Distributed Systems (5DV147)

Group Communication

Fall 2014

Point-to-point communication

- Participants need to exist at the same time
  - Establish communication
- Participants need to know address of each other and identities
- Not a good way to communicate with several participants

Indirect communication

Indirect communication

- Communication through an intermediary
  - No direct coupling between the sender and the receiver(s)
- Space uncoupling – no need to know identity of receiver(s) and vice versa
  - Participants can be replaced, updated, replicated, or migrated
- Time uncoupling – independent lifetimes
  - Requires persistence in the communication channel

Good for …

- Scenarios where users connect and disconnect *very* often
  - Mobile environments, messaging services, forums
- Event dissemination where receivers may be unknown and change often
  - RSS, events feeds in financial services
- Scenarios with very large number of participants
  - Google Ads system, Spotify
- Commonly used in cases when change is anticipated
  - need to provide dependable services

… but there are also some disadvantages

- ❑ Performance overhead introduced by adding a level of indirection
  - ➢ Reliable message delivery, ordering → (-) effect on scalability
- ❑ More difficult to manage because lack of direct coupling
- ❑ Difficult to achieve end-to-end properties
  - ➢ Real time behavior
  - ➢ Security

Commonalities

- Some processes write information into an abstraction and some other reads from that abstraction

Communication-based: { a queue, a group, a channel } — Potential to scale to very large systems
- ✓ Key is routing infrastructure

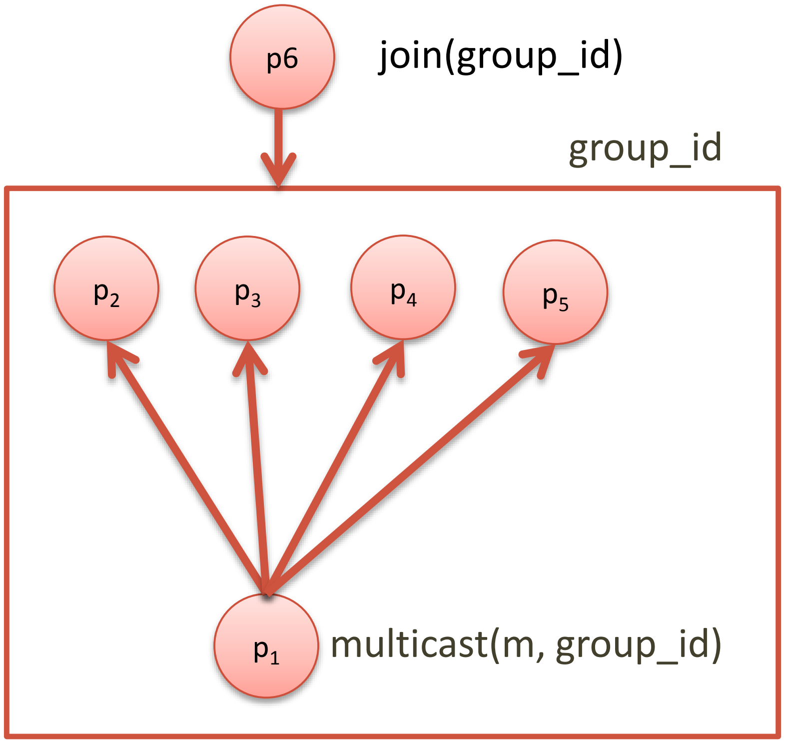
State-based: { an array of memory, a space (whiteboard) } — Need to maintain consistent view of shared state

Group Communication

Characteristics

- Indirect communication
  - Communication through an intermediary
  - No direct coupling between the sender and the receiver(s)
- Group communication
  - Messages sent to a group of processes and delivered to all members of the group

Groups (of processes)



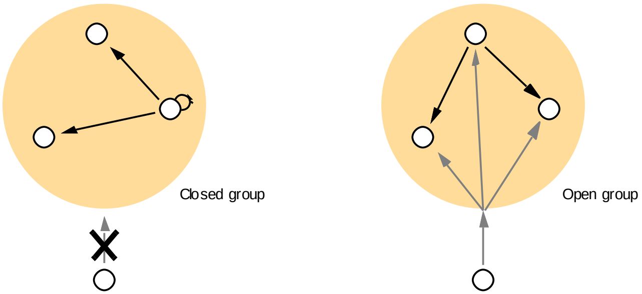
- One-to-many communication
  - Provide reliability and ordering guarantees
- Group management functionality
  - Maintain membership
  - Detect failure of member(s)

Types of groups

Closed or open

A group is *closed* if only members of the group can multicast to it

A group is *open* if processes outside the group may send to it



Overlapping or non-overlapping

In *overlapping* groups, processes may be members of multiple groups

In *non-overlapping* groups, processes may belong to at most one group

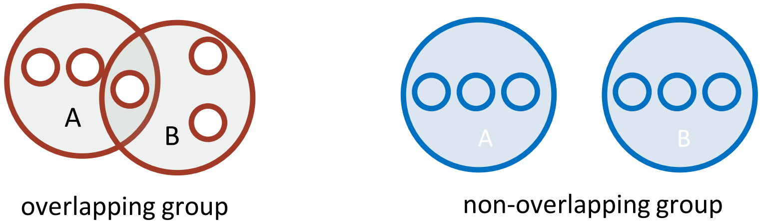


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 6.2**

Group membership management

- Interface for group membership changes
  - Create and destroy groups, add or remove members to a group
- Failure detection
  - Mark processes as suspected or unsuspected and remove those processes that have (suspected) failed
  - Notify members of group membership changes
  - Processes that join or leave
  - Perform group address expansion
  - From group id to individual group members (current)

Multicast

Receive versus Deliver

Receive: message has arrived and will be processed

Deliver: message is allowed to reach upper layer

Unreliable (basic) multicast (using reliable unicast)

- ❑ Send (unicast) to each other process in the group!
- ❑ What if sender fails halfway through?

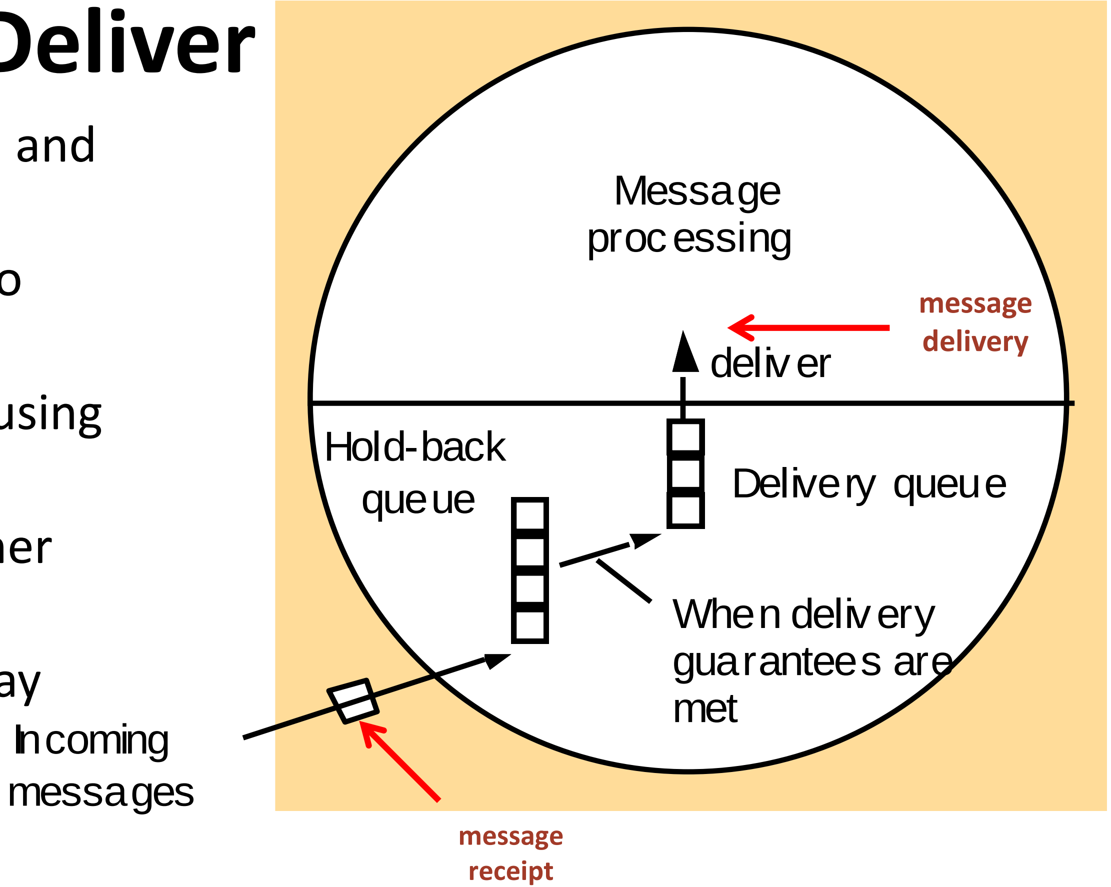


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 15.10**

Basic Multicast

- Use a reliable one-to-one send operation

  B-multicast (g, m):

  for each process p ϵ g, send (p, m)

  receive(m) at p:

  B-deliver(m) at p

Reliable multicast

- Integrity
  - Messages delivered at most once
- Validity
  - If a correct process multicasts message m, it will eventually deliver m
- Agreement
  - If a correct process delivers m, then all correct processes in the group will eventually deliver m

Reliable multicast algorithm

- ❑Use basic multicast to send to all (including self)
- ❑When basic multicast delivers, check if message has been received before
  1. If it has, do nothing further
  2. If not, and sender is not own process

     Basic multicast message to others
  3. Deliver message to upper layer

Integrity? Validity? Agreement? Yes!

Insane amounts of traffic? Yes! Every message is sent `sizeof(group)` to each process!

- A single message will be sent 100 times if we just have 10 processes

Message Orderings

Message orderings

1. Unordered
2. FIFO
3. Total
4. Causal
5. Hybrid orderings such as Total-Causal & Total-FIFO

FIFO ordering

FIFO ordering

- Intuition

  Messages from a process should be delivered in the order in which they were sent

- Solution

  Sender numbers the messages, receivers hold back those that have been received out of order

Process P1

$S(p_1,g)$ → # of messages that p has sent to the group

$R(p_2,g)$ → sequence # of latest message that p_1 has delivered from p_2 that was sent to g

$R(p_3,g)$

n members of g

$R(p_n,g)$

FO-multicast

Send $S(p_i,g)$ with message

B-multicast message

Increment by $S(p_i,g)$ 1

FO-deliver

If $S=R(p_j, g) + 1$

FO-deliver and set $R(p_j, g) = S$

If $S>R(p_j, g) + 1$

Place in hold-back queue until

$S = R(p_j, g) + 1$

Total ordering

Total ordering

❑Intuition

Messages from all processes should get a (unique) group wide ordering number, so all processes can deliver messages in a single order!

> Mental pitfall: the order itself does not have to make any sense, as long as all processes abide by it!

Implementing total ordering

- Sequencer
  - Simple
  - Central server (= single point of failure)
- ISIS-algorithm
  - Not as simple
  - Distributed
  - Study on your own!

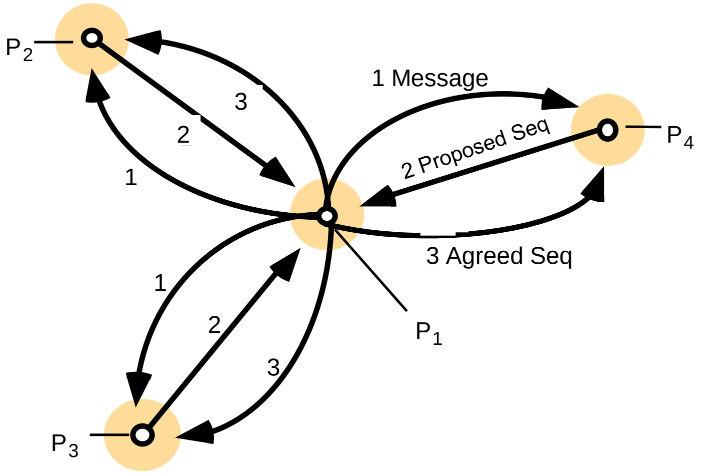


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 15.14**

Sequencer

- Sequencer is logically external to the group
- Messages are sent to all members, including sequencer
  - Initially, have no "ordering" number
- Sequencer maps message identifiers to ordering numbers
  - Multicasts mapping to group
  - Once a message has an ordering number, it can be delivered according to that number

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g

1 *B-multicast*($g \cup \{sequencer(g)\}$, <*m*, *i*>); Send the to g and sequencer

On B-deliver(<*m*, *i*>) *with* $g = group(m)$ Wait until right time to deliver (given by sequence # from sequencer)

2 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 $S = r_g$;

4 *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* $g = group(m)$

3 *B-multicast*(*g*, <"order", *i*, s_g>); multicast sequential # to g

$s_g := s_g + 1$; sequence # is totally ordered

Sequencer – final notes

- Note, again, that the ordering is completely up to the sequencer
  - It could collect all messages for half an hour and then assign numbers according to how many a’s there are in the message
  - While annoying to use, this is still a total order, and all processes will have to follow it!

Causal ordering

Causal ordering

❑Intuition

Captures causal (cause and effect) relationships via happened-before ordering

Vector clocks ensure that replies are delivered after the message that they are replying to

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

On initialization

$V_i^g[j] := 0$ $(j = 1, 2..., N)$; ← zeroes the vector clock

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$; ← Increases own clock

$B\text{-}multicast(g, <V_i^g, m>)$; ← Multicast message and vector clock

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

$V_i^g[j] := V_i^g[j] + 1$; ← Increases vector clock for p_j

delivers previously messages from p_j → (annotating $V_j^g[j] = V_i^g[j] + 1$)

It has delivered any message that p_j delivered at that time that it multicasted this message → (annotating $V_j^g[k] \le V_i^g[k]$ $(k \ne j)$)

Hybrid orderings

Hybrid orderings

- Causal order is not unique
  - Concurrent messages

    ...neither is FIFO
  - FIFO only guarantees per process not inter-process
- Total order only guarantees a unique order
  - Combine with others to get stronger delivery semantics!

Summary

- Group communication
  - One-to-many, indirect communication
- Different types of groups
  - Open, closed, overlapping, and non-overlapping
- Reliability in group communication
  - Integrity, validity, and agreement
- Group membership management
  - Changes, failure detection, notification of membership changes, group address expansion

Summary

- ❑ Multicast, reliable and unreliable
- ❑ Message ordering
  - ➢ The ordering in delivering messages is necessary in some cases
  - ➢ Ordering is expensive in terms of delivery latency and bandwidth consumption
  - ➢ FIFO – order messages from each sender
  - ➢ Causal – order messages across senders
  - ➢ Total – same message ordering on all recipients

Next Lecture

Consensus