss the ME1, ME2, and ME3 requirements (Why ME3 not met)?
- Discuss the performance of the algorithm
Ricart and Agrawala’s Multicast-based Algorithm

On initialization
state := RELEASED;

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

On receipt of a request <T, p_i> at p_j (i ≠ j)
if (state = HELD or (state = WANTED and (T, p_j) < (T, 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;

Basic idea? uses multicast to announce entry request and enter it only all the other processes have replied to this request

Multicast Synchronization Example

P3 is not interested in critical section
P1 and p2 request it concurrently, with their timestamps 41 and 34, respectively

• Discuss the ME1, ME2, and ME3 requirements (proof?)
• Discuss the performance of the algorithm
Maekawa’s Algorithm

- On initialization
  
  \[
  \text{state} := \text{RELEASED};
  \text{voted} := \text{FALSE};
  \]

- For \( p_i \) to enter the critical section
  
  \[
  \text{state} := \text{WANTED};
  \text{Multicast request to all processes in } V_i;
  \text{Wait until (number of replies received} = K)\;
  \text{state} := \text{HELD};
  \]

- On receipt of a request from \( p_i \) at \( p_j \)
  
  \[
  \text{if (state} = \text{HELD or voted} = \text{TRUE})
  \text{then}
  \text{queue request from } p_i \text{ without replying;}
  \text{else}
  \text{send reply to } p_j;
  \text{voted} := \text{TRUE};
  \text{end if}
  \]

- For \( p_j \) to exit the critical section
  
  \[
  \text{state} := \text{RELEASED};
  \text{Multicast release to all processes in } V_i;
  \]

- On receipt of a release from \( p_i \) at \( p_j \)
  
  \[
  \text{if (queue of requests is non-empty)}
  \text{then}
  \text{remove head of queue – from } p_k, \text{ say;}
  \text{send reply to } p_k;
  \text{voted} := \text{TRUE};
  \text{else}
  \text{voted} := \text{FALSE};
  \text{end if}
  \]

Fault Tolerance of the Algorithms

- What happens when messages are lost?
  
  - Master/Slave
  - Ring-based
  - Richart and Arrawala

- What happens when a process crashes?
  
  - Master/Slave
  - Ring-based
  - Richart and Arrawala
Elections

- Election: to choose a unique process to play a particular role
  - In “central-server” mutual exclusion: to choose which of the processes to play the role of the server, and to choose a replacement if needed
  - It is essential all the processes agree on the choice

- Action and State
  - To call the election (N processes could call N concurrent elections)
  - An any point of time, a process is either a participant or non-participant

ID Selection and Requirements of Election

- Without loss of generality, the elected process be chosen as the one with the largest identifier
  - How to make sure the IDs are unique and totally ordered?
  - Example: <1/load, i> where process index i is for tie-break

- Requirements
  - E1 (safety): a participant process pi has elected_i = ⊥ or P, where P is chosen as the non-crashed process at the end of the election with the largest ID
  - E2 (liveness): all processes pi participate and eventually set elected_i ≠ ⊥ or crash

- Performance metrics
  - Total bandwidth utilization, proportional to the number of messages sent
  - Turnaround time for the algorithm: the number of serialized message transmission times between the initiation and termination of a single run
Initially, each process is non-participant. Any process can begin an election, by marking itself as a participant, placing its ID in an election message and send to its clockwise neighbor.

When a process receives an election message, it compares the ID in the message with its own:
(a) If arrived ID is greater, forward it to neighbor
(b) If arrived ID is smaller and the receiver is not a participant, substitute with its ID and forward; but it does not forward the message if it is already a participant (why?)
(c) On forwarding an election message, the process marks itself as a participant.

When election is completed?

If the received ID is that of the receiver itself, this process must be the coordinator as its ID must be the greatest.

What it to do next?

It marks itself as a non-participant and sends an elected message to its neighbor announcing its election and enclosing its identity.

Can node failure be considered?

- Discuss the E1 and E2 requirements (considering concurrent elections)
- Discuss the performance of the algorithm
The Bully Algorithm

• Assumptions
  • Processes may crash during an election, but message delivery between processes is reliable
  • System is synchronous: it uses timeout to detect a process failure
  • Each process knows which processes have higher IDs that it can communicate with all such processes
    • what is the case in the Ring-based algorithm?

• Messages
  • Election: to announce an election
  • Answer: a message in response to an election message
  • Coordinator: a message to announce the ID of the elected process

• Failure detection
  • A reliable failure detector is based on threshold $T = 2T_{\text{trans}} + T_{\text{process}}$

The Bully Algorithm Example

When a process is started to replace a crashed process, it begins an election.

Stage 1

How a process with the highest ID proceed if it wants to elect?

Stage 2

How a process with a lower ID proceed if it wants to elect?

Stage 3

How about E2 requirement?

Stage 4

How about E1 requirement if no process is replaced, or if processes that have crashed are replaced by processes with the same ID?

How about the performance? When is the best case and worst case?

Eventually.....

The election of coordinator $p_2$, after the failure of $p_4$ and then $p_3$
Review: Group Communication

- **Multicast (one-to-many):**
  - Sends a single message from a process to each of the members of a group of processes; usually transparent group membership
  - Failures and ordering are more complicated than one-to-one

Case 1: fault tolerance

Case 2: resource location

Case 3: update replicas

Case 4: event notification

Revisit of IP Multicast

- **IP multicast** – an implementation of group communication
  - built on top of IP (note IP packets are addressed to computers)
  - allows the sender to transmit a single IP packet to a set of computers that form a multicast group (a class D internet address with first 4 bits 1110)
  - Dynamic membership of groups; can send to a group with or without joining it
  - To join, make a socket join a group (s.joinGroup(group)) enabling it to receive messages to the group

- **Failure model**
  - Omission failures \( \Rightarrow \) some but not all members may receive a message.
    - e.g. a recipient may drop message, or 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. The aim is for members of a group to receive copies of messages sent to the group.
- Many different delivery guarantees are possible.
  - e.g. agree on the set of messages received or on delivery ordering.
- A process can multicast by the use of a single operation instead of a send to each member.
  - For example in IP multicast by Java aSocket.send(aMessage).
  - The single operation allows for:
    - efficiency i.e. send once on each link, using hardware multicast when available, e.g. multicast from a computer in London to two in Beijing.
    - delivery guarantees e.g. can’t make a guarantee if multicast is implemented as multiple sends and the sender fails (or sender fails halfway, leading to some-yes-some-no). IP multicast does not guarantee ordering or reliability, but can be enhanced.

System Model of Multicast

- 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 - which are the destinations of multicast messages.
- In general process $p$ can belong to more than one group (for simplicity of discussion, at most one group at a time here).
- Operations:
  - $multicast(g, m)$ sends message $m$ to all members of process group $g$.
  - $deliver(m)$ is called to get a multicast message delivered. It is different from $receive$ as it may be delayed to allow for ordering or reliability.
- 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, a member delivers to itself
  - they are useful for coordination of groups of cooperating servers
- Open
  - they are useful for notification of events to groups of interested processes

Revisit: Reliability of One-to-One Communication

- **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;
- **integrity**:  
  - the message received is identical to one sent, and no messages are delivered twice (could be received many times).

How do we achieve validity?

- **validity** - by use of acknowledgements and retries

How do we achieve integrity?

- **integrity**  
  - by use checkums, reject duplicates (e.g. due to retries),
  - If allowing for malicious users, use security techniques
Basic Multicast

- A correct process will eventually deliver the message provided the **multicaster does not crash**
  - Does IP multicast give this guarantee?
- The primitives are called **B-multicast** and **B-deliver**
- A straightforward but ineffective method of implementation:
  - use a reliable 1-1 send (i.e. with integrity and validity as above)
    To **B-multicast**(g,m): for each process \( p \in g \), send\((p, m)\);
    On receive \((m)\) at \( p \): **B-deliver** \((m)\) at \( p \)
- Problem
  - if the number of processes is large, the protocol will suffer from **ack-implosion**

What are ack-implosions?

An implementation of B-Multicast may be achieved over IP multicast; how?

Implementation of Basic Multicast over IP Multicast

- How to use IP multicast which does not guarantee the message delivery, with sequence numbers and negative-ack for one-to-one retransmission?
Reliable Multicast

- Reliable multicast 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. Furthermore, \( p \in \text{group}(m) \) and \( m \) was supplied to a multicast operation by \( \text{sender}(m) \)
  - how messages are distinguished?
- Validity - if a correct process multicasts \( m \), it will eventually deliver \( m \) (self-delivery by some other correct process including itself)
- Non-uniform Agreement (atomicity; all or nothing) - if a correct process delivers \( m \) then all correct processes in \( \text{group}(m) \) will eventually deliver \( m \), even if multicaster crashes

Is agreement a property of the B-multicast that is based on a reliable one-to-one send operation? Why or why not?

Reliable Multicast Algorithm (over Basic Multicast)

- Processes can belong to several closed groups

**On initialization**

\[
\text{Received} := \{\};
\]

When a message is B-delivered, the recipient B-multicasts it to the group, then R-delivers it. Duplicates are detected.

**For process \( p \) to R-multicast message \( m \) to group \( g \)**

\[
\text{B-multicast}(g, m);
\]

If \( p \in g \) is included as a destination

**On B-deliver(\( m \)) at process \( q \) with \( g = \text{group}(m) \)**

If \( m \notin \text{Received} \)

\[
\text{Received} := \text{Received} \cup \{m\};
\]

If \( q \neq p \) then B-multicast(\( g, m \)); end if

R-deliver \( m \);

What can you say about the performance of this algorithm?
Reliability Properties of R-Multicast over B-Multicast

• **Validity** - a correct process B-delivers to itself
• **Integrity** - because the reliable 1-1 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 can be implemented efficiently over IP multicast by holding back messages until every member can receive them.

Is this algorithm correct in an asynchronous system? Why or why not?

---

Reliable Multicast over IP Multicast

• **How about the efficiency of R-Multicast based on B-Multicast?**
• **Is IP multicast often successful or not?**
• **Is it necessary that processes send separate ACKs, or process can piggyback ACKs on the messages that they send to the group?**
• **Can R-multicast be constructed based upon IP multicast?**
• **How to proceed?**
Reliable Multicast over IP Multicast

- This protocol assumes groups are closed. It uses:
  - piggybacked acknowledgement messages (what the piggybacked values for?)
  - negative acknowledgements when messages are missed
- Process $p$ maintains:
  - $S_p^g$, a message sequence number for each group it belongs to and
  - $R_p^g$, 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 tive acks for messages received in the form $<q, R_q^g>$
  - IP multicasts the message to $g$, increments $S_p^g$ by 1
- A process on receipt by of a message to $g$ with $S$ from $p$
  - If $S = R_p^g + 1$ R-deliver the 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 $R < R_p^g$ (for enclosed ack $<q,R>$)
    * then it missed messages and requests them (from whom?) with neg-acknowledgements
    * puts new message in hold-back queue for later delivery (necessary?ordering?)

The Hold-back Queue for Arriving Multicast Messages

- The hold back queue is not necessary for reliability as in the implementation using IP multicast, but it simplifies the protocol, allowing sequence numbers to represent sets of messages. Hold-back queues are also used for ordering protocols.
Reliability Properties of R-Multicast over IP-Multicast

- **Integrity** – duplicate messages detected and rejected.
  IP multicast uses checksums to reject corrupt messages.

- **Validity** – due to IP multicast

- **Agreement** – processes can detect missing messages. They must keep copies of messages they can re-transmit.

- Discarding of copies of messages that are no longer needed:
  - When piggybacked acknowledgments arrive, note which processes have received messages. When all processes in $g$ have the message, discard it.
  - Problem of a process that stops sending - use 'heartbeat' messages.

- This protocol has been implemented in a practical way

---

Uniform Properties

- Uniform agreement: If a process, whether it is correct or fails, delivers message $m$, then all correct processes in group$(m)$ will eventually deliver $m$.
  - The protocol is correct even if the multicaster crashes
  - Non-uniform agreement works for correct processes only

```
On initialization
  Received := \{\};

For process $p$ to R-multicast message $m$ to group $g$
  B-multicast($g, m$); // $p \in g$ is included as a destination

On B-deliver($m$) at process $q$ with $g = \text{group}(m)$
  if ($m \notin \text{Received}$) then
    Received := Received $\cup \{m\}$;
    if ($q \neq p$) then B-multicast($g, m$); end if
  R-deliver $m$; // if reorder with "if...end if"
    // non-uniform agreement?
    // uniform agreement?
```

4/6/2011
Ordered Multicast

- The basic multicast algorithm delivers messages to processes in an arbitrary order. A variety of orderings may be implemented:
  - FIFO ordering
    - If a correct process issues `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) \rightarrow multicast(g, m')`, where `\rightarrow` is the happened-before relation between messages in group `g`, then any correct process that delivers `m'` will deliver `m` before `m'` (can a correct process deliver `m`, but not `m'`, still meeting causal ordering?)
  - Total ordering
    - If a correct process delivers message `m` before it delivers `m'`, then any other correct process that delivers `m'` will deliver `m` before `m'`
- Ordering is expensive in delivery latency and bandwidth consumption

Total, FIFO and Causal Ordering of Multicast Messages

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

Note the FIFO-related messages `F_1` and `F_2` and the causally related messages `C_1` and `C_2`. 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. Therefore the less expensive orderings (e.g. FIFO or causal) are chosen for applications for which they are suitable.
Total, FIFO and Causal Ordering of Multicast (Cont.)

- Does causal ordering imply FIFO ordering?
- Can FIFO ordering be partial ordering?
- Can causal ordering be partial ordering?
- Can the totally ordered messages be delivered in the opposite order to the physical time at which they were sent?
- Is total ordering necessarily also a FIFO or causal ordering?
- Does FIFO, causal, or total ordering assume reliability?

- A reliable totally ordered multicast is called atomic multicast
- Hybrids: FIFO-total, causal-total, and their reliable ones, but why no FIFO-causal?

Display from a Bulletin Board Program

- Users run bulletin board applications which multicast messages
- One multicast group per topic (e.g. os.interesting)
- Require reliable multicast - so that all members receive messages
- Ordering:

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Mike</td>
<td>Mach</td>
</tr>
<tr>
<td>24</td>
<td>Joseph</td>
<td>Microkernels</td>
</tr>
<tr>
<td>25</td>
<td>Mike</td>
<td>Re: Microkernels</td>
</tr>
<tr>
<td>26</td>
<td>Kevin</td>
<td>RPC performance</td>
</tr>
<tr>
<td>27</td>
<td>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)
FIFO Ordering over Basic Multicast

- We discuss FIFO ordered multicast with operations FO-multicast and FO-deliver for non-overlapping groups. It 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
  - $R^p_g$, the sequence number of the latest message to $g$ that $p$ delivered from $q$

- For $p$ to FO-multicast a message to $g$, it 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 $S > R^q_g + 1$ then $p$ places message in hold-back queue until intervening messages have been delivered. (note that B-multicast does eventually deliver messages unless the sender crashes)

FIFO Ordering by Reliable Multicast over IP Multicast

- This protocol assumes groups are closed. It uses:
  - piggybacked acknowledgement messages
  - negative acknowledgements when messages are missed

- Process $p$ maintains:
  - $S^p_g$, a message sequence number for each group it belongs to and
  - $R^p_g$, 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 +tive acks for messages received in the form $<q, R^p_g>$
  - IP multicasts the message to $g$, increments $S^p_g$ by 1.

- A process on receipt by of a message to $g$ with $S$ from $p$
  - If $S = R^p_g + 1$ R-deliver the 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 $R < R^p_g$ (for enclosed ack $<q,R>$)
    - then it missed messages and requests them (from whom?) with neg-acknowledgements
    - puts new message in hold-back queue for later delivery (necessary? ordering?)
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
  1. using a sequencer (only for non-overlapping groups)
  2. 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 \(<m_{\text{order}}, i, S>\) with \( g = \text{group}(m_{\text{order}}) \)
     wait until \(<m, i>\) in hold-back queue and \( S = r_g \);
   - TO-deliver \( m \) \( \text{// (after deleting it from the hold-back queue)} \)
     \( r_g = S + 1 \);

2. Algorithm for sequencer of \( g \)
   - On initialization: \( s_g := 0 \)
   - On B-deliver \(<m, i>\) with \( g = \text{group}(m) \)
     B-multicast \(<\text{"order"}, i, s_g>\);
     \( s_g := s_g + 1 \);

A process wishing to TO-multicast \( m \) to \( g \) attaches a unique id, \( \text{id}(m) \) 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 #s 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 the potential problems with using a single sequencer?**

Kaashoek’s protocol uses hardware-based multicast

The sender transmits one message to sequencer, then the sequencer multicasts the sequence number and the message; but IP multicast is not as reliable as B-multicast so the sequencer stores messages in its history buffer for retransmission on request members notice messages are missing by inspecting sequence numbers

**What can the sequencer do about its history buffer becoming full?**

The ISIS Algorithm for Total Ordering

- ISIS: the process collectively agree on the assignment of sequence numbers to messages in a distributed fashion

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
**ISIS Total Ordering - Agreement of Sequence Numbers**

- Each process, \( q \) keeps:
  - \( A_q \): the largest agreed sequence number it has seen and
  - \( P_q \): 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
  - \( P_q := \text{Max}(A_q, P_q) + 1 \).
  - 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 \( A_q := \text{Max}(A_q, a) \), attach \( a \) to the message and re-order hold-back queue.

---

**Discussion of Total Ordering in ISIS Protocol**

- **Hold-back queue**
  - proof of total ordering

- ordered with the message with the smallest sequence number at the front of the queue

- when the agreed number is added to a message, the queue is re-ordered

- when the message at the front has an agreed id, it is transferred to the delivery queue
  - even if agreed, those not at the front of the queue are not transferred

- every process agrees on the same order and delivers messages in that 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 happened before relation (on multicast messages only)
  - that is, ordering imposed by one-to-one messages is not taken into account
- It uses vector timestamps - that count the number of multicast messages from each process that happened before the next message to be multicast

Causal Ordering Using Vector Timestamps

- Each process has its own vector timestamp
- 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

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-multicasting ($g, <V_i^g, m>$)**

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

CO-deliver $m$;  // after removing it from the hold-back queue

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

Note: a process can immediately CO-deliver to itself its own messages (not shown)
Example: Causal ordering using vector timestamps

What this node got to deliver?

Comments on Causal Ordering Using Vector Timestamps

- 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
  \( V[j] := \max(V[j], t[j]) \) 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’ etc 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.)

Consensus – not covered in class
Summary

- Multicast communication can specify requirements for reliability and ordering, in terms of integrity, validity and agreement
- B-multicast
  - a correct process will eventually deliver a message provided the multicaster does not crash
- reliable multicast
  - in which the correct processes agree on the set of messages to be delivered;
  - we showed two implementations: over B-multicast and IP multicast
- delivery ordering
  - FIFO, total and causal delivery ordering.
  - FIFO ordering by means of senders’ sequence numbers
  - total ordering by means of a sequencer or 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