Broadcast and Multicast

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo

Multi-Process Communication

Distributed systems often send messages between multiple processes.

It can be convenient to model that as a single send with multiple recipients.

Modern communication networks do not always allow this.

Protocols above the transport layer can simulate this.

Unicast

Traditional network communications are unicast:
one sender, one receiver.

TCP in particular is always unicast.

Unicast is the only reliably available method on the Internet.

This is a combination of several factors:

- The minimal requirements for IP's network of networks
- Stateless routing of individual packets
- Unicast's natural proof-of-stake limiting abuse potential

Broadcast

Broadcast sends messages to all recipients on a network.

This is typically only available on local area networks.

The abuse and misconfiguration potential for broadcast is very high!

(Imagine sending a message to every computer on the Internet!)

Not all underlying networks are capable of broadcast.

Multicast

Then there is something in between: group multicast.

Multicast sends a single message to more than one recipient.

Like broadcast, this works on local networks.

There is also Internet routed multicast!
(But you probably can't use it.)

Multicast makes a lot of sense for distributed systems:

- Not every system on the network is part of the system
- Maybe more than one of them is

Emulating Broadcast or Multicast

Broadcast and multicast can be emulated with unicast.

A process makes multiple unicast connections, one to each recipient.

It sends a unicast message on each connection.

This is more work for the sender.

It does not require specific assistance from the network!

Multicast as a Service

Assume that the hosts $g_0, \ldots, g_n$ are in a group G.

Hosts from G wish to send messages to all other hosts in G.

We will model group multicast as a service.

It has three operations:

- MSend(m, G): Send the message m to every host in G
- MRecv(m): Receive the message m from the network
- Deliver(m): Asynchronously deliver m to the application

Note that MRecv and Deliver are different steps!

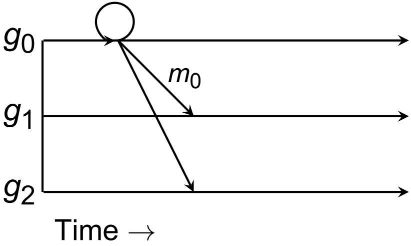
Simple Multicast from Unicast

A host can emulate multicast using unicast as follows:

MSend(m, G):
for each $g_i \in G$:
Send(m, g_i)

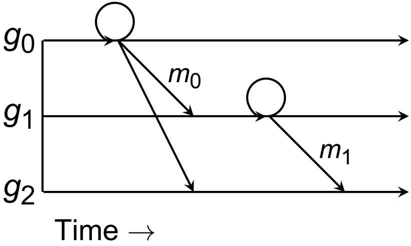
MRecv(m):
Deliver(m)

Sending a Message



g_0 issues MSend(m_0, G)

Sending a Message



g_1 issues MSend(m_1, G) but fails before sending to g_0

Simple Multicast Properties

What are the properties of this multicast?

Cost:

- $|G|$ messages to send to $|G|$ hosts

Reliability:

- If the sender does not fail, the messages are delivered

How does this seem?

Simple Multicast Properties

What are the properties of this multicast?

Cost:

- $|G|$ messages to send to $|G|$ hosts

Reliability:

- If the sender does not fail, the messages are delivered

How does this seem?

Cost: great! Reliability: could be better…

We will use MSend and MRecv to build more reliable protocols.

Building Reliable Multicast

We can build a reliable multicast using MSend and Deliver.

This multicast ensures that if any process receives a message, all processes receive the message.

We define two new primitives in terms of simple multicast:

R_MCast(*m*, *G*):
  MSend(*m*, *G*)

R_MRecv(*m*):
  if *m* has not previously been received:
    MSend(*m*, *G*)
  Deliver(*m*)

Reliable Multicast Requirements

The receive rule requires message identification.

Unique identifiers can solve this:

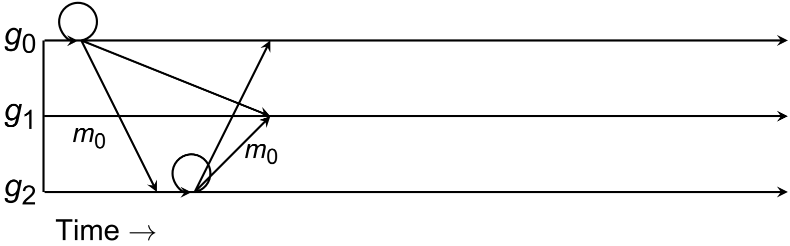
- Globally unique (such as content addressing)
- Locally unique (each process $g_i \in G$ keeps a counter)

This each process delivers a message at most once.

Sending a Message

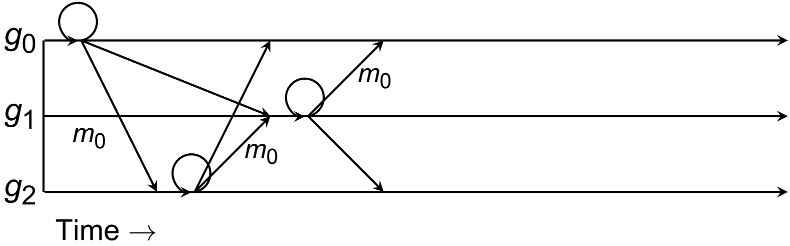
g_0 issues R_MCast(m_0, G)

Sending a Message



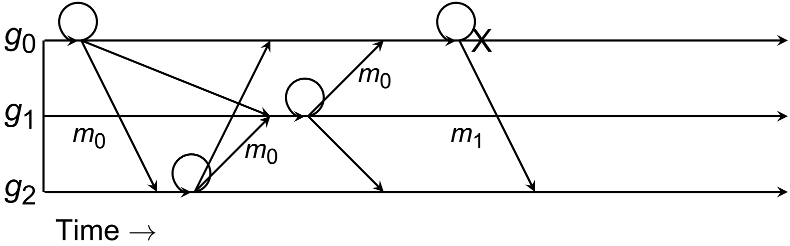
g_2 processes R_MRecv(m_0) and issues R_MCast(m_0, G)

Sending a Message



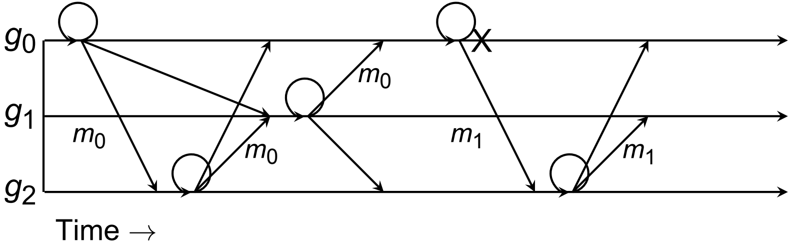
g_1 processes R_MRecv(m_0) and issues R_MCast(m_0, G)

Sending a Message



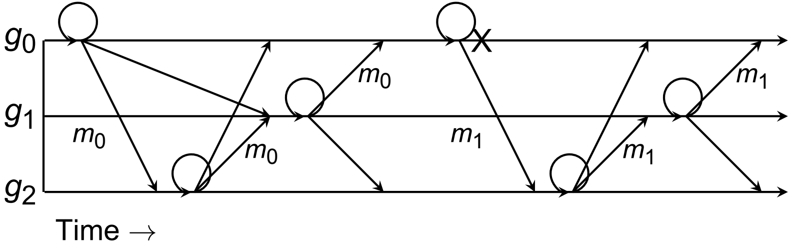
g_0 issues R_MCast(m_1, G) but fails before sending to g_1

Sending a Message



g_2 processes R_MRecv(m_1) and issues R_MCast(m_1, G)

Sending a Message



g_1 processes R_MRecv(m_1) and issues R_MCast(m_1, G)

Reliable Multicast Properties

Cost:

- Much more expensive than simple multicast!
- $O(|G|^2)$ messages are sent.

Reliability:

- If any receiver does not fail, all non-failed receivers receive the message.

If the cost can be borne, the benefit is agreement on what was received.

Ordering Messages

There is value to ordering multicast messages.

There are three meaningful orderings for message delivery:

- First-in-First-out (FIFO)
- Causal
- Total

FIFO Ordering

FIFO ordering preserves the message order from each process.

Messages from different processes may be reordered, however.

Formally:
If a process p sends m followed by m', then every correct process that delivers m' must have already delivered m.

Remember that Chandy-Lamport snapshots required FIFO delivery.

Causal Ordering

Causal ordering preserves the causal relationship between messages.

This is like Lamport clocks [4] or vector clocks [5].

Formally:
If $\text{MSend}(m, G) \to \text{MSend}(m', G)$, then every correct process that delivers m' must have already delivered m.

Causal ordering implies FIFO ordering.

Total Ordering

Total ordering preserves the order of all messages across all processes.

Formally:
If any correct process delivers m before m', then every correct process that delivers m' must have already delivered m.

Total ordering does not imply causal ordering!

Observations on FIFO Ordering

Note that TCP connections preserve FIFO ordering.

This is only true for messages sent on the same connection.

Multicast messaging often uses UDP.

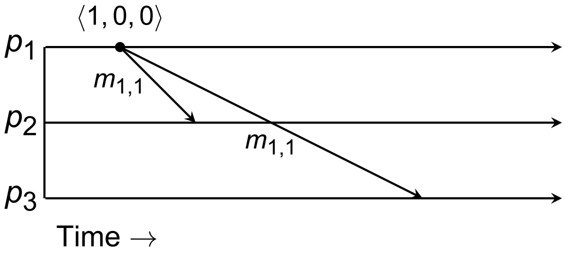
This is because TCP handshakes are expensive.

Similar techniques to reliable multicast or TCP sequence numbers can be used.

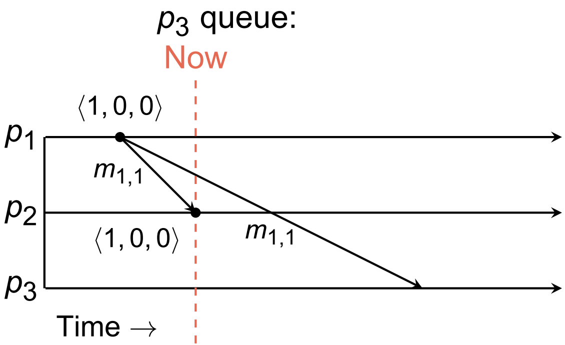
Note that this requires a receive buffer and queuing!

FIFO Sequence

p_3 queue:



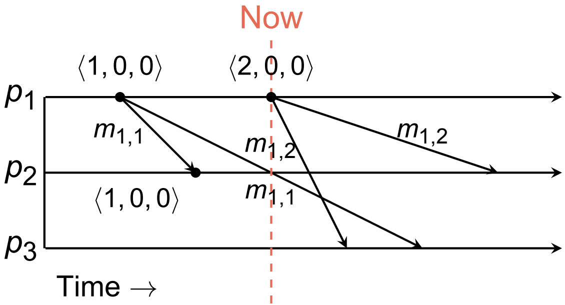
FIFO Sequence



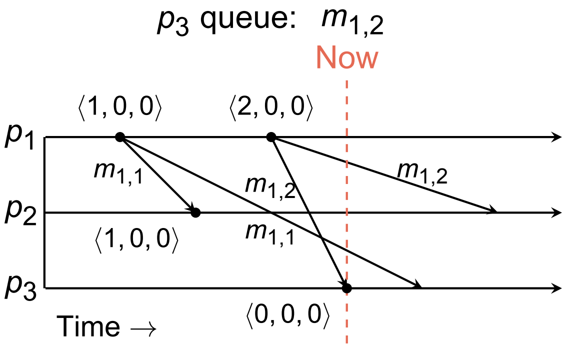
Process p_2 can immediately deliver $m_{1,1}$ as it is in-order from p_1.

FIFO Sequence

p_3 queue:

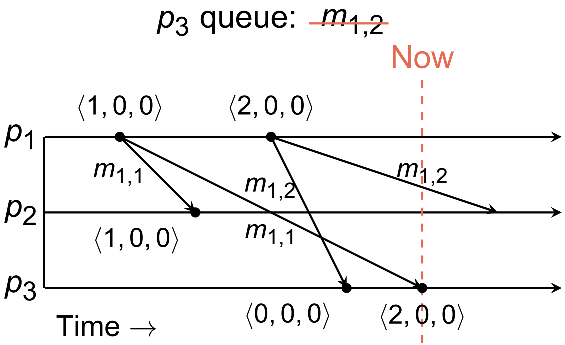


FIFO Sequence



Process p_3 cannot deliver $m_{1,2}$ as it is waiting for $m_{1,1}$!
It must queue $m_{1,2}$.

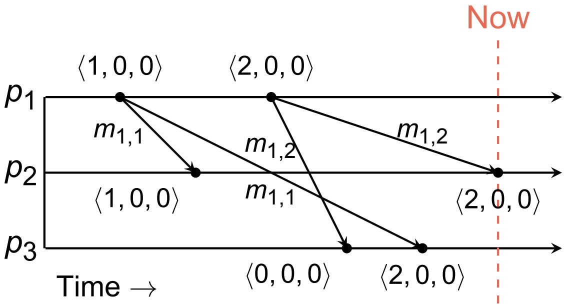
FIFO Sequence



Process p_3 can immediately deliver $m_{1,1}$.
Process p_3 can deliver the queued $m_{1,2}$.

FIFO Sequence

p_3 queue:



Process p_2 can immediately deliver $m_{1,2}$.

Summary

- Distributed systems benefit from group communication
- Internet communication is mostly unicast
- Broadcast and multicast can be built from unicast
- Relatively simple protocols can achieve all-or-nothing delivery
- FIFO delivery requires only a TCP-like sequence number

Next Time …

- Causal and total ordered multicast

References I

Required Readings

[3] Ajay D. Kshemkalyani and Mukesh Singhal. Distributed Computing: Principles, Algorithms, and Systems. Chapter 6: Intro, 6.1 through 6.1.3, 6.4. Cambridge University Press, 2008. ISBN: 978-0-521-18984-2.

Optional Readings

[1] Kenneth P. Birman and Thomas A. Joseph. “Reliable Communication in the Presence of Failures”. In: 5.1 (Feb. 1987), pp. 47–76. DOI: 10.1145/7351.7478. URL: https://search.lib.buffalo.edu/permalink/01SUNY_BUF/12pkqkt/cdi_gale_infotracacademiconefile_A6048598.

References II

[2] Ajay D. Kshemkalyani and Mukesh Singhal. Distributed Computing: Principles, Algorithms, and Systems. Chapter 6: 6.5. Cambridge University Press, 2008. ISBN: 978-0-521-18984-2.

[4] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: 21.7 (July 1978). Ed. by R. Stockton Gaines, pp. 558–565. URL: https://dl-acm-org.gate.lib.buffalo.edu/doi/pdf/10.1145/359545.359563.

References III

[5] Friedemann Mattern. “Virtual Time and Global States of Distributed Systems”. In: Proceedings of the Workshop on Parallel and Distributed Algorithms. Elsevier Science Publishers B.V., Oct. 1988, pp. 215–226. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1068.1331.

To retrieve a copy of this material, or related materials, see `https://www.cse.buffalo.edu/~eblanton/`.