Multicast for coordination

Marco Aiello

based on cdk5.net

Introduction to multicast

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

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

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

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

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 in java *aSocket.send*(*aMessage*)

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 in java *aSocket.send*(*aMessage*)
  - The single operation allows for:

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 in 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

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 in 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. Can also do ordering

Revision of IP multicast

Revision of IP multicast

- IP multicast – an implementation of group communication

Revision of IP multicast

- IP multicast – an implementation of group communication
  - built on top of IP (note IP packets are addressed to computers)

Revision 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)

Revision 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

Revision 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 multicast, send a UDP datagram with a multicast address

Revision 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 multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup*(*group*) - Fig 4.17) enabling it to receive messages to the group

Revision 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 multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup*(*group*) - Fig 4.17) enabling it to receive messages to the group
- Multicast routers

Revision 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 multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup(group)* - Fig 4.17) enabling it to receive messages to the group
- Multicast routers
  - Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.

Revision 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 multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup*(*group*) - Fig 4.17) enabling it to receive messages to the group
- Multicast routers
  - Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.
- Failure model

Revision 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 multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup*(*group*) - Fig 4.17) enabling it to receive messages to the group
- Multicast routers
  - Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.
- Failure model
  - Omission failures ⇒ some but not all members may receive a message.

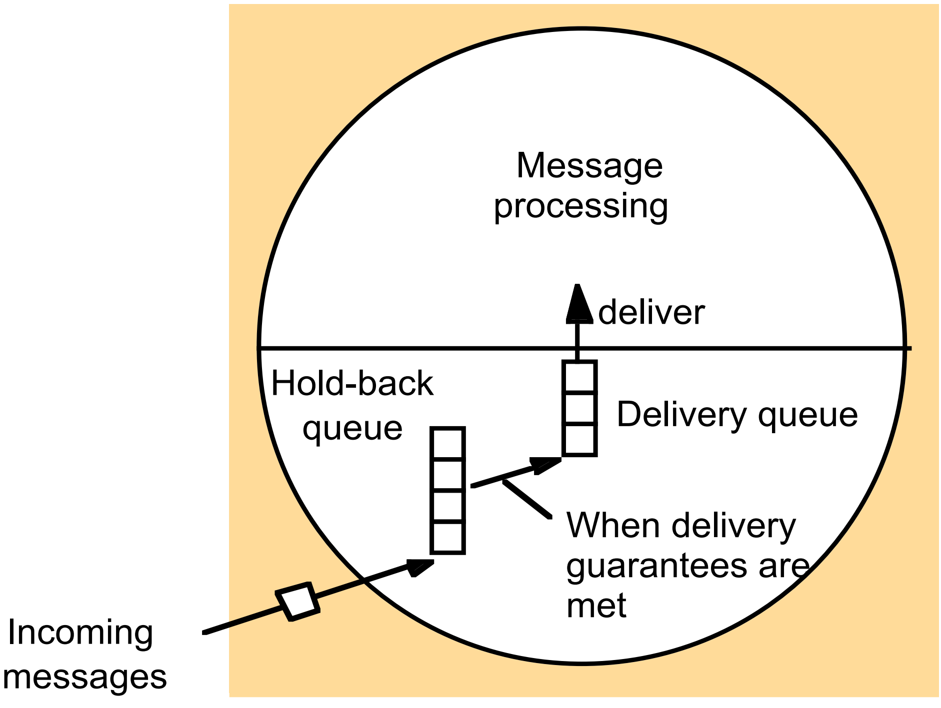
Revision of IP multicast

- 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 multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup(group)* - Fig 4.17) enabling it to receive messages to the group
- Multicast routers
  - Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.
- Failure model
  - Omission failures ⇒ some but not all members may receive a message.
    - e.g. a recipient may drop message, or a multicast router may fail

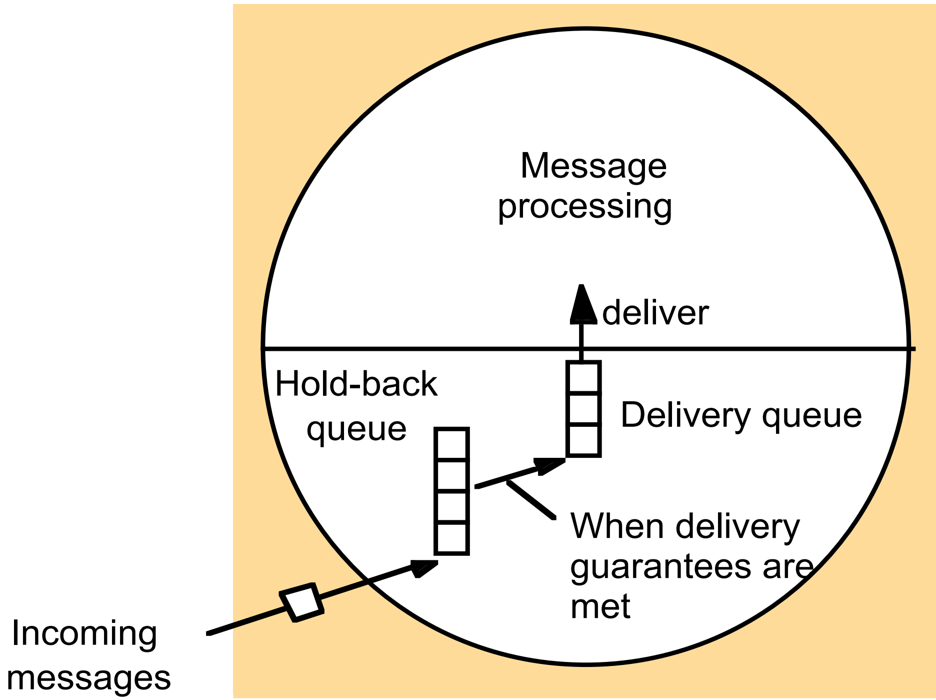
Revision of IP multicast

- 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 multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup*(*group*) - Fig 4.17) enabling it to receive messages to the group
- Multicast routers
  - Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.
- Failure model
  - Omission failures ⇒ 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

The hold-back queue for arriving multicast messages



The hold-back queue for arriving multicast messages



System model

System model

- The system consists of a collection of processes which can communicate *reliably* over 1-1 channels

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)

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 - which are the destinations of multicast messages

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 - which are the destinations of multicast messages
- In general process *p* can belong to more than one group

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 - which are the 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*) is called to get a multicast message delivered. It is different from *receive* as it may be delayed to allow for ordering or reliability.

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 - which are the 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*) 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*)

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 - which are the 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*) 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

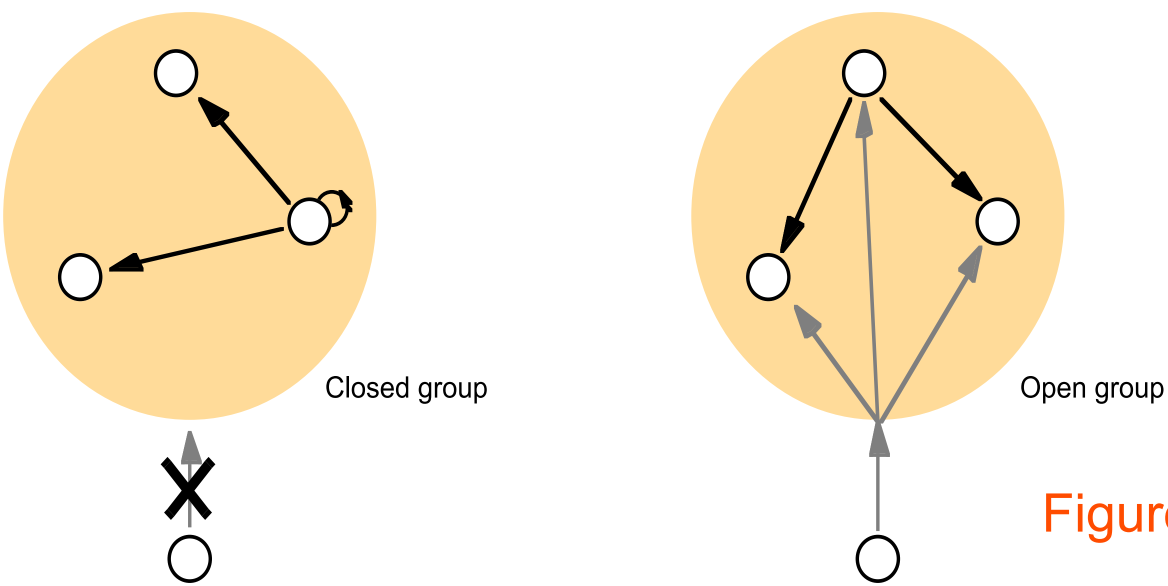


Figure 11.9

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

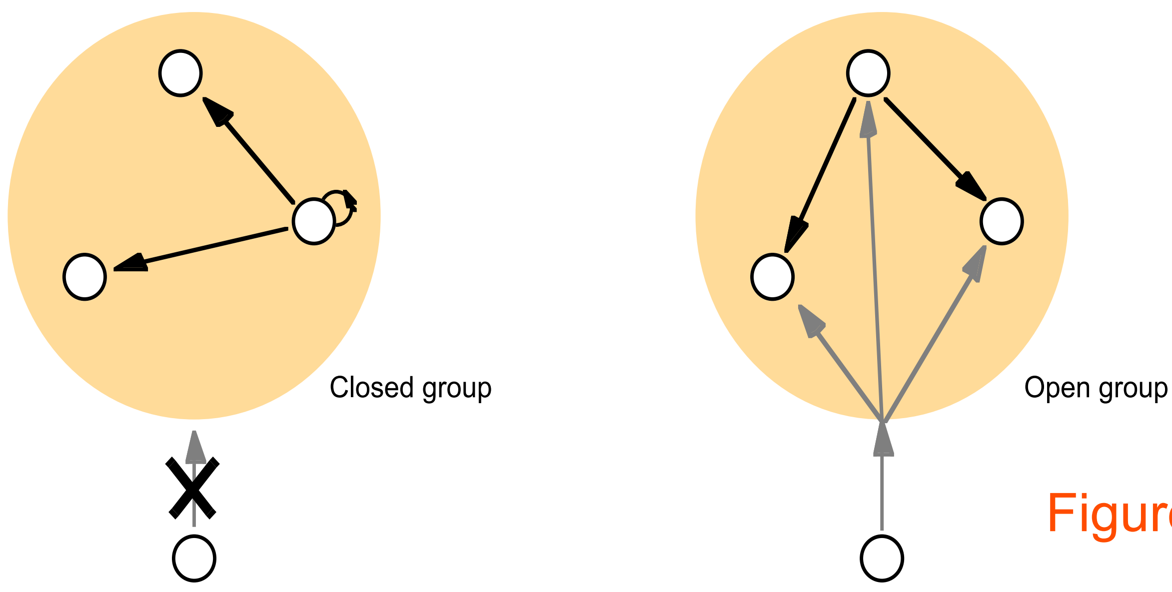


Figure 11.9

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

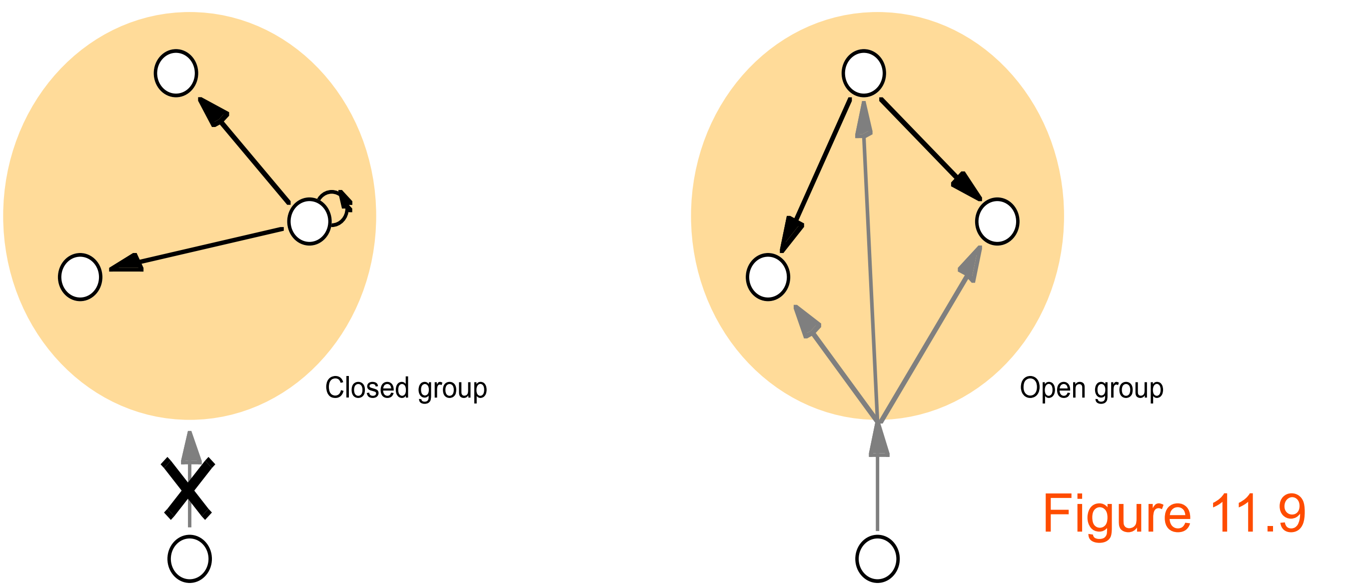


Figure 11.9

Open and closed groups

Does IP multicast support 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

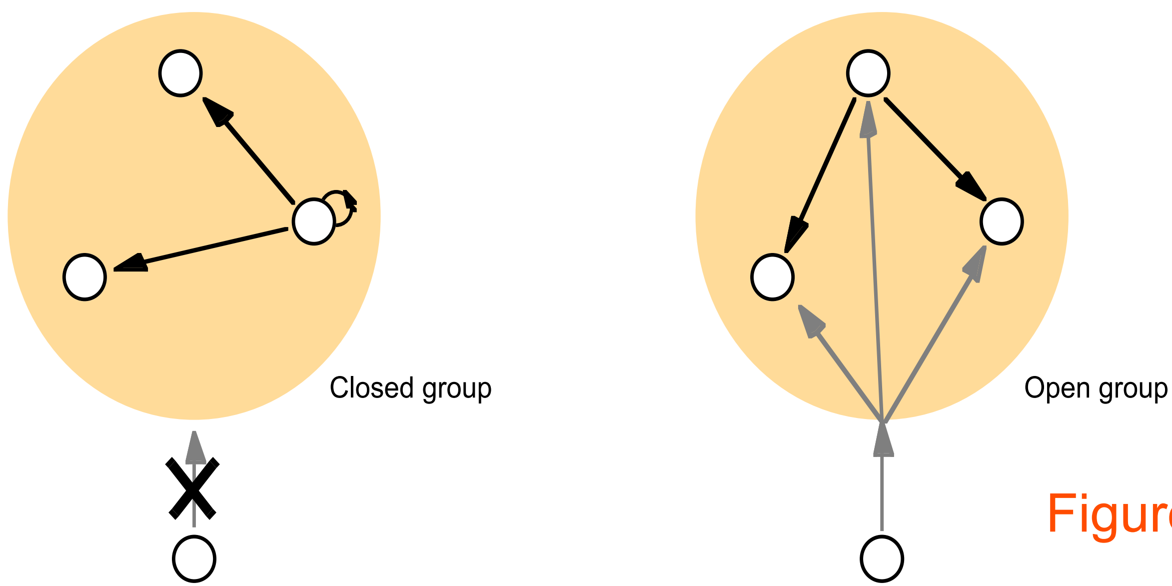


Figure 11.9

Reliability of one-to-one communication

Reliability of one-to-one communication

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:

Reliability of one-to-one communication

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:
- *validity*:
  - any message in the outgoing message buffer is eventually delivered to the incoming message buffer;

Reliability of one-to-one communication

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:
- *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.

How do we achieve validity?

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:
- *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.

How do we achieve validity?

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:
- *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.

validity - by use of acknowledgements and retries

How do we achieve integrity?

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:
- *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.

validity - by use of acknowledgements and retries

How do we achieve integrity?

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:
- *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.

validity - by use of acknowledgements and retries

integrity

- *by* use checksums, reject duplicates (e.g. due to retries).
- If considering malicious users in the system model, use security techniques

Basic multicast

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

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 *B-multicast* and *B-deliver*

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 *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 $p \varepsilon g$, *send*(*p*, *m*);

    On *receive* (*m*) at *p*: *B-deliver* (*m*) at *p*

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 *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?

- 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 *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?

- 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 *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*

A practical implementation of Basic Multicast may be achieved over IP multicast

Implementation of basic multicast over IP

Implementation of basic multicast over IP

- Each process *p* maintains:

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and

Implementation of basic multicast over IP

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

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g
  - it piggybacks S_g^p on the message m, using IP multicast to send it

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g
  - it piggybacks S_g^p on the message m, using IP multicast to send it
  - the piggybacked sequence numbers allow recipients to learn about messages they have not received

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g
  - it piggybacks S_g^p on the message m, using IP multicast to send it
  - the piggybacked sequence numbers allow recipients to learn about messages they have not received
- On *receive* (g, m, S) at p:

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g
  - it piggybacks S_g^p on the message m, using IP multicast to send it
  - the 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

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g
  - it piggybacks S_g^p on the message m, using IP multicast to send it
  - the 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 the message because it has been delivered before

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g
  - it piggybacks S_g^p on the message m, using IP multicast to send it
  - the 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 the message because it has been delivered before
  - if $S > R^q_g + 1$ note that a message is missing, request missing message from sender. (will use a hold-back queue to be discussed later on)

Implementation of basic multicast over IP

- Each process p maintains:
  - a sequence number, S^p_g for each group it belongs to and
  - R^q_g, the sequence number of the latest message it has delivered from process q
- For process p to *B-multicast* a message m to group g
  - it piggybacks S_g^p on the message m, using IP multicast to send it
  - the 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 the message because it has been delivered before
  - if $S > R^q_g + 1$ note that a message is missing, request missing message from sender. (will use a hold-back queue to be discussed later on)
- If the sender crashes, then a message may be delivered to some members of the group but not others.

Reliable multicast

Reliable multicast

- The protocol is correct even if the multicaster crashes

Reliable multicast

- The protocol is correct even if the multicaster crashes
- it satisfies criteria for *validity, integrity* and *agreement*

Reliable multicast

- The protocol is correct even if the multicaster crashes
- it satisfies criteria for *validity, integrity* and *agreement*
- it provides operations *R-multicast* and *R-deliver*

Reliable multicast

- The 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 $p \in group(m)$ and *m* was supplied to a multicast operation by *sender*(*m*)

Reliable multicast

- The 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* ε *group*(*m*) and *m* was supplied to a multicast operation by *sender*(*m*)
- *Validity* - if a correct process multicasts *m*, it will eventually deliver *m*

Reliable multicast

- The 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* ε *group*(*m*) and *m* was supplied to a multicast operation by *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 *group*(*m*) will eventually deliver *m*

Reliable multicast

- The 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* ε *group*(*m*) and *m* was supplied to a multicast operation by *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 *group*(*m*) will eventually deliver *m*

integrity as for 1-1 communication

Reliable multicast

- The 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* ε *group*(*m*) and *m* was supplied to a multicast operation by *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 *group*(*m*) will eventually deliver *m*

integrity as for 1-1 communication

validity - simplify by choosing sender as the one process

Reliable multicast

- The 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 $p \in group(m)$ and *m* was supplied to a multicast operation by *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 *group*(*m*) will eventually deliver *m*

integrity as for 1-1 communication

validity - simplify by choosing sender as the one process

agreement - all or nothing - atomicity, even if multicaster crashes

- processes can belong to several closed groups

?processes can belong to several closed groups

Figure 11.10

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 = group(m)
    if $(m \notin Received)$
    then
        $Received := Received \cup \{m\};$
        if $(q \neq p)$ *then B-multicast(g, m); end if*
        R-deliver m;
    end if

On initialization

  Received := {};

Figure 11.10

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* $g = group(m)$

  if ($m \notin Received$)

  then

    $Received := Received \cup \{m\}$;

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

    R-deliver m;

  end if

primitives R-multicast and R-deliver

to R-multicast a message, a process B-multicasts it to processes in the group including itself

On initialization

  Received := {};

Figure 11.10

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 = group*(*m*)

  if ($m \notin$ *Received*)

  then

    Received := *Received* $\cup \{m\}$;

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

    R-deliver m;

  end if

primitives R-multicast and R-deliver

to R-multicast a message, a process B-multicasts it to processes in the group including itself

On initialization

Received := {};

Figure 11.10

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

For process p

B-multica

On B-deliver(m) at process q with g = group(m)

if $(m \notin Received)$

then

$Received := Received \cup \{m\};$

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

R-deliver m;

end if

Validity - a correct process will B-deliver to itself

to R-multicast a message, a process B-multicasts it to processes in the group including itself

Figure 11.10

On initialization
    Received := {};

For process
    B-multica

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

On B-deliver(m) at process q with g = group(m)
    if ($m \notin$ *Received*)
    then
        Received := *Received* $\cup$ {*m*};
        if ($q \neq p$) *then B-multicast(g, m); end if*
        R-deliver m;
    end if

Integrity - because the reliable 1-1 channels used for *B-multicast* guarantee integrity

to R-multicast a message, a process B-multicasts it to processes in the group including itself

On initialization

Received :

Figure 11.10

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

For process

B-multica

On B-deliver(m) at process q with g = group(m)

if $(m \notin Received)$

then

$Received := Received \cup \{m\}$;

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

R-deliver m;

end if

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.

to R-multicast a message, a process B-multicasts it to processes in the group including itself

Figure 11.10

On initialization

Received := {};

For process p

B-multica

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

On B-deliver(m) at process q with g = group(m)

if $(m \notin Received)$

then

$Received := Received \cup \{m\}$;

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

R-deliver m;

end if

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.

to R-multicast a message, a process B-multicasts it to processes in the group including itself

On initialization

Received := {};

Figure 11.10

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

For process p

B-multica

On B-deliver(m) at process q with g = group(m)

if ($m \notin$ *Received*)

then

Received := *Received* $\cup$ {*m*};

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

R-deliver m;

end if

What can you say about the performance of this algorithm?

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.

to R-multicast a message, a process B-multicasts it to processes in the group including itself

Figure 11.10

On initialization

Received := {};

For process p ...

B-multica...

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

On B-deliver(m) *at process q with* $g = group(m)$

if $(m \notin Received)$

then

$Received := Received \cup \{m\}$;

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

R-deliver m;

end if

Is this algorithm correct in an asynchronous system? algorithm?

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.

to R-multicast a message, a process B-multicasts it to processes in the group including itself

Figure 11.10

On initialization
Received := {};

For process p
B-multica

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

On B-deliver(m) *at process q with* $g = group(m)$
if $(m \notin Received)$
then
$Received := Received \cup \{m\}$;
if $(q \neq p)$ *then B-multicast*(g, m); *end if*
R-deliver m;
end if

Reliable multicast can be implemented efficiently over IP multicast by holding back messages until every member can receive them.

Reliable multicast over IP multicast

Reliable multicast over IP multicast

- This protocol assumes groups are closed. It uses:

Reliable multicast over IP multicast

- This protocol assumes groups are closed. It uses:
  - piggybacked acknowledgement messages

Reliable multicast over IP multicast

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

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:

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

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^q_g sequence number of latest message received from process *q* to *g*

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^q_g sequence number of latest message received from process *q* to *g*
- For process *p* to *R-multicast* message *m* to group *g*

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^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 give acks for messages received in the form <*q*, $R^q g$ >

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^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 give 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

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

the piggybacked values in a message allow recipients to learn about messages they have not yet received

- For process *p* to *R-multicast* message *m* to group *g*
  - piggyback S^p_g and give 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

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

the piggybacked values in a message allow recipients to learn about messages they have not yet received

- For process *p* to *R-multicast* message *m* to group *g*
  - piggyback S^p_g and give 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*

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:

the piggybacked values in a message allow recipients to learn about messages they have not yet received

- For process *p* to *R-multicast* message *m* to group *g*
  - piggyback S^p_g and give 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*
  - Iff $S=R^p_g+1$ *R-deliver* the message and increment R^p_g by 1

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:

the piggybacked values in a message allow recipients to learn about messages they have not yet received

- For process *p* to *R-multicast* message *m* to group *g*
  - piggyback S^p_g and give 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*
  - Iff $S=R^p_g+1$ *R-deliver* the message and increment R^p_g by 1
  - If $S \leq R^p_g$ discard the message

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:

the piggybacked values in a message allow recipients to learn about messages they have not yet received

- For process *p* to *R-multicast* message *m* to group *g*
  - piggyback S^p_g and give 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*
  - Iff $S=R^p_g+1$ *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 $R<R^q_g$ (for enclosed ack $<q,R>$)

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:

the piggybacked values in a message allow recipients to learn about messages they have not yet received

- For process p to *R-multicast* message m to group g
  - piggyback S^p_g and give 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
  - Iff $S=R^p_g+1$ *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 $R < R^q_g$ (for enclosed ack <q,R>)
    - then it has missed messages and requests them with negative acknowledgements

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:

the piggybacked values in a message allow recipients to learn about messages they have not yet received

- For process *p* to *R-multicast* message *m* to group *g*
  - piggyback S^p_g and give 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*
  - Iff $S=R^p_g+1$ *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 $R<R^q_g$ (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

The hold-back queue for arriving multicast messages

The hold-back queue for arriving multicast messages

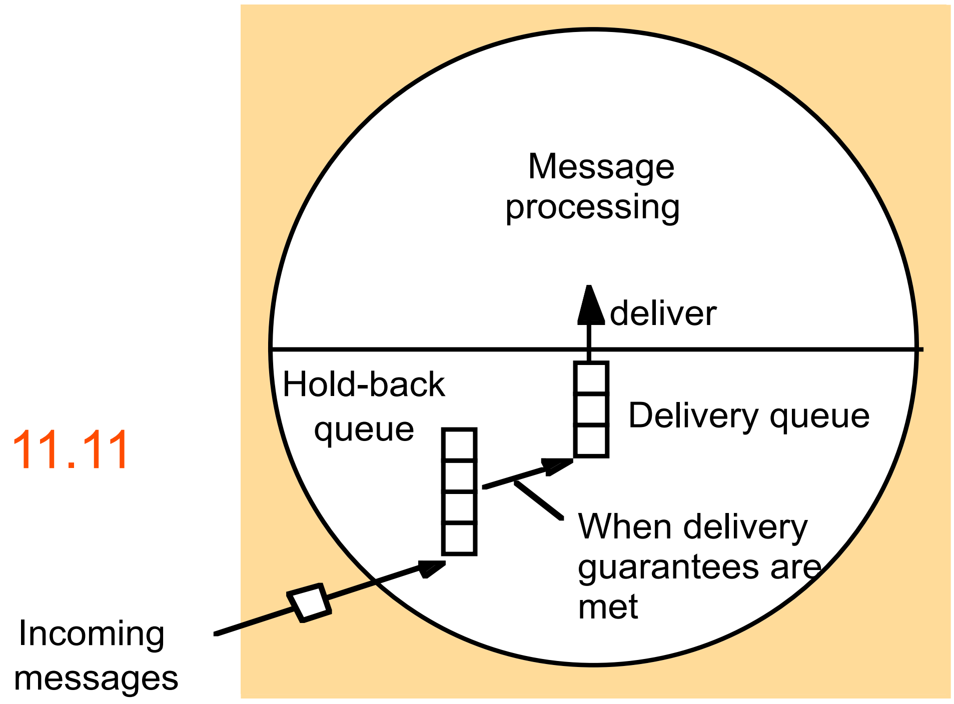
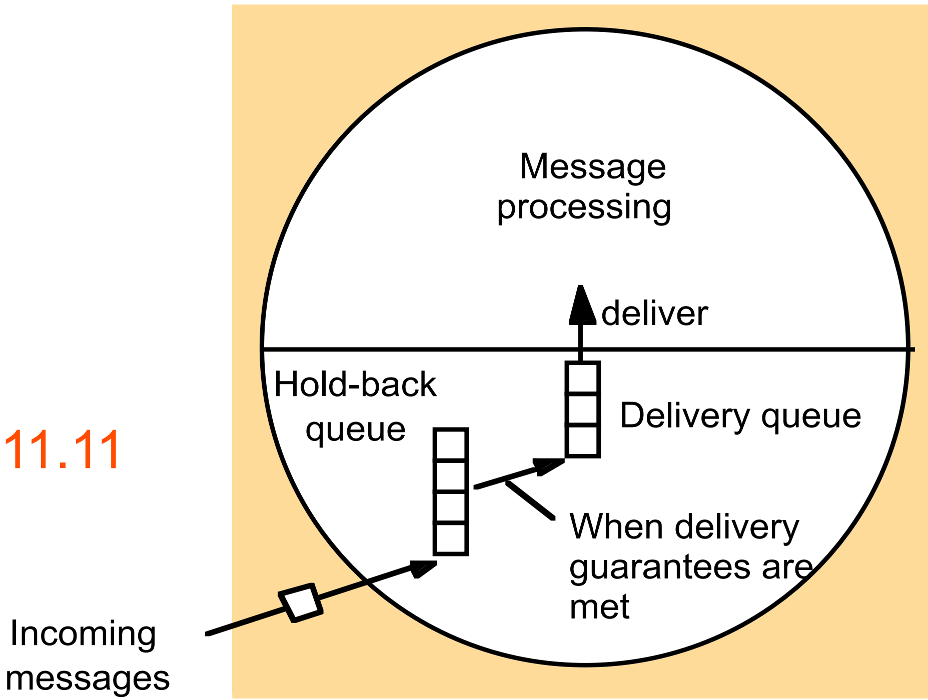


Figure 11.11

The hold-back queue for arriving multicast messages

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

Figure 11.11



Reliability properties of reliable multicast over IP

Reliability properties of reliable multicast over IP

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

Reliability properties of reliable multicast over IP

- *Integrity* - duplicate messages detected and rejected. IP multicast uses checksums to reject corrupt messages
- *Validity* - due to IP multicast in which sender delivers to itself

Reliability properties of reliable multicast over IP

- *Integrity* - duplicate messages detected and rejected. IP multicast uses checksums to reject corrupt messages
- *Validity* - due to IP multicast in which sender delivers to itself
- *Agreement* - processes can detect missing messages. They must keep copies of messages they have delivered so that they can re-transmit them to others.

Reliability properties of reliable multicast over IP

- *Integrity* - duplicate messages detected and rejected. IP multicast uses checksums to reject corrupt messages
- *Validity* - due to IP multicast in which sender delivers to itself
- *Agreement* - processes can detect missing messages. They must keep copies of messages they have delivered so that they can re-transmit them to others.
- discarding of copies of messages that are no longer needed :
  - when piggybacked acknowledgements 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.

Ordered multicast

Ordered multicast

- The basic multicast algorithm delivers messages to processes in an arbitrary order. A variety of orderings may be implemented:

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'* .

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'* .

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'* .
- 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'*.

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'* .
- 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

Total, FIFO and causal ordering of multicast messages

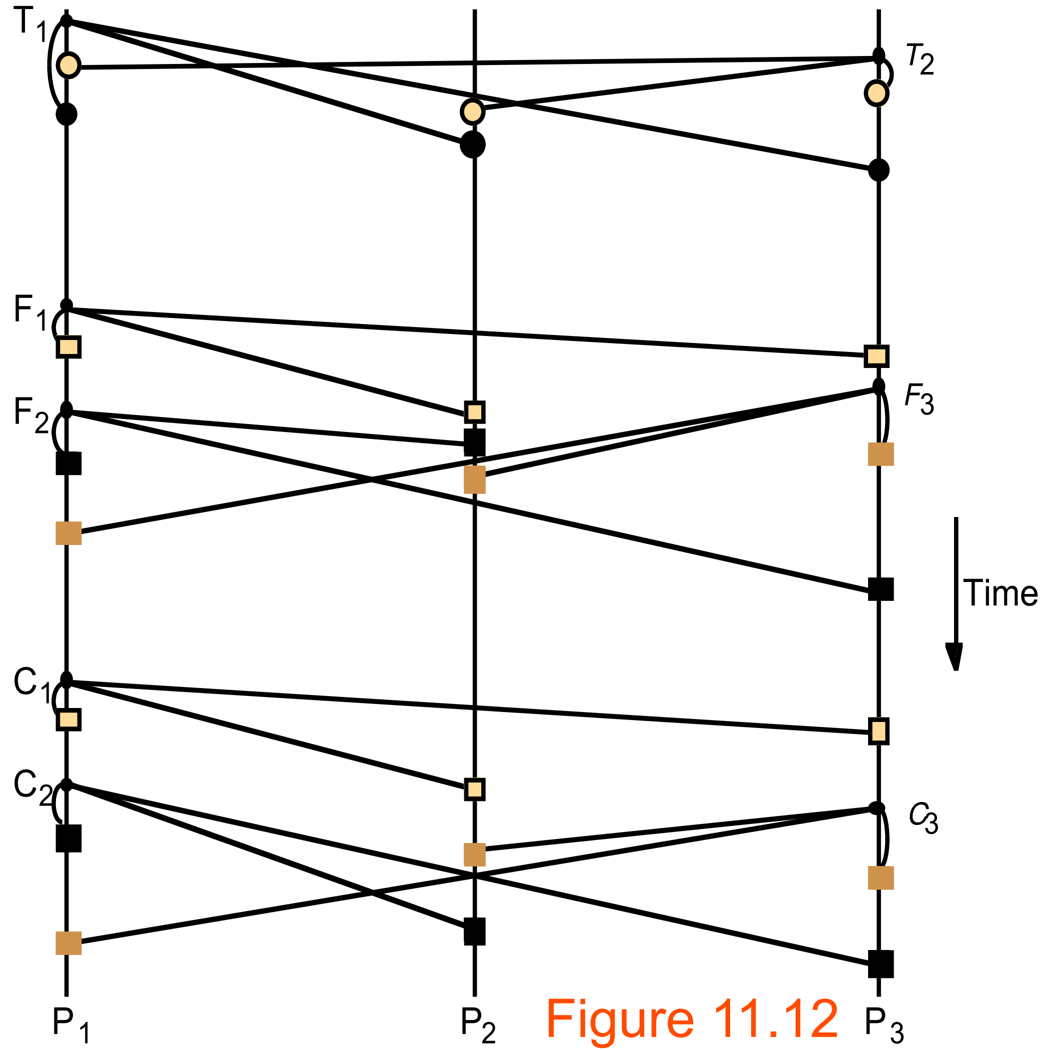


Figure 11.12

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

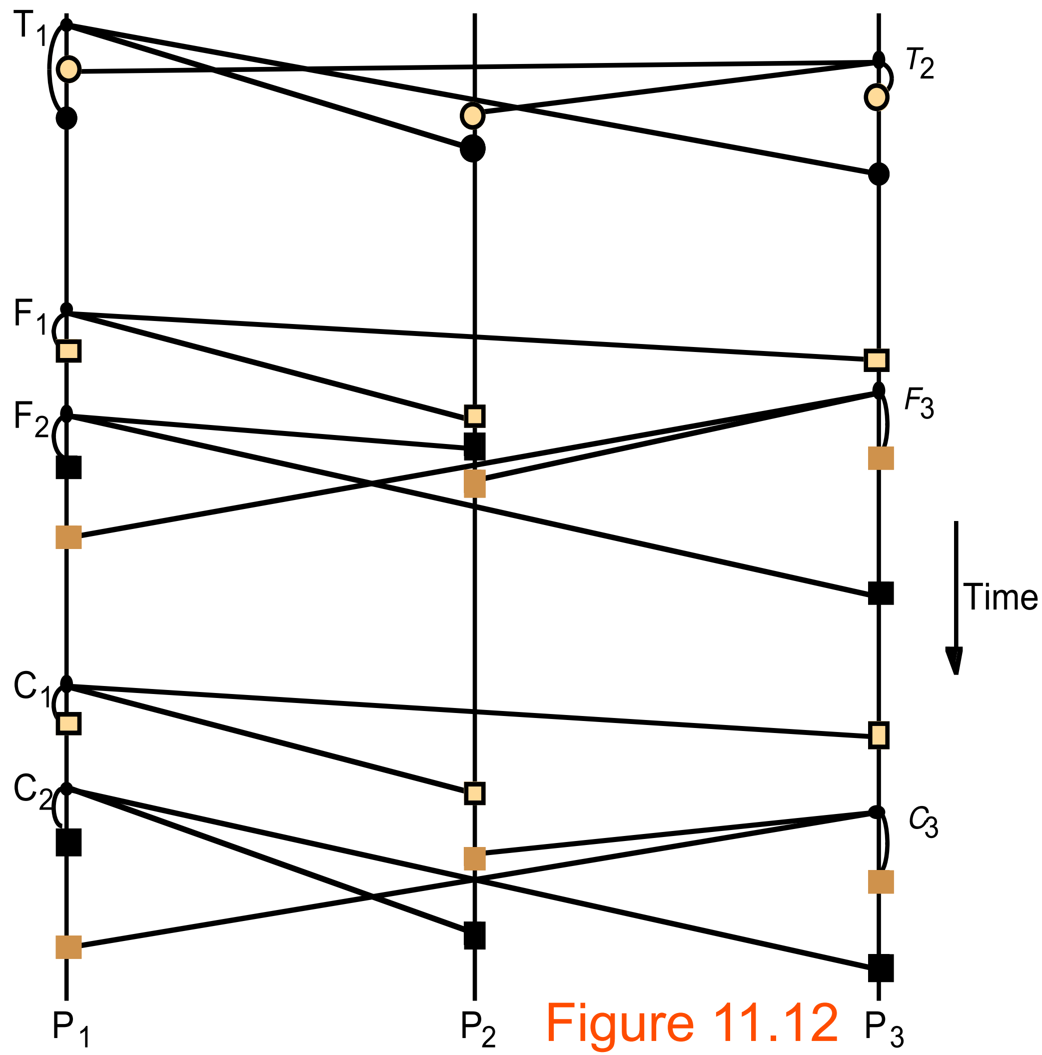


Figure 11.12

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

Note the FIFO-related messages F_1 and F_2

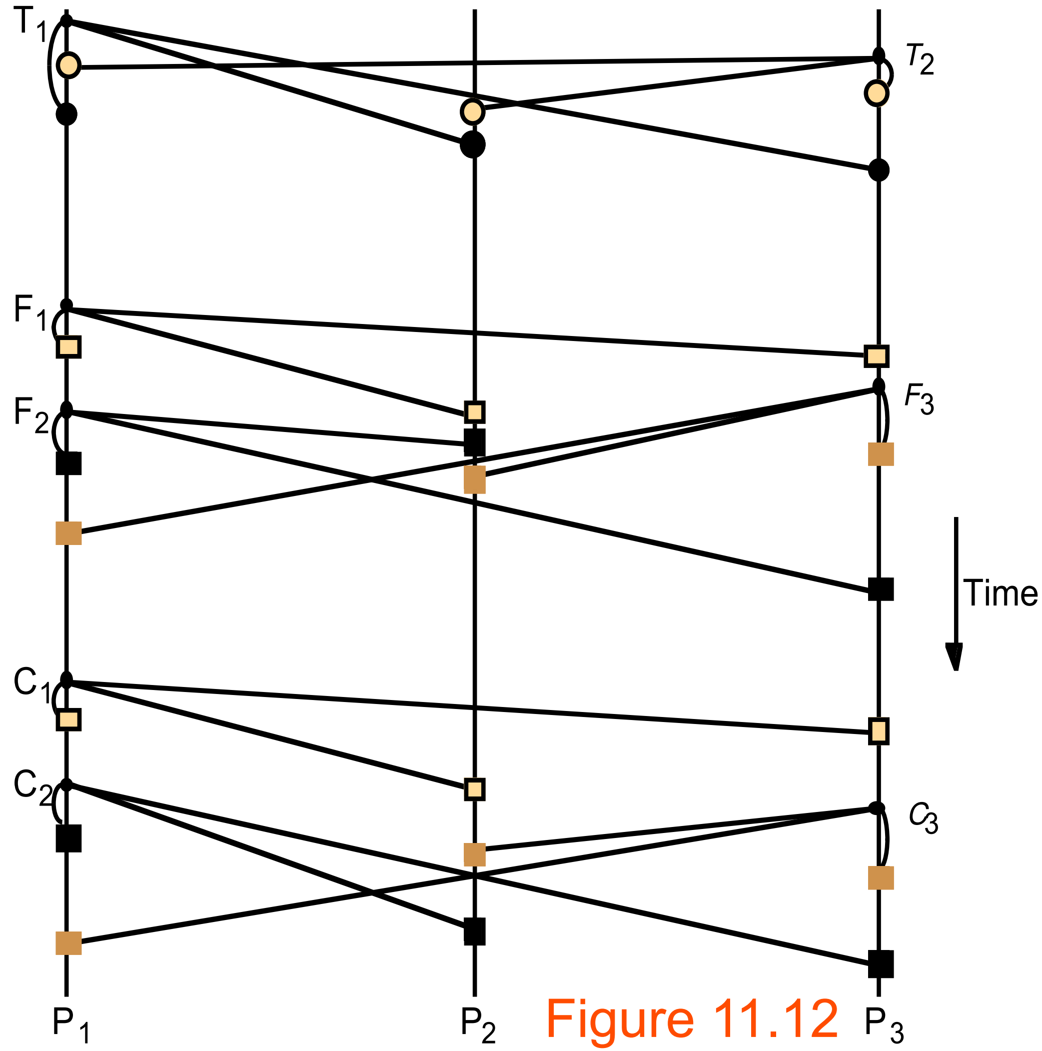


Figure 11.12

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

Note the FIFO-related messages F_1 and F_2

and the causally related messages C_1 and C_3

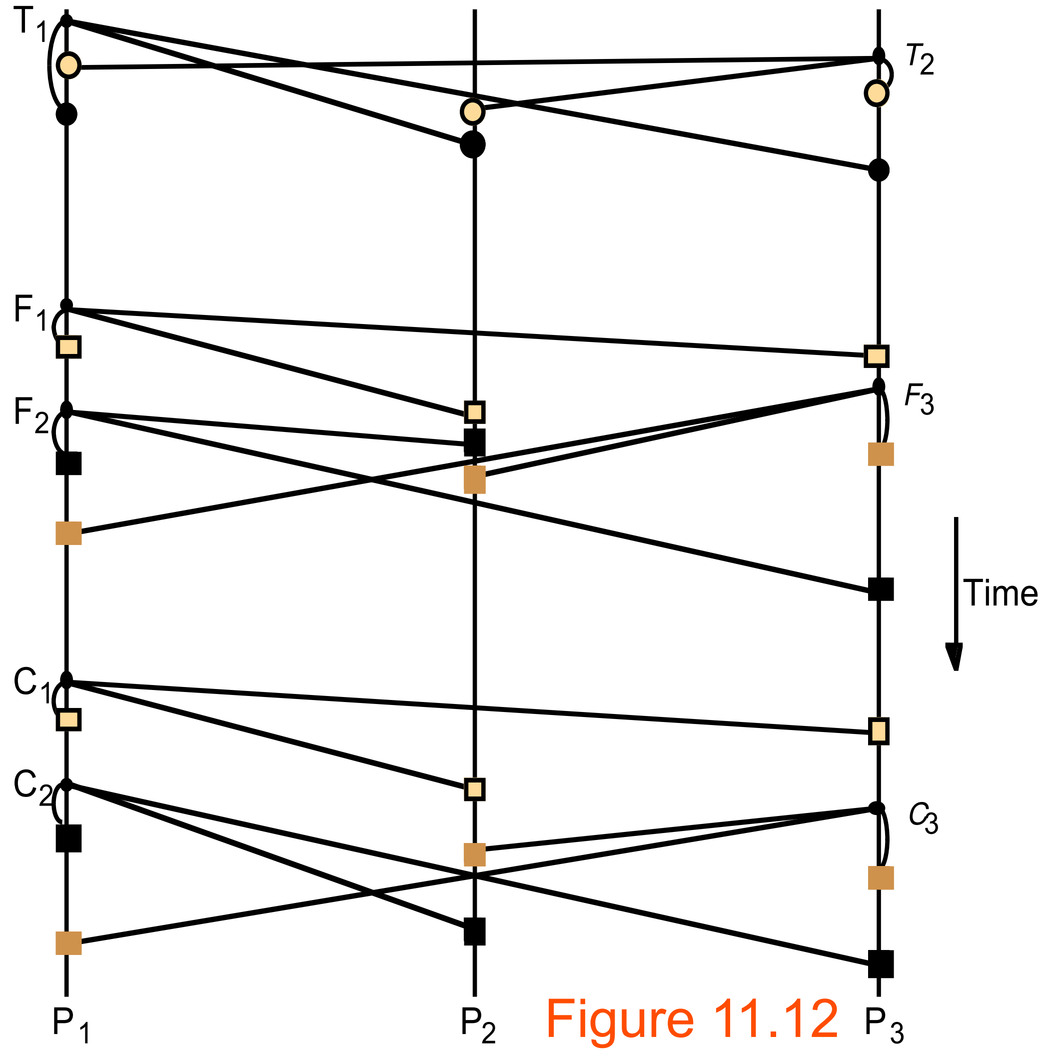


Figure 11.12

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

Note the FIFO-related messages F_1 and F_2

and the causally related messages C_1 and C_3

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

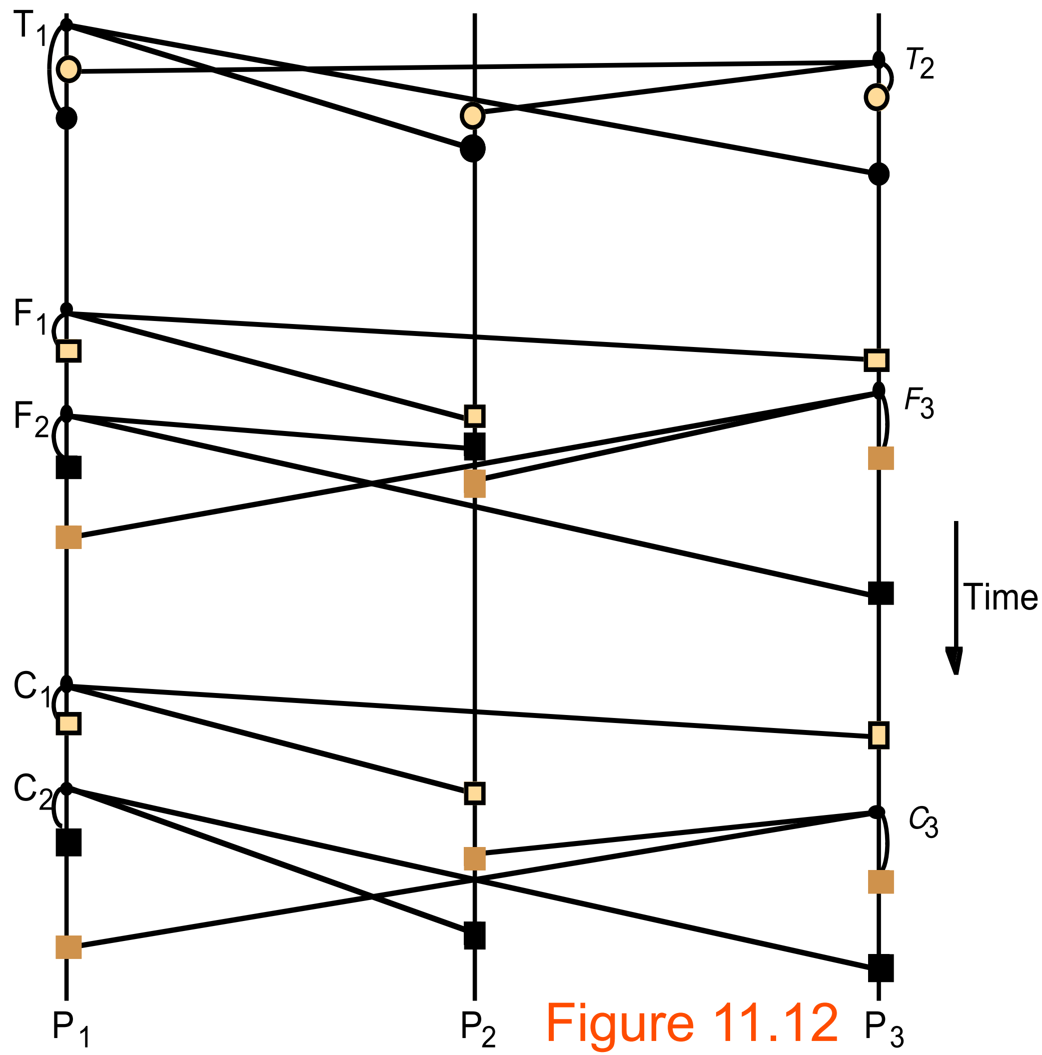


Figure 11.12

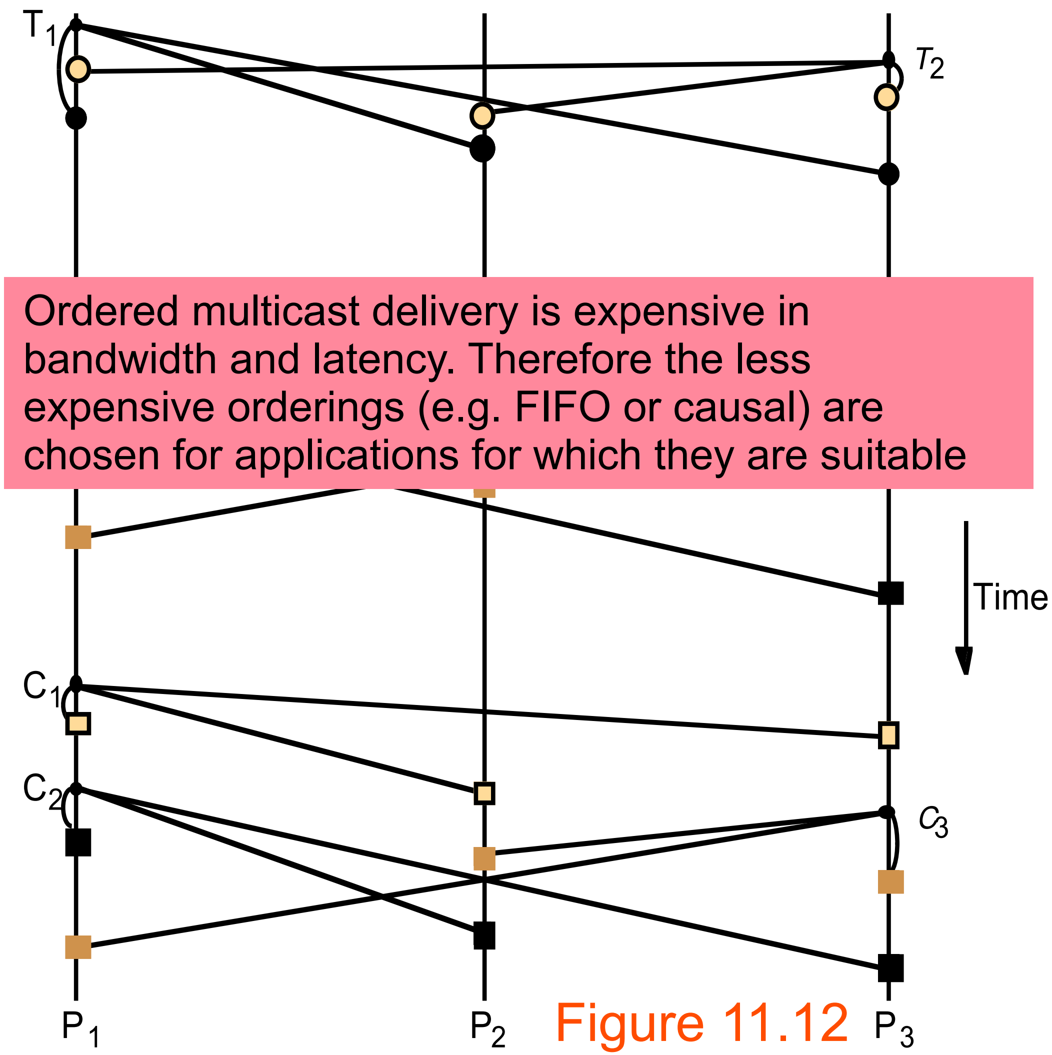
Total, FIFO and causal ordering of multicast messages

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

Note the FIFO-related messages F_1 and F_2

and the causally related messages C_1 and 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. Therefore the less expensive orderings (e.g. FIFO or causal) are chosen for applications for which they are suitable

Figure 11.12

Display from a bulletin board program

Display from a bulletin board program

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 11.13

Display from a bulletin board program

- Users run bulletin board applications which multicast messages

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 11.13

Display from a bulletin board program

- Users run bulletin board applications which multicast messages
- One multicast group per topic (e.g. *os.interesting*)

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 11.13

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

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 11.13

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:

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 11.13

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:

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

FIFO (gives sender order

Figure 11.13

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:

Bulletin board: *os.interesting*

Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

causal (makes replies come after original message)

FIFO (gives sender order

Figure 11.13

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:

total (makes the numbers the same at all sites)

Bulletin board: *os.interesting*

Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

causal (makes replies come after original message)

FIFO (gives sender order

Figure 11.13

Implementation of FIFO ordering over basic multicast

Implementation of 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

Implementation of 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^q_g the sequence number of the latest message to *g* that *p* delivered from *q*

Implementation of 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^q_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

Implementation of 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^q_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.

Implementation of 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^q_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)

Implementation of totally ordered multicast

Implementation of totally ordered multicast

- The general approach is to attach *totally ordered identifiers* to multicast messages

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

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

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*

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

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)

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

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$(g \cup \{sequencer(g)\}, <m, i>)$;

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

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

Figure 11.14

Total ordering using a sequencer

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

1. Algorithm for group membe

On initialization: $r_g := 0$;

To TO-multicast message m to group g

    B-multicast($g \cup \{sequencer(g)\}$, $<m, i>$);

On B-deliver($<m, i>$) *with* $g = group(m)$

    Place $<m, i>$ in hold-back queue;

On B-deliver($m_{order} = <$“order”, $i, S>$) *with* $g = group(m_{order})$

    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* $g = group(m)$

    B-multicast(g, $<$“order”, $i, s_g>$);

    $s_g := s_g + 1$;

Figure 11.14

Total ordering using a sequencer

1. Algorithm for group member

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

On initialization: $r_g := 0$;

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

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

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

2. Algorithm for sequencer of *g*

On initialization: $s_g := 0$;

On B-deliver$(<m, i>)$ *with* $g = group(m)$
$\quad$*B-multicast*$(g, <\text{"order"}, i, s_g>)$;
$\quad s_g := s_g + 1$;

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.

Figure 11.1

Total ordering using a sequencer

1. Algorithm for group member

On initialization: $r_g := 0$;

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

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

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

Figure 11.14

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

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.

Total ordering using a sequencer

1. Algorithm for group membe

On initialization: $r_g := 0$;

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

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

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

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<m, i>) *with* $g = group(m)$
    B-multicast(g, <"order", i, s_g>);
    $s_g := s_g + 1$;

Figure 11.1

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

Discussion of sequencer protocol

- Since sequence numbers are defined by a sequencer, we have total ordering.

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

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 the potential problems with using a single sequencer?

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 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?

- 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 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

Members piggyback on their messages the latest sequence number they have seen

full?

- 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 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

ull?

seen

What happens when some member stops multicasting?

- 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 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

Members that do not multicast send heartbeat messages (with a sequence number)

- 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 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

The ISIS algorithm for total ordering

The ISIS algorithm for total ordering

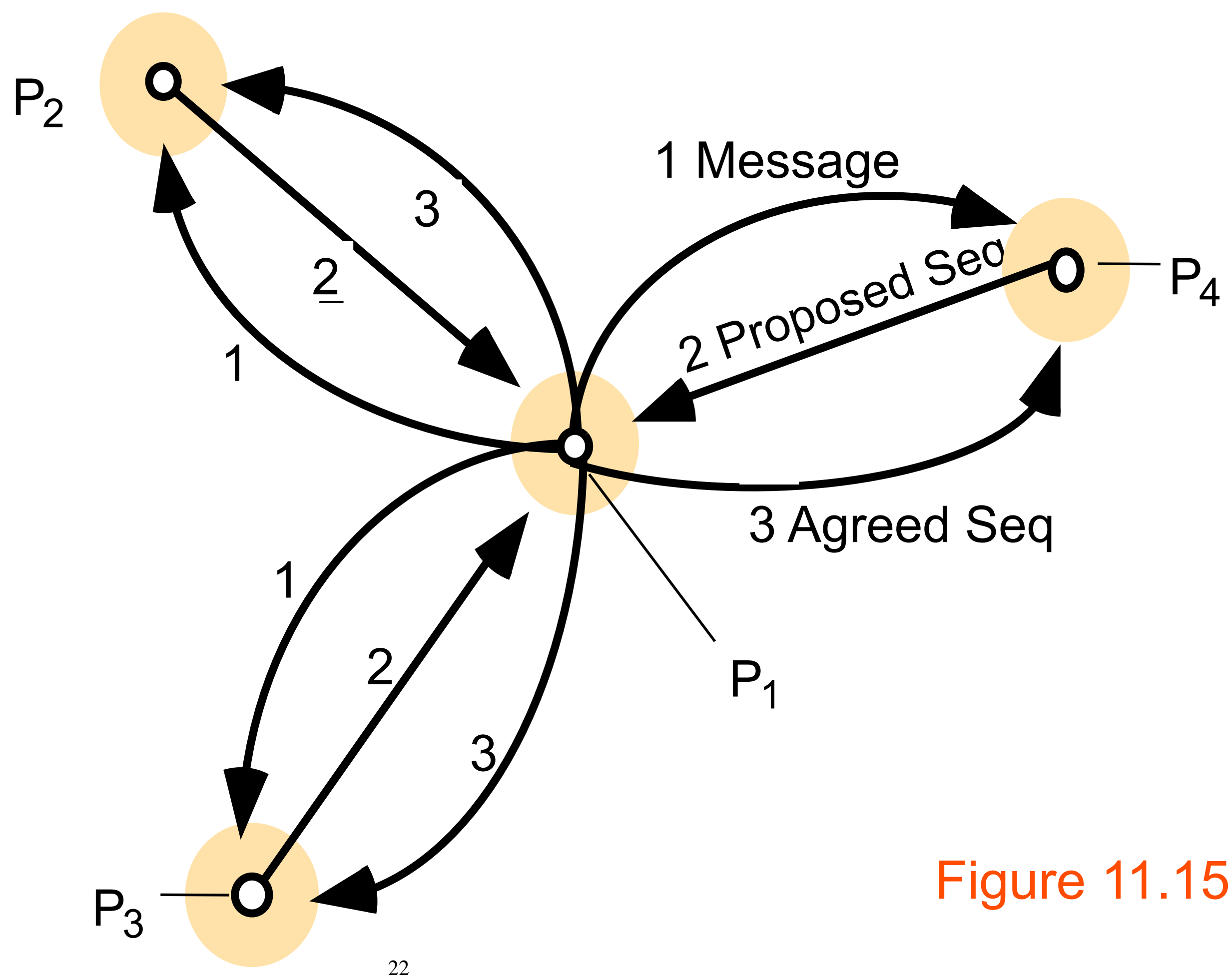


Figure 11.15

The ISIS algorithm for total ordering

1. the process *P1 B-multicasts* a message to members of the group

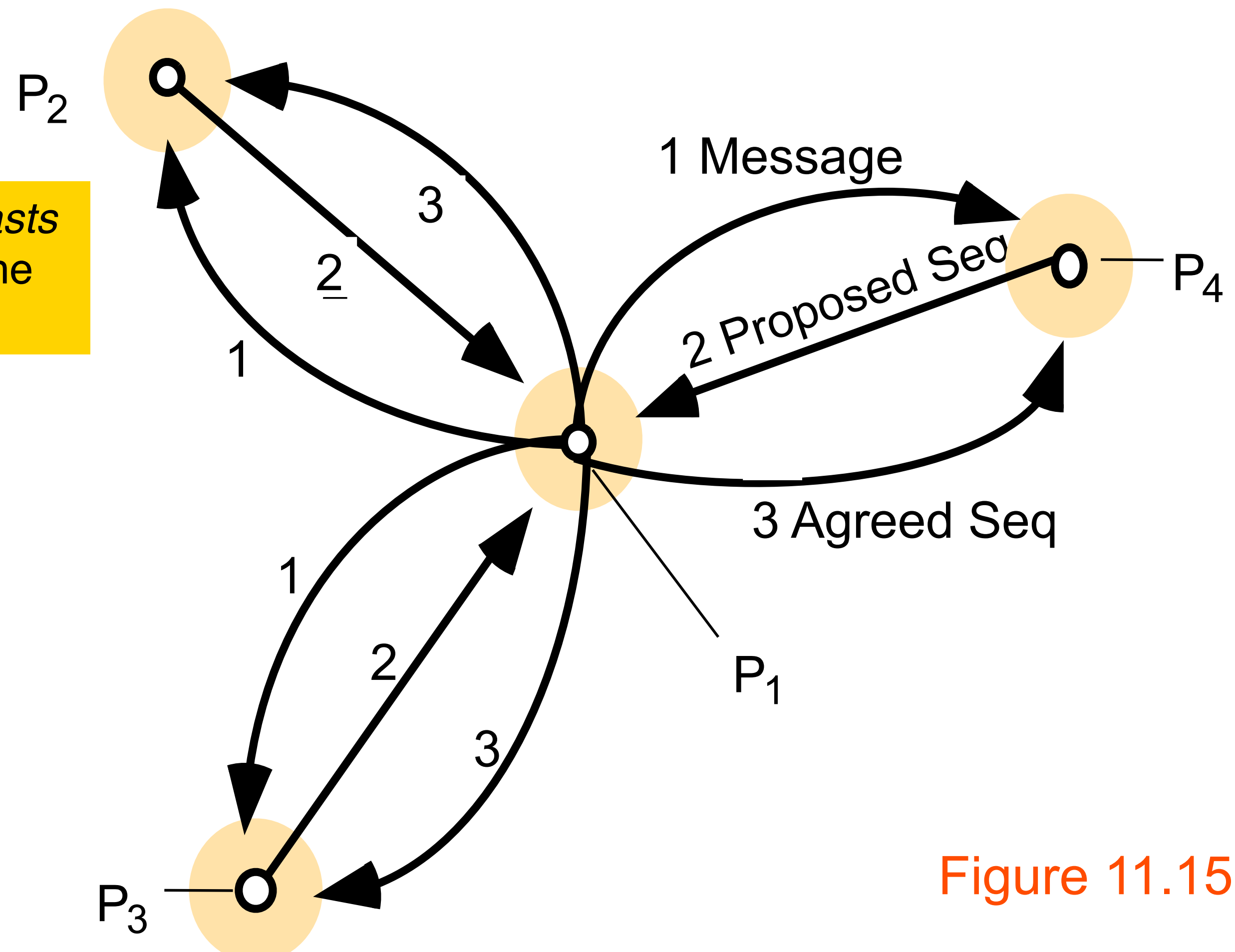


Figure 11.15

The ISIS algorithm for total ordering

1. the process *P1 B-multicasts* a message to members of the group

2. the receiving processes propose numbers and return them to the sender

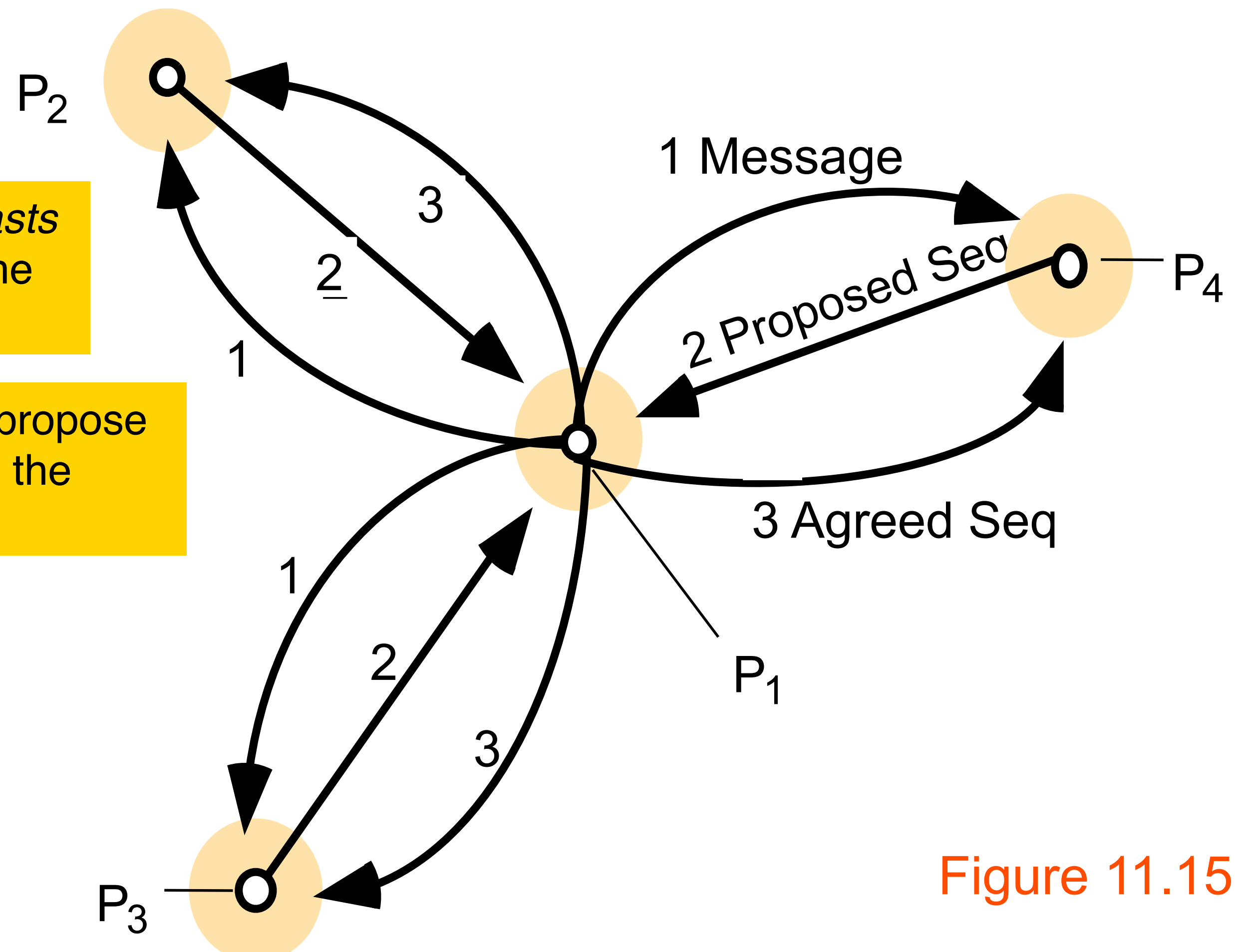


Figure 11.15

The ISIS algorithm for total ordering

1. the process *P1 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

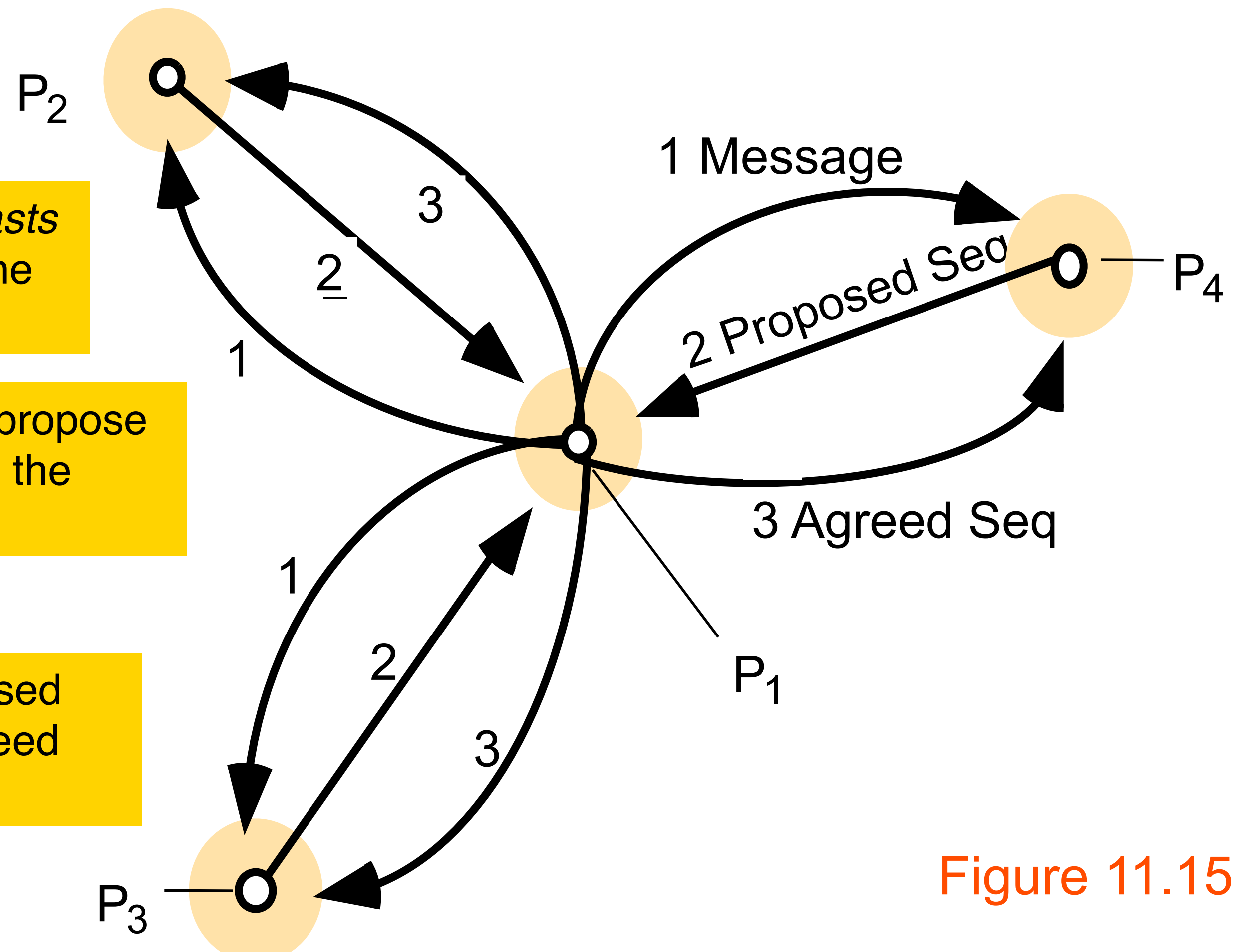


Figure 11.15

The ISIS algorithm for total ordering

- this protocol is for open or closed groups

1. the process *P1 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

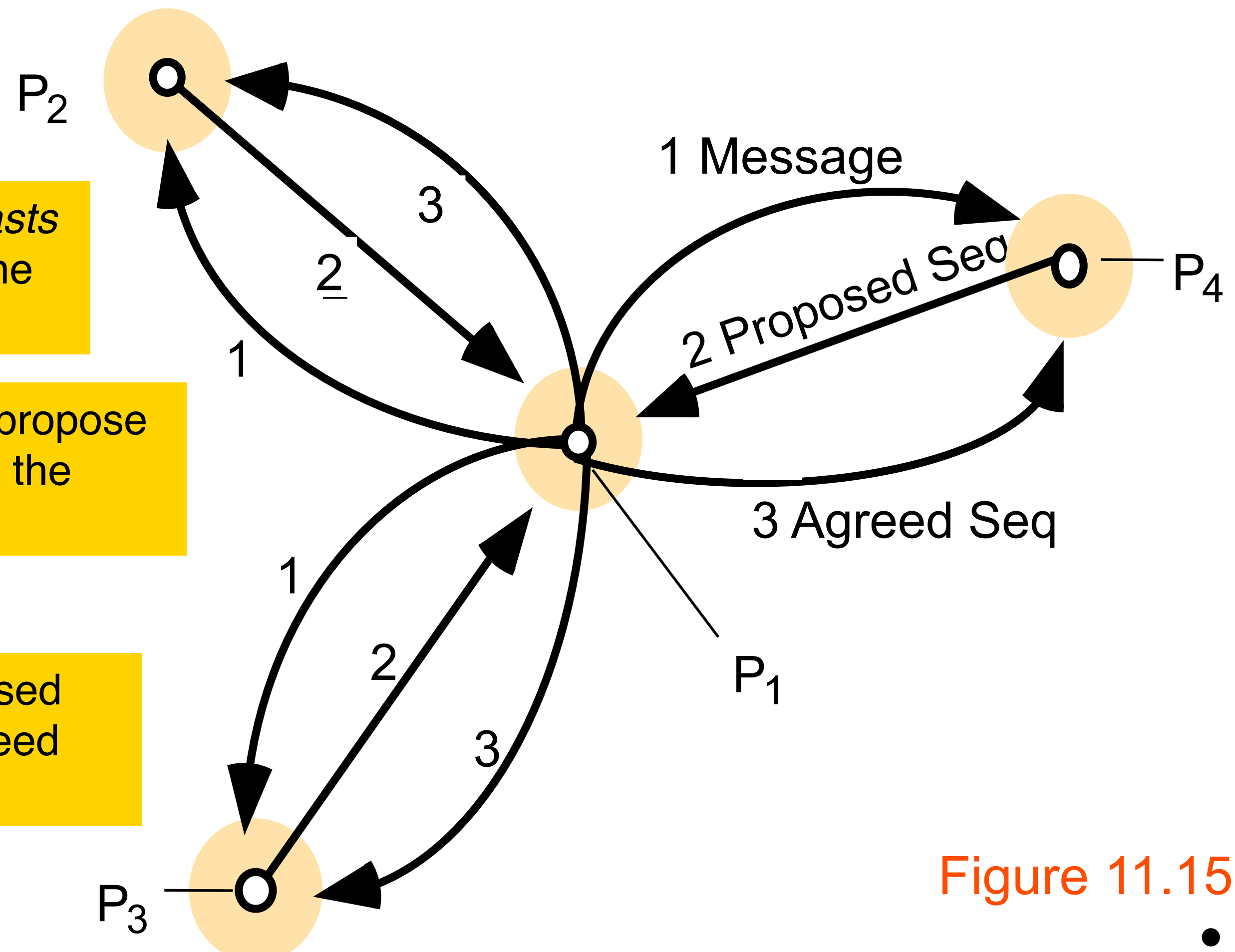


Figure 11.15

ISIS total ordering - agreement of sequence

ISIS total ordering - agreement of sequence

- 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

ISIS total ordering - agreement of sequence

- 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*.

ISIS total ordering - agreement of sequence

- 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
  - $P^q_g := \mathrm{Max}(A^q_g, P^q_g) + 1$.
  - assigns the proposed sequence number to the message and places it in its hold-back queue

ISIS total ordering - agreement of sequence

- 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
  - $P^q_g := \mathrm{Max}(A^q_g, P^q_g) + 1$.
  - 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 $A^q_g := \mathrm{Max}(A^q_g, a)$, attach *a* to the message and re-order hold-back queue.

Discussion of ordering in ISIS protocol

Discussion of ordering in ISIS protocol

- Hold-back queue

Discussion of ordering in ISIS protocol

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

Discussion of ordering in ISIS protocol

- Hold-back queue
  - 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

Discussion of ordering in ISIS protocol

- Hold-back queue
- 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

Discussion of ordering in ISIS protocol

- Hold-back queue
- 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.

Discussion of ordering in ISIS protocol

- Hold-back queue
- 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

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

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

Causal ordering using vector timestamps

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

On initialization

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

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 $V_j^g[k] \le V_i^g[k] \ (k \ne j);$

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

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

Figure 11.16

Causal ordering using vector timestamps

each process has its own vector timestamp

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

On initialization

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

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; // after removing it from the hold-back queue

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

Figure 11.16

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

On initialization

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

To CO-multicast message m to group g

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

B-multicast$(g, \langle V_i^g, m \rangle);$

On B-deliver$(\langle V_j^g, m \rangle)$ *from* p_j, *with* $g = group(m)$

place $\langle V_j^g, m \rangle$ in hold-back queue;

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

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

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

Figure 11.16

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

On initialization

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

To CO-multicast message m to group g

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

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

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

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; // after removing it from the hold-back queue

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

Figure 11.16

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

On initialization

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

To CO-multicast message m to gr

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

B-multicast$(g, \langle V_i^g, m \rangle);$

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

On B-deliver$(\langle V_j^g, m \rangle)$ *from*

place $\langle V_j^g, m \rangle$ in hold-bac

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

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

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

Figure 11.16

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

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

then it CO-delivers the message and updates its timestamp

Algorithm for group member p_i

On initialization

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

To CO-multicast message m to gr

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

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

On B-deliver$(<V_j^g, m>)$ *from*

place $<V_j^g, m>$ in hold-bac

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

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

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

Figure 11.16

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

On initialization

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

To CO-multicast message m to gr

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

B-multicast$(g, \langle V_i^g, m \rangle);$

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

On B-deliver$(\langle V_j^g, m \rangle)$ *from*

place $\langle V_j^g, m \rangle$ in hold-bac

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

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

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

Figure

then it CO-delivers the message and updates its timestamp

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

Comments

Comments

- after delivering a message from p_j, process p_i updates its vector timestamp
  - by adding 1 to the jth element of its timestamp

Comments

- after delivering a message from p_j, process p_i updates its vector timestamp
  - by adding 1 to the jth element of its timestamp
- compare the vector clock rule where $V_i[j] := \max(V_i[j], t[j])$ for j=1, 2, ...N
  - in this algorithm we know that only the jth element will increase

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

Comments

- after delivering a message from p_j, process p_i updates its vector timestamp
  - by adding 1 to the jth element of its timestamp
- compare the vector clock rule where $V_i[j] := \max(V_i[j], t[j])$ for j=1, 2, ...N
  - in this algorithm we know that only the jth 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.

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, ...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