# Algorithms and Distributed Systems 2019/2020 (Lab Two)

**MIEI - Integrated Master in Computer Science and Informatics**
Specialization block

**João Leitão** (jc.leitao@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

# Class structure:

- One Unstructured Overlay: HyParView.
- One Partially Unstructured Overlay: Plumtree.

# Class structure:

- **One Unstructured Overlay: HyParView.**
- One Partially Unstructured Overlay: Plumtree.

# Unstructured Overlay Management

**Reactive strategy:**

Partial views are updated only as a reaction to some external event.
*e.g.* Scamp.

**Cyclic strategy:**

Partial views are updated as a result of some periodically operation.
*e.g.* Cyclon.

# HyParView

**HyParView: a membership protocol for reliable gossip-based broadcast***

João Leitão
University of Lisbon
jleitao@lasige.di.fc.ul.pt

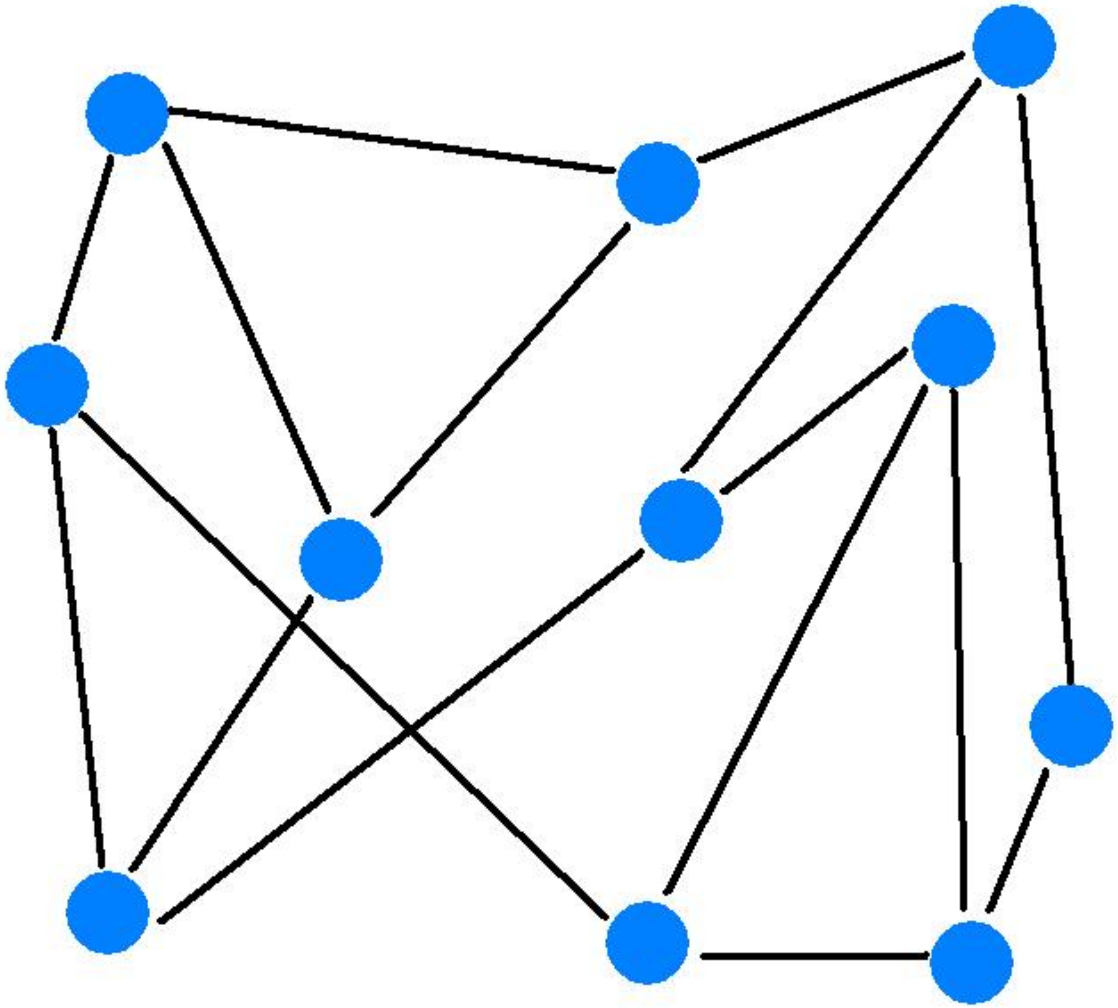
José Pereira
University of Minho
jop@di.uminho.pt

Luís Rodrigues
University of Lisbon
ler@di.fc.ul.pt

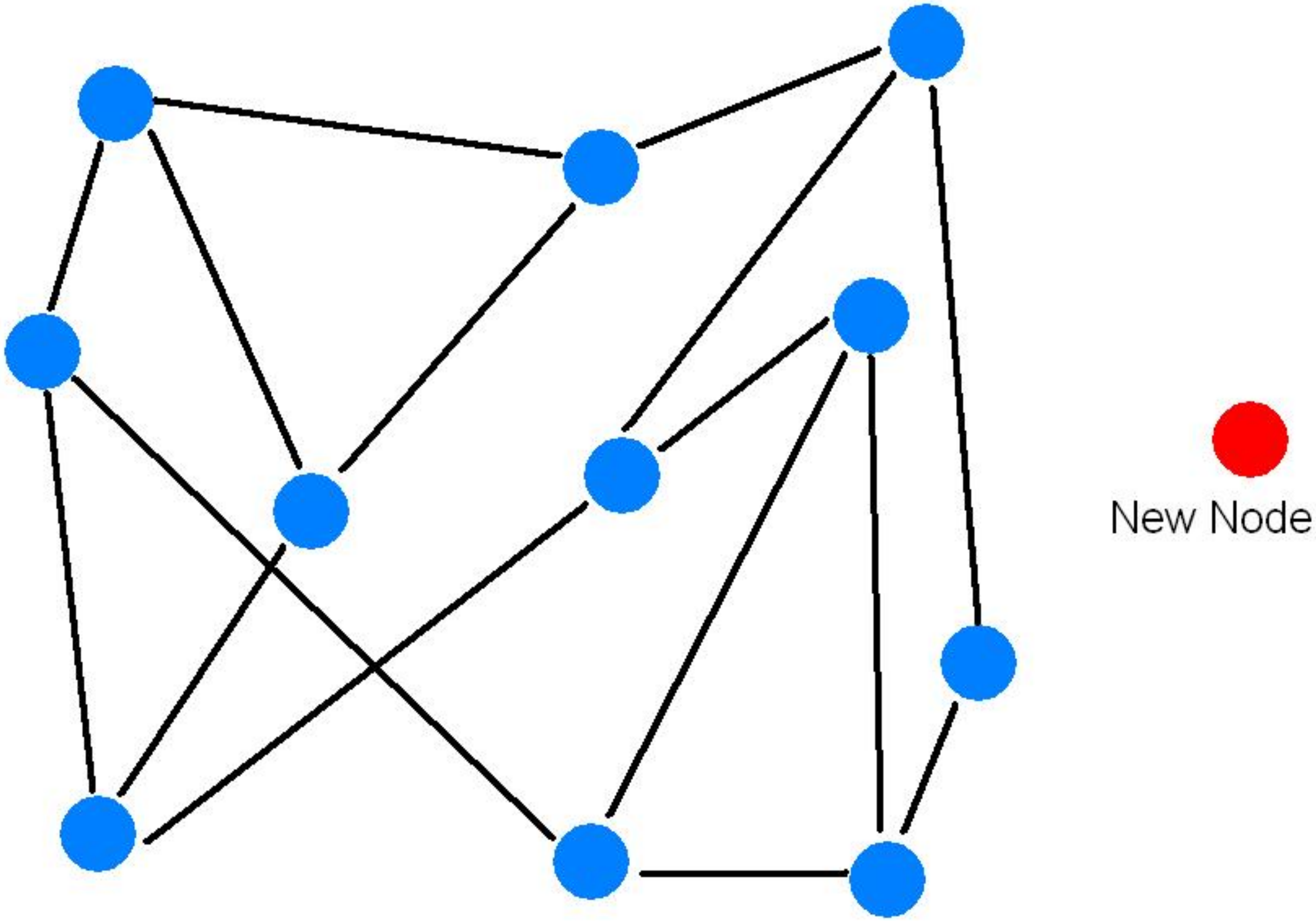
# HyParView

- Name stands for **Hy**brid **Par**tial **View**
  - It maintains two distinct partial views.
  - Active view:
    - Small sized (*fanout*+1).
    - Symmetric.
    - TCP connections are maintained with nodes in this views.
    - Used to disseminate messages.
  - Passive view:
    - Larger size in order of: $k \times \#(\text{active View})$
    - Used for fault tolerance.

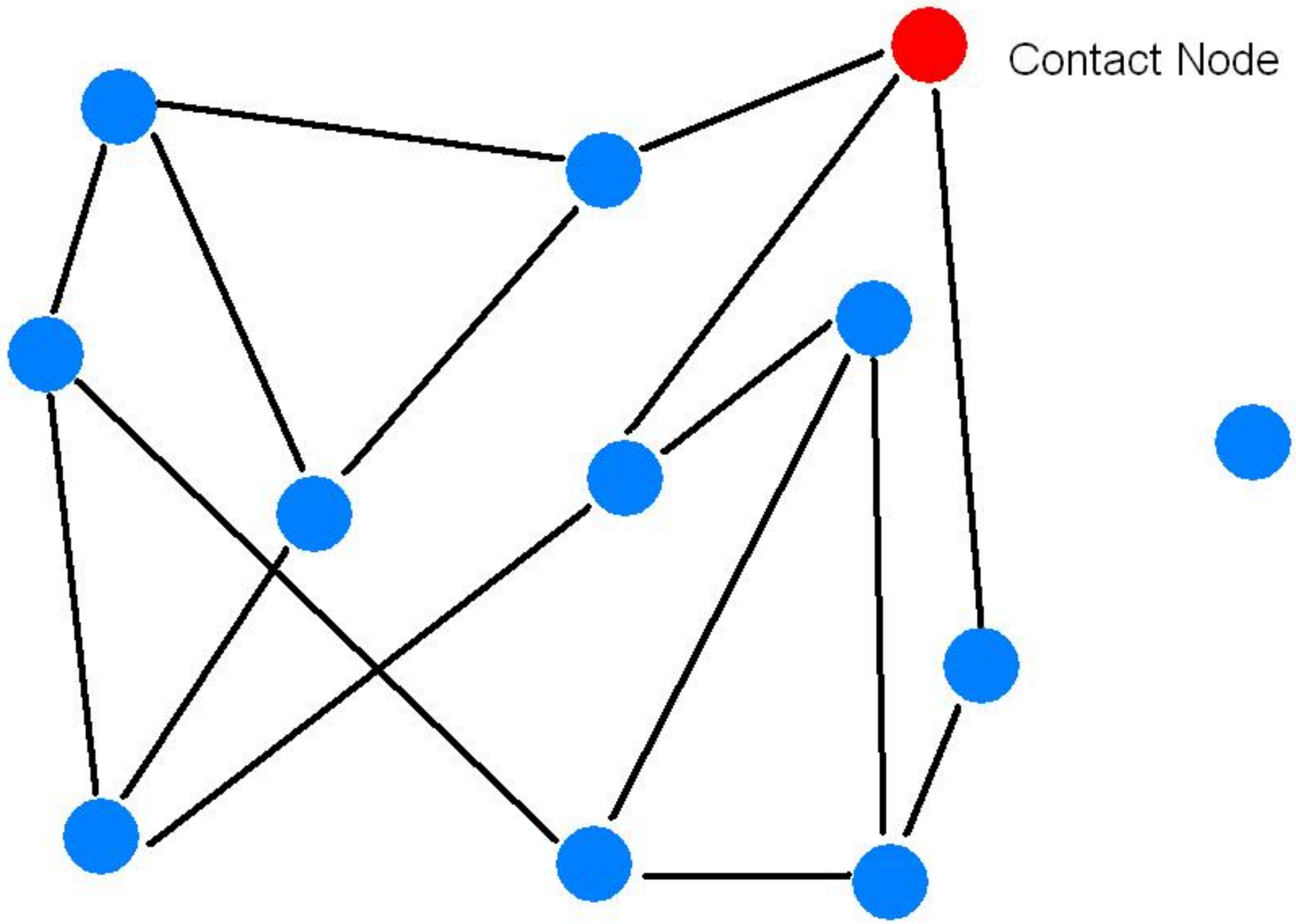
# HyParView Join Mechanism



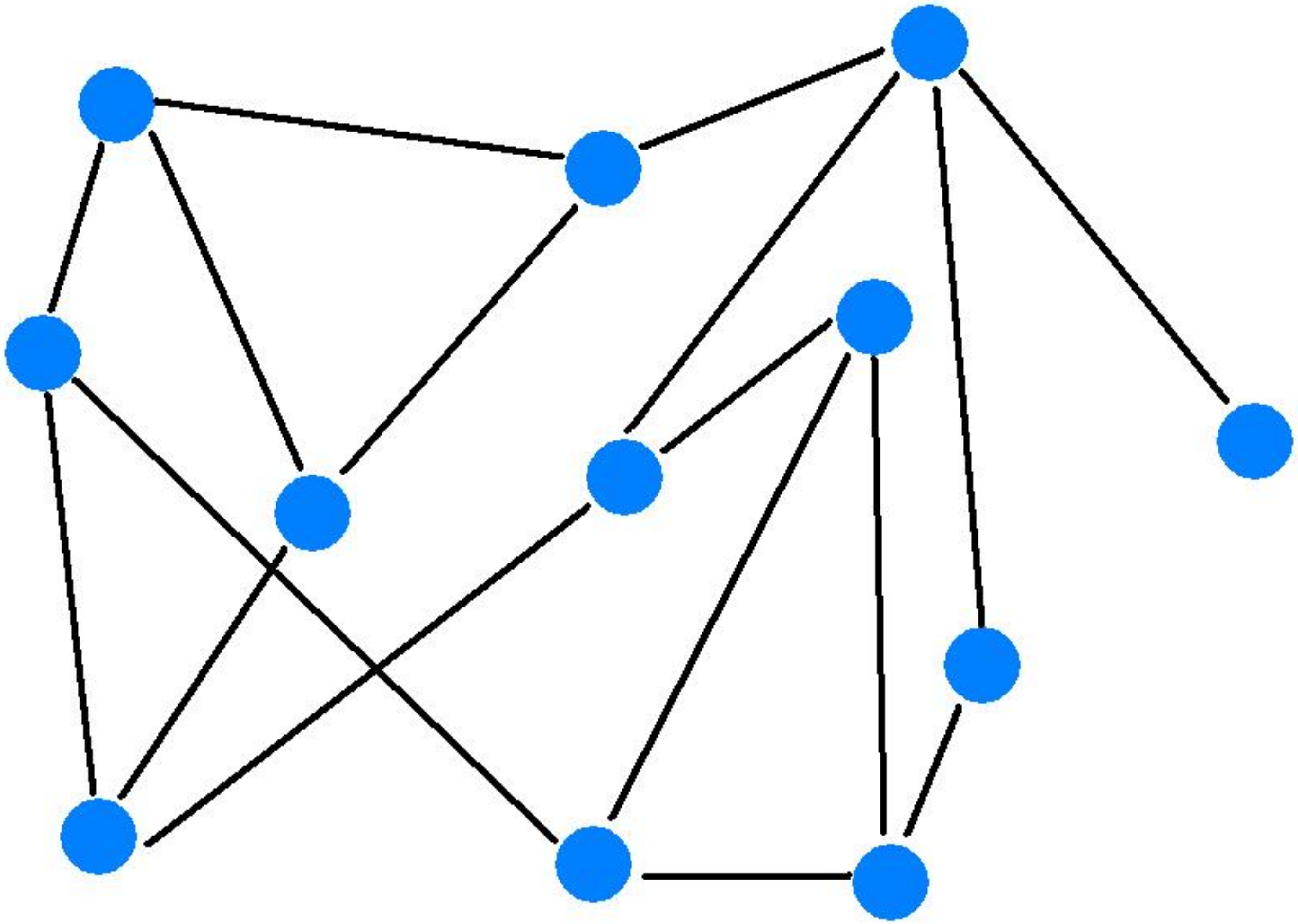
# HyParView Join Mechanism



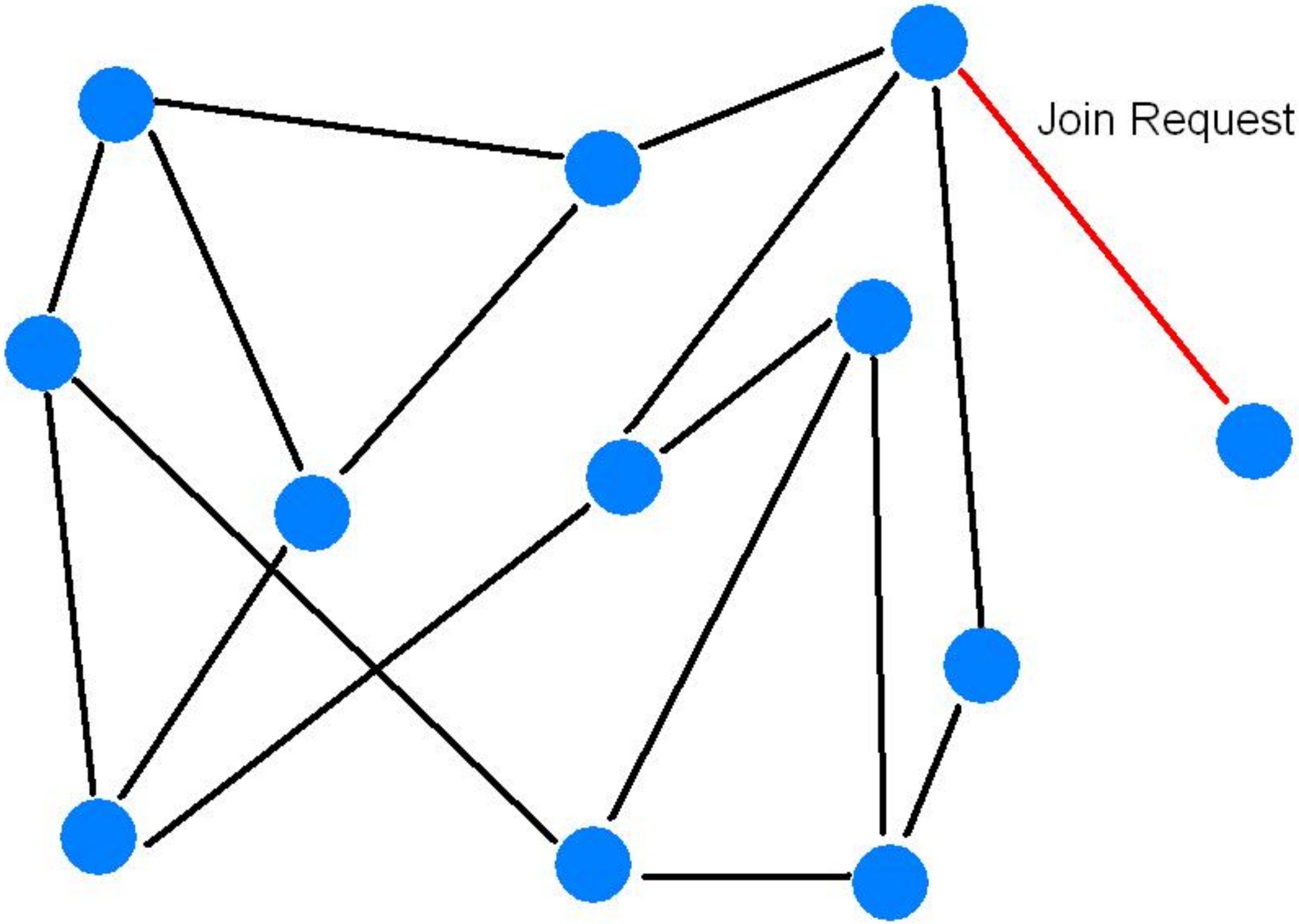
# HyParView Join Mechanism



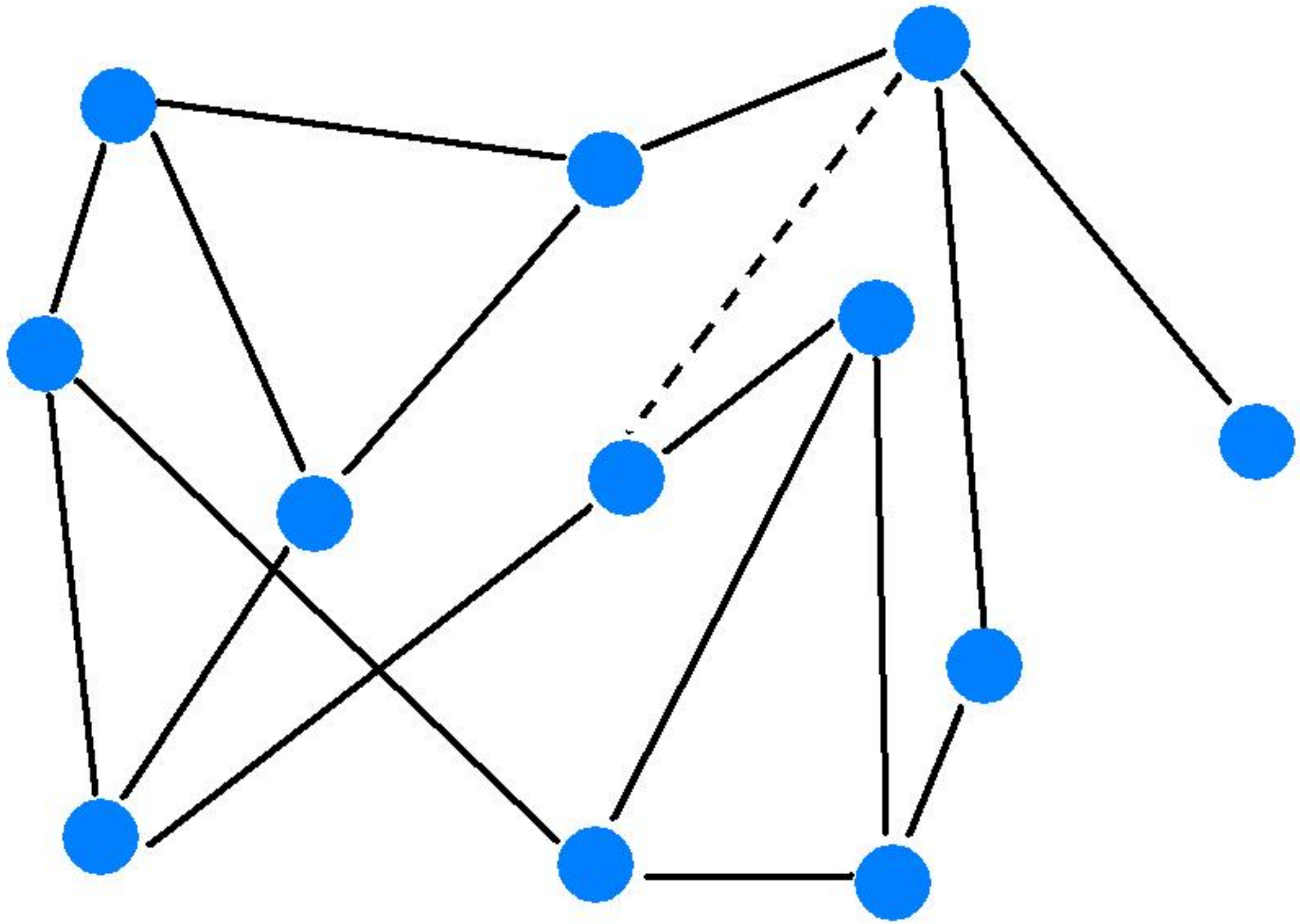
# HyParView Join Mechanism



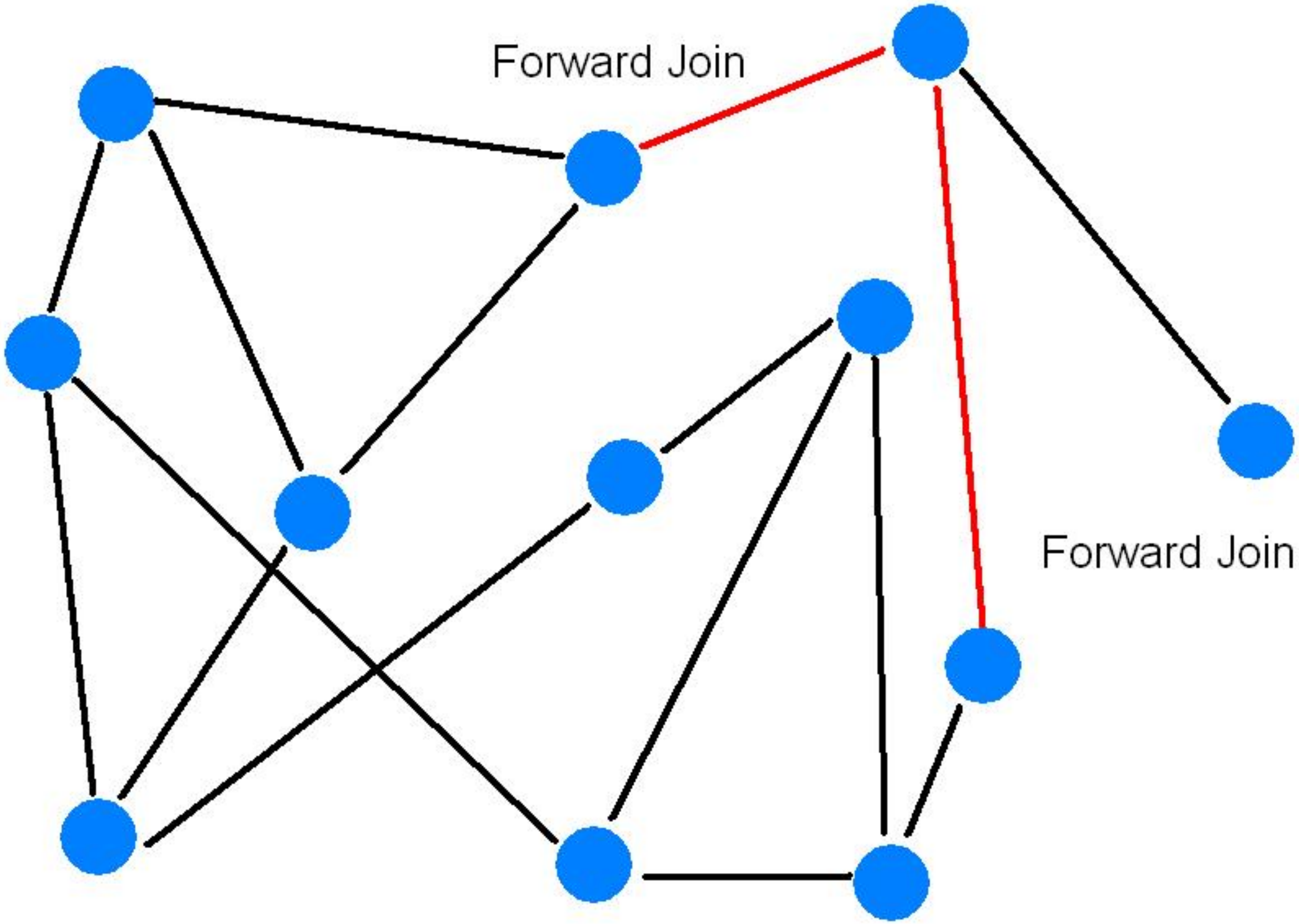
# HyParView Join Mechanism



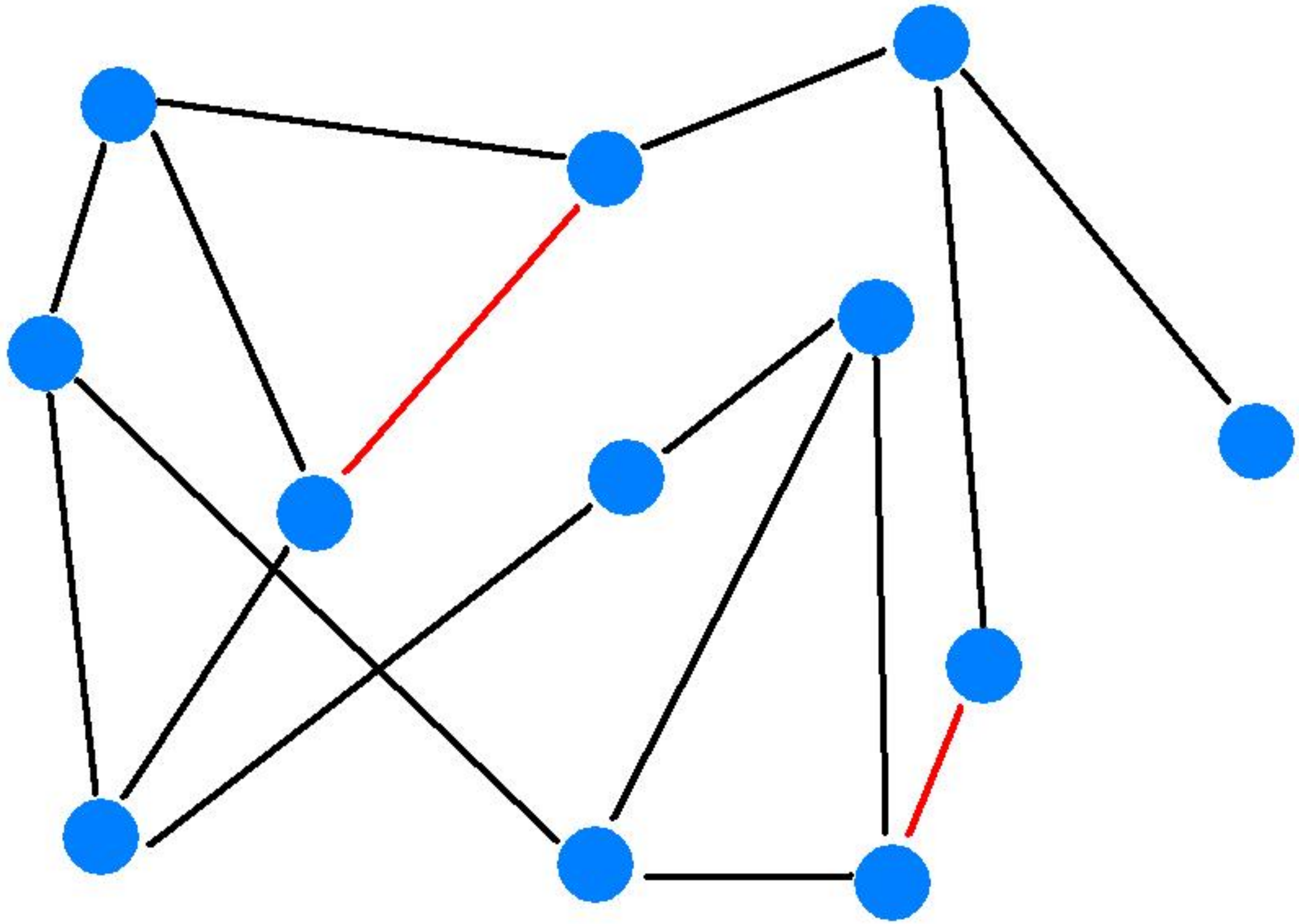
# HyParView Join Mechanism



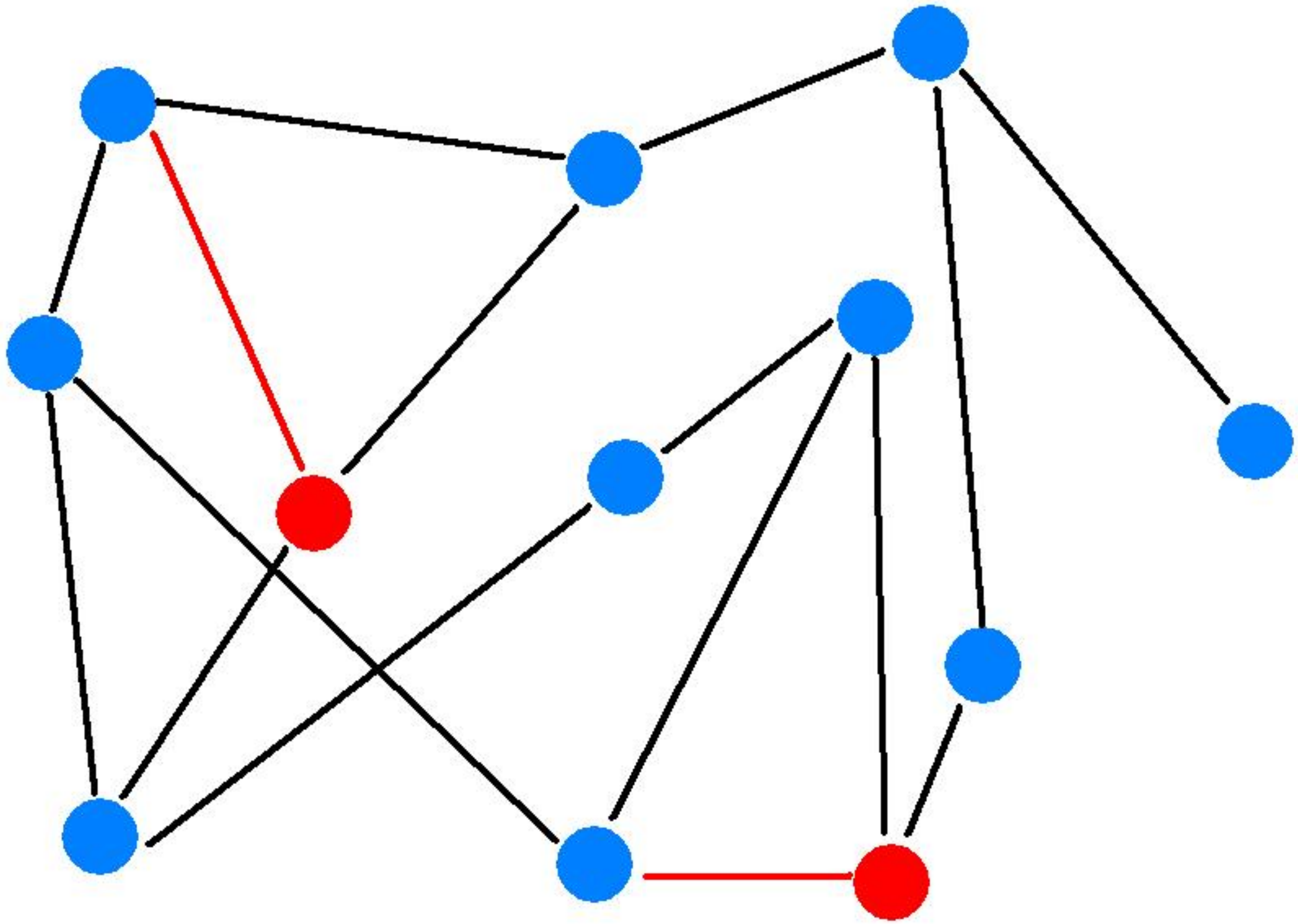
# HyParView Join Mechanism



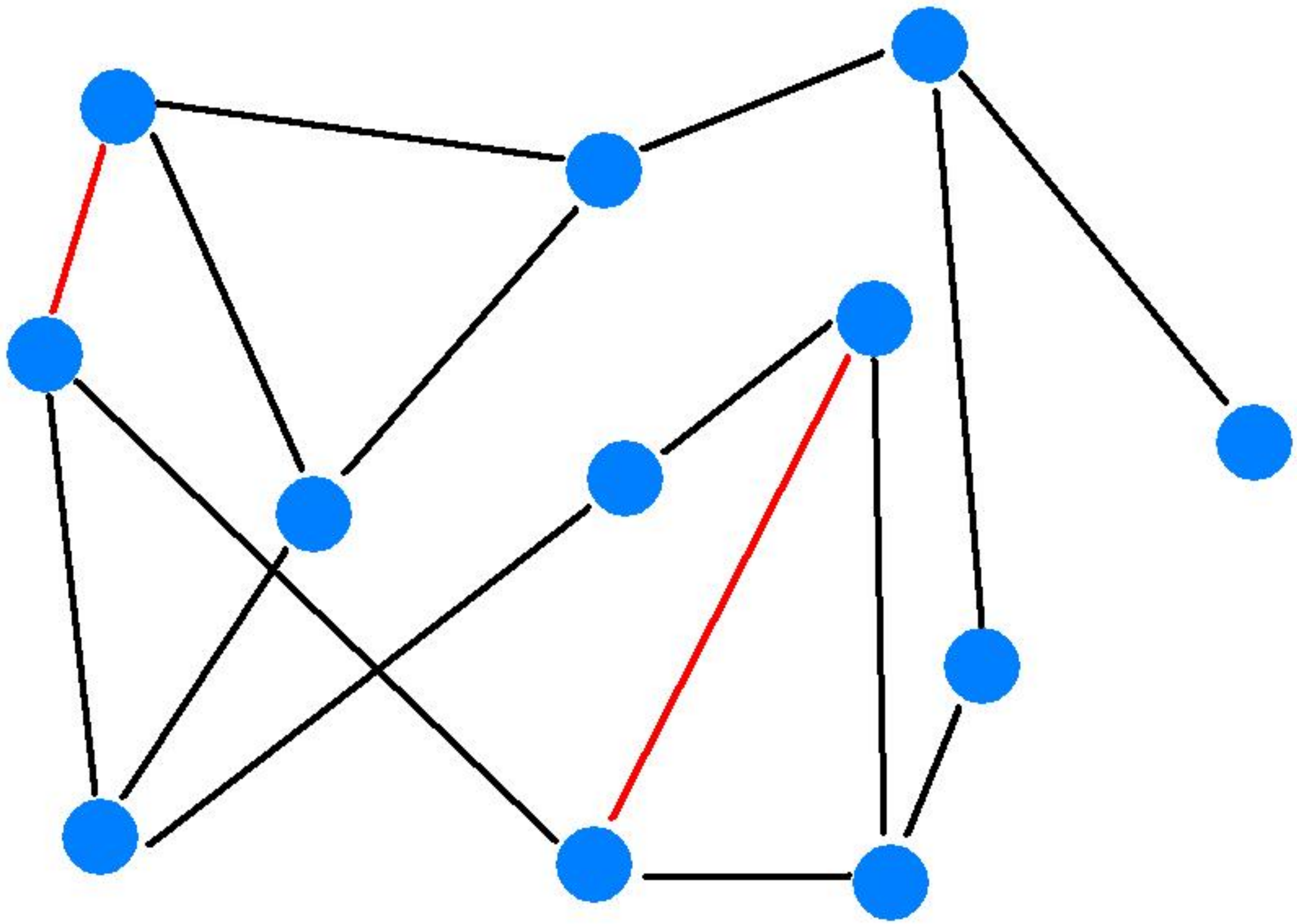
# HyParView Join Mechanism



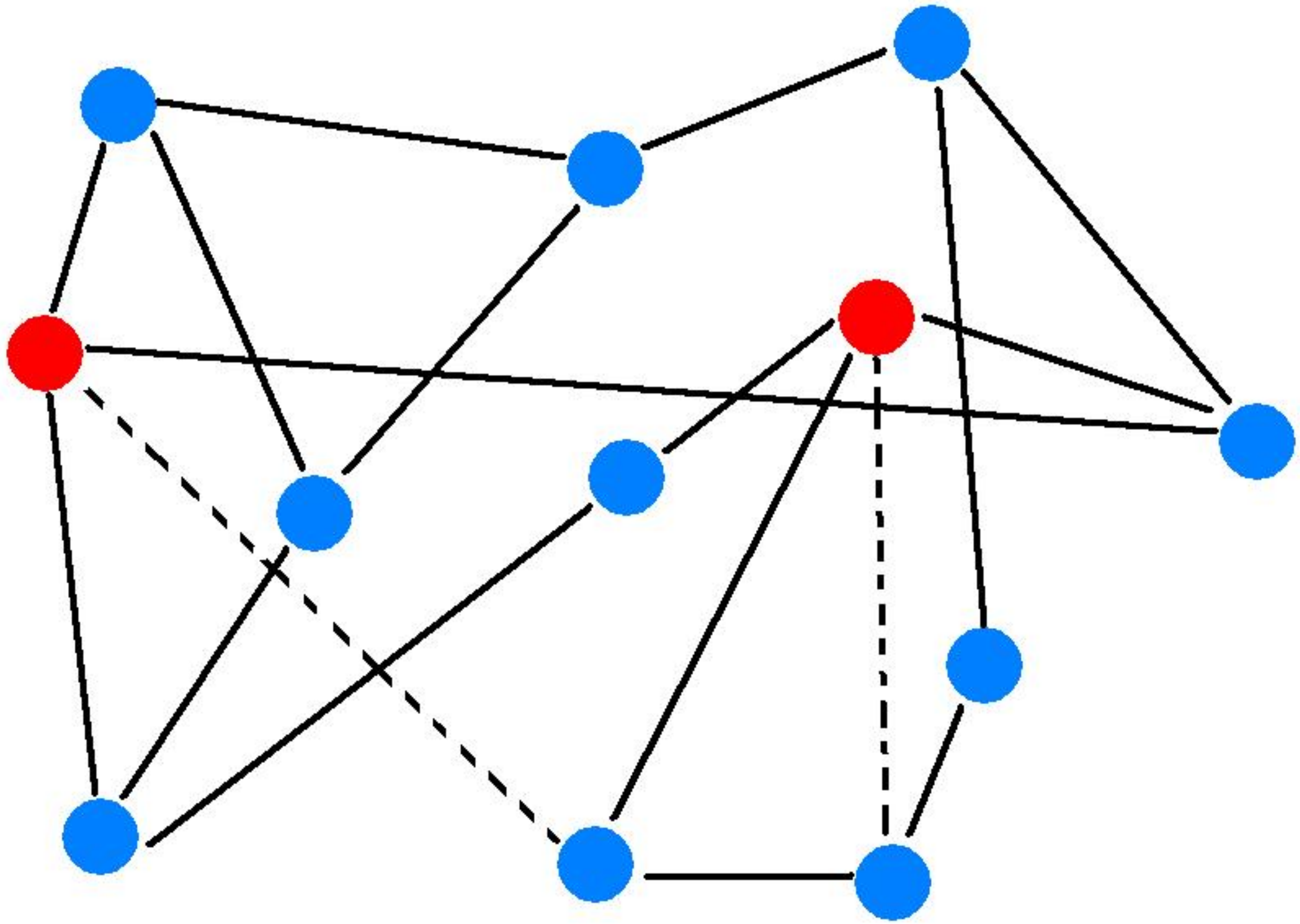
# HyParView Join Mechanism



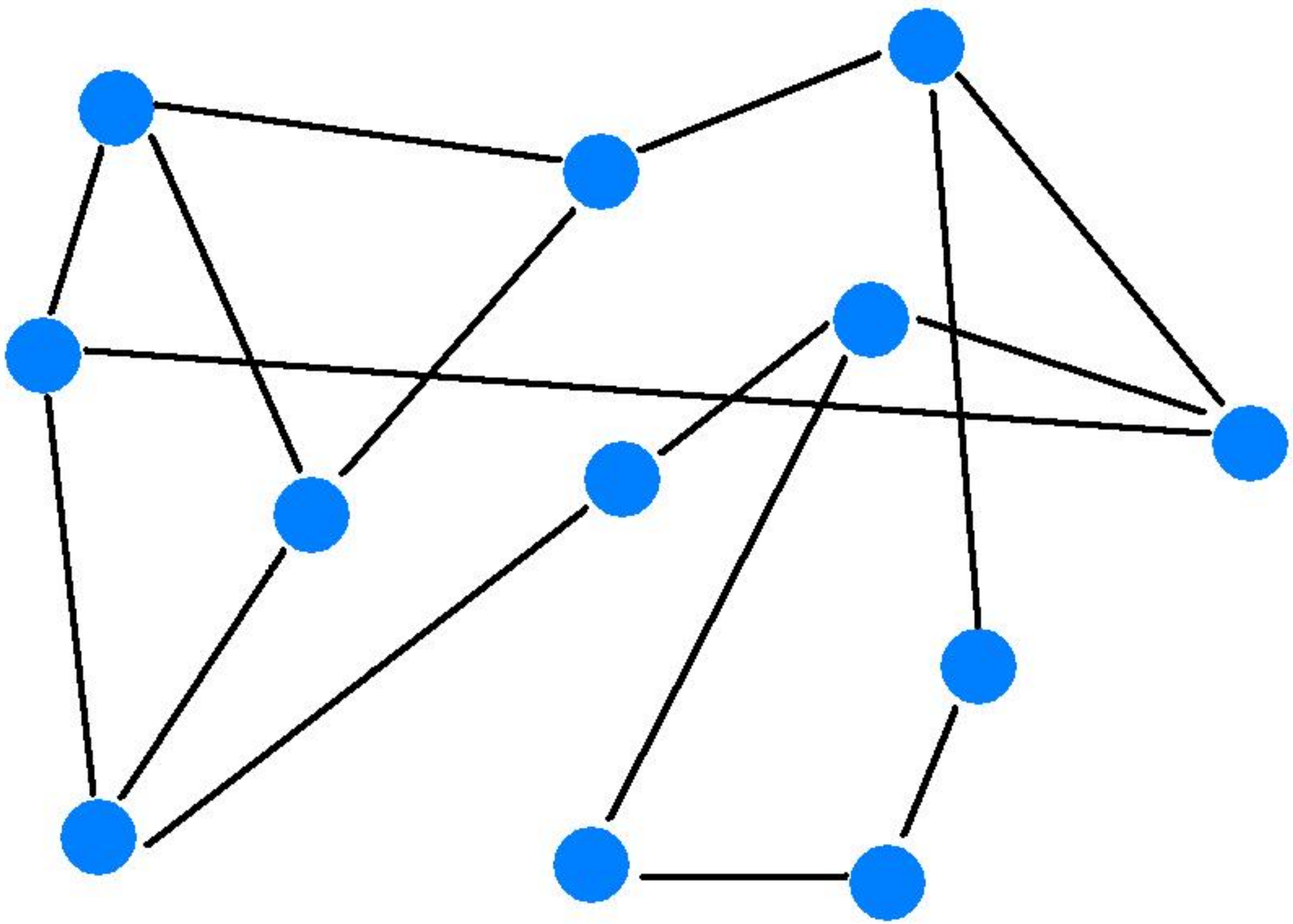
# HyParView Join Mechanism



# HyParView Join Mechanism



# HyParView Join Mechanism



# HyParView: Active View

- Maintained through a reactive strategy.
- TCP is used as an unreliable failure detector.
- When a node suspects a neighbor has failed:
  - The suspected neighbor is removed from the active view.
  - Random nodes from the passive view are contacted in order to promote them to the active view.

# HyPar View: Passive View

- Maintained through a cyclic strategy.
- Periodically each node starts a shuffle process with a random node.
- They exchange messages with samples from their partial views:
  - That node that starts the process send both himself and some elements from his active view.
  - His peer only send elements from his passive view.
- Both nodes update their passive views with the information received in the process.

# HyParView: The Algorithm

**Algorithm 1**: Join mechanism

**Data**:
myself: the identifier of the local node
activeView: a node active partial view
passiveView: a node passive view
contactNode: a node already present in the overlay
newNode: the node joining the overlay
ARWL: Active random walk length
PRWL: Passive random walk length

**upon** init **do**
    Send(JOIN, contactNode, myself);     addNodeActiveView(contactNode)

**upon** *Receive*(JOIN, newNode) **do**
    **call** addNodeActiveView(newNode)
    **foreach** $n \in$ activeView and $n \neq$ newNode **do**
        Send(FORWARDJOIN, $n$, newNode, ARWL, myself)

**upon** *Receive*(FORWARDJOIN, newNode, timeToLive, sender) **do**
    **if** timeToLive== 0||#activeView== 1 **then**
        **call** addNodeActiveView(newNode)
    **else**
        **if** timeToLive==PRWL **then**
            **call** addNodePassiveView(newNode)
        $n \longleftarrow n \in$ activeView and $n \neq$ sender
        Send(FORWARDJOIN, $n$, newNode, timeToLive-1, myself)

# HyParView: The Algorithm

**Algorithm 2**: View manipulation primitives

**Data**:
activeView: a node active partial view
passiveView: a node passive view

```
procedure dropRandomElementFromActiveView do
    n ⟵ n ∈ activeView
    Send(DISCONNECT, n, myself)
    activeView ⟵ activeView \{n}
    passiveView ⟵ passiveView ∪{n}

procedure addNodeActiveView(node) do
    if node ≠ myself and node ∉ activeView then
        if isfull(activeView) then
            call dropRandomElementFromActiveView
        activeView ⟵ activeView ∪ node

procedure addNodePassiveView(node) do
    if node ≠ myself and node ∉ activeView and node ∉ passiveView then
        if isfull(passiveView) then
            n ⟵ n ∈ passiveView
            passiveView ⟵ passiveView \{n}
        passiveView ⟵ passiveView ∪ node

upon Receive(DISCONNECT, peer) do
    if peer ∈ activeView then
        activeView ⟵ activeView \ {peer}
        call addNodePassiveView(peer)
```

# HyParView: The Algorithm

**Algorithm 2**: View manipulation primitives

**Data**:
activeView: a node active partial view
passiveView: a node passive view

```
procedure dropRandomElementFromActiveView do
    n ⟵ n ∈ activeView
    Send(DISCONNECT, n, myself)
    activeView ⟵ activeView \{n}
    passiveView ⟵ passiveView ∪{n}

procedure addNodeActiveView(node) do
    if node ≠ myself and node ∉ activeView then
        if isfull(activeView) then
            call dropRandomElementFromActiveView
        activeView ⟵ activeView ∪ node

procedure addNodePassiveView(node) do
    if node ≠ myself and node ∉ activeView and node ∉ passiveView then
        if isfull(passiveView) then
            n ⟵ n ∈ passiveView
            passiveView ⟵ passiveView \{n}
        passiveView ⟵ passiveView ∪ node

upon Receive(DISCONNECT, peer) do
    if peer ∈ activeView then
        activeView ⟵ activeView \ {peer}
        call addNodePassiveView(peer)
```

# HyParView: Message Dissemination

- The size of the active view is *fanout*+1.
- Whenever a node receives a message for the first time:
  - It forwards it to all nodes in the active view.
  - Except the one from which he received the message.
- Message dissemination is performed:
  - Deterministically in a random overlay...
  - ...by flooding an overlay with small degree.
  - Each node tests all its active view each time it forwards a message.

# HyParView: Message Dissemination

**Algorithm 3**: Eager push protocol

**Data**:
myself: the identifier of the local node
receivedMsgs: a list of received messages identifiers
f: the fanout value

```
upon event Broadcast(m) do
    mID ⟵ hash(m + myself)
    peerList ⟵ getPeer(f,null)
    foreach p ∈ peerList do
        trigger Send(GOSSIP, p, m, mID, myself)
    trigger Deliver(m)
    receivedMsgs ⟵ receivedMsgs ∪ {mID}

upon event Receive(GOSSIP, m, mID, sender) do
    if mID ∉ receivedMsgs then
        receivedMsgs ⟵ receivedMsgs ∪ {mID}
        trigger Deliver(m)
        peerList ⟵ getPeer(f,sender)
        foreach p ∈ peerList do
            trigger Send(GOSSIP, p, m, mID, myself)
```

# HyParView: Is this really Good?

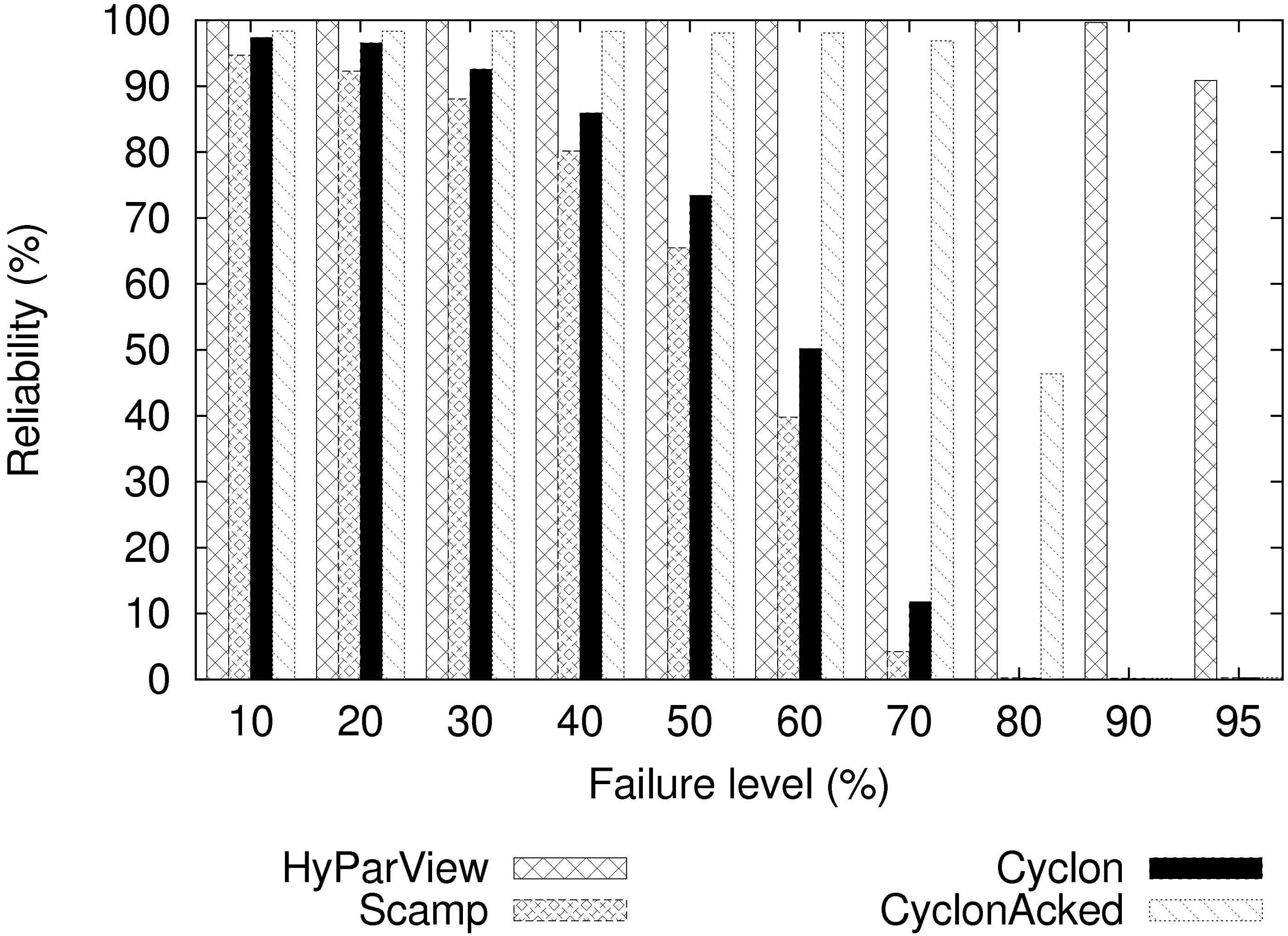



Figure 4.2: Average reliability for 1000 messages

# Class structure:

- One Unstructured Overlay: HyParView.

- **One Partially Unstructured Overlay: Plumtree.**

# Class structure:

- One Unstructured Overlay: HyParView.

- **<u>One Partially Unstructured Overlay: Plumtree.</u>**

Plumtree is not required for the Phase 1 of the course project, but you might want to think about it on the second phase.

# Plumtree Protocol

26th IEEE International Symposium on Reliable Distributed Systems

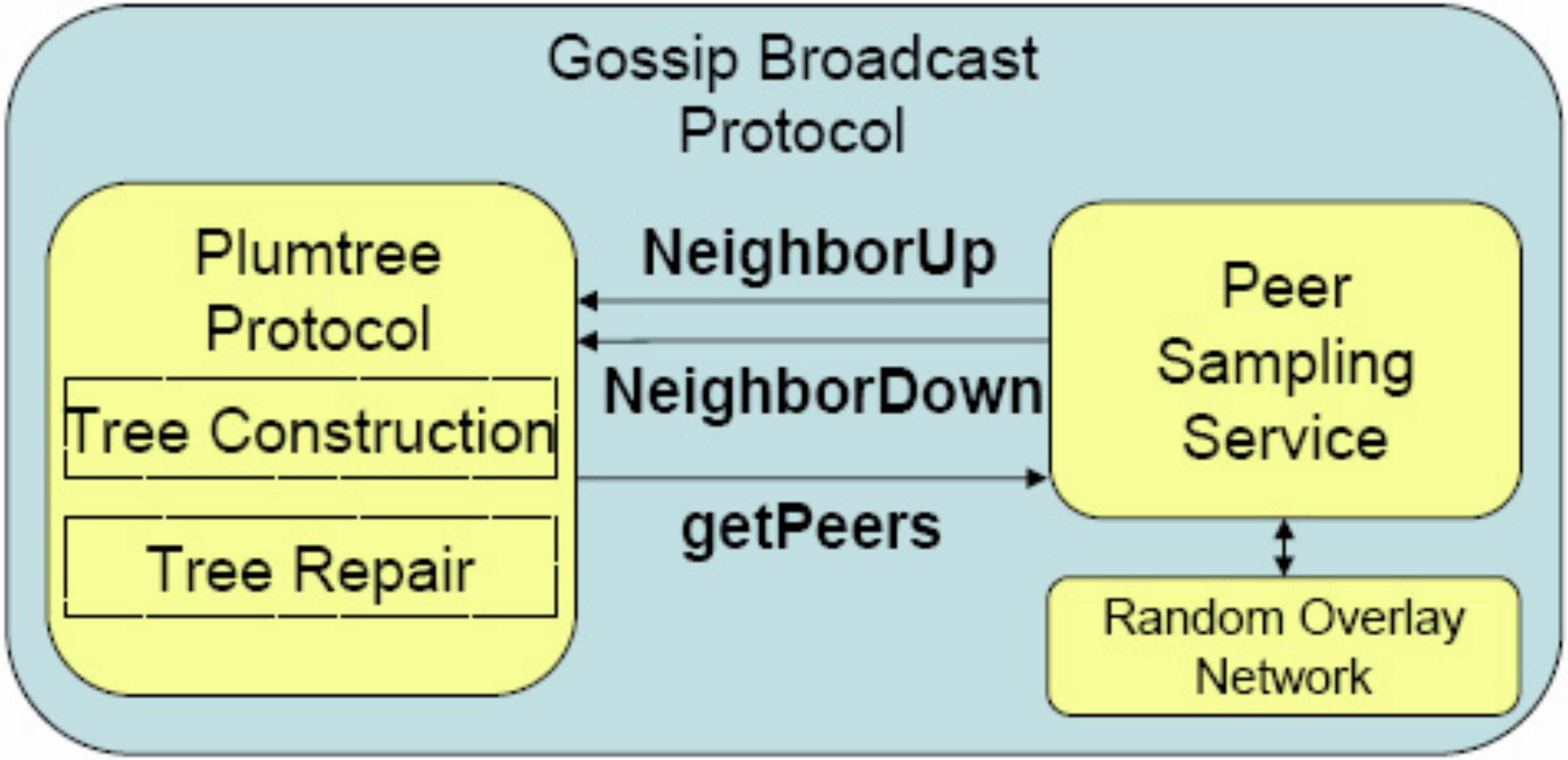
**Epidemic Broadcast Trees** *

João Leitão
University of Lisbon
jleitao@lasige.di.fc.ul.pt

José Pereira
University of Minho
jop@di.uminho.pt

Luís Rodrigues
University of Lisbon
ler@di.fc.ul.pt

# Plumtree Architecture



# Plumtree Requirements of the Unstructured Overlay Network

- Ensure connectivity
- Scalable
- Reactive membership

**Optionally:**

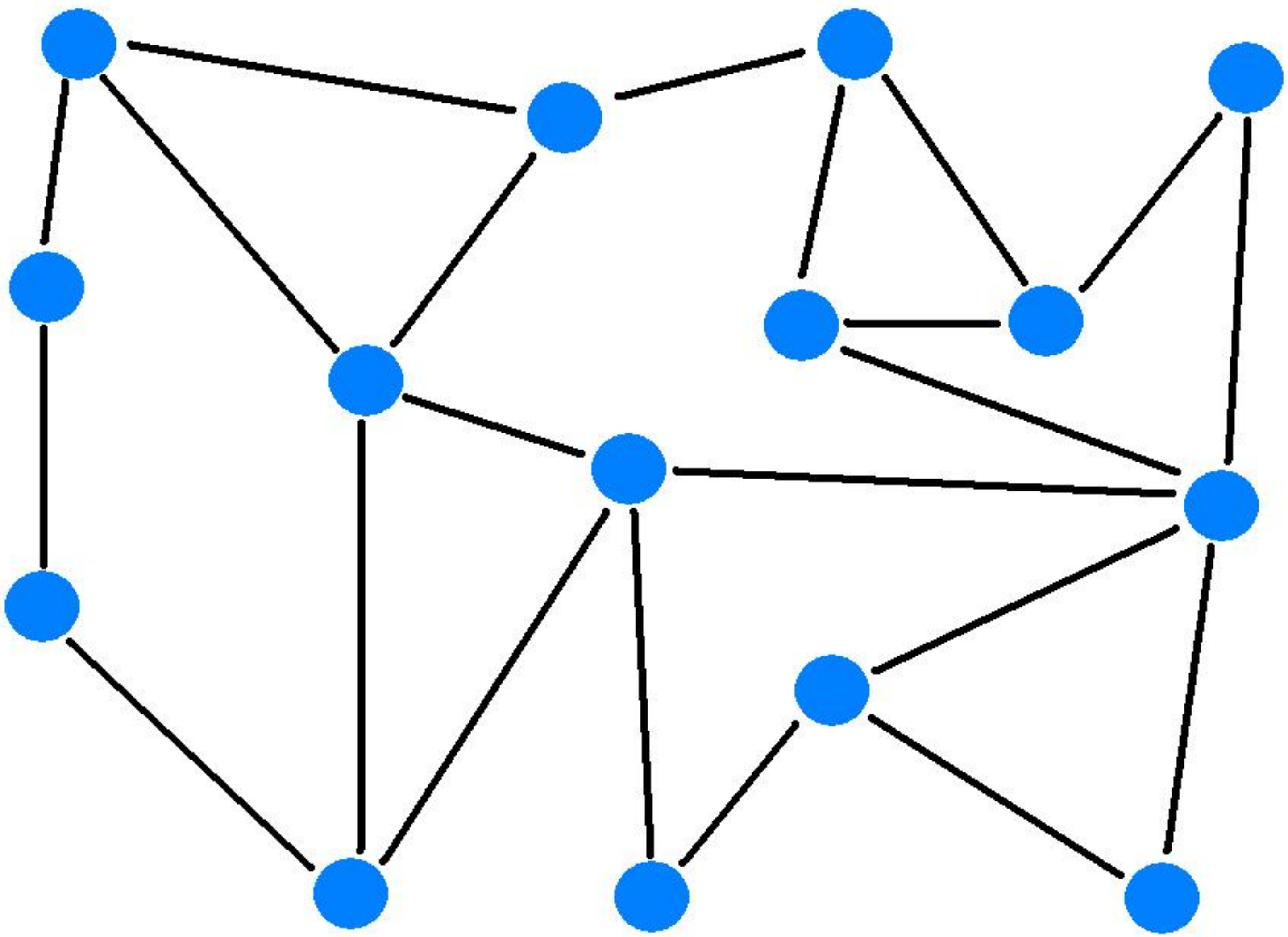
- Symmetric partial views

# Plumtree: The Intuition

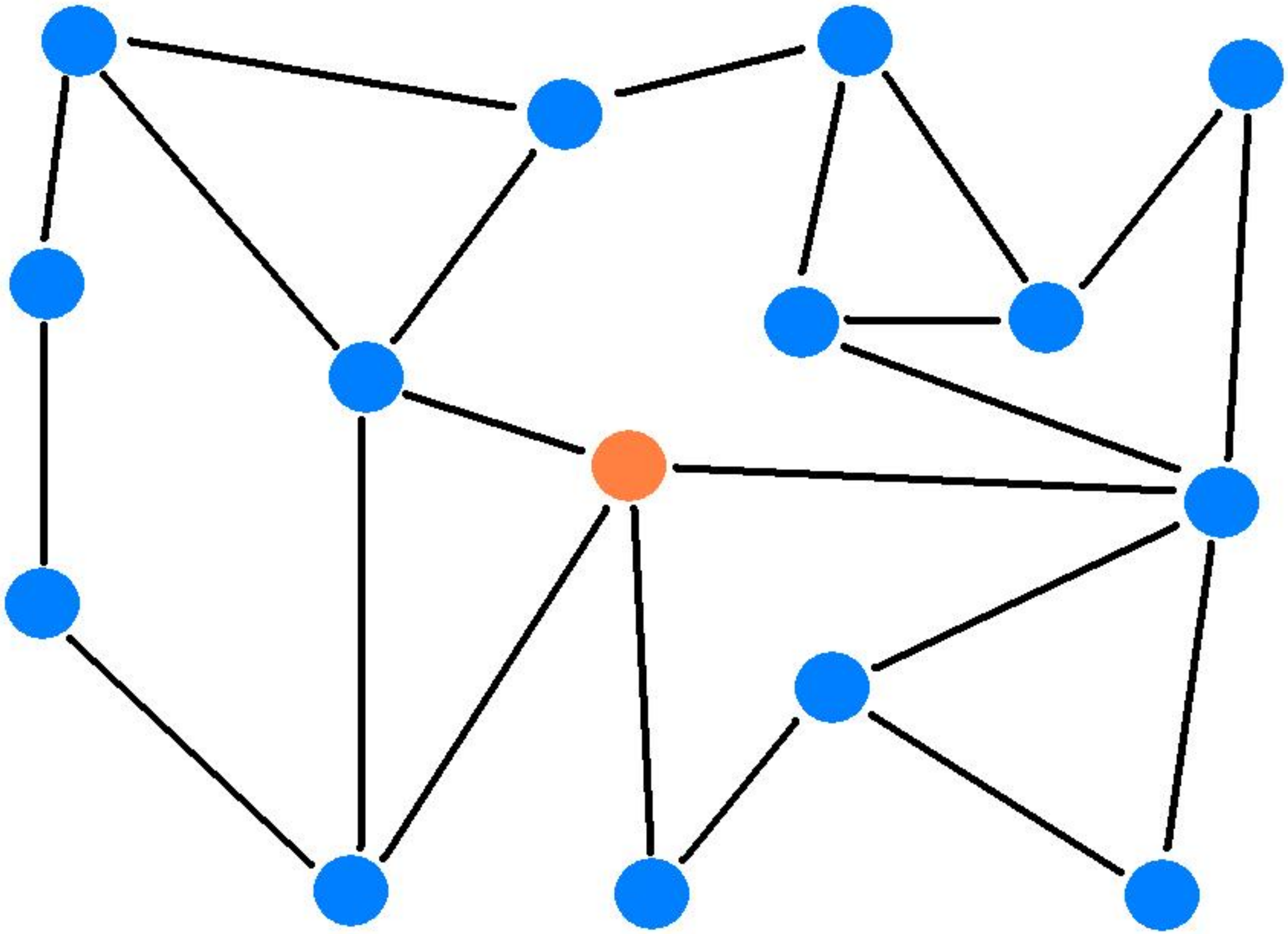
- Think about a gossip protocol (eager push) operating on top of a static overlay network, where you actually use flooding.

- What is going to happen here?

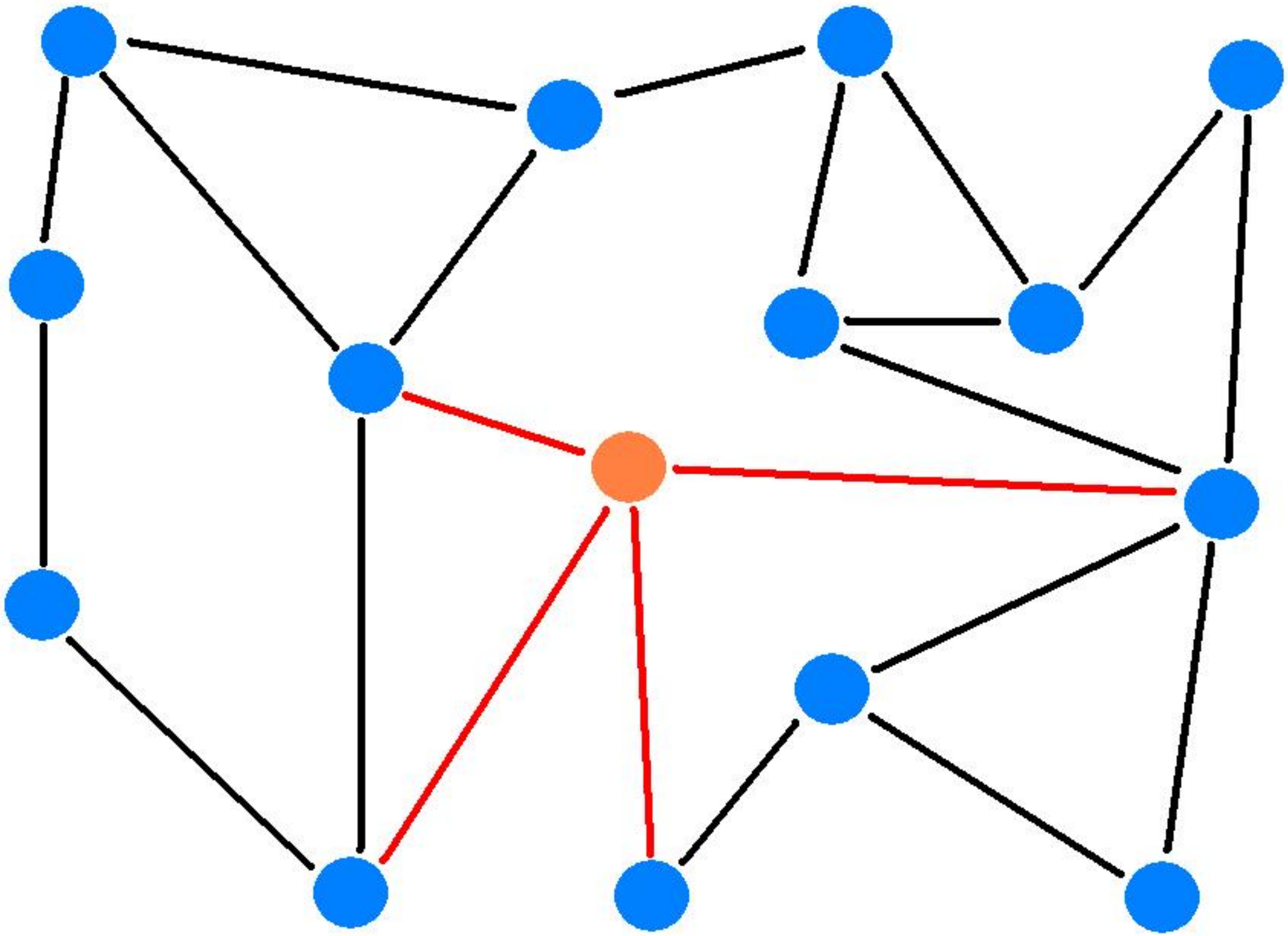
# Plumtree: The Intuition



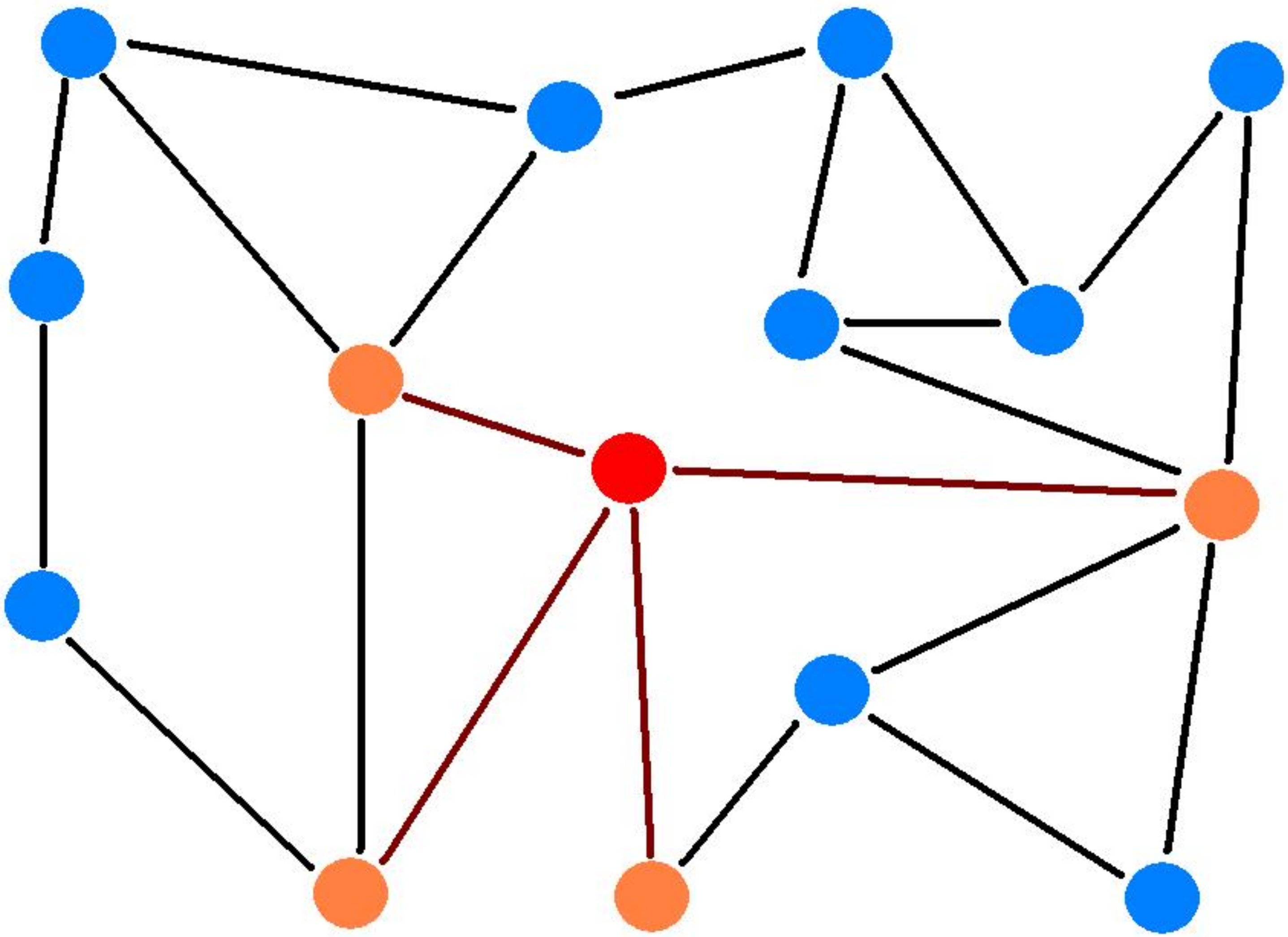
# Plumtree: The Intuition



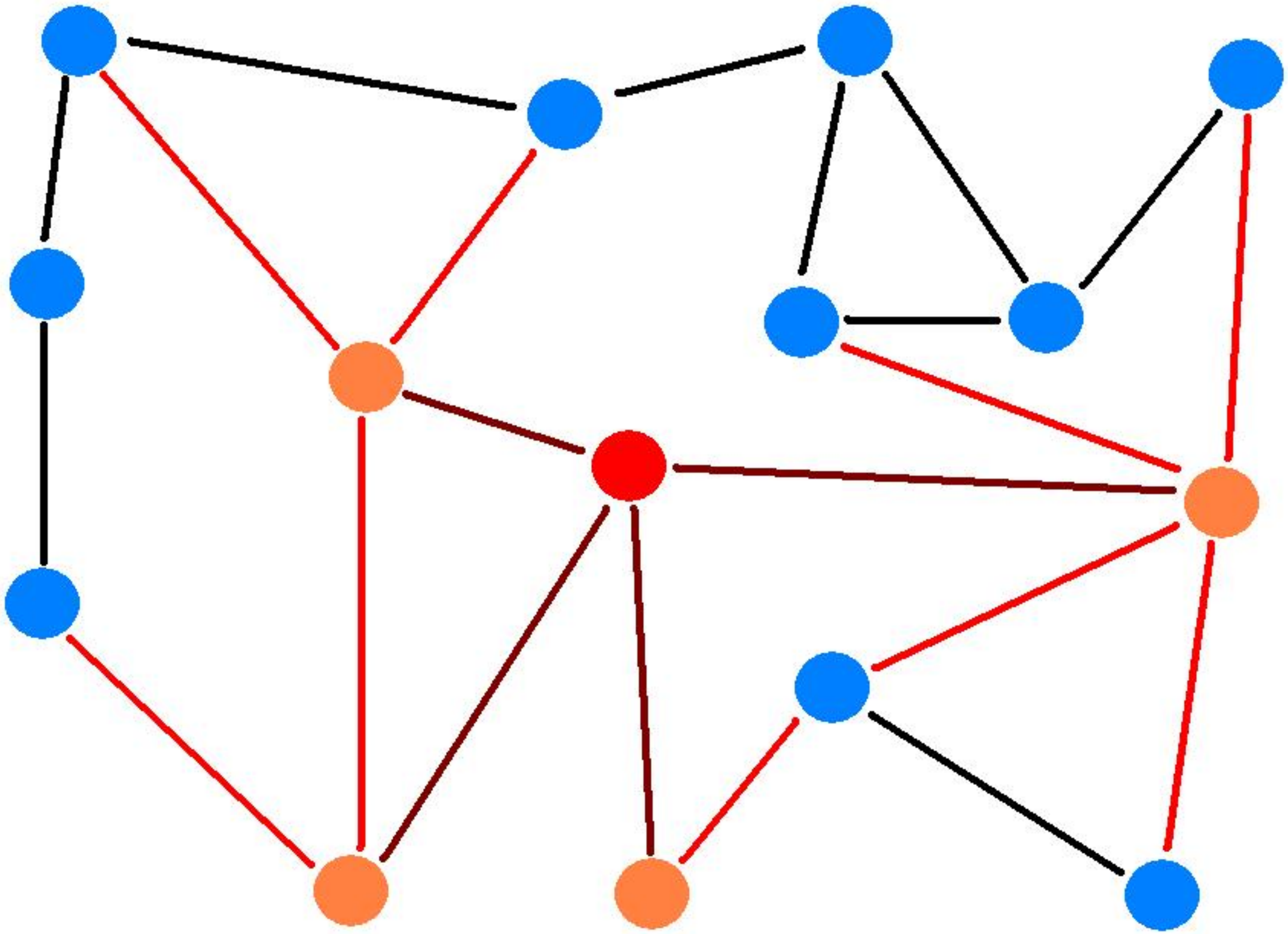
# Plumtree: The Intuition



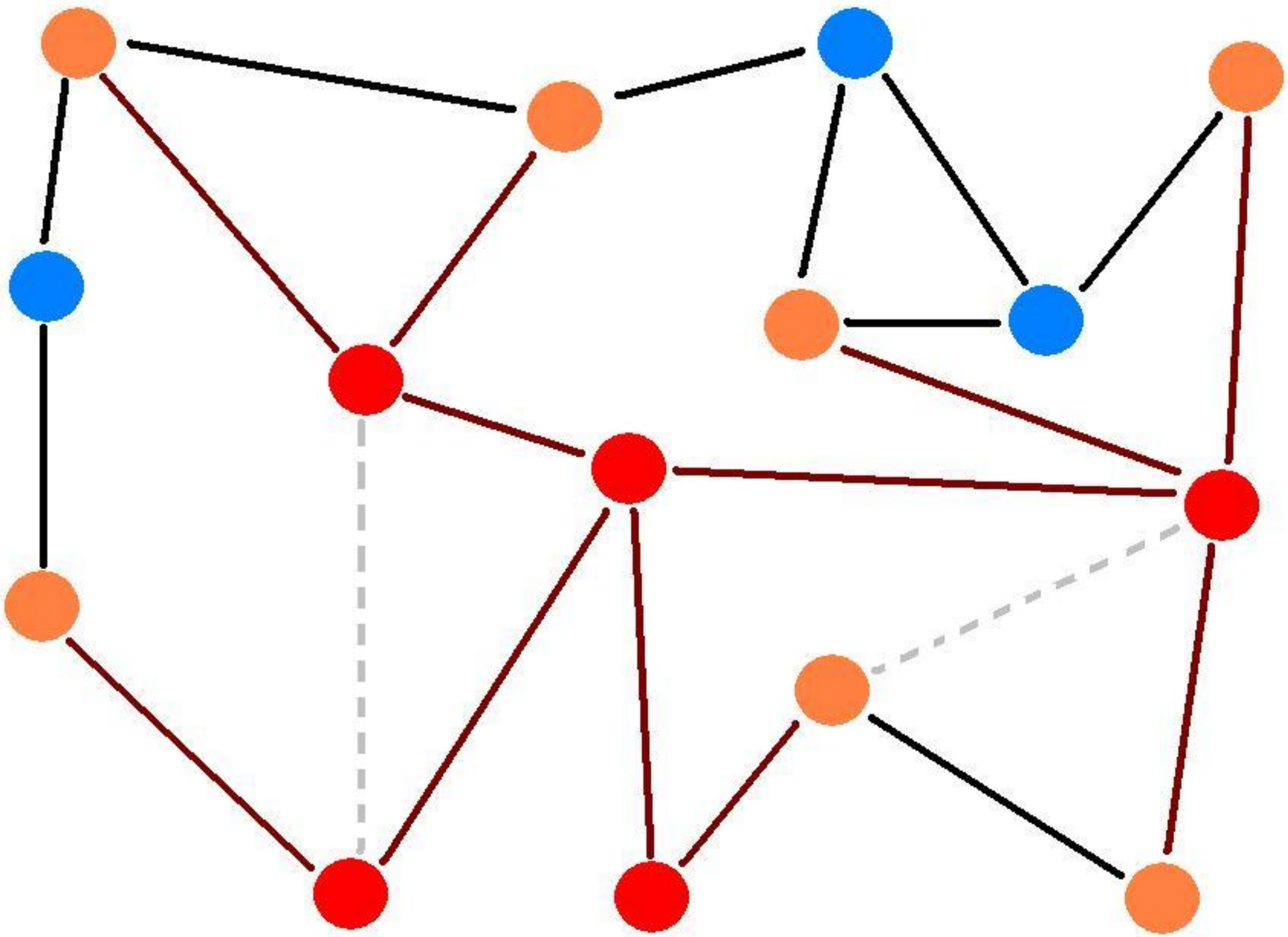
# Plumtree: The Intuition



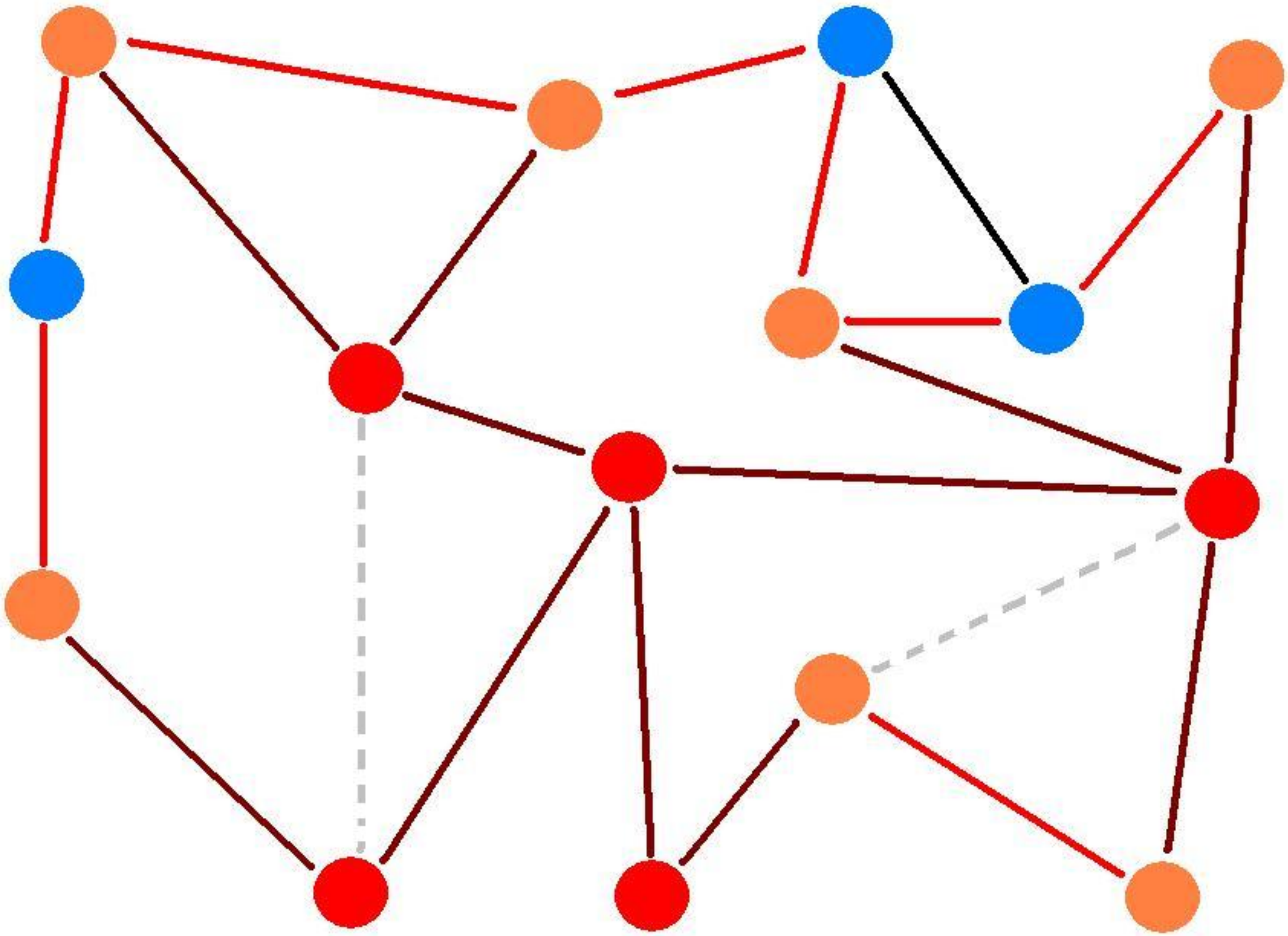
# Plumtree: The Intuition



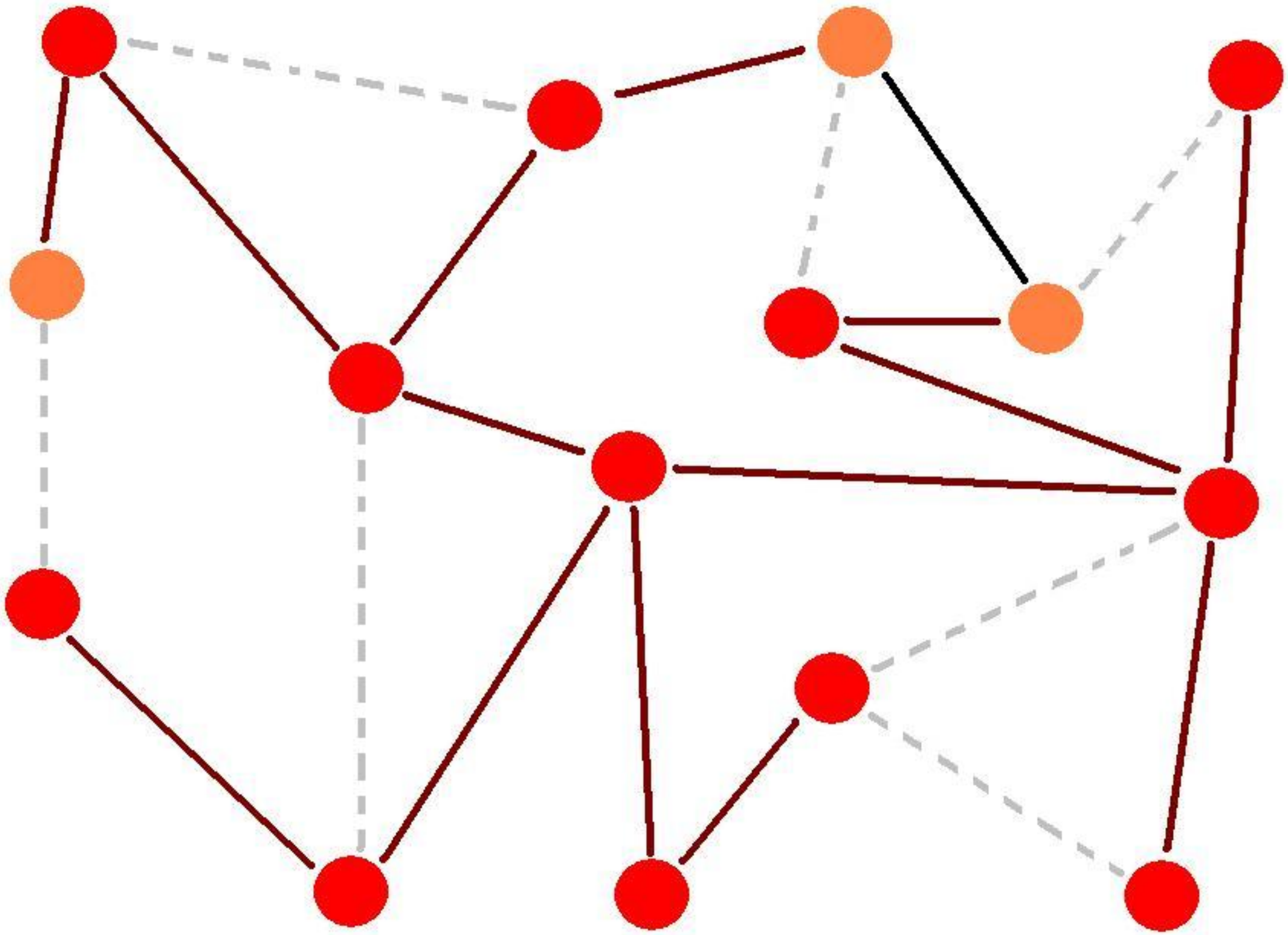
# Plumtree: The Intuition



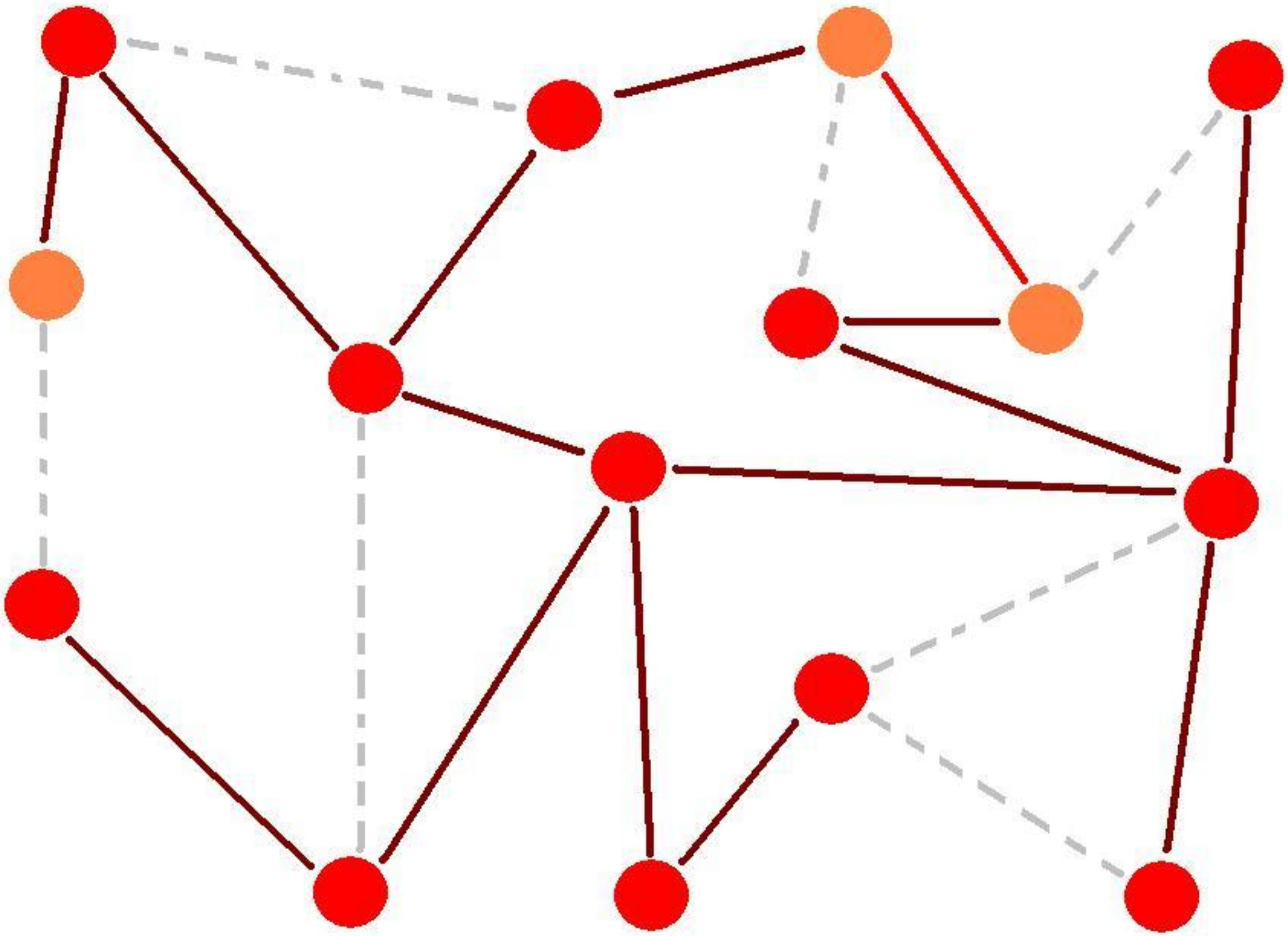
# Plumtree: The Intuition



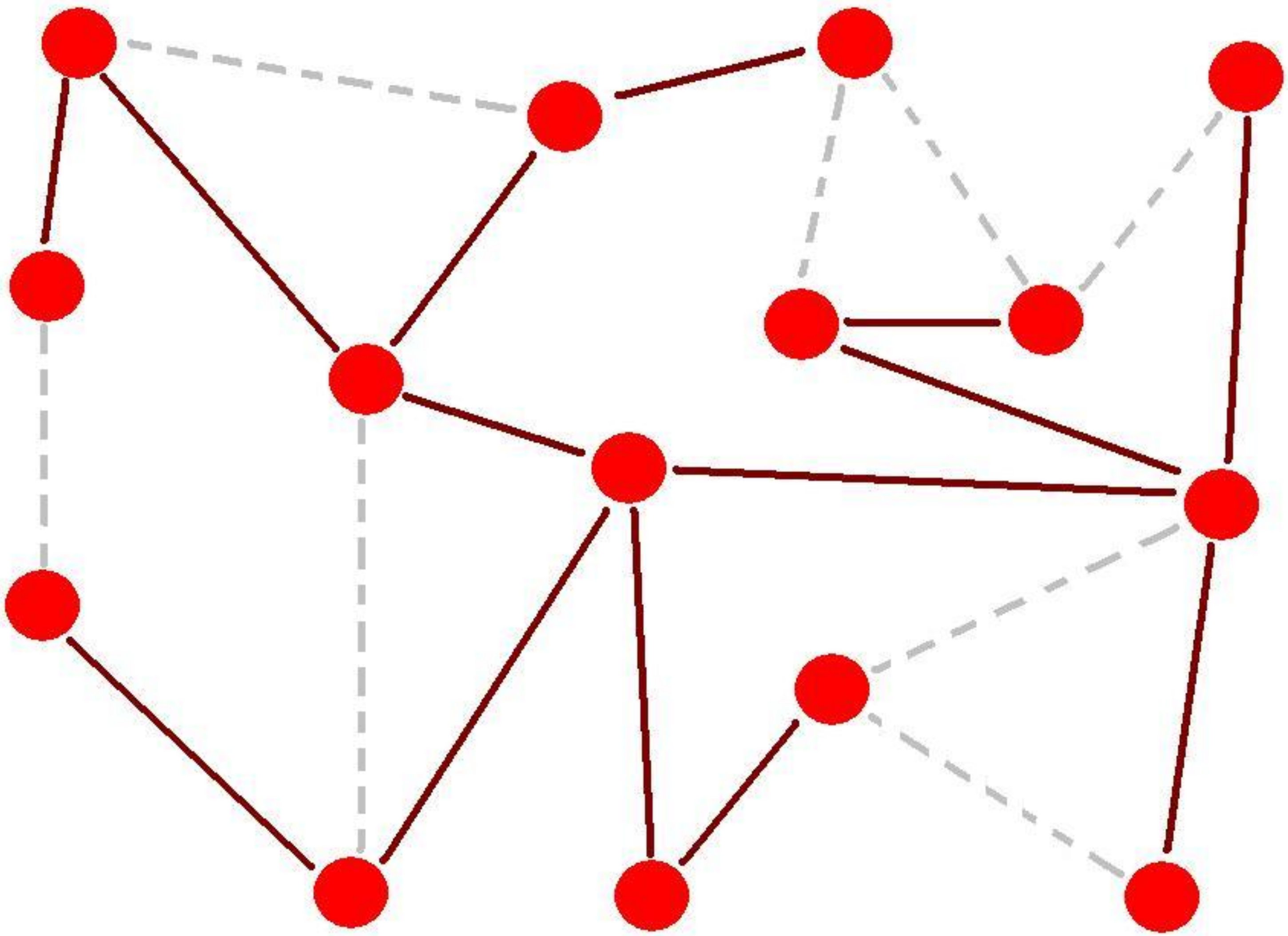
# Plumtree: The Intuition



# Plumtree: The Intuition



# Plumtree: The Intuition



# Plumtree Overview

Plumtree: **p**ush-**l**azy-p**u**sh **m**ulticast **tree**.

- Protocol operates as any gossip protocol, each node gossips with $t$ neighbors provided by a peer sampling service by combining eager push and lazy push.
- The support the operation of the protocol, two distinct sets of neighbors are maintained:
  - eagerPushPeers
  - lazyPushPeers
- During initialization $t$ nodes are obtained from the peer sampling service and inserted in the eagerPushPeers.
- TCP is used between nodes as it offers a better reliability and an additional fault tolerance mechanism.

# Plumtree: Tree Construction

- Intuition: Use the links in the overlay that generated a message delivery.
- After initialization all links in the overlay are candidates do belong to the broadcast tree.
- When a message is broadcasted, all links used to disseminate a redundant message are pruned from the tree.
- If the random overlay is connected, after the dissemination of a message the tree will cover all nodes.

# Plumtree: Tree Repair

- If a node fails some nodes might become disconnected from the embedded tree.
- When a node receives a announcement for a payload message it did not received it starts a timer.
- If the timer expires before receiving the payload message the node explicitly request the payload.
- Moves the node to whom it makes the request to its eagerPushPeers list.
- The node that receives the request sends the payload.
- Moves the requester to its eagerPushPeers list.
- At the end of this process the embedded tree is repaired.

# Plumtree: The Algorithm

- State

**Algorithm 4**: Internal data structure

**Data**:
myself: the identifier of the local node
receivedMsgs: a list of received messages identifiers
f: the push fanout value
eagerPushPeers: a list of neighbors whom links form the spanning tree
lazyPushPeers: a list of neighbors whom links does not belong to the spanning tree
lazyQueue: a list of tuples {mID,node,round}

# Plumtree: The Algorithm

- Tree Construction

**Algorithm 5**: Spanning tree construction algorithm

```
procedure dispatch do
    announcements ⟵ policy (lazyQueue) //set of IHave messages
    trigger Send(announcements)
    lazyQueue ⟵ lazyQueue \announcements

procedure EagerPush (m, mID, round, sender) do
    foreach p ∈ eagerPushPeers: p ≠sender do
        trigger Send(Gossip, p, m, mID, round, myself)

procedure LazyPush (m, mID, round, sender) do
    foreach p ∈ lazyPushPeers: p ≠sender do
        lazyQueue ⟵ (textscIHave(p, m, mID, round, myself)
    call dispatch()

upon event Init do
    eagerPushPeers ⟵ getPeer(f)
    lazyPushPeers ⟵ ∅
    lazyQueue ⟵ ∅
    missing ⟵ ∅
    receivedMsgs ⟵ ∅

upon event Broadcast(m) do
    mID ⟵ hash(m+myself)
    call EagerPush (m, mID, 0, myself)
    call lazyPush (m, mID, 0, myself)
    trigger Deliver(m)
    receivedMsgs ⟵ receivedMsgs ∪ {mID}

upon event Receive(Gossip, m, mID, round, sender) do
    if mID ∉ receivedMsgs then
        trigger Deliver(m)
        receivedMsgs ⟵ receivedMsgs ∪ {mID}
        if ∃ (id,node,r) ∈ missing :id=mID then
            cancel Timer(mID)
        call EagerPush (m, mID, round+1, myself)
        call lazyPush (m, mID, round+1, myself)
        eagerPushPeers ⟵ eagerPushPeers ∪ {sender}
        lazyPushPeers ⟵ lazyPushPeers \ {sender}
        call Optimize (m, mID, round, sender) // optional
    else
        eagerPushPeers ⟵ eagerPushPeers \ {sender}
        lazyPushPeers ⟵ lazyPushPeers ∪ {sender}
        trigger Send(Prune, sender, myself)

upon event Receive(Prune, sender) do
        eagerPushPeers ⟵ eagerPushPeers \ {sender}
        lazyPushPeers ⟵ lazyPushPeers ∪ {sender}
```

# Plumtree: The Algorithm

- Tree Repair

**Algorithm 6**: Spanning tree repair algorithm

```
upon event Receive(IHave, mID, round, sender) do
    if mID ∉ receivedMsgs do
        if ∄ Timer(id): id=mID do
            setup Timer(mID, timeout_1)
        missing ⟵ missing ∪ {(mID,sender,round)}

upon event Timer(mID) do
    setup Timer(mID, timeout_2)
    (mID,node,round) ⟵ removeFirstAnnouncement(missing, mID)
    eagerPushPeers ⟵ eagerPushPeers ∪ {node}
    lazyPushPeers ⟵ lazyPushPeers \ {node}
    trigger Send(Graft,node,mID,round,myself)

upon event Receive(Graft, mID, round, sender) do
    eagerPushPeers ⟵ eagerPushPeers ∪ {sender}
    lazyPushPeers ⟵ lazyPushPeers \ {sender}
    if mID ∈ receivedMsgs do
        trigger Send(Gossip, sender, m, mID, round, myself)
```

# Plumtree: Multiple Senders

## Small number of senders

- One Plumtree instance for each sender.
- Lower latency.

## Large number of senders

- One single Plumtree instance shared by all senders.
- Higher latency.

# Summary

- You know everything you need to know to do the entire phase 1 of the project.
- We already identified some (expected) limitations that we will find in this phase, start thinking how we can address those limitations (that is going to be one of the focus on phase 2).