Distributed Algorithms 2016/17
Flooding, Broadcast and Echo

Odej Kao | Complex and Distributed IT Systems

With material of R. Karnapke | KBS

Overview

Flooding

- Distribution of Information (e.g. node-ID) with or without confirmation to all nodes using *all edges*

Echo

- Distribution of information to all nodes using all edges with selective confirmation
- Collecting information
- Construction of a spanning tree

Broadcast

- Distribution of information to all nodes with or without acknowledgement with special topologies

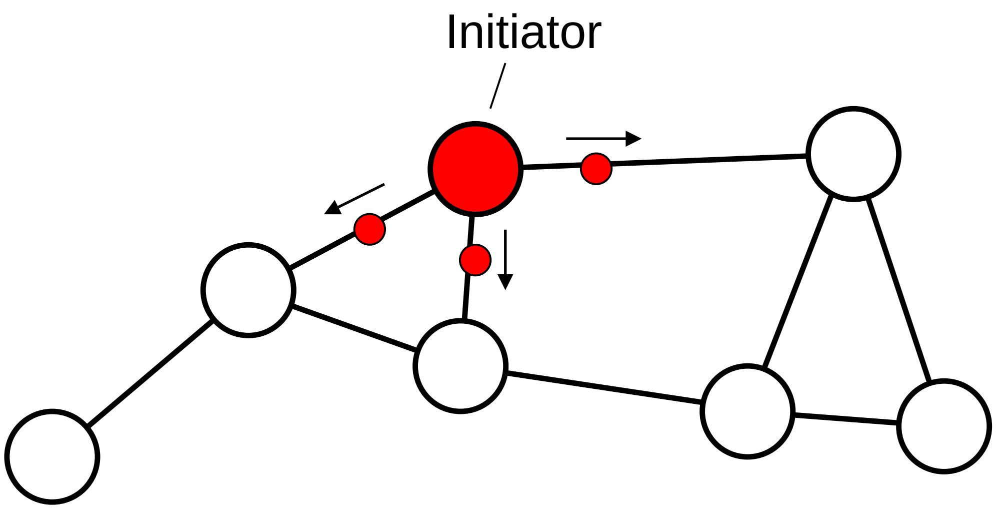
Multicast

- Distribution of information to a specific group of nodes with or without acknowledgement

Flooding

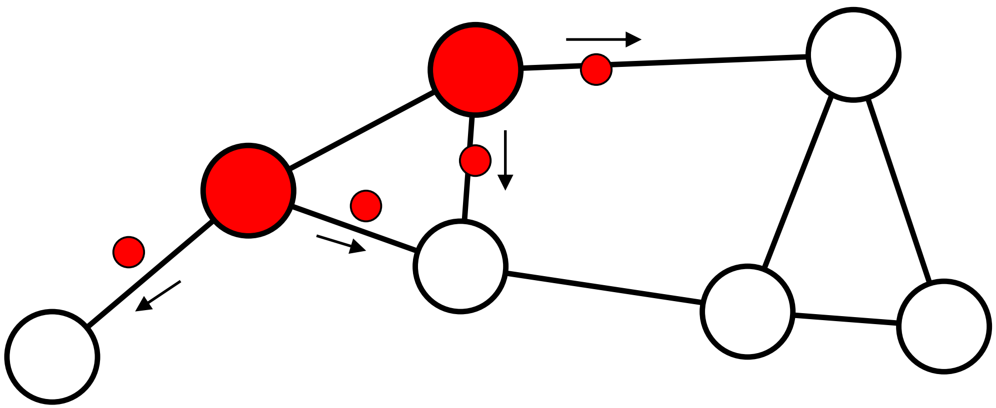
Information Distribution with Flooding

- Precondition: Connected topology
- Principle:
  Each node tells a *new* rumor to *all other* neighbors



Information Distribution with Flooding

- Every node tells a *new* rumor that it got from one of its neighbors to all other neighbors
- Already known rumors are ignored
- Step by step all nodes will be informed



Flooding-Algorithm

```
I: {NOT informed}
   SEND <info> TO all neighbors
   informed := TRUE;

R: {A message <info> is received}
   IF NOT informed THEN
        SEND <info> TO all other neighbors;
        informed := TRUE;
   FI
```

Initially, **`informed == FALSE`** for all processes
Action **`I`** is carried out by the initiator spontaneously
Are several competing initiators allowed?

Information Distribution with Flooding

How many messages are sent?

- Let *n* be the number of nodes and *e* the number of edges

Information Distribution with Flooding

How many messages are sent?

- Let n be the number of nodes and e the number of edges
- Each node sends over all his incident edges
  → $+2e$ messages
- But not back over its activation edge
  → $-n$ messages
- Exception: initiator (has no activation edge)
  → $+1$ message
- ⇒ Altogether $2e - n + 1$ messages

> How does the initiator know that all nodes were reached? → Termination detection (but how?)

Flooding with Confirmation

- Two message types: Explorers and confirmations

- A process acknowledges an explorer with a confirmation, as soon as it has received a confirmation for all explorers sent by itself due to the receiving of that explorer
  - First received explorer (activation edge):
    confirmation after arrival of #*neighbor* – 1 receipts
    → leafs send confirmation immediately
  - Further explorer: confirmation sent immediately

- Algorithm terminates, if the initiator received a confirmation from every neighbor

Flooding with Confirmation I (wrong!)

```
I: {NOT informed} // Executed by Initiator
    SEND Explorer TO all Neighbors;
    informed := TRUE;

{Explorer from neighbor N is received}
    IF NOT informed THEN
        SEND Explorer TO all Neighbors except N;
        informed := TRUE;
        A := N;
    FI

{Confirmation is received}
    Count := Count + 1;
    IF (NOT Initiator) AND (Count == #Neighbors - 1) THEN
        SEND Confirmation TO Neighbor A;
    FI
    IF Initiator AND (Count == #Neighbors) THEN
        Exit; // Algorithm is terminated.
    FI
```

Initially, **`informed == false`** and **`Count == 0`** for all nodes

Flooding with Confirmation II (right)

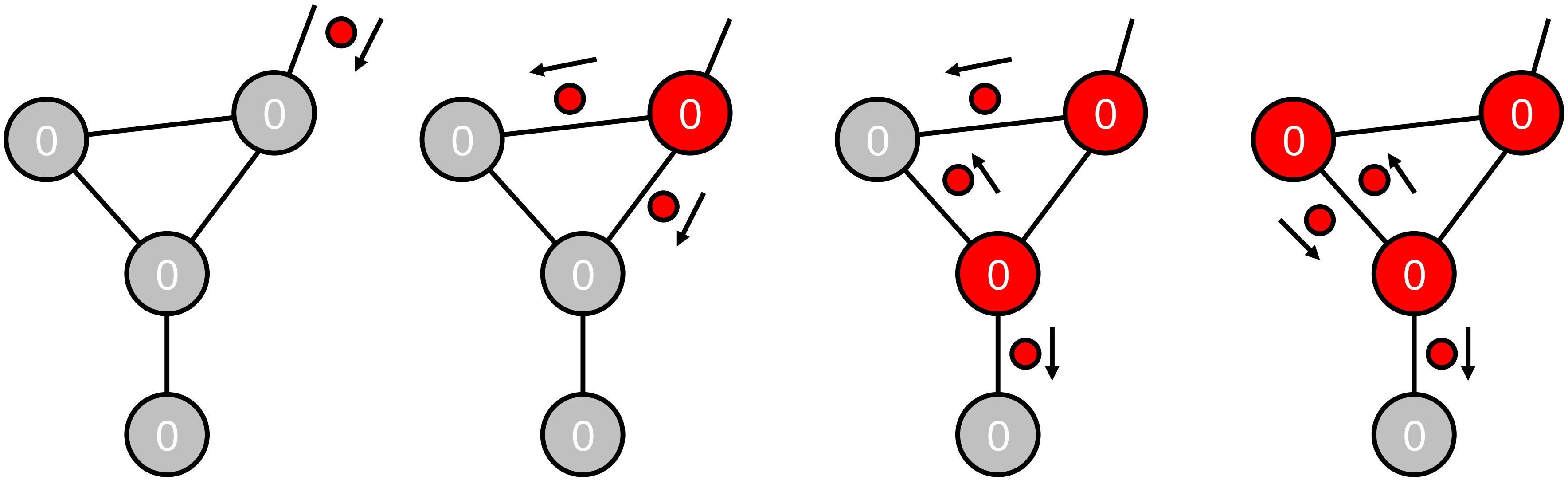
```
I: {NOT informed} // Executed by Initiator
    SEND Explorer TO all Neighbors;
    informed := TRUE;

{Explorer from neighbor N is received}
    IF NOT informed THEN
        SEND Explorer TO all Neighbors except N;
        informed := TRUE;
        A := N;
    ELSE
        SEND Confirmation TO N;
    FI

{Confirmation is received}
    Count := Count + 1;
    IF (NOT Initiator) AND (Count == #Neighbors – 1) THEN
        SEND Confirmation TO Neighbor A;
    FI
    IF Initiator AND (Count == #Neighbors) THEN
        Exit; // Algorithm is terminated.
    FI
```

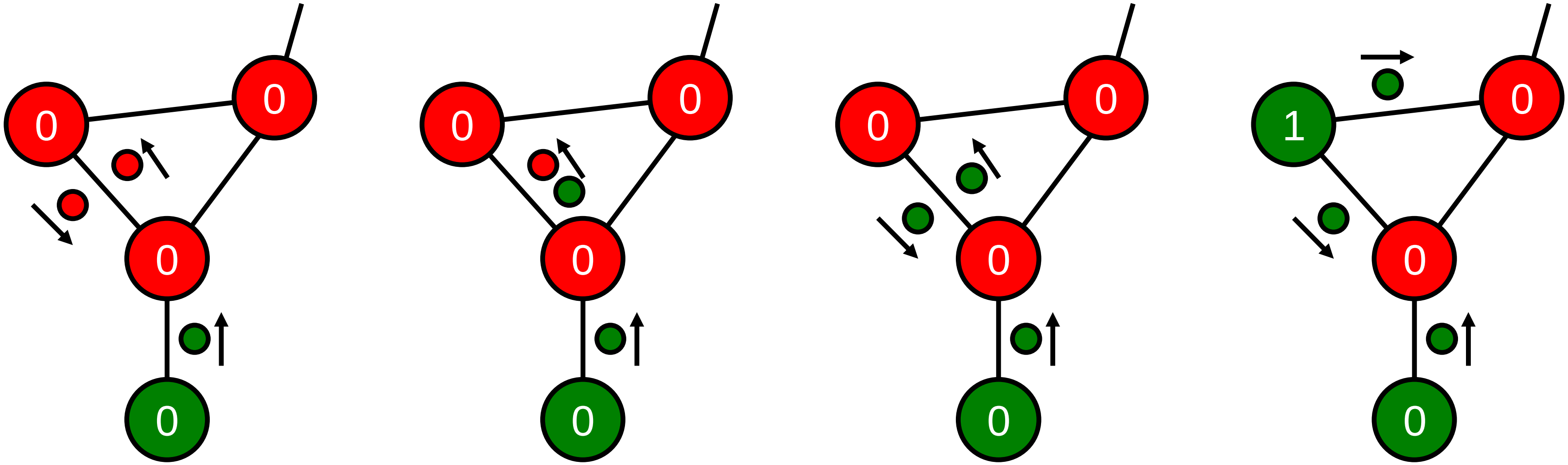
Initially, **`informed == false`** and **`Count == 0`** for all nodes

Flooding with Confirmation - Example

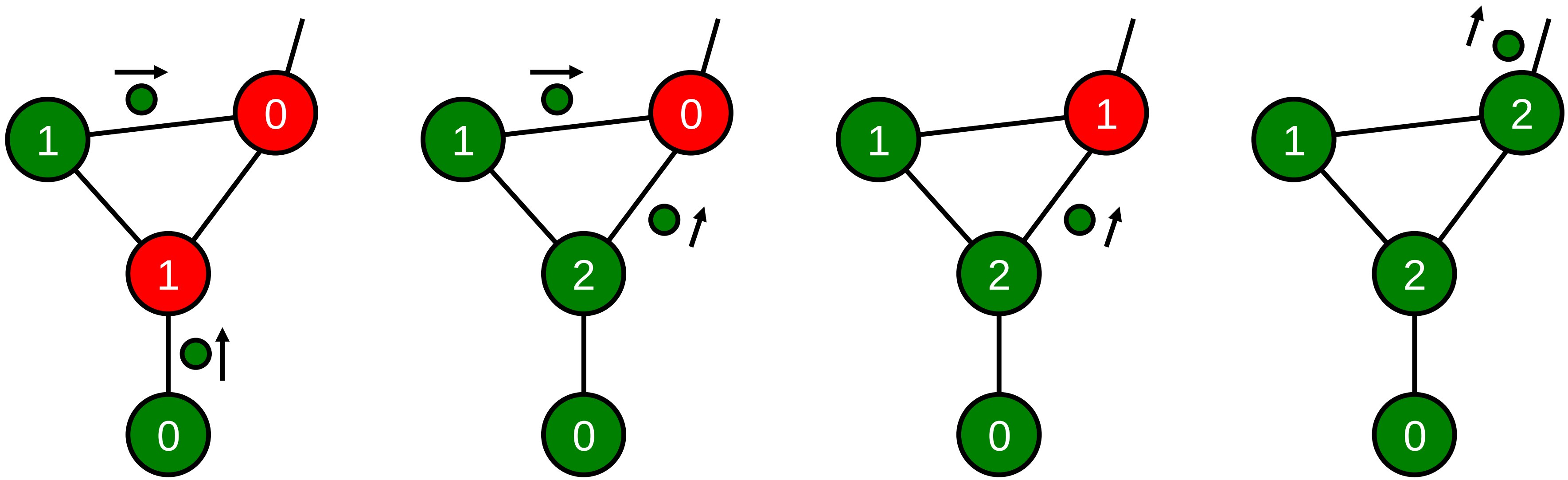


Here, the number of received confirmations is counted.

Flooding with Confirmation - Example



Flooding with Confirmation - Example



Flooding with Confirmation

How many explorers altogether?

How many confirmations altogether?

Altogether?

Flooding with Confirmation

How many explorers altogether?

- Every node sends an explorer on all edges → +2*e* explorer
- But not on its activation edge → -*n* explorer
- Exception initiator → +1 explorer
- $2e - n + 1$ explorer

How many confirmations altogether?

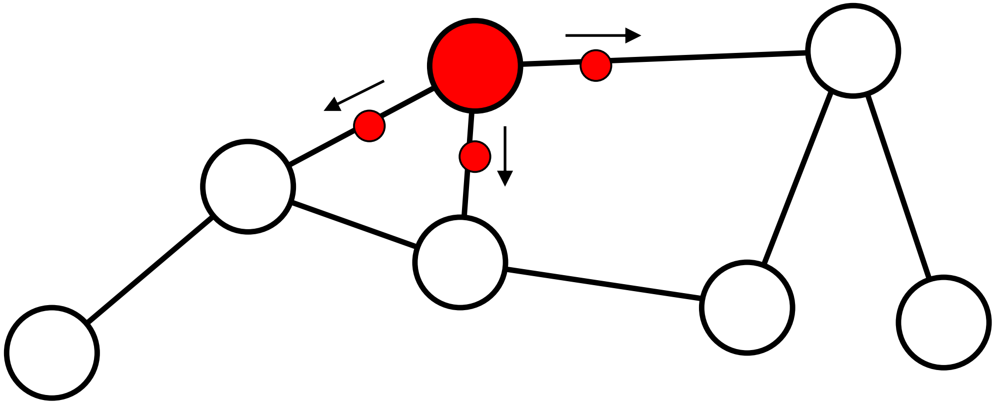
- Every node gets a confirmation on every edge → +2e messages
- But not on its activation edge → -*n* messages
- Exception initiator → +1 message
- $2e - n + 1$ confirmations

Altogether: $4e - 2n + 2$ messages, double the number for flooding without confirmation

Echo

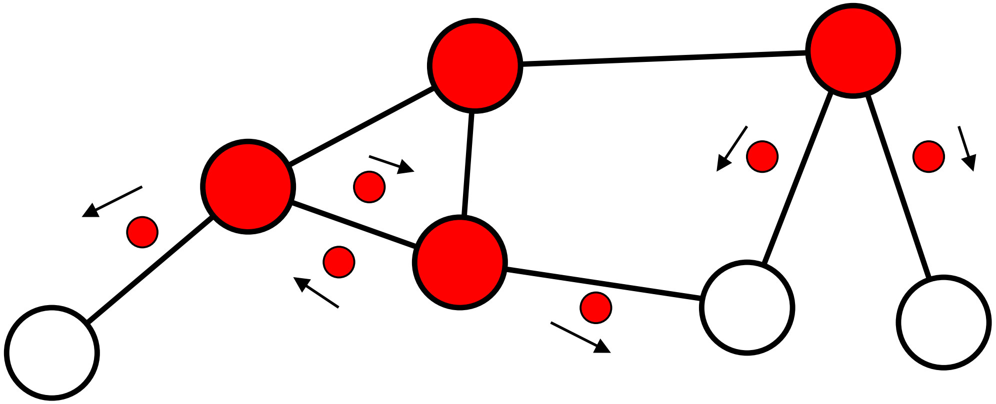
Echo-Algorithm

- Initially all nodes are *white*
- The unique initiator becomes *red* and sends red messages (explorers) to all its neighbors



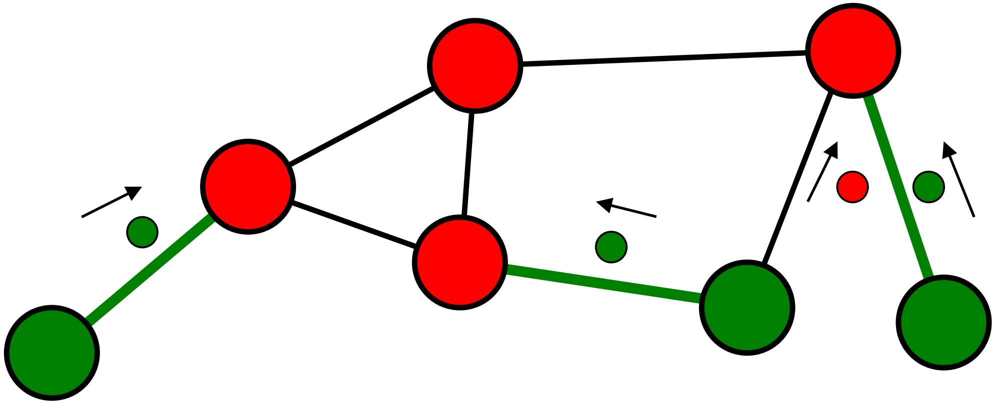
Echo-Algorithm

- A white node, receiving an explorer, becomes red itself and memorizes that „first“ edge (*activation edge*) and sends explorers to all its neighbors
- On an edge, where two explorers meet, the cycle is broken (i.e., the explorers are swallowed)



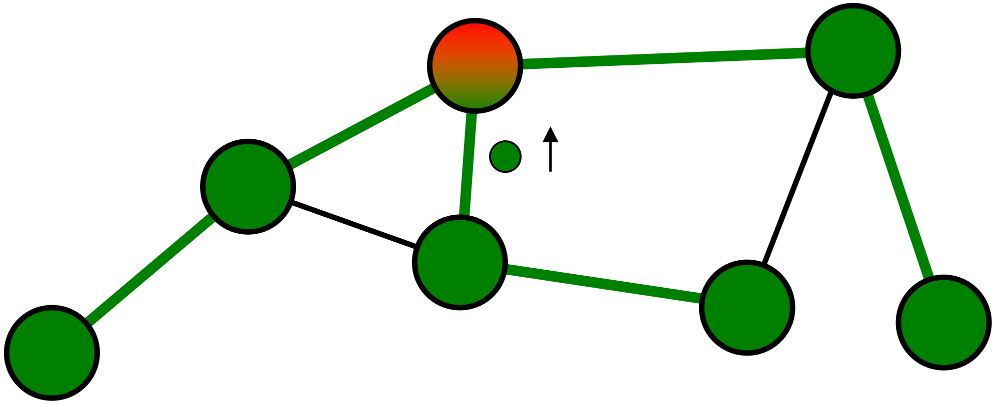
Echo-Algorithm

- A red node which has received an explorer *or* echo over *all* its edges becomes green and sends a green echo over its „first“ edge which also becomes green
- Leafs immediately send an echo when receiving an explorer



Echo-Algorithm

- By and by all nodes and a part of the edges turn green
- The algorithm terminates when the initiator turns green
- That happens when the last echo or the last explorer arrives



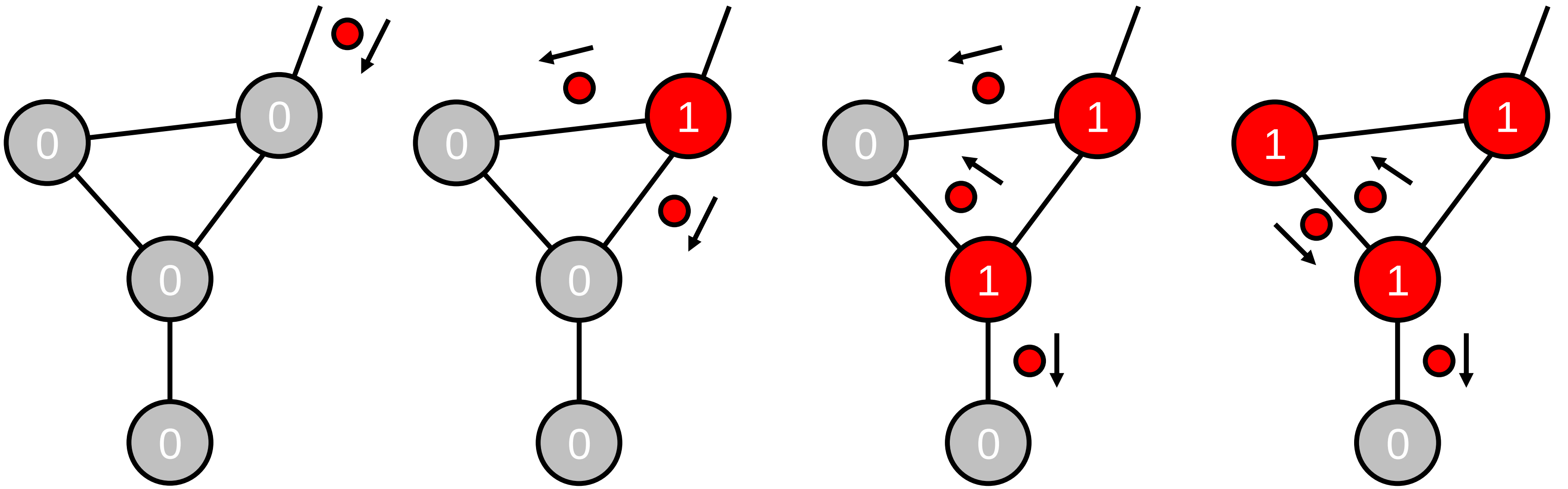
Echo-Algorithm

```
I: {NOT informed} // executed by the initiator
   SEND <Explorer> TO all Neighbors;
   informed := TRUE;

R: {a message from neighbor N is received}
   IF NOT informed THEN // Must be the first explorer
      SEND <Explorer> TO all Neighbors except N;
      informed := TRUE;
      A := N;
   FI
   Count := Count + 1;
   IF Count == #Neighbors THEN
      IF NOT Initiator THEN
         SEND <Echo> TO Neighbor A;
      ELSE
         EXIT; // Algorithm has terminated
      FI
   FI
```

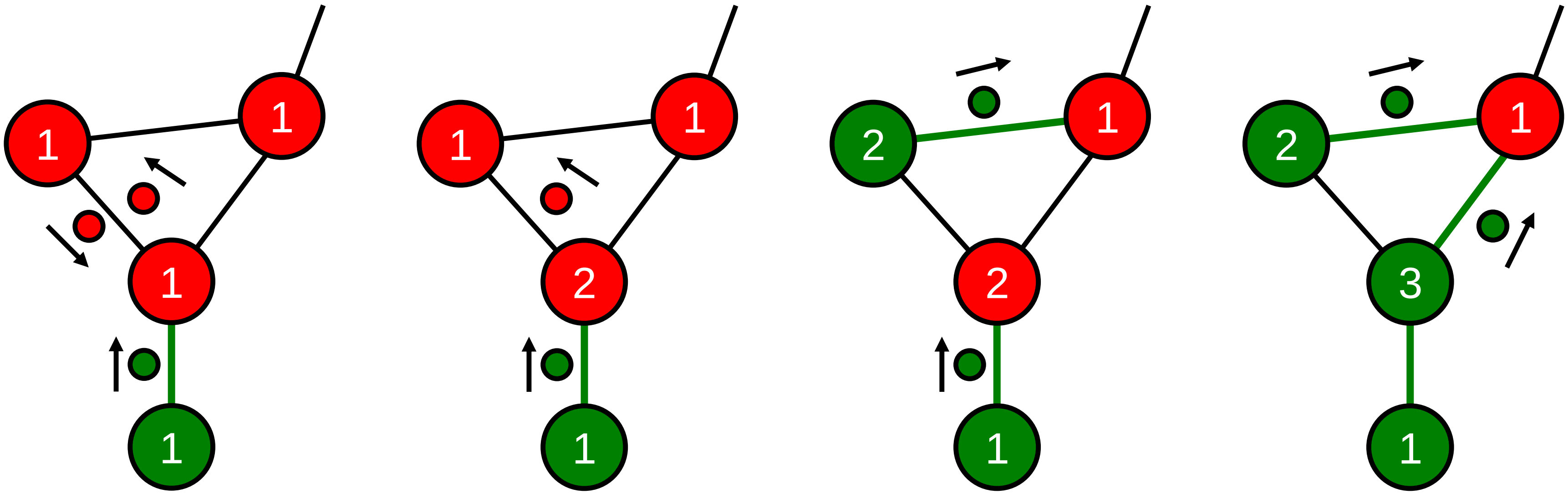
Initially, `informed == false` and `Count == 0` for all nodes

Echo-Algorithm – Example

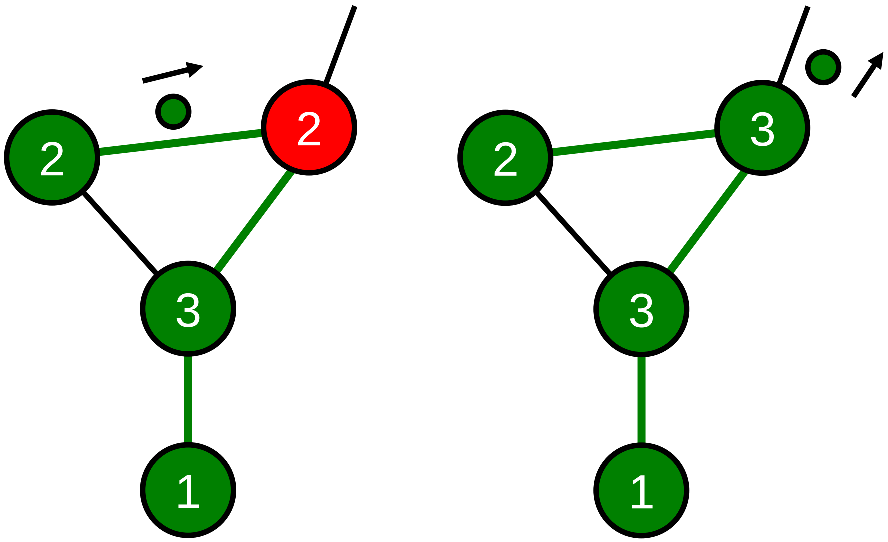


Here, the number of already received explorers *and* echos is counted.

Echo-Algorithm – Example



Echo-Algorithm – Example



Echo-Algorithm – Characteristics

Exactly two messages run over every edge

- Either an explorer and an echo running in the opposite direction or two explorers running in opposite directions

Parallel traversing of a (connected non-directional) graph with 2*e* messages

- Every node sends an explorer on all edges → +2*e* explorer
- Exception activation edge → -*n* explorer
- Exception initiator → +1 explorer
- Every node sends an echo on the activation edge
  → +*n* echos
- Exception initiator → -1 echo

Echo-Algorithm – Characteristics

The Echo-algorithm is a wave algorithm

Forth wave: becoming red

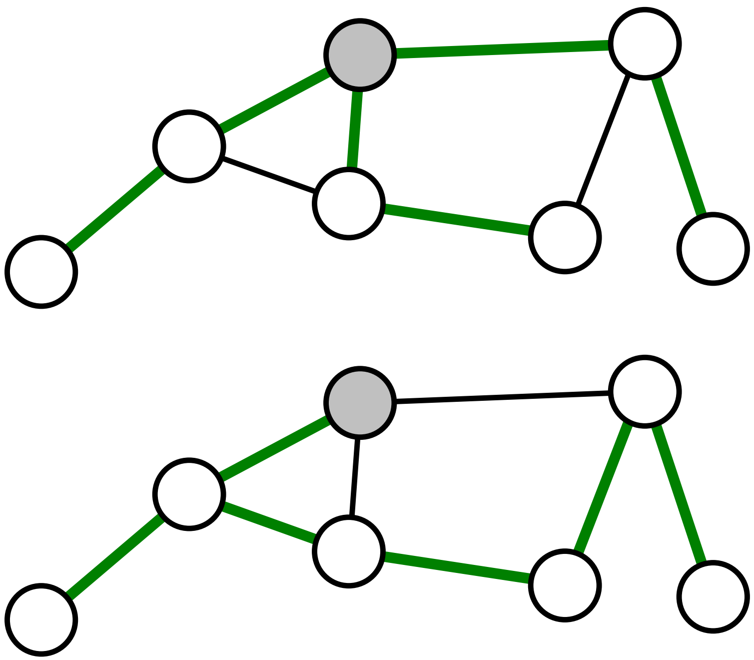
- Distribution of information (to all nodes over all edges)

Back wave: becoming green

- Collecting of information
  (of potentially all nodes over the activation edges)

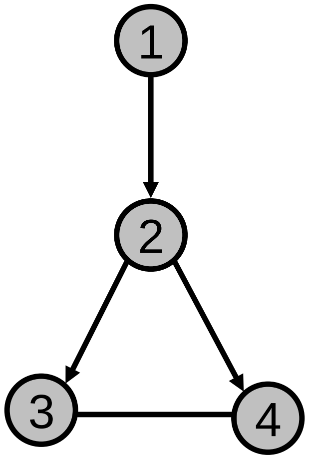
Echo-Algorithm – Characteristics

- Echo-edges form a spanning tree
- Depending on the message delays, the spanning tree looks differently because fast edges are preferred



Improvement of the Echo-Algorithm?

- Idea: Avoid the visit of nodes which are known to be visited by other explorers
- Together with an explorer, a set of taboo nodes z is sent and received
- The sent taboo set by the initiator is
  $z = \langle neighbors\ of\ initiator \rangle \cup \langle initiator \rangle$
- Explorers only sent to the set of neighbors y which are not in z.
- Thus, the new taboo set $z' = z \cup y$ is attached
- Advantage: Saving of messages
  - Extreme cases: tree and complete graph
- Disadvantages:
  - message length $O(n)$
  - identity of neighbors has to be known



E.g. the message of 2 and 3 contains the info that 4 does not have to be visited.

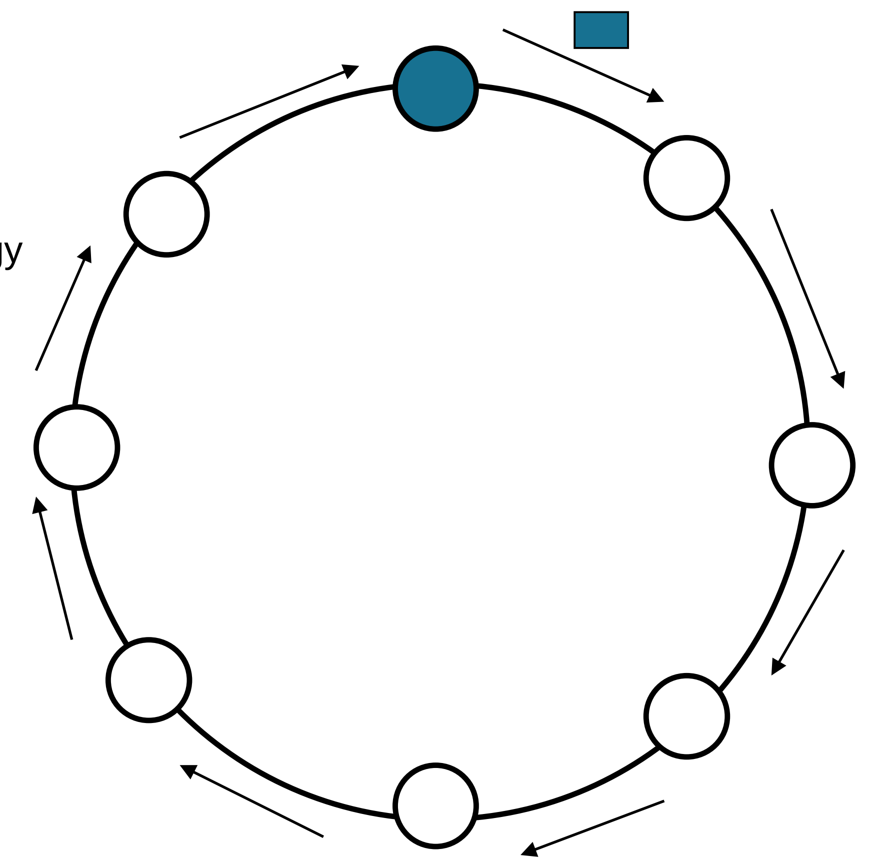
Broadcast

Broadcast on Special Topologies

- Broadcast: Sending of a message to all nodes, optionally also with confirmation
- Flooding realizes a broadcast on arbitrary connected undirected topologies
  - Especially fault-tolerant because all edges are used for the distribution of information
- For special topologies, a broadcast with less messages is possible, provided the algorithm knows which topology is underlying
  - Less error-tolerant because, in the aimed case, each node is only reached over a single edge → $n - 1$ messages
- Exemplary topologies: Rings, trees, hypercubes

Broadcast on Unidirectional Rings

- Token circulates with message
- All nodes are informed, if the token reaches the initiator again
- *n* messages
- A ring can also be overlaid by another topology
  → logical ring



Broadcast on Trees

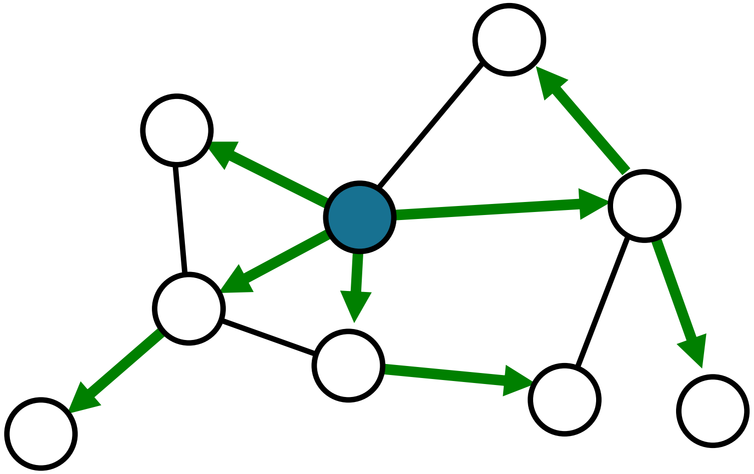
Tree has $n - 1$ edges

One message goes over each edge

For the confirmation (if required) one additional message goes over every edge

Tree can be overlaid by another topology

→ Spanning tree



Broadcast on Hypercubes?

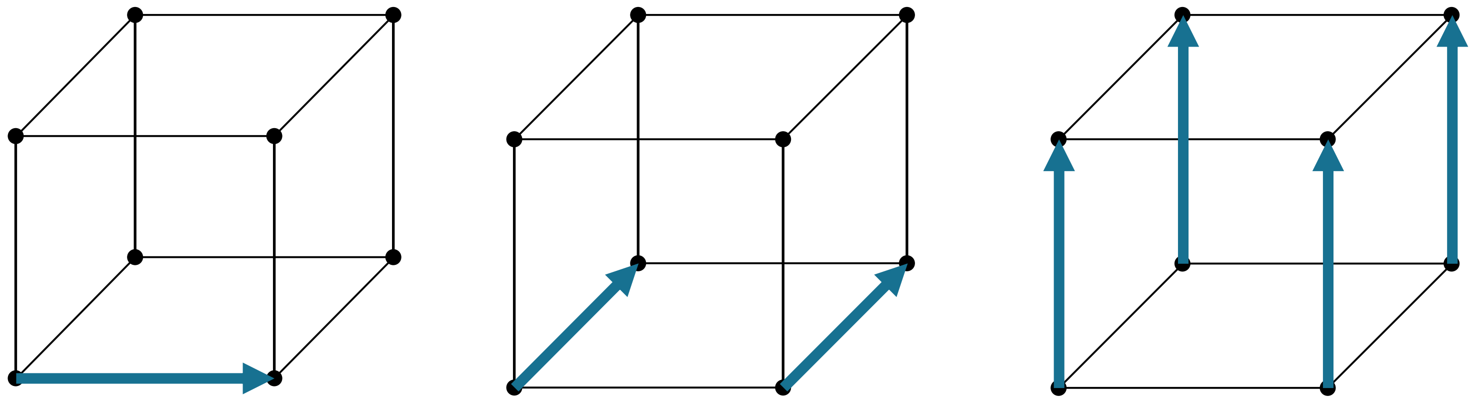
Initiator may have number 00...00 (binary)

Broadcast on Hypercubes

Initiator may have number 00...00 (binary)

Analogous to recursive construction of a hypercube

- Initiator sends in dimension 1
- Then all nodes of dimension 1 in dimension 2
- Then all nodes of dimension 2 in dimension 3
- ...



Broadcast on Hypercubes

Unit Time Complexity

- After *d* cycles all nodes are informed
- Is that optimal?

Message complexity

- $1 + 2 + 4 + ... + 2^{d-1} = 2^d - 1 = n - 1$
- Is that optimal?

Multicast

Pairwise exchange of messages is not optimal if communication takes place between a sending process and a group of receiving processes

- A Multicast Operation sends a single message from one process to each member of a group of processes
  - Membership is usually transparent to the sender

Multicast - Motivation

Multicast operations provide a useful infrastructure for several Distributed Systems problems:

- Fault tolerance based on replicated services:
  - Client requests are multicast to a group of identical, replicated servers -> clients are served even if some of the servers fail
- Service Discovery:
  - Multicast messages can be used to locate available Discovery Services and look up particular services
- Better performance through replicated data:
  - If replicated data changes, multicast messages can be used to propagate changes to all copies
- Event notifications:
  - Notify a group of processes interested in certain events that a new event happened

Multicast – IP Multicast

IP Multicast provides a multicast infrastructure on top of IP

- A multicast group is specified by a Class D IP-Address
  - First 4 bits are specified as 1110 (e.g. 224.0.1.1)
  - For administrative purposes, the address space is divided into blocks, examples:
    - Local Network Control Block (224.0.0.0 to 224.0.0.225)
    - Internet Control Block (224.0.1.0 to 224.0.1.225) – e.g. Network Time Protocol NTP (224.0.1.1)
  - Permanent and temporary addresses exist (e.g. NTP is permanent)
  - In order to create a multicast group, a free multicast-address is required (further reading!)
    - Nodes send leave and join messages (coordination based on Internet Group Management Protocol - IGMP)

Multicast – IP Multicast

IP Multicast provides following failure model:

- Messages send via multicast use datagrams (no session!) -> same failure characteristics as UDP datagrams
  - Omission failures are possible
  - In contrast to pointwise communication, some but not all members of a multicast group may receive messages
  - ➢ IP Multicast can be called unreliable multicast
  - What means reliable multicast?

Multicast – Reliable IP Multicast

Based on the definition of reliable communication, reliable multicast must satisfy following properties

- Note that we use the term deliver instead of receive

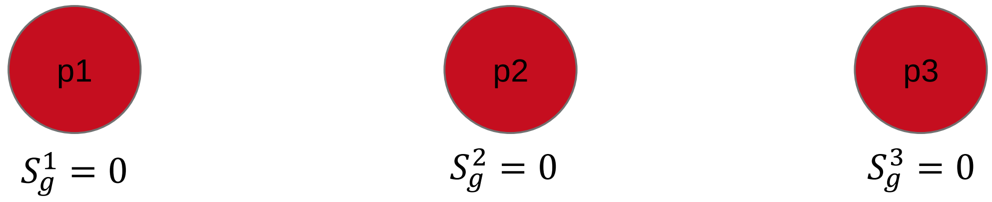
- Integrity: A correct process p delivers a message m at most once. Furthermore, p must belong to the multicast-group m was send to.
- Validity: If a correct process multicasts message m, then it will eventually deliver m.
- Agreement: If a correct process delivers m, then all other correct processes in the same multicast-group will eventually deliver m.

Multicast – Reliable IP Multicast

Reliable IP-Multicast can be implemented using piggybacked acknowledgements and negative acknowledgements

- piggybacked ack: acknowledgement of delivery is attached to other message
- Negative acknowledgement: notify other process that a message is missing

- We assume only one group g
- Each process maintains a sequence number S_g^p
  - Initally set to zero
  - Each process increments its sequence number when it multicasts a message



Multicast – Reliable IP Multicast

Reliable IP-Multicast can be implemented using piggybacked acknowledgements and negative acknowledgements

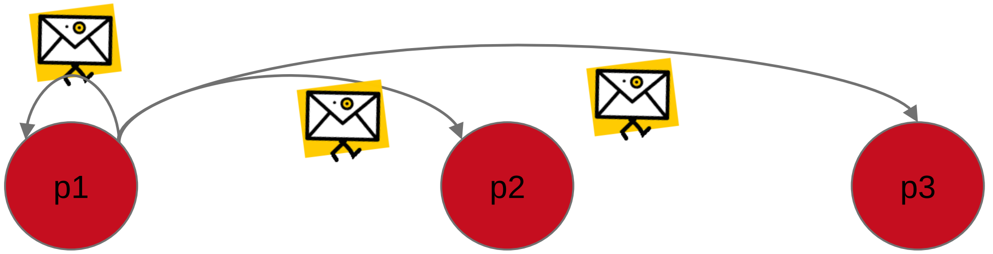
- Additionally, each process records the sequence number it has delivered from other processes: R_g^q
- For the example, we assume that p2 and p3 already have multicast a message

p1	p2	p3
$S_g^1 = 0$	$S_g^2 = 1$	$S_g^3 = 1$
$R_g^1 = 0$	$R_g^1 = 0$	$R_g^1 = 0$
$R_g^2 = 1$	$R_g^2 = 1$	$R_g^2 = 1$
$R_g^3 = 1$	$R_g^3 = 1$	$R_g^3 = 1$

Multicast – Reliable IP Multicast

Reliable IP-Multicast can be implemented using piggybacked acknowledgements and negative acknowledgements

- When p1 multicasts a message m, it piggybacks its sequence number S_g^1 and a set of acknowledgements of the form $< q, R_g^q >$
- E.g.: m={payload, 0, <2, 1>, <3, 1>}
- The acknowledgement states the sequence number of the latest message p1 has delivered from q since it last multicast a message

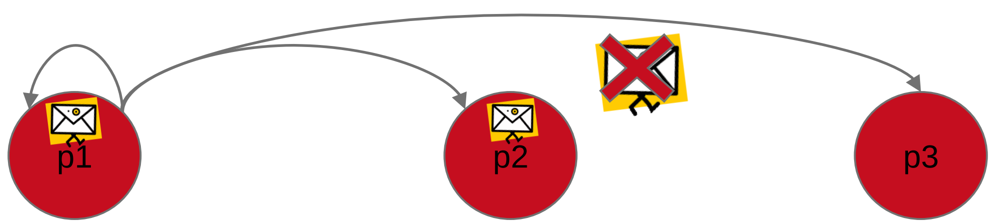


p1	p2	p3
$S_g^1 = 1$	$S_g^2 = 1$	$S_g^3 = 1$
$R_g^1 = 0$	$R_g^1 = 0$	$R_g^1 = 0$
$R_g^2 = 1$	$R_g^2 = 1$	$R_g^2 = 1$
$R_g^3 = 1$	$R_g^3 = 1$	$R_g^3 = 1$

Multicast – Reliable IP Multicast

Reliable IP-Multicast can be implemented using piggybacked acknowledgements and negative acknowledgements

- Assume p1 and p2 deliver the message
- p3 does not due to loss or delay of the message, however, p1 send a new message m={payload, 1, <1, 1>, <2, 1>, <3, 1>}

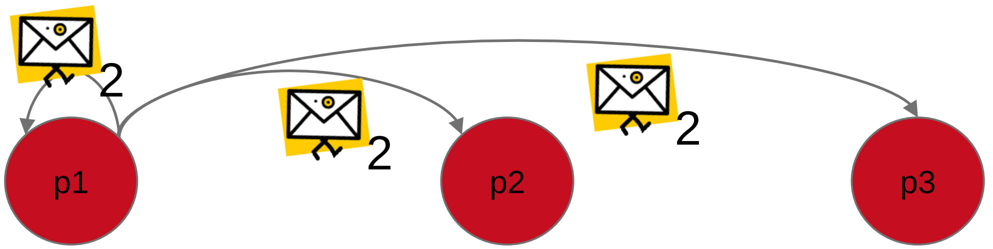


p1	p2	p3
$S_g^1 = 1$	$S_g^2 = 1$	$S_g^3 = 1$
$R_g^1 = 1$	$R_g^1 = 1$	$R_g^1 = 0$
$R_g^2 = 1$	$R_g^2 = 1$	$R_g^2 = 1$
$R_g^3 = 1$	$R_g^3 = 1$	$R_g^3 = 1$

Multicast – Reliable IP Multicast

Reliable IP-Multicast can be implemented using piggybacked acknowledgements and negative acknowledgements

- On receipt of the new message, p3 compares the piggybacked sequence number with R_g^1
  - The message will only be delivered if S == R_g^1 (R will be incremented in this case)
  - If any process sees a message with S ≤ R_g^q, the message has been seen before and is discarded



$S_g^1 = 1$
$R_g^1 = 1$
$R_g^2 = 1$
$R_g^3 = 1$

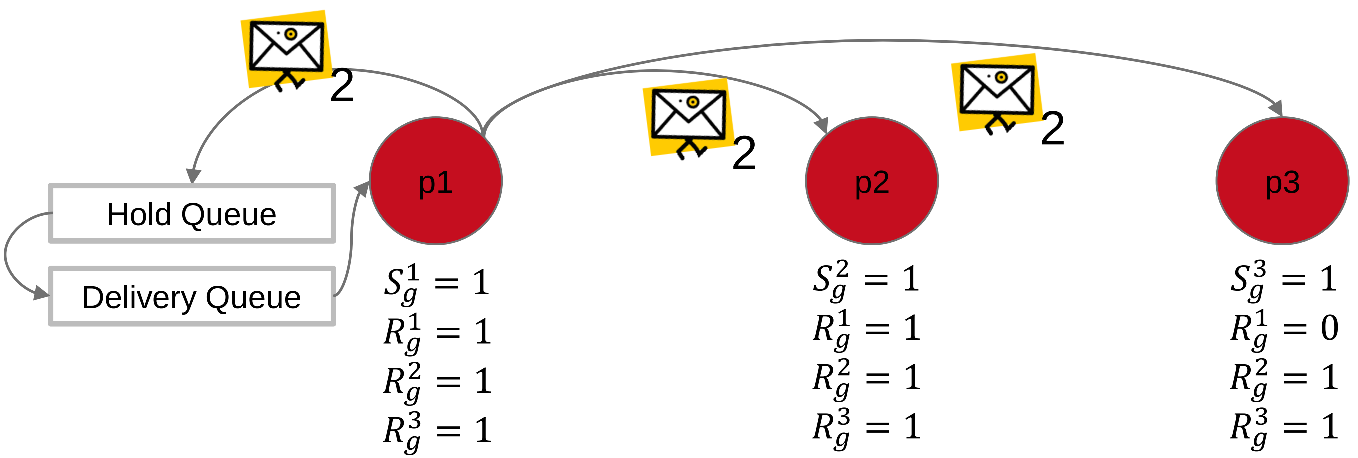
$S_g^2 = 1$
$R_g^1 = 1$
$R_g^2 = 1$
$R_g^3 = 1$

$S_g^3 = 1$
$R_g^1 = 0$
$R_g^2 = 1$
$R_g^3 = 1$

Multicast – Reliable IP Multicast

Reliable IP-Multicast can be implemented using piggybacked acknowledgements and negative acknowledgements

- On receipt of the new message, p3 compares the piggybacked sequence number with R_g^1
  - If S > R_g^q, the message is stored in a hold-back queue and missing messages are requested before delivery
  - A negative acknowledgment is sent to either the sender or any other process that has already acknowledged the delivery of a missing message



Literature

E. Chang. *Echo algorithms: Depth parallel operations on graphs*. IEEE Transactions on Software Engineering, 8(4):391--400, 1982.