Distributed Systems

The second half of *Concurrent and Distributed Systems*

https://www.cl.cam.ac.uk/teaching/current/ConcDisSys

Dr. Martin Kleppmann (mk428@cam)

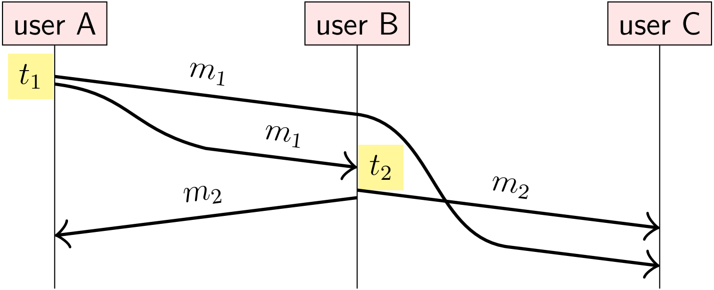
University of Cambridge

Computer Science Tripos, Part IB

Lecture 4

Broadcast protocols and logical time

Physical timestamps inconsistent with causality



$m_1 = (t_1,$ "A says: The moon is made of cheese!"$)$
$m_2 = (t_2,$ "B says: Oh no it isn't!"$)$

Problem: even with synced clocks, $t_2 < t_1$ is possible.
Timestamp order is inconsistent with expected order!

Logical vs. physical clocks

- Physical clock: count number of **seconds elapsed**
- Logical clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be **inconsistent with causality**.

Logical vs. physical clocks

- Physical clock: count number of **seconds elapsed**
- Logical clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be **inconsistent with causality**.

Logical clocks: designed to **capture causal dependencies**.

$$(e_1 \to e_2) \implies (T(e_1) < T(e_2))$$

Logical vs. physical clocks

- Physical clock: count number of **seconds elapsed**
- Logical clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be **inconsistent with causality**.

Logical clocks: designed to **capture causal dependencies**.

$$(e_1 \to e_2) \Longrightarrow (T(e_1) < T(e_2))$$

We will look at two types of logical clocks:

- Lamport clocks
- Vector clocks

Lamport clocks algorithm

on initialisation **do**
    $t := 0$ ▷ each node has its own local variable t
end on

on any event occurring at the local node **do**
    $t := t + 1$
end on

on request to send message m **do**
    $t := t + 1$; send (t, m) via the underlying network link
end on

on receiving (t', m) via the underlying network link **do**
    $t := \max(t, t') + 1$
    deliver m to the application
end on

Lamport clocks in words

- Each node maintains a counter t,
  incremented on every local event e
- Let $L(e)$ be the value of t after that increment
- Attach current t to messages sent over network
- Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

Lamport clocks in words

- Each node maintains a counter t,
  incremented on every local event e
- Let $L(e)$ be the value of t after that increment
- Attach current t to messages sent over network
- Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

Properties of this scheme:

- If $a \rightarrow b$ then $L(a) < L(b)$

Lamport clocks in words

- Each node maintains a counter t,
  incremented on every local event e
- Let $L(e)$ be the value of t after that increment
- Attach current t to messages sent over network
- Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

Properties of this scheme:

- If $a \rightarrow b$ then $L(a) < L(b)$
- However, $L(a) < L(b)$ does not imply $a \rightarrow b$

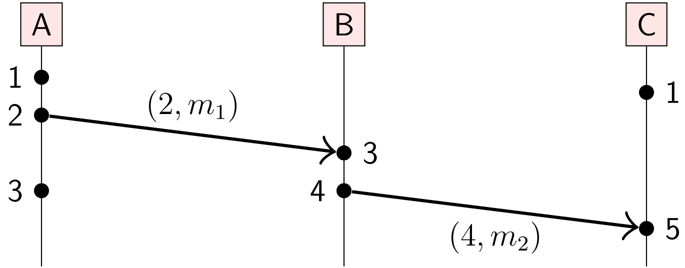
Lamport clocks in words

- Each node maintains a counter t,
  incremented on every local event e
- Let $L(e)$ be the value of t after that increment
- Attach current t to messages sent over network
- Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

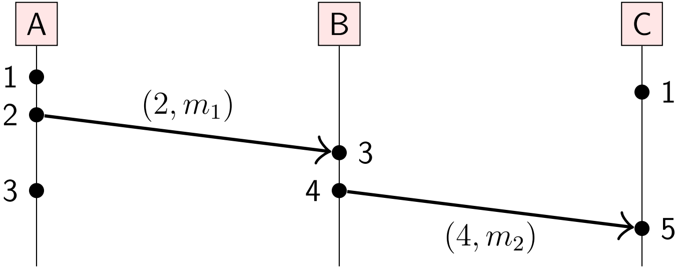
Properties of this scheme:

- If $a \to b$ then $L(a) < L(b)$
- However, $L(a) < L(b)$ does not imply $a \to b$
- Possible that $L(a) = L(b)$ for $a \neq b$

Lamport clocks example

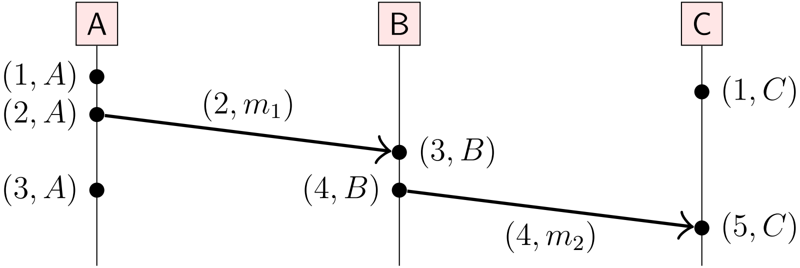


Lamport clocks example



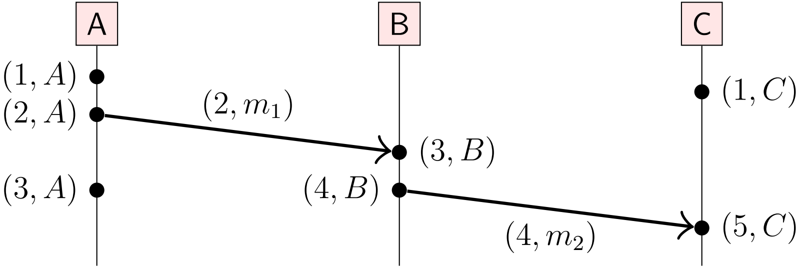
Let $N(e)$ be the node at which event e occurred.
Then the pair $(L(e), N(e))$ **uniquely identifies** event e.

Lamport clocks example



Let $N(e)$ be the node at which event e occurred.
Then the pair $(L(e), N(e))$ **uniquely identifies** event e.

Lamport clocks example



Let $N(e)$ be the node at which event e occurred.
Then the pair $(L(e), N(e))$ **uniquely identifies** event e.

Define a **total order** $\prec$ using Lamport timestamps:

$$(a \prec b) \iff (L(a) < L(b) \lor (L(a) = L(b) \land N(a) < N(b)))$$

This order is **causal**: $(a \rightarrow b) \Longrightarrow (a \prec b)$

Vector clocks

Given Lamport timestamps $L(a)$ and $L(b)$ with $L(a) < L(b)$ we can't tell whether $a \rightarrow b$ or $a \parallel b$.

If we want to detect which events are concurrent, we need **vector clocks**:

Vector clocks

Given Lamport timestamps $L(a)$ and $L(b)$ with $L(a) < L(b)$ we can't tell whether $a \rightarrow b$ or $a \parallel b$.

If we want to detect which events are concurrent, we need **vector clocks**:

- Assume n nodes in the system, $N = \langle N_1, N_2, \ldots, N_n \rangle$

Vector clocks

Given Lamport timestamps $L(a)$ and $L(b)$ with $L(a) < L(b)$ we can't tell whether $a \rightarrow b$ or $a \parallel b$.

If we want to detect which events are concurrent, we need **vector clocks**:

- Assume n nodes in the system, $N = \langle N_1, N_2, \ldots, N_n \rangle$
- Vector timestamp of event a is $V(a) = \langle t_1, t_2, \ldots, t_n \rangle$
- t_i is number of events observed by node N_i

Vector clocks

Given Lamport timestamps $L(a)$ and $L(b)$ with $L(a) < L(b)$ we can't tell whether $a \rightarrow b$ or $a \parallel b$.

If we want to detect which events are concurrent, we need **vector clocks**:

- Assume n nodes in the system, $N = \langle N_1, N_2, \ldots, N_n \rangle$
- Vector timestamp of event a is $V(a) = \langle t_1, t_2, \ldots, t_n \rangle$
- t_i is number of events observed by node N_i
- Each node has a current vector timestamp T
- On event at node N_i, increment vector element $T[i]$

Vector clocks

Given Lamport timestamps $L(a)$ and $L(b)$ with $L(a) < L(b)$ we can't tell whether $a \rightarrow b$ or $a \parallel b$.

If we want to detect which events are concurrent, we need **vector clocks**:

- Assume n nodes in the system, $N = \langle N_1, N_2, \ldots, N_n \rangle$
- Vector timestamp of event a is $V(a) = \langle t_1, t_2, \ldots, t_n \rangle$
- t_i is number of events observed by node N_i
- Each node has a current vector timestamp T
- On event at node N_i, increment vector element $T[i]$
- Attach current vector timestamp to each message
- Recipient merges message vector into its local vector

Vector clocks algorithm

on initialisation at node N_i **do**
  $T := \langle 0, 0, \dots, 0 \rangle$ ▷ local variable at node N_i
end on

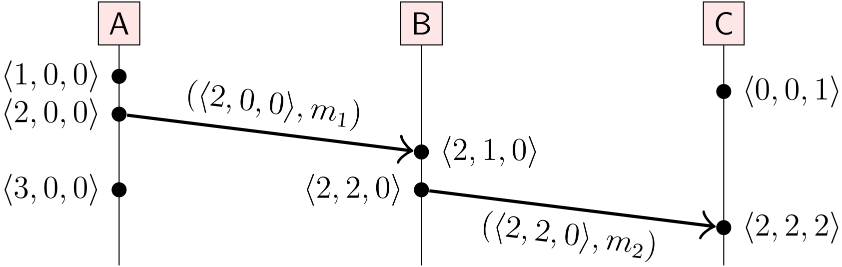
on any event occurring at node N_i **do**
  $T[i] := T[i] + 1$
end on

on request to send message m at node N_i **do**
  $T[i] := T[i] + 1$; send (T, m) via network
end on

on receiving (T', m) at node N_i via the network **do**
  $T[j] := \max(T[j], T'[j])$ for every $j \in \{1, \dots, n\}$
  $T[i] := T[i] + 1$; deliver m to the application
end on

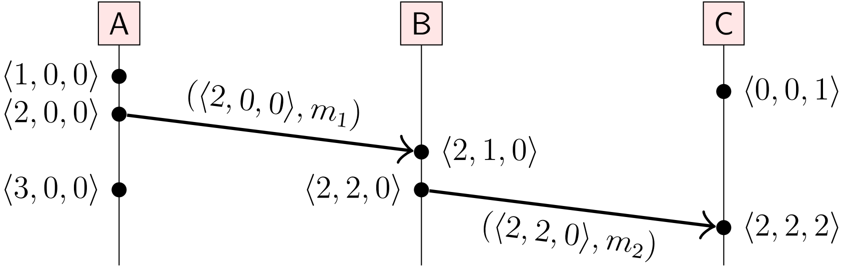
Vector clocks example

Assuming the vector of nodes is $N = \langle A, B, C \rangle$:



Vector clocks example

Assuming the vector of nodes is $N = \langle A, B, C \rangle$:



The vector timestamp of an event e represents a set of events, e and its causal dependencies: $\{e\} \cup \{a \mid a \rightarrow e\}$

For example, $\langle 2, 2, 0 \rangle$ represents the first two events from A, the first two events from B, and no events from C.

Vector clocks ordering

Define the following order on vector timestamps (in a system with n nodes):

- $T = T'$ iff $T[i] = T'[i]$ for all $i \in \{1, \dots, n\}$
- $T \leq T'$ iff $T[i] \leq T'[i]$ for all $i \in \{1, \dots, n\}$
- $T < T'$ iff $T \leq T'$ and $T \neq T'$
- $T \parallel T'$ iff $T \not\leq T'$ and $T' \not\leq T$

Vector clocks ordering

Define the following order on vector timestamps (in a system with n nodes):

- $T = T'$ iff $T[i] = T'[i]$ for all $i \in \{1, \dots, n\}$
- $T \leq T'$ iff $T[i] \leq T'[i]$ for all $i \in \{1, \dots, n\}$
- $T < T'$ iff $T \leq T'$ and $T \neq T'$
- $T \parallel T'$ iff $T \not\leq T'$ and $T' \not\leq T$

$V(a) \leq V(b)$ iff $(\{a\} \cup \{e \mid e \rightarrow a\}) \subseteq (\{b\} \cup \{e \mid e \rightarrow b\})$

Vector clocks ordering

Define the following order on vector timestamps (in a system with n nodes):

- $T = T'$ iff $T[i] = T'[i]$ for all $i \in \{1, \dots, n\}$
- $T \leq T'$ iff $T[i] \leq T'[i]$ for all $i \in \{1, \dots, n\}$
- $T < T'$ iff $T \leq T'$ and $T \neq T'$
- $T \parallel T'$ iff $T \nleq T'$ and $T' \nleq T$

$V(a) \leq V(b)$ iff $(\{a\} \cup \{e \mid e \to a\}) \subseteq (\{b\} \cup \{e \mid e \to b\})$

Properties of this order:

- $(V(a) < V(b)) \iff (a \to b)$
- $(V(a) = V(b)) \iff (a = b)$
- $(V(a) \parallel V(b)) \iff (a \parallel b)$

Broadcast protocols

Broadcast (multicast) is **group communication**:

- One node sends message, all nodes in group deliver it

Broadcast protocols

Broadcast (multicast) is **group communication**:

- One node sends message, all nodes in group deliver it
- Set of group members may be fixed (static) or dynamic

Broadcast protocols

Broadcast (multicast) is **group communication**:

- One node sends message, all nodes in group deliver it
- Set of group members may be fixed (static) or dynamic
- If one node is faulty, remaining group members carry on

Broadcast protocols

Broadcast (multicast) is **group communication**:

- One node sends message, all nodes in group deliver it
- Set of group members may be fixed (static) or dynamic
- If one node is faulty, remaining group members carry on
- Note: concept is more general than IP multicast
  (we build upon point-to-point messaging)

Broadcast protocols

Broadcast (multicast) is **group communication**:

- One node sends message, all nodes in group deliver it
- Set of group members may be fixed (static) or dynamic
- If one node is faulty, remaining group members carry on
- Note: concept is more general than IP multicast
  (we build upon point-to-point messaging)

Build upon system models from lecture 2:

- Can be **best-effort** (may drop messages) or
  reliable (non-faulty nodes deliver every message,
  by retransmitting dropped messages)

Broadcast protocols

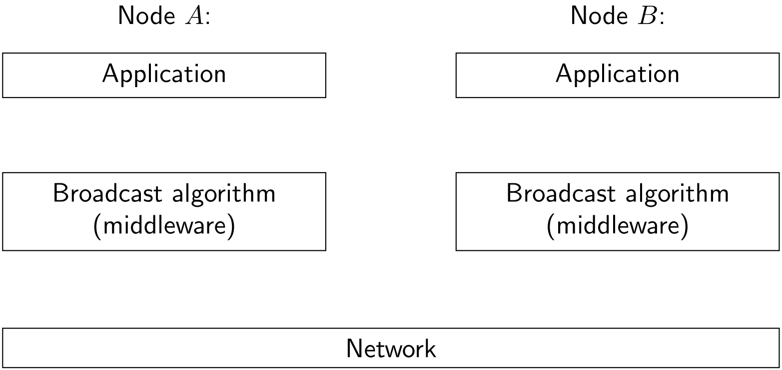
Broadcast (multicast) is **group communication**:

- One node sends message, all nodes in group deliver it
- Set of group members may be fixed (static) or dynamic
- If one node is faulty, remaining group members carry on
- Note: concept is more general than IP multicast (we build upon point-to-point messaging)

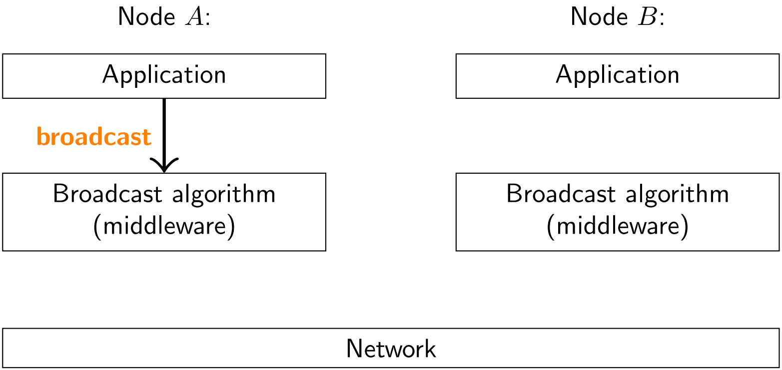
Build upon system models from lecture 2:

- Can be **best-effort** (may drop messages) or **reliable** (non-faulty nodes deliver every message, by retransmitting dropped messages)
- Asynchronous/partially synchronous timing model $\Longrightarrow$ **no upper bound** on message latency

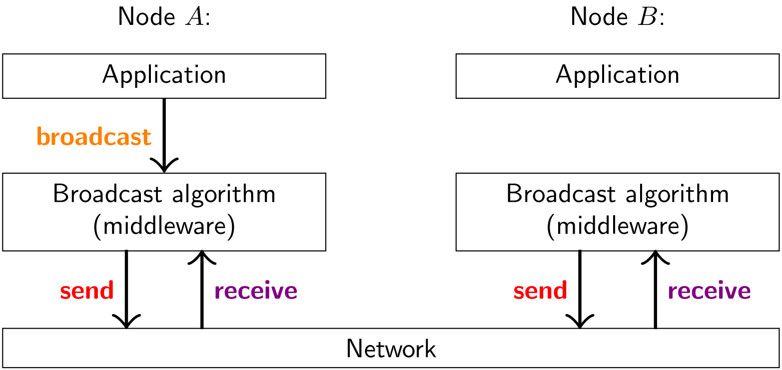
Receiving versus delivering



Receiving versus delivering

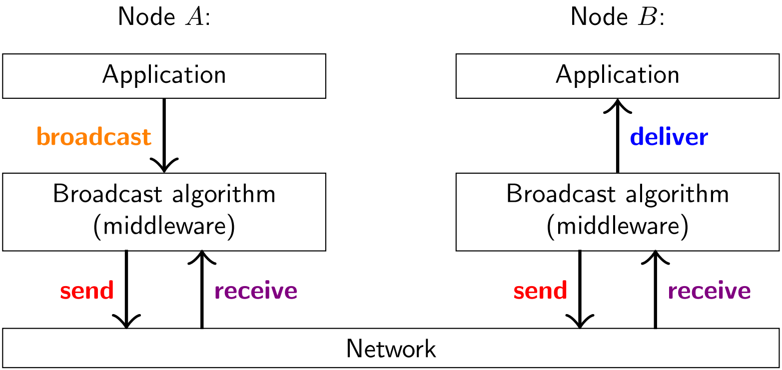


Receiving versus delivering



Assume network provides point-to-point **send**/**receive**

Receiving versus delivering



Assume network provides point-to-point **send**/**receive**

After broadcast algorithm **receives** message from network, it may buffer/queue it before **delivering** to the application

Forms of reliable broadcast

FIFO broadcast:
If m_1 and m_2 are broadcast by the same node, and broadcast$(m_1) \rightarrow$ broadcast(m_2), then m_1 must be delivered before m_2

Forms of reliable broadcast

FIFO broadcast:
If m_1 and m_2 are broadcast by the same node, and broadcast$(m_1) \rightarrow$ broadcast(m_2), then m_1 must be delivered before m_2

Causal broadcast:
If broadcast$(m_1) \rightarrow$ broadcast(m_2) then m_1 must be delivered before m_2

Forms of reliable broadcast

FIFO broadcast:
If m_1 and m_2 are broadcast by the same node, and broadcast$(m_1) \rightarrow$ broadcast(m_2), then m_1 must be delivered before m_2

Causal broadcast:
If broadcast$(m_1) \rightarrow$ broadcast(m_2) then m_1 must be delivered before m_2

Total order broadcast:
If m_1 is delivered before m_2 on one node, then m_1 must be delivered before m_2 on all nodes

Forms of reliable broadcast

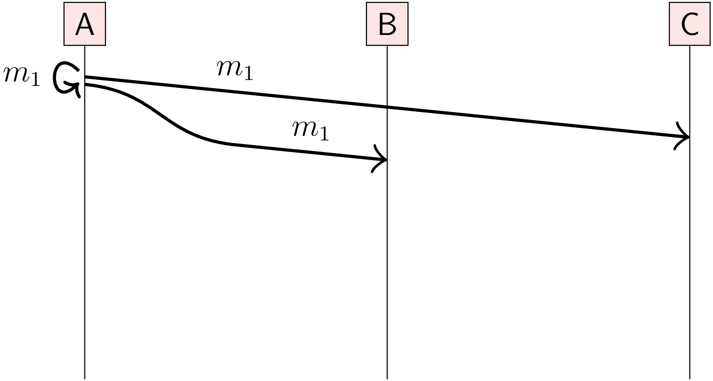
FIFO broadcast:
If m_1 and m_2 are broadcast by the same node, and broadcast$(m_1) \rightarrow$ broadcast(m_2), then m_1 must be delivered before m_2

Causal broadcast:
If broadcast$(m_1) \rightarrow$ broadcast(m_2) then m_1 must be delivered before m_2

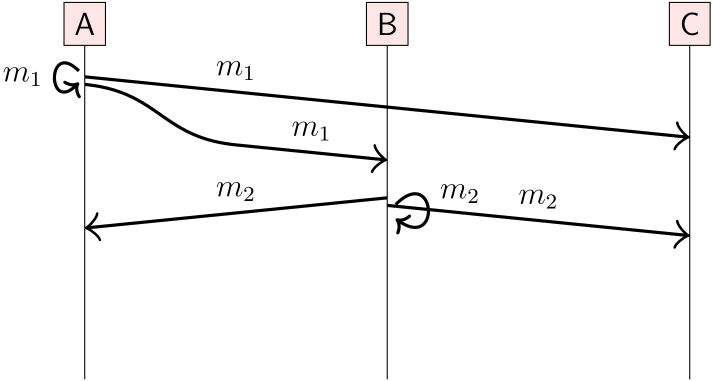
Total order broadcast:
If m_1 is delivered before m_2 on one node, then m_1 must be delivered before m_2 on all nodes

FIFO-total order broadcast:
Combination of FIFO broadcast and total order broadcast

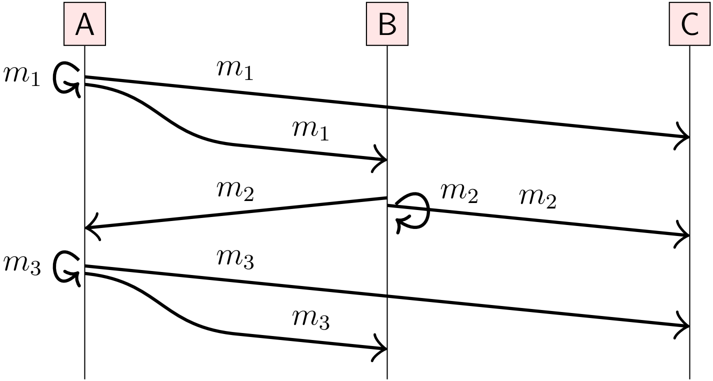
FIFO broadcast



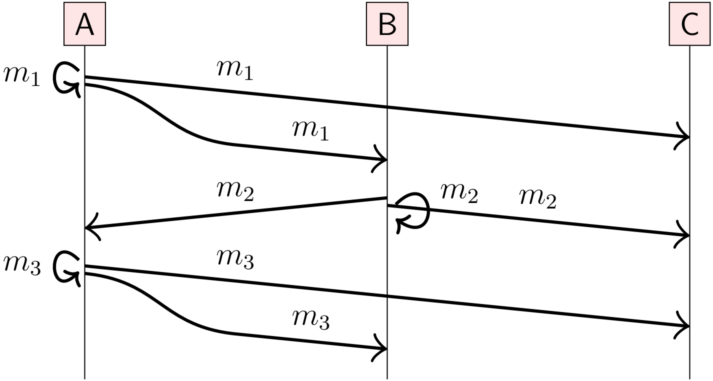
FIFO broadcast



FIFO broadcast

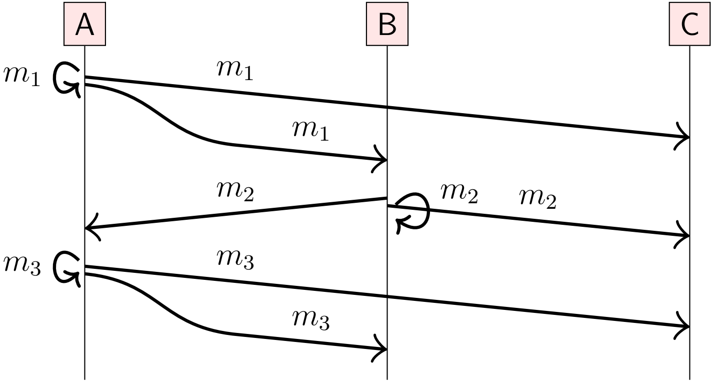


FIFO broadcast



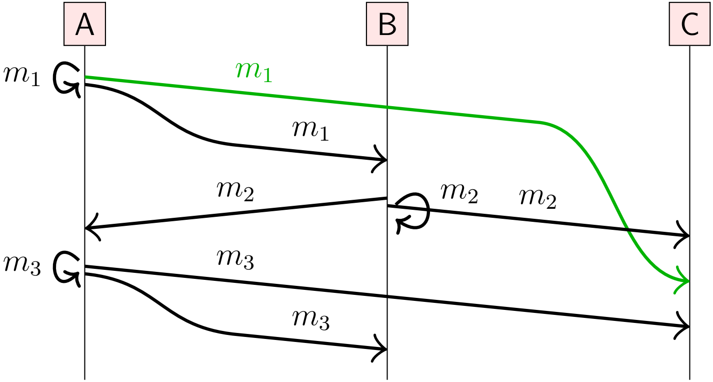
Messages sent by the same node must be delivered in the order they were sent.
Messages sent by different nodes can be delivered in any order.

FIFO broadcast



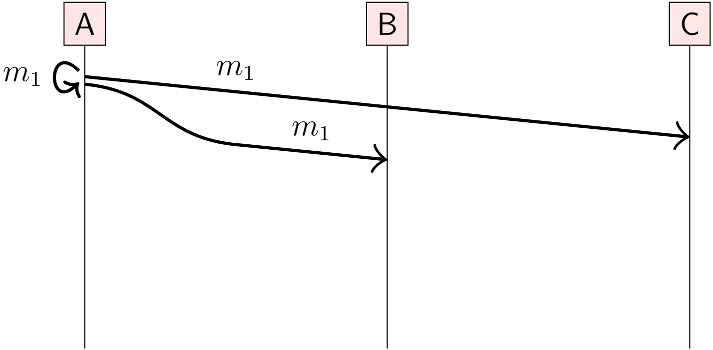
Messages sent by the same node must be delivered in the order they were sent.
Messages sent by different nodes can be delivered in any order.
Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

FIFO broadcast

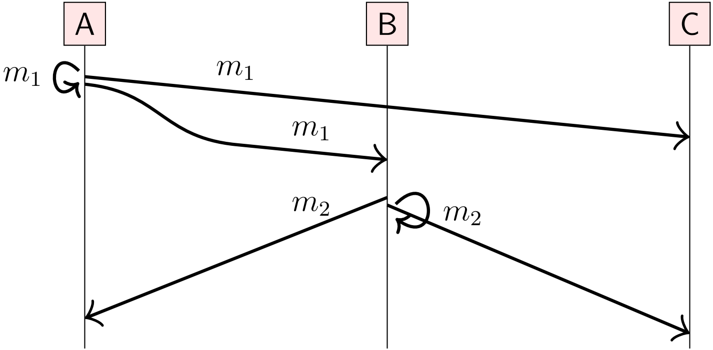


Messages sent by the same node must be delivered in the order they were sent.
Messages sent by different nodes can be delivered in any order.
Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

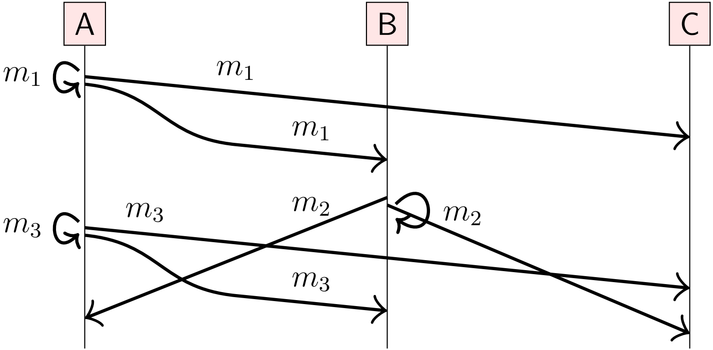
Causal broadcast



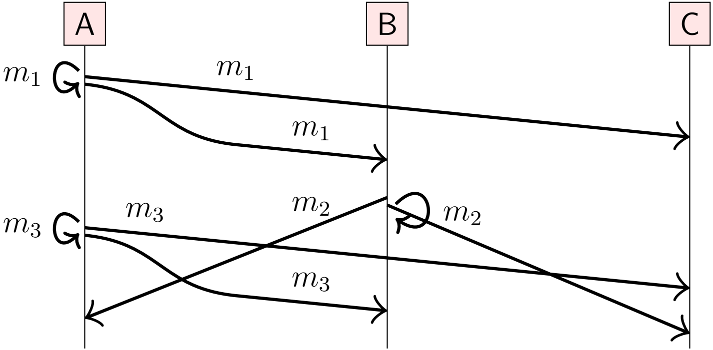
Causal broadcast



Causal broadcast

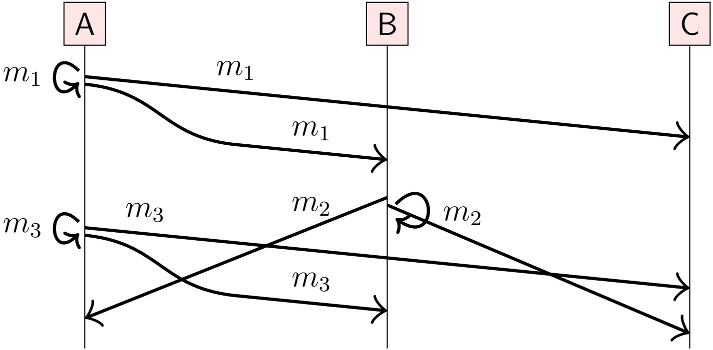


Causal broadcast



Causally related messages must be delivered in causal order.
Concurrent messages can be delivered in any order.

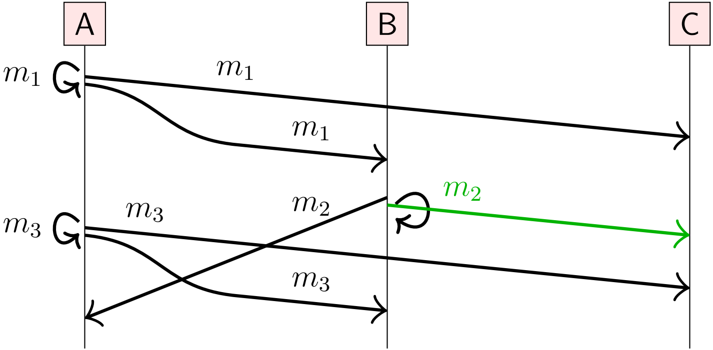
Causal broadcast



Causally related messages must be delivered in causal order.
Concurrent messages can be delivered in any order.

Here: broadcast$(m_1) \rightarrow$ broadcast(m_2) and
broadcast$(m_1) \rightarrow$ broadcast(m_3)
$\Longrightarrow$ valid orders are: (m_1, m_2, m_3) or (m_1, m_3, m_2)

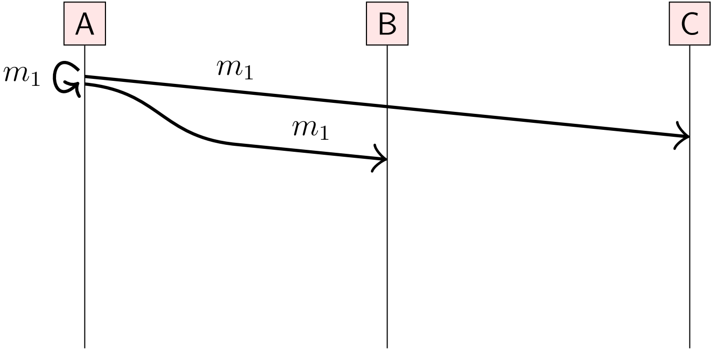
Causal broadcast



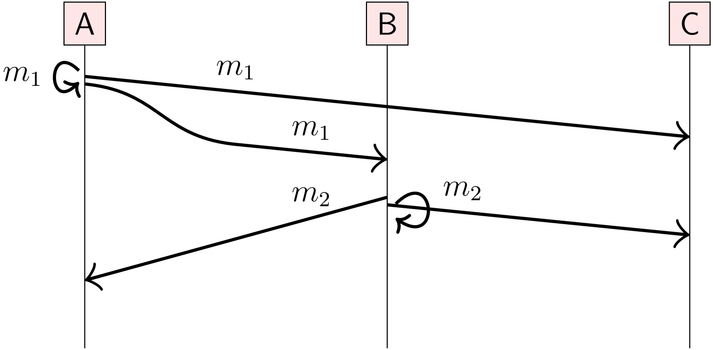
Causally related messages must be delivered in causal order.
Concurrent messages can be delivered in any order.

Here: broadcast$(m_1) \rightarrow$ broadcast(m_2) and
broadcast$(m_1) \rightarrow$ broadcast(m_3)
$\Longrightarrow$ valid orders are: (m_1, m_2, m_3) or (m_1, m_3, m_2)

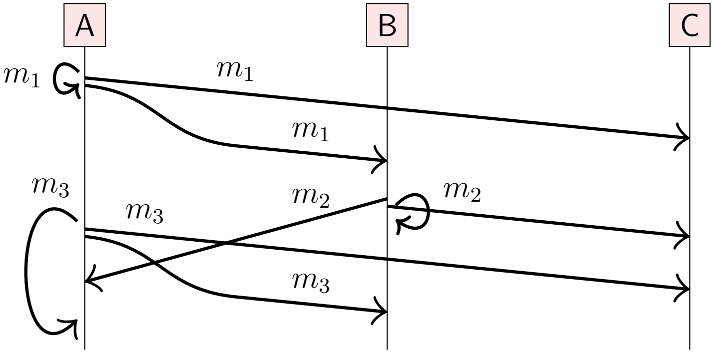
Total order broadcast (1)



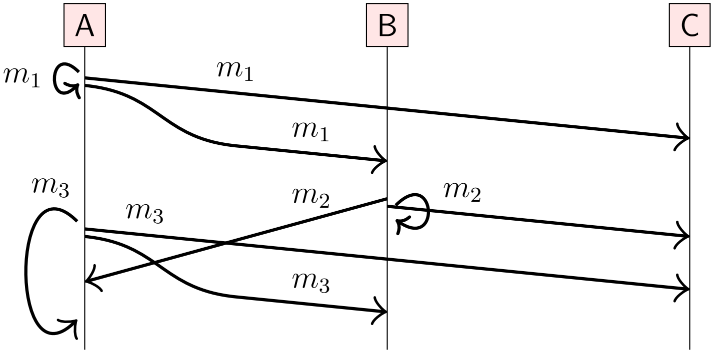
Total order broadcast (1)



Total order broadcast (1)

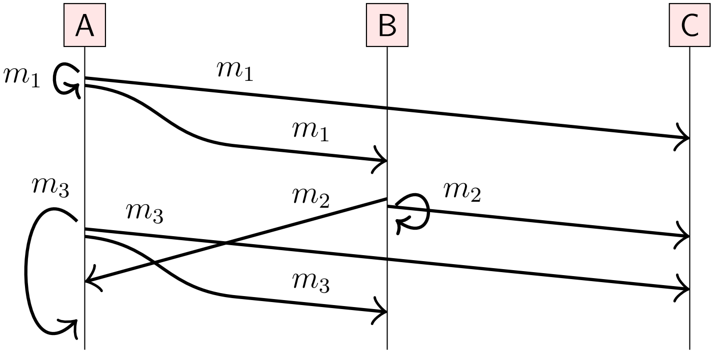


Total order broadcast (1)



All nodes must deliver messages in **the same** order
(here: m_1, m_2, m_3)

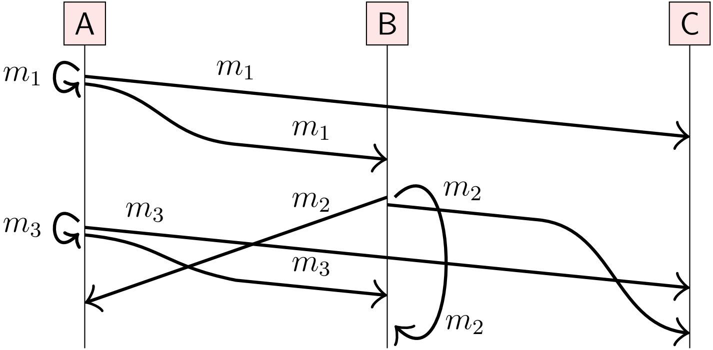
Total order broadcast (1)



All nodes must deliver messages in **the same** order
(here: m_1, m_2, m_3)

This includes a node's deliveries to itself!

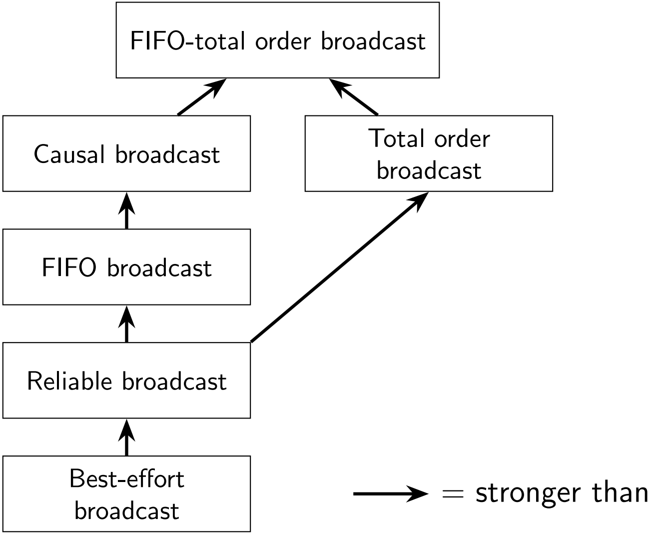
Total order broadcast (2)



All nodes must deliver messages in **the same** order (here: m_1, m_3, m_2)

This includes a node's deliveries to itself!

Relationships between broadcast models



Broadcast algorithms

Break down into two layers:

1. Make best-effort broadcast reliable by retransmitting dropped messages
2. Enforce delivery order on top of reliable broadcast

Broadcast algorithms

Break down into two layers:

1. Make best-effort broadcast reliable by retransmitting dropped messages
2. Enforce delivery order on top of reliable broadcast

First attempt: **broadcasting node sends message directly** to every other node

- Use reliable links (retry + deduplicate)

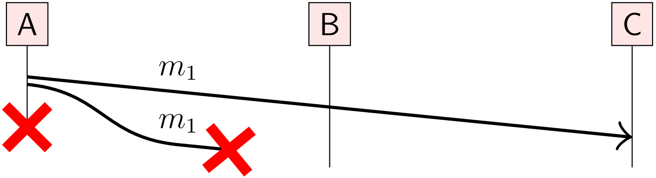
Broadcast algorithms

Break down into two layers:

1. Make best-effort broadcast reliable by retransmitting dropped messages
2. Enforce delivery order on top of reliable broadcast

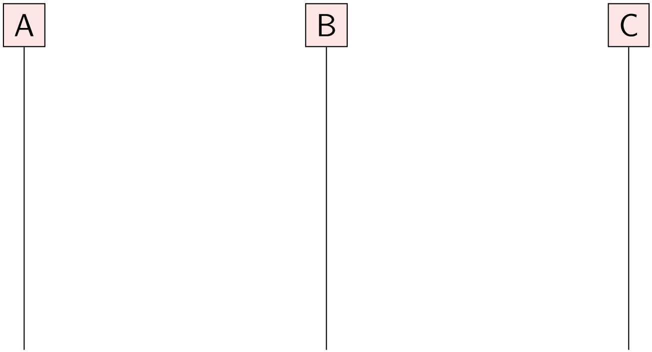
First attempt: **broadcasting node sends message directly** to every other node

- Use reliable links (retry + deduplicate)
- Problem: node may crash before all messages delivered



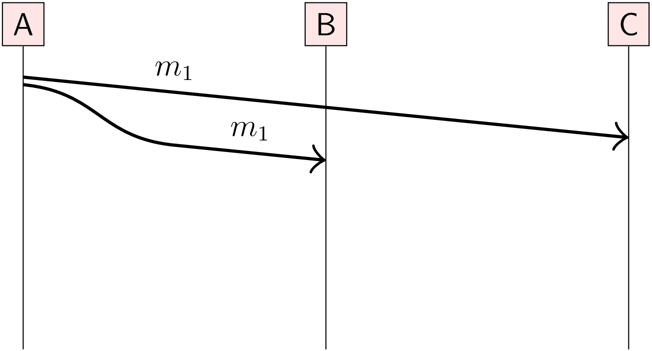
Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



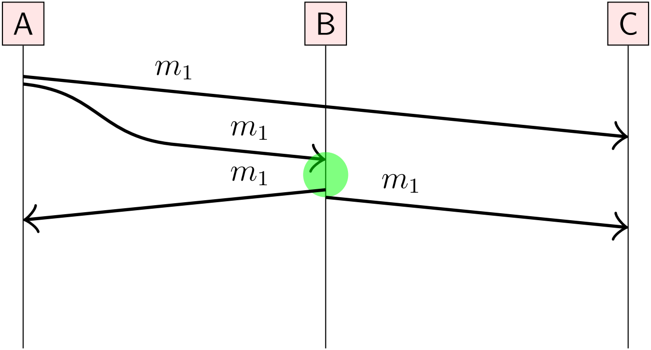
Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



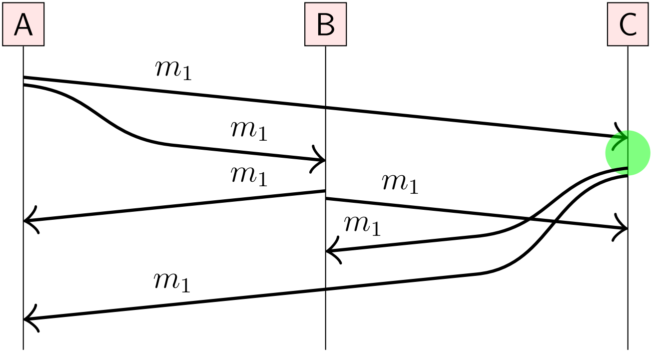
Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



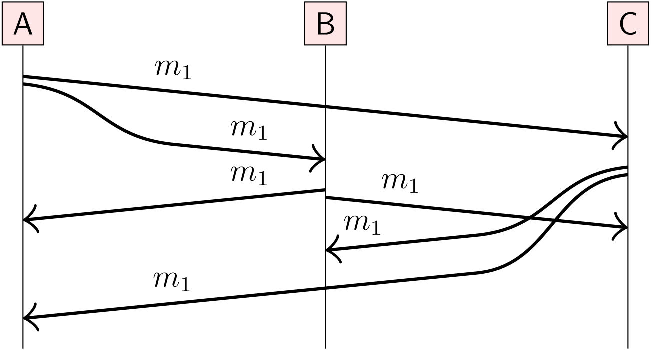
Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



Eager reliable broadcast

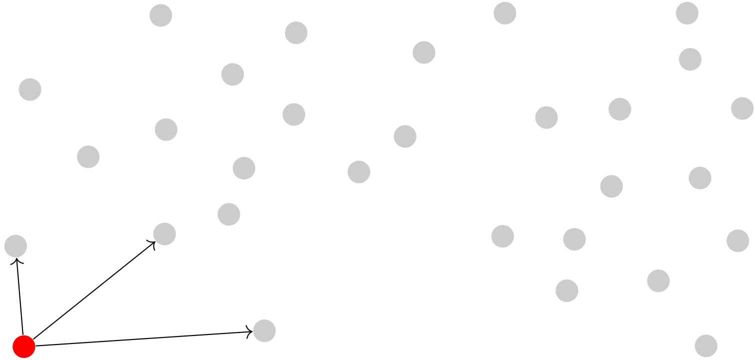
Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



Reliable, but... up to $O(n^2)$ messages for n nodes!

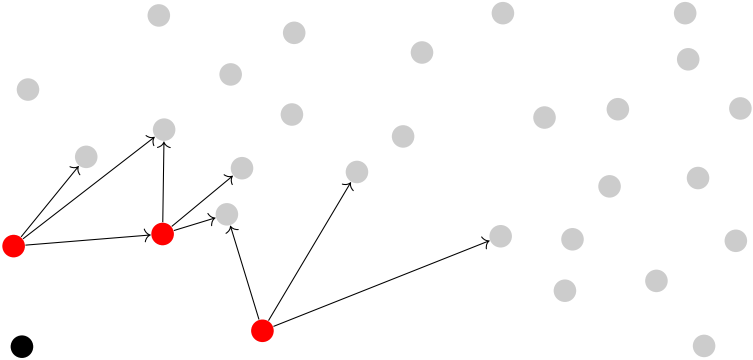
Gossip protocols

Useful when broadcasting to a large number of nodes.
Idea: when a node receives a message for the first time, **forward it to 3 other nodes**, chosen randomly.



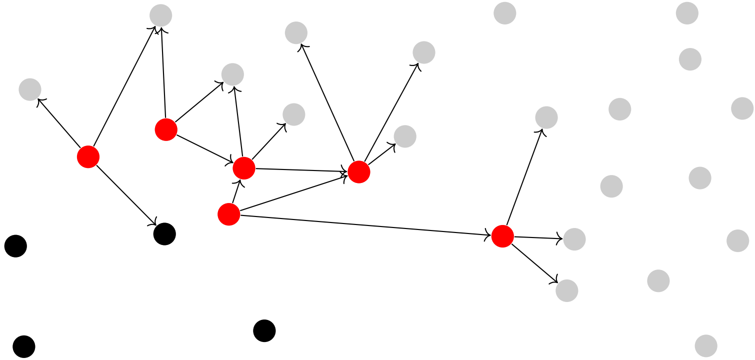
Gossip protocols

Useful when broadcasting to a large number of nodes.
Idea: when a node receives a message for the first time, **forward it to 3 other nodes**, chosen randomly.



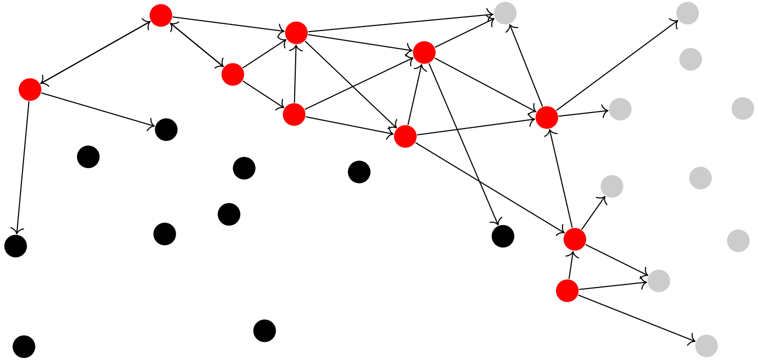
Gossip protocols

Useful when broadcasting to a large number of nodes.
Idea: when a node receives a message for the first time, **forward it to 3 other nodes**, chosen randomly.



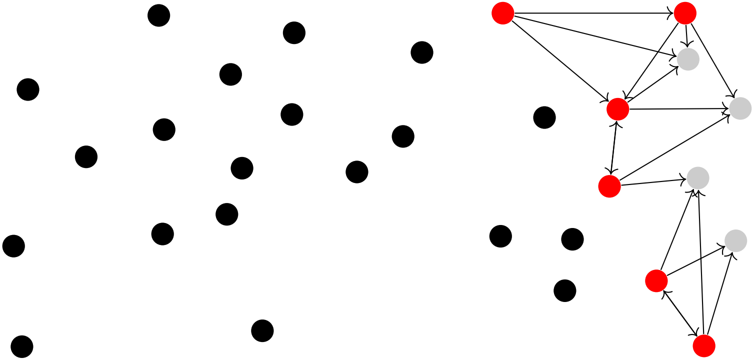
Gossip protocols

Useful when broadcasting to a large number of nodes.
Idea: when a node receives a message for the first time, **forward it to 3 other nodes**, chosen randomly.



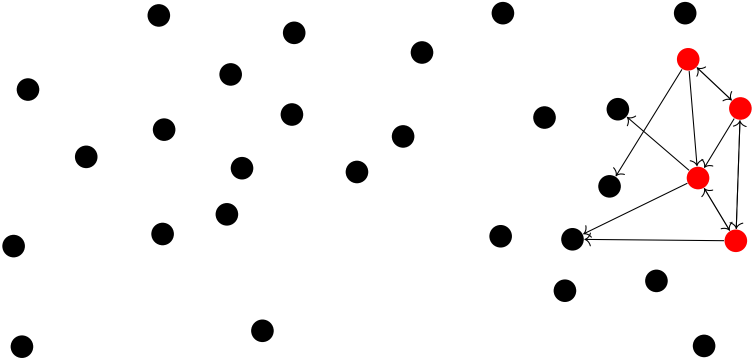
Gossip protocols

Useful when broadcasting to a large number of nodes.
Idea: when a node receives a message for the first time, **forward it to 3 other nodes**, chosen randomly.



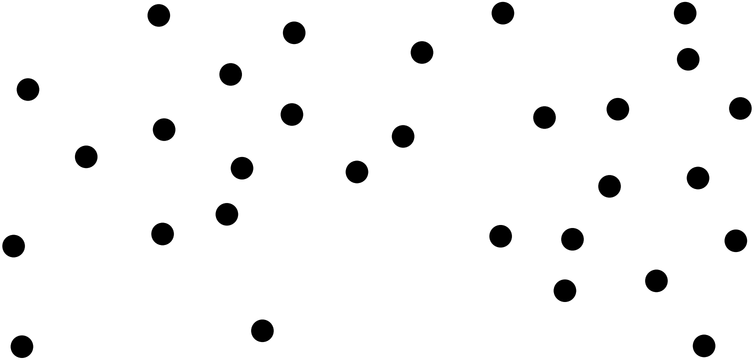
Gossip protocols

Useful when broadcasting to a large number of nodes.
Idea: when a node receives a message for the first time, **forward it to 3 other nodes**, chosen randomly.



Gossip protocols

Useful when broadcasting to a large number of nodes.
Idea: when a node receives a message for the first time, **forward it to 3 other nodes**, chosen randomly.



Eventually reaches all nodes (with high probability).

FIFO broadcast algorithm

on initialisation **do**
 $sendSeq := 0;\ delivered := \langle 0, 0, \dots, 0 \rangle;\ buffer := \{\}$
end on

on request to broadcast m at node N_i **do**
 send $(i, sendSeq, m)$ via reliable broadcast
 $sendSeq := sendSeq + 1$
end on

on receiving msg from reliable broadcast at node N_i **do**
 $buffer := buffer \cup \{msg\}$
 while $\exists sender, m.\ (sender, delivered[sender], m) \in buffer$ **do**
  deliver m to the application
  $delivered[sender] := delivered[sender] + 1$
 end while
end on

Causal broadcast algorithm

on initialisation **do**
  $sendSeq := 0;\ delivered := \langle 0, 0, \ldots, 0 \rangle;\ buffer := \{\}$
end on

on request to broadcast m at node N_i **do**
  $deps := delivered; deps[i] := sendSeq$
  send $(i, deps, m)$ via reliable broadcast
  $sendSeq := sendSeq + 1$
end on

on receiving msg from reliable broadcast at node N_i **do**
  $buffer := buffer \cup \{msg\}$
  while $\exists (sender, deps, m) \in buffer.\ deps \leq delivered$ **do**
    deliver m to the application
    $buffer := buffer \setminus \{(sender, deps, m)\}$
    $delivered[sender] := delivered[sender] + 1$
  end while
end on

Vector clocks ordering

Define the following order on vector timestamps (in a system with n nodes):

- $T = T'$ iff $T[i] = T'[i]$ for all $i \in \{1, \dots, n\}$
- $T \leq T'$ iff $T[i] \leq T'[i]$ for all $i \in \{1, \dots, n\}$
- $T < T'$ iff $T \leq T'$ and $T \neq T'$
- $T \parallel T'$ iff $T \not\leq T'$ and $T' \not\leq T$

Total order broadcast algorithms

Single leader approach:

- One node is designated as leader (sequencer)
- To broadcast message, send it to the leader; leader broadcasts it via FIFO broadcast.

Total order broadcast algorithms

Single leader approach:

- One node is designated as leader (sequencer)
- To broadcast message, send it to the leader;
  leader broadcasts it via FIFO broadcast.
- Problem: leader crashes $\implies$ no more messages delivered
- Changing the leader safely is difficult

Total order broadcast algorithms

Single leader approach:

- One node is designated as leader (sequencer)
- To broadcast message, send it to the leader;
  leader broadcasts it via FIFO broadcast.
- Problem: leader crashes $\Longrightarrow$ no more messages delivered
- Changing the leader safely is difficult

Lamport clocks approach:

- Attach Lamport timestamp to every message
- Deliver messages in total order of timestamps

Total order broadcast algorithms

Single leader approach:

- One node is designated as leader (sequencer)
- To broadcast message, send it to the leader;
  leader broadcasts it via FIFO broadcast.
- Problem: leader crashes $\Longrightarrow$ no more messages delivered
- Changing the leader safely is difficult

Lamport clocks approach:

- Attach Lamport timestamp to every message
- Deliver messages in total order of timestamps
- Problem: how do you know if you have seen all messages with timestamp $< T$? Need to use FIFO links and wait for message with timestamp $\geq T$ from *every* node