# SCRIBE

## A Large-Scale and Decentralised Application-Level Multicast Infrastructure

---

João Nogueira

Tecnologias de Middleware
DI - FCUL - 2006

# Agenda

- Motivation
- Pastry
- Scribe
- Scribe Protocol
- Experimental Evaluation
- Conclusions

# Motivation

- Network-level IP multicast was proposed over a decade ago
  - Some protocols have added reliability to it (e.g. SRM, RMTP)
- However, the use of multicast in real applications has been limited because of the lack of wide scale deployment and the issue of how to track membership
  - As a result, application-level multicast has gained in popularity
- Algorithms and systems for scalable group management and scalable, reliable propagation of messages are still active research areas:
  - For such systems, the challenge remains to build an infrastructure that can scale to, and tolerate the failure modes of, the general Internet, while achieving low delay and effective use of network resources

# Overview

- Scribe is a large-scale, decentralised application-level multicast infrastructure built upon *Pastry*
    - *Pastry* is a scalable, self-organising peer-to-peer location and routing substrate with good locality properties
- Scribe provides efficient application-level multicast and is capable of scaling to a large number of groups, of multicast sources and of members per group
- Scribe and Pastry adopt a fully decentralised peer-to-peer model, where each participating node has equal responsibilities
    - Scribe builds a multicast tree, formed by joining the *Pastry* routes from each member to a *rendezvous point* associated with the group
    - Membership maintenance and message dissemination in Scribe leverage the robustness, self-organisation, locality and reliability properties of *Pastry*

# Pastry

- Pastry is a peer-to-peer location and routing substrate
  - Forms a robust, self-organising overlay network in the Internet
  - Any Internet-connected host that runs the Pastry software and has proper credentials can participate in the overlay network
- Each Pastry node has a unique 128-bit identifier: *nodeID*
- The set of existing *nodeID's* is uniformly distributed
- Given a message and a key, Pastry reliably routes the message to the Pastry live node with *nodeID* numerically closest to the key

# Pastry
# Complexity

- On a network of *N* nodes, Pastry can route to any node in less than $log_2^b N$ *steps on average*
  - *b* is a configuration parameter with typical value 4

- With concurrent node failures, eventual delivery is guaranteed unless *l/2* or more adjacent nodes fail simultaneously
  - *l* is a configuration parameter with typical value 16

- The tables required in each Pastry node have $(2^b - 1) * log_2^b N + l$ entries
  - Each entry maps a *nodeID* to the associated node's IP address

- After a node failure or the arrival of a new node, the invariants in all routing tables can be restored by exchanging $O( log_2^b N )$ messages

# Pastry
# Routing

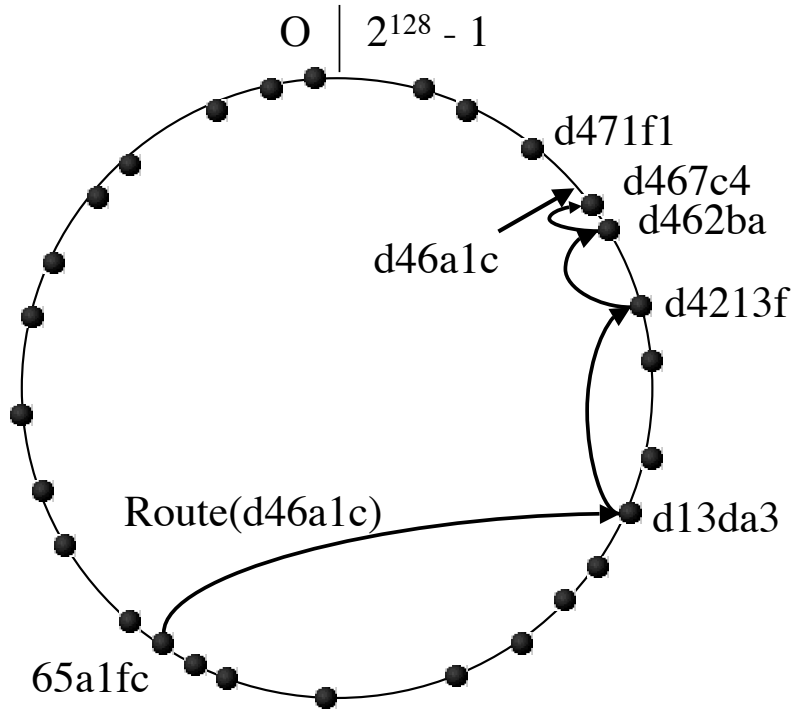
- For the purpose of routing, *nodeID's* and keys are thought of as a sequence of digits with base $2^b$

- A node's routing table is organised into $log_{2^b} N$ rows with $2^b-1$ entries each

- In addition to the routing table, each node maintains IP addresses for the nodes in its leaf-set, i.e. the nodes with the $l/2$ numerically closest larger *nodeID's* and the ones with the $l/2$ numerically closest smaller *nodeID's*.

# Pastry
# Routing Table Example

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *0* | *1* | *2* | *3* | *4* | *5* | | *7* | *8* | *9* | *a* | *b* | *c* | *d* | *e* | *f* |
| *x* | *x* | *x* | *x* | *x* | *x* | | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* |
| | | | | | | | | | | | | | | | |
| *6* | *6* | *6* | *6* | *6* | | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* |
| *0* | *1* | *2* | *3* | *4* | | *6* | *7* | *8* | *9* | *a* | *b* | *c* | *d* | *e* | *f* |
| *x* | *x* | *x* | *x* | *x* | | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* |
| | | | | | | | | | | | | | | | |
| *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | | *6* | *6* | *6* | *6* | *6* |
| *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | | *5* | *5* | *5* | *5* | *5* |
| *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | | *b* | *c* | *d* | *e* | *f* |
| *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | | *x* | *x* | *x* | *x* | *x* |
| | | | | | | | | | | | | | | | |
| *6* | | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* | *6* |
| *5* | | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* | *5* |
| *a* | | *a* | *a* | *a* | *a* | *a* | *a* | *a* | *a* | *a* | *a* | *a* | *a* | *a* | *a* |
| *0* | | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *a* | *b* | *c* | *d* | *e* | *f* |
| *x* | | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* | *x* |



# Pastry
# API

- In a simplified manner, Pastry exports the following API to applications:
  - *nodeID* = pastryInit( *credentials* ): causes the local node to join an existing Pastry network (or start a new one) and initialise all relevant state
  - route( *msg, key* ): causes Pastry to route the given message to the live node with *nodeID* numerically closest to *key* using the overlay network
  - send( *msg, IP-addr* ): causes Pastry to send the message directly to the node with the specified IP address; if the node is alive, the message is delivered using the deliver operation

# Pastry
# API

- deliver( *msg*, *key* ): called by Pastry when a message is received that was either sent using the route or send operations

- forward( *msg*, *key*, *nextID* ): called by Pastry just before a message is forwarded to the node with *nodeID* = *nextID*; forwarding is started by a route operation; the application may change the content of the message and/or set *nextID* to null to terminate the message routing at the local node

- newLeafs( *leafSet* ): called by Pastry whenever there is a change in the leaf-set; this provides the application with an opportunity to adjust application-specific invariants based on the leaf-set

# Scribe

- Scribe is a scalable application-level multicast infrastructure built on top of *Pastry*
- Consists of a network of *Pastry* nodes, where each node run the Scribe application software
- Any Scribe node may **create** a group:
  - Other nodes may **join** the group or **multicast** messages to all members of the group (provided they have the appropriate credentials)
- Scribe ensures only best-effort delivery of multicast messages and specifies no particular delivery order
- Groups may have multiple sources of multicast messages and many members

# Scribe API

- Scribe offers a simple API to its applications:
  - create( *credentials*, *groupID* ): creates a group with identifier *groupID*; the *credentials* are used for access control
  - join( *credentials*, *groupID*, *msgHandler* ): causes the local node to join the group *groupID*; all subsequently received multicast messages for that group are passed to *msgHandler*
  - leave( *credentials*, *groupID* ): causes the local node to leave group *groupID*
  - multicast( *credentials*, *groupID*, *msg* ): causes the message *msg* to be multicast within the group with identifier *groupID*

# Scribe Protocol

- ***forward*** is invoked by *Pastry* immediately before a message is **forwarded** to the next node (with *nodeID* = *nextID*)

```
(1) forward( msg, key, nextID )
(2)    switch( msg.type )
(3)        JOIN:                    if !(msg.group C groups)
(4)                                     groups = groups U msg.group;
(5)                                     route( msg, msg.group );
(6)                                 groups[msg.group].children U msg.source;
(7)                                 nextID = null; // Stop routing the original message
```

# Scribe Protocol

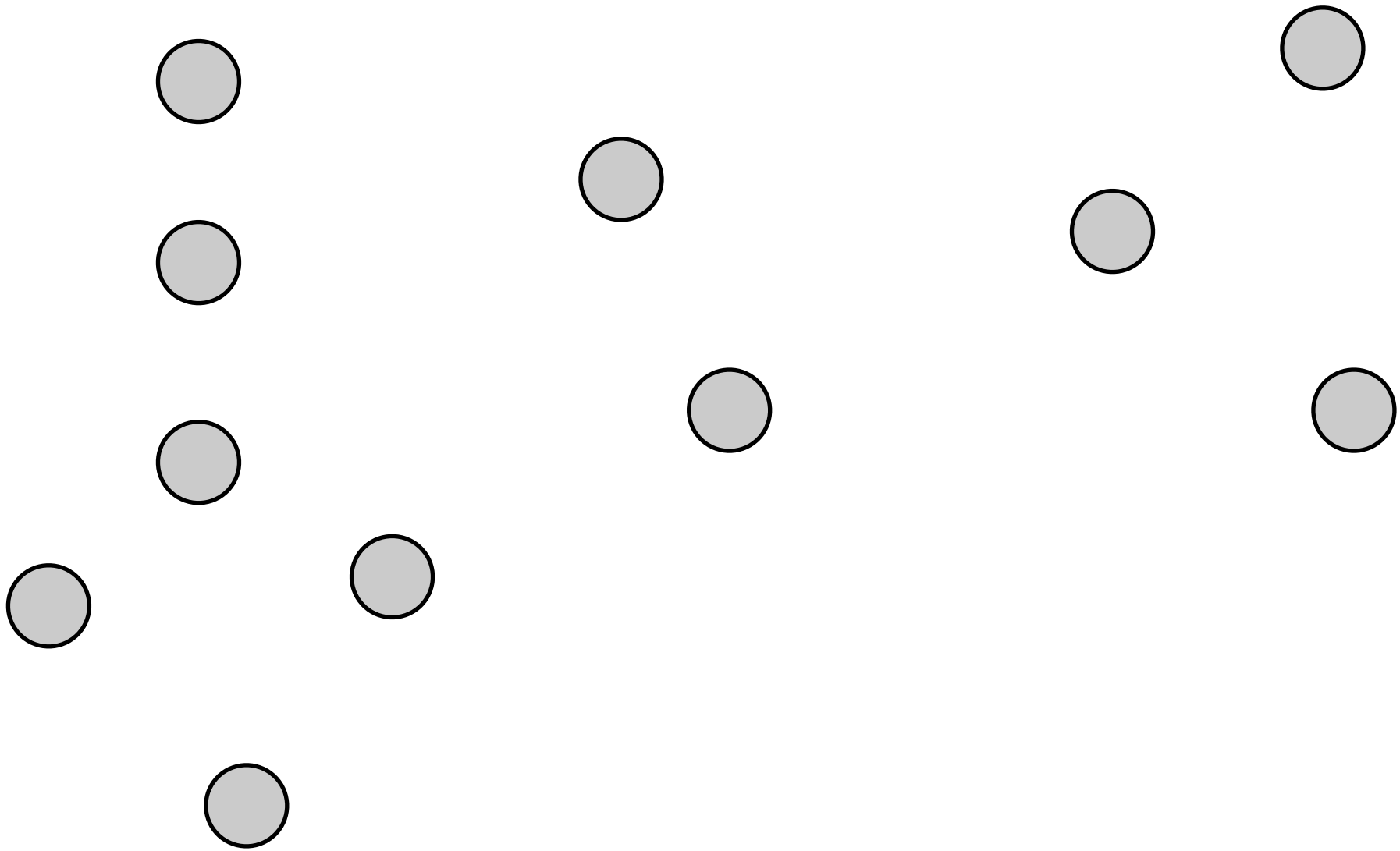
- ***deliver*** is invoked by *Pastry* when a message is received and the local node's *nodeID* is numerically closest to the key among all live nodes **or** when a message that was transmitted via send is received

```
(1) deliver( msg, key )
(2)    switch( msg.type )
(3)        CREATE:              groups = groups U msg.group;
(4)        JOIN:                groups[msg.group].children U msg.source;
(5)        MULTICAST:           V node in groups[msg.group].children
(6)                                 send( msg, node );
(7)                             if memberOf( msg.group )
(8)                                 invokeMessageHandler( msg.group, msg );
(9)        LEAVE:               groups[msg.group].children / msg.source;
(10)                            if ( |groups[msg.group].children| == 0 )
(11)                                send( msg, groups[msg.group].parent );
```

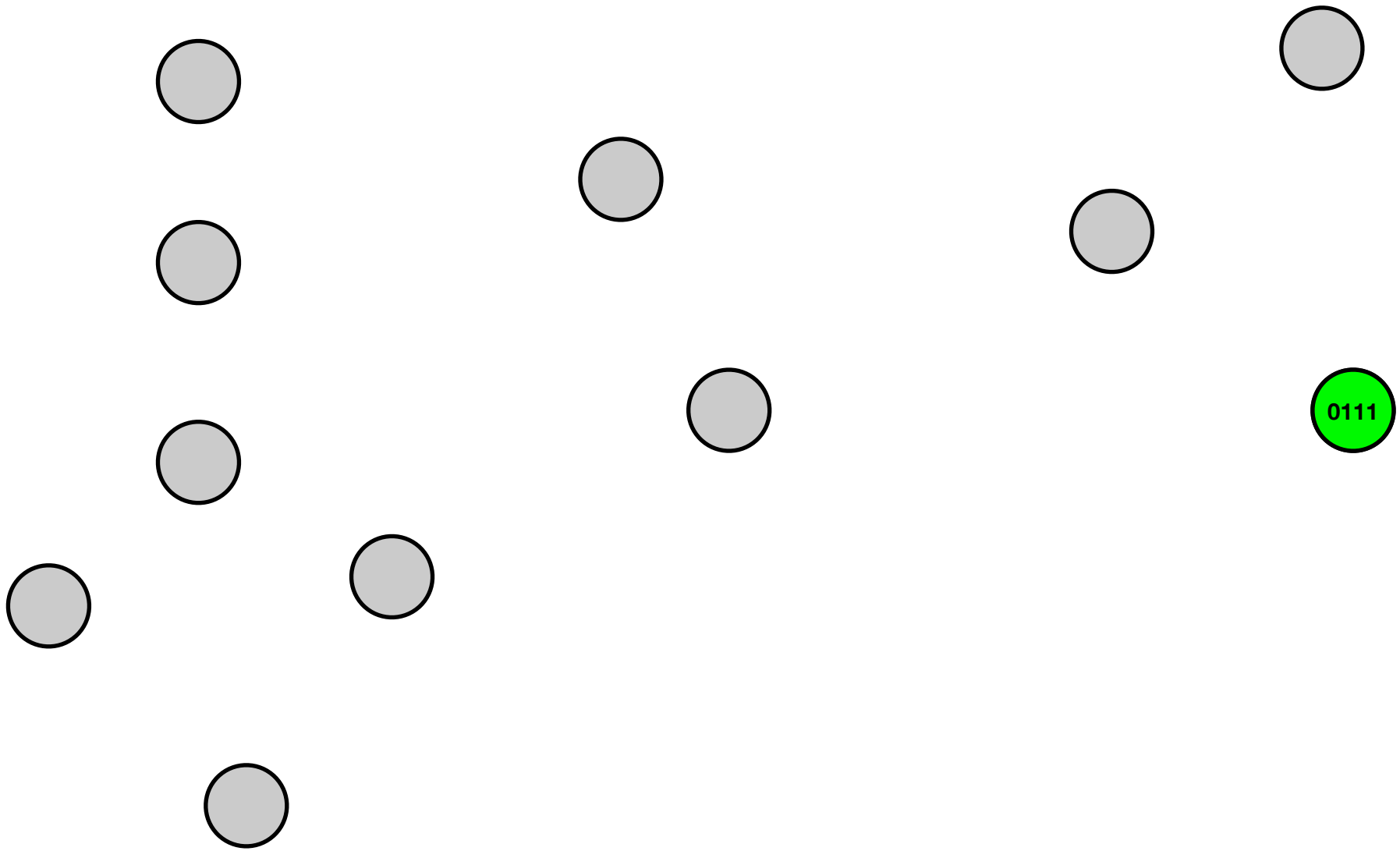
# Scribe Protocol
## Group Management

- Each group has an unique identifier: *groupID*
- The Scribe node with the numerically closest *nodeID* to *groupID* is that group's *rendezvous point*
  - The *rendezvous point* is the root of the group's multicast tree
- *groupID* is the hash of the group's name concatenated with the creator's name:
  - A collision-resistant hash function that guarantees an even distribution of *groupID*'s (e.g. *SHA-1*) is used to compute the identifiers
  - Since *Pastry's nodeID's* are also uniformly distributed, this ensures an even distribution of groups across *Pastry* nodes

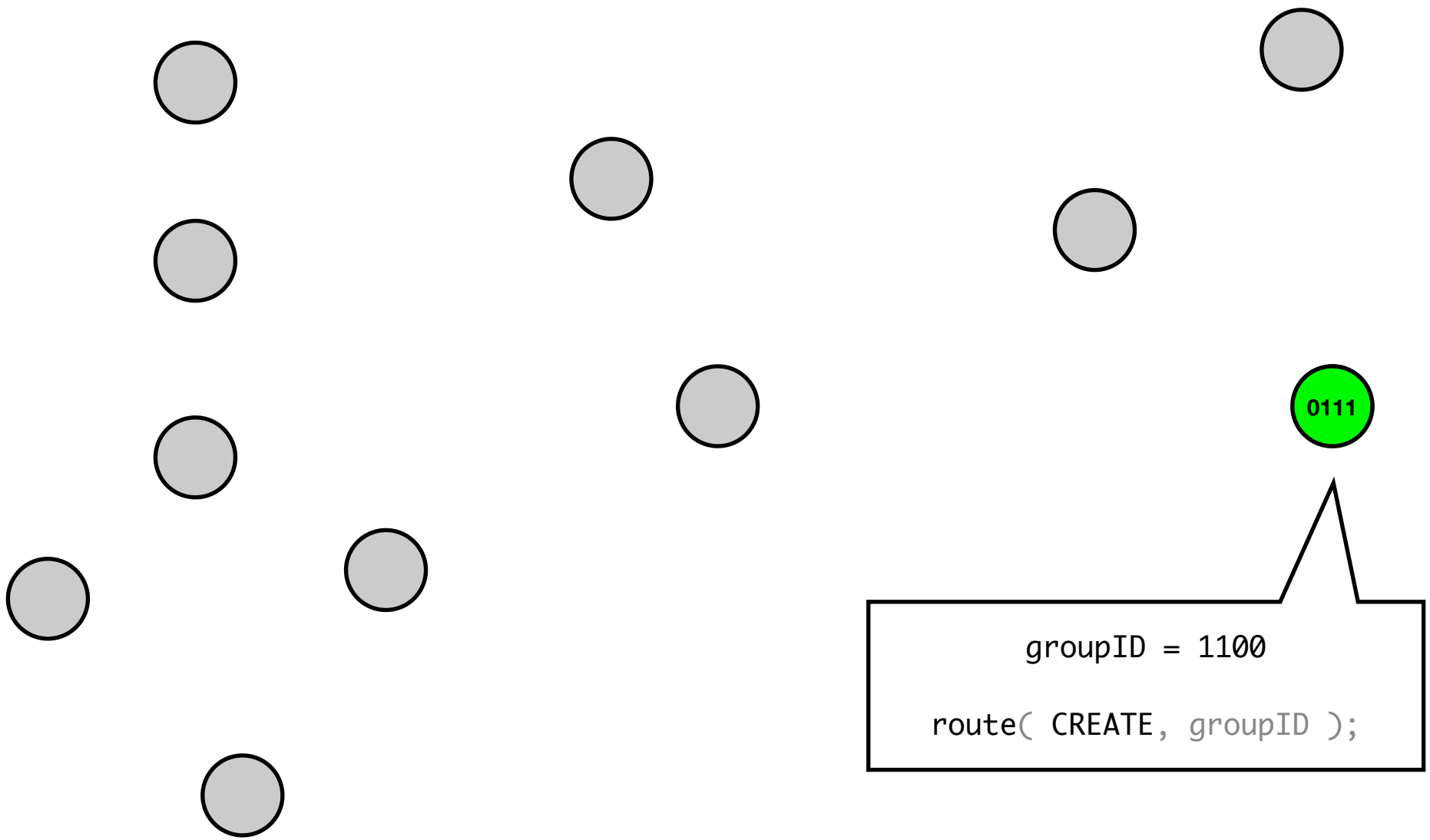
# Criar um grupo



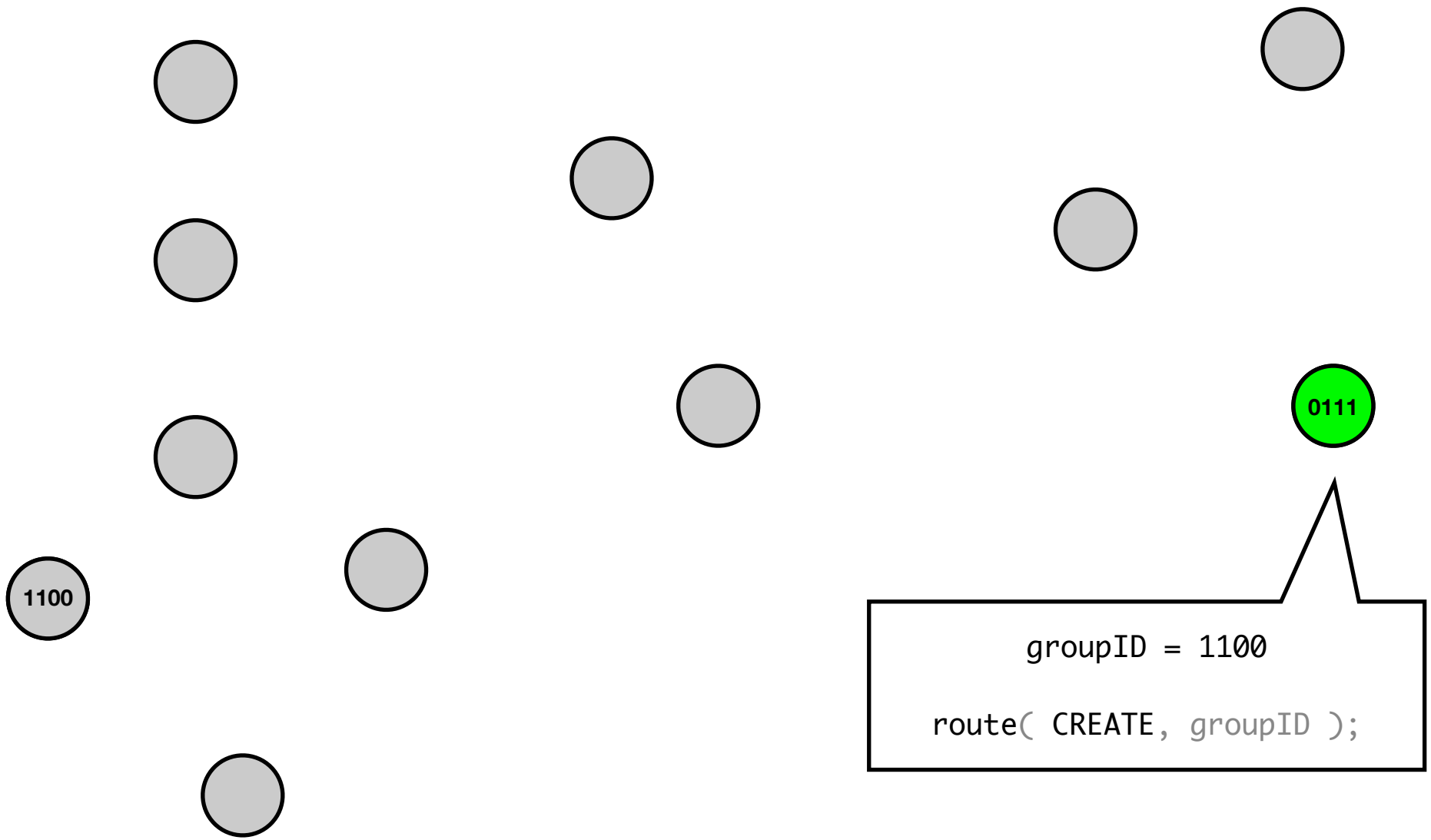
# Criar um grupo



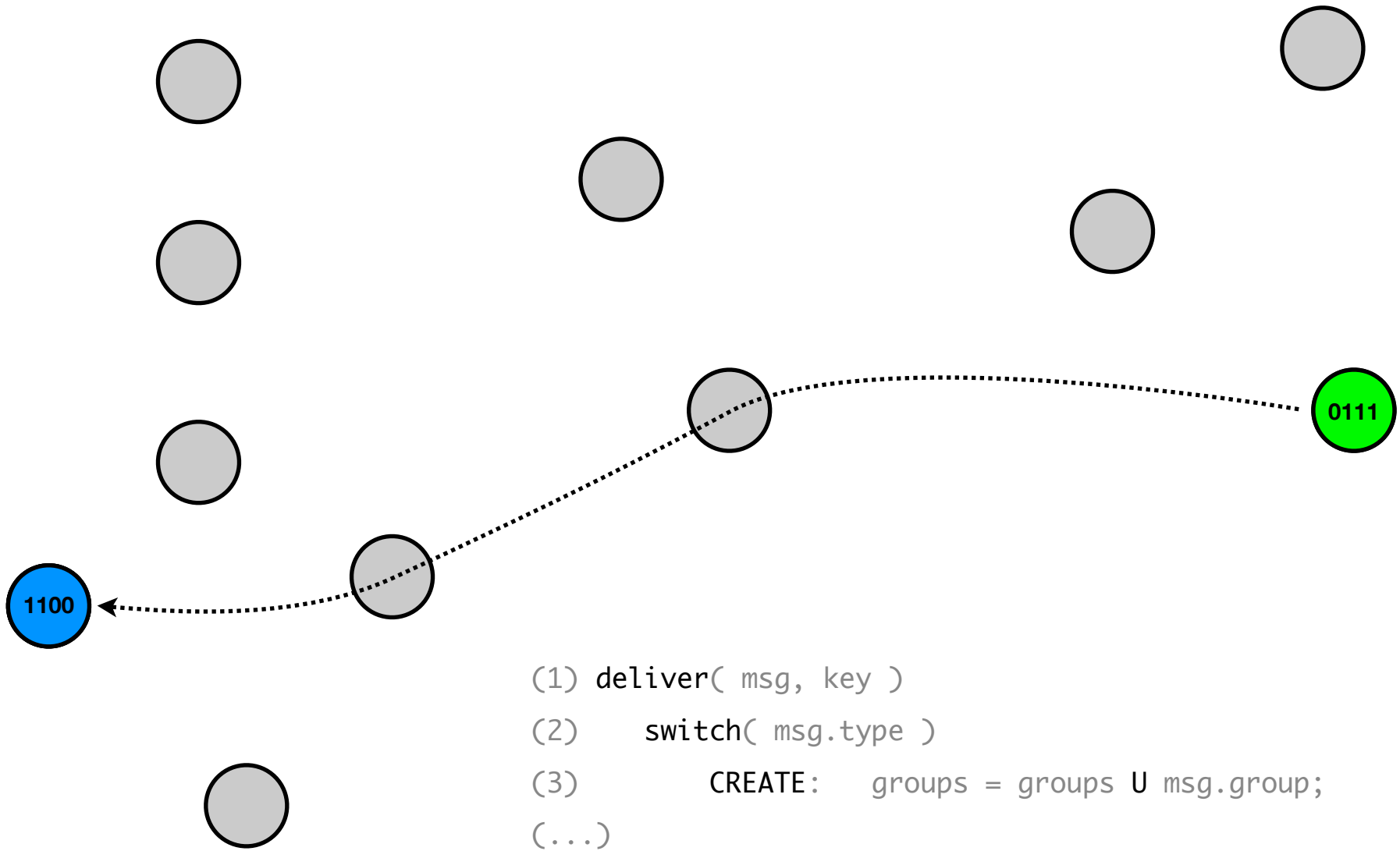
# Criar um grupo



# Criar um grupo



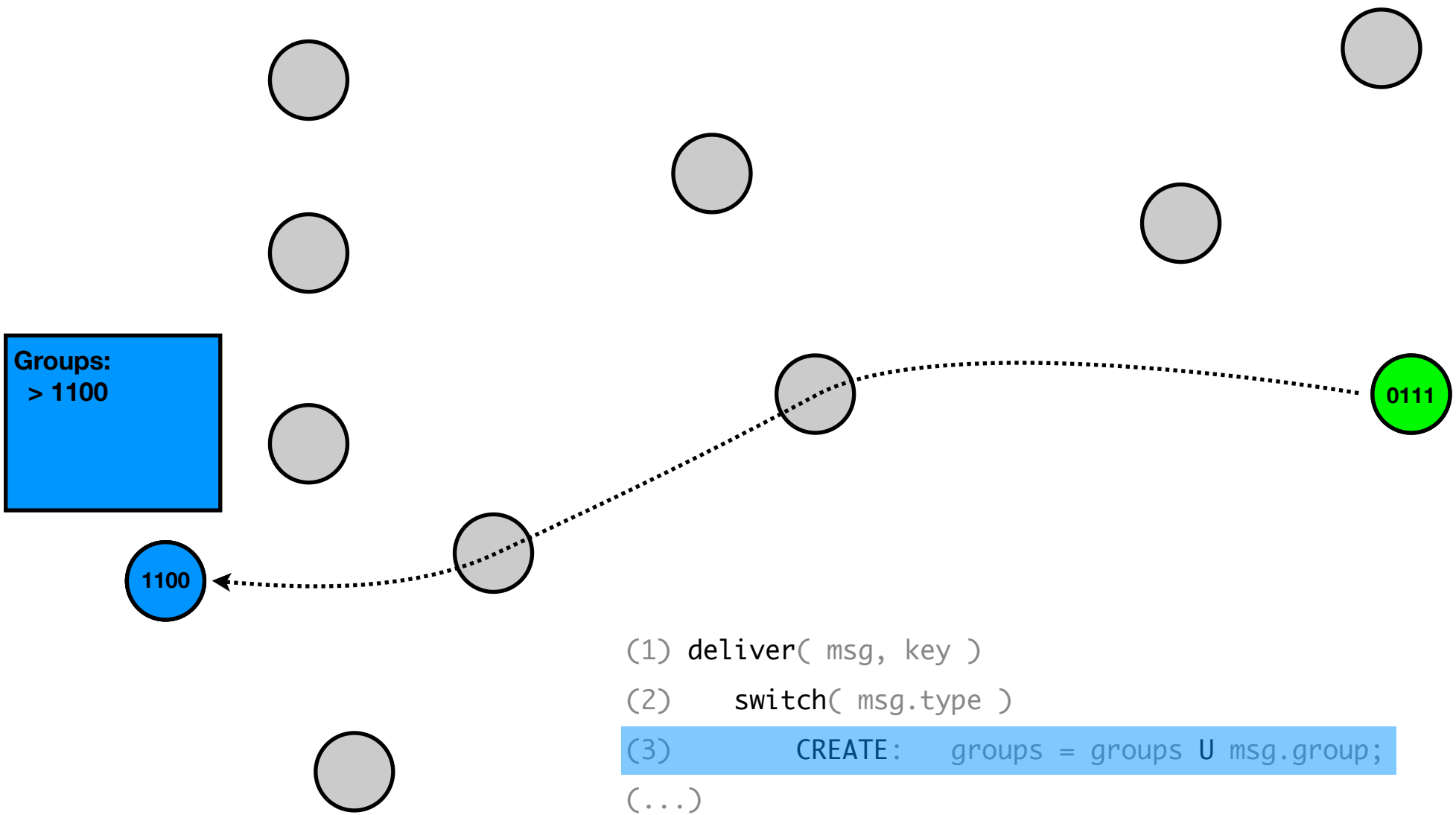
# Algoritmo > Gestão de Grupos
## Criar um grupo




```
(1) deliver( msg, key )
(2)     switch( msg.type )
(3)         CREATE:   groups = groups U msg.group;
(...)
```

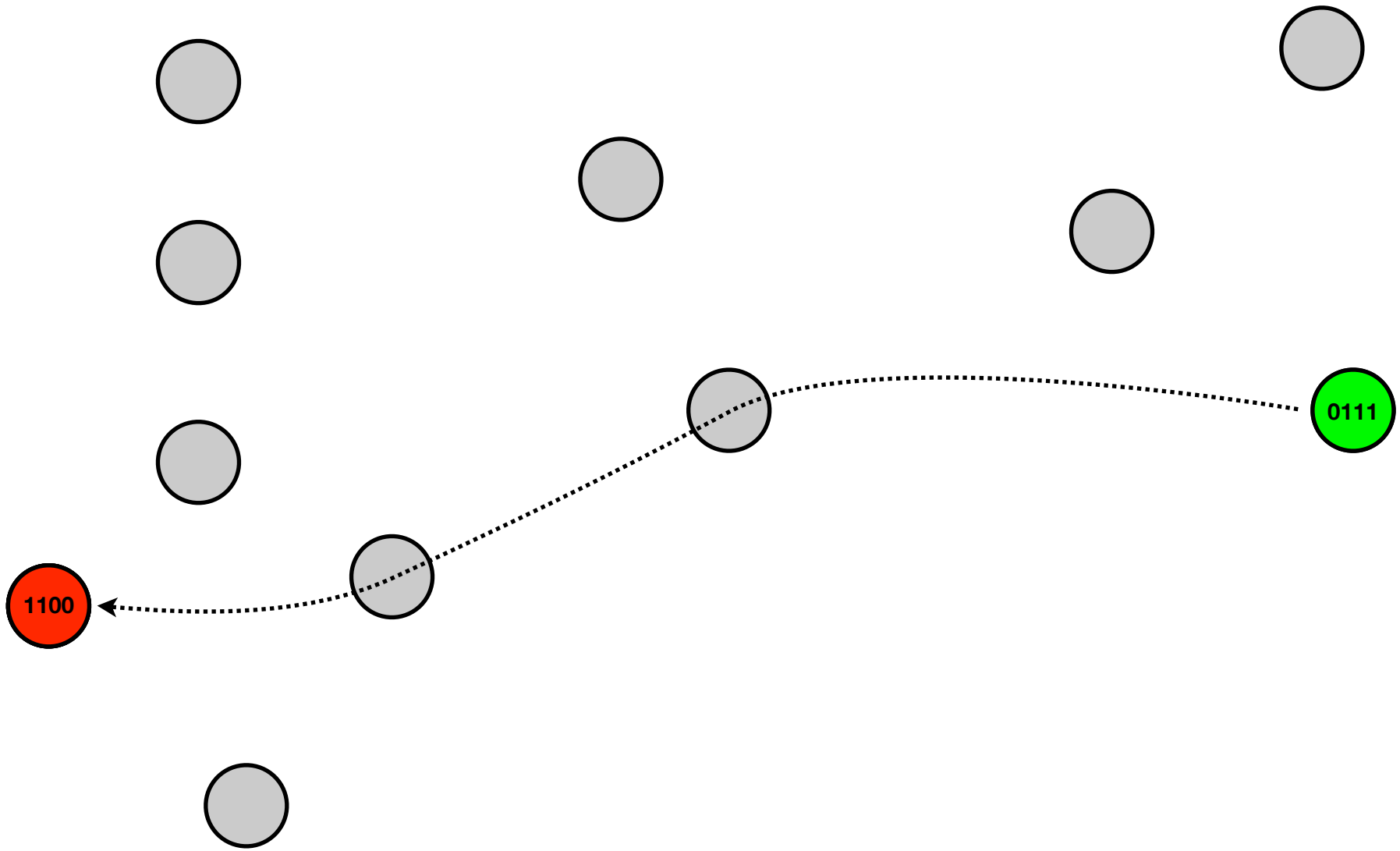
# Algoritmo > Gestão de Grupos

## Criar um grupo




```
(1) deliver( msg, key )
(2)     switch( msg.type )
(3)         CREATE:   groups = groups U msg.group;
(...)
```

# Criar um grupo

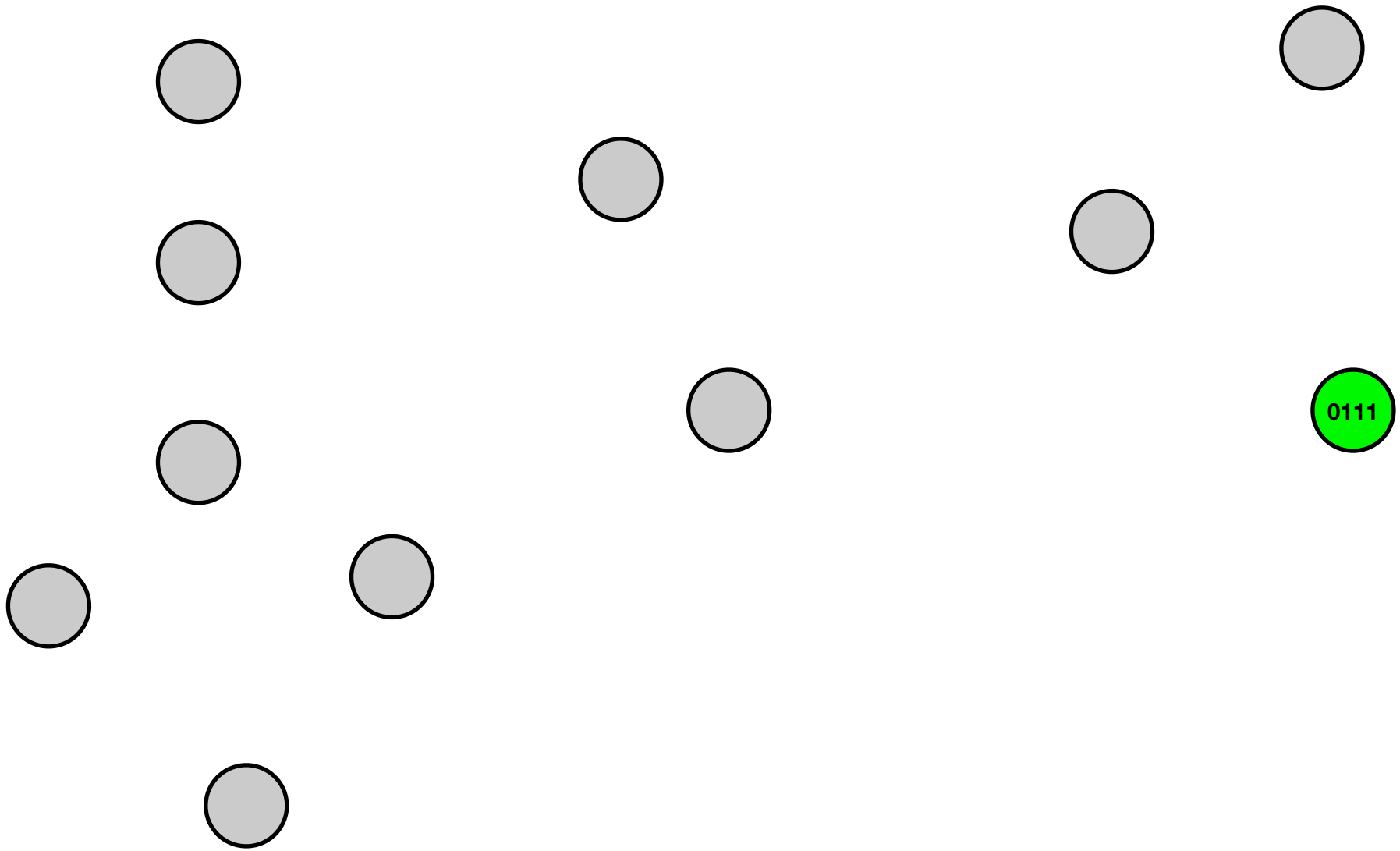


# Scribe Protocol
## Membership Management

- To **join** a group, a node sends a JOIN message to the group's *rendezvous point* using *Pastry's* route operation:
  - *Pastry* makes sure the message arrives to its destination
  - The forward method is invoked at each node along the route
  - Each of those nodes intercepts the JOIN message and:
    - If it did not have record of that group, adds it to its group list and sends a new JOIN message, similar to the prior one but with itself as the source
    - Adds the original source to that group's children list and drops the message

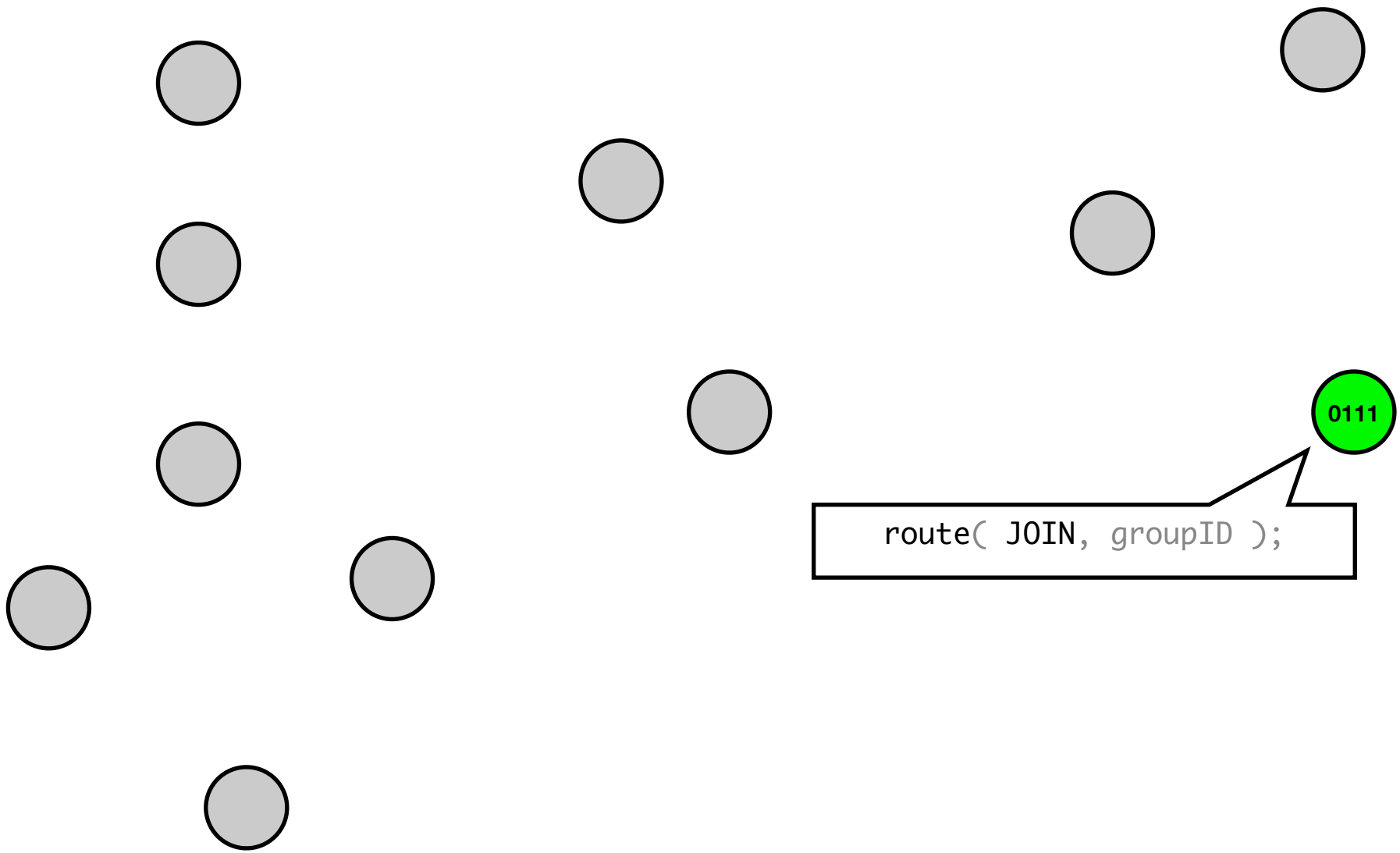
- To **leave** a group, a node records locally that it left the group:
  - When it no longer has children in that group's children table, it sends a LEAVE message to its parent
  - A leave message removes the sender from its parent's children table for that specific group

# Joining a Group



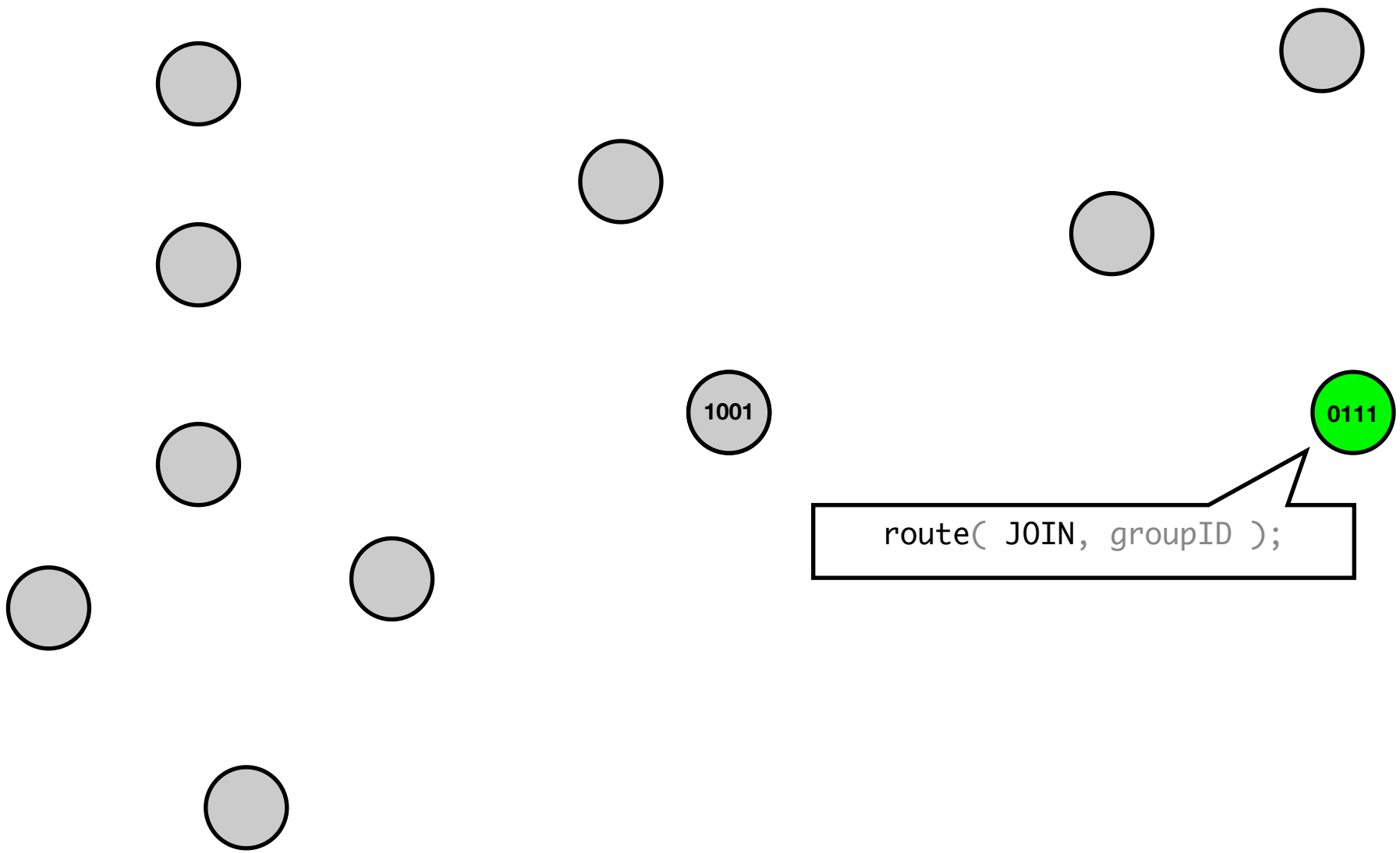
# Scribe Protocol > Membership Management

## Joining a Group



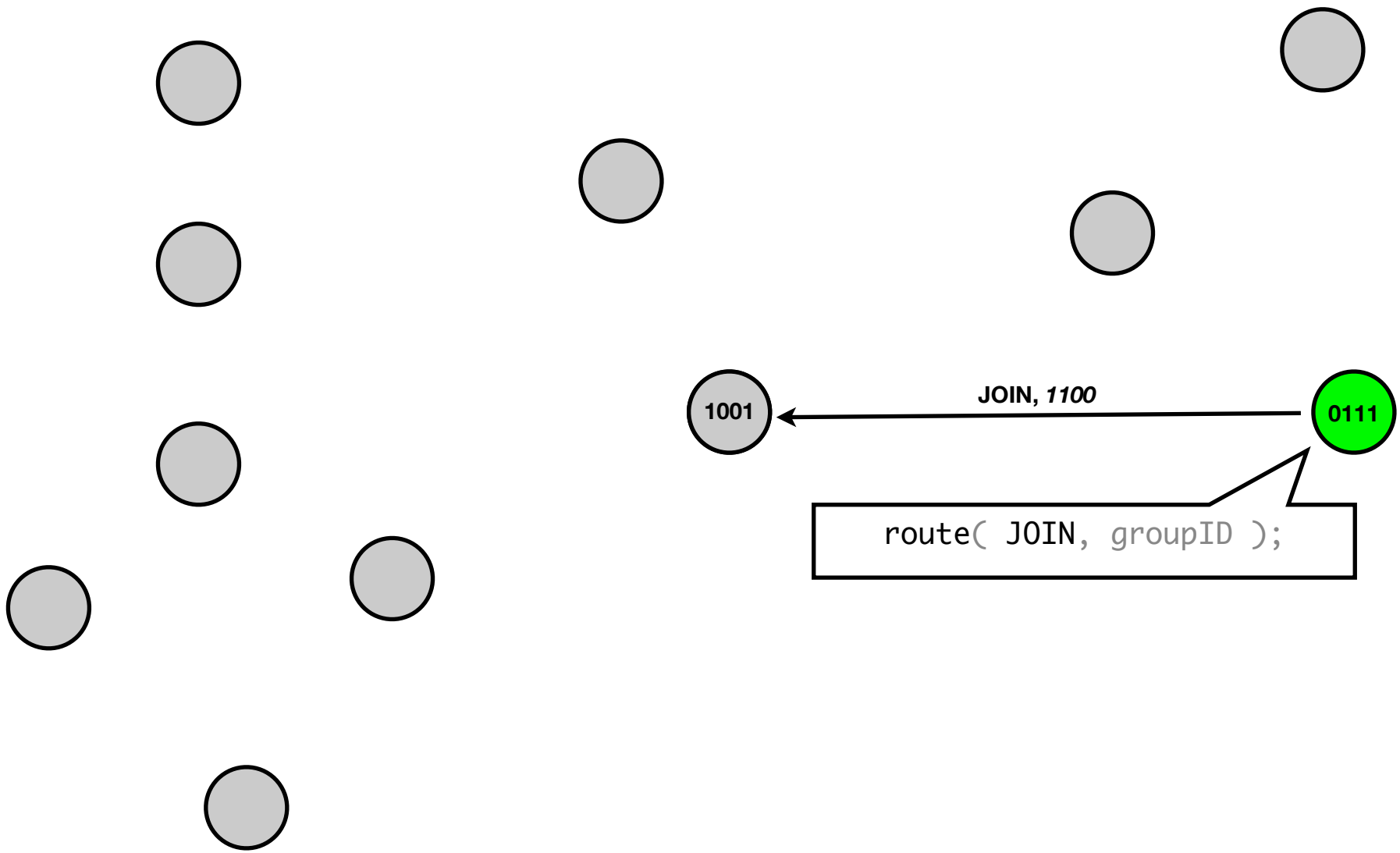
# Scribe Protocol > Membership Management

## Joining a Group



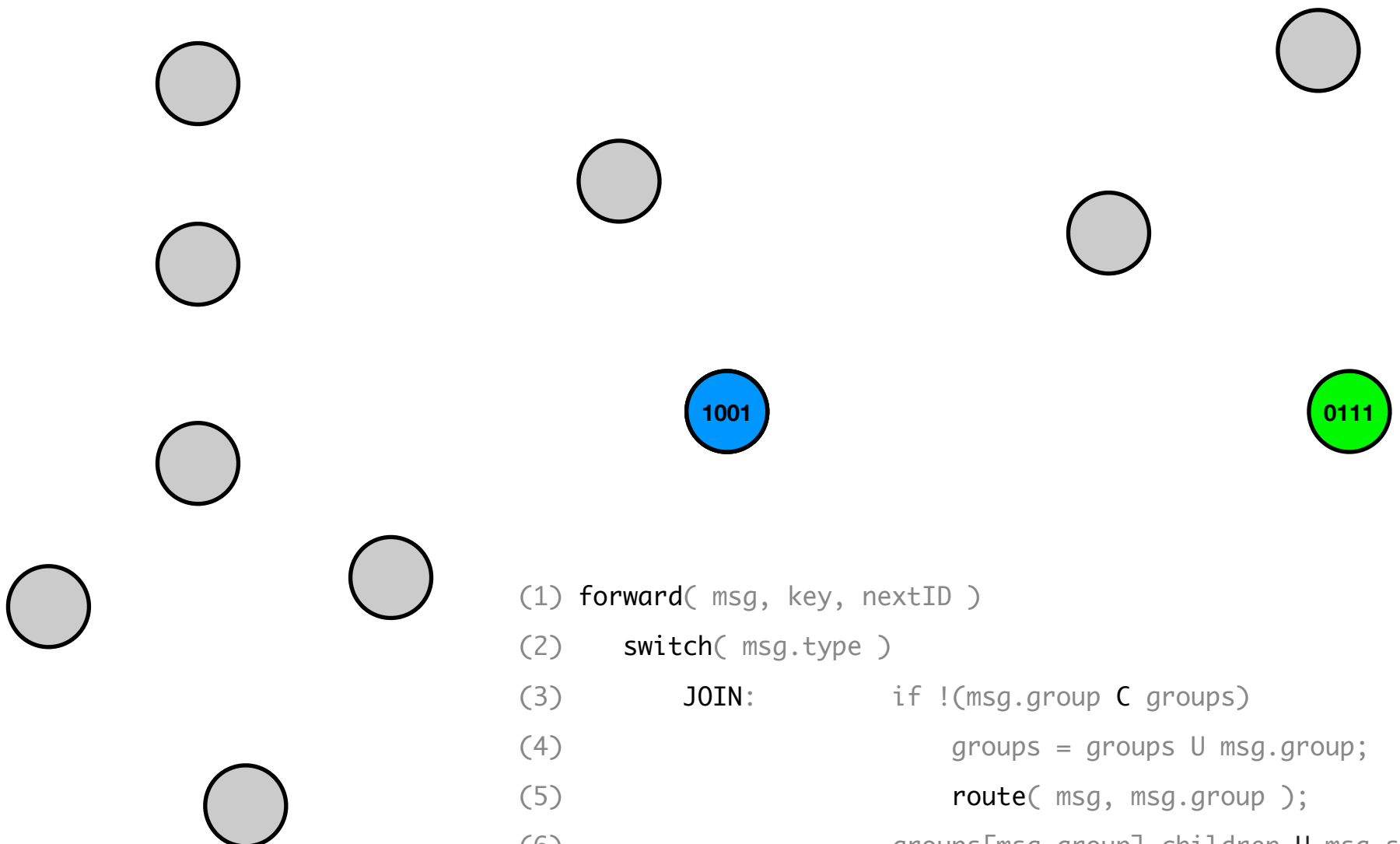
# Scribe Protocol > Membership Management

## Joining a Group



# Scribe Protocol > Membership Management

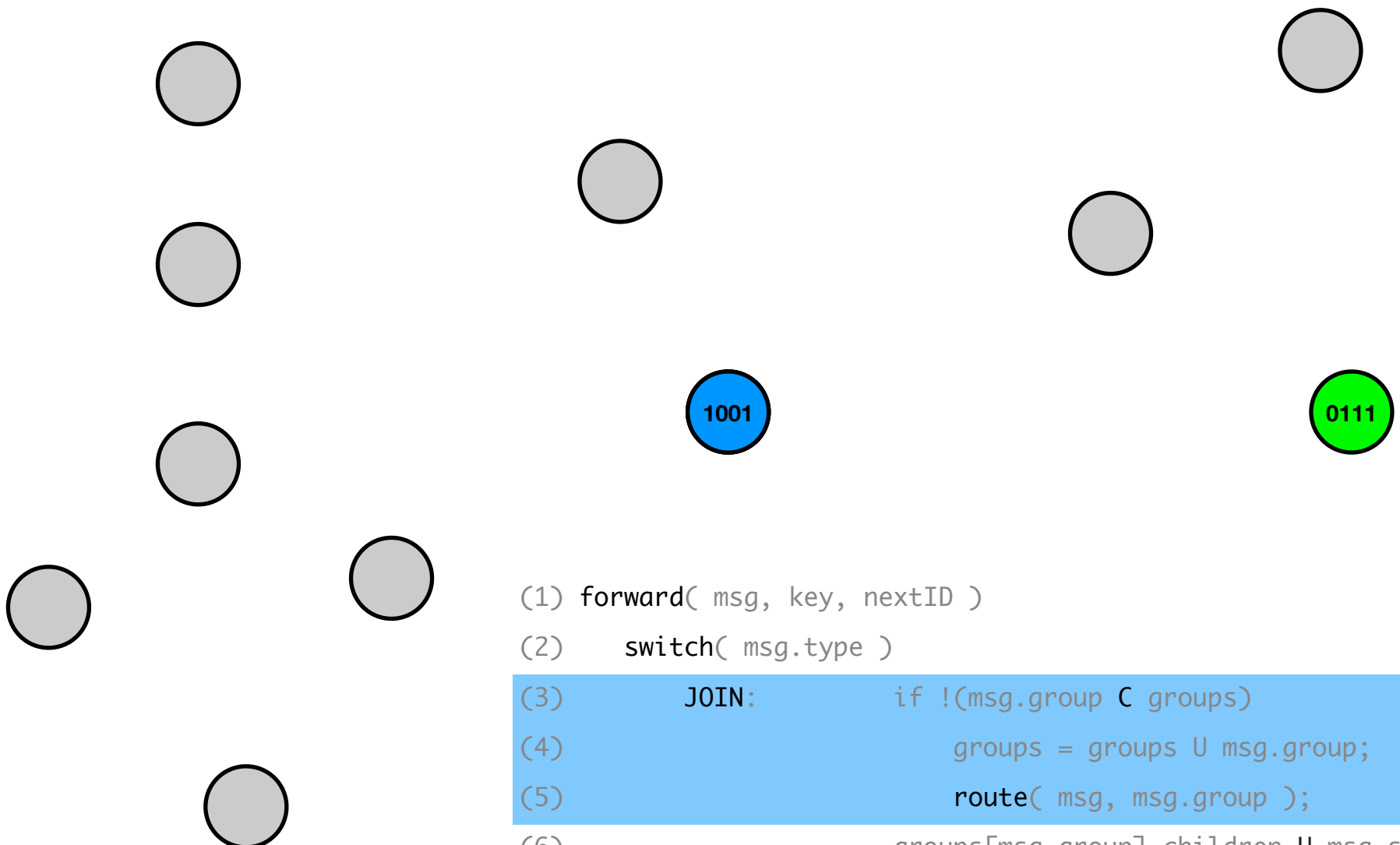
## Joining a Group




```
(1) forward( msg, key, nextID )
(2)    switch( msg.type )
(3)        JOIN:          if !(msg.group C groups)
(4)                           groups = groups U msg.group;
(5)                           route( msg, msg.group );
(6)                       groups[msg.group].children U msg.source;
(7)                       nextID = null;
```

# Scribe Protocol > Membership Management

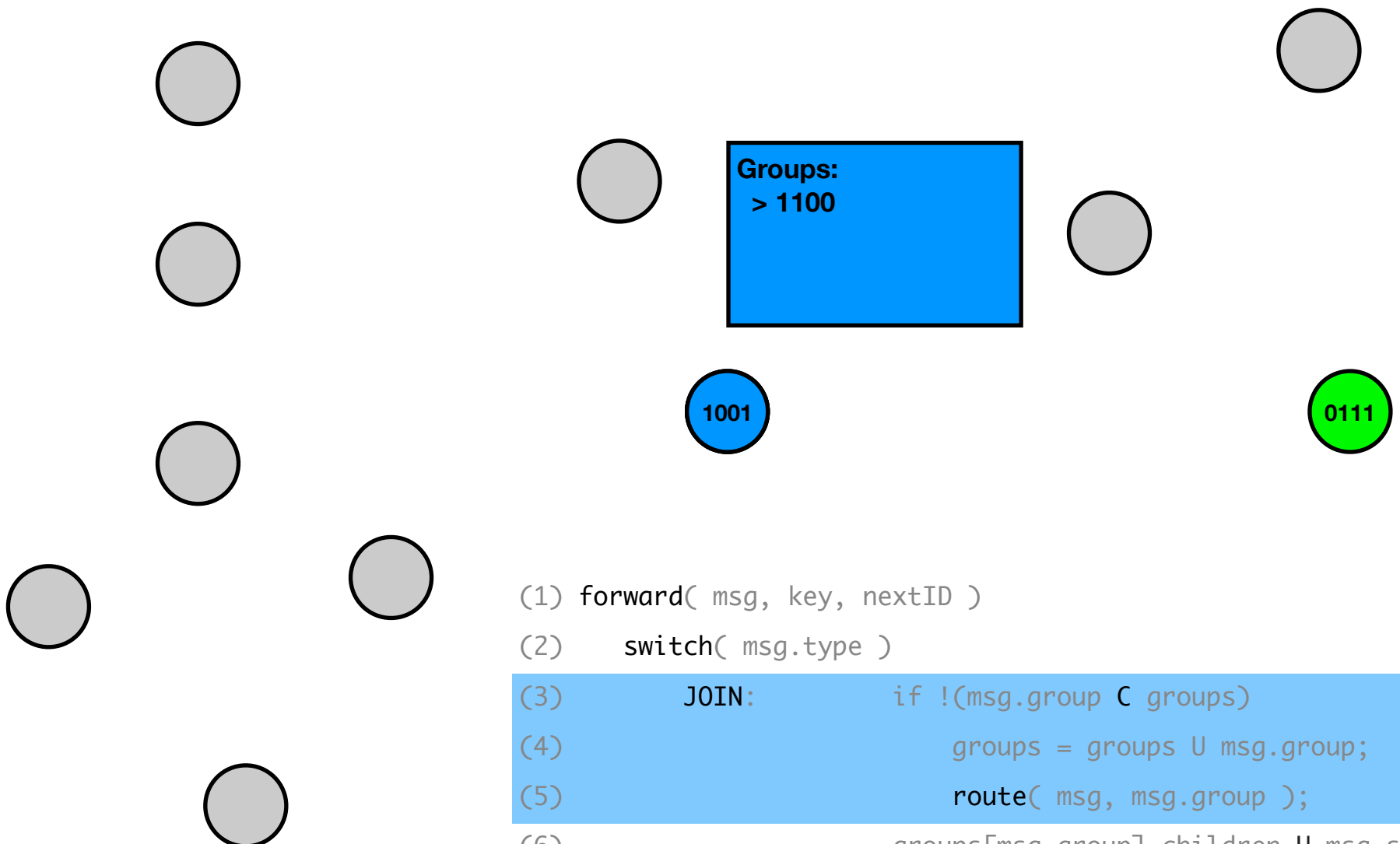
## Joining a Group




```
(1) forward( msg, key, nextID )
(2)    switch( msg.type )
(3)        JOIN:          if !(msg.group C groups)
(4)                           groups = groups U msg.group;
(5)                           route( msg, msg.group );
(6)                       groups[msg.group].children U msg.source;
(7)                       nextID = null;
```

# Scribe Protocol > Membership Management

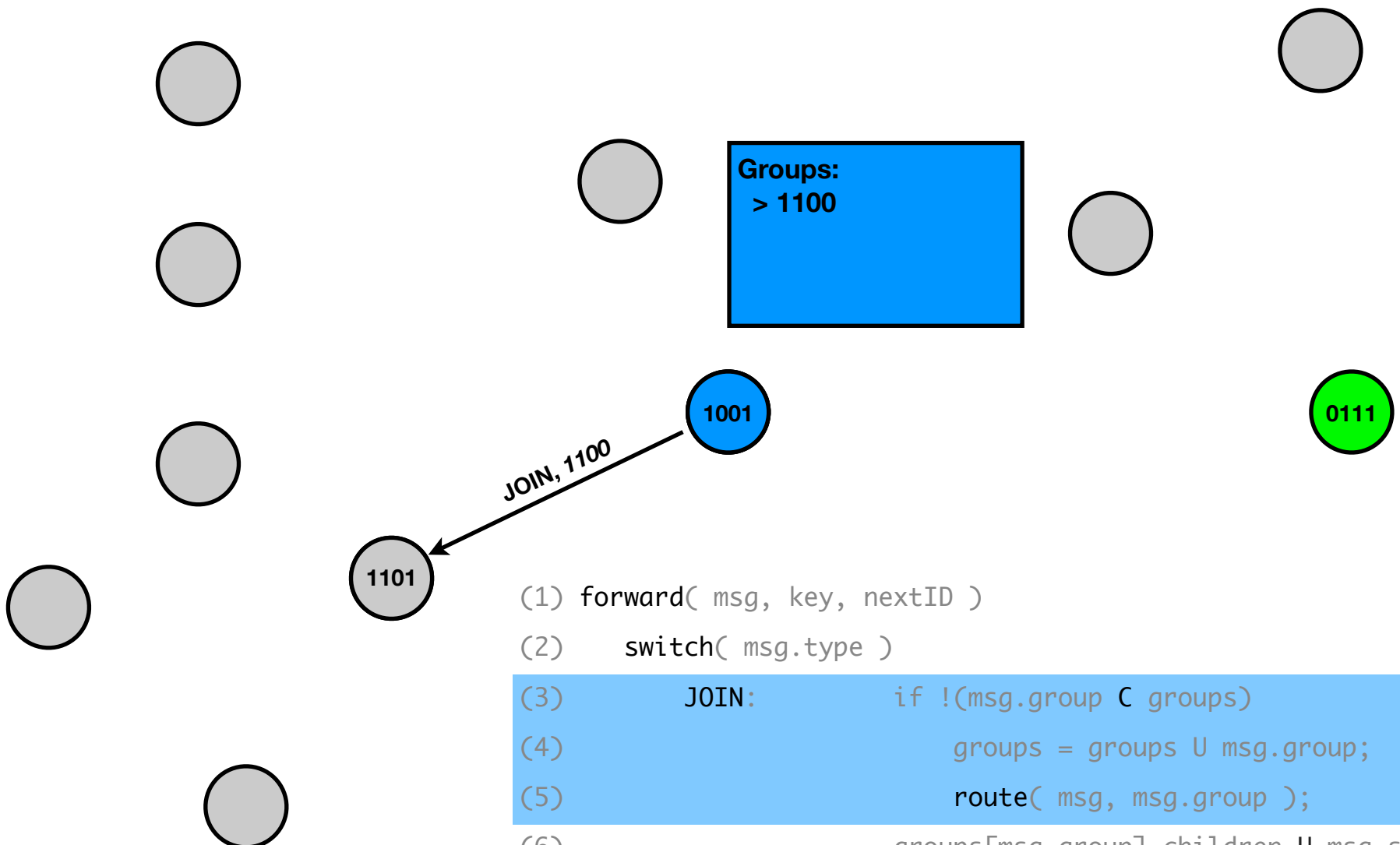
## Joining a Group




```
(1) forward( msg, key, nextID )
(2)     switch( msg.type )
(3)         JOIN:           if !(msg.group C groups)
(4)                             groups = groups U msg.group;
(5)                             route( msg, msg.group );
(6)                         groups[msg.group].children U msg.source;
(7)                         nextID = null;
```

# Scribe Protocol > Membership Management

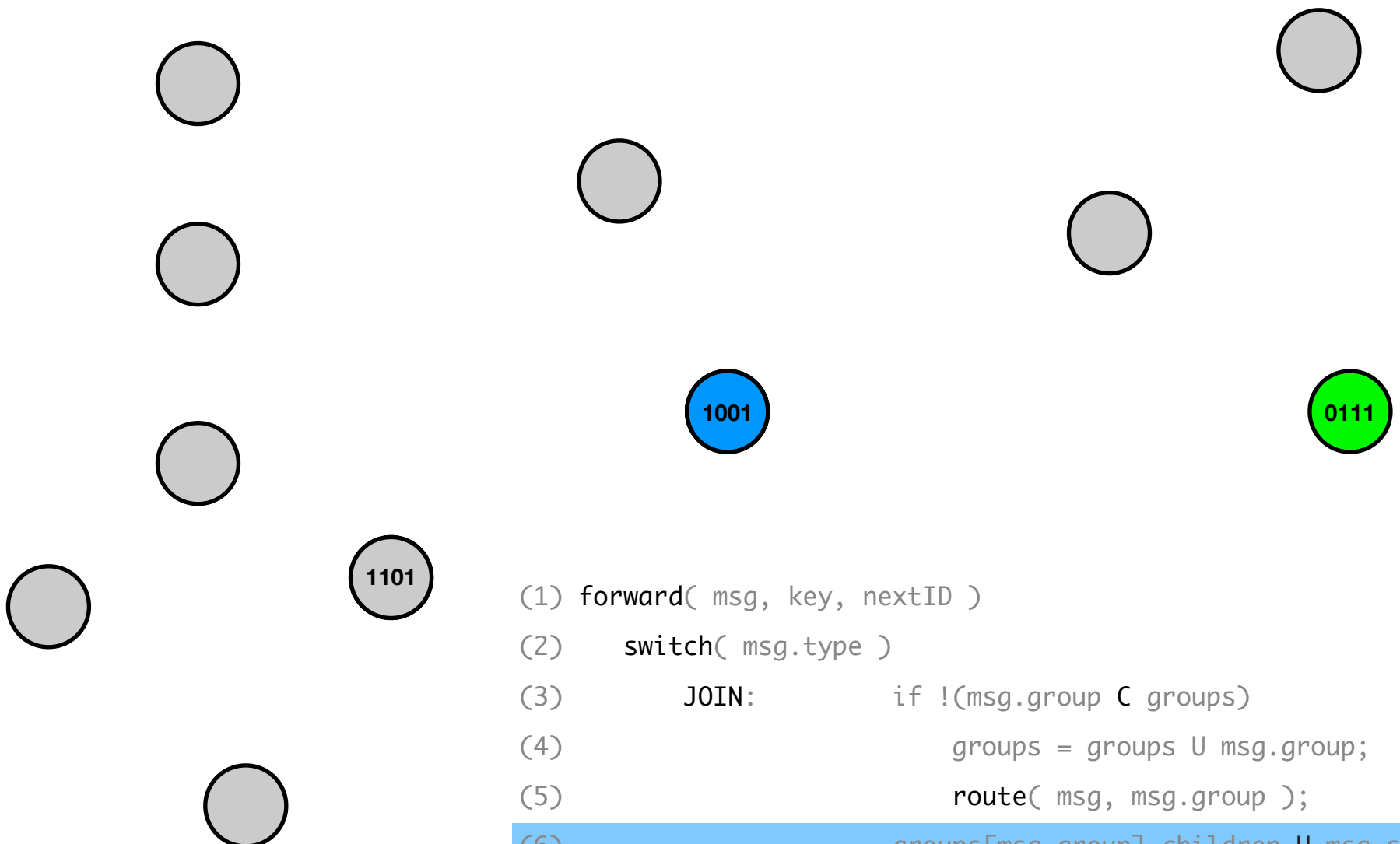
## Joining a Group




```
(1) forward( msg, key, nextID )
(2)     switch( msg.type )
(3)         JOIN:           if !(msg.group C groups)
(4)                             groups = groups U msg.group;
(5)                             route( msg, msg.group );
(6)                         groups[msg.group].children U msg.source;
(7)                         nextID = null;
```

# Scribe Protocol > Membership Management

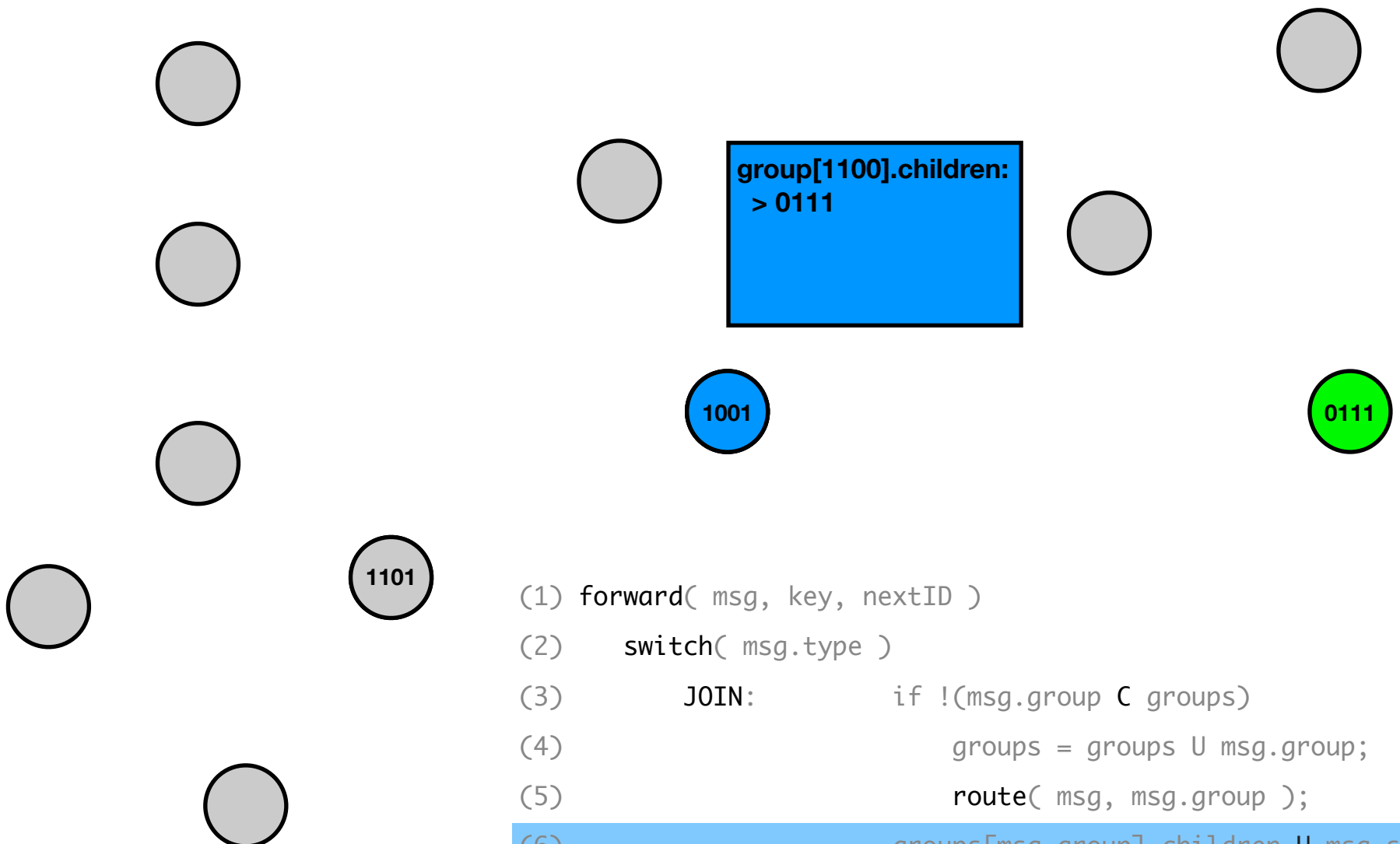
## Joining a Group




```
(1) forward( msg, key, nextID )
(2)    switch( msg.type )
(3)        JOIN:          if !(msg.group C groups)
(4)                           groups = groups U msg.group;
(5)                           route( msg, msg.group );
(6)                       groups[msg.group].children U msg.source;
(7)                       nextID = null;
```

# Scribe Protocol > Membership Management

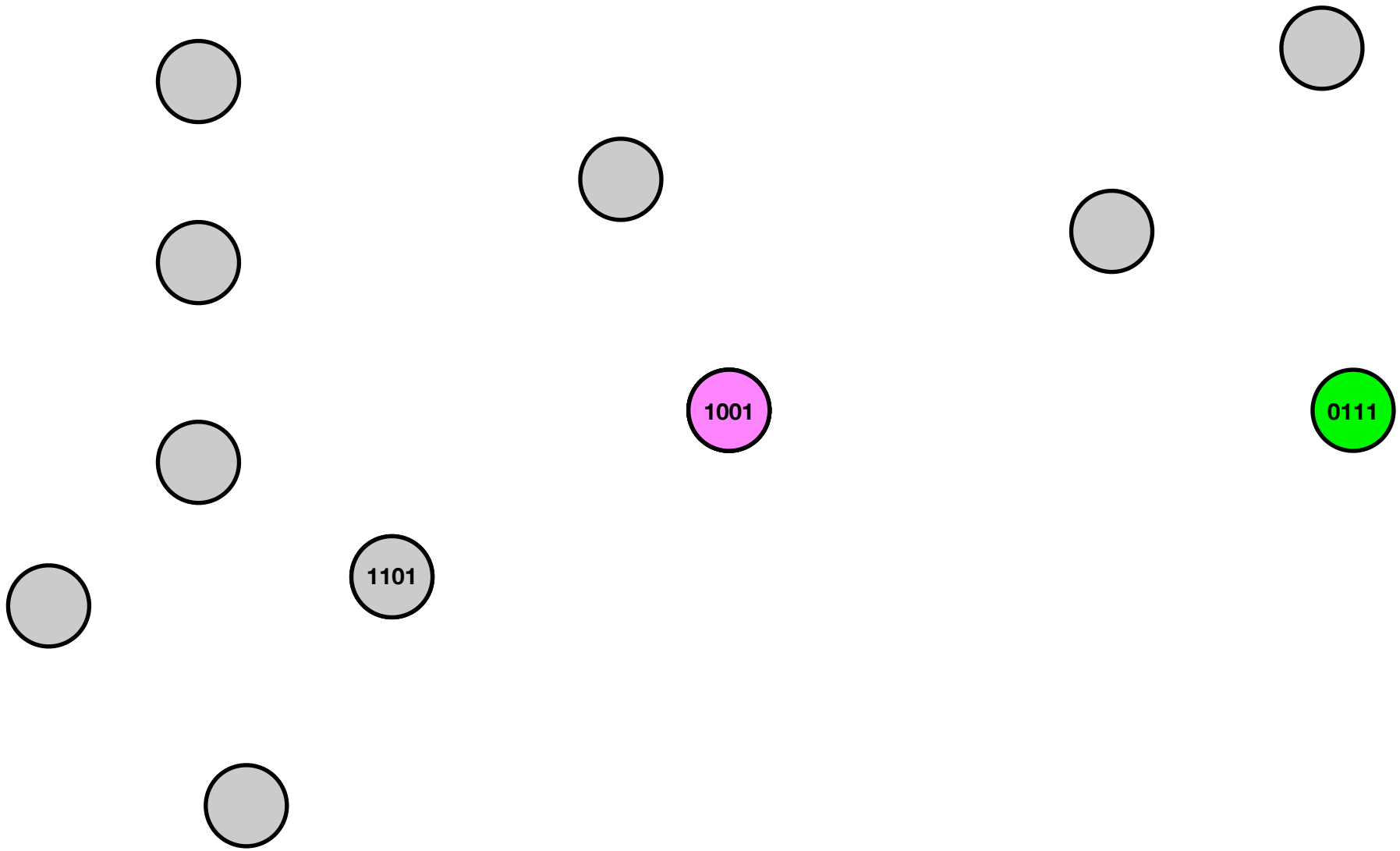
## Joining a Group




```
(1) forward( msg, key, nextID )
(2)    switch( msg.type )
(3)        JOIN:          if !(msg.group C groups)
(4)                           groups = groups U msg.group;
(5)                           route( msg, msg.group );
(6)                       groups[msg.group].children U msg.source;
(7)                       nextID = null;
```

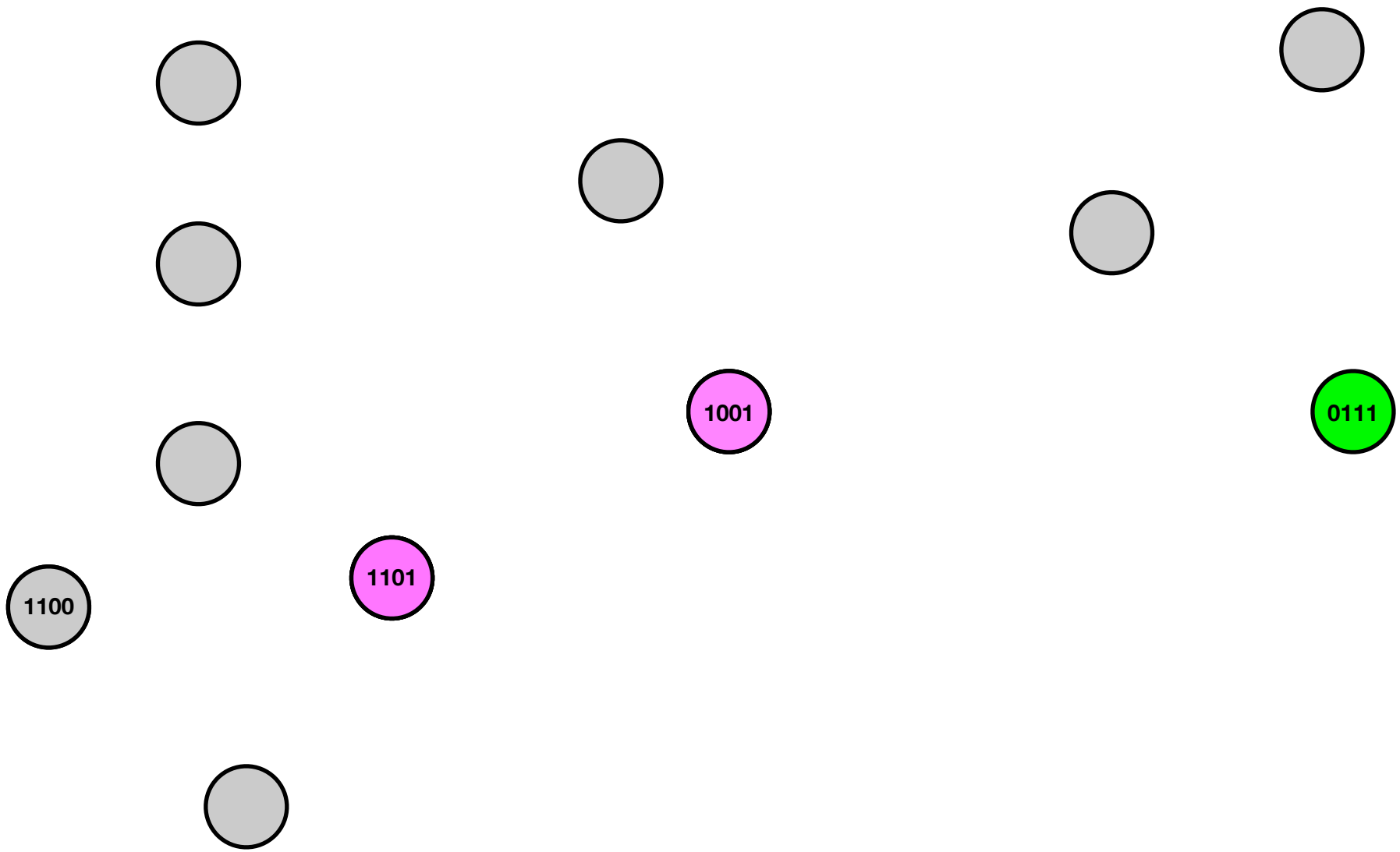
# Scribe Protocol > Membership Management

## Joining a Group

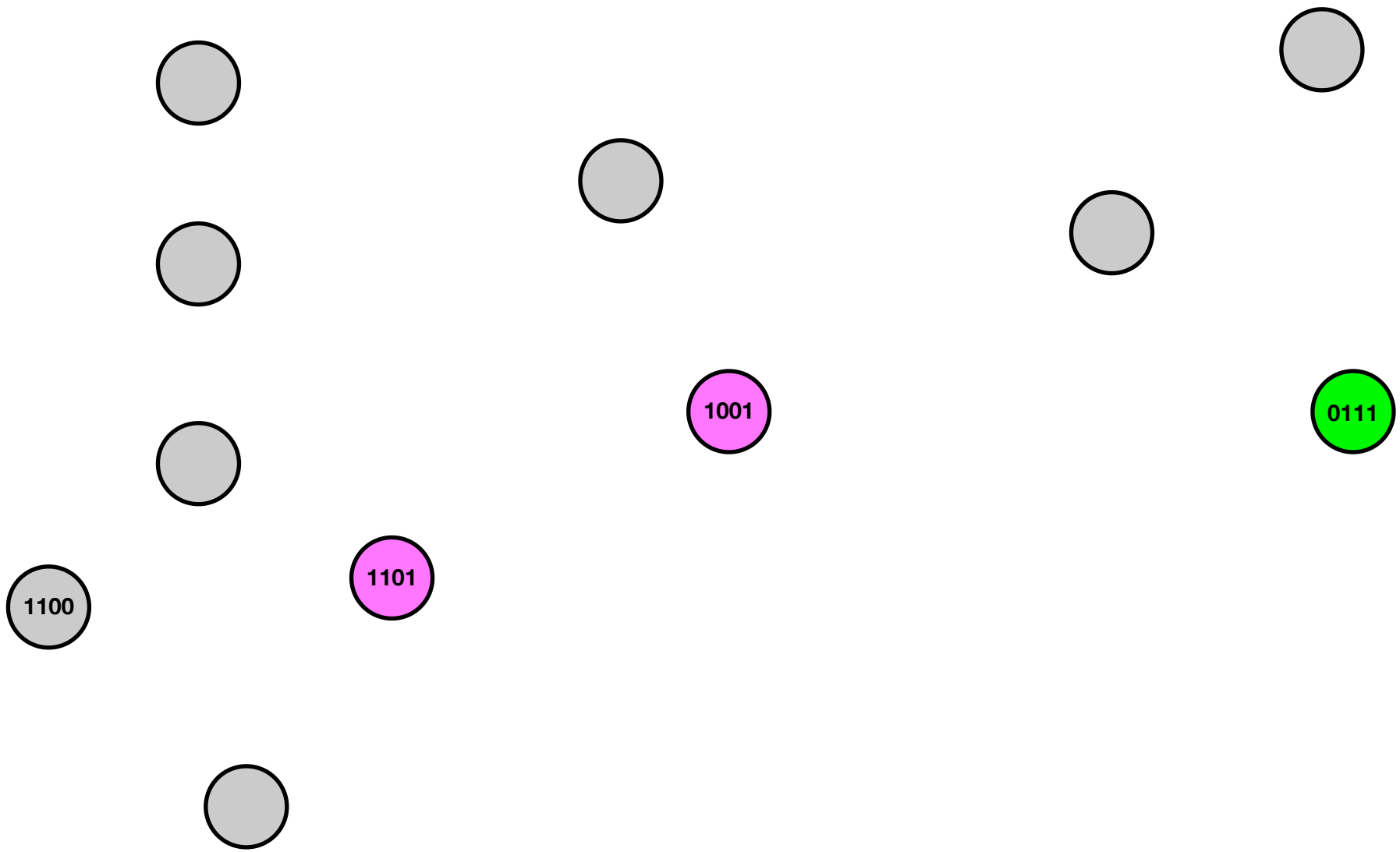


# Scribe Protocol > Membership Management

## Joining a Group

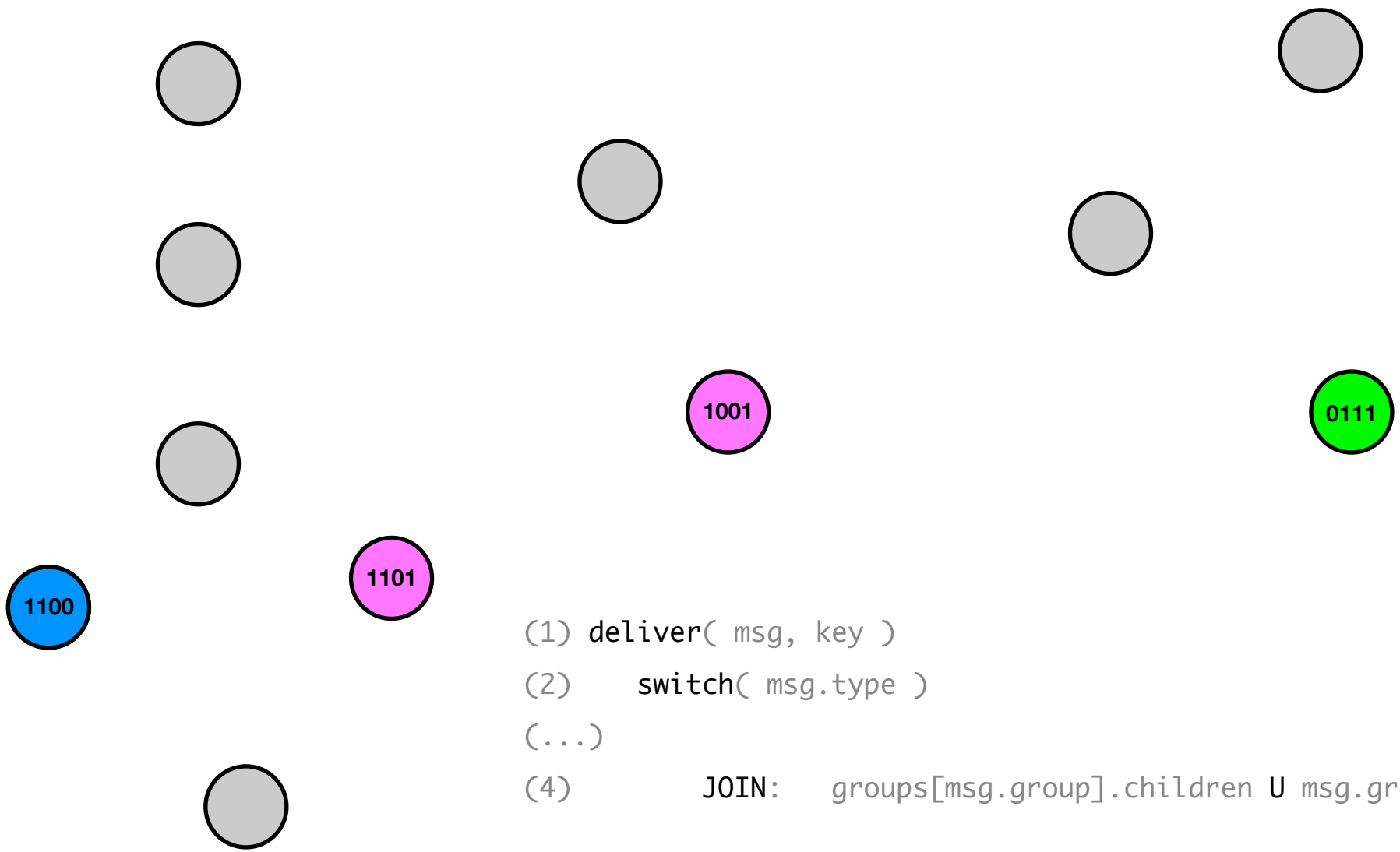


# Joining a Group



# Scribe Protocol > Membership Management

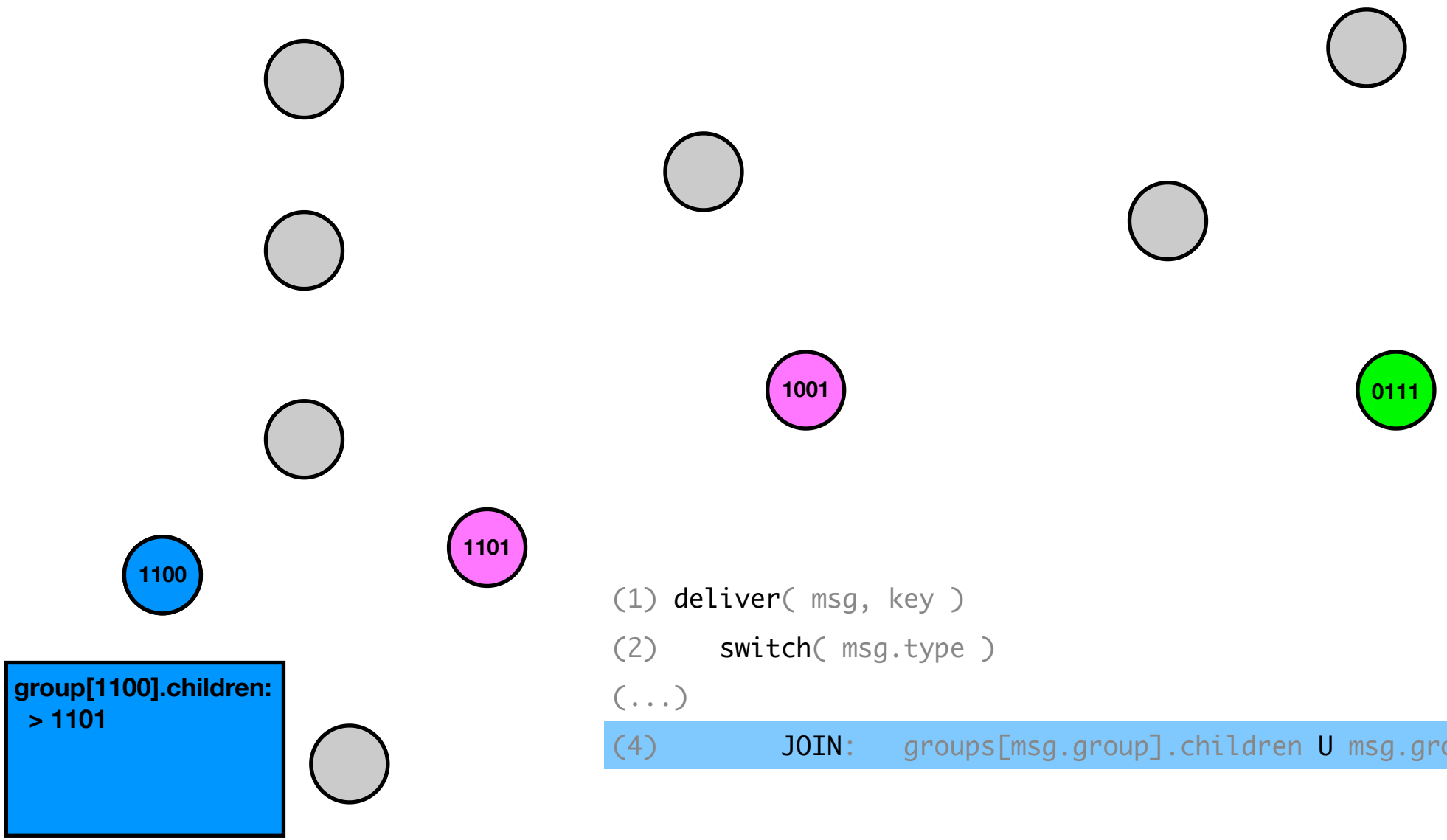
## Joining a Group




```
(1) deliver( msg, key )
(2)    switch( msg.type )
(...)
(4)        JOIN:   groups[msg.group].children U msg.group;
```

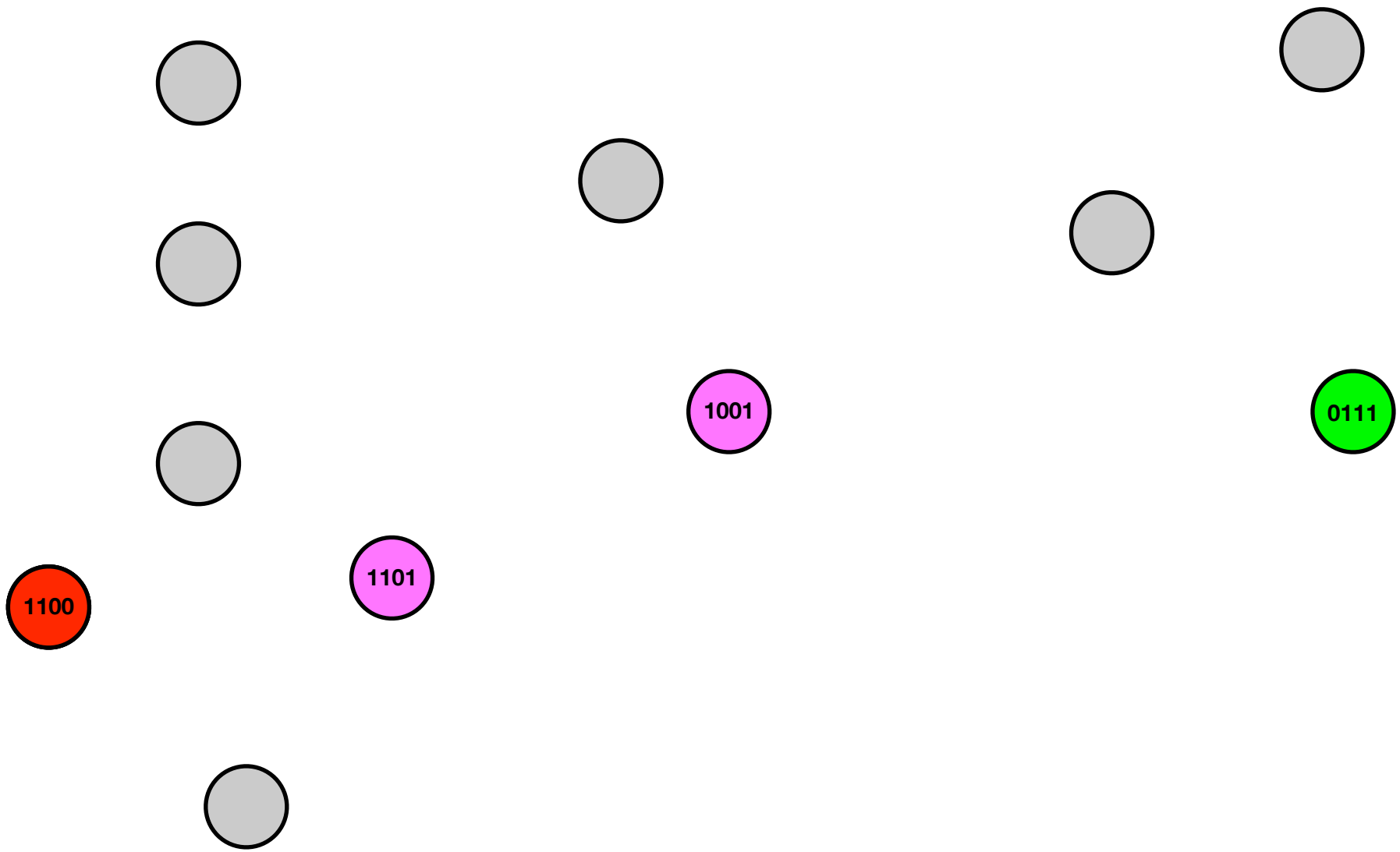
# Scribe Protocol > Membership Management

## Joining a Group

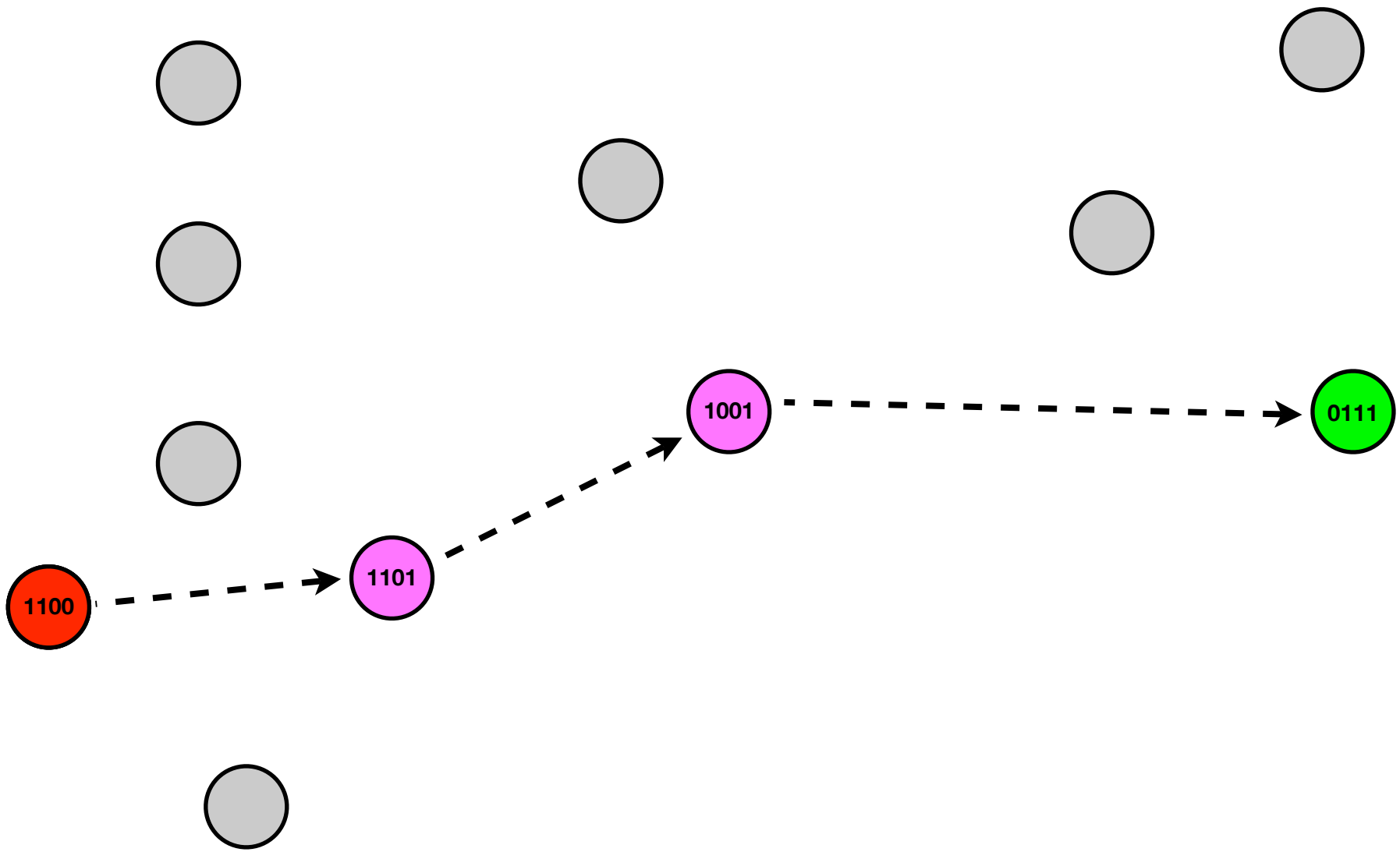



```
(1) deliver( msg, key )
(2)     switch( msg.type )
(...)
(4)         JOIN:   groups[msg.group].children U msg.group;
```

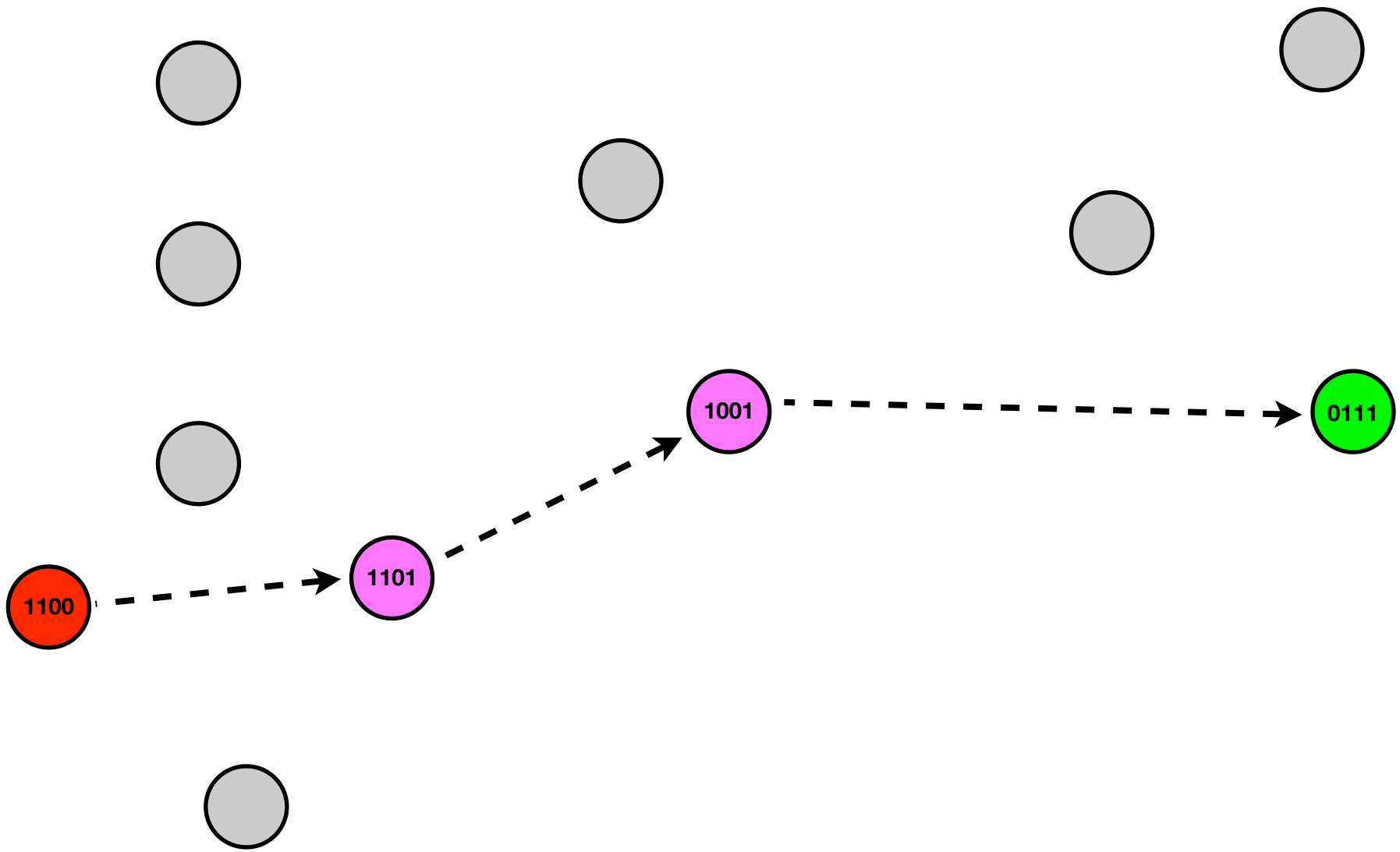
# Joining a Group



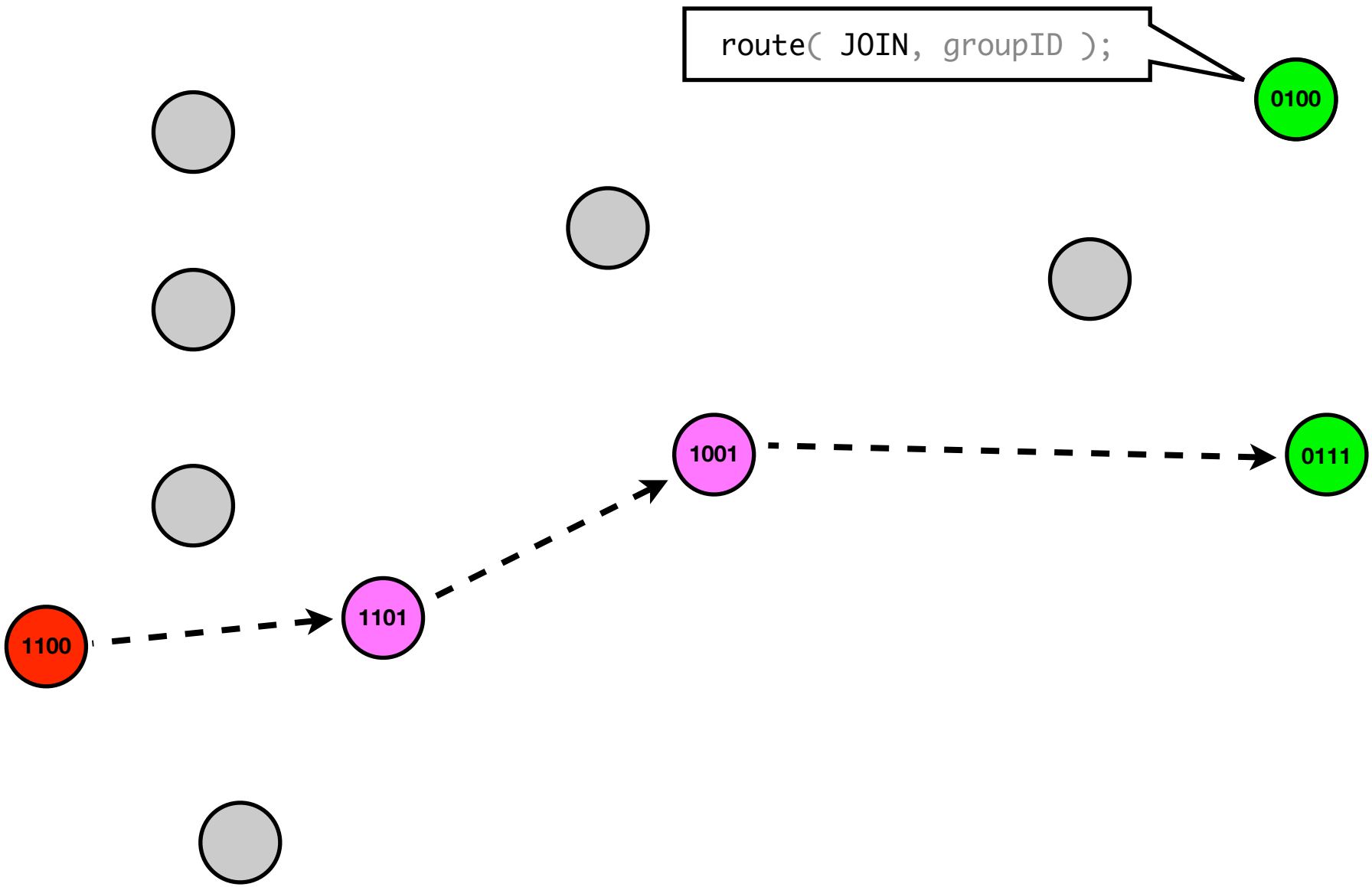
# Joining a Group



# Joining a Group

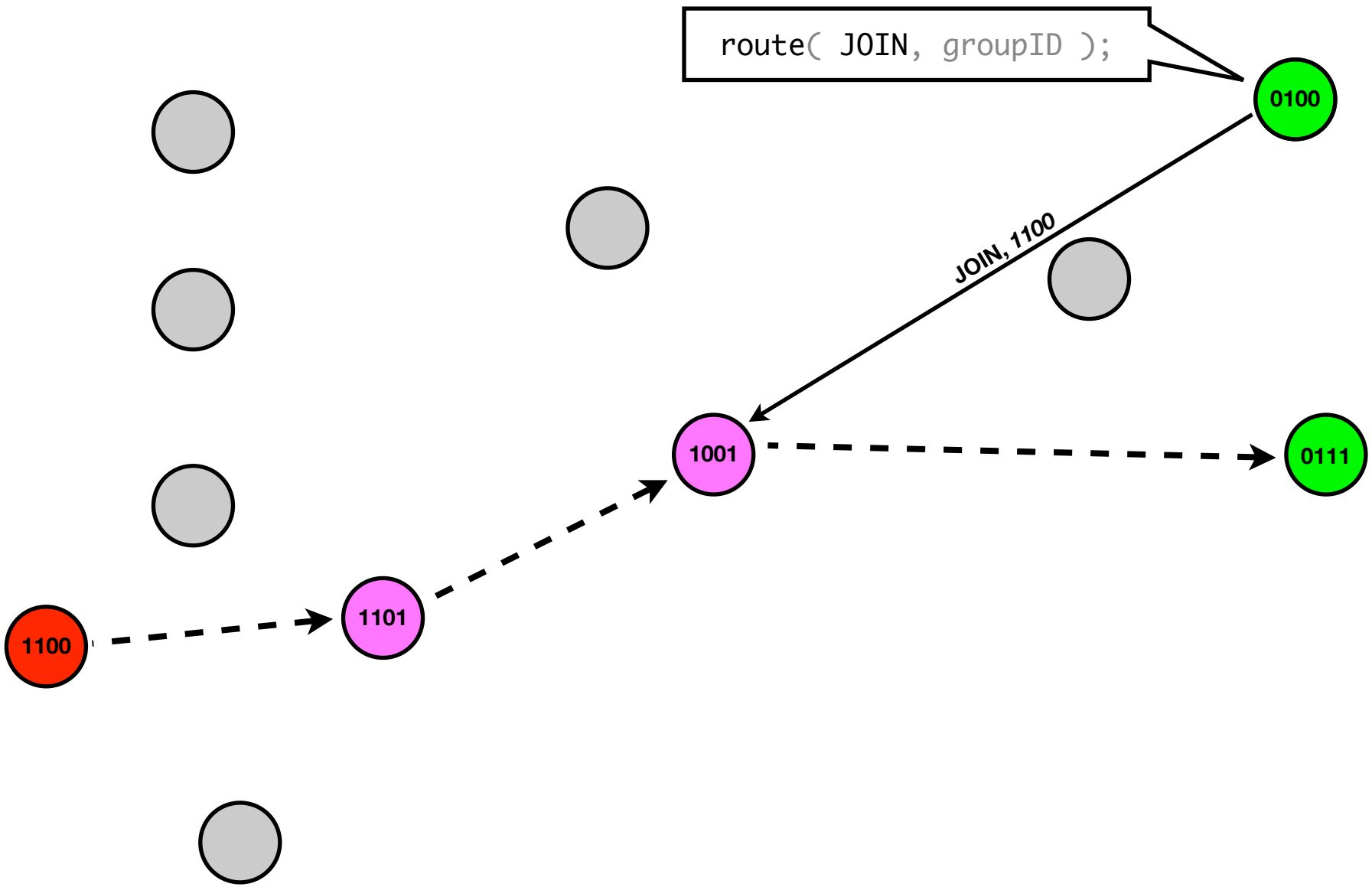


# Joining a Group



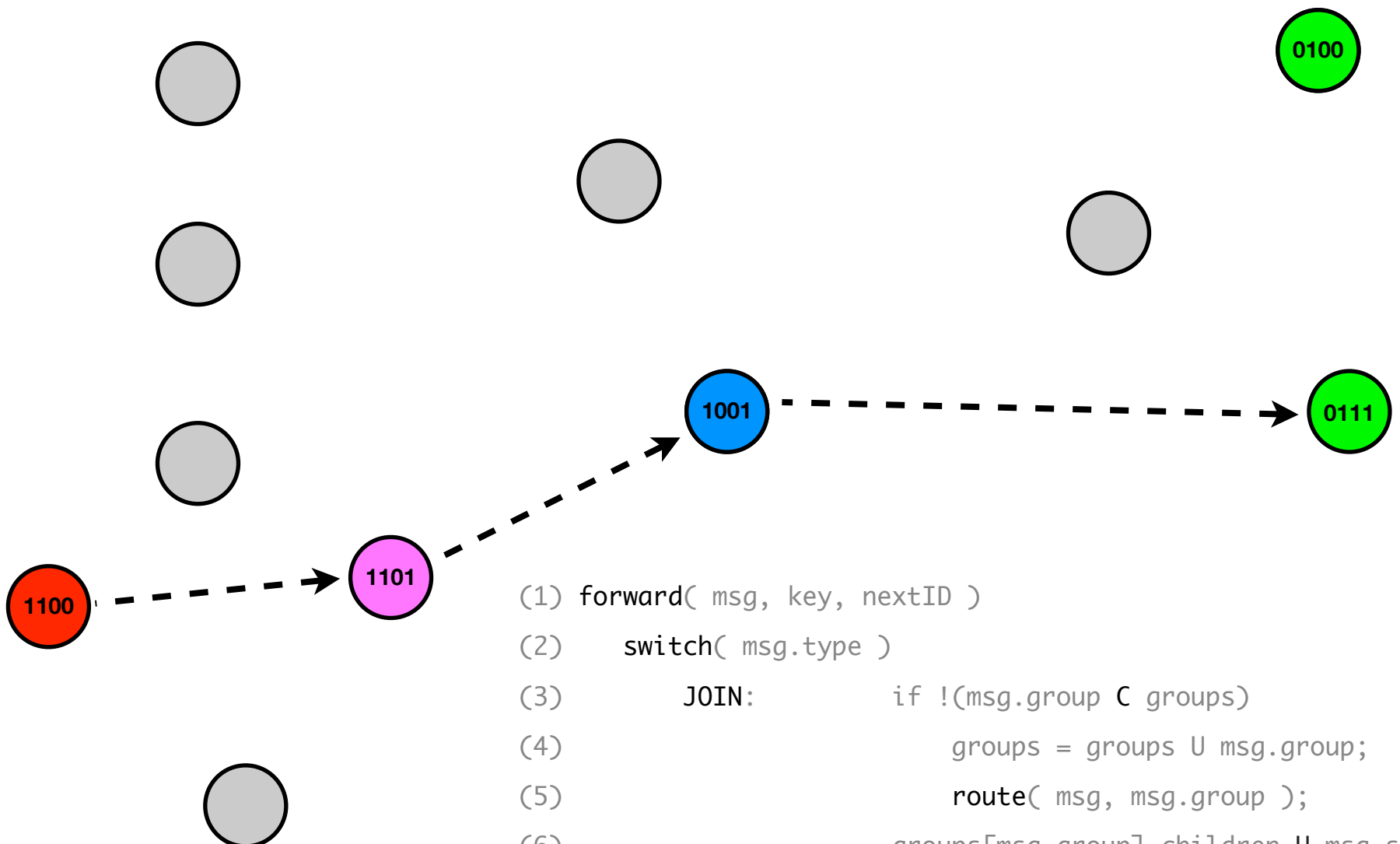
# Scribe Protocol > Membership Management

## Joining a Group



# Scribe Protocol > Membership Management

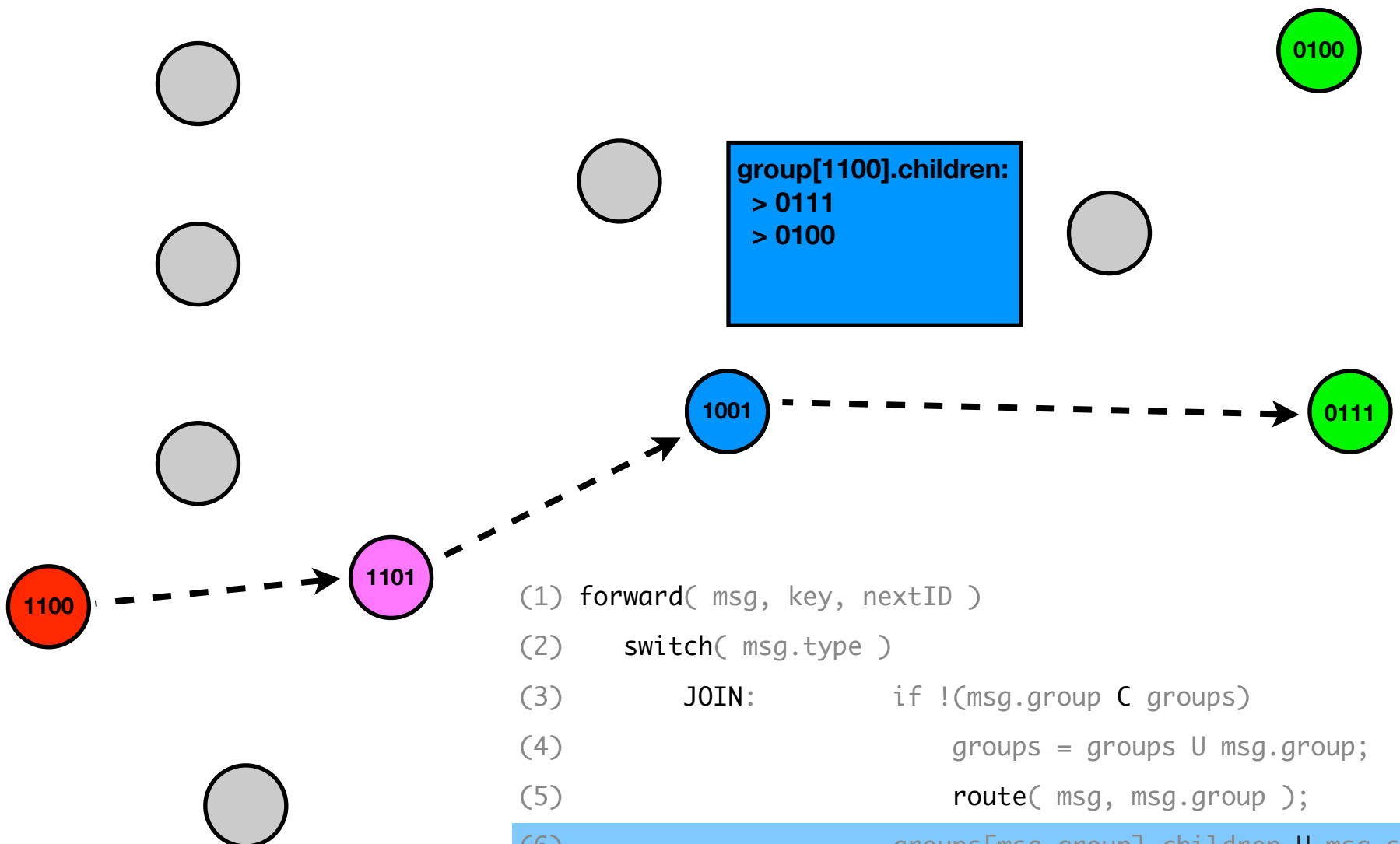
## Joining a Group




```
(1) forward( msg, key, nextID )
(2)    switch( msg.type )
(3)        JOIN:          if !(msg.group C groups)
(4)                           groups = groups U msg.group;
(5)                           route( msg, msg.group );
(6)                       groups[msg.group].children U msg.source;
(7)                       nextID = null;
```

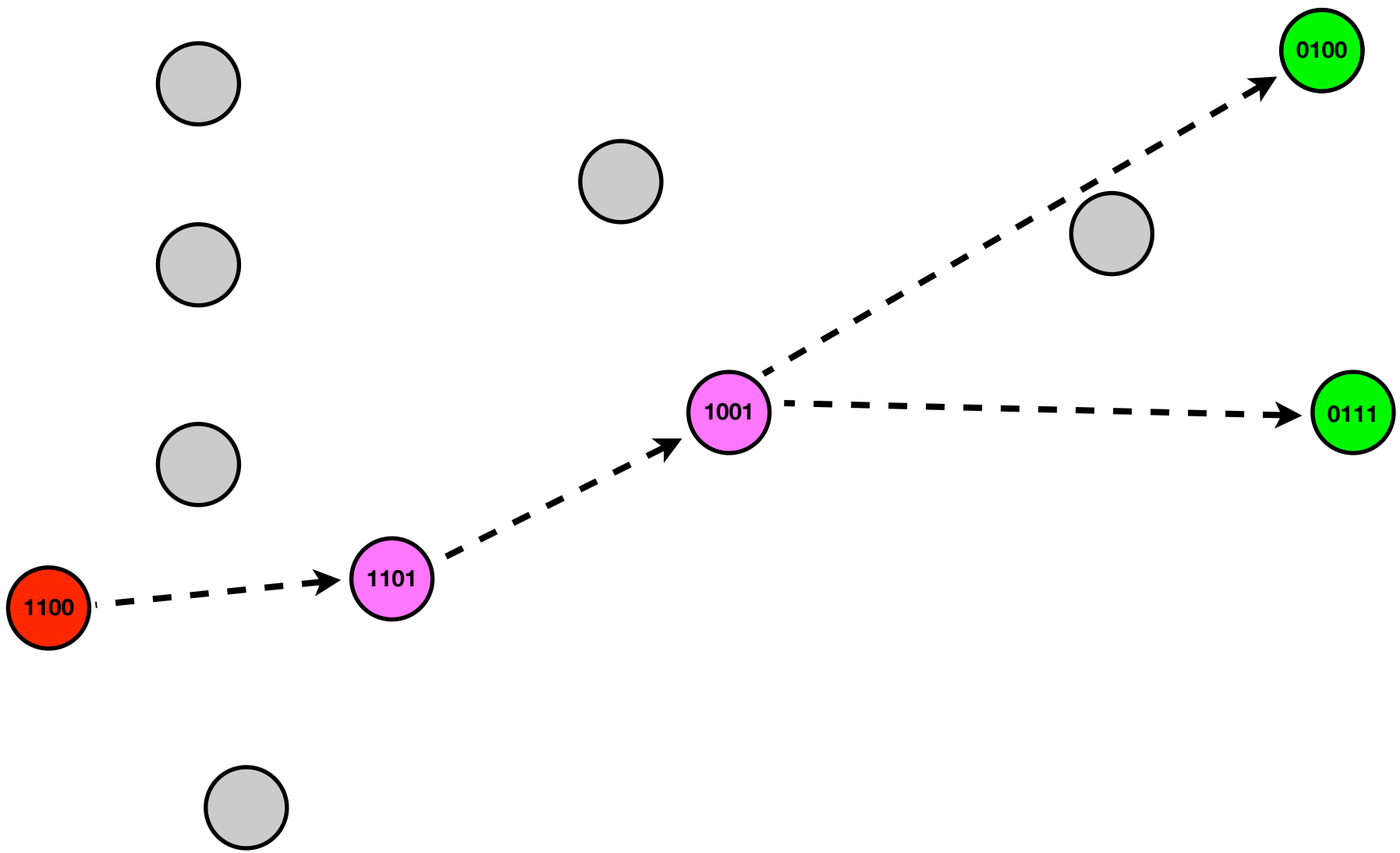
# Scribe Protocol > Membership Management

## Joining a Group

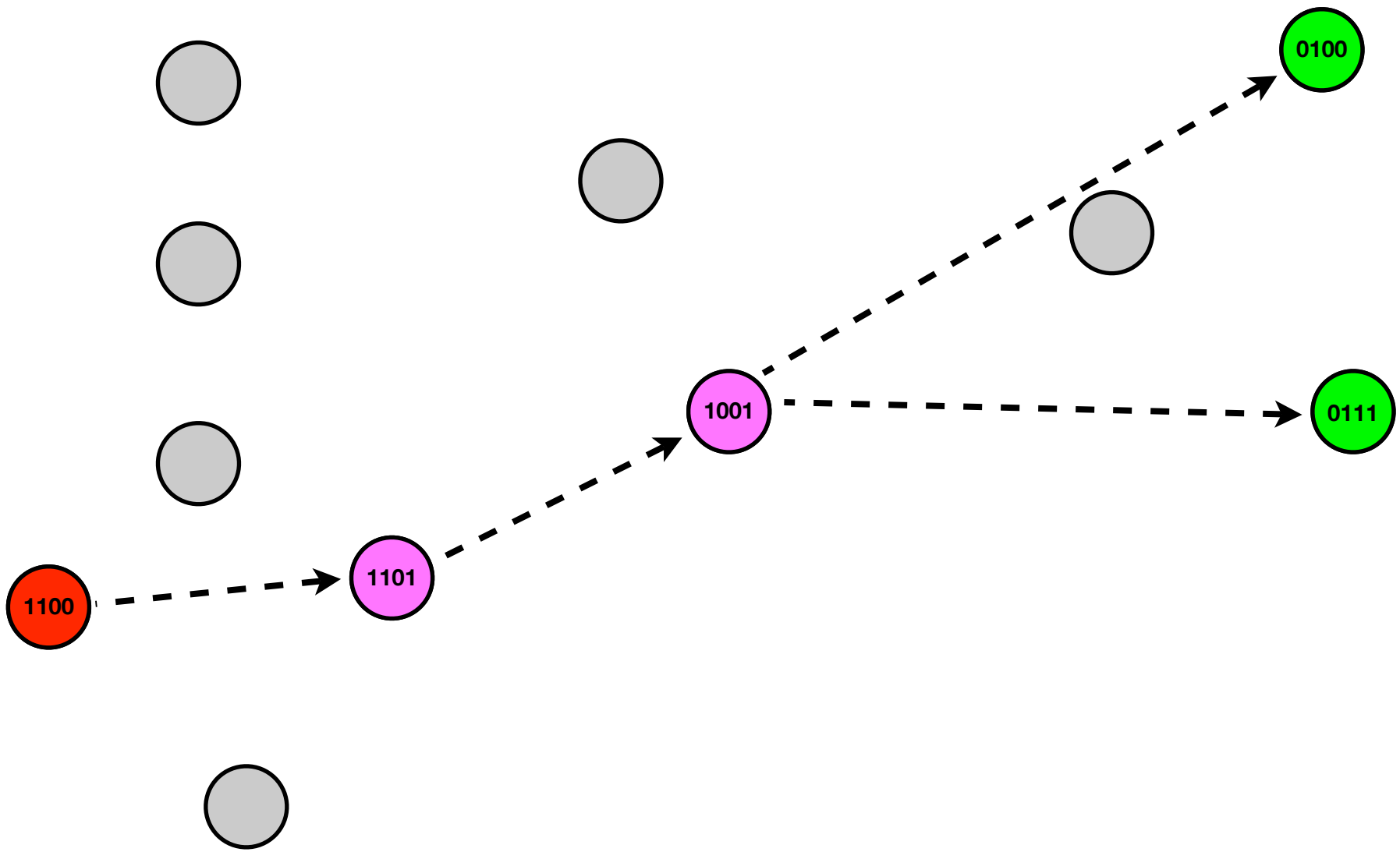



```
(1) forward( msg, key, nextID )
(2)    switch( msg.type )
(3)        JOIN:          if !(msg.group C groups)
(4)                           groups = groups U msg.group;
(5)                           route( msg, msg.group );
(6)                       groups[msg.group].children U msg.source;
(7)                       nextID = null;
```

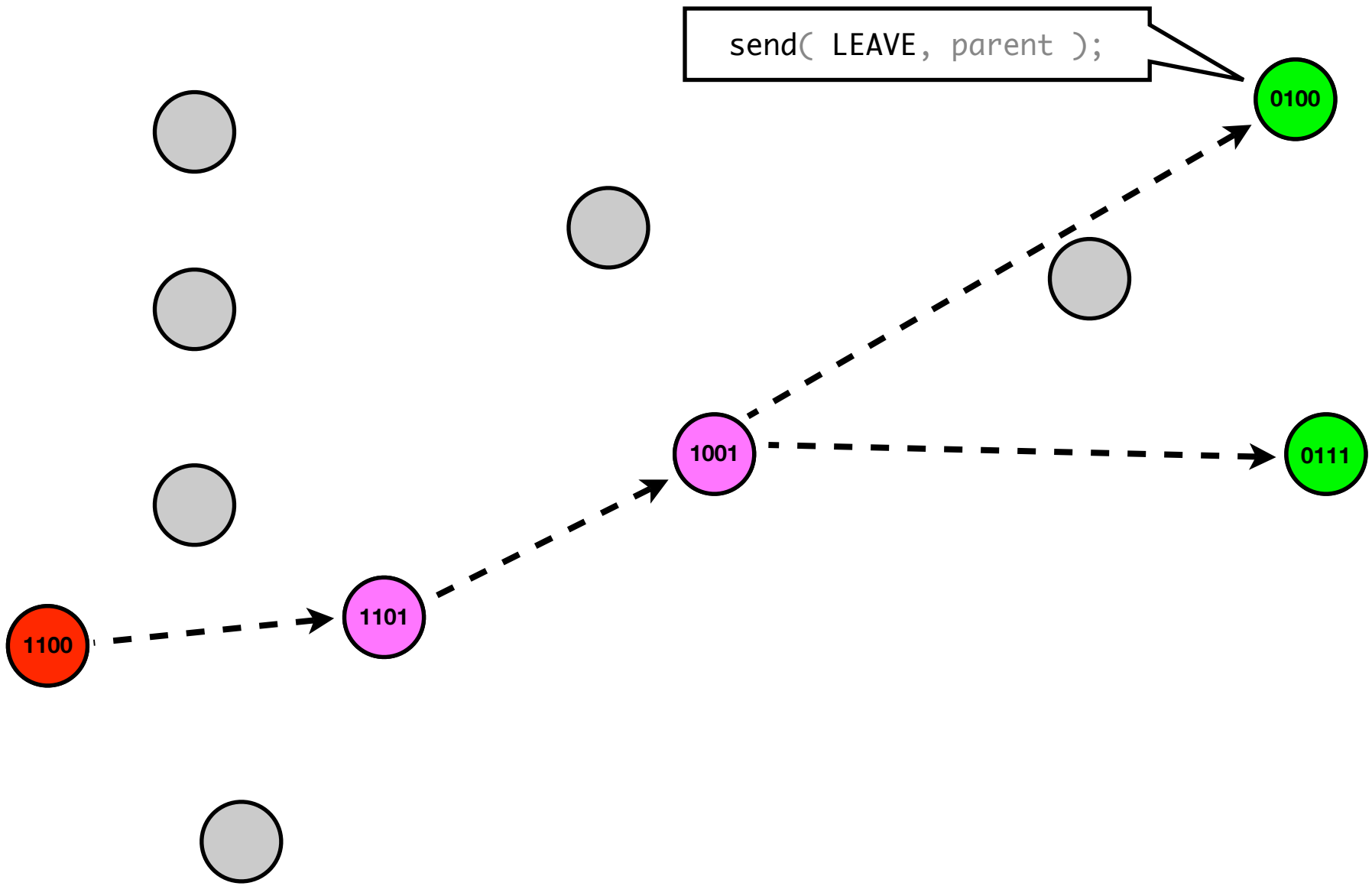
# Joining a Group



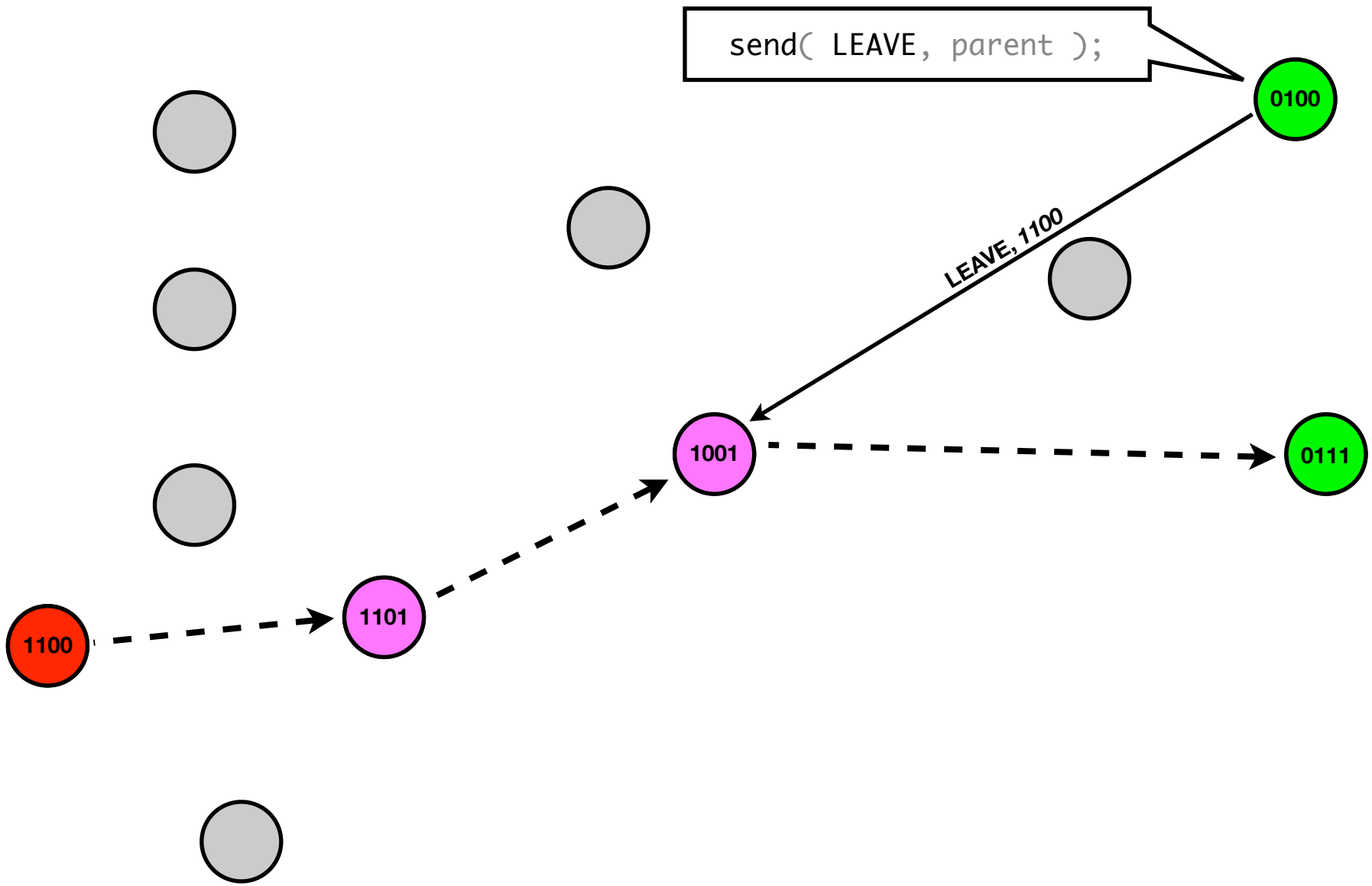
# Leaving a Group



# Leaving a Group

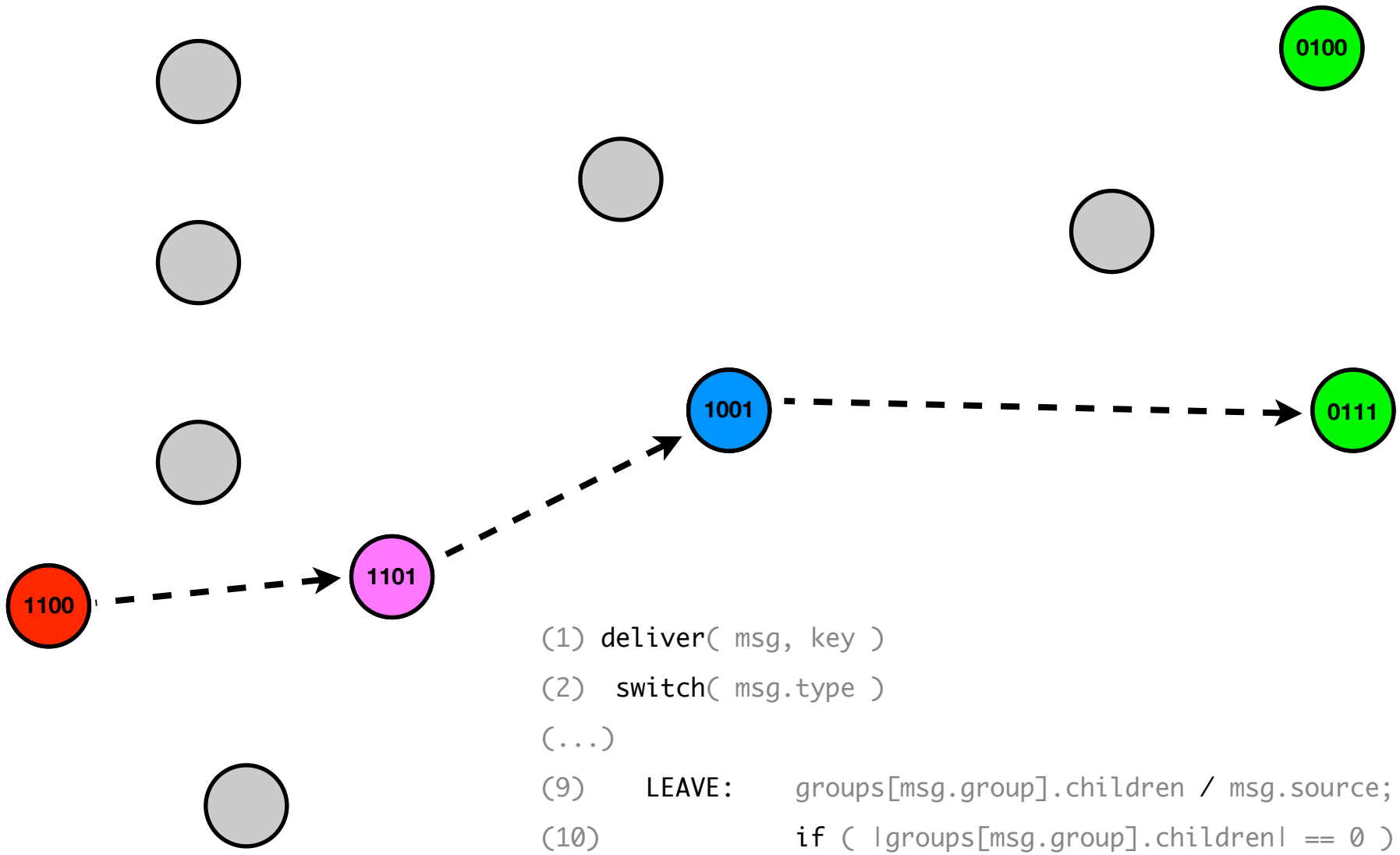


# Leaving a Group



# Scribe Protocol > Membership Management

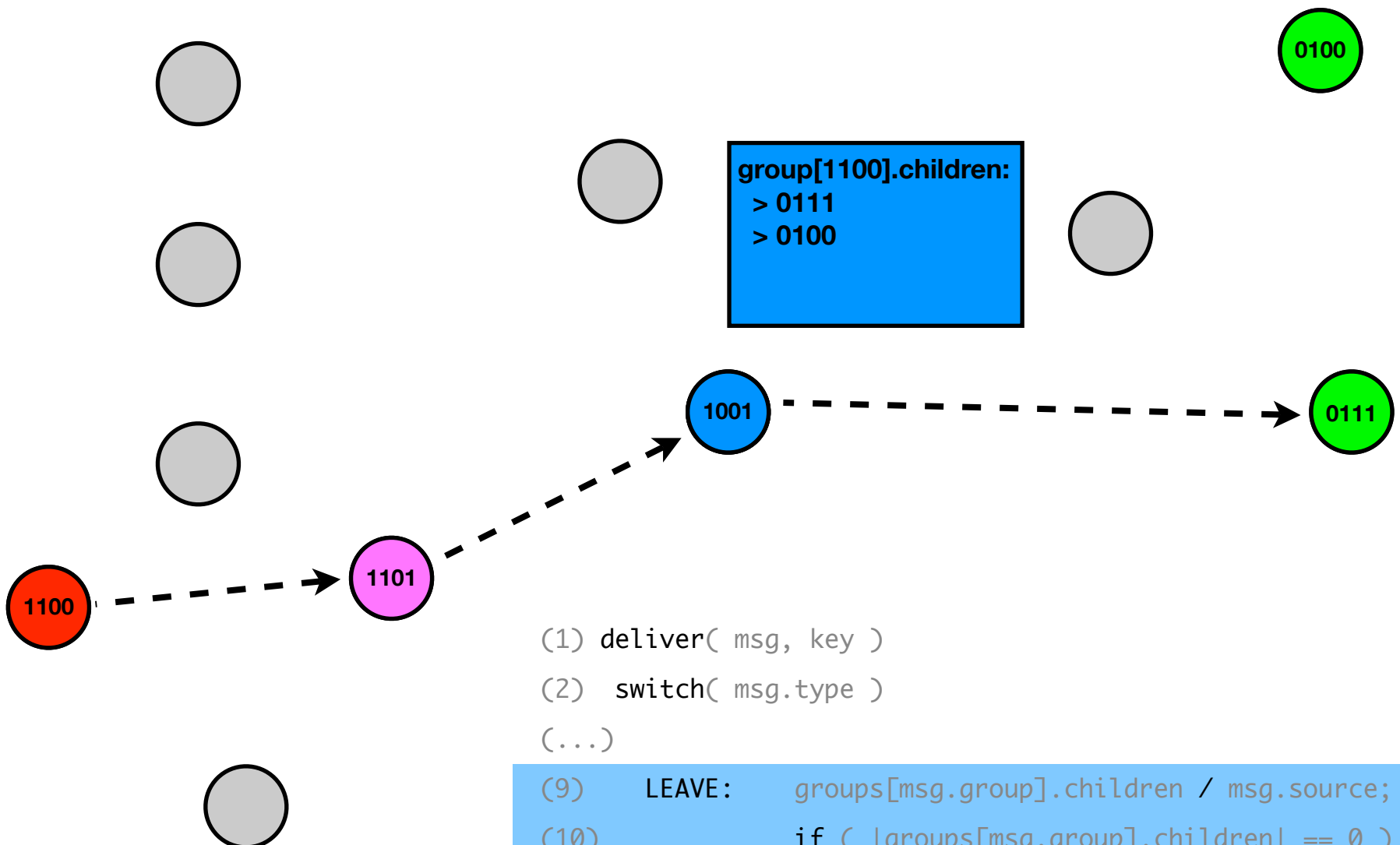
## Leaving a Group




```
(1) deliver( msg, key )
(2)  switch( msg.type )
(...)
(9)    LEAVE:    groups[msg.group].children / msg.source;
(10)             if ( |groups[msg.group].children| == 0 )
(11)                 send( msg, groups[msg.group].parent );
```

# Scribe Protocol > Membership Management

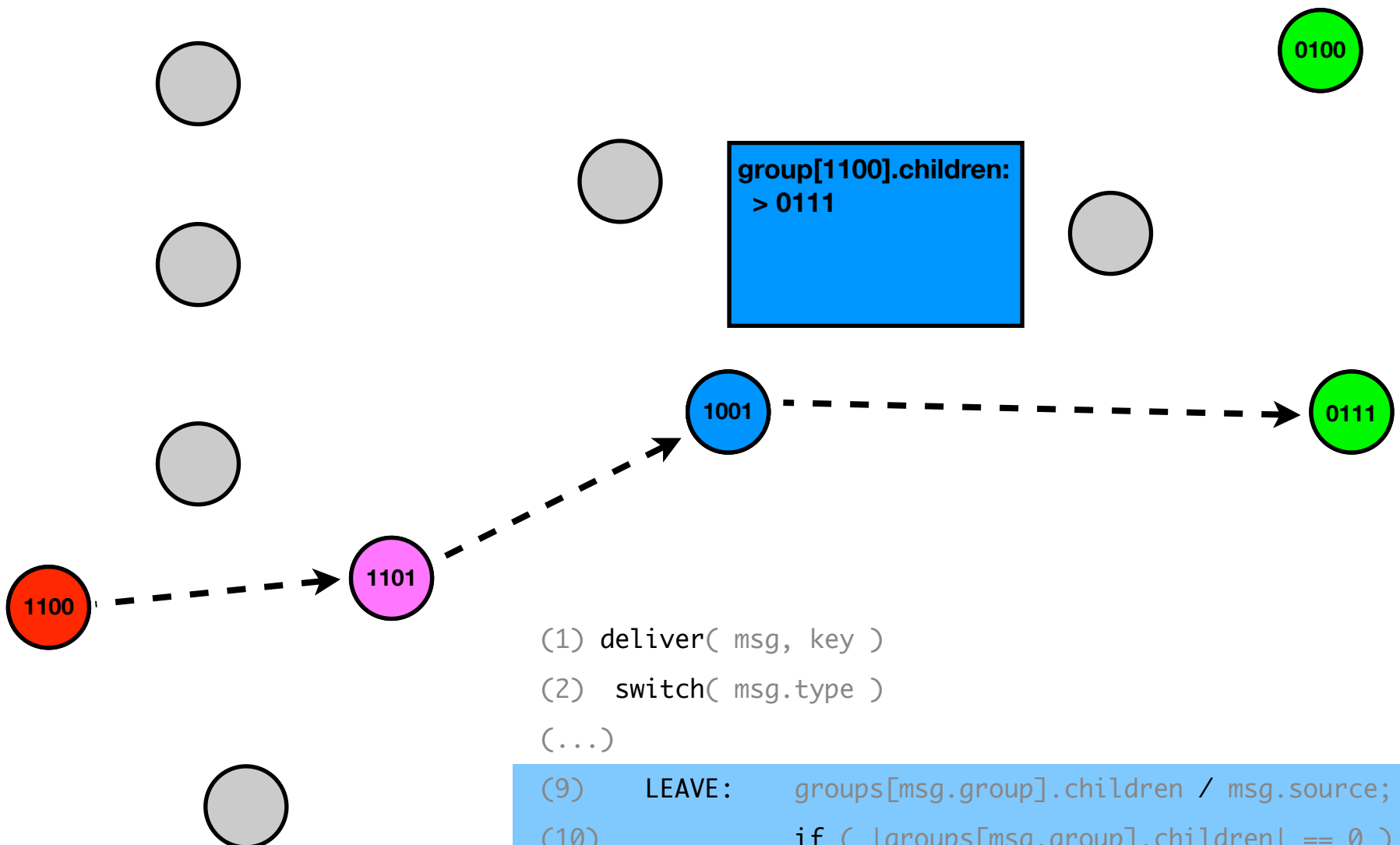
## Leaving a Group




```
(1) deliver( msg, key )
(2)  switch( msg.type )
(...)
(9)    LEAVE:    groups[msg.group].children / msg.source;
(10)             if ( |groups[msg.group].children| == 0 )
(11)                 send( msg, groups[msg.group].parent );
```

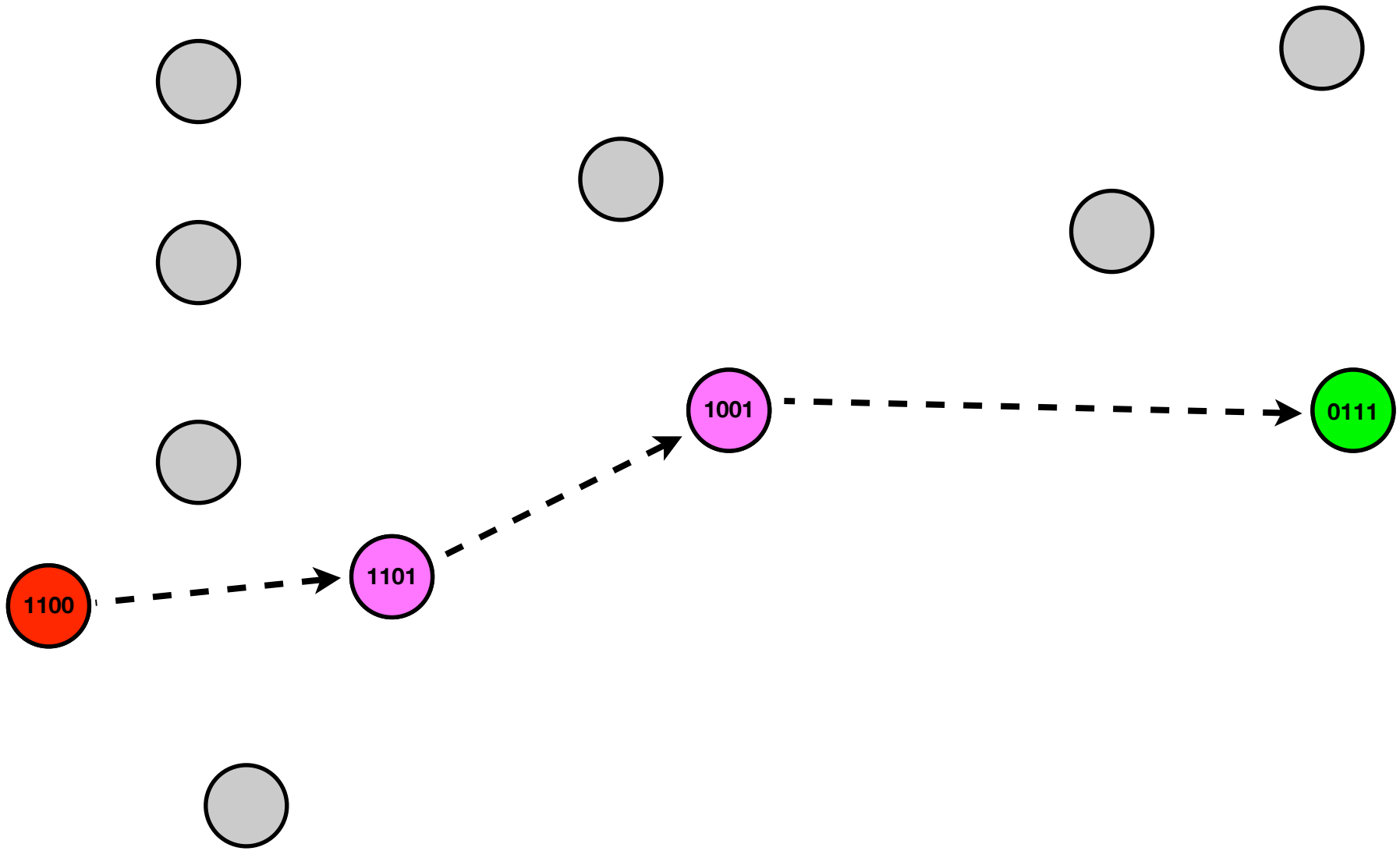
# Scribe Protocol > Membership Management

## Leaving a Group

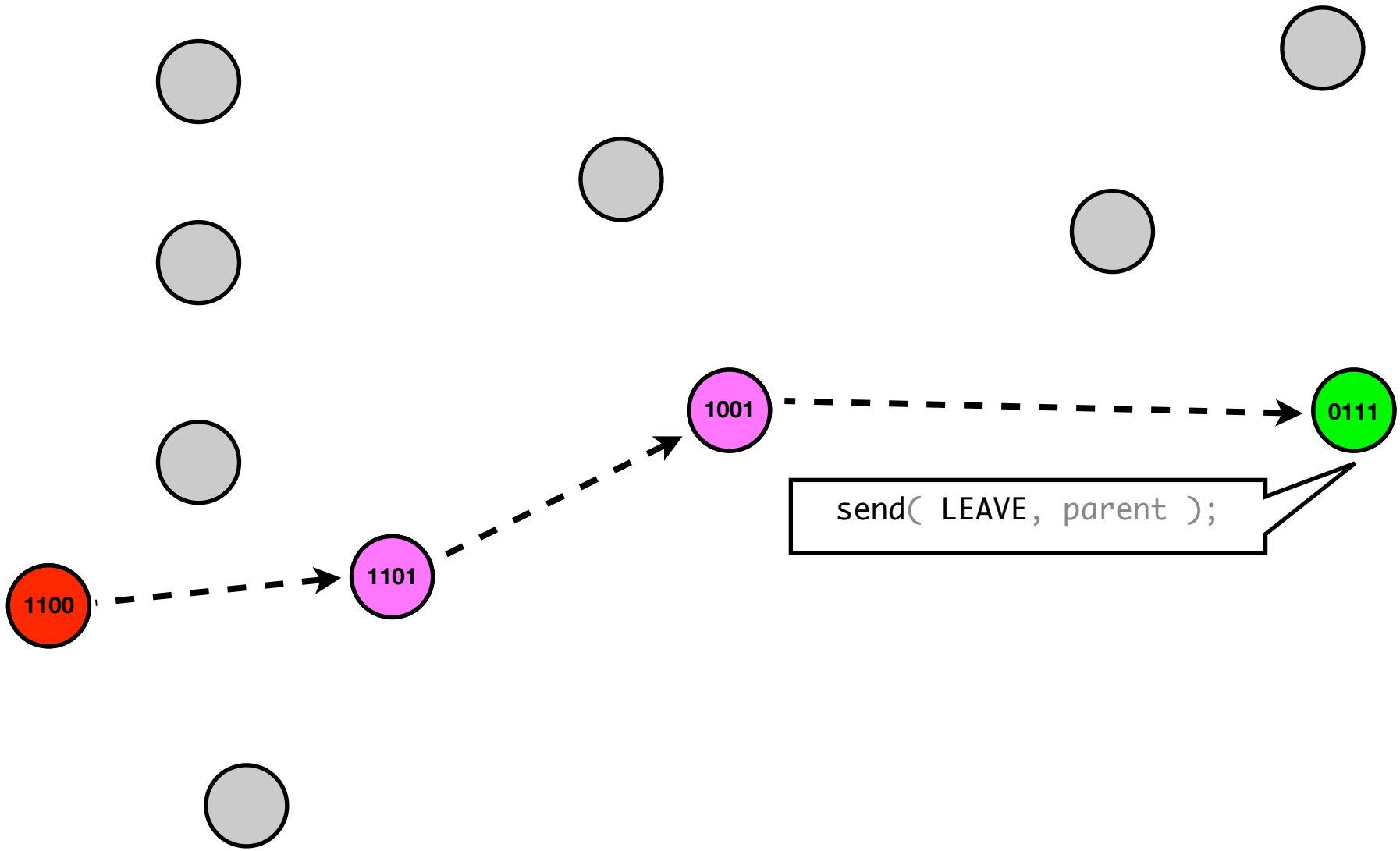



```
(1) deliver( msg, key )
(2)  switch( msg.type )
(...)
(9)    LEAVE:    groups[msg.group].children / msg.source;
(10)             if ( |groups[msg.group].children| == 0 )
(11)                 send( msg, groups[msg.group].parent );
```

# Leaving a Group

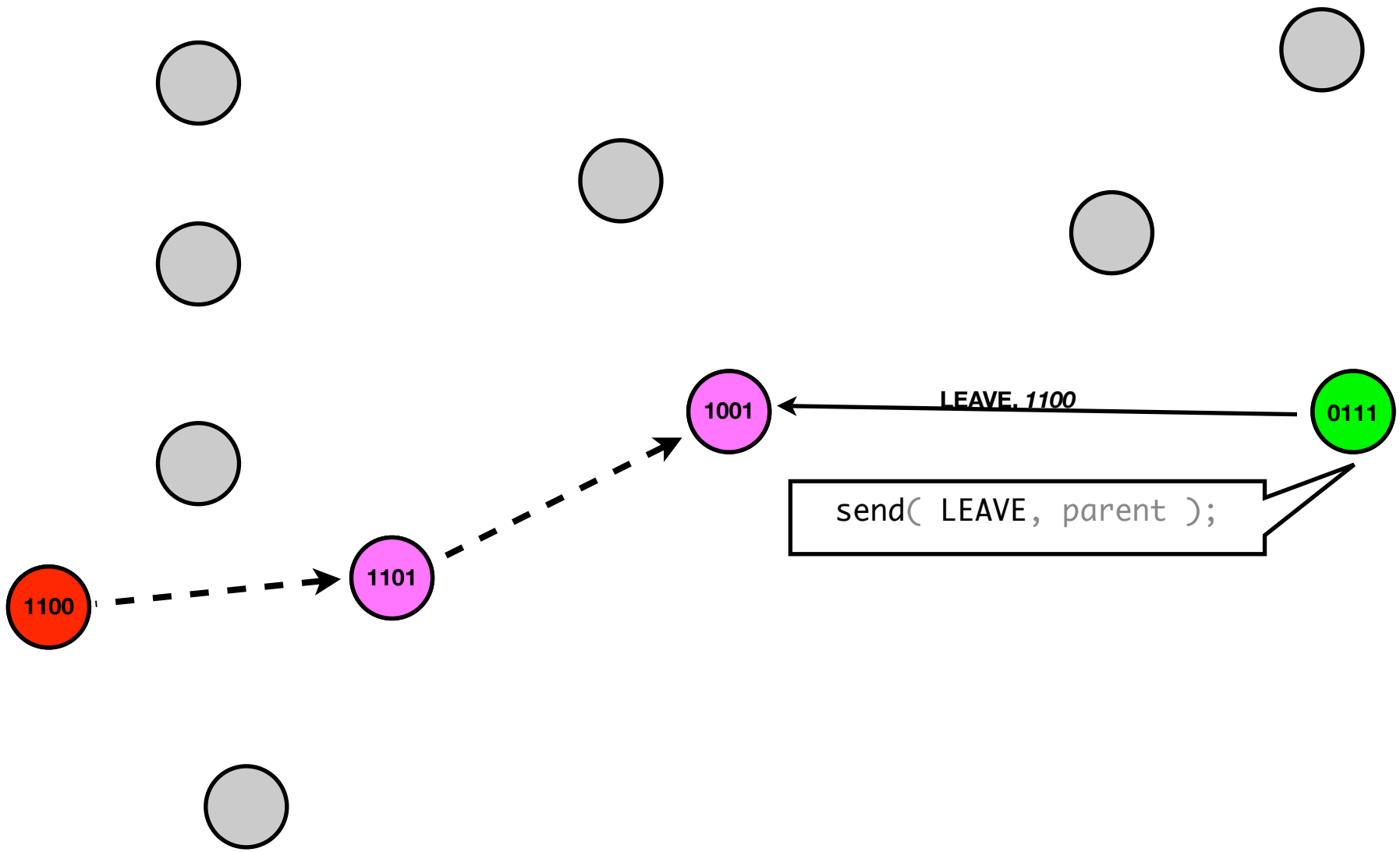


# Leaving a Group

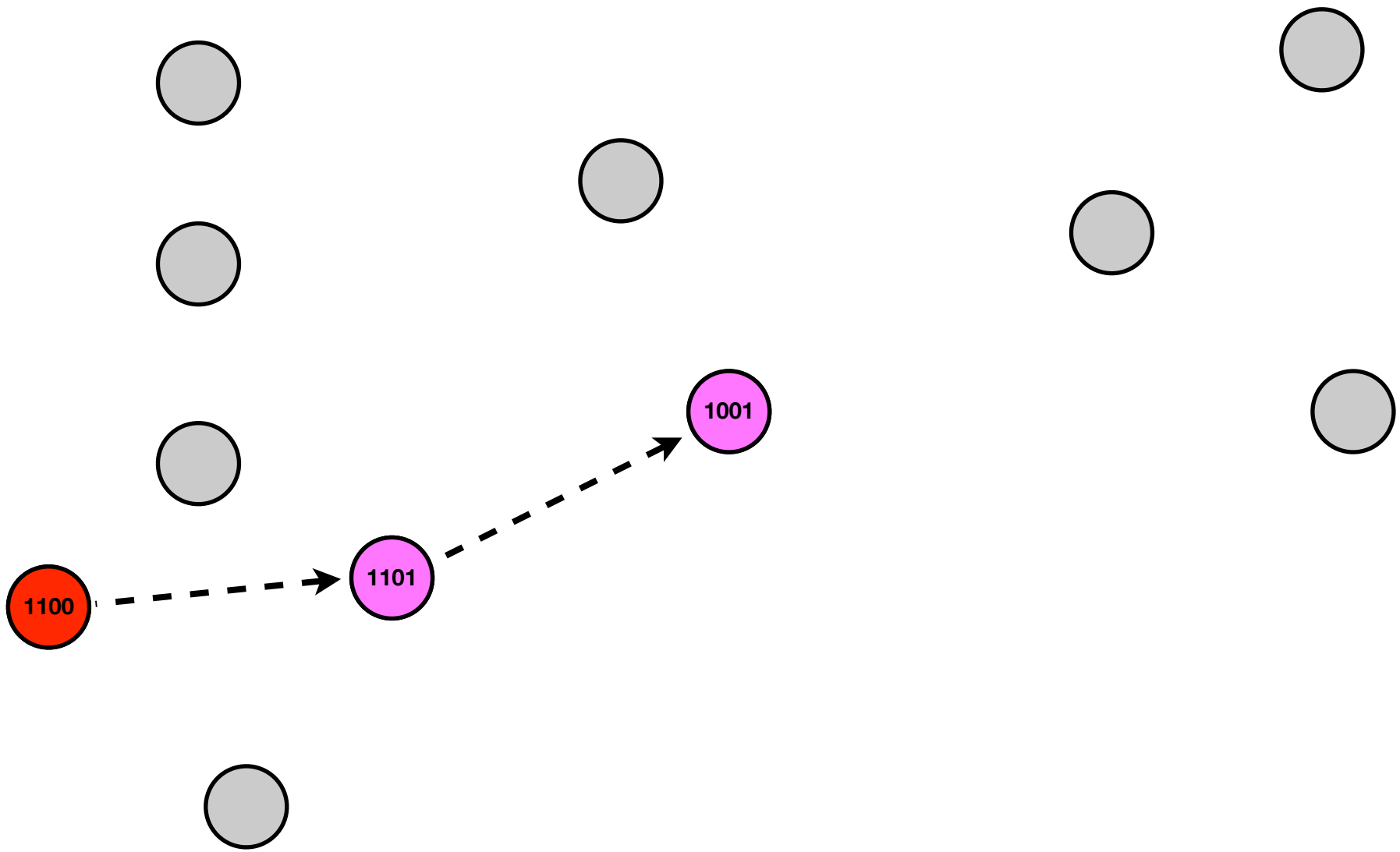


# Scribe Protocol > Membership Management

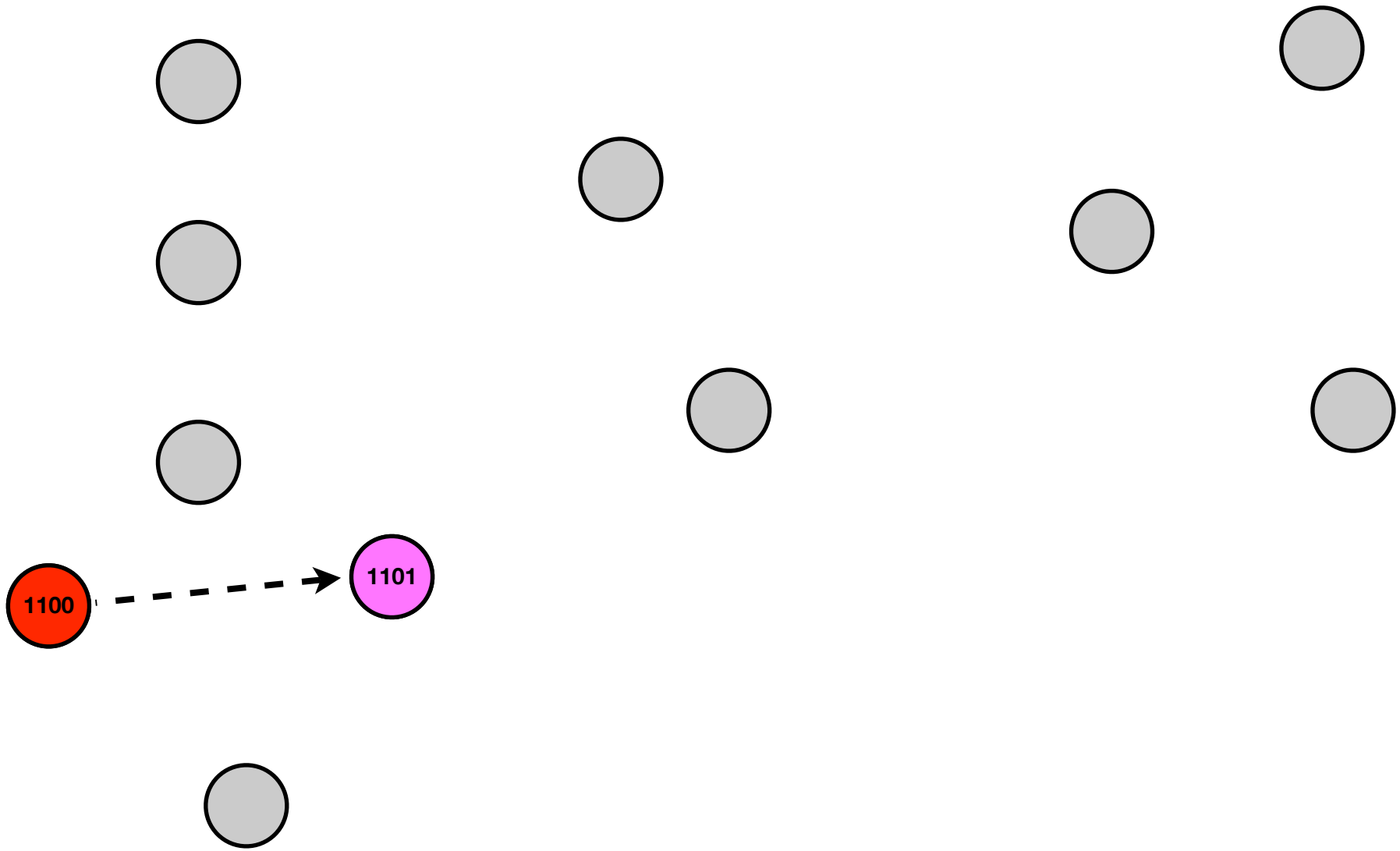
## Leaving a Group



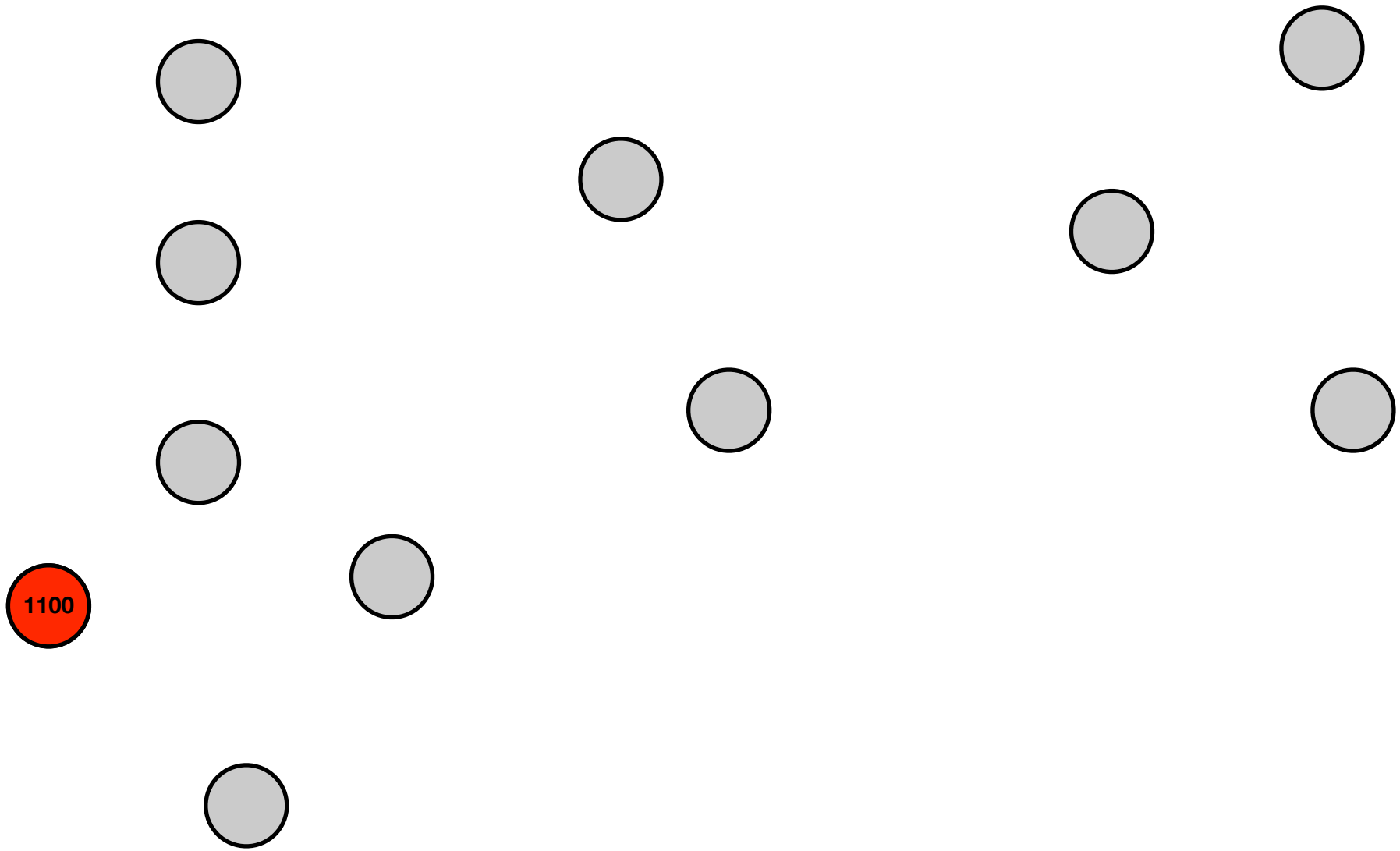
# Leaving a Group



# Leaving a Group



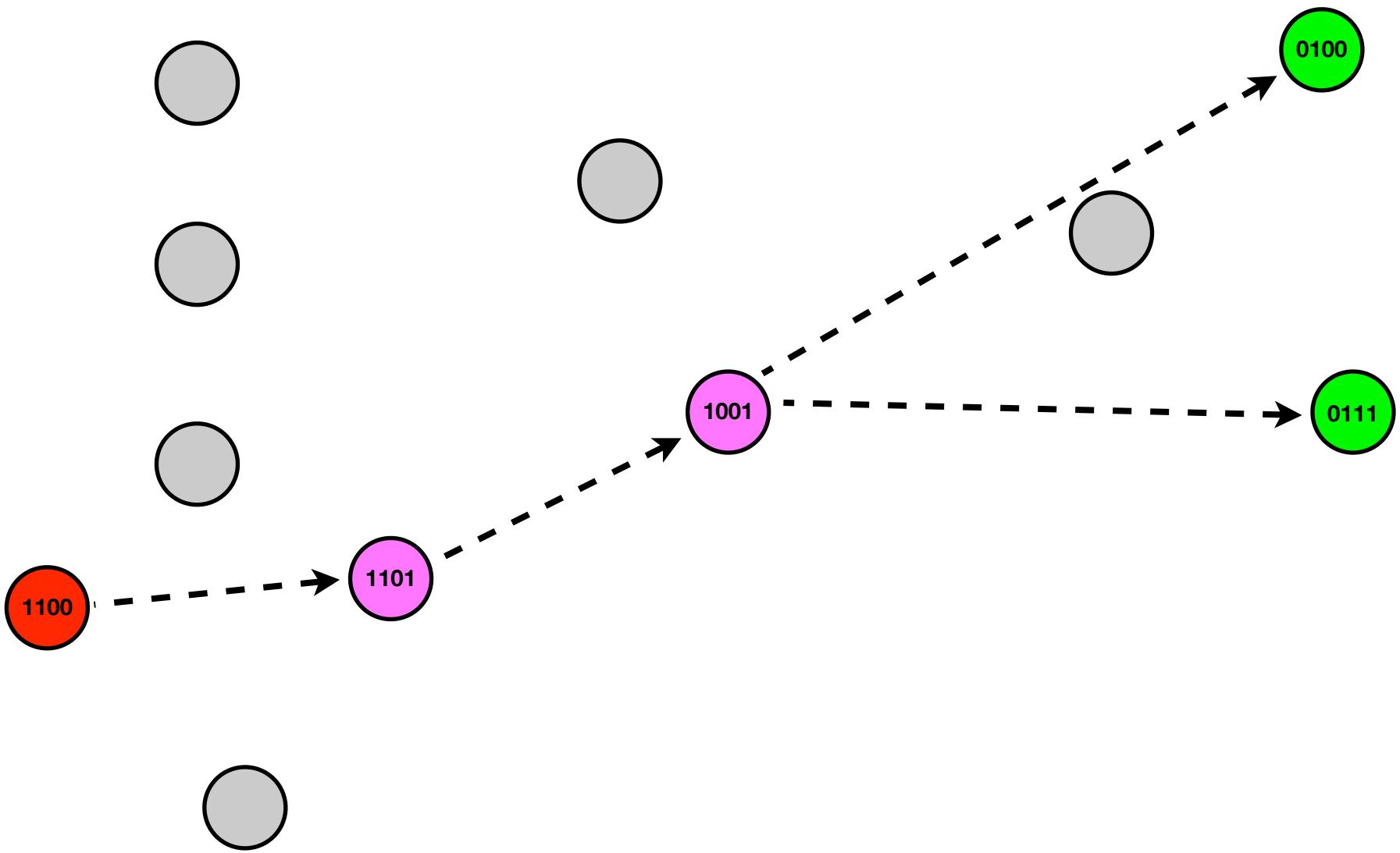
# Leaving a Group



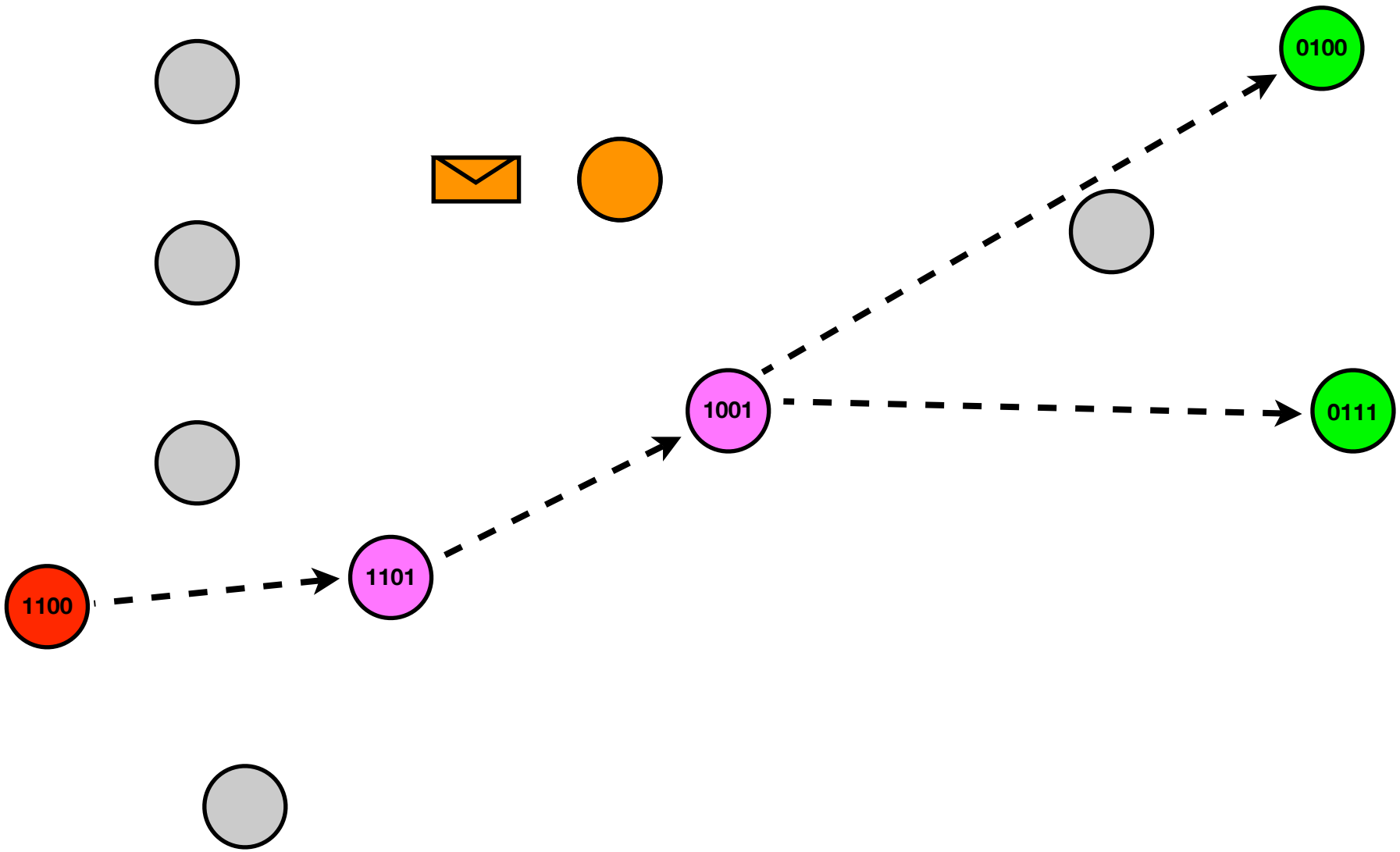
# Scribe Protocol
## Multicast Message Dissemination

- Multicast sources use Pastry to locate the rendezvous point of a group:
  - Call route( MULTICAST, *groupID* ) the first time and ask it to return its IP address
  - They now cache the IP address for subsequent multicasts to avoid routing the requests through Pastry:
    - To multicast some message, they use send( MULTICAST, *rendezVous* )
    - The message is sent directly to the rendezvous point
  - The rendezvous point performs access control functions and then disseminates the message to its children that belong to the group
    - The children also send the message to their children in the group and so on

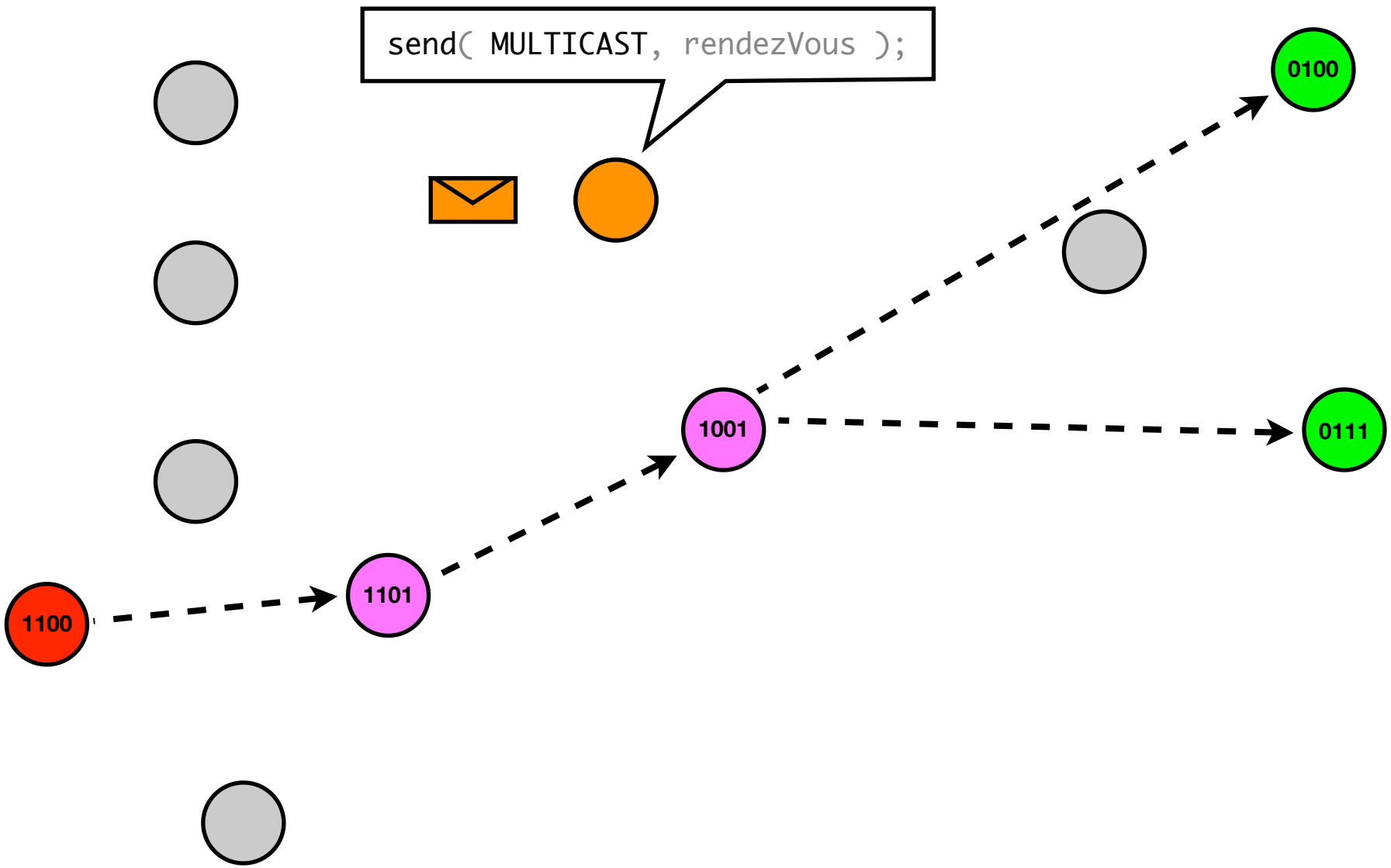
# Sending a Multicast Message



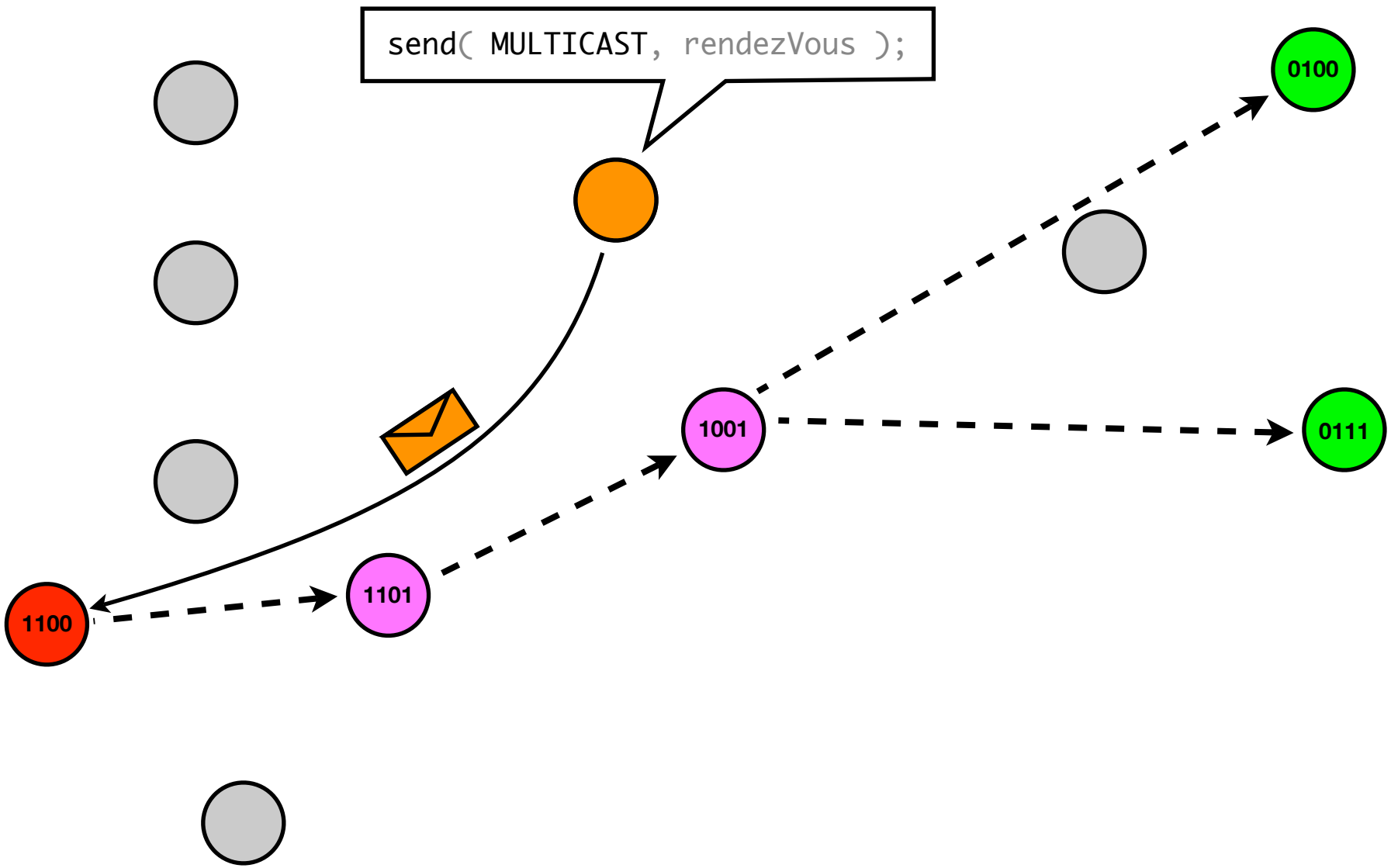
# Sending a Multicast Message



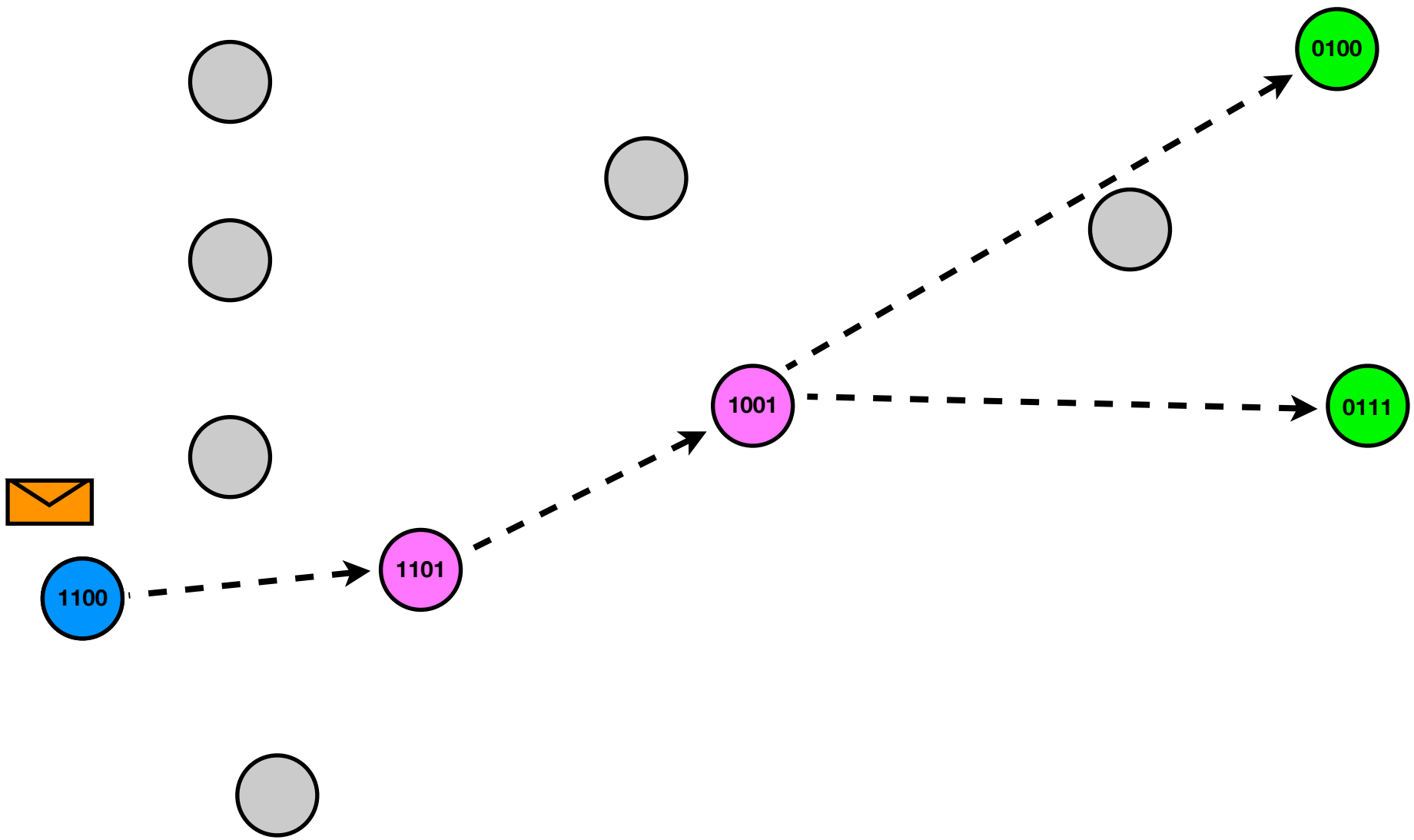
# Sending a Multicast Message



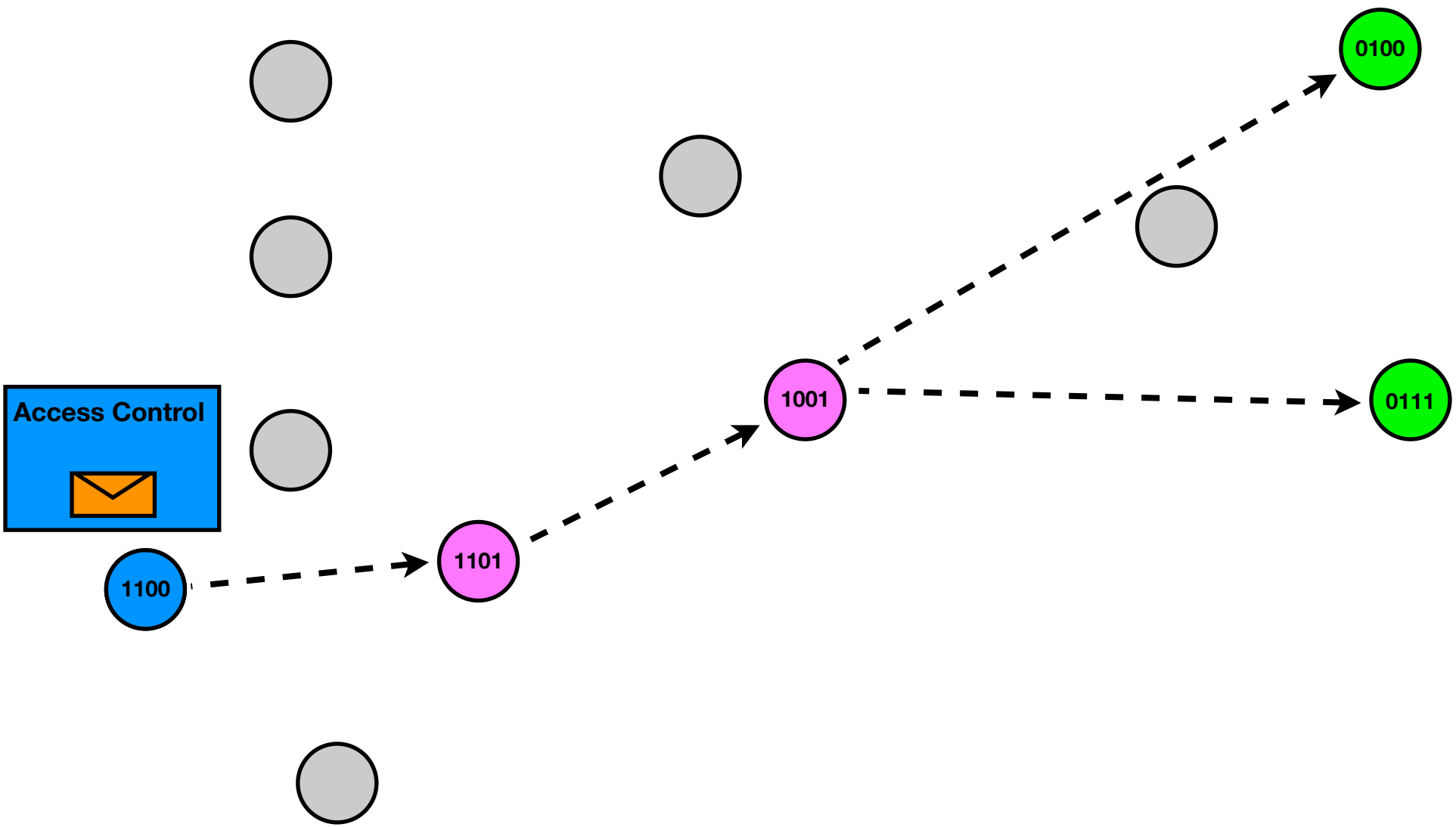
# Sending a Multicast Message



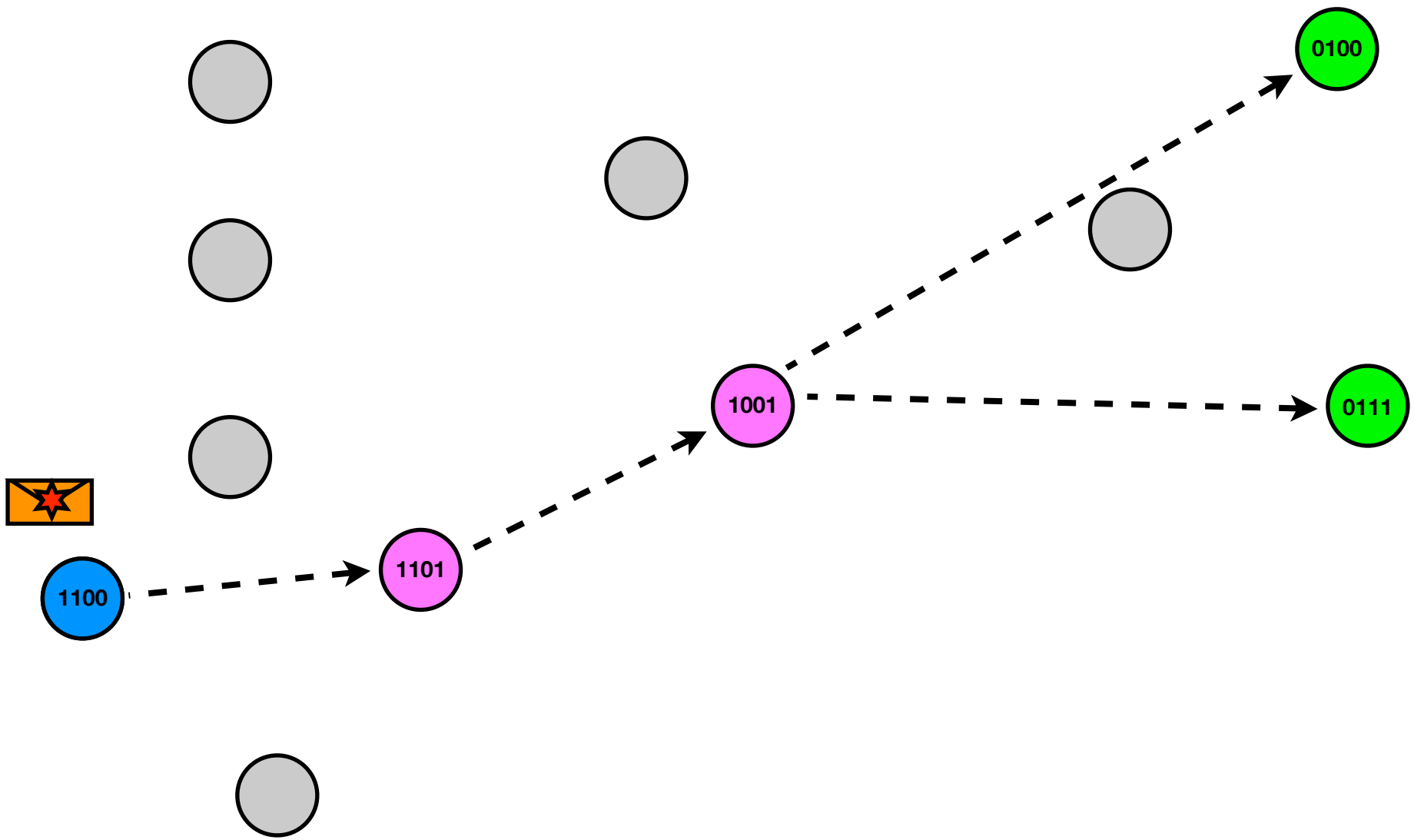
# Sending a Multicast Message



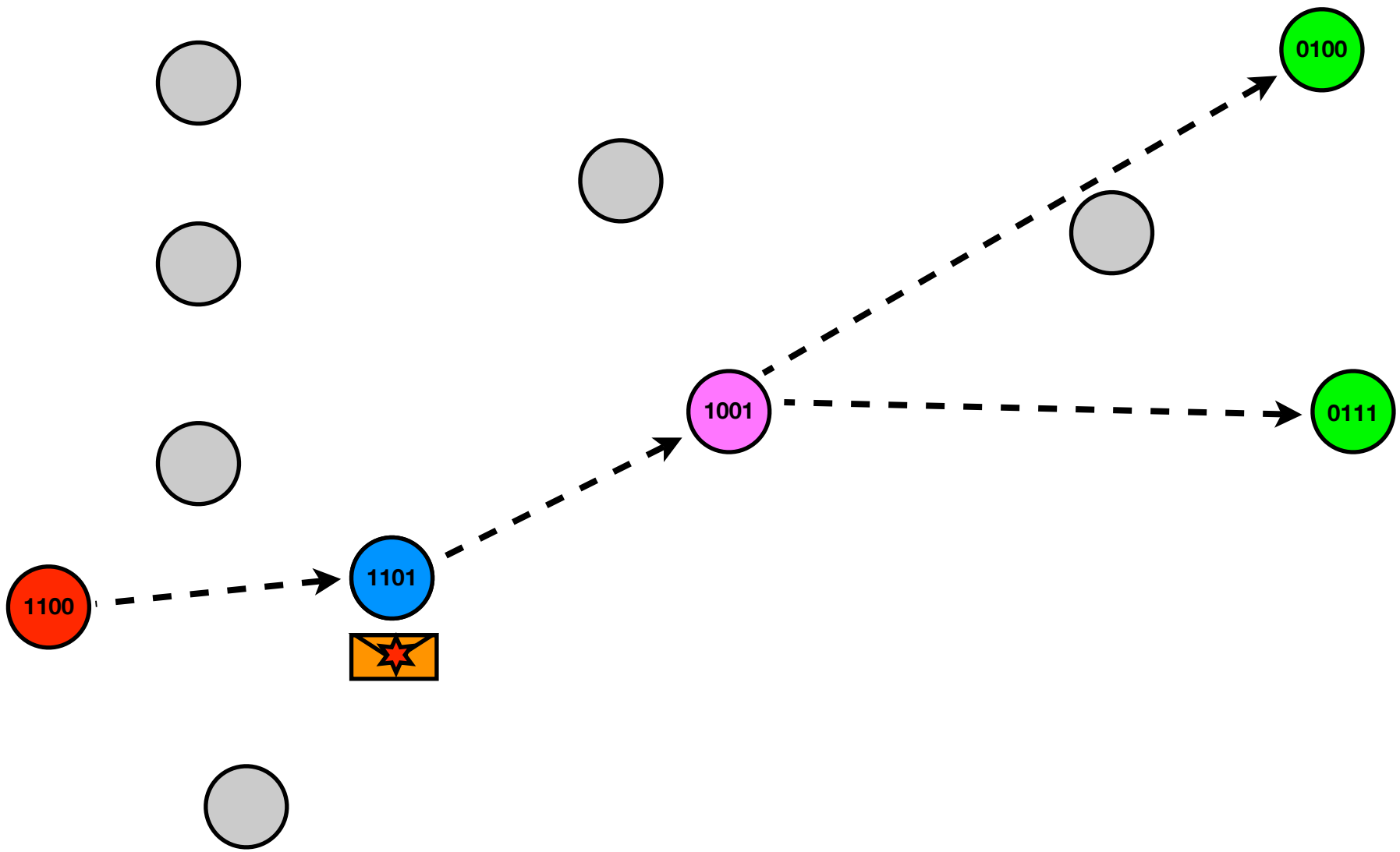
# Sending a Multicast Message



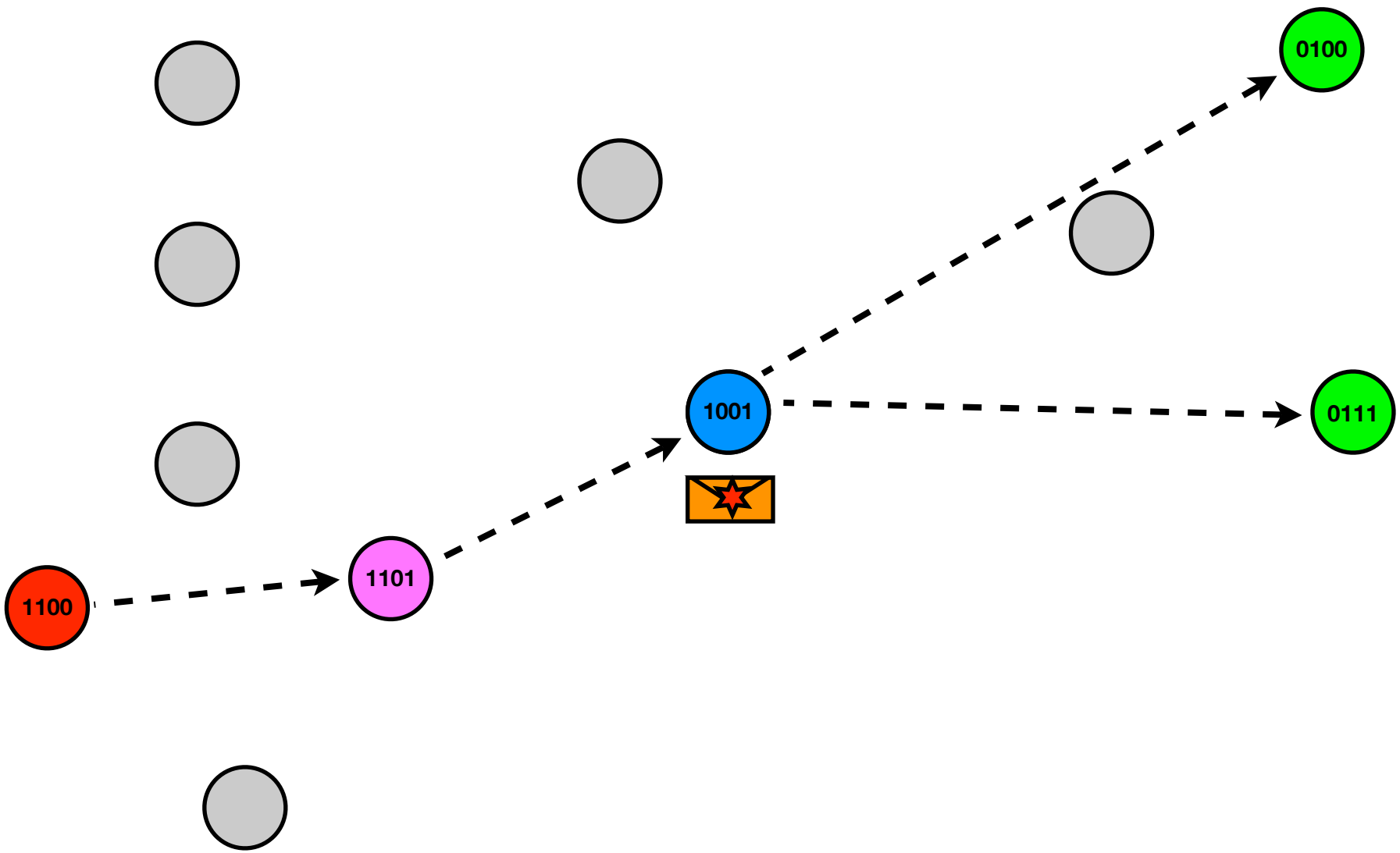
# Sending a Multicast Message



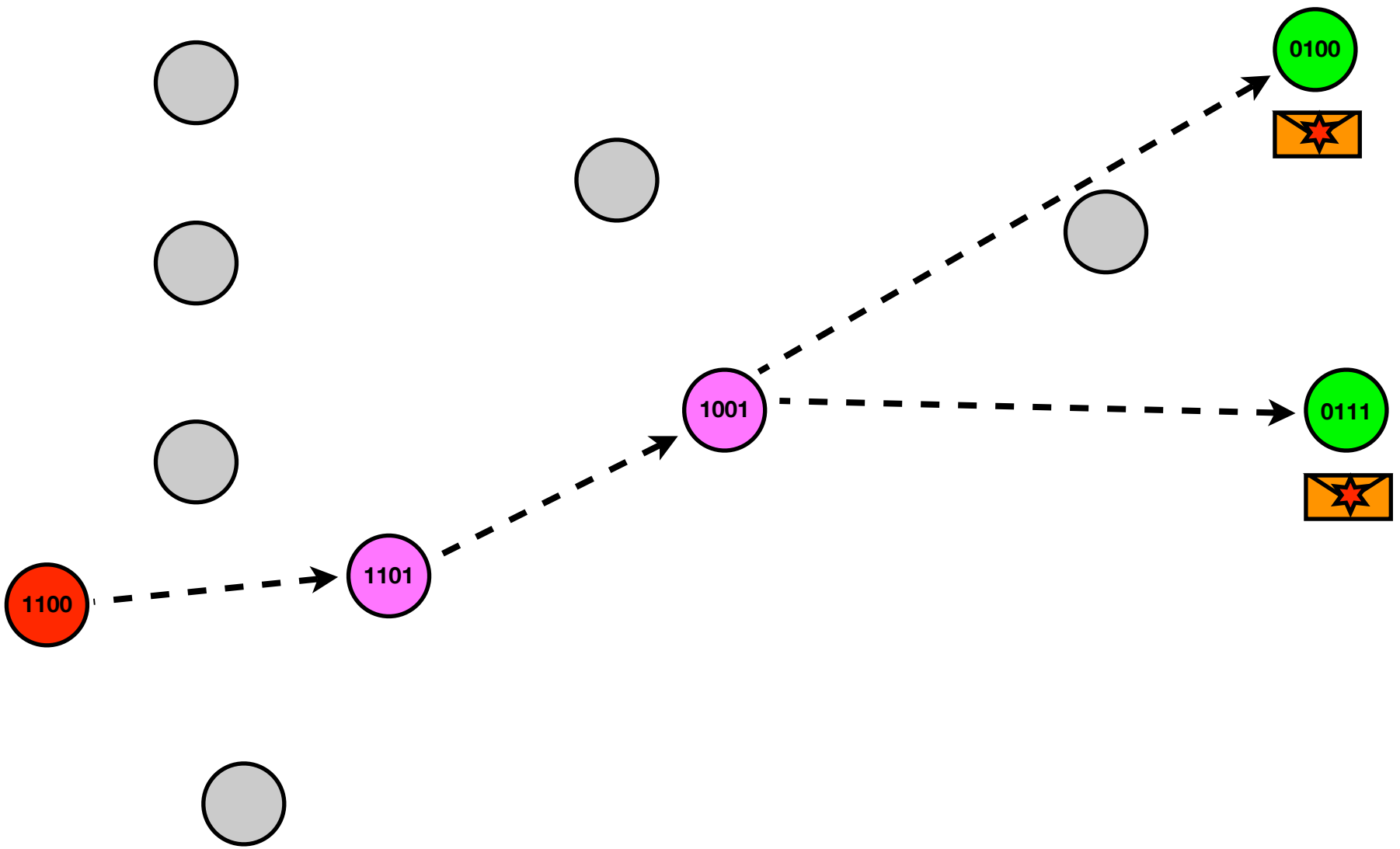
# Sending a Multicast Message



# Sending a Multicast Message



# Sending a Multicast Message



# Scribe Protocol
## Reliability

- Applications using group multicast services may have diverse reliability requirements
  - e.g. reliable and ordered delivery of messages, best-effort delivery
- Scribe offers only best-effort guarantees
  - Uses TCP to disseminate messages reliably from parents to their children in the multicast tree and for flow control
  - Uses Pastry to repair the multicast tree when a forwarder fails
- Provides a framework for applications to implement stronger reliability guarantees

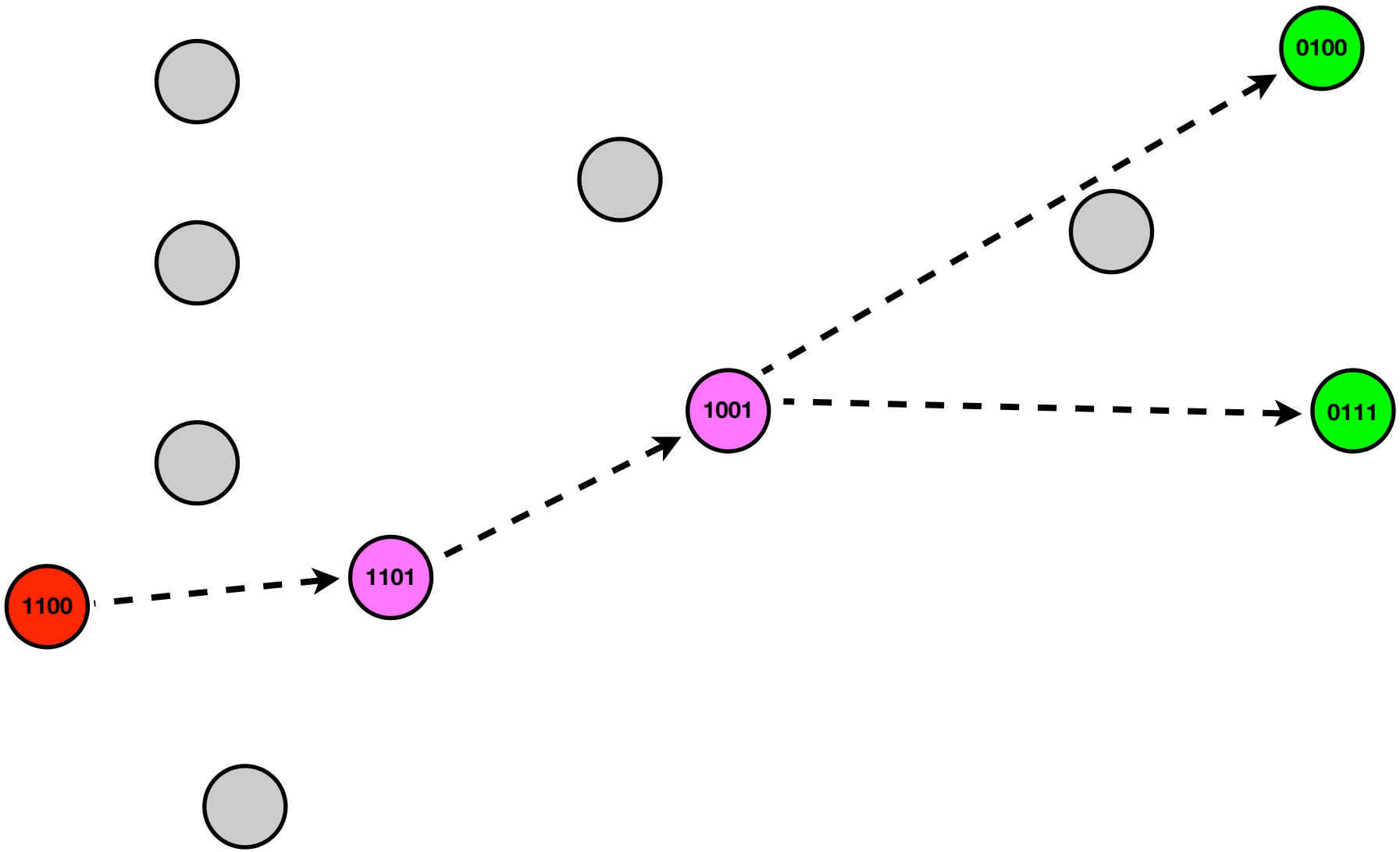
# Scribe Protocol > Reliability
## Repairing the Multicast Tree

- Each non-leaf node sends a heartbeat message periodically to its children
  - Multicast messages serve as implicit ‘alive’ signals, avoiding the need for explicit heartbeats in many cases

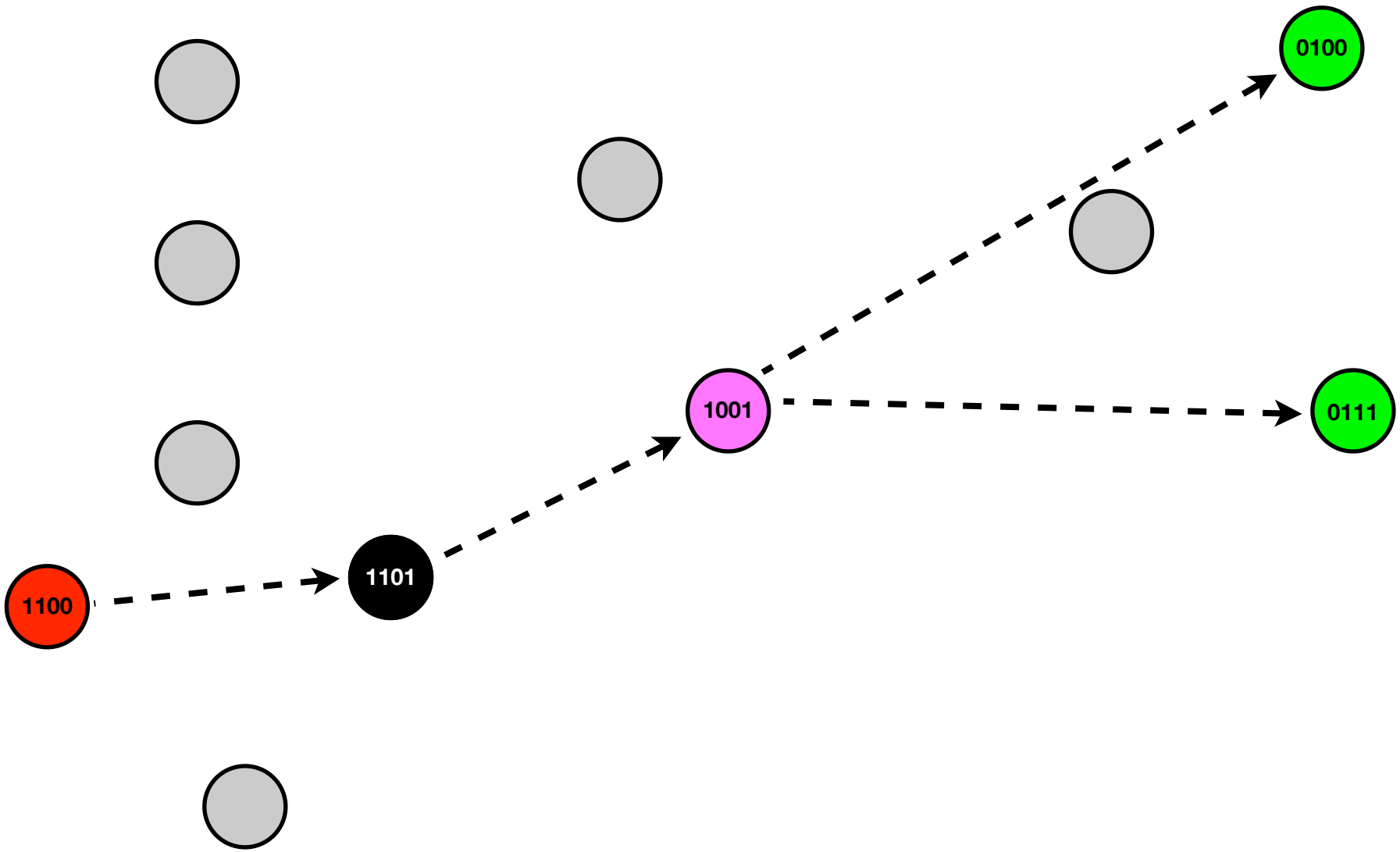
- A child suspects its parent is faulty when he fails to receive heartbeat messages
  - Upon detection of a failed parent, the node asks Pastry to route a JOIN message to *groupID* again
  - Pastry will route the message using an alternative path (i.e. to a new parent), thus repairing the multicast tree

- Children table entries are discarded unless they are periodically refreshed by an explicit message from the child

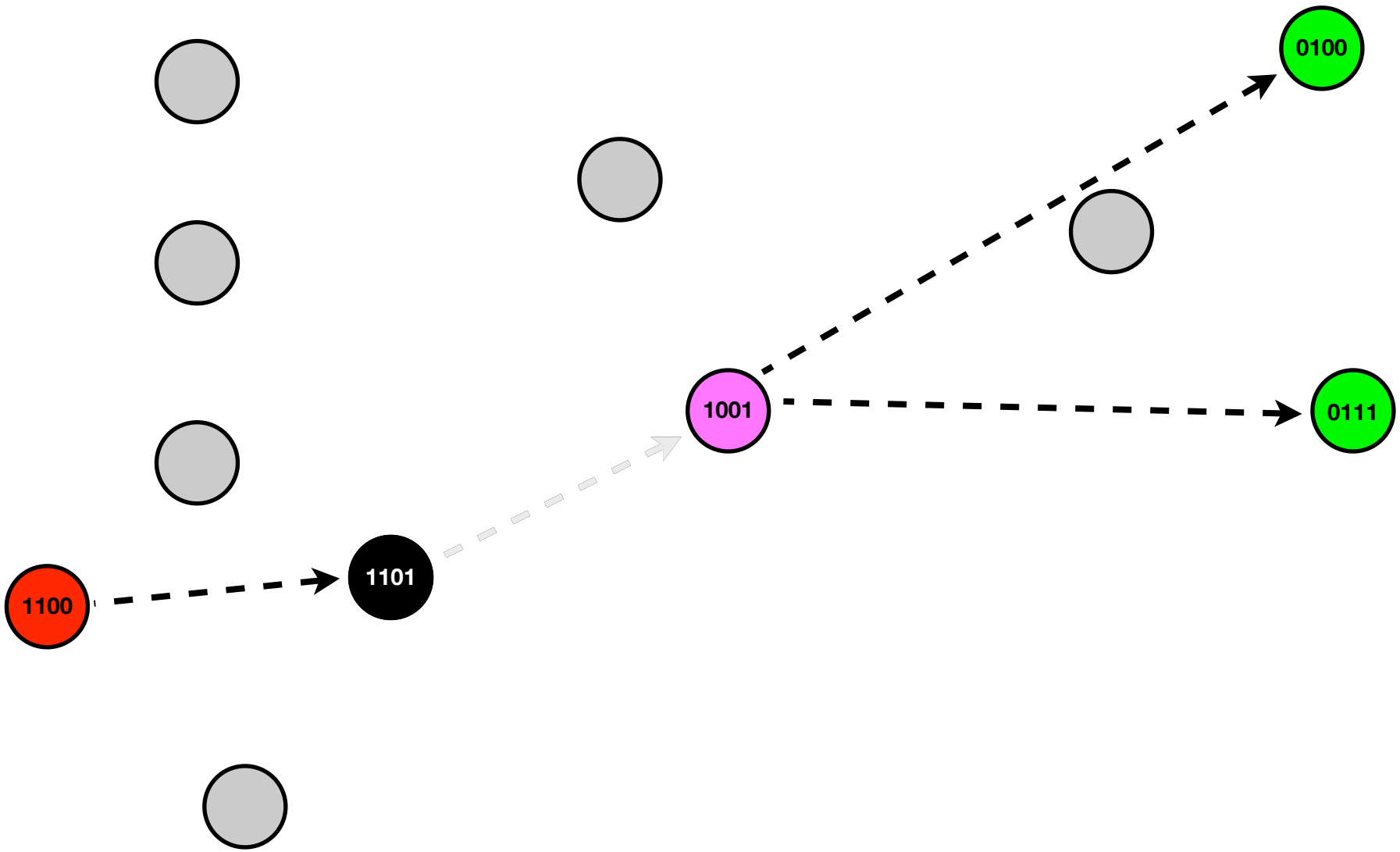
# Repairing the Multicast Tree



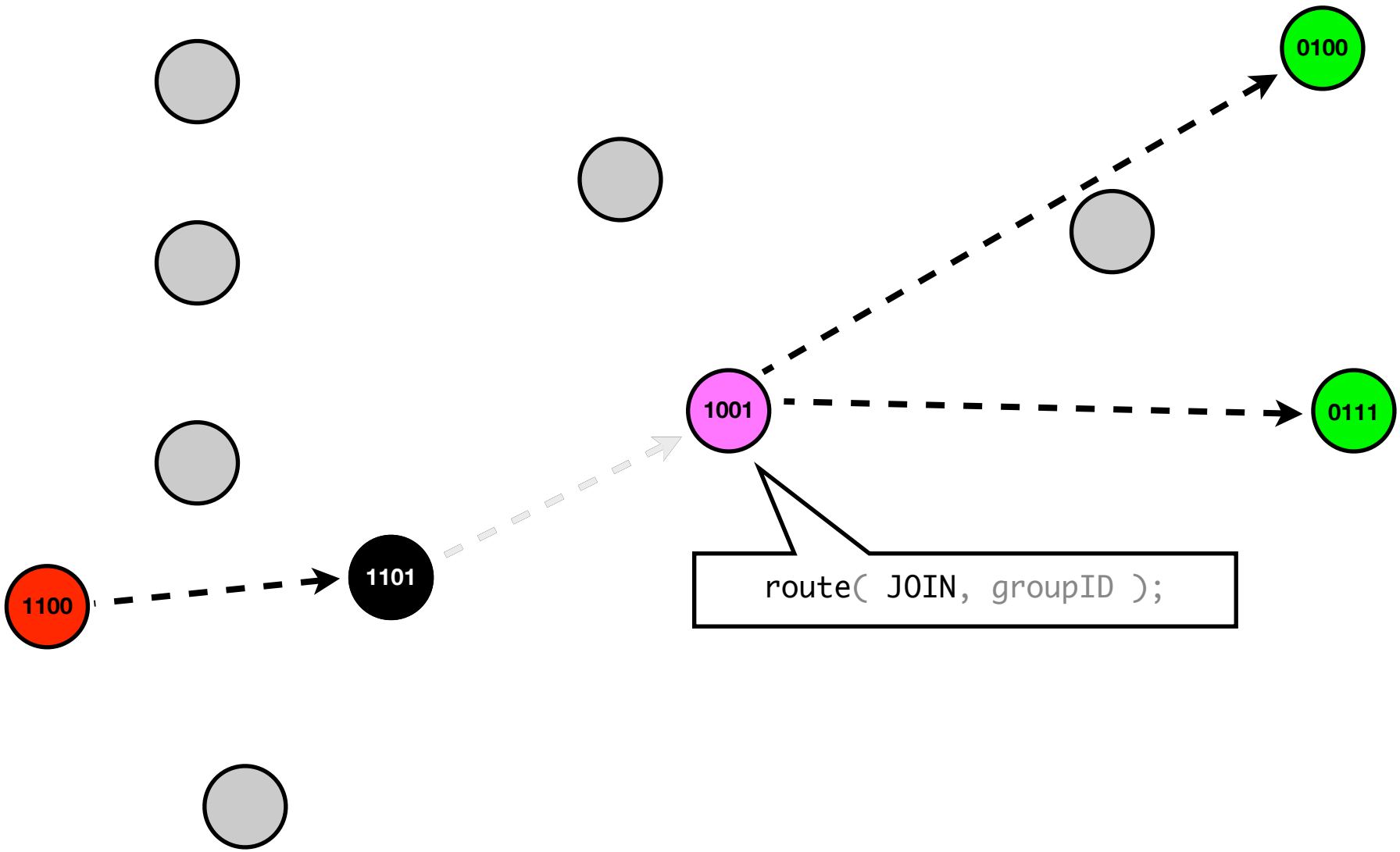
# Repairing the Multicast Tree



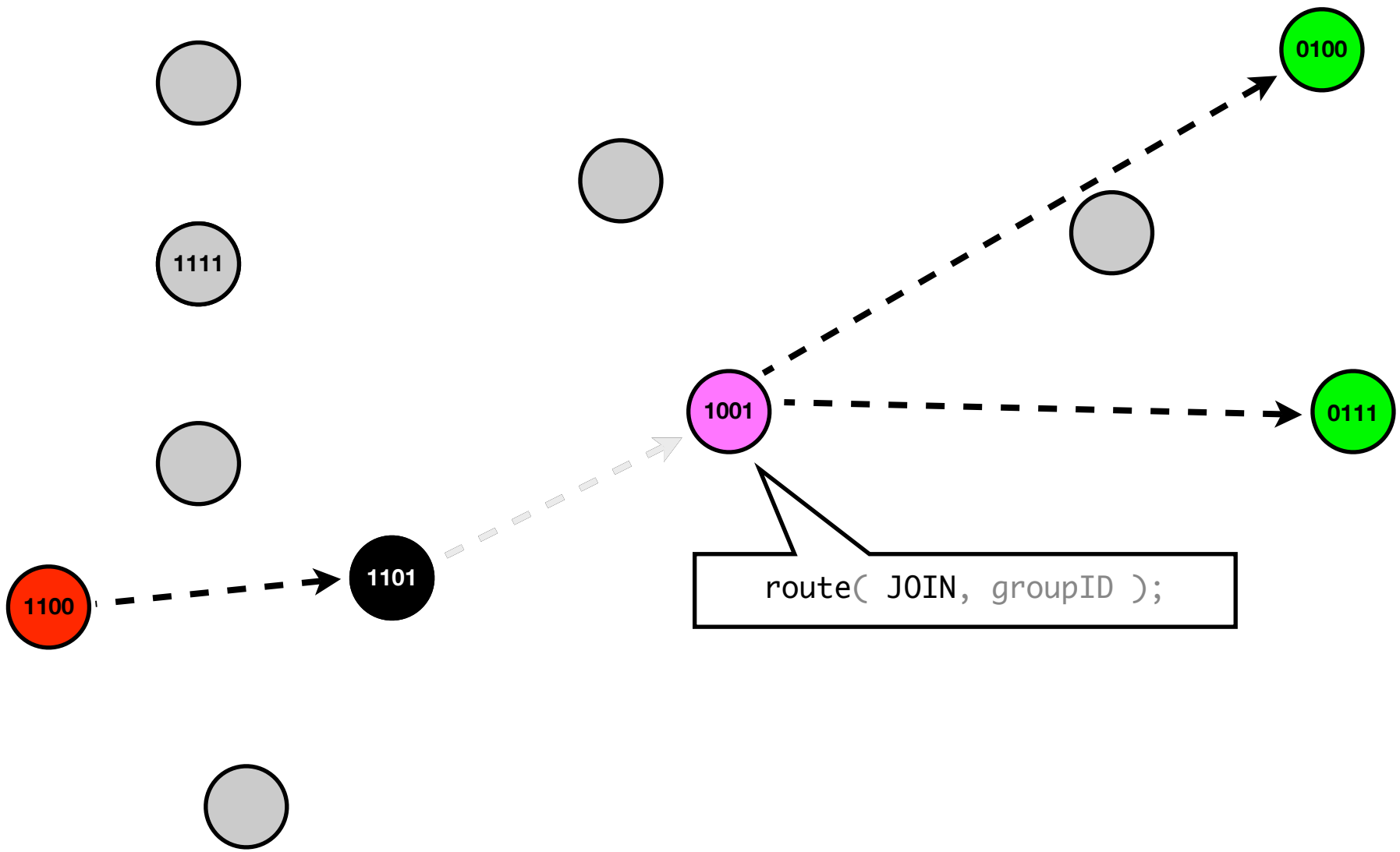
# Repairing the Multicast Tree



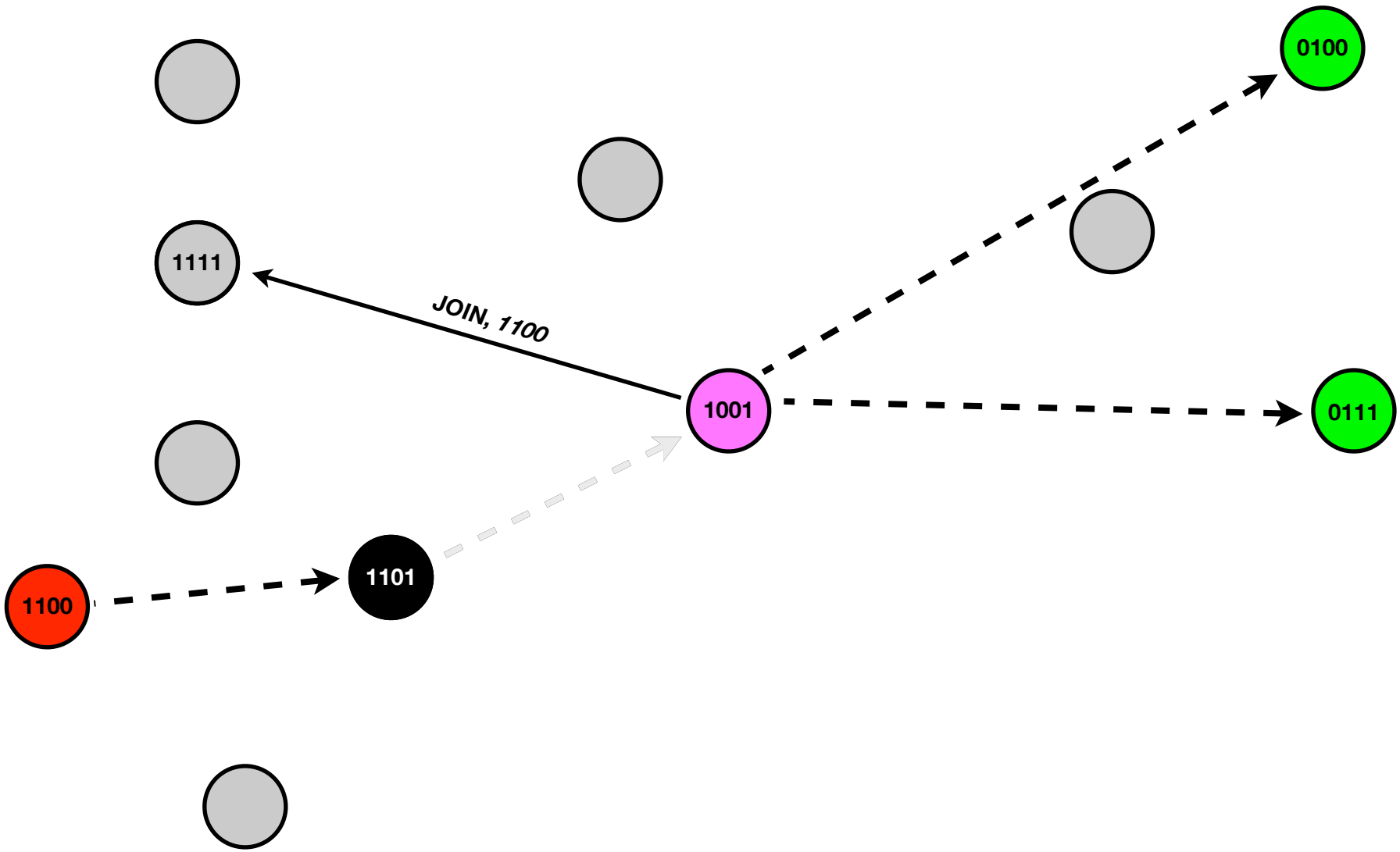
# Repairing the Multicast Tree



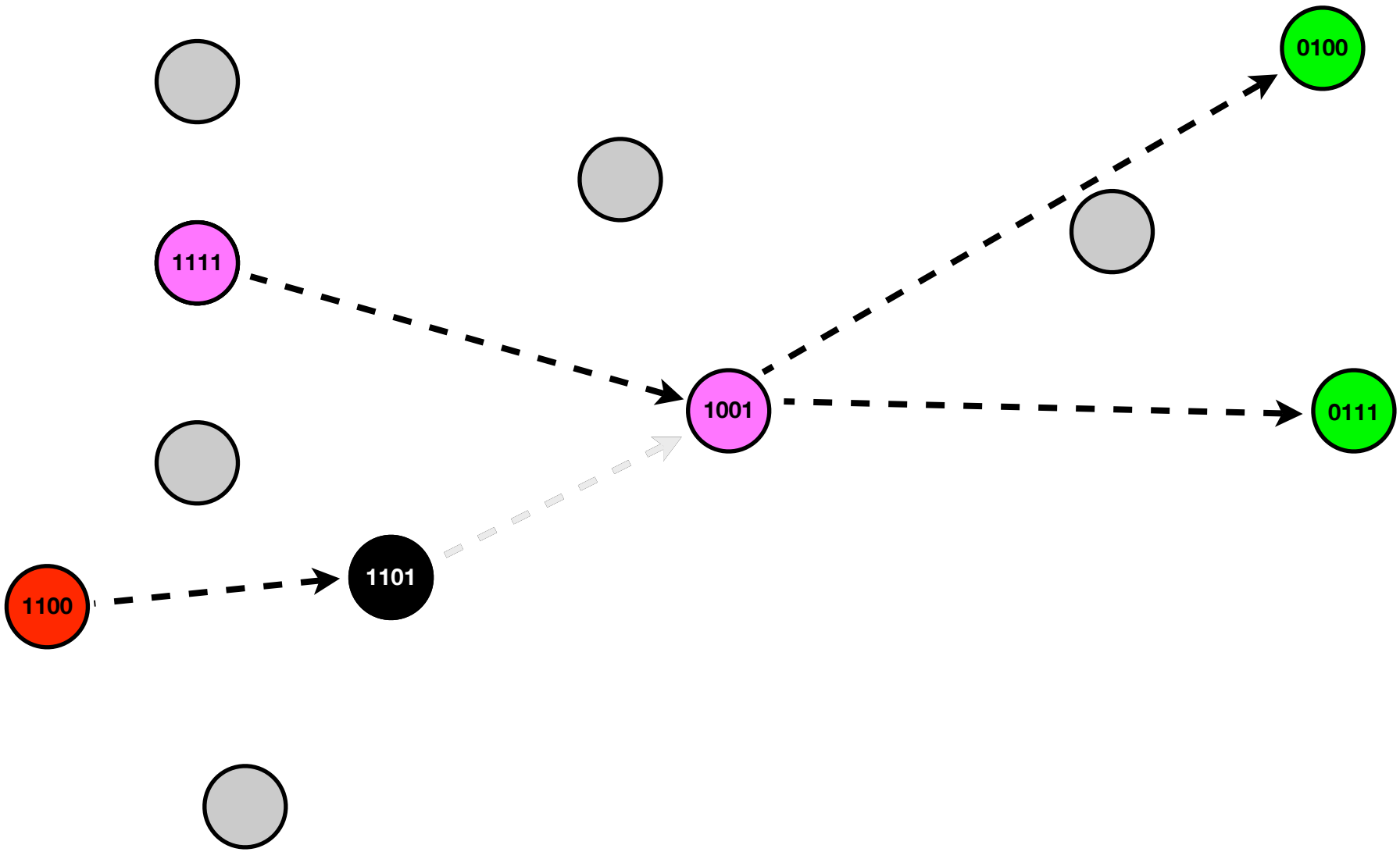
# Repairing the Multicast Tree



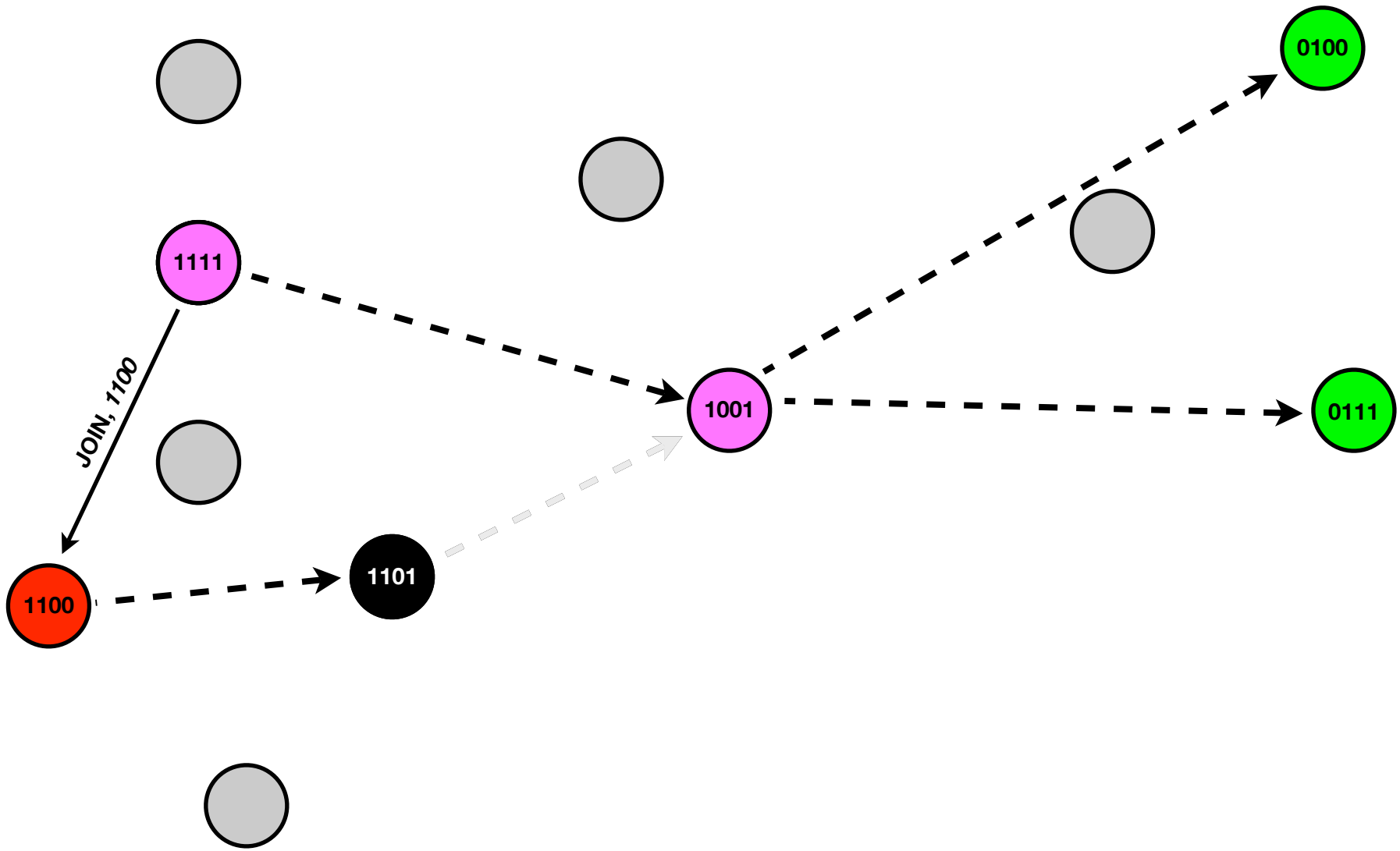
# Repairing the Multicast Tree



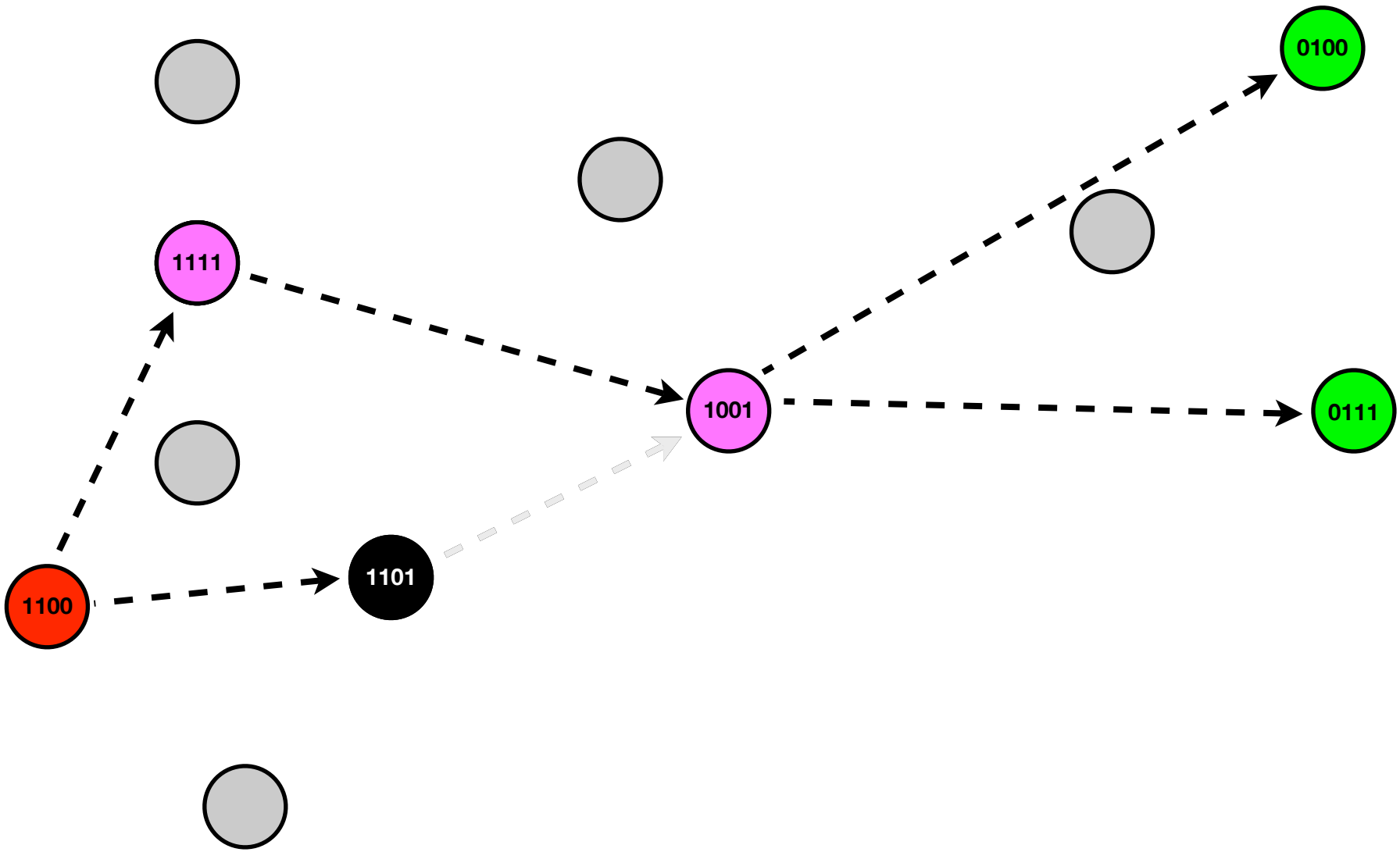
# Repairing the Multicast Tree



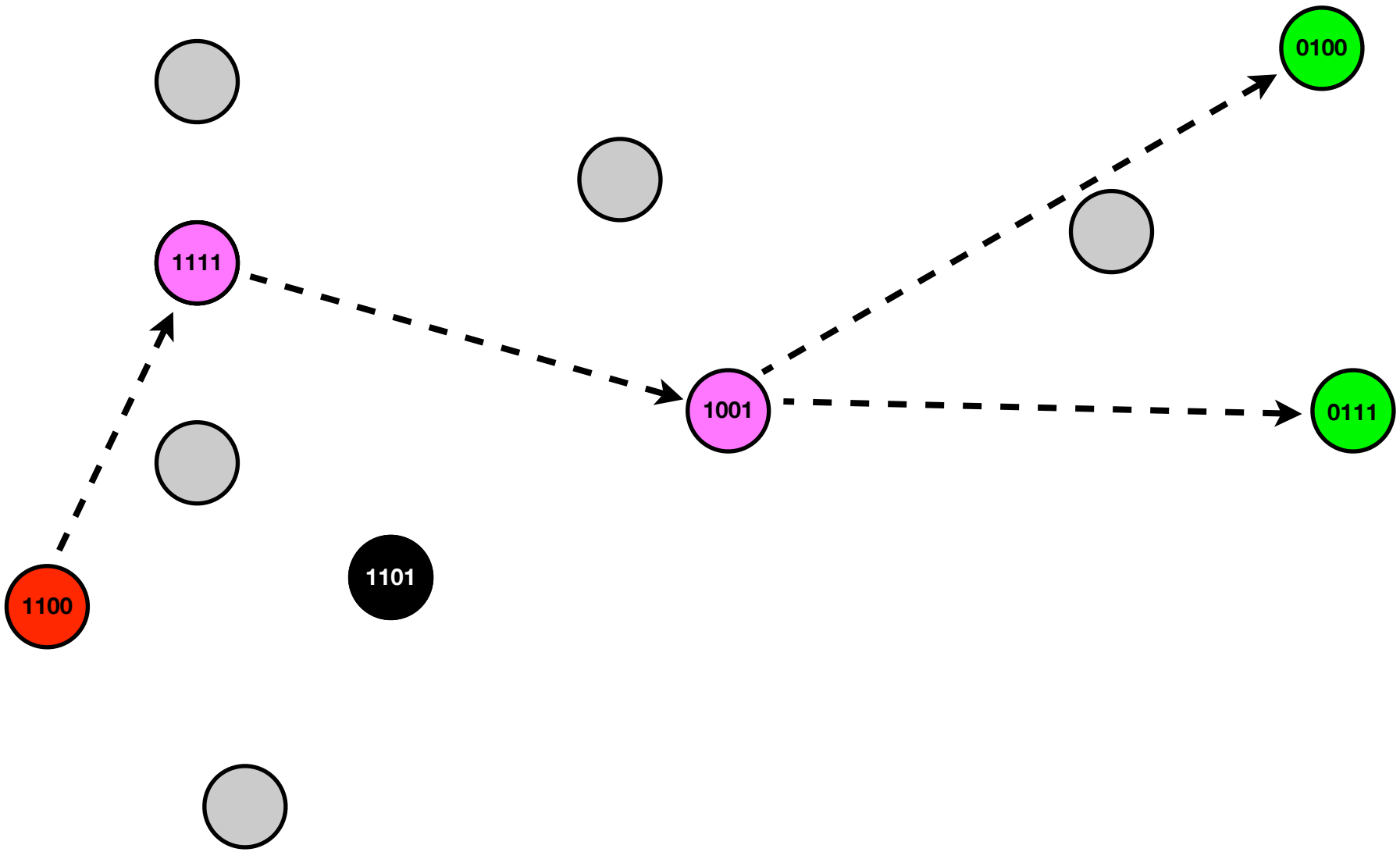
# Repairing the Multicast Tree



# Repairing the Multicast Tree



# Repairing the Multicast Tree



# Scribe Protocol > Reliability
# Repairing the Multicast Tree

- Scribe can also tolerate the failure of multicast tree roots (*rendezvous points*):
  - The state associated with the *rendezvous point* (group creator, access control data, etc.) is replicated across the *k* closest nodes to the root
    - These nodes are in the leaf set of the rendezvous *and* a typical value for *k* is 5
  - If the root fails, its children detect the fault and send JOIN messages again through Pastry's route operation
    - Pastry routes the JOIN messages to the new root: the live node with the numerically closest *nodeID* to the *groupID* (as before)
    - This node now takes the place of the *rendezvous point*
    - Multicast senders find the *rendezvous* point like before: also routing through Pastry

# Scribe Protocol > Reliability
## Providing Additional Guarantees

- Scribe offers reliable, ordered delivery of multicast messages only if the TCP connections between the nodes in the tree do not break
- Scribe also offers a set of upcalls, shall applications want to implement stronger reliability guarantees on top of it:
  - forwardHandler( *msg* )
    - It is s invoked by Scribe before the node forwards a multicast message (*msg*) to its children
    - The method can modify *msg* before it is forwarded
  - joinHandler( *msg* )
    - It is invoked by Scribe after a new child is added to one of the node's children tables.
    - *msg* is the JOIN message

# Providing Additional Guarantees (2)

- faultHandler( *msg* )
  - It is invoked by Scribe when a node suspects that its parent is faulty
  - *msg* is the JOIN message that is to be sent to repair the tree
  - The method may modify *msg* before it is sent

# Scribe Protocol > Reliability
# Providing Additional Guarantees (3)

- Using these handlers, an example of an ordered and reliable multicast implementation on top of Scribe is:
  - The forwardHandler is defined such that:
    - the root assigns a sequence number to each multicast message
    - recently multicast messages are buffered by each node in the tree (including the root)
  - Messages are retransmitted after the multicast tree is repaired:
    - The faultHandler includes the last sequence number *n* delivered to the node in the JOIN message
    - The joinHandler retransmits every message above *n* to the new child
  - The messages must be buffered longer than the maximum amount of time it takes to repair the multicast tree after a TCP connection breaks

# Providing Additional Guarantees (4)

- To tolerate root failures, its full state must be replicated
  - e.g. running an algorithm like Paxos on a set of replicas chosen from the root's leaf-set, to ensure strong data consistency
  - Scribe will automatically choose a new root (using Pastry) when the old one fails: it just needs to start off using the replicated state and updating it as needed

# Experimental Evaluation

# Experimental Evaluation

- A prototype Scribe implementation was evaluated using a specially developed packet-level, discreet event simulator
  - The simulator models the propagation delay on the physical links but it does not model queuing delay nor packet losses
  - No cross traffic was included in the experiments

# Experimental Evaluation

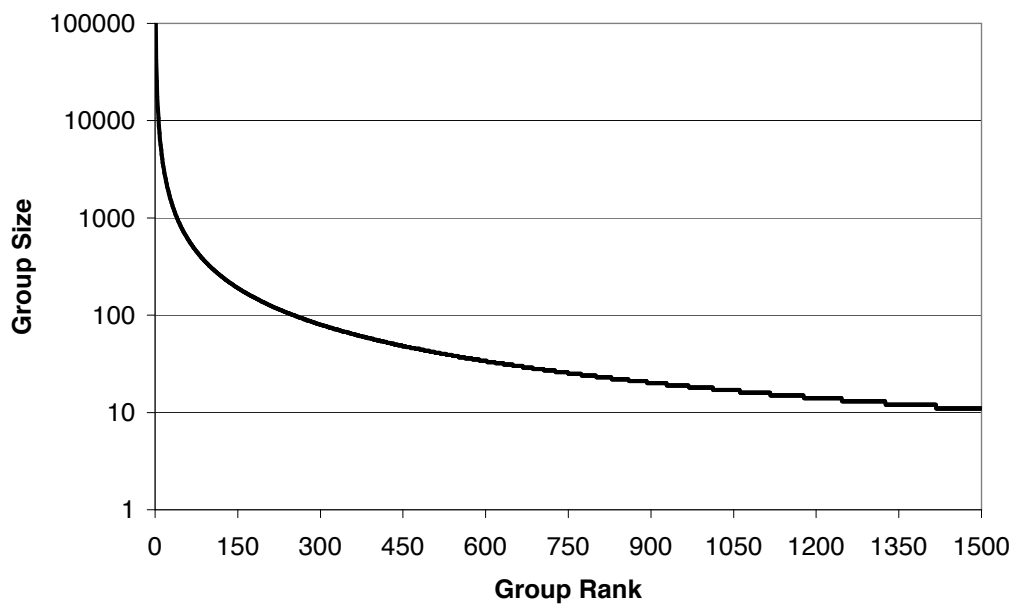
- A prototype Scribe implementation was evaluated using a specially developed packet-level, discreet event simulator
  - The simulator models the propagation delay on the physical links but it does not model queuing delay nor packet losses
  - No cross traffic was included in the experiments
- The simulation ran on a network topology of 5050 routers generated by the Georgia Tech random graph generator (using the transit-stub model)
  - The Scribe code didn't run on the routers, but on 100,000 end nodes, randomly assigned to routers with uniform probability
  - Each end system was directly attached to its assigned router by a LAN link

# Experimental Evaluation

- A prototype Scribe implementation was evaluated using a specially developed packet-level, discreet event simulator
  - The simulator models the propagation delay on the physical links but it does not model queuing delay nor packet losses
  - No cross traffic was included in the experiments
- The simulation ran on a network topology of 5050 routers generated by the Georgia Tech random graph generator (using the transit-stub model)
  - The Scribe code didn't run on the routers, but on 100,000 end nodes, randomly assigned to routers with uniform probability
  - Each end system was directly attached to its assigned router by a LAN link
- IP multicast routing used a shortest-path tree formed by the merge of the unicast routes from the source to each recipient. Control messages were ignored

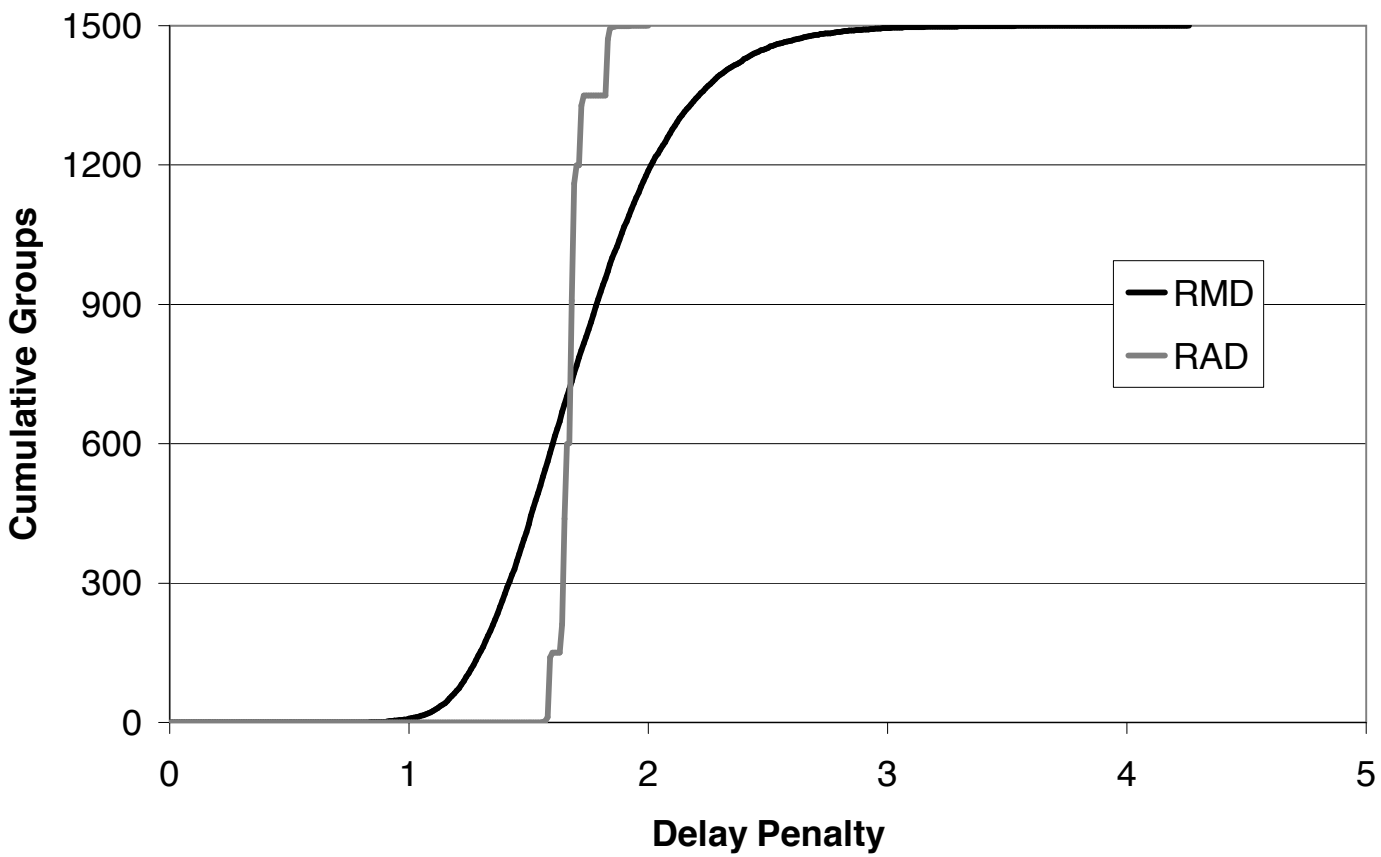
# Experimental Evaluation (2)

- Scribe groups are ranked by size its members were uniformly distributed over the set of nodes:
  - The size of the group with rank *r* is given by: *gsize*(*r*) = (int) (N . r $^{-1.25}$ + 0.5) where N is the total number of nodes
  - There were 1,500 groups and 100,000 nodes (N)
  - The exponent 1,25 was chosen to ensure a minimum group size of 11 (which appears to be typical of Instant Messaging applications)
  - The maximum group size is 100,000 (*r = 1*) and the sum of all group sizes is 395,247



# Experimental Evaluation
## Delay Penalty

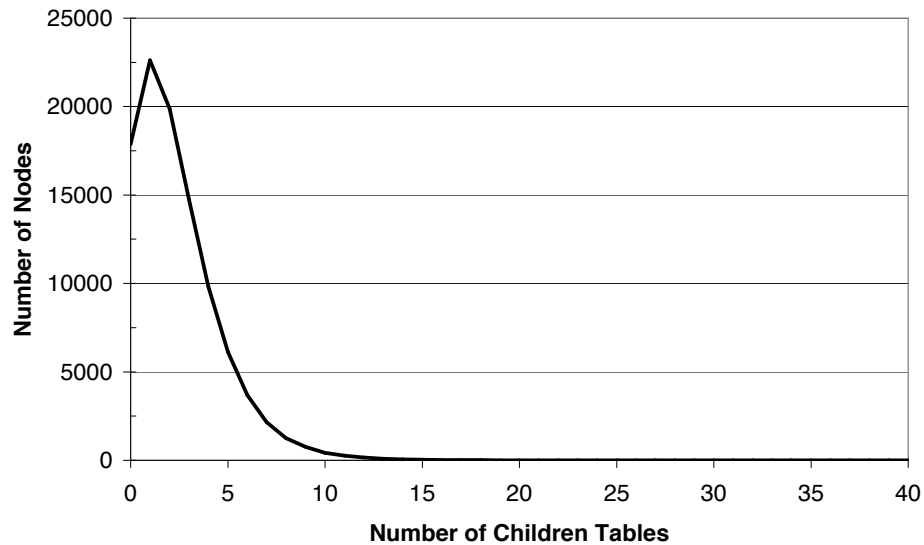
- Comparison of multicast delays between Scribe and IP multicast using two metrics:
  - RMD is the ratio between the maximum delay using Scribe and the maximum delay using IP multicast
  - RAD is the ratio between the average delay using Scribe and the maximum delay using IP multicast

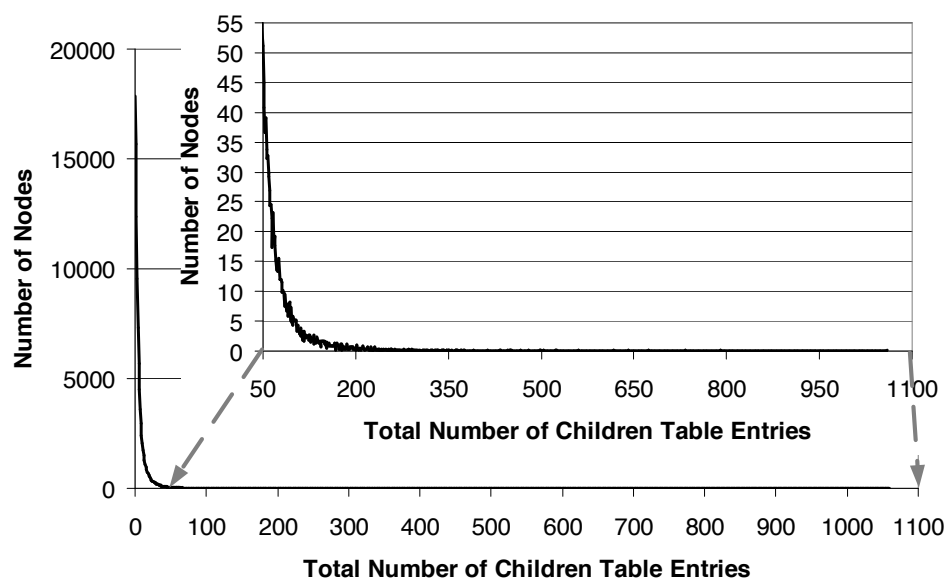



- 50% of groups have RAD=1.68 and RMD=1.69
- In the worst case, the maximum RAD is 2 and the maximum RMD is 4.26

# Experimental Evaluation
## Node Stress

- The number of nodes with non-empty children tables and the number of entries in each node's children table were measured
- With 1,500 groups, the mean number of non-empty children tables per node is 2.4
  - The median number is 2
  - The maximum number of tables is 40
- The mean number of entries on the nodes' children tables is 6.2
  - The median is 3
  - The maximum is 1059

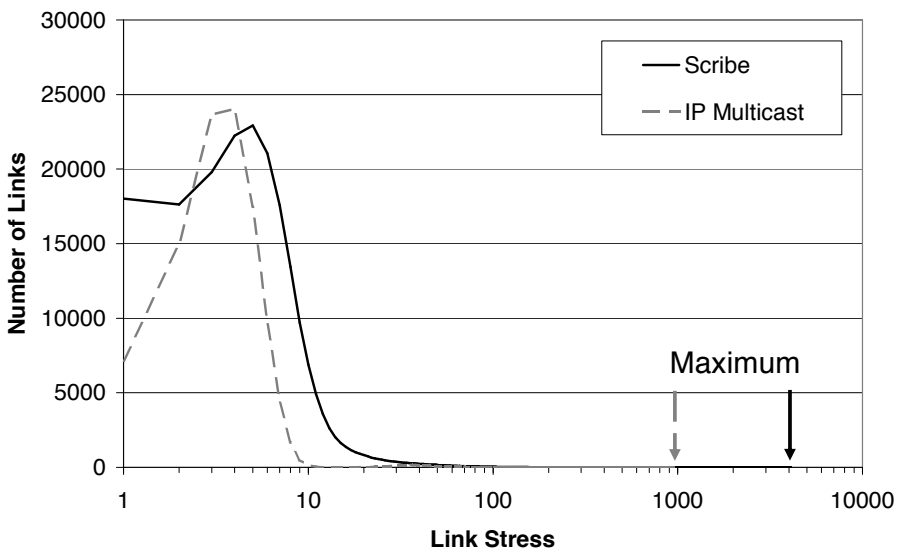


# Experimental Evaluation
## Link Stress

- The number of packets sent over each link was measured for both Scribe and IP multicast

- The total number of links was 1,035,295 and the total number of messages was 2,489,824 for Scribe and 758,853 for IP multicast
  - The mean number of messages per link is:
    - 2.4 for Scribe
    - 0,7 for IP multicast
  - The maximum link stress is:
    - 4031 for Scribe
    - 950 for IP multicast
- Maximum link stress for naïve IP multicast implementation (all unicast transmissions) is 100,000



# Experimental Evaluation
## Bottleneck Remover

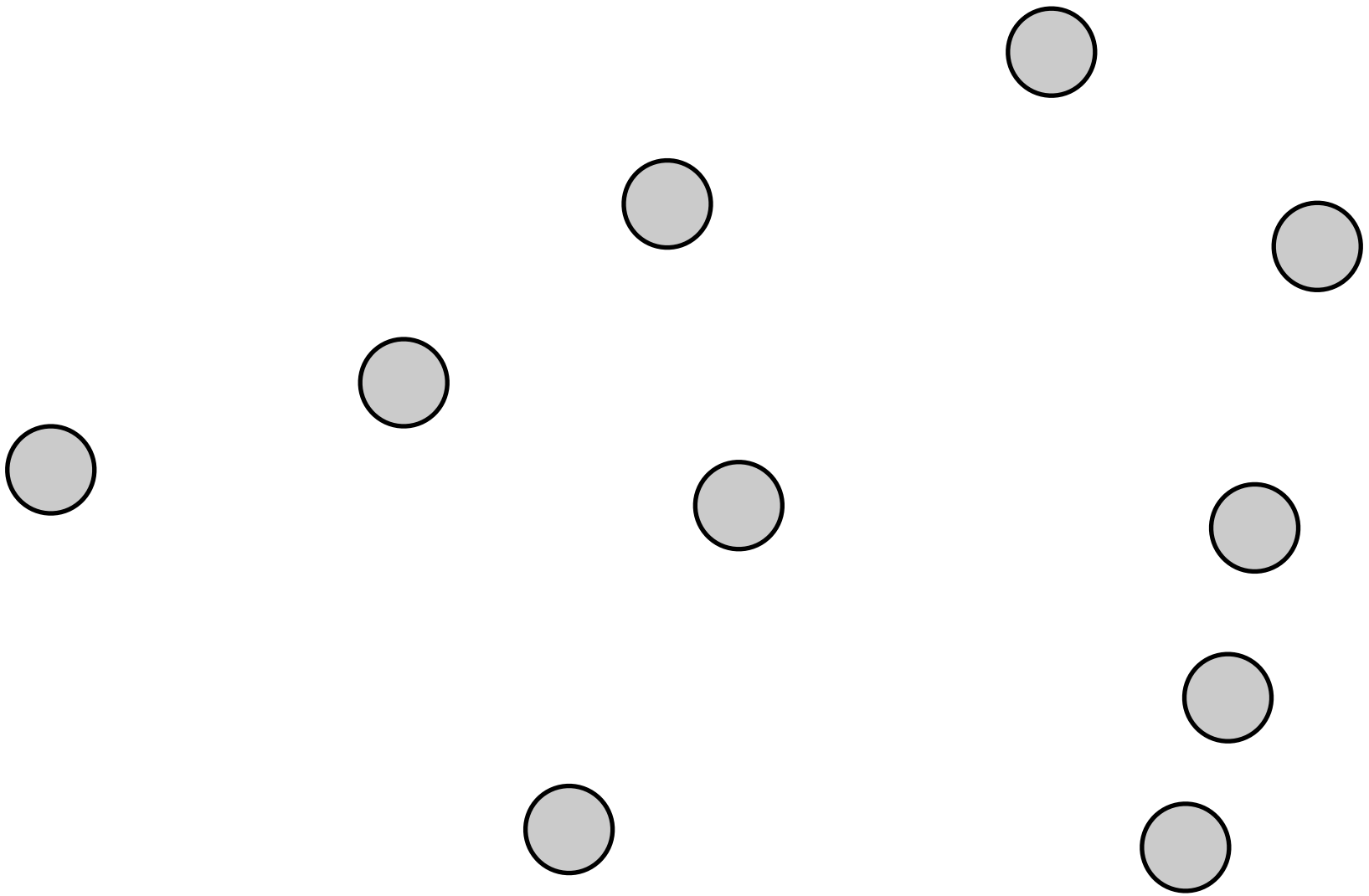
- The base mechanism for building multicast trees in Scribe assumes that all nodes have equal capacity and strives to distribute load evenly across all nodes
  - Although, in several deployment scenarios some nodes may have less computational power or bandwidth available than others
- The distribution of children table entries has a long tail: the nodes at the end may become bottlenecks under high load conditions

- The Bottleneck Remover is a simple algorithm to remove bottlenecks when they occur:
  - The algorithm allows nodes to bound the amount of multicast forwarding they do by off-loading children to other nodes

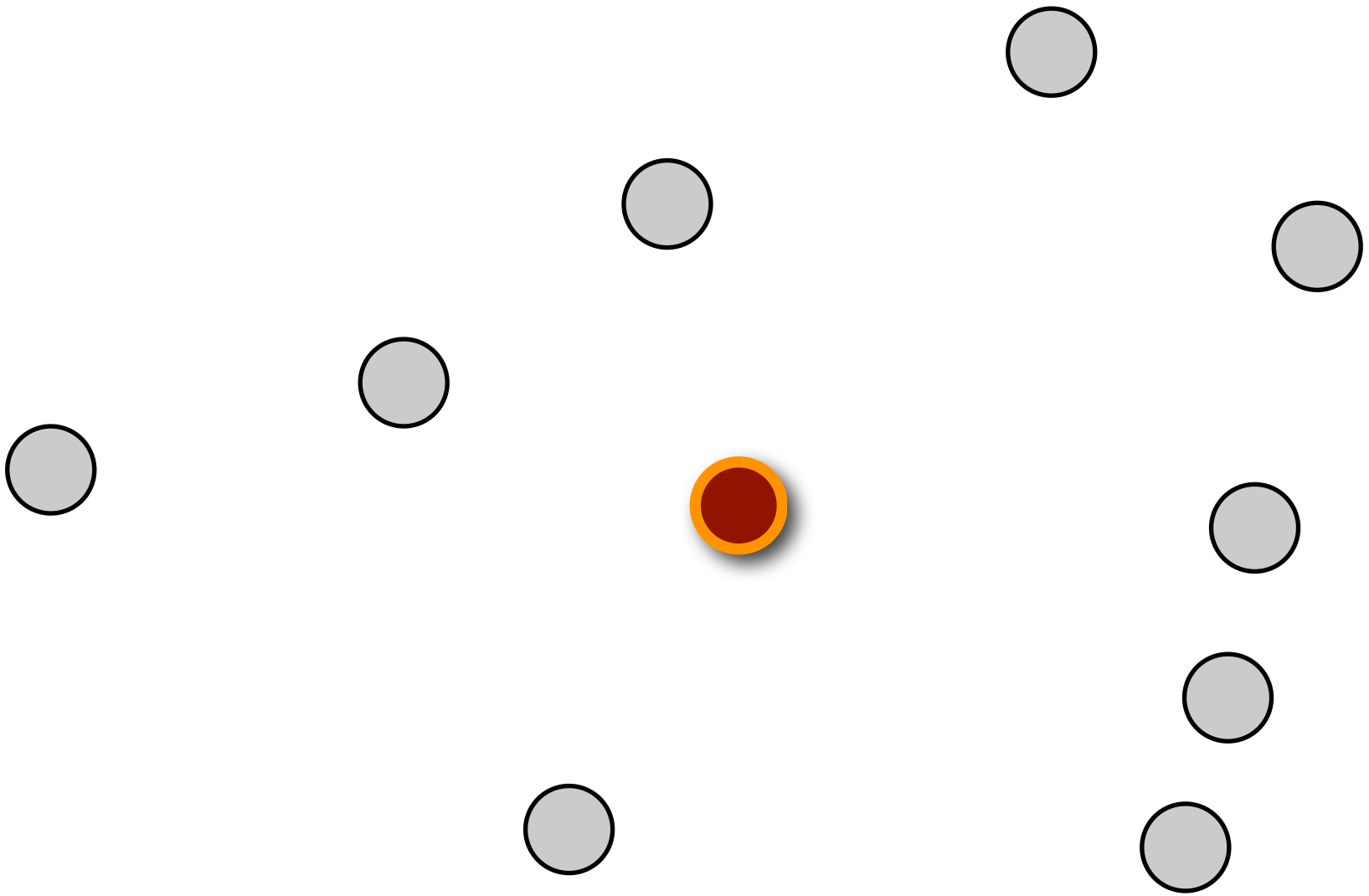
# Experimental Evaluation
## Bottleneck Remover (2)

- The Bottleneck Remover works as follows:
  - When a node detects that it is overloaded, it selects the group that consumes the most resources and chooses the child in this group that is farthest away (according to the proximity metric)
  - The parent drops the child by sending it a message containing the children table for the group along with the delays between each child and the parent
  - When the child receives such a message it performs the following operations:
    1. It measures the delay between itself and other nodes in the received children table
    2. It computes the total delay between itself and the parent via each node
    3. It sends a JOIN message to the node that provides the smallest combined delay, hence minimising the transmission time to reach its parent through one of its previous siblings

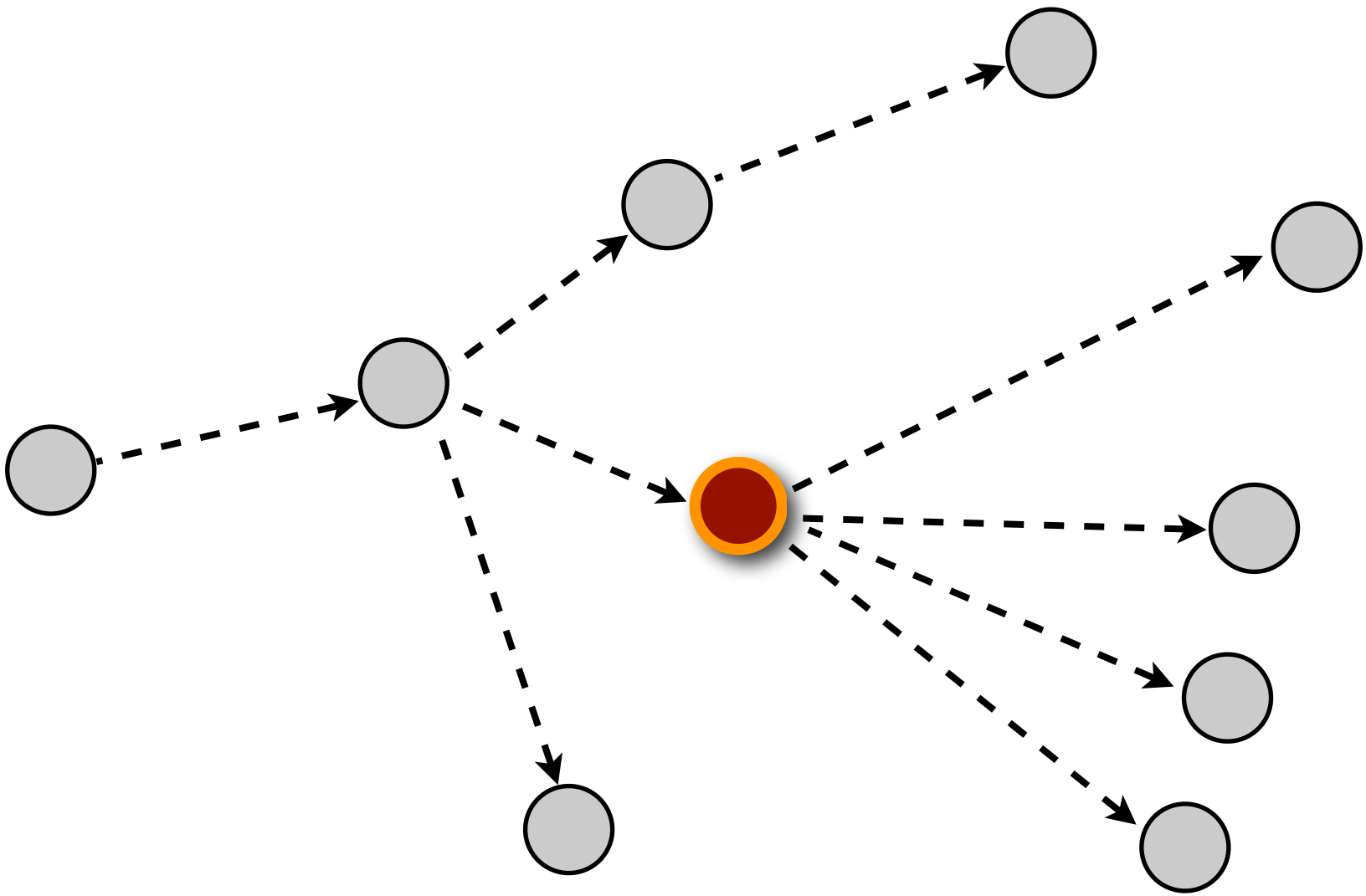
# Experimental Evaluation
## Bottleneck Remover (3)



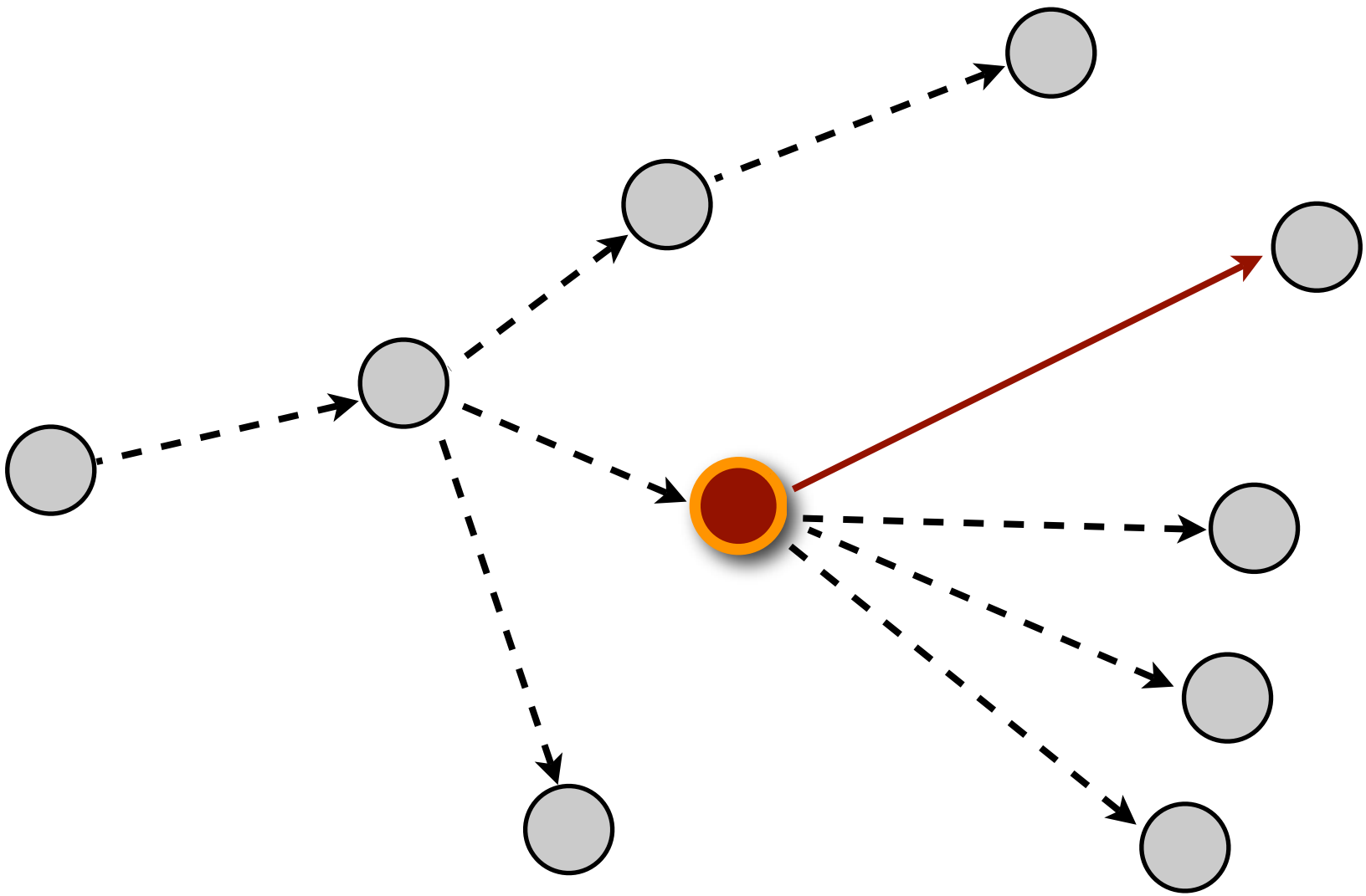
# Experimental Evaluation
## Bottleneck Remover (3)



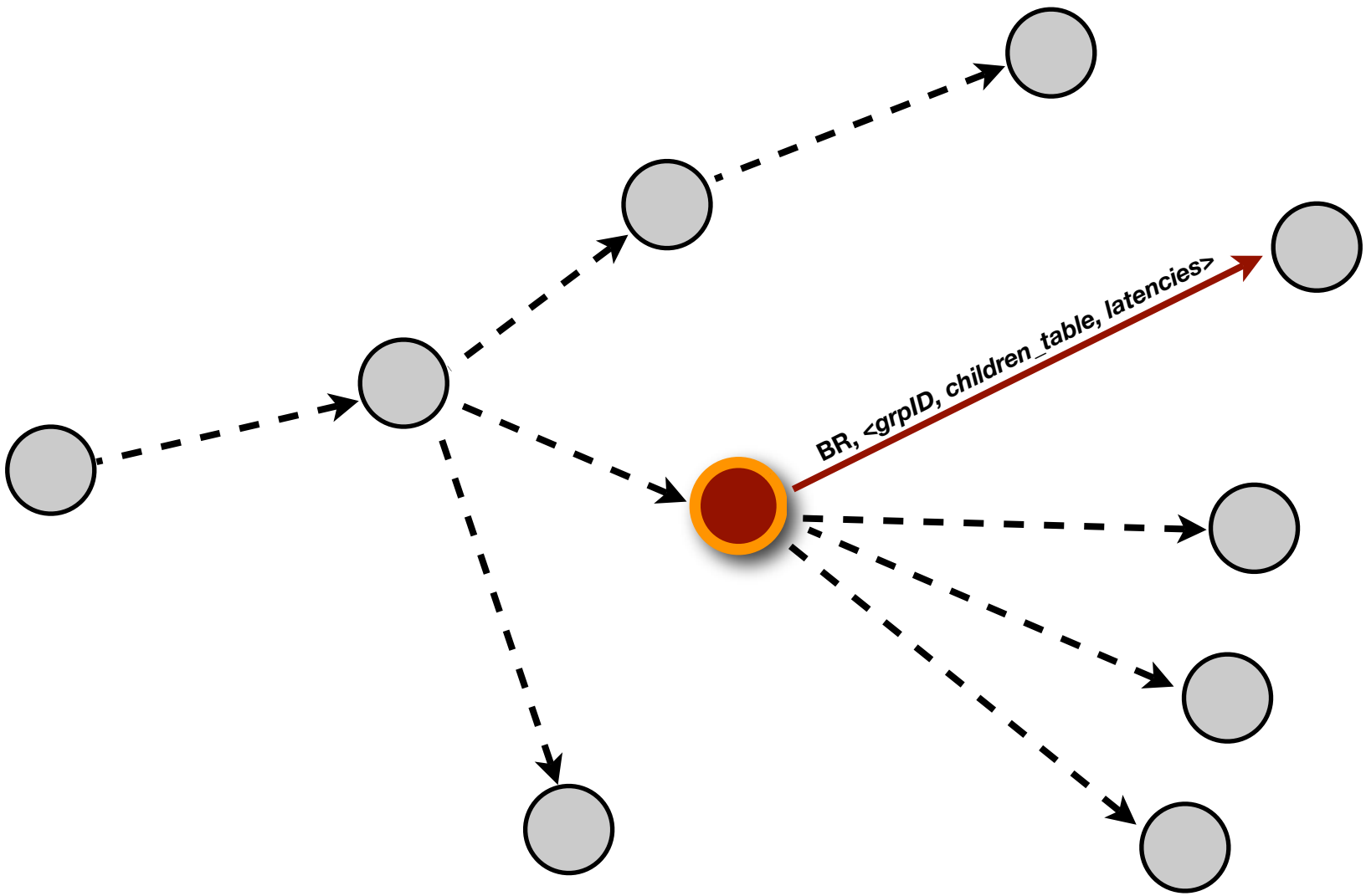
# Experimental Evaluation
## Bottleneck Remover (3)



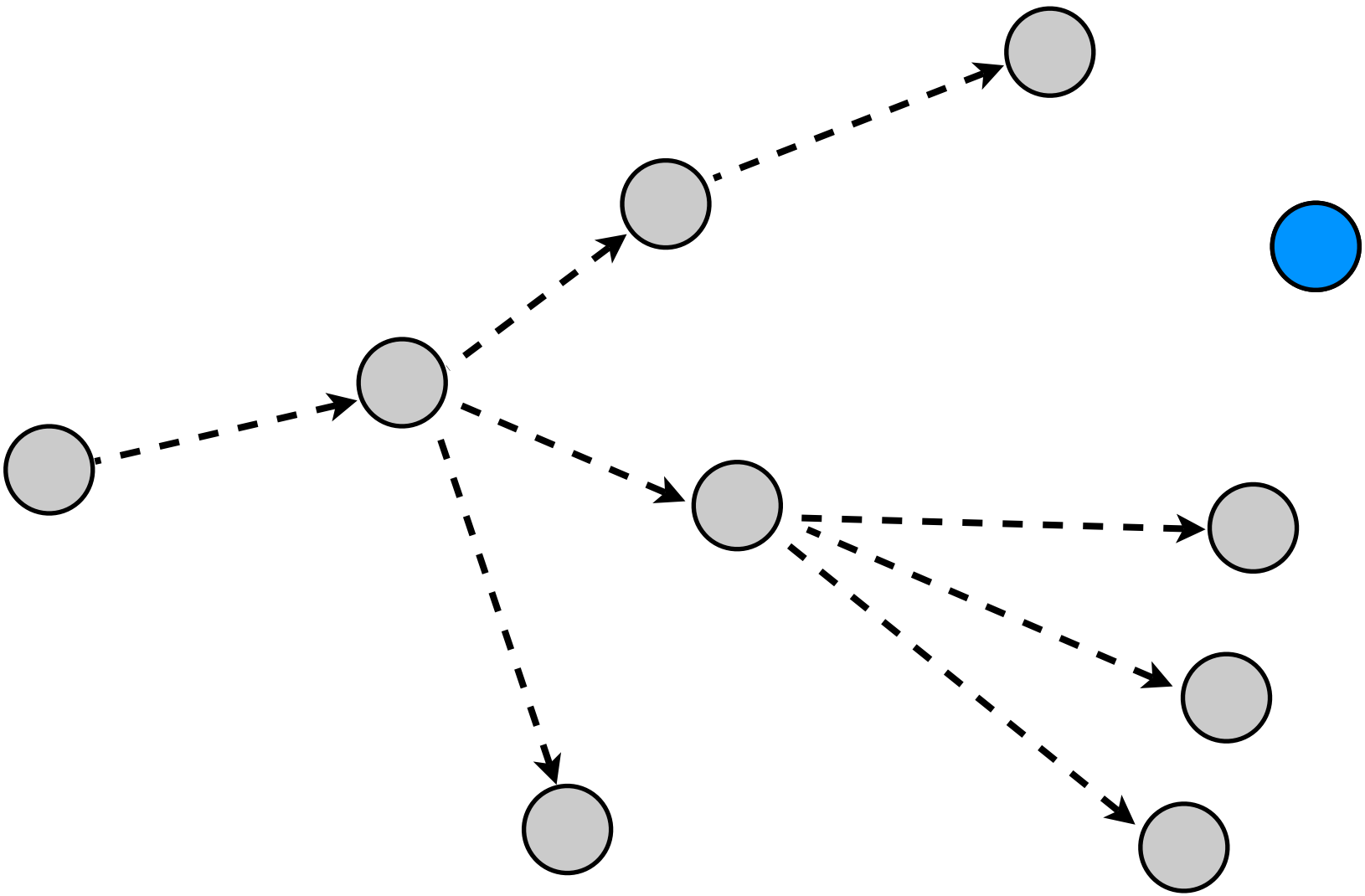
# Experimental Evaluation
## Bottleneck Remover (3)



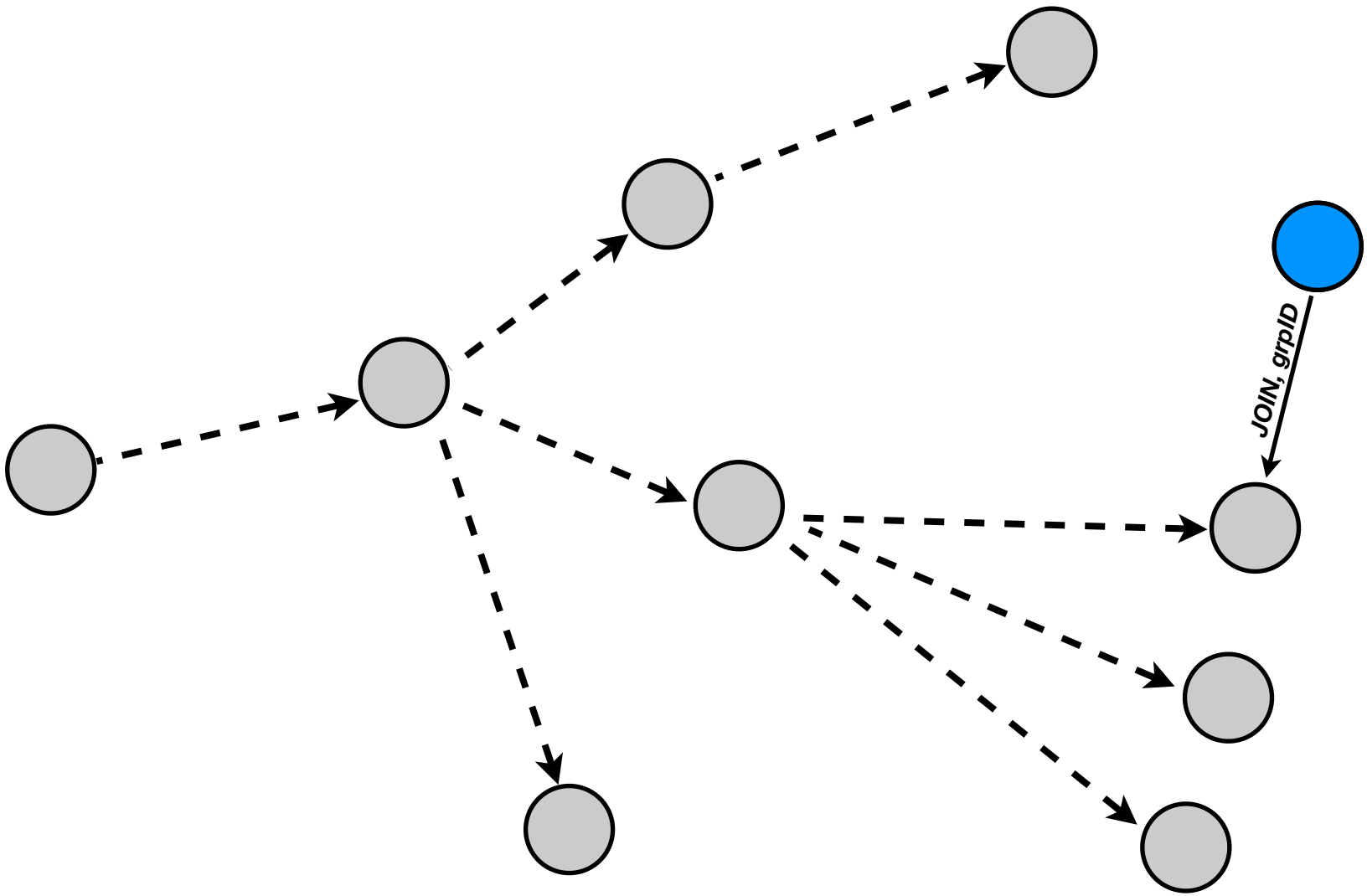
# Experimental Evaluation
## Bottleneck Remover (3)



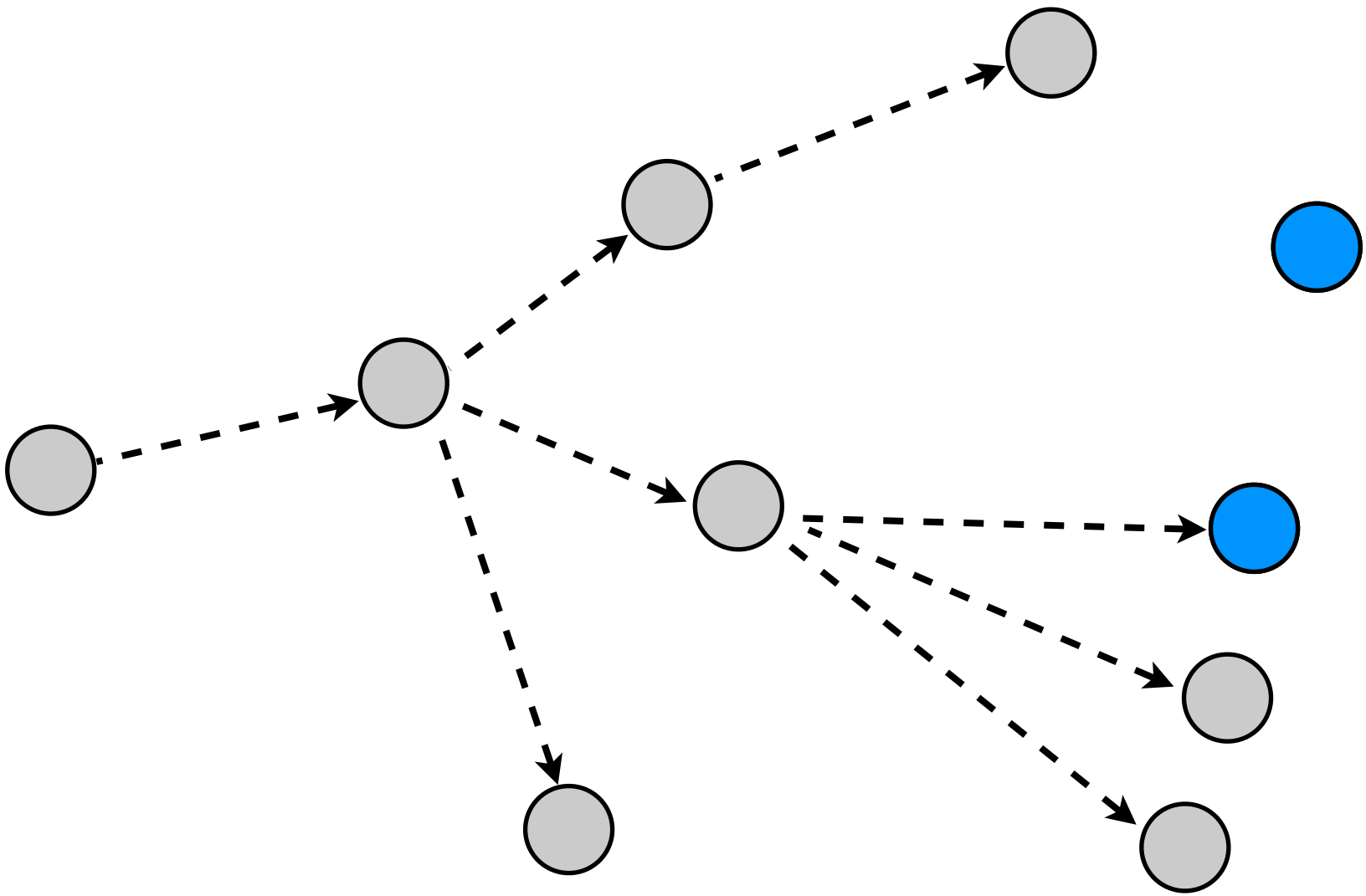
# Experimental Evaluation
## Bottleneck Remover (3)



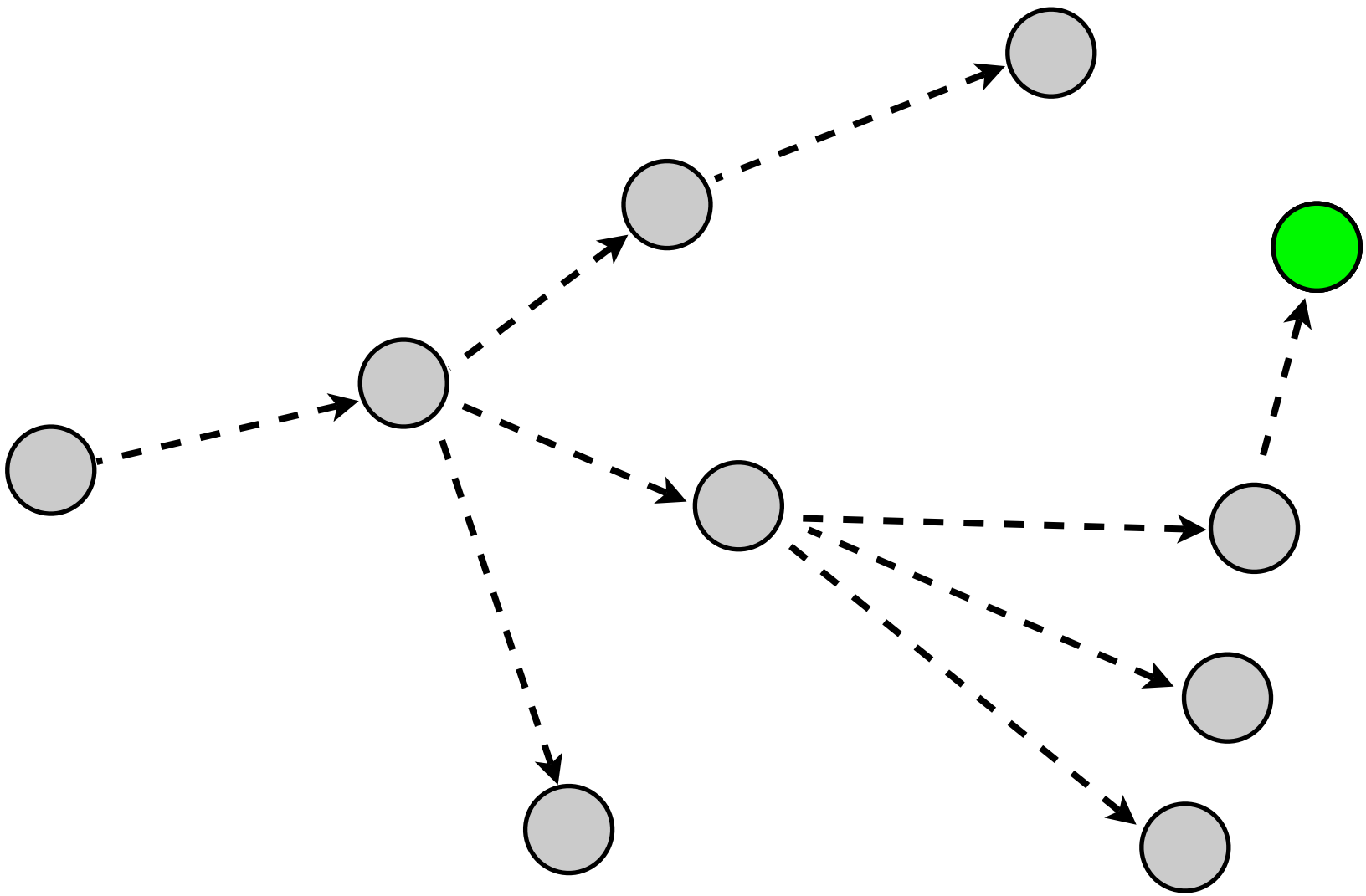
# Experimental Evaluation
## Bottleneck Remover (3)



# Experimental Evaluation
## Bottleneck Remover (3)

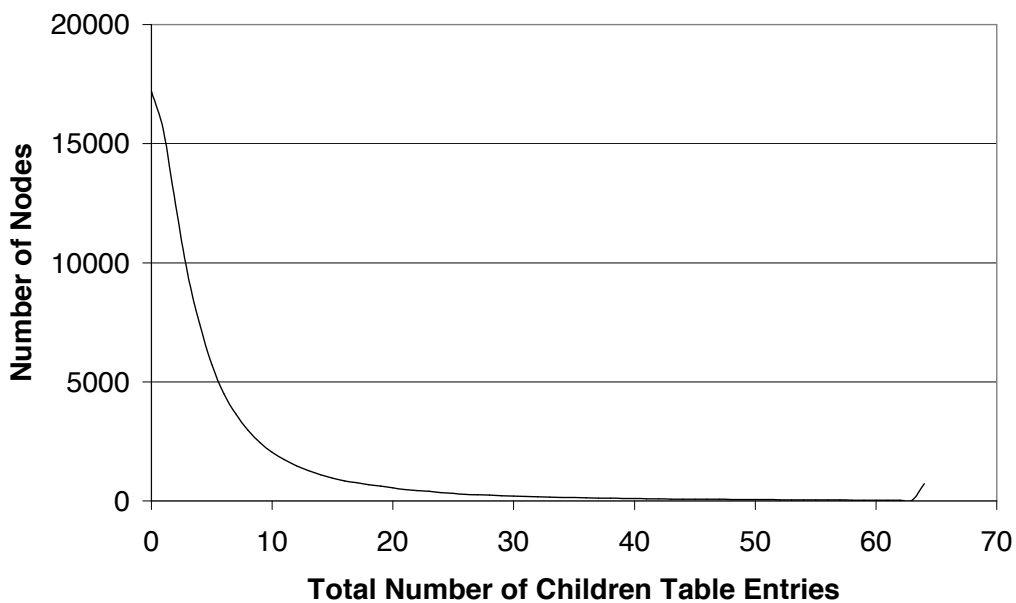


# Experimental Evaluation
## Bottleneck Remover (3)



# Experimental Evaluation
## Bottleneck Remover (3)

- This algorithm may (with low probability) introduce routing loops
  - Those are detected by having each parent *p* propagate, to its children, the *nodeID's* in the path from root to *p*
- Other drawback is the increase of the link stress for joining:
  - The average link stress increases from 2.4 to 2.7 (maximum increases from 4031 to 4728)
- On the other hand, it bounded the number of children per node at 64 in the experiments

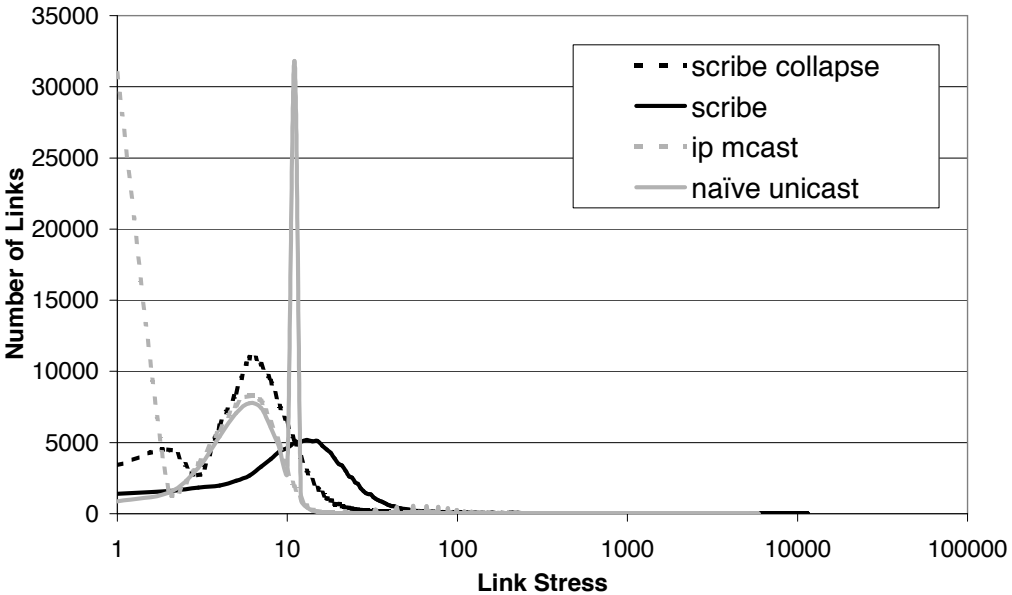


# Experimental Evaluation
## Scalability With Many Small Groups

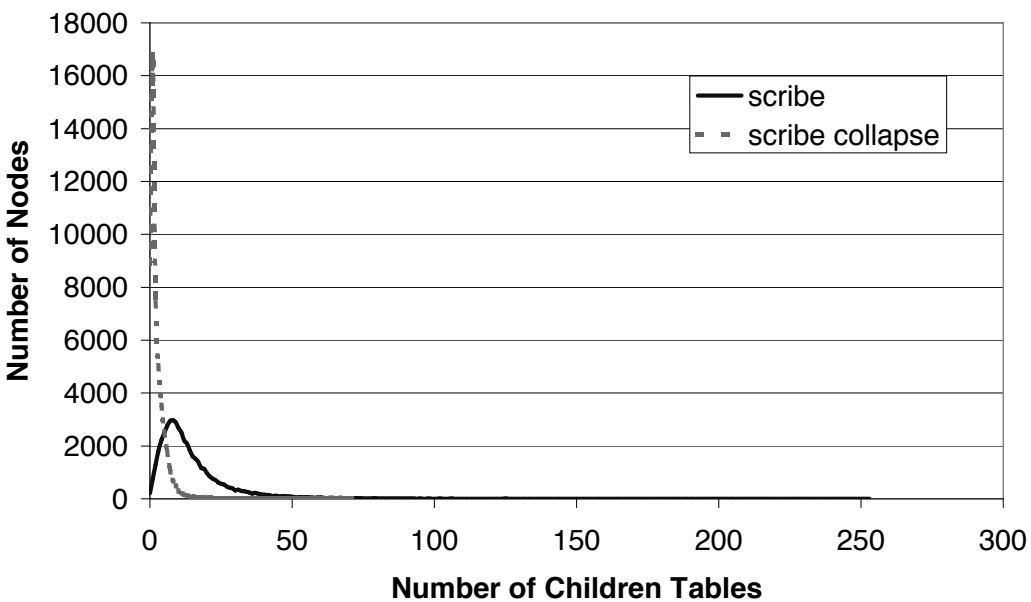
- This additional experiment was ran to evaluate Scribe's scalability with a large number of groups
- The setup was similar to the others, except that there were 50,000 Scribe nodes and 30,000 groups with 11 members each
- The number of children tables and children table entries per node were measured
- Results show:
  - Scribe scales well because it distributes children tables and children table entries evenly across the nodes
  - Scribe multicast trees are not efficient for small groups
- To mitigate this last problem, an algorithm to produce more efficient trees for small groups was implemented:
  - Trees are built as before but the algorithm collapses long paths in the tree by removing nodes that are not members of a group and have only one entry in the group's children table
  - These new results are labelled "*scribe collapse*"

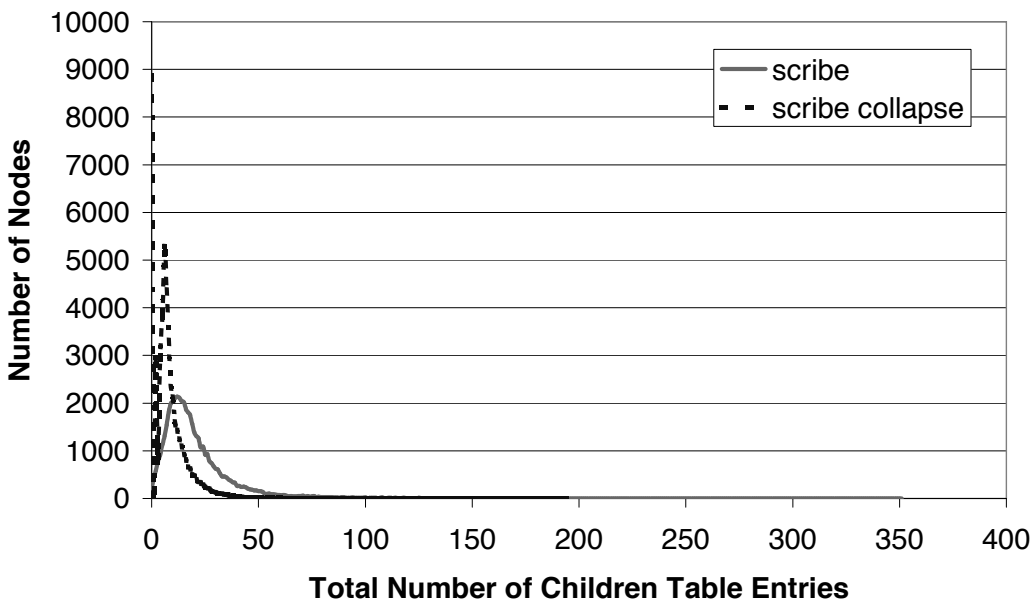
# Experimental Evaluation

## Scalability With Many Small Groups




Scribe Collapse reduced the average link stress from 6.1 to 3.3 and the average number of children per node from 21.2 to 8.5



# References


- M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "**Scribe: A large-scale and descentralized application-level multicast infrastructure**", *IEEE Journal on Selected Areas in Communications, Vol. 20, No. 8*, Oct. 2002

- A. Rowstron and P. Druschel, "**Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems**" *in Proc. IFIP/ACM Middleware 2001*, Nov. 2001