DS Lecture 5
Multicast

October 5, 2023

Michele Albano
mialb@cs.aau.dk

DEIS
Aalborg University
Denmark

AALBORG UNIVERSITY
DENMARK

I was gonna tell you guys a joke about UDP, but you might not get it.

- The TCP Stack

AALBORG UNIVERSITY
DENMARK

What is it?

What is multicast?

A **multicast** is **one-to-many** communication between a single process and a **specific group** of processes such that **all members of the group receive the message** .

What is it?

Examples

- Algorithms with failover/replication/redundancy
  - DNS
  - Databases
  - Caches
  - Banks!
- One-to-many
  - Streaming of TV/Radio
  - Industrial Systems
- Many-to-many
  - Skype
  - Teams
  - . . .

What is it?

Big Question:

How do we guarantee that everyone gets the same information?

What is it?

Big Question:

How do we guarantee that everyone gets the same information? And what do we mean by **same** ?

Agenda

- IP Multicast
  - Hardware Support
  - Problems
- Requirements
- Basic Multicast
- Reliable Multicast
- Ordered Multicast
- FIFO Multicast
- Totally Ordered Multicast
- Causally Ordered Multicast
  - Vector Clocks

Disclaimer

- We assume closed groups, and
- We assume static groups,
- Not discussing multiple groups.

Good news : All algorithms shown work both in sync and async networks

IP Multicast

- Use Internet Group Management Protocol (IGMP)
- Get IP of group
  - IPv4: 224.0.0.0 - 239.255.255.255 (-224.0.0.255 for permanent)
  - IPv6: FF00::/8
- Build on UDP over IP
- Careful of firewalls/NAT

Hardware Support

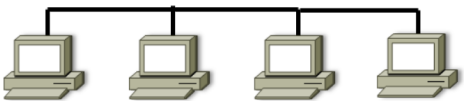


Figure: No Hardware Support

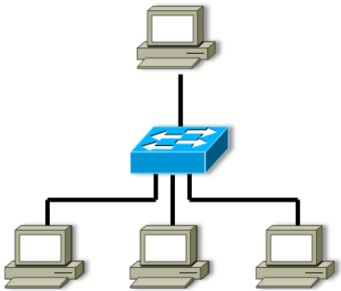
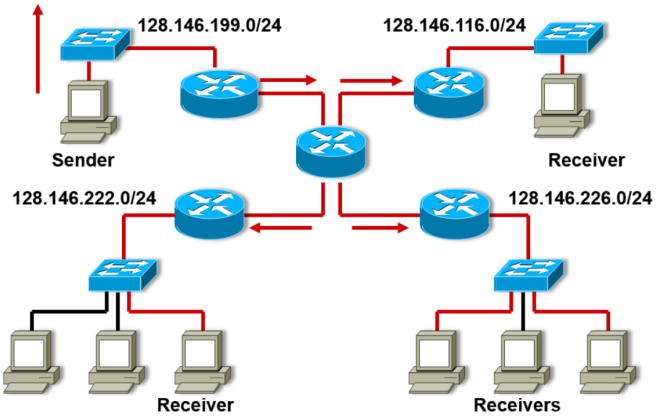


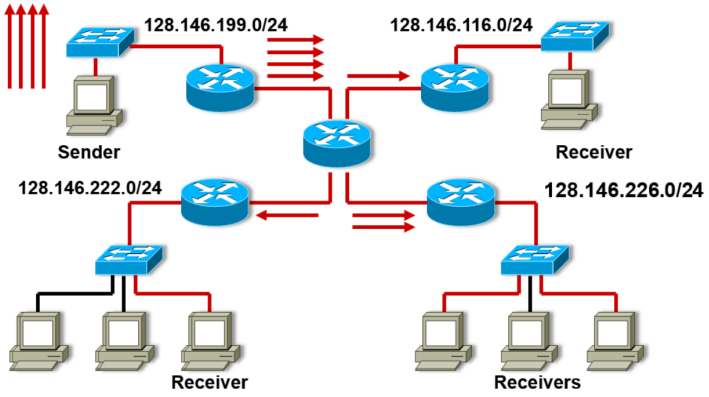
Figure: Hardware Support

Wide Area Network
Hardware Support



Wide Area Network

No Hardware Support

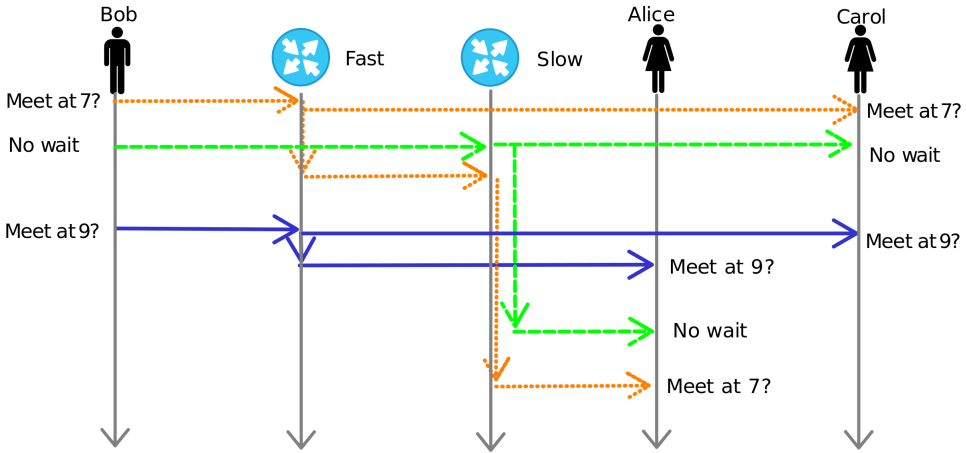


Problems
IP Multicast

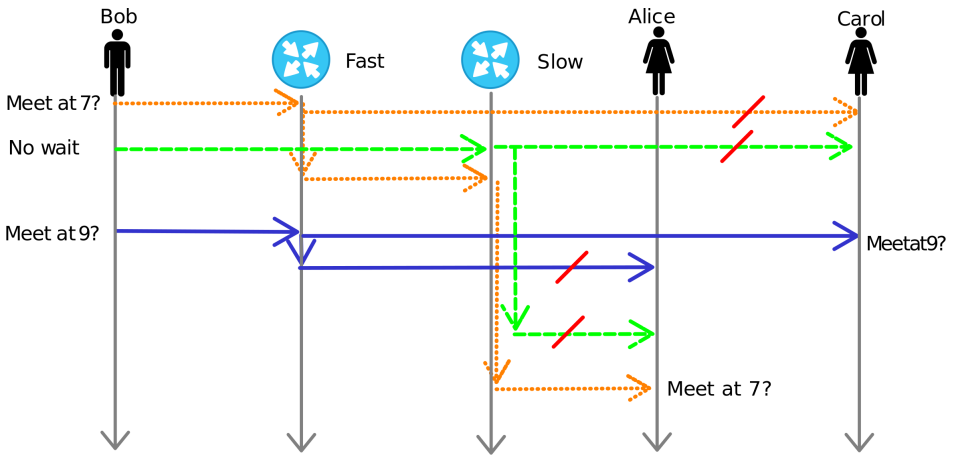
UDP has no guarantees

- No re-transmission
  - no reception garuanteed
  - one attempt only
- No ordering
  - Messages are delivered in arbitrary order

Problems
IP Multicast



Problems
IP Multicast



Requirements
Multicast

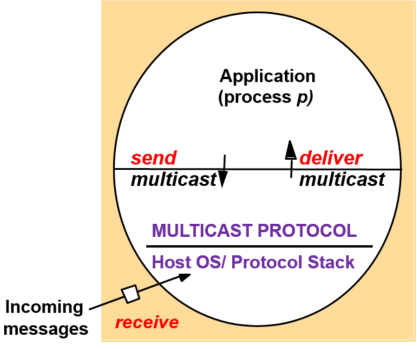
Assuming

- Reliable 1:1 communication
- Sender might crash
- No order

Guarantees

- if a message is sent, it is **delivered** exactly once
- messages are eventually **delivered** to non-crashed (correct) processes

Delivery
Multicast



Delivery
Multicast


```
1
2  # Implemented by a multicast user
3  class MulticastListener:
4
5      # Send message to the application layer
6      def deliver(self, content):
7          raise NotImplementedError
8
9      # Not needed strictly, but we use it to handle non-multicast messages
10     def forward(self, message):
11         raise NotImplementedError
12
13 # Implemented by a multicasting protocol
14 class MulticastService:
15     # Called when the application wants to do a multicast
16     def send(self, content):
17         raise NotImplementedError
```

Basic Multicast


```
1  class BasicMulticast(Device, MulticastService):
2
3      def __init__(self, index: int, number_of_devices: int, medium: Medium, application: MulticastListener):
4          super().__init__(index, number_of_devices, medium)
5          self._application = application
6          self._outbox = [] # needed for technical reasons w. framework
7
8      def run(self):
9          while True:
10             for ingoing in self.medium().receive_all():
11                 self.handle_ingoing(ingoing)
12             while len(self._outbox) > 0:
13                 msg = self._outbox.pop(0)
14                 self.send_to_all(msg)
15             self.medium().wait_for_next_round()
16
17     def handle_ingoing(self, ingoing: MessageStub):
18         if isinstance(ingoing, MulticastMessage):
19             self._application.deliver(ingoing.content())
20         else: self._application.forward(ingoing)
21
22     def send_to_all(self, content):
23         for id in self.medium().ids():
24             # we purposely send to ourselves also!
25             message = MulticastMessage(self.index(), id, content)
26             self.medium().send(message)
27
28     def send(self, message):
29         # would normally send directly (and blocking) to the network
30         self._outbox.append(copy.deepcopy(message))
```

Basic Multicast

- Sender can fail
- Reliable send = ACK implosion

Reliable Multicast - Properties

- Integrity
  - Messages are delivered at most once
- Validity
  - A process delivers to itself (or crashes)
- Agreement
  - All correct processes deliver, or no correct process deliver

Reliable Multicast


```
1  class ReliableMulticast(MulticastListener, MulticastService, Device):
2
3      def __init__(...):
4          super().__init__(index, number_of_devices, medium)
5          self._application = application
6          self._b_multicast = BasicMulticast(index, number_of_devices, medium, self)
7          self._seq_number = 0  # not strictly needed, but helps giving messages a unique ID
8          self._received = set()
9
10     def send(self, content):
11         self._b_multicast.send((self.index(), self._seq_number, content))
12         self._seq_number += 1
13
14     def deliver(self, message):
15         (origin_index, seq_number, content) = message
16         if message not in self._received and origin_index is not self.index():
17             self._b_multicast.send(message)
18             self._received.add(message)
19             self._application.deliver(content)
20
21     def run(self):
22         self._b_multicast.run()
23
24     def forward(self, message):
25         self._application.forward(message)
```

Reliable Multicast

- Integrity
- Validity
- Agreement

Reliable Multicast

- Integrity
  - Yes
- Validity

- Agreement

Reliable Multicast

- Integrity
  - Yes
- Validity
  - Yes
- Agreement

Reliable Multicast

- Integrity
  - Yes
- Validity
  - Yes
- Agreement
  - Yes

Reliable Multicast

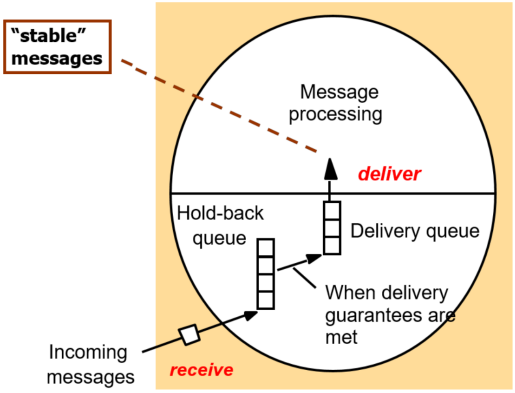
- Integrity
  - Yes
- Validity
  - Yes
- Agreement
  - Yes
- 1 multicast = $O(N^2)$ messages in the network

Reliable Multicast over IP

Fix: Steal ideas from TCP

- Use sequence numbers
  - duplicates
  - lost messages
- use “hold-back”-construction
  - wait for re-transmission
  - replication of messages
- keep track of sequence numbers of others
- “gossip” sequence numbers

Delivery
Reliable Multicast over IP



Reliable Multicast over IP


```
1  class ReliableIPMulticast(MulticastListener,
       MulticastService, Device):
2      def __init__(...):
3          super().__init__(index, number_of_devices, medium)
4          self._application = application
5          self._b_multicast = BasicMulticast(index,
               number_of_devices, medium, self)
6          self._seq_numbers = [0 for _ in medium.ids()]
7          self._received = {}
8
9      def run(self):
10         self._b_multicast.run()
11
12     def deliver(self, message):
13         (origin_index, seq_numbers, content) = message
14         seq_nr = seq_numbers[origin_index]
15         self._received[(origin_index, seq_nr)] = message
16         if self._seq_numbers[origin_index] <= seq_nr:
17             self.try_deliver()
18             self.nack_missing(seq_numbers)
19
20     def send(self, content):
21         o_seq = self._seq_numbers[self.index()]
22         self._received[(self.index(), o_seq] = content
23         self._b_multicast.send((self.index(), self.
               _seq_numbers, content))
24         self.try_deliver()
```

```
23     def forward(self, message):
24         if isinstance(message, NACK):
25             content = self._received[
26                 (self.index(), message.seq_number())]
27             self.medium().send(Resend(self.index(),
28                 message.source,
29                 (self.index(), self._seq_numbers, content)))
30         elif isinstance(message, Resend):
31             self.deliver(message.message())
32         else: self._application.forward(message)
33
34     def try_deliver(self):
35         for (oid, seqnr), content in self._received.items():
36             if self._seq_numbers[oid] == seqnr:
37                 self._application.deliver(content)
38                 self._seq_numbers[oid] += 1
39                 self.try_deliver()  # recursively!
40                 return
41
42     def nack_missing(self, n_seq: list[int]):
43         for id in range(0, len(n_seq)):
44             nid = self._seq_numbers[id] + 1
45             for mid in range(nid, n_seq[id]):
46                 self.medium().send(
47                     NACK(self.index(), id, mid))
```

Reliable Multicast over IP

- Integrity
- Validity
- Agreement

- No drops, good ordering

Reliable Multicast over IP

- Integrity
  - Yes (IP also does checksum!)
- Validity
- Agreement
- No drops, good ordering

Reliable Multicast over IP

- Integrity
  - Yes (IP also does checksum!)
- Validity
  - Yes
- Agreement

- No drops, good ordering

Reliable Multicast over IP

- Integrity
  - Yes (IP also does checksum!)
- Validity
  - Yes
- Agreement
  - . . . not really (Exercise)

- No drops, good ordering

Reliable Multicast over IP

- Integrity
  - Yes (IP also does checksum!)
- Validity
  - Yes
- Agreement
  - . . . not really (Exercise)
- Two problems, which? (Exercise)
  - One theoretical
  - One practical
- No drops, good ordering

Reliable Multicast over IP

- Integrity
  - Yes (IP also does checksum!)
- Validity
  - Yes
- Agreement
  - . . . not really (Exercise)
- Two problems, which? (Exercise)
  - One theoretical
  - One practical
- No drops, good ordering $= O(N)$ messages!

Reliable Multicast over IP

- Integrity
  - Yes (IP also does checksum!)
- Validity
  - Yes
- Agreement
  - . . . not really (Exercise)
- Two problems, which? (Exercise)
  - One theoretical
  - One practical
- No drops, good ordering $= O(N)$ messages!
  - . . . how many NACK's can we send for one message?
  - . . . when to send NACK's?
  - . . . lot of small NACK/Resend?

Ordered Multicast

- FIFO Ordered
  - Messages from p_n are received at p_k in order send by p_n
- Total Ordered
  - All messages are recieved in same order at p_n and p_k
- Causally Ordered
  - If p_n recives m_1 before m_2, then m_1 **happened-before** m_2

Ordered Multicast

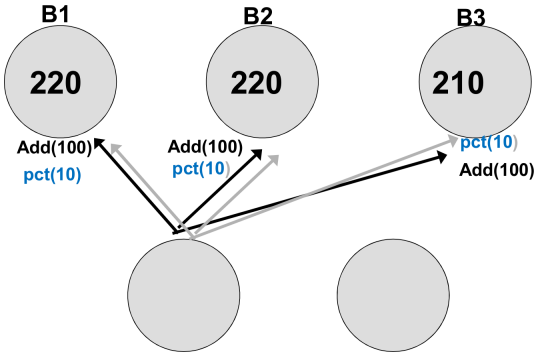


Figure: Unordered Multicast

Ordered Multicast

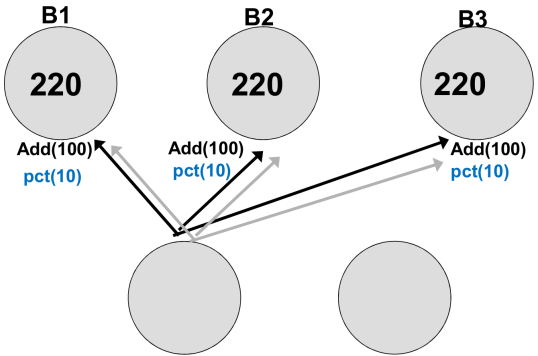


Figure: FIFO Multicast

Ordered Multicast

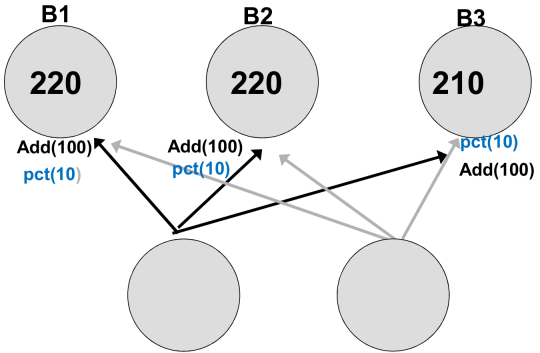


Figure: FIFO Multicast - Fails too

Ordered Multicast

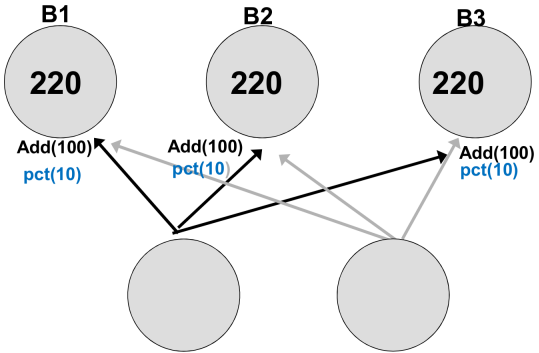


Figure: Total Order - good enough?

Ordered Multicast

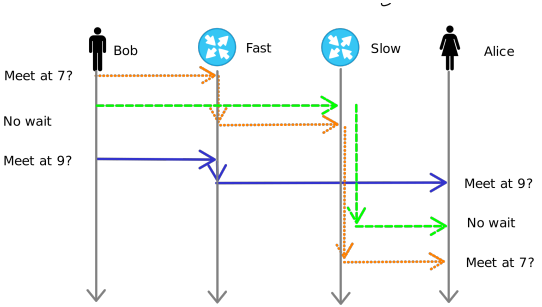


Figure: Alice gets the messages in a total order!

FIFO Multicast

Reliable IP-Multicast is FIFO

Why?

FIFO Multicast

Reliable IP-Multicast is FIFO

Why?

We respect sequence-numbers of sender!

Totally Ordered Multicast

Idea:

- do as FIFO, but only one sequence-number,
- each message has a unique id/hash,
- agree globally on “next”-message:
  1. use global sequencer, or
  2. use negotiation (ISIS)

Totally Ordered Multicast (Sequencer)


```
1   def __init__(...):
2       super().__init__(index, number_of_devices, medium)
3       self._application = application
4       self._b_multicast = BasicMulticast(index,
                number_of_devices, medium, self)
5       self._l_seq = 0
6       self._g_seq = 0
7       self._order = {}
8       self._received = {}
9
10  def send(self, content):
11      self._b_multicast.send((self.index(), self._l_seq,
            content))
12      self._l_seq += 1
13
14  def try_deliver(self):
15      for mid, order in self._order.items():
16          if order == self._g_seq and
                                mid in self._received:
17              self._g_seq += 1
18              self._application.
                      deliver(self._received[mid])
19              self.try_deliver()
20              return
21
22  def run(self):
23      self._b_multicast.run()
24
25
```

```
26  def forward(self, message):
27      self._application.forward(message)
28
29  def deliver(self, message):
30      if not isinstance(message, Order):
31          (sid, sseq, content) = message
32          mid = (sid, sseq)
33          if self.index() == 0:
34              # index 0 is global sequencer
35              self._order[mid] = self._g_seq
36              self._b_multicast.send(
37                                  Order(mid, self._g_seq))
38              self._application.deliver(message)
39              self._g_seq += 1
40          else:
41              self._received[mid] = content
42              self.try_deliver()
43      elif self.index() != 0:
44          # index 0 is global sequencer
45          self._order[message.message_id()] = message.order()
46          self.try_deliver()
```

Totally Ordered Multicast (Sequencer)

Problems

- Sequencer is bottle-neck
- Single point of failure

Bonus

- What breaks w. IP-multicast instead of B-multicast?

Totally Ordered Multicast (Sequencer)

Problems

- Sequencer is bottle-neck
- Single point of failure

Bonus

- What breaks w. IP-multicast instead of B-multicast?
- Packet-loss = deadlock of process!
- Solution: reliable underlying multicast

Totally Ordered Multicast (ISIS)

Idea: Negotiate next ID

1. Process *p* broadcasts message *m*
2. Every other process *q* responds to *p* with a proposal
3. *p* picks largest proposed value, broadcasts

The Trick

Track “largest proposed value” and “largest agreed value” at each process

Totally Ordered Multicast (ISIS)


```
1  def __init__(...):
2      super().__init__(...)
3      self._application = application
4      self._b_multicast = BasicMulticast(index,
5              number_of_devices, medium, self)
6      self._l_seq = 0  # local sequence
7      self._g_seq = 0  # global sequence
8      self._a_seq = -1  # last agreed
9      self._p_seq = -1  # last proposed
10     self._order = {}  # order of messages
11     self._votes = {}  # votes of messages
12     self._hb_q = {}  # holdback of messages
13
14 def run(self):
15     self._b_multicast.run()
16
17 def send(self, content):
18     self._b_multicast.send(
19                 (self.index(), self._l_seq, content))
20     self._votes[(self.index(), self._l_seq)] = []
21     self._l_seq += 1
22
23 def try_deliver(self):
24     for mid, content in self._hb_q.items():
25         if mid in self._order:
26             if self._order[mid] == self._g_seq:
27                 self._g_seq += 1
28                 self._application.deliver(content)
29                 return self.try_deliver()
```

```
30 def deliver(self, message):
31     if isinstance(message, Order):
32         self._order[message.message_id()] = message.order()
33         self._a_seq = max(self._a_seq, message.order())
34         self.try_deliver()
35     else:
36         (sid, sseq, content) = message
37         self._hb_q[(sid, sseq)] = content
38         self._p_seq = max(self._a_seq, self._p_seq) + 1
39         # We should technically send proposer ID for tie-breaks
40         self.medium().send(
41             Vote(self.index(), sid, self._p_seq, (sid, sseq))
42         )
43
44 def forward(self, message):
45     if isinstance(message, Vote):
46         votes = self._votes[message.message_id()]
47         votes.append(message.order())
48         if len(votes) == self.number_of_devices():
49             self._b_multicast.send(
50                 Order(message.message_id(), max(votes))
51             )
52     else:
53         self._application.forward(message)
```

Is it totally ordered?

- Let us consider processes A and B, and messages m and n
- By executing the protocol, A assigns timestamp 1 to m and 2 to n, and it delivers m before n
- Let us consider that B delivers n before m?
  - Let us imagine that, based on received Votes, B considers that m has a timestamp larger than 1, and that's why B delivers n before m. This scenario is not possible, since the final timestamp can only grow, and the final timestamp of n must be 1.
  - Let us imagine that the proposed timestamp for n is currently lower than m's, and thus B wants to deliver n before m. This scenario is not possible since the final timestamp of n must be 2, thus the current timestamp of n is not final thus n cannot be delivered yet.
  - Let us imagine that B knows nothing about m, and n has already its final timestamp of 2 thus B wants to deliver it. This scenario is not possible since, in this case, B will vote for a timestamp larger than 2 for m, leading to a final timestamp larger than 2, which is not possible since the final timestamp of m must be 1.

Totally Ordered Multicast (ISIS)

- Good: Reliable crash-detection = robust
  - Sequence numbers are monotonically increasing
  - Nobody will deliver “early”
- Bad: every broadcast requires negotiation (3 rounds)
  - Sequencer has 2 rounds

Causally Ordered Multicast

Idea

- Order events by **happened-before** relationship,
- Use “vectored” not-quite-Lamport clocks (Vector Clocks), and
- Track only “send” as an event.

Lamport Clocks are not always enough

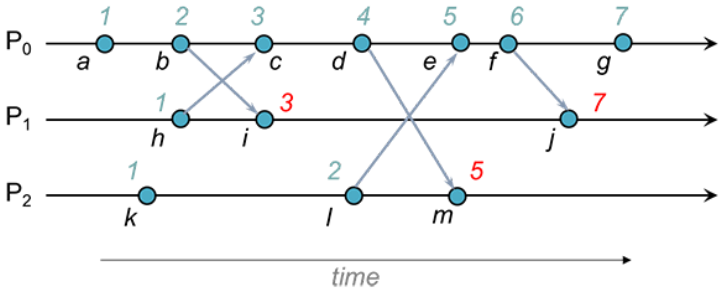


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

Not-quite-Lamport clocks, and they are vectors

- keep track of “last known time” of other processes
- “gossip” about “last known time” during communication

Vector Clocks

Let V_i be the vector of process $p_i \in \{p_0, \ldots, p_n\}$, then

- initially $V_i[j] = 0$ for all $j \in 0 \ldots n$,
- before event $V_i'[i] = V_i'[i] + 1$,
- attach V to any message sent,
- on recive of V' we let $V''[j] = max(V[j], V'[i])$ for $j \in 0 \ldots n$.

Vector Clocks

Given two vectors V and W,

- $V = W$ if all values match,
  - for all $j \in 0 \ldots n$, $V[j] = W[j]$
- $V \leq W$ if all values in V are less than or equal those in W,
  - for all $j \in 0 \ldots n$, $V[j] \leq W[j]$
- $V < W$ if all values in V are less than or equal to W **and** $W \neq V$
  - $V \leq W$ and $V \neq W$

Vector Clocks

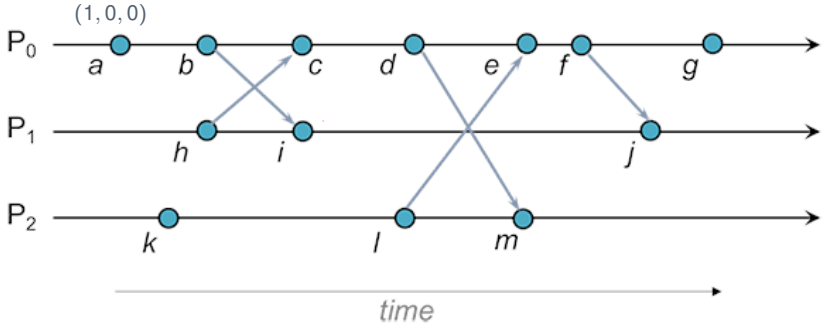


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

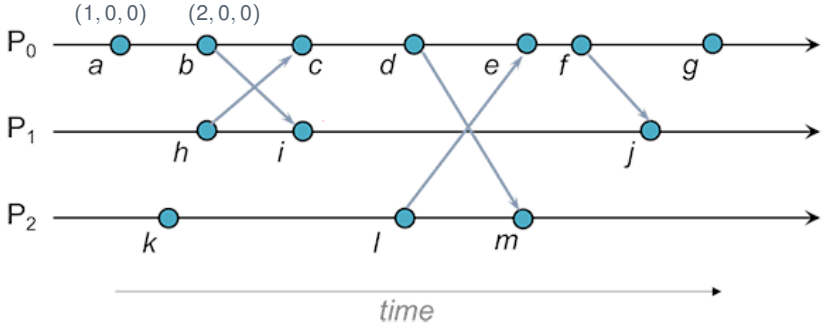


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

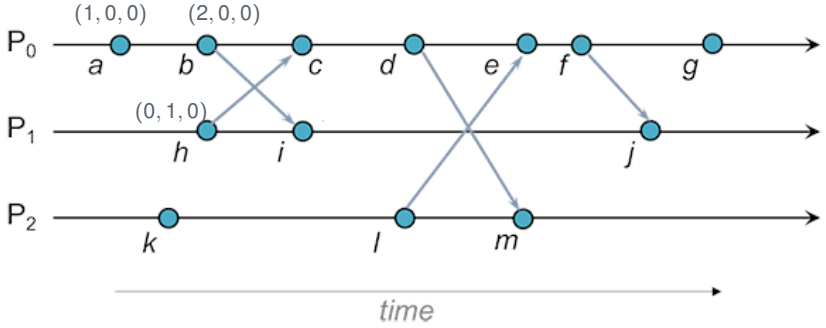


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

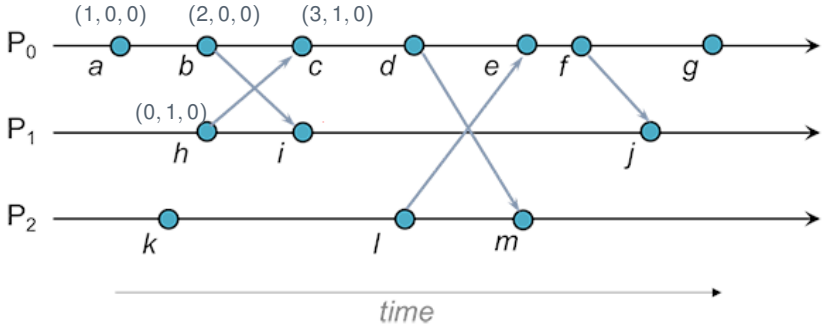


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

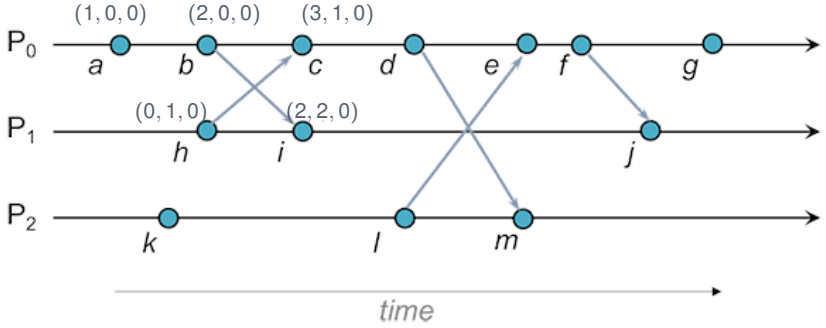


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

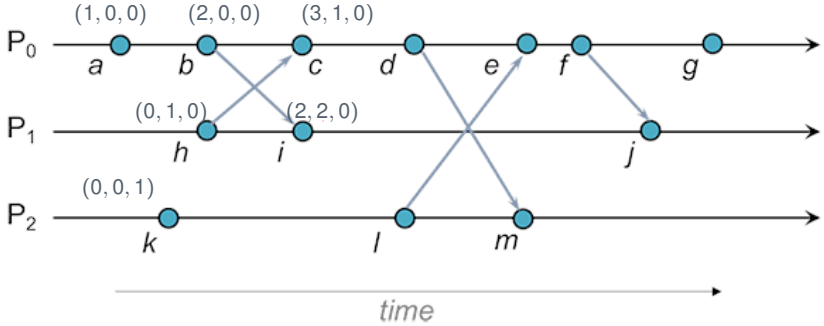


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

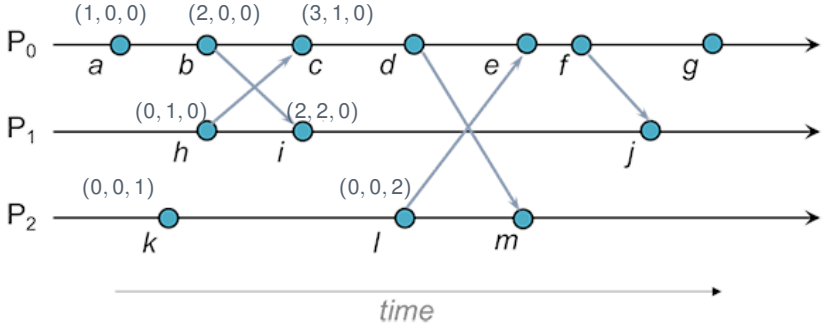


Figure: Does *l* happen between *h* and *i*?

Vector Clocks

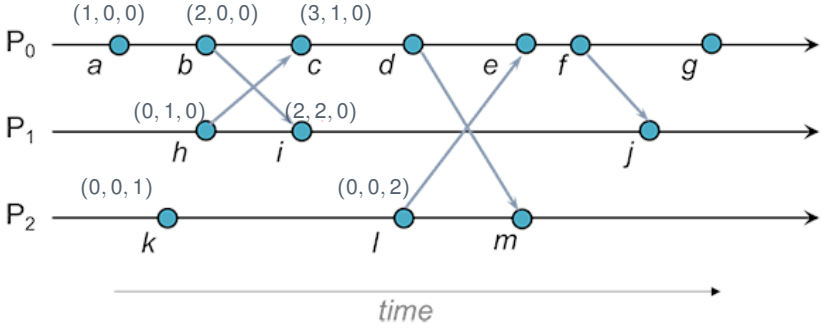


Figure: *l* happens concurrently with *h* and *i*

Causal Ordered Multicast


```
1  def __init__(...):
2      super().__init__(...)
3      self._application = application
4      self._b_multicast = BasicMulticast(index,
           number_of_devices, medium, self)
5      self._n_vect = [-1 for _ in self.medium().ids()]
6      self._hb_q = []
7
8  def send(self, content):
9      self._n_vect[self.index()] += 1
10     self._b_multicast.send((self._n_vect, self.index(),
           content))
11
12 def deliver(self, message):
13     self._hb_q.append(message)
14     self.try_deliver()
```

```
15 def forward(self, message):
16     self._application.forward(message)
17
18 def try_deliver(self):
19     for (vec, index, content) in self._hb_q:
20         if self.is_next(vec, index):
21             self._application.deliver(content)
22             self._n_vect[index] += 1
23             return self.try_deliver()
24
25 def is_next(self, vec, index):
26     if vec[index] != self._n_vect[index] + 1:
27         return False
28     for i in self.medium().ids():
29         if i != index and vec[i] > self._n_vect[i]:
30             return False
31     return True
32
33 def run(self):
34     self._b_multicast.run()
```

Causal Ordered Multicast

Notice

- Causal order implies FIFO
- Causal order does **not** imply Total
- Good: No extra communication for order!
- Can be reliable and totally ordered, how?

Causal Ordered Multicast

Notice

- Causal order implies FIFO
- Causal order does **not** imply Total
- Good: No extra communication for order!
- Can be reliable and totally ordered, how?
  - Use R-multicast instead of B-multicast.

Causal Ordered Multicast

Notice

- Causal order implies FIFO
- Causal order does **not** imply Total
- Good: No extra communication for order!
- Can be reliable and totally ordered, how?
  - Use R-multicast instead of B-multicast.
  - Use CO multicast in any TO multicast algorithm instead of B-multicast

Lesson of Today

- Multicast is notoriously difficult
  - be careful of homebrewed solutions
- All algorithms shown work in async setting

Not discussed

- Many can be combined/layered
  - reliable causally ordered multicast over IP)
- Multiple groups = extra complexity
- Totally Ordered Reliable mutlticast = impossible in async setting