Algorithms and Distributed Systems 2022/2023 (Lecture Three)

MIEI - Integrated Master in Computer Science and Informatics

MEI – Master in Computer Science and Informatics

Specialization block

Nuno Preguiça (nmp@fct.unl.pt)

Alex Davidson (a.davidson@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Acknowledgments

- Slides mostly from João Leitão.

Lecture structure:

- Unstructured Overlay Networks Usage
- Resource Location Problem (Exact Match Queries)
- Structured Overlay Networks.
- Partially Structured Overlay Networks.

Unstructured Overlay Networks

- Main properties:
  - Random topology.
  - Low maintenance cost.
  - Eventual Global Connectivity.

Unstructured Overlay Networks

- Main properties:
  - Random topology.
  - Low maintenance cost.
  - Eventual Global Connectivity.
- Why is this useful?

Unstructured Overlay Networks

- Main properties:
  - Random topology.
  - Low maintenance cost.
  - Eventual Global Connectivity.
- Why is this useful?
  - Message dissemination (Broadcast).

Unstructured Overlay Networks

- Main properties:
  - Random topology.
  - Low maintenance cost.
  - Eventual Global Connectivity.
- Why is this useful?
  - Message dissemination (Broadcast).
  - Replication of data across large number of nodes.
  - Monitoring.
  - Resource Location.

Unstructured Overlay Networks

- Main properties:
  - Random topology.
  - Low maintenance cost.
  - Eventual Global Connectivity.
- Why is this useful?
  - Message dissemination (Broadcast).
  - Replication of data across large number of nodes.
  - Monitoring.
  - **Resource Location.**

Resource Location

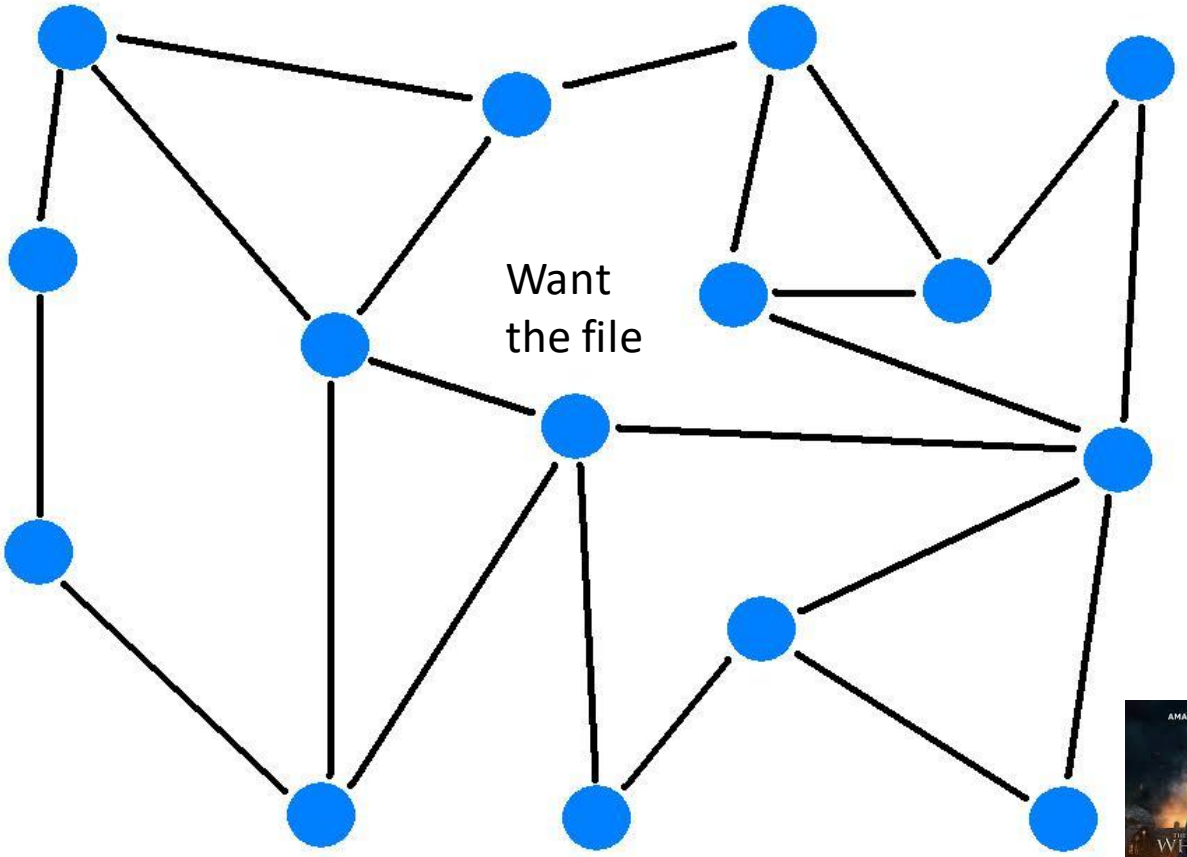
- A Definition:

Given a set of processes containing different sets of resources, locate the processes that contain resources with a given set of properties.

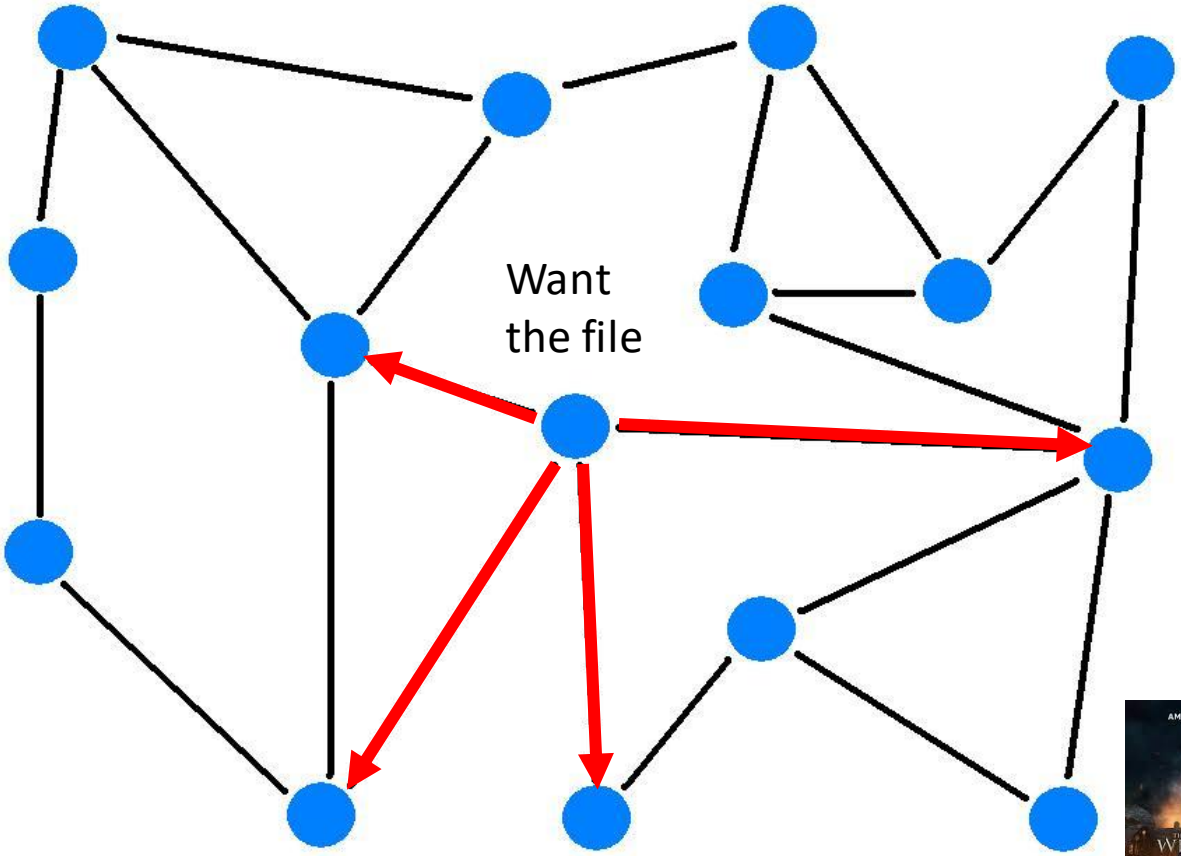
- One possible concretization:
  - File Sharing Applications.
  - Processes own a set of files that have properties.
  - Locate the processes (and files) that match a given set of criteria (e.g., Extension = mkv, size >= 1Gb, Name contains “The Wheel of Time” and “S01E01”).

How do we solve the resource location problem?

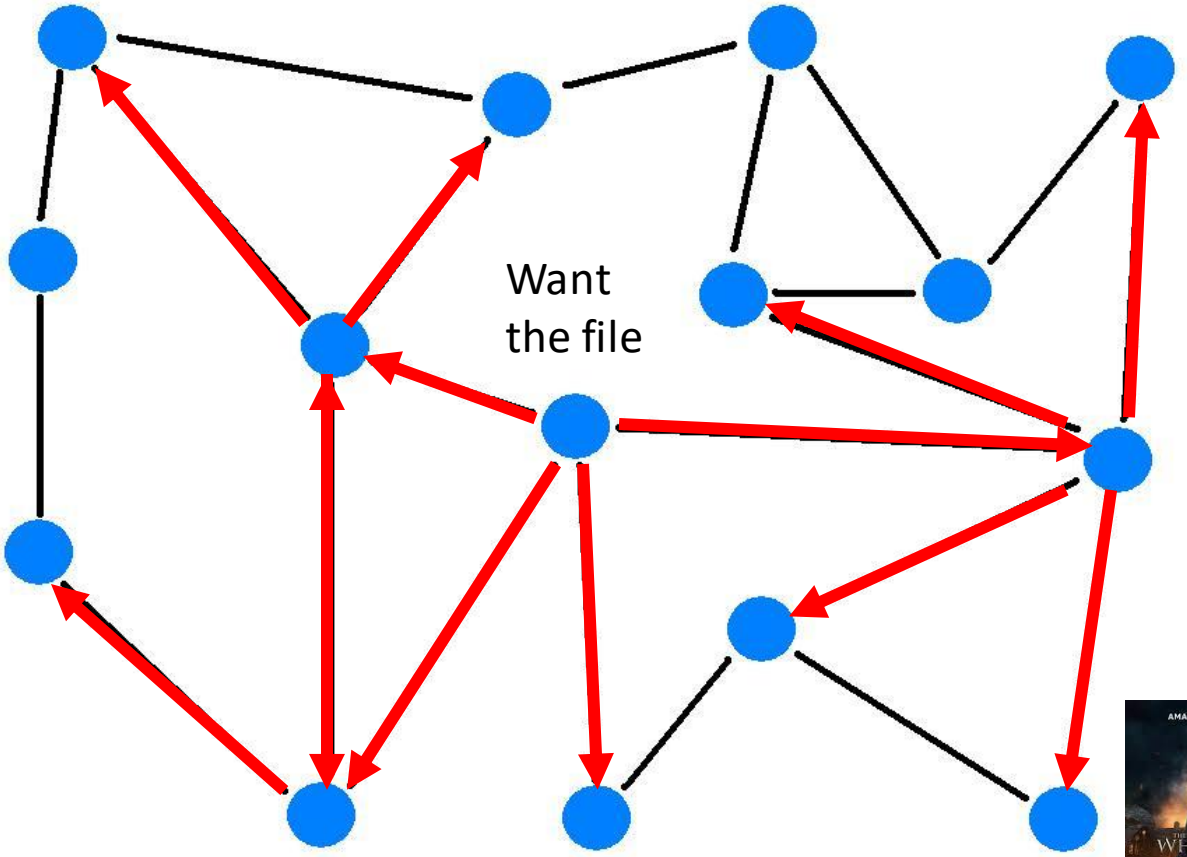
How do we solve the resource location problem?



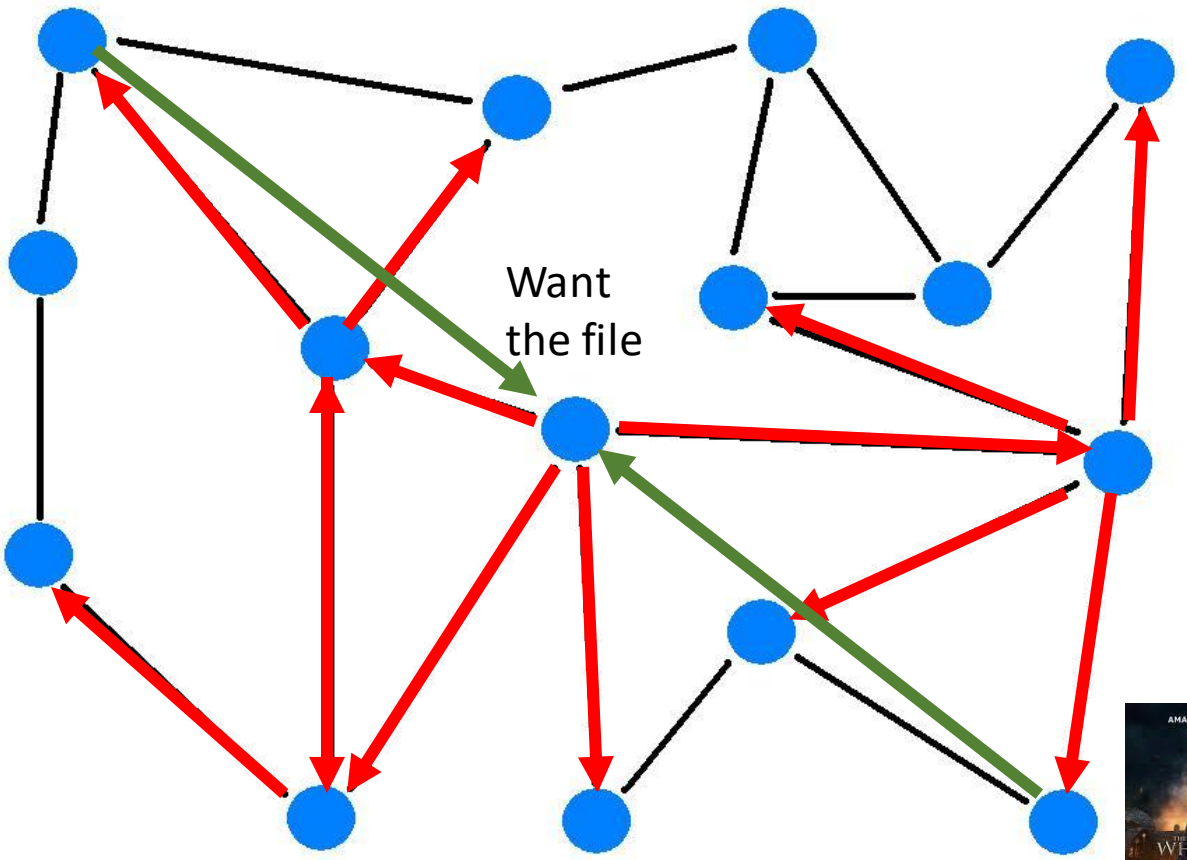
How do we solve the resource location problem?



How do we solve the resource location problem?

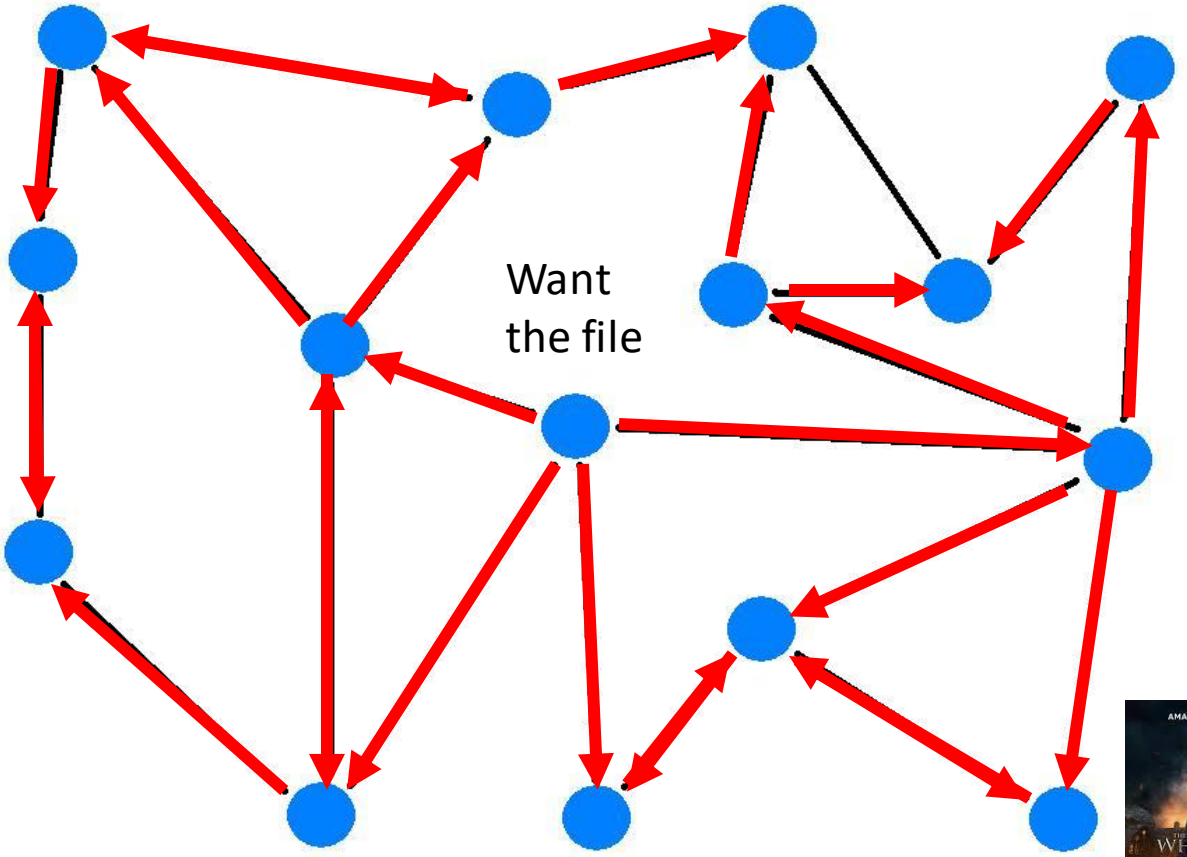


How do we solve the resource location problem?



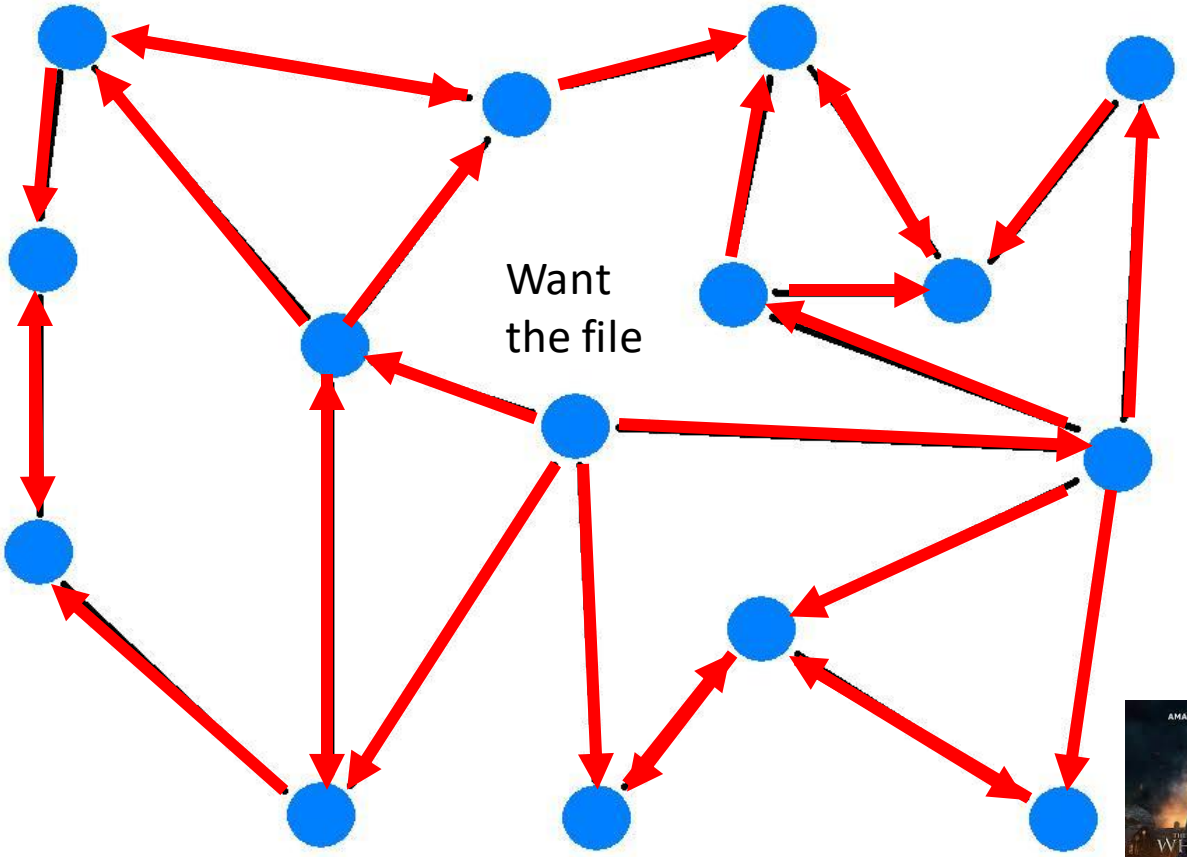
Since we do not know where else other resources might be, we continue…

How do we solve the resource location problem?



Since we do not know where else other resources might be, we continue…

How do we solve the resource location problem?



Since we do not know where else other resources might be, we continue…

Resource Location

- That is the gossip variant called Flood.
- This was the solution implemented by a protocol named Gnutella (Version 1.0)
- It was used to support the search in file sharing applications in the first decade of 2000s (Similar to Shareaza, Limewire, eMule)
- What is the problem with this solution?

Resource Location

- That is the gossip variant called Flood.
- This was the solution implemented by a protocol named Gnutella (Version 1.0)
- It was used to support the search in file sharing applications in the first decade of 2000s (Similar to Shareaza, Limewire, eMule)
- What is the problem with this solution?
  - Too many messages are generated and forwarded among processes, which might overload them…

Resource Location

- What could we do to address the problem of too many messages being disseminated?

Resource Location

- What could we do to address the problem of too many messages being disseminated?
- Two solutions were employed in the past:
  - Flooding with limited horizon.
  - Super-Peer Networks (Popularized in Gnutella V2).

Flooding with limited horizon

- When a query message is disseminated, it carries a value (e.g, hopCount) that is initially set to zero.
- This value is incremented whenever the message is retransmitted.
- Processes stop forwarding the message when the hopCount value reaches a given threshold.

(symmetrically, can be done with a time to live – ttl – value that starts with some value and is decremented at each step)

Flooding with limited horizon

- When a query message is disseminated, it carries a value (e.g, hopCount) that is initially set to zero.
- This value is incremented whenever the message is retransmitted.
- Processes stop forwarding the message when the hopCount value reaches a given threshold.
- Why is this good ->

- Why is this bad ->

Flooding with limited horizon

- When a query message is disseminated, it carries a value (e.g, hopCount) that is initially set to zero.
- This value is incremented whenever the message is retransmitted.
- Processes stop forwarding the message when the hopCount value reaches a given threshold.
- Why is this good -> Most messages are generated later in the dissemination (and most redundant messages also).
- Why is this bad -> You no longer have guarantees of finding all relevant resources.

Unstructured Overlays based on Super-Peers.

- A small fraction of processes (those that have more resources, are more powerful, or simply more stable) are promoted to Super-Peers.
- Super-Peers form an unstructured overlay among them.
- Regular processes connect to a super-peer and transmit to it the index of its resources.
- Queries are forwarded to the super-peer and then disseminated only among super-peers.

Unstructured Overlays based on Super-Peers.

- A small fraction of processes (those that have more resources, are more porwerfull, or simply more stable) are promoted to Super-Peers.
- Super-Peers form an unstructured overlay among them.
- Regular processes connect to a super-peer and transmit to it the index of its resources.
- Queries are forwarded to the super-peer and then are disseminated only among super-peers.
- Why is this good ->
- Why is this bad ->

Unstructured Overlays based on Super-Peers.

- A small fraction of processes (those that have more resources, are more porwerfull, or simply more stable) are promoted to Super-Peers.
- Super-Peers form an unstructured overlay among them.
- Regular processes connect to a super-peer and transmit to it the index of its resources.
- Queries are forwarded to the super-peer and then are disseminated only among super-peers.
- Why is this good -> Significantly reduces the number of messages.
- Why is this bad -> How do you decide which process should be a super-peer? Load in the system is highly unbalanced.

How to manage the underlying overlay?

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

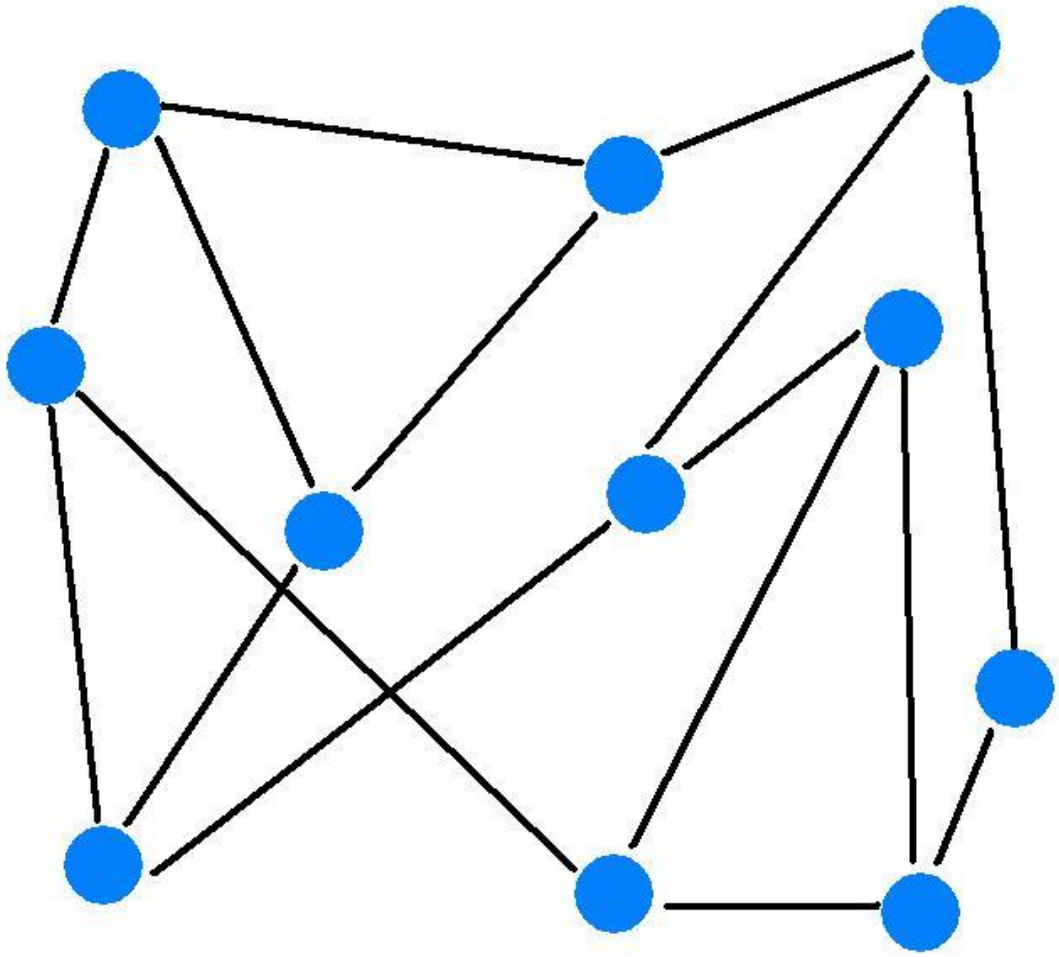
José Pereira
University of Minho
jop@di.uminho.pt

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

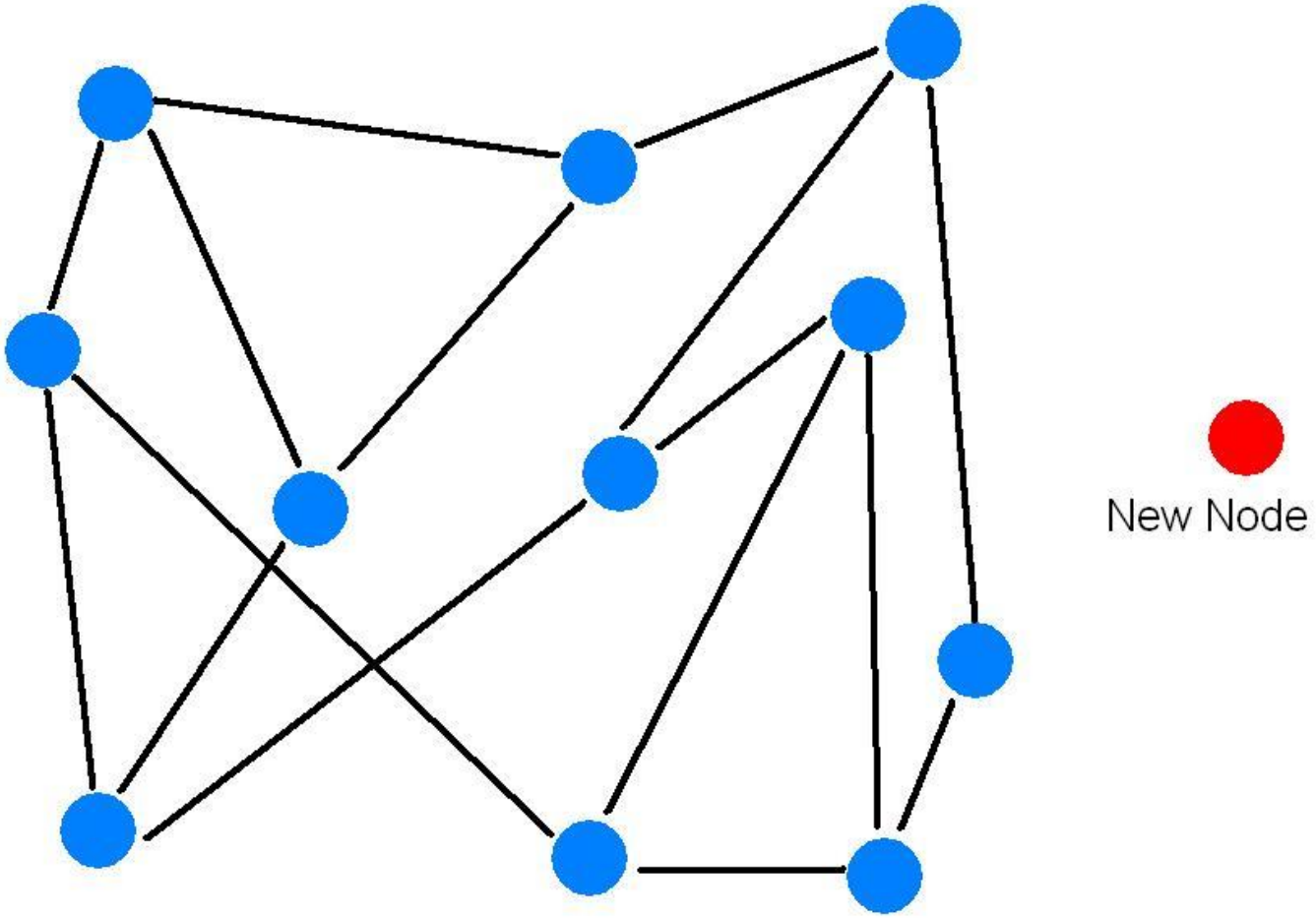
HyParView

- 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: $k \times \#$(active View)
    - Used for fault tolerance.

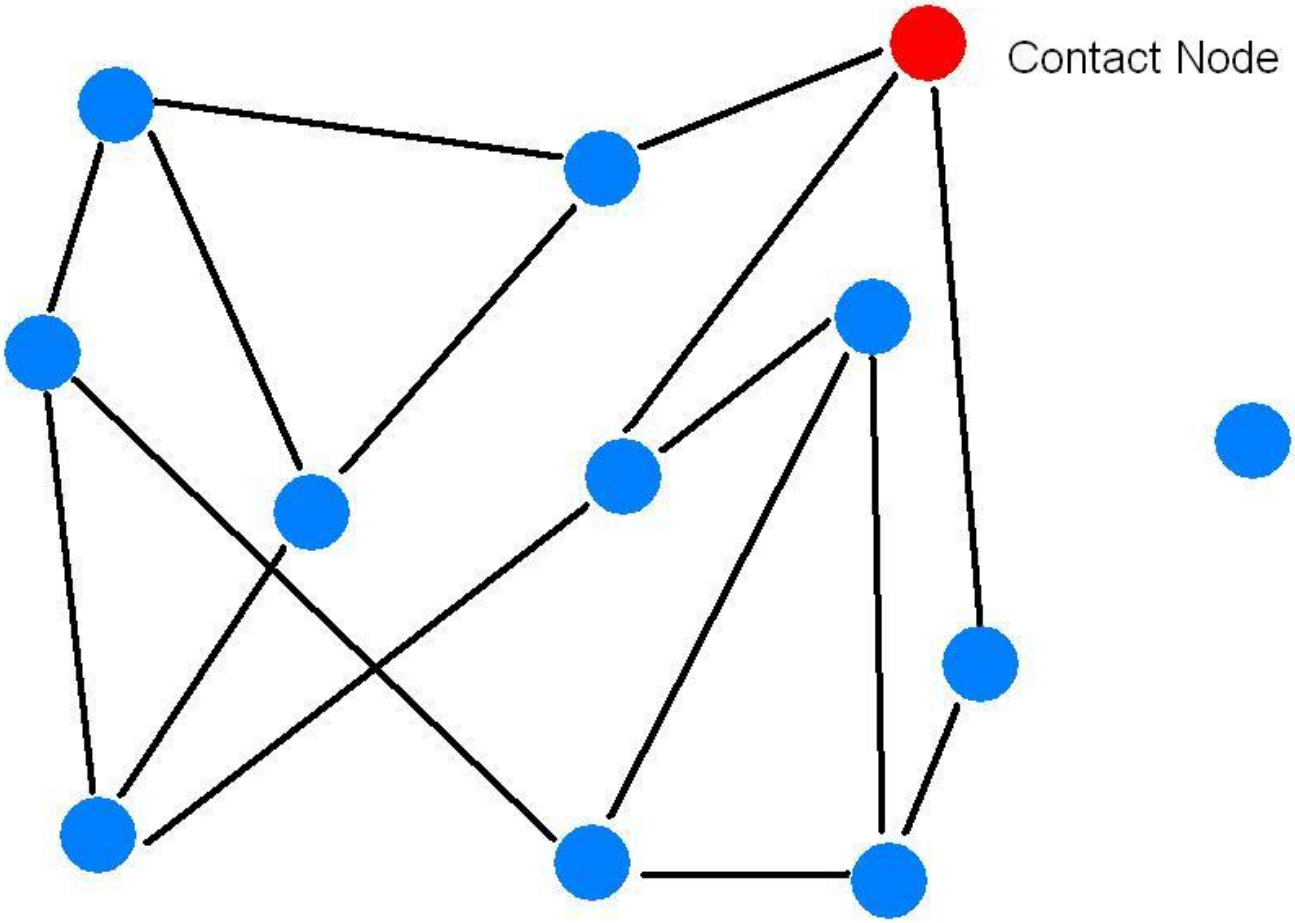
HyParView Join Mechanism



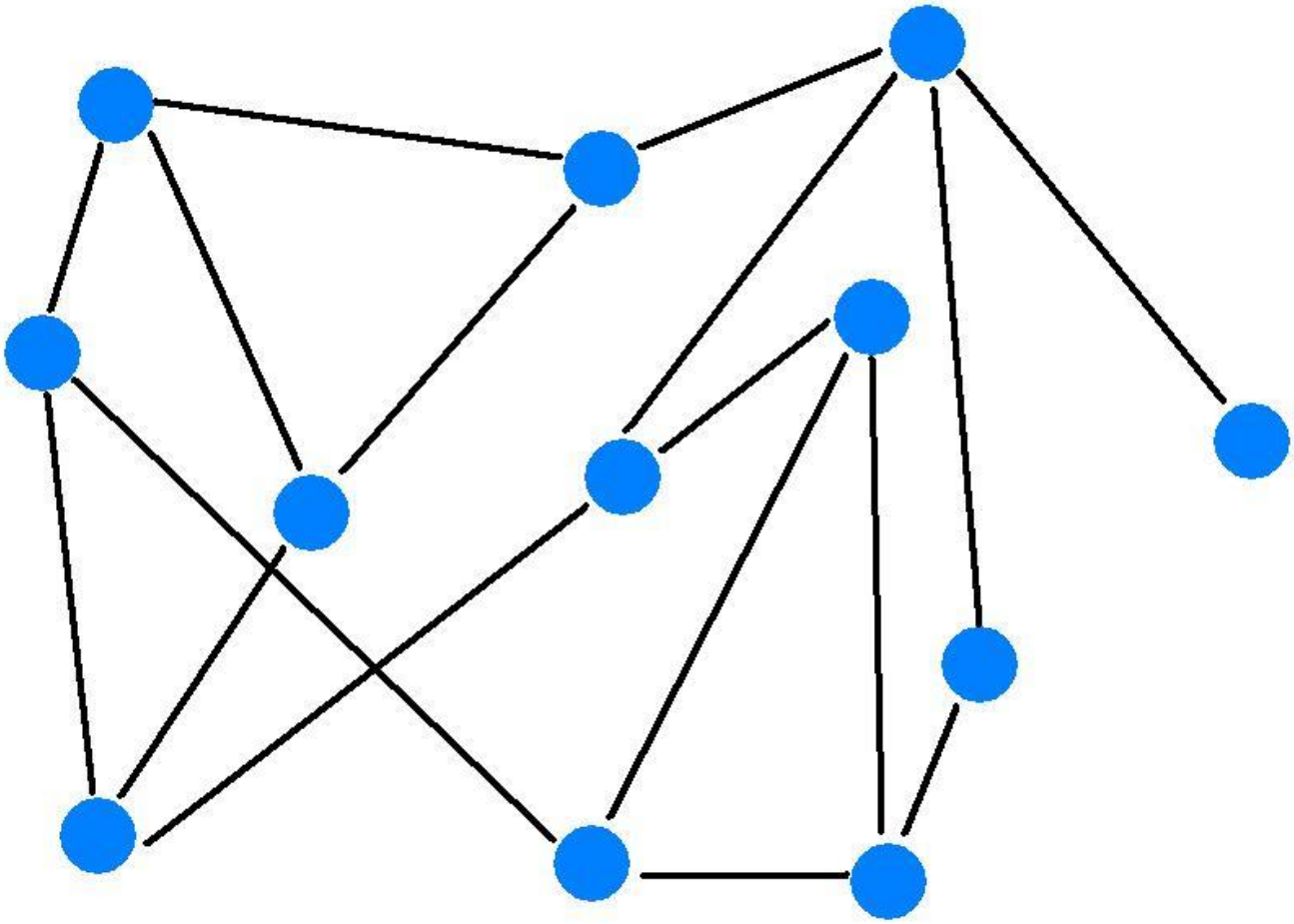
HyParView Join Mechanism



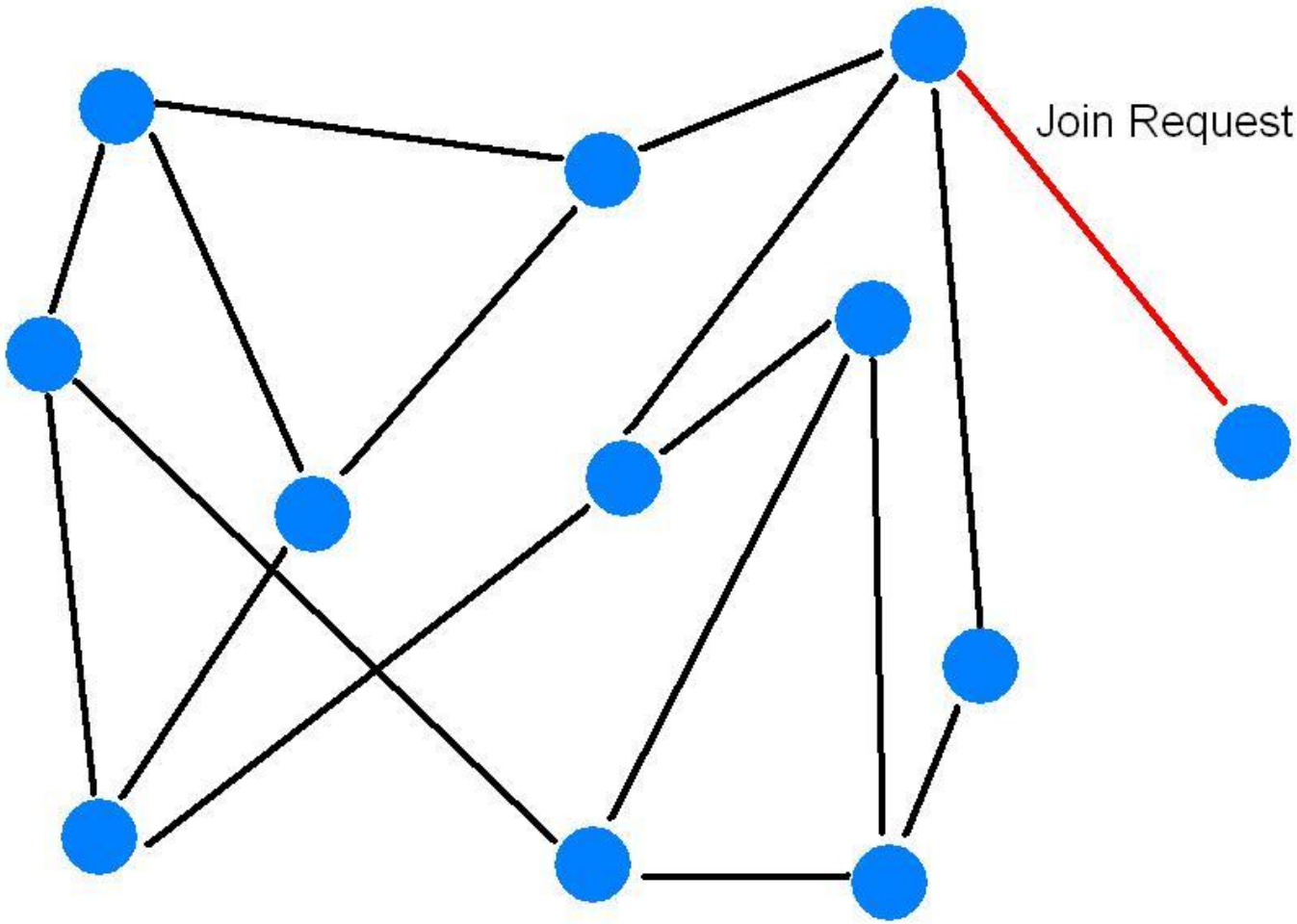
HyParView Join Mechanism



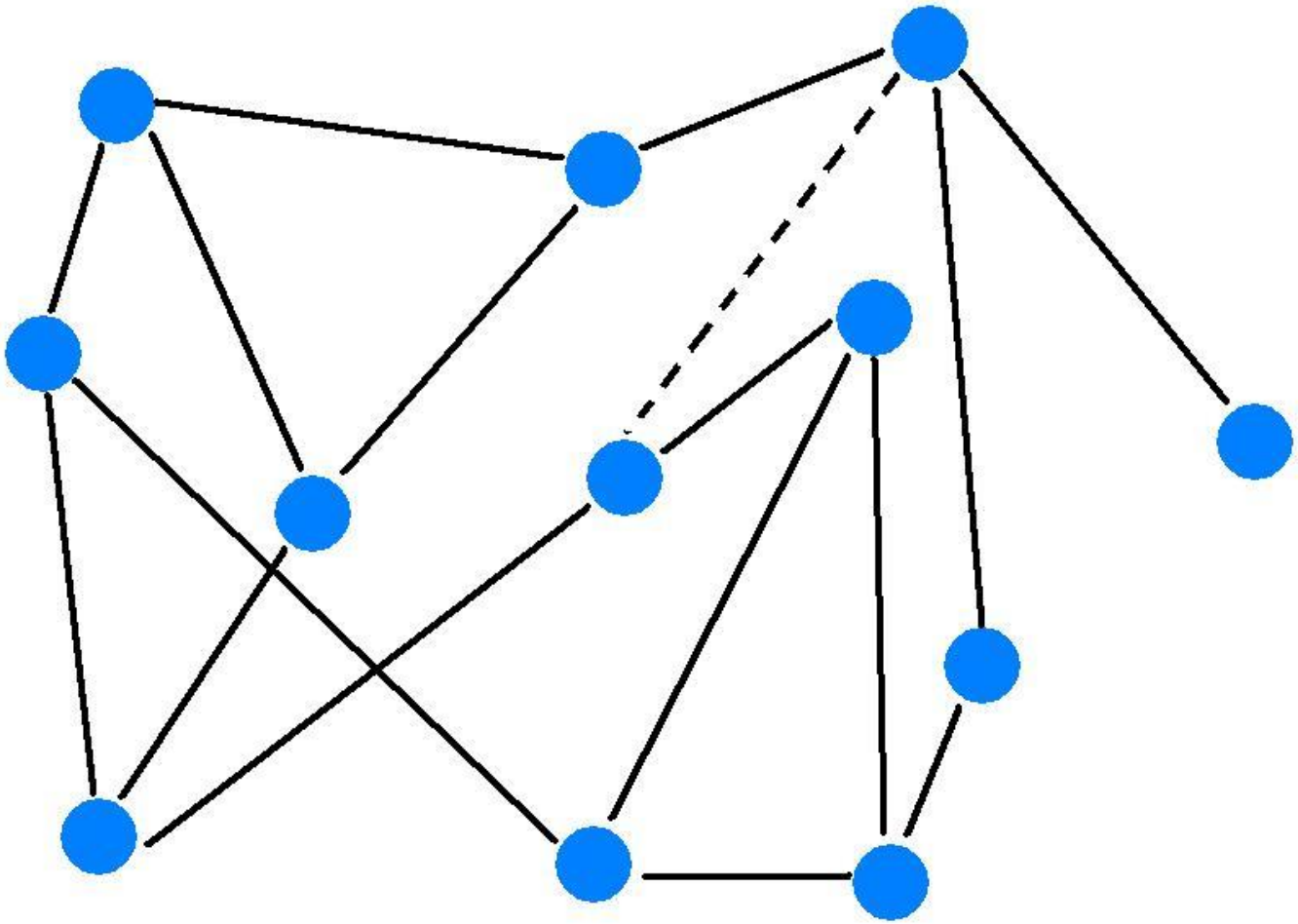
HyParView Join Mechanism



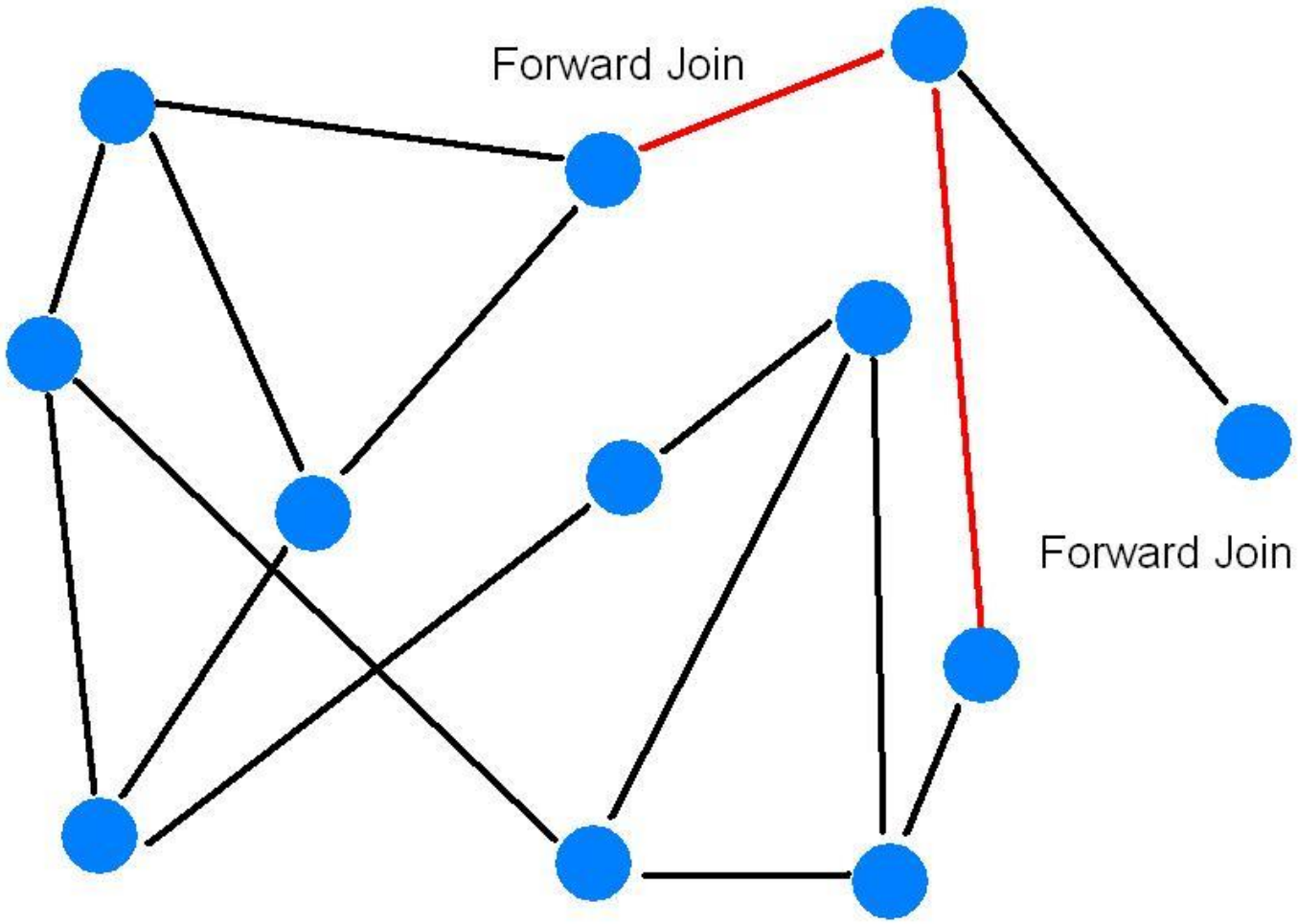
HyParView Join Mechanism



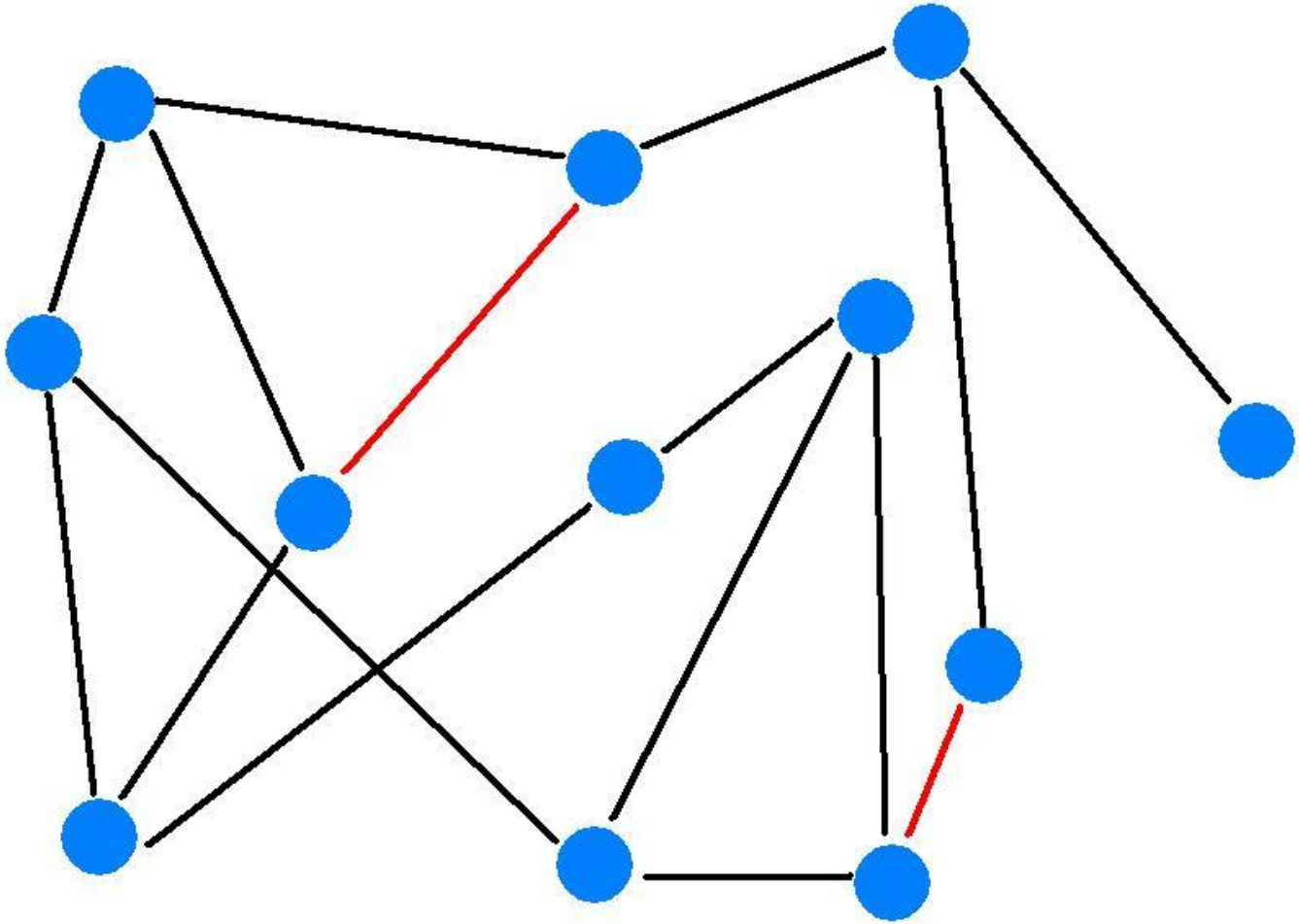
HyParView Join Mechanism



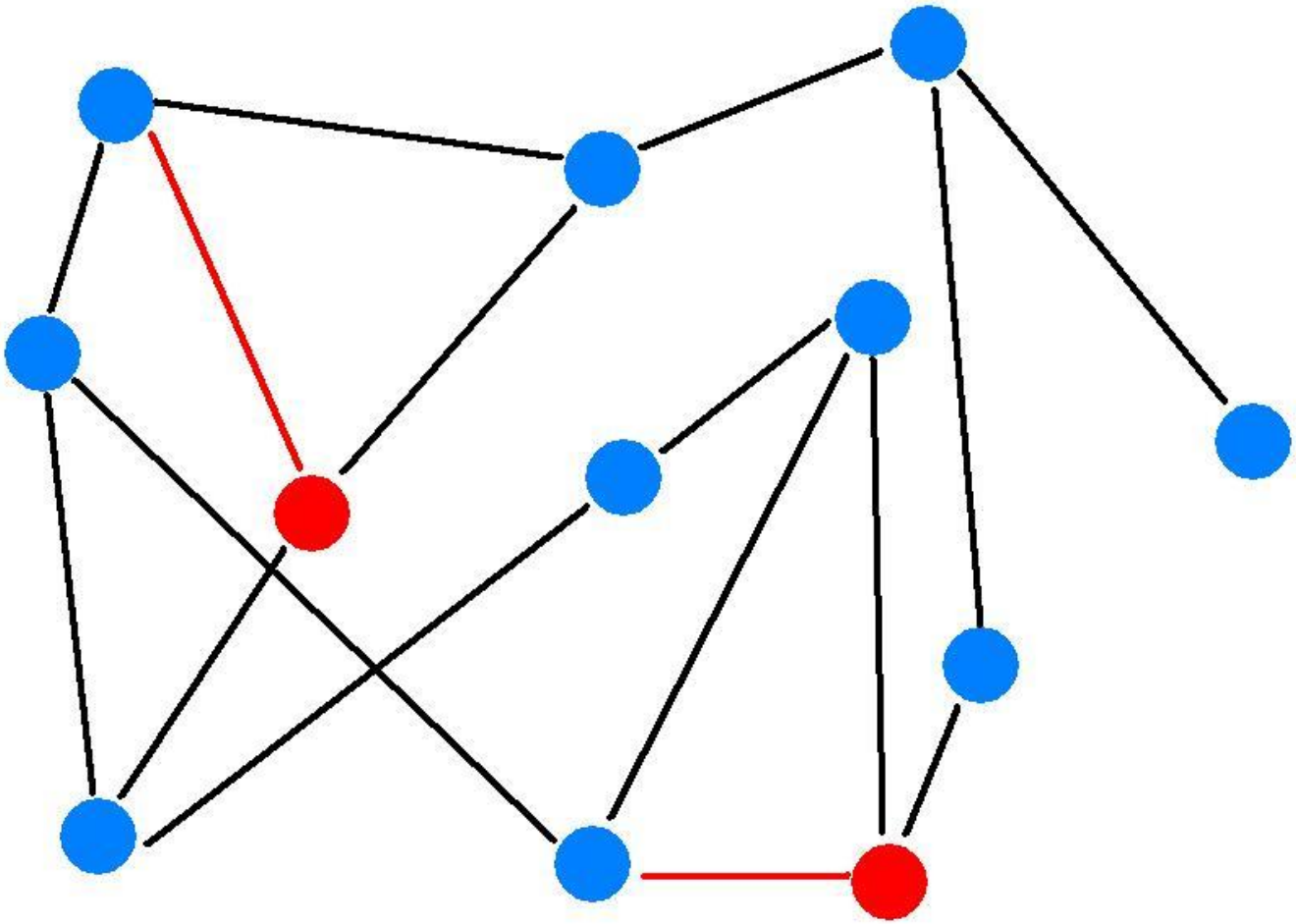
HyParView Join Mechanism



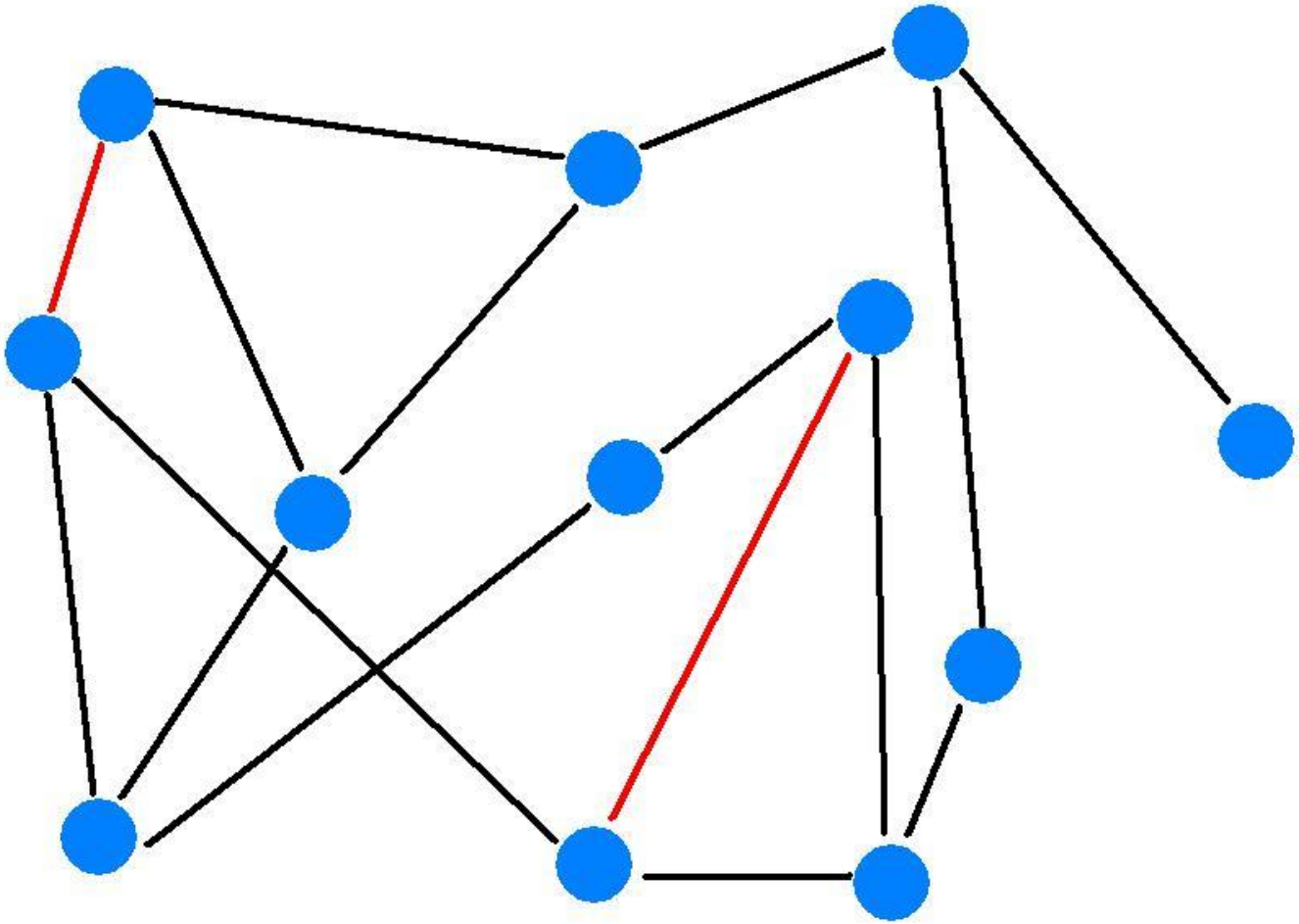
HyParView Join Mechanism



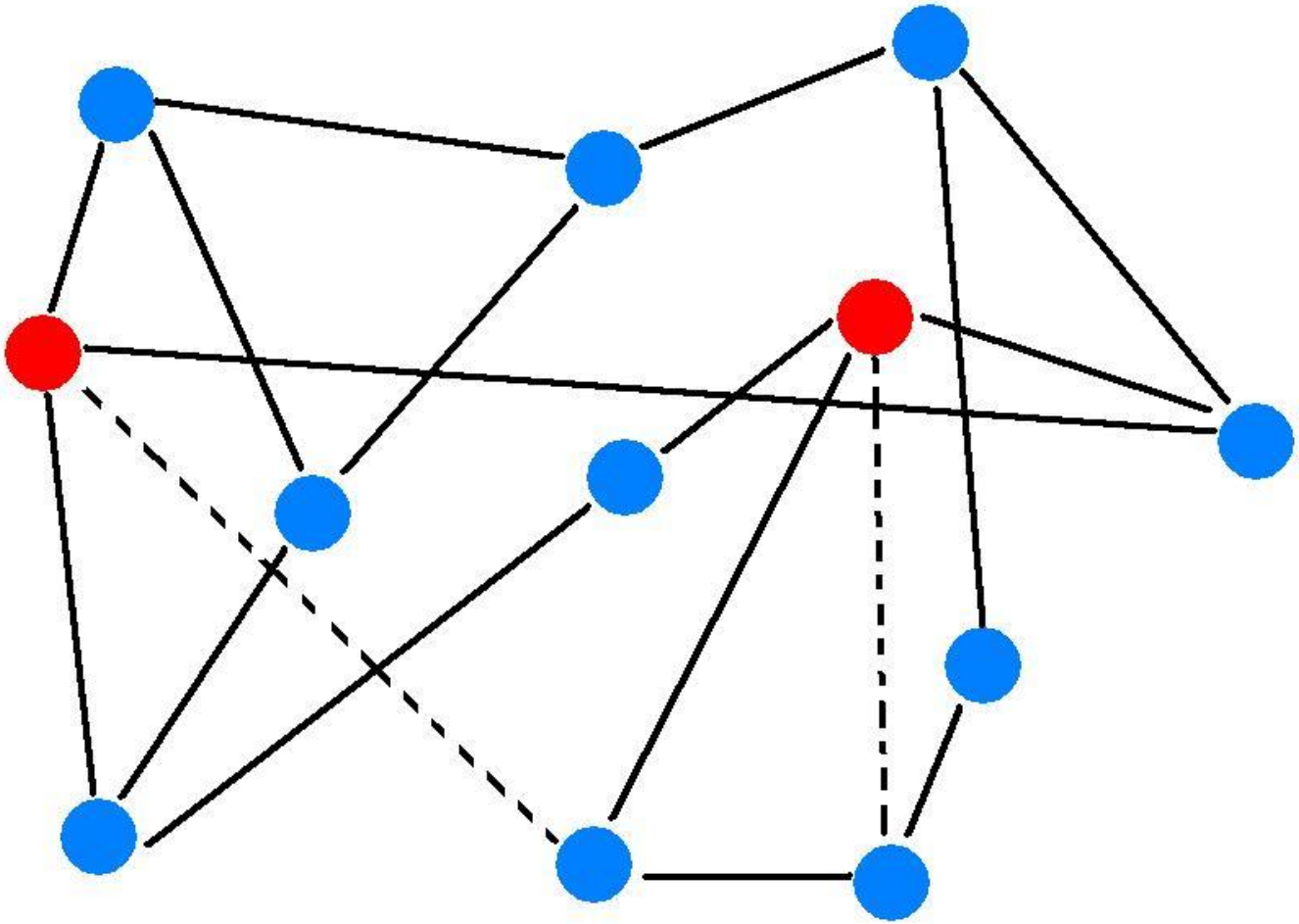
HyParView Join Mechanism



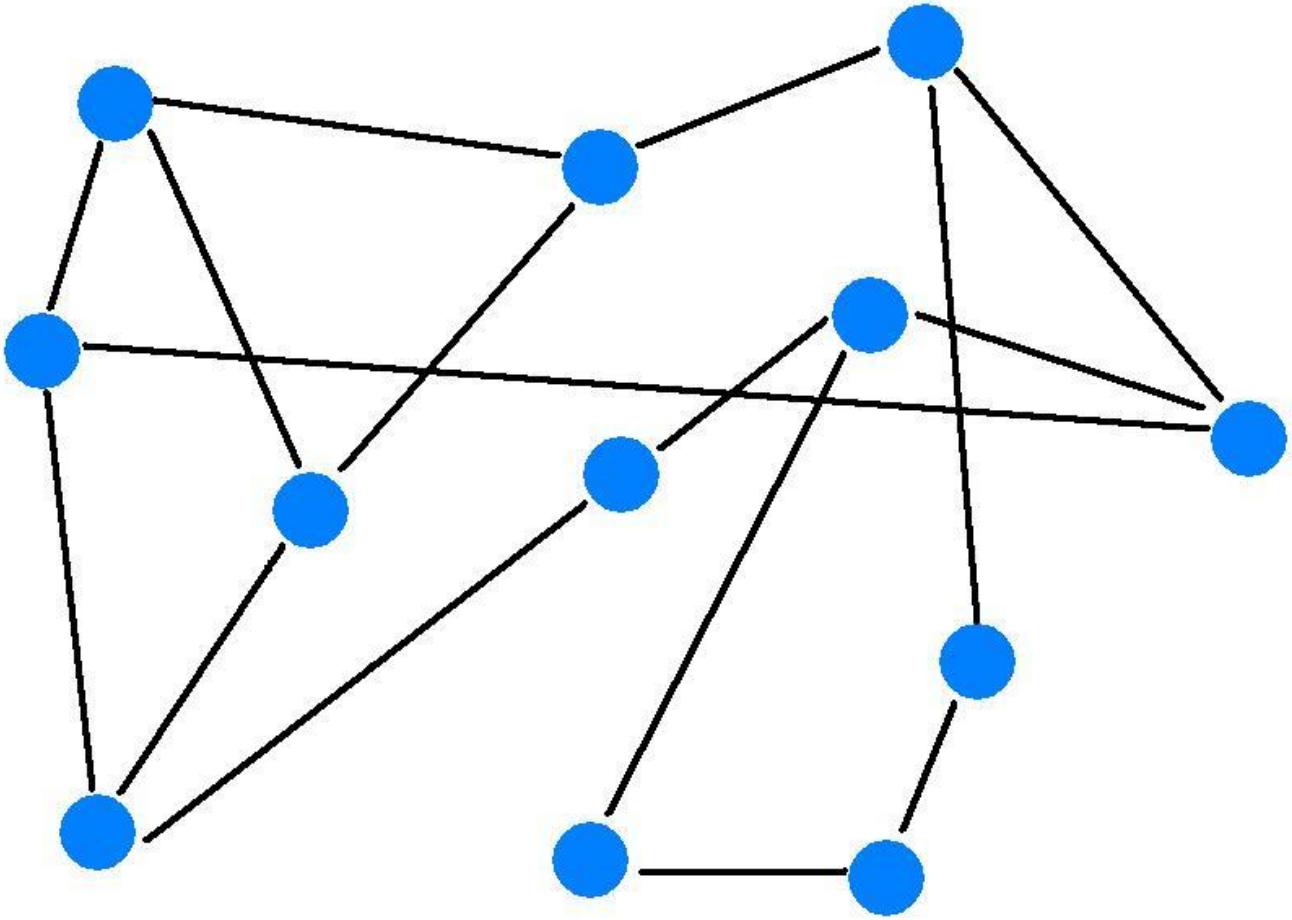
HyParView Join Mechanism



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.

HyParView: 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

```
1   upon init do
2       Send(JOIN, contactNode, myself);

3   upon Receive(JOIN, newNode) do
4       call addNodeActiveView(newNode)
5       foreach n ∈ activeView and n ≠ newNode do
6           Send(FORWARDJOIN, n, newNode, ARWL, myself)

7   upon Receive(FORWARDJOIN, newNode, timeToLive, sender) do
8       if timeToLive== 0||#activeView== 1 then
9           call addNodeActiveView(newNode)
10      else
11          if timeToLive==PRWL then
12              call addNodePassiveView(newNode)
13          n ⟵ n ∈ activeView and n ≠ sender
14          Send(FORWARDJOIN, n, newNode, timeToLive-1, myself)
```

addNodeActiveView(contactNode)

HyParView: The Algorithm

Algorithm 2: View manipulation primitives

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

```
1  procedure dropRandomElementFromActiveView do
2      n ⟵ n ∈ activeView
3      Send(DISCONNECT, n, myself)
4      activeView ⟵ activeView \{n}
5      passiveView ⟵ passiveView ∪{n}

6  procedure addNodeActiveView(node) do
7      if node ≠ myself and node ∉ activeView then
8          if isfull(activeView) then
9              call dropRandomElementFromActiveView
10         activeView ⟵ activeView ∪ node

11 procedure addNodePassiveView(node) do
12     if node ≠ myself and node ∉ activeView and node ∉ passiveView then
13         if isfull(passiveView) then
14             n ⟵ n ∈ passiveView
15             passiveView ⟵ passiveView \{n}
16         passiveView ⟵ passiveView ∪ node

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

HyParView: The Algorithm

Algorithm 2: View manipulation primitives

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

```
1  procedure dropRandomElementFromActiveView do
2      n ⟵ n ∈ activeView
3      Send(DISCONNECT, n, myself)
4      activeView ⟵ activeView \{n}
5      passiveView ⟵ passiveView ∪{n}

6  procedure addNodeActiveView(node) do
7      if node ≠ myself and node ∉ activeView then
8          if isfull(activeView) then
9              call dropRandomElementFromActiveView
10         activeView ⟵ activeView ∪ node

11 procedure addNodePassiveView(node) do
12     if node ≠ myself and node ∉ activeView and node ∉ passiveView then
13         if isfull(passiveView) then
14             n ⟵ n ∈ passiveView
15             passiveView ⟵ passiveView \{n}
16         passiveView ⟵ passiveView ∪ node

17 upon Receive(DISCONNECT, peer) do
18     if peer ∈ activeView then
19         activeView ⟵ activeView \ {peer}
20         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

```
1   upon event Broadcast(m) do
2       mID ⟵ hash(m + myself)
3       peerList ⟵ getPeer(f,null)
4       foreach p ∈ peerList do
5           trigger Send(Gossip, p, m, mID, myself)
6       trigger Deliver(m)
7       receivedMsgs ⟵ receivedMsgs ∪ {mID}

8   upon event Receive(Gossip, m, mID, sender) do
9       if mID ∉ receivedMsgs then
10          receivedMsgs ⟵ receivedMsgs ∪ {mID}
11          trigger Deliver(m)
12          peerList ⟵ getPeer(f,sender)
13          foreach p ∈ peerList do
14              trigger Send(Gossip, p, m, mID, myself)
```

HyParView: Is this really Good?

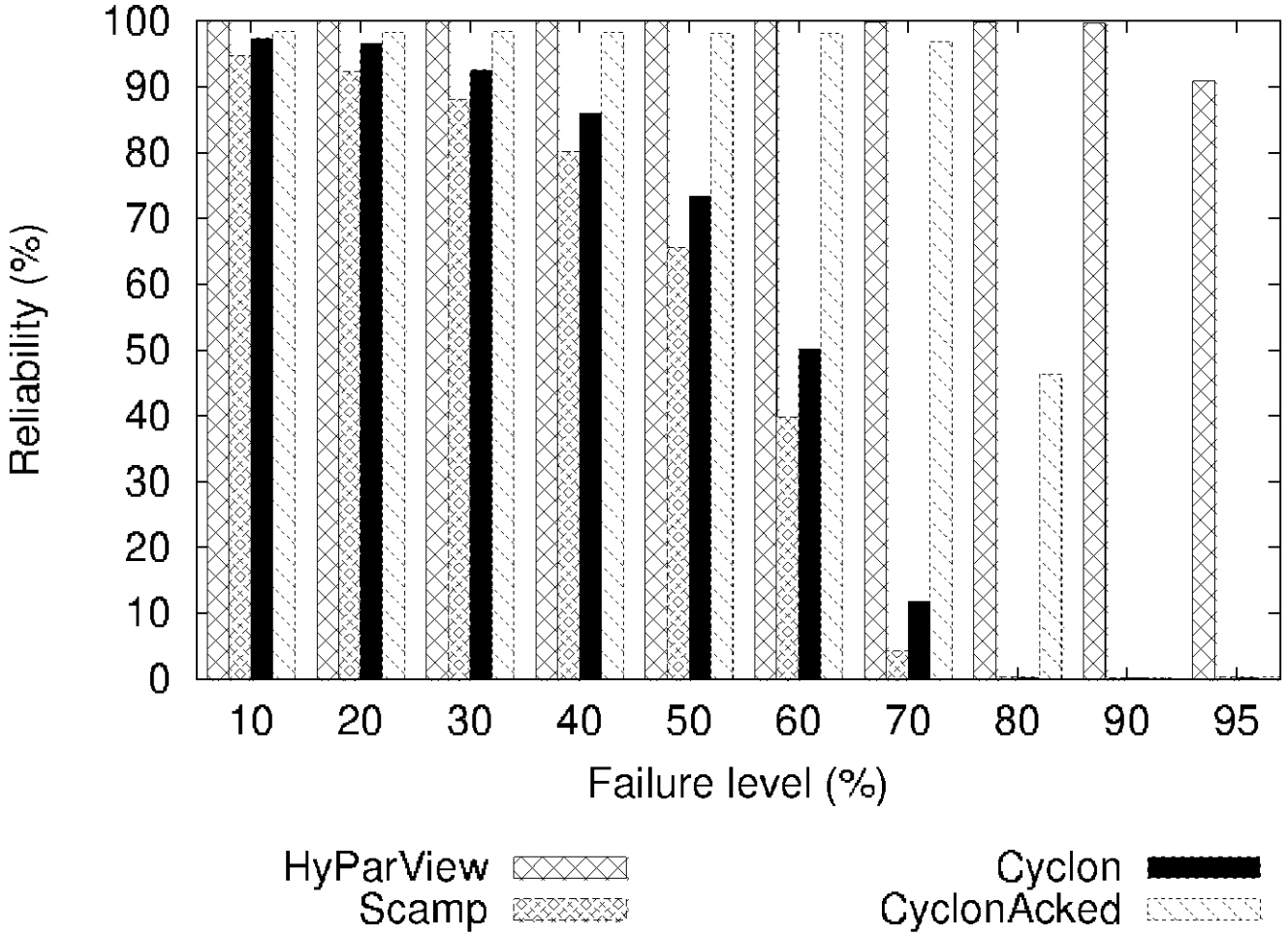


Figure 4.2: Average reliability for 1000 messages

What if, you known exactly the resource you are looking for?

- Do we need all of this?
- Can we do something better?

What if, you known exactly the resource you are looking for?

- Do we need all of this?
- Can we do something better?

(although we might need to slightly readjust the problem we are solving)

Resource Location (Exact Match)

- A Definition:

Given a set of processes containing different sets of resources, locate the processes that contain a given resource given its unique identifier.

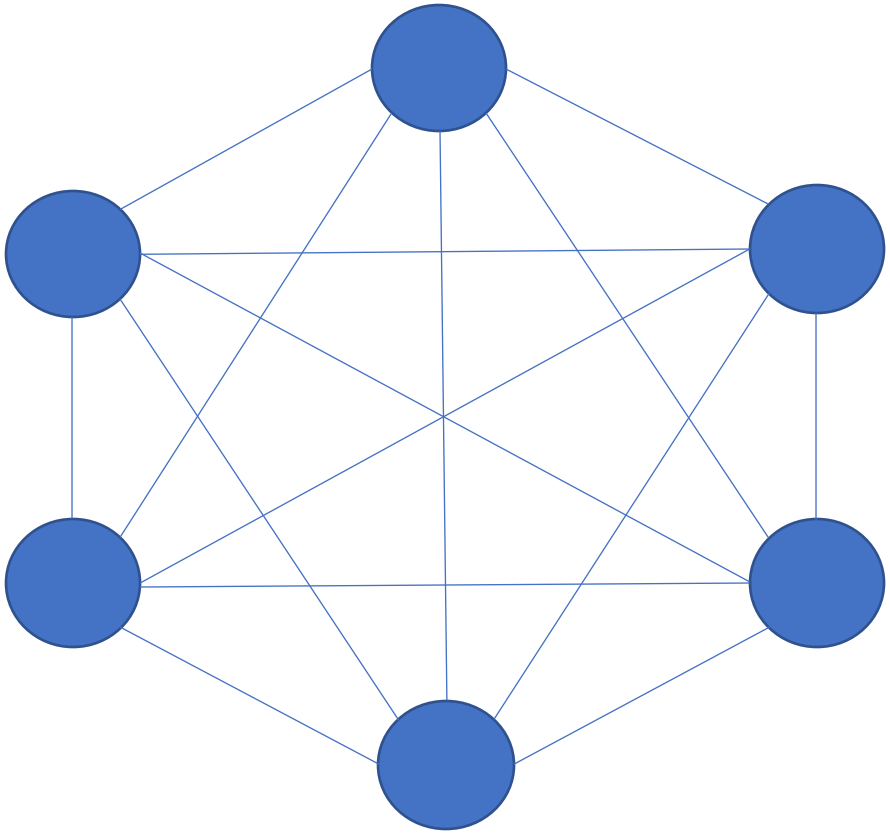
- One possible concretization:
  - File Sharing Applications.
  - Processes own a set of files that have different names which are self-explanatory/predictable.
  - Locate the processes that have a file named: “Wheel of Time - S01E01”.

Consistent Hashing

- We can build a distributed index of resources among all processes by doing the following:
- We pick a hash function that generates hash values in the interval [0, G].
- We attribute to each process an identifier within the interval [0, G] (all processes must have a different identifier).
- For each resource in the system, we compute the hash of its identifier, and store information about it in the process with the closest identifier.

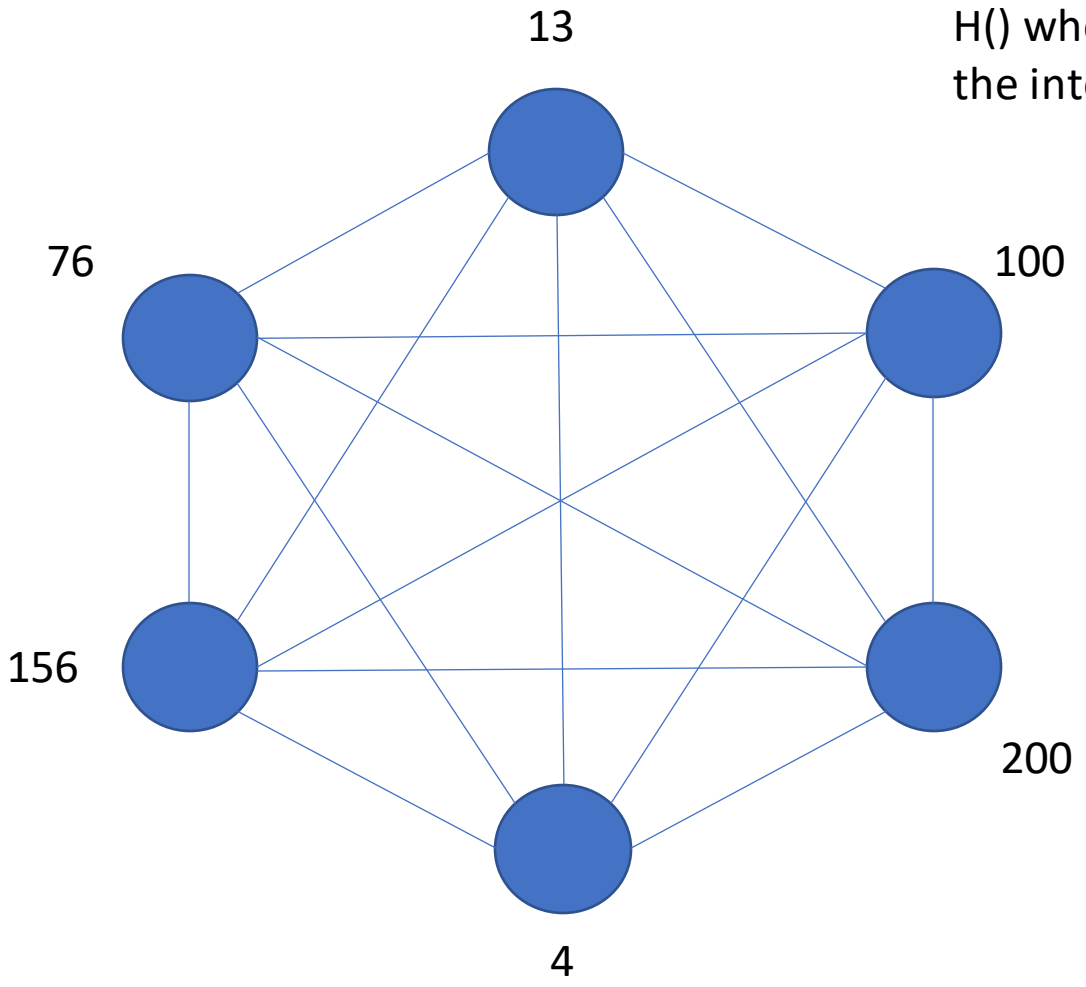
Consistent Hashing

Let’s assume full membership information.



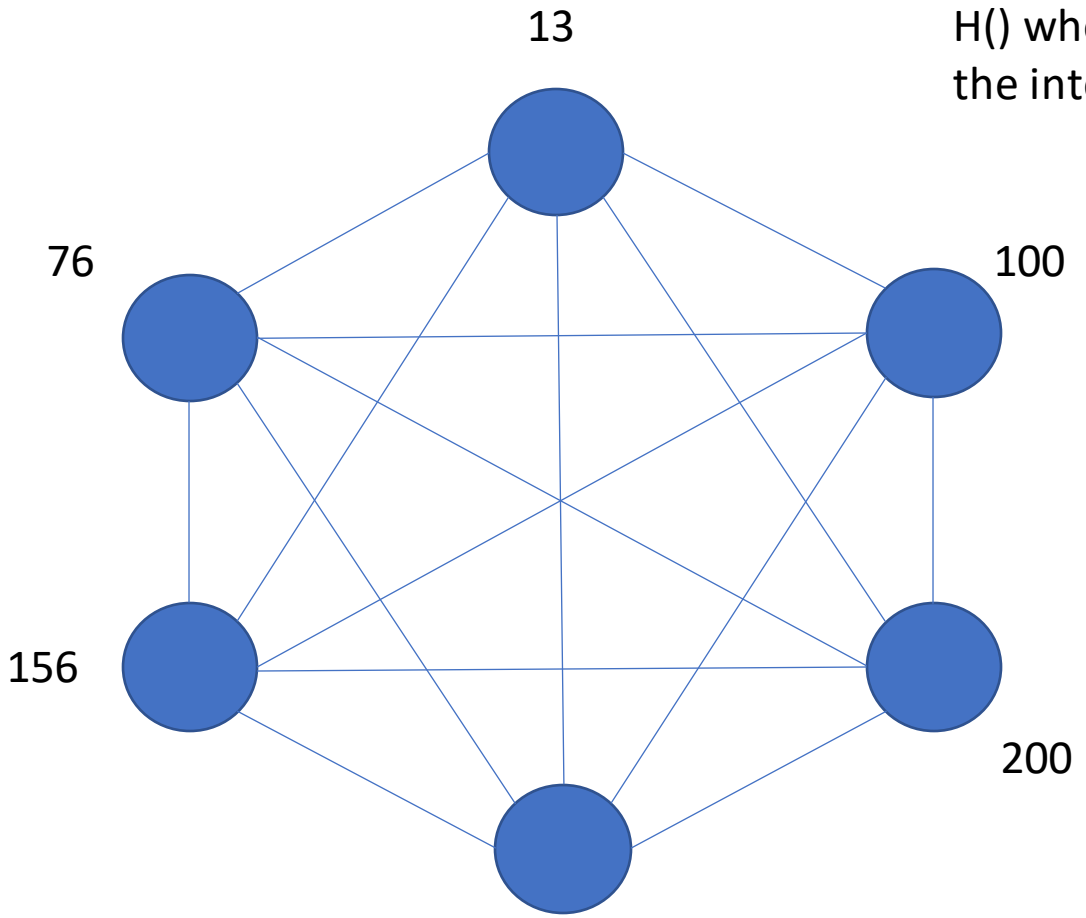
Consistent Hashing

Assume a hash function H() whose result is within the interval: [0, 256].



Consistent Hashing

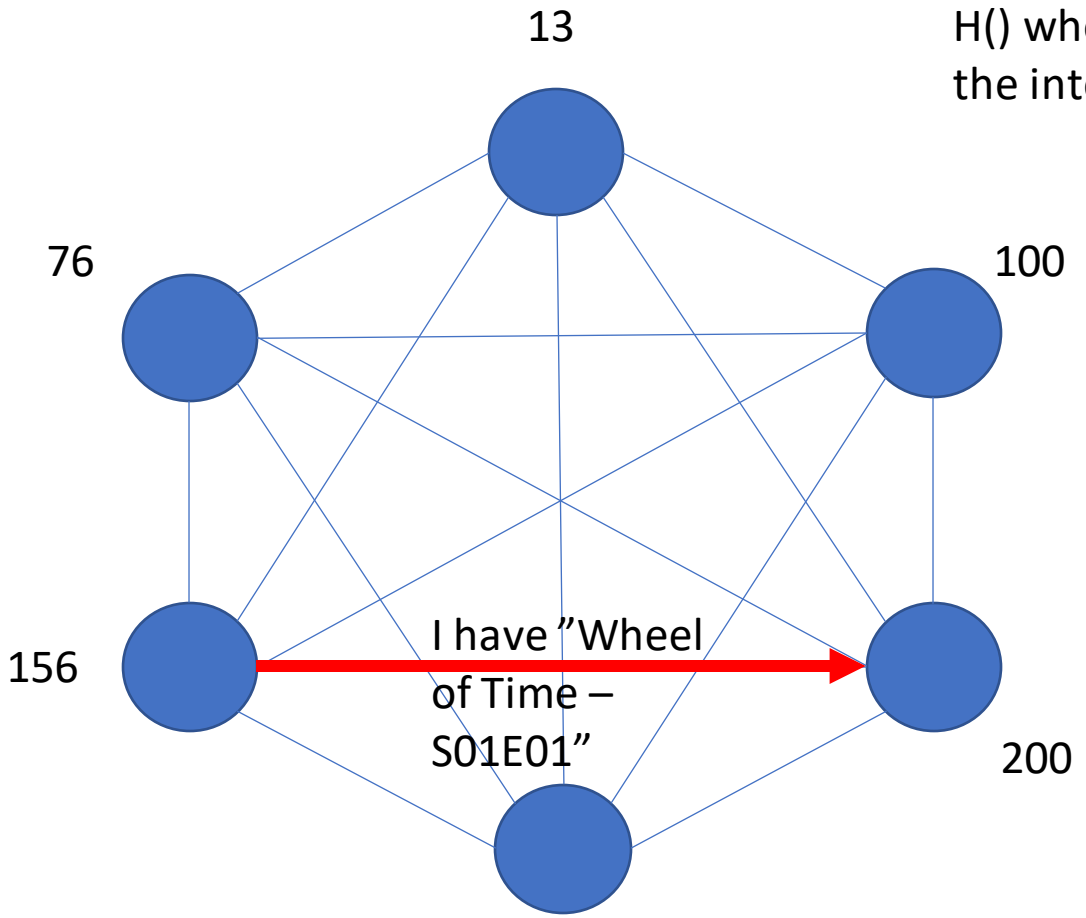
Assume a hash function H() whose result is within the interval: [0, 256].



H(Wheel of Time – S01E01) = 201

Consistent Hashing

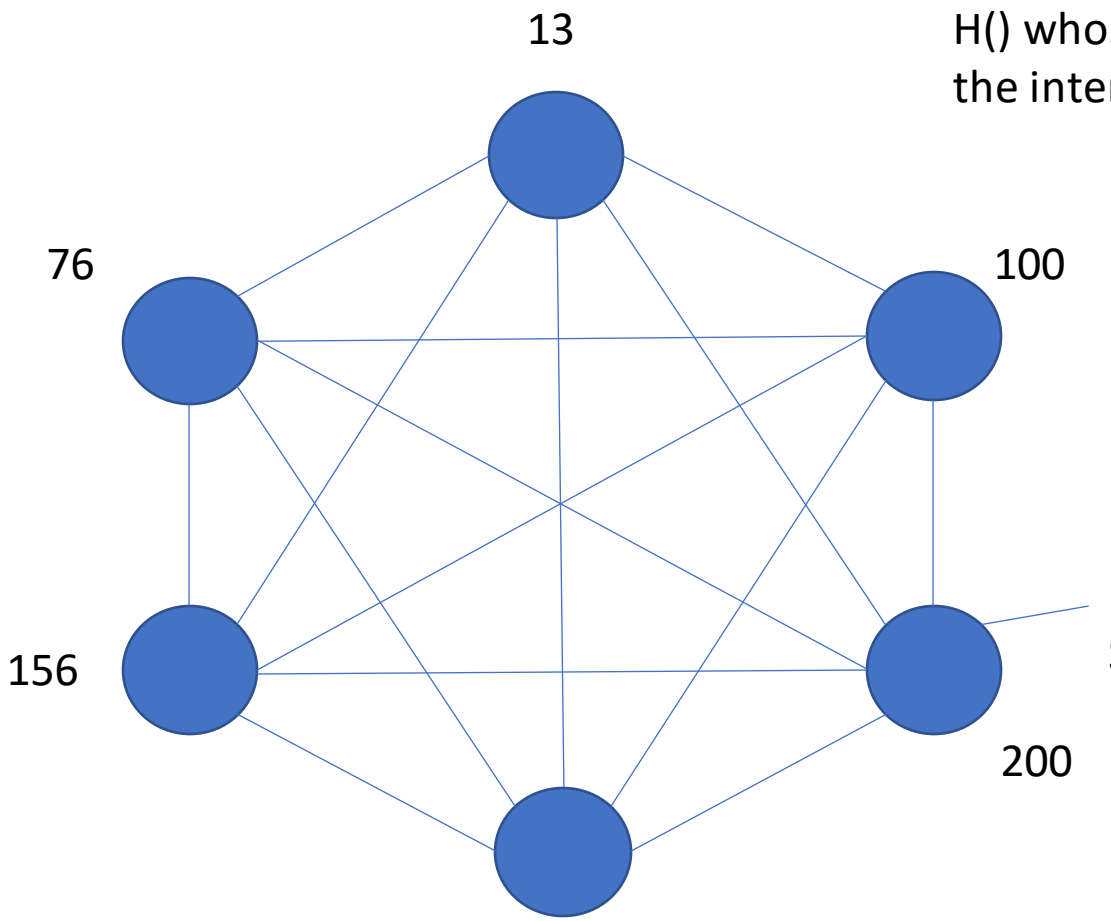
Assume a hash function H() whose result is within the interval: [0, 256].



H(Wheel of Time – S01E01) = 201

Consistent Hashing

Assume a hash function H() whose result is within the interval: [0, 256].



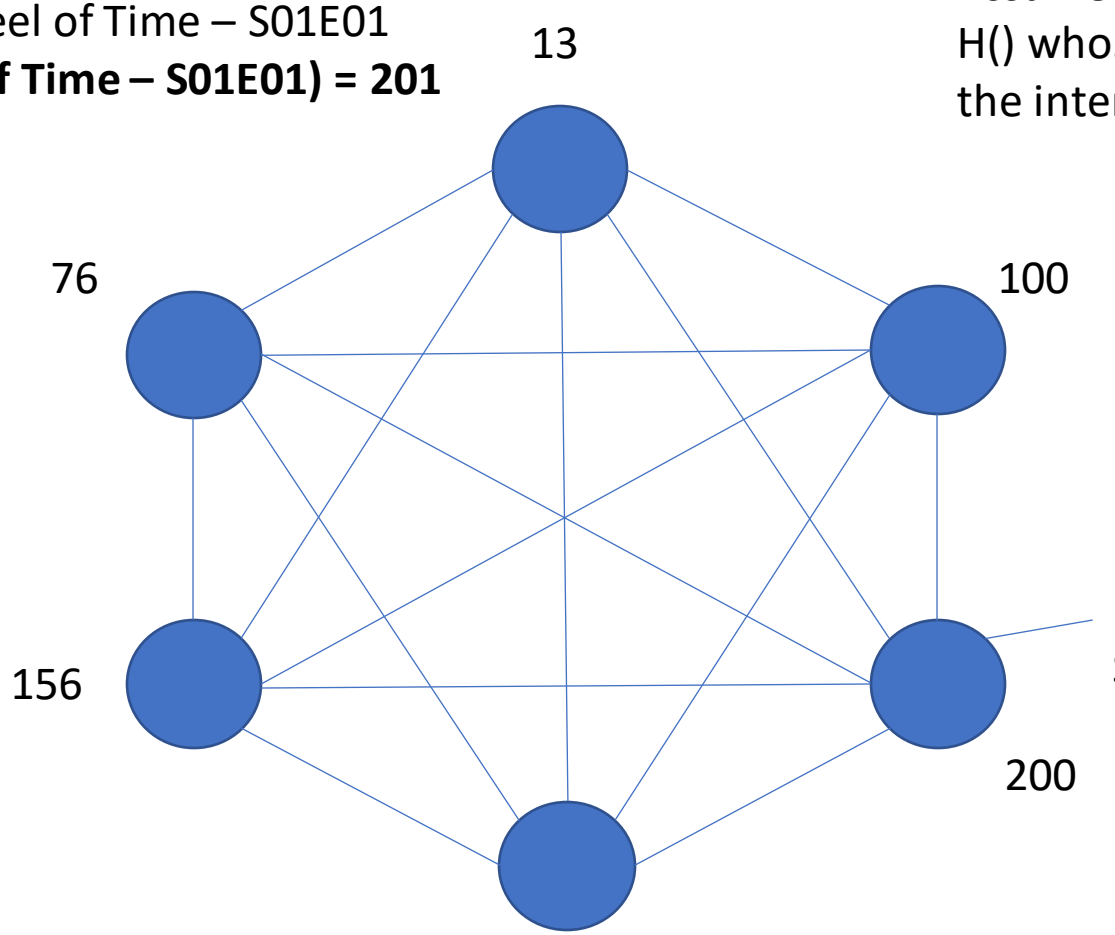
"Wheel of Time – S01E01" @ 156

H(Wheel of Time – S01E01) = 201

Consistent Hashing

I want Wheel of Time – S01E01
H(Wheel of Time – S01E01) = 201

Assume a hash function H() whose result is within the interval: [0, 256].



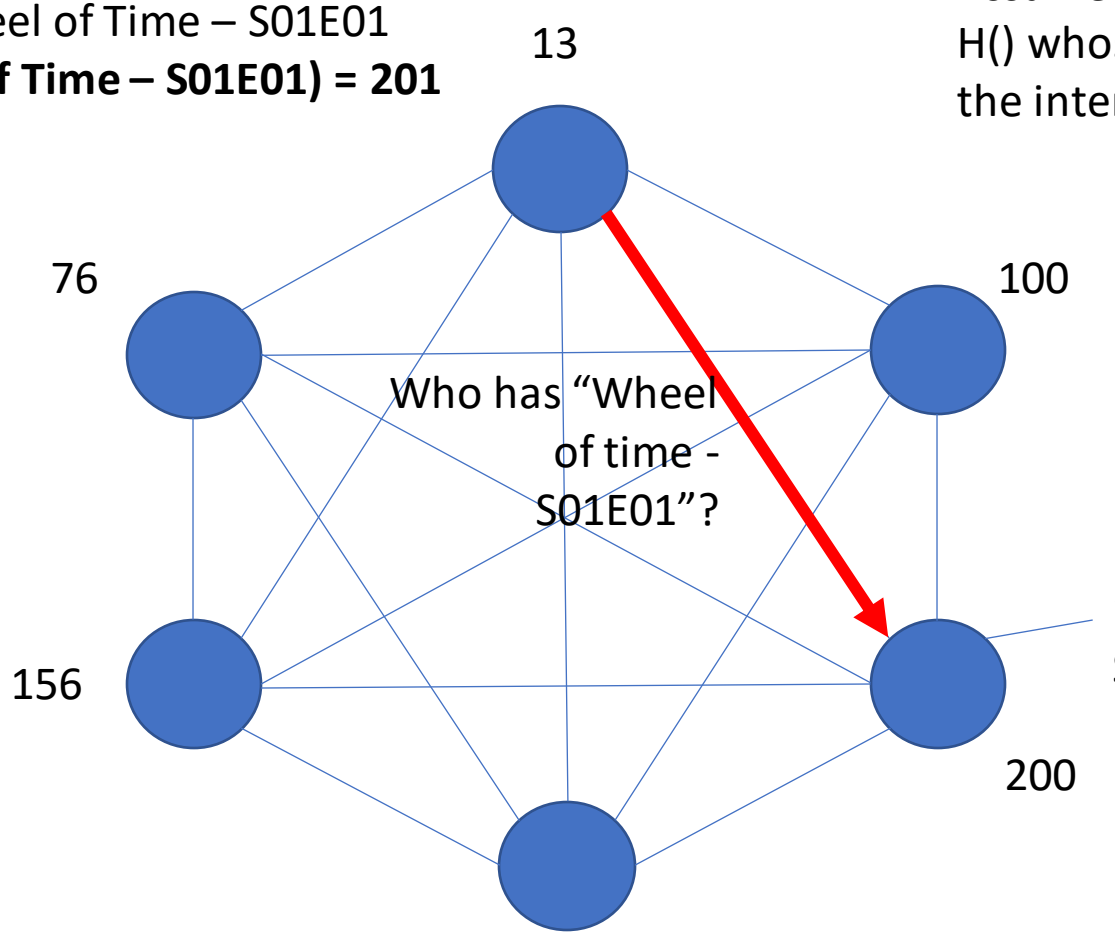
"Wheel of Time – S01E01" @ 156

H(Wheel of Time – S01E01) = 201

Consistent Hashing

I want Wheel of Time – S01E01
H(Wheel of Time – S01E01) = 201

Assume a hash function H() whose result is within the interval: [0, 256].

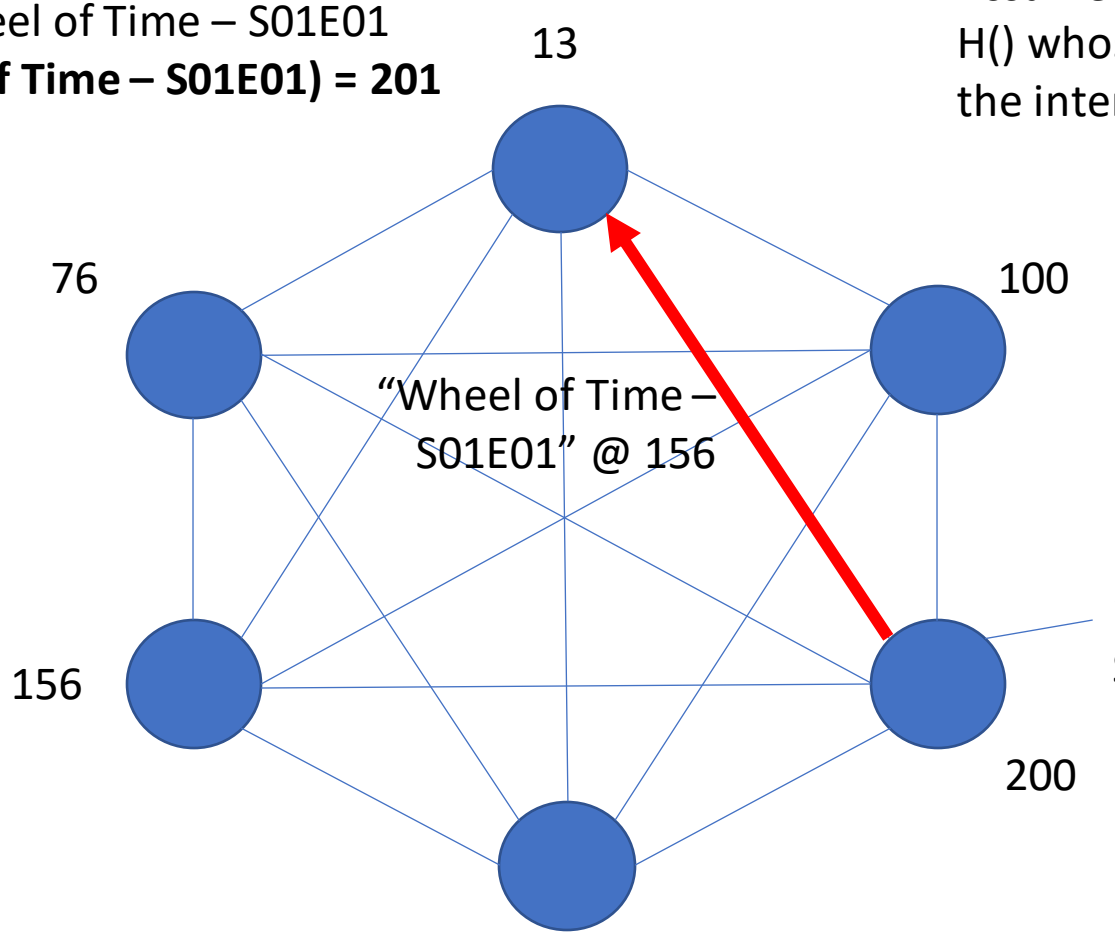


H(Wheel of Time – S01E01) = 201

Consistent Hashing

I want Wheel of Time – S01E01
H(Wheel of Time – S01E01) = 201

Assume a hash function H() whose result is within the interval: [0, 256].

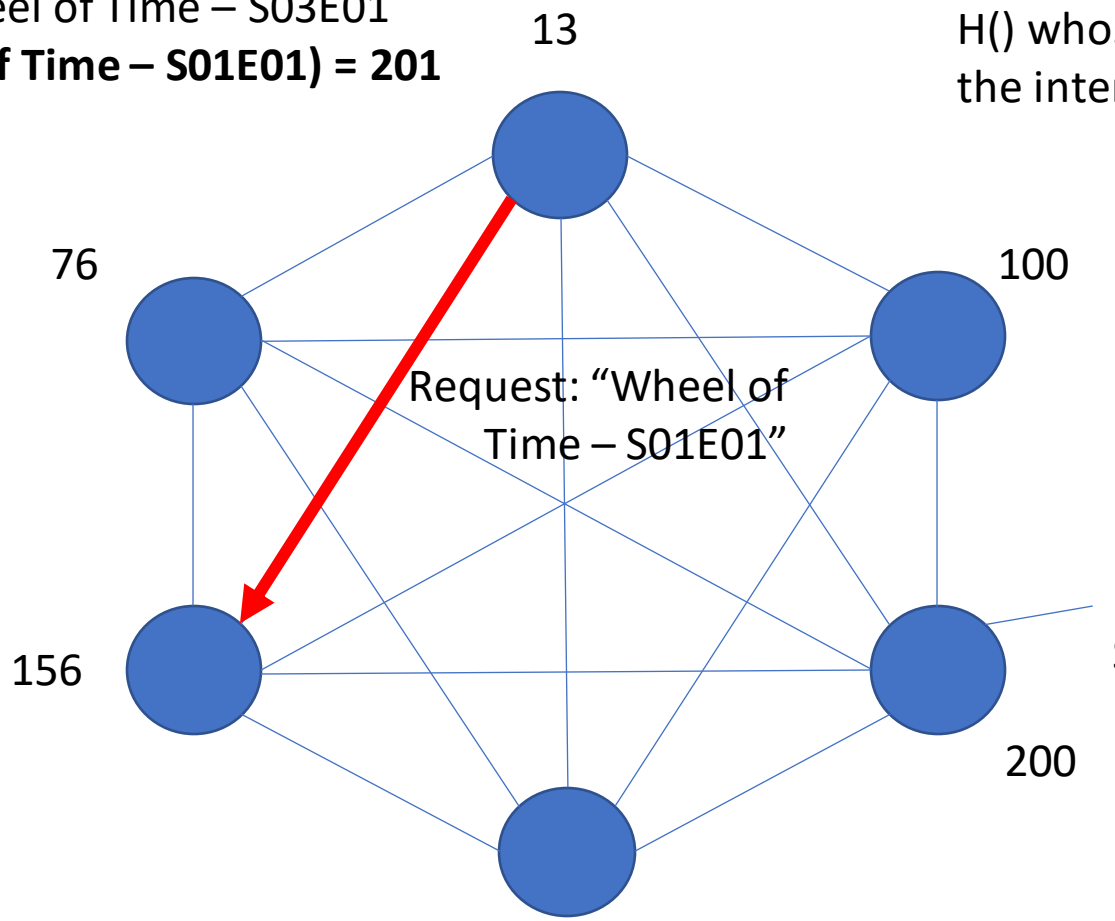


H(Wheel of Time – S01E01) = 201

Consistent Hashing

I want Wheel of Time – S03E01
H(Wheel of Time – S01E01) = 201

Assume a hash function H() whose result is within the interval: [0, 256].



H(Wheel of Time – S01E01) = 201

Consistent Hashing

- Consistent Hashing leverages the fact that independent processes will obtain the same value when applying the same hash function to the same arbitrary input.
- When we have the full membership (i.e., every process in the system knowns all other) this allows to build a **One-Hop Distributed Hash Table (DHT).**
- One-Hop DHTs are one of the foundations of many modern NoSQL Datastores such as Cassandra, Dynamo, MongoDB, ...

What if we have a huge number of processes??

- Should we still use a one-hop DHT?
- Is there a better alternative?

What if we have a huge number of processes??

- Should we still use a one-hop DHT?
- Is there a better alternative?

- Problem: Full membership implies that every process must know all other processes, if the system is large, changes in the membership might be frequent, and the cost to keep this information up-to-date becomes too expensive.
- Solution: partial views (e.g., Cyclon or HyParView).

What if we have a huge number of processes??

- Should we still use a one-hop DHT?
- Is there a better alternative alternative?

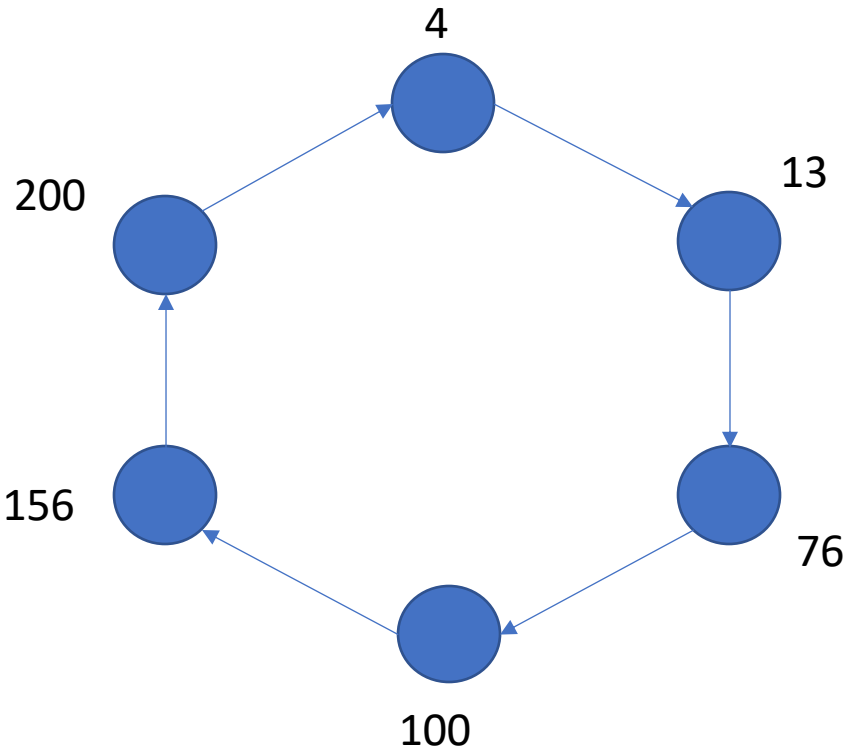
- Problem: Full membership implies that every process has to know all other processes, if the system is large, changes in the membership might be frequent, and the cost to keep this information up-to-date becomes too expensive.
- Solution: partial views (e.g., Cyclon or HyParView).
- **Caviat: You need to (efficiently) find a process given its identifier.**

Structured Overlay Networks

- An overlay network composed of logical links between processes, whose topology has properties known a-priori.
- Many times, these properties are related with the identifiers of nodes (but there are exceptions).

Structured Overlay Networks (first step)

- The most common overlay topology in structured overlay networks are rings, that connect nodes in order considering their identifiers.



This is already very good:

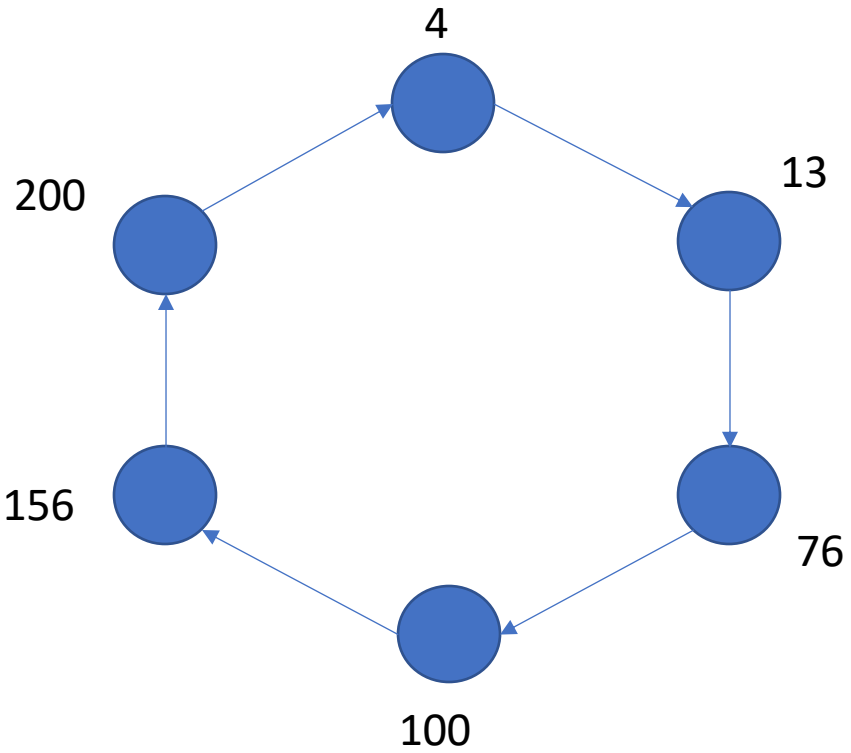
- You can find the process that is responsible by a given number.
- Just by going over the ring, until you find it.

In fact, the ring structure, in this type of structured overlay is the prime correctness criteria.

Structured Overlay Networks (first step)

- T… d
  c… n
  c…

Why is this not good enough?



This is already very good:

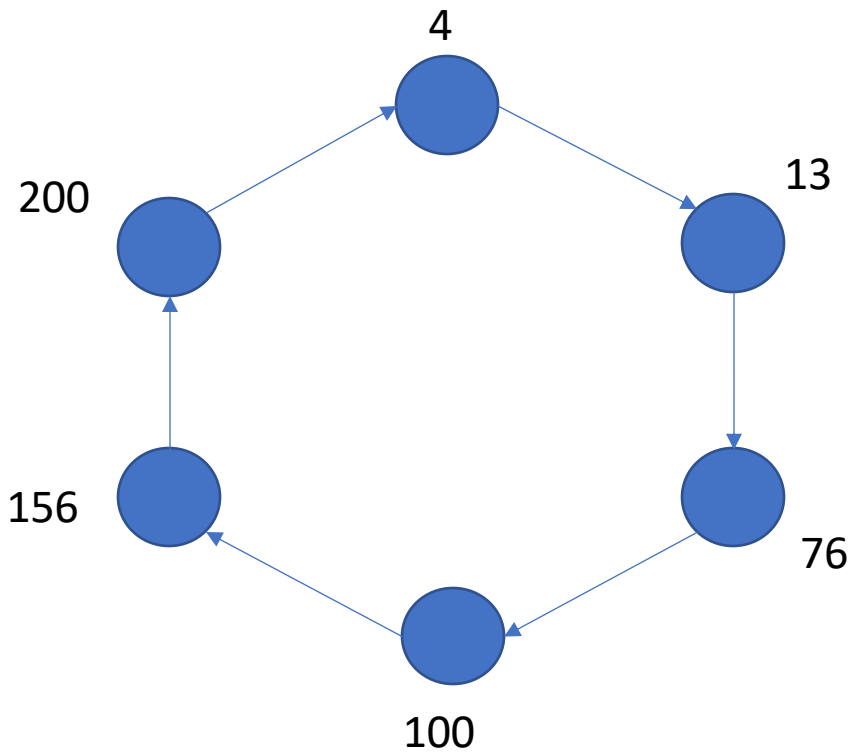
- You can find the process that is responsible by a given number.
- Just by going over the ring, until you find it.

In fact, the ring structure, in this type of structured overlay is the prime correctness criteria.

Structured Overlay Networks (first step)

- T… d
  c… n
  c…

Why is this not good enough?
Long paths between processes.



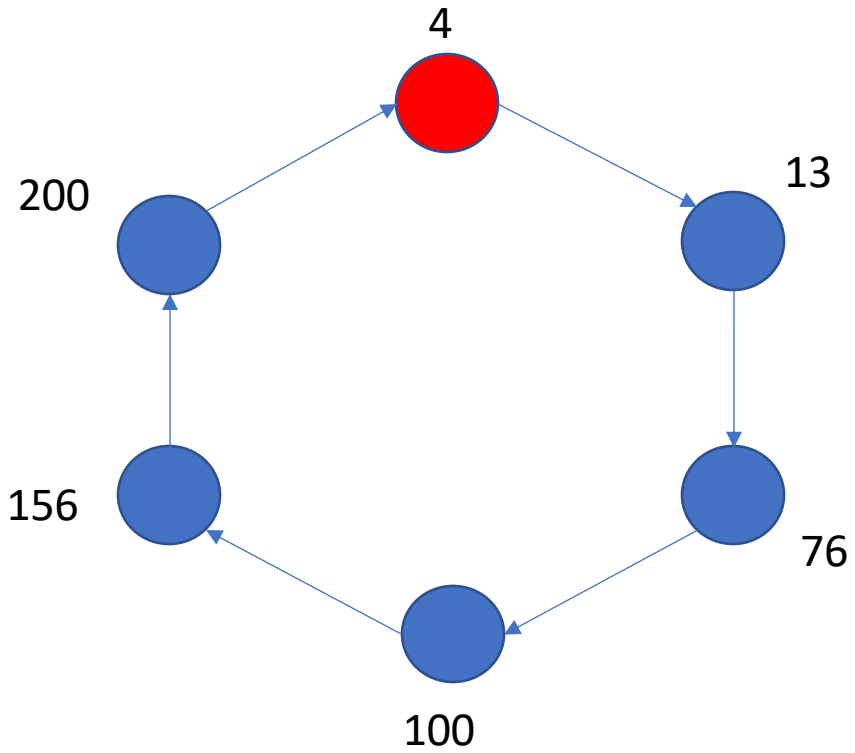
This is already very good:

- You can find the process that is responsible by a given number.
- Just by going over the ring, until you find it.

In fact, the ring structure, in this type of structured overlay is the prime correctness criteria.

Structured Overlay Networks (second step)

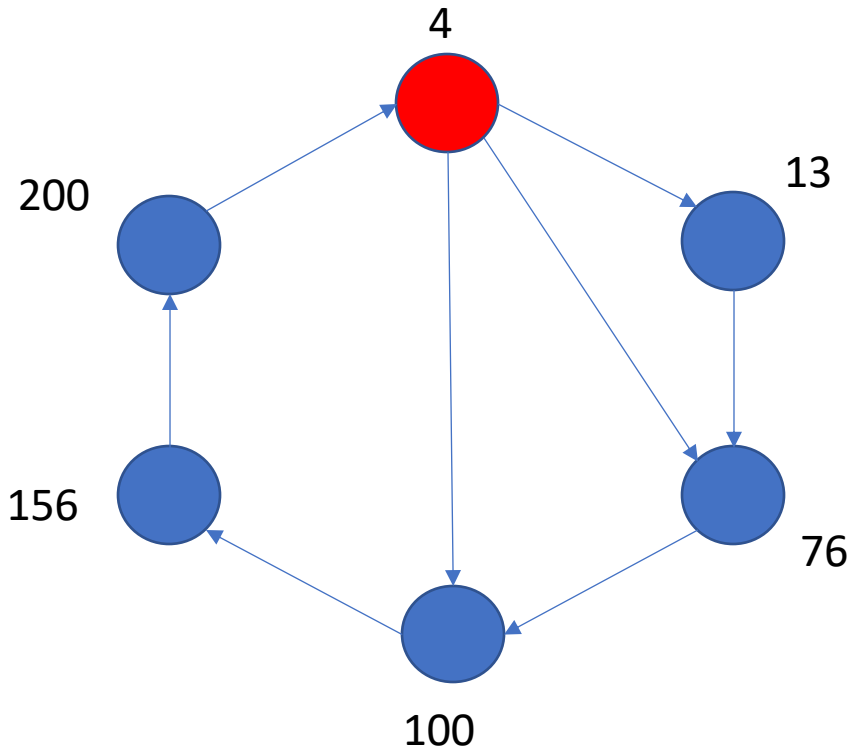
- The most common overlay topology in structured overlay networks are rings, that connect nodes in order considering their identifiers.



We add some additional overlay links (yes in the order of ln(# π)) to speed up things.

Structured Overlay Networks (second step)

- The most common overlay topology in structured overlay networks are rings, that connect nodes in order considering their identifiers.



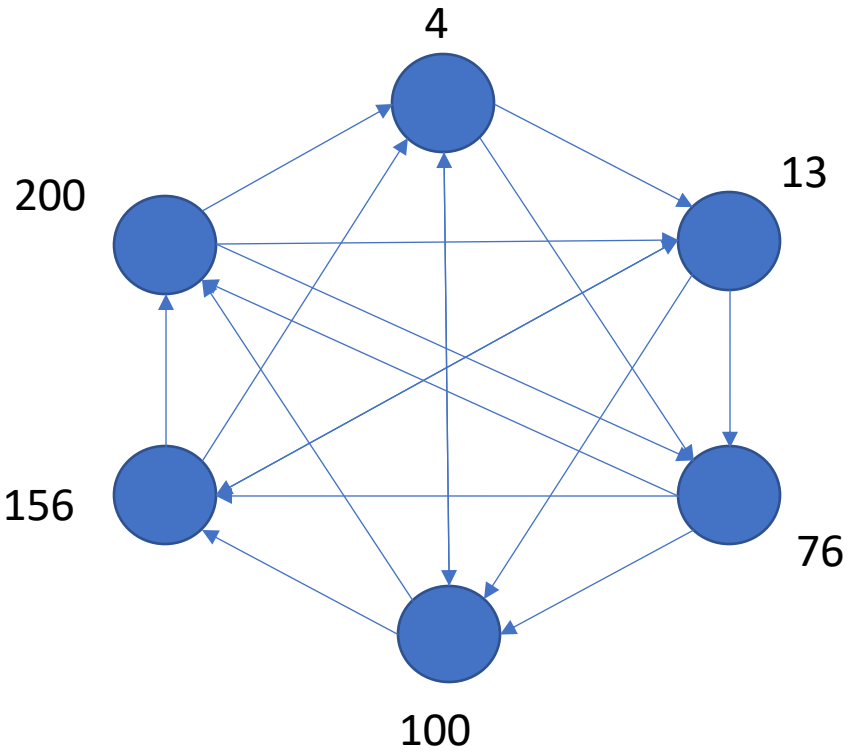
We add some additional overlay links (yes in the order of ln(# π)) to speed up things.

This solves two problems:

- **We now have information to deal with faults.**

Structured Overlay Networks (second step)

- The most common overlay topology in structured overlay networks are rings, that connect nodes in order considering their identifiers.



We add some additional overlay links (yes in the order of ln(# π)) to speed up things.

This solves two problems:

- **We now have information to deal with faults.**
- **But now, I can also reach any other process in a logarithmic number of hops (I can always reduce in half the distance to my target at each hop).**

Structured Overlay Networks: Relevant Examples

- There are a few relevant examples of such algorithms in the Literature:
- Chord (Canonical Academic Example)
- Pastry (Similar principles, different Algorithm)
- Kadmelia (and its famous implementation Kad)

Structured Overlay Networks: Relevant Examples

- There are a few relevant examples of such algorithms in the Literature:
- **Chord (Canonical Academic Example)**
- Pastry (Similar principles, different Algorithm)
- Kadmelia (and its famous implementation Kad)

The Chord Protocol

Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

Ion Stoica[†], Robert Morris[‡], David Liben-Nowell[‡], David R. Karger[‡], M. Frans Kaashoek[‡], Frank Dabek[‡], Hari Balakrishnan[‡]

The Chord Protocol

- What is the state kept by each node?
- Assuming that process identifiers have m bits.

finger[k]	first node on circle that succeeds $(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
successor	the next node on the identifier circle; *finger*[1].*node*
predecessor	the previous node on the identifier circle

The Chord Protocol

- The key functionality of Chord (i.e., its interface), given an identifier, find the process responsible for managing that identifier.

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    n' = closest_preceding_node(id);
    return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i] ∈ (n, id))
      return finger[i];
  return n;
```

The Chord Protocol

- How do we manage the finger table (that materialize the ring and extra paths)
- The “Init” step (which is named “create” in the paper pseudocode)

```
// create a new Chord ring.
n.create()
    predecessor = nil;
    successor = n;

// join a Chord ring containing node n'.
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);
```

The Chord Protocol

- How do we manage the finger table (that materialize the ring and extra paths)
- Ensuring that the successor is correct:

```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';
```

The Chord Protocol

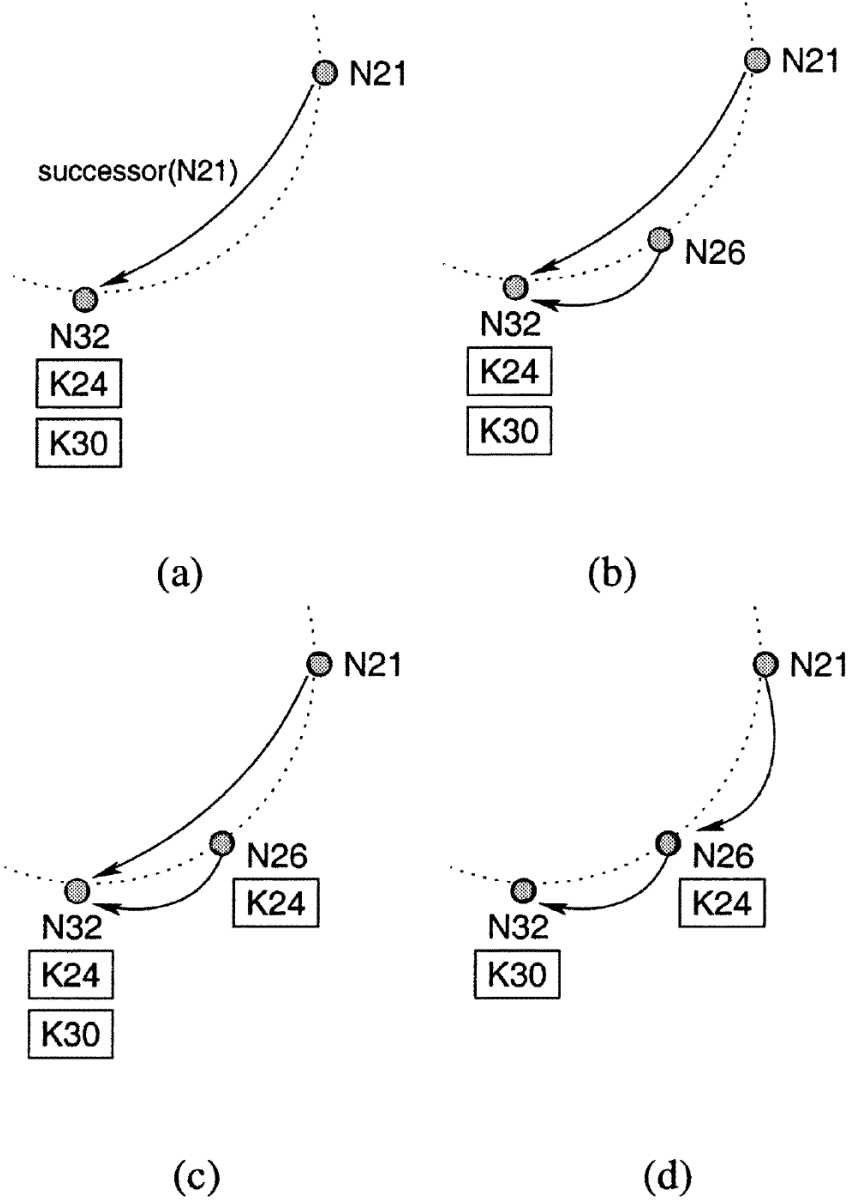


Fig. 7. Example illustrating the join operation. Node 26 joins the system between nodes 21 and 32. The arcs represent the successor relationship. (a) Initial state: node 21 points to node 32. (b) Node 26 finds its successor (i.e., node 32) and points to it. (c) Node 26 copies all keys less than 26 from node 32. (d) The stabilize procedure updates the successor of node 21 to node 26.

The Chord Protocol

- How do we manage the finger table (that materialize the ring and extra paths)
- Ensuring that all other links are correct:
  - fix_fingers
  - check_predecessor

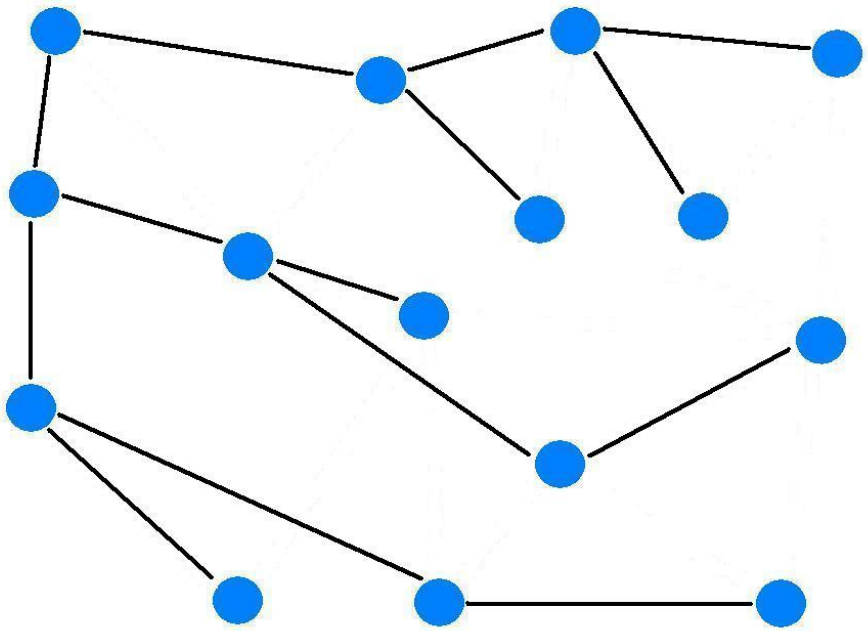
```
// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find_successor(n + 2^(next-1));

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
```

Other Structured Overlay Networks.

- The definition only states that the topology of the overlay has some a-priori known property.
- One can be that nodes are organized in an ordered ring, which is good to find and route information among nodes.
- Can you think of another one?

Other Structured Overlay Networks.

- Tree-based overlay networks:
  - Good for disseminate messages with low overhead.
  - Also, good to aggregate information.



Overview of Overlays:

- Unstructured (or Random):
  - + : Easy to build and maintain.
  - + : Robust to failures (any failed process can be replaced by any other failed process).
  - - : Limited efficiency for some use cases (locate a particular object or process for instance).
- Structured
  - + : Provides efficiency for particular types of applications (application-level routing, exact-search, broadcast).
  - - : Less robust to failures (a failed process can only be replaced – in another process partial view – by a limited number of other processes).
  - - : Somewhat more complex algorithms.

A final note on Gossip and Overlays

- Many of these works were originated by the Peer-to-Peer community, which might lead to the question “why does this still matters now-a-days?”

A final note on Gossip and Overlays

- Many of these works were originated by the Peer-to-Peer community, which might lead to the question “why does this still matters now-a-days?” or “why does this still matters now-a-days, Netflix is a thing now, so who wants to share files online?”

A final note on Gossip and Overlays

- Many of these works were originated by the Peer-to-Peer community, which might lead to the question “why does this still matters now-a-days?” or “why does this still matters now-a-days, Netflix is a thing now, so who wants to share files online?”

A final note on Gossip and Overlays

- Many of these works were originated by the Peer-to-Peer community, which might lead to the question "why does this still matters now-a-days?" or "why does this still matters now-a-days, Netflix is a thing now, so who wants to share files online?"
- All these P2P approaches have found new applications in the context of cloud-computing, namely, to help manage very large systems running in the cloud.

A final note on Gossip and Overlays

- Many of these works were originated by the Peer-to-Peer community, which might lead to the question “why does this still matters now-a-days?” or “why does this still matters now-a-days, Netflix is a thing now, so who wants to share files online?”
- All these P2P approaches have found new applications in the context of cloud-computing, namely, to help manage very large systems running in the cloud.
- Now-a-Days we live on the Web 3.0, Internet-of-Things, and Edge Computing eras... Many of these solutions can be useful there.

Web 3.0

"(…) The technologies which will make up these properties include microformats, data mining, natural language search and machine learning. Web 3.0 will also be more focused on peer-to-peer (P2P) technologies such as blockchain. Other technologies such as open APIs, data formats and open sourced software may also be used while developing Web 3.0 applications. (…)"

-- From "Web 3.0" extracted from WhatIs.com

Web 3.0

- Blockchain
  - Large number of miners (individual processes) compete to solve a crypto puzzle. When the puzzle is solved, a new block is formed and must be disseminated to all miners.
- Challenges:
  - Tracking membership (i.e., active miners)
  - Disseminate blocks when they are mined

Web 3.0

- Blockchain
  - Large number of miners (individual processes) compete to solve a crypto puzzle. When the puzzle is solved, a new block is formed and must be disseminated to all miners.
- Challenges:
  - Tracking membership (i.e., active miners)
    - Unstructured overlay network
  - Disseminate blocks when they are mined
    - Flood based broadcast

Web 3.0

- Decentralized/Collaborative storage and serving of data (potentially to serve web pages).
- Interplanetary File System (IPFS)
  - Contents are divided into blocks. Both content descriptions (i.e., which blocks are part of it) and individual blocks are attributed an identifier based on their content. Location of content is distributed and fetched by anyone that knows the identifier.
- Challenges:
  - How to build the distributed index
  - How to fetch blocks quickly

Web 3.0

- Decentralized/Collaborative storage and serving of data (potentially to serve web pages).
- Interplanetary File System (IPFS)
  - Contents are divided into blocks. Both content descriptions (i.e., which blocks are part of it) and individual blocks are attributed an identifier based on their content. Location of content is distributed and fetched by anyone that knows the identifier.
- Challenges:
  - How to build the distributed index
    - Use a Distributed Hash Table (Kademlia)
  - How to fetch blocks quickly
    - Gossip protocol to accelerate access to blocks (bitswap)

Edge Computing Area

- Cloud datacenter can scale (virtual notion of infinite resources).
- Network links to cloud datacenters however cannot easily scale.

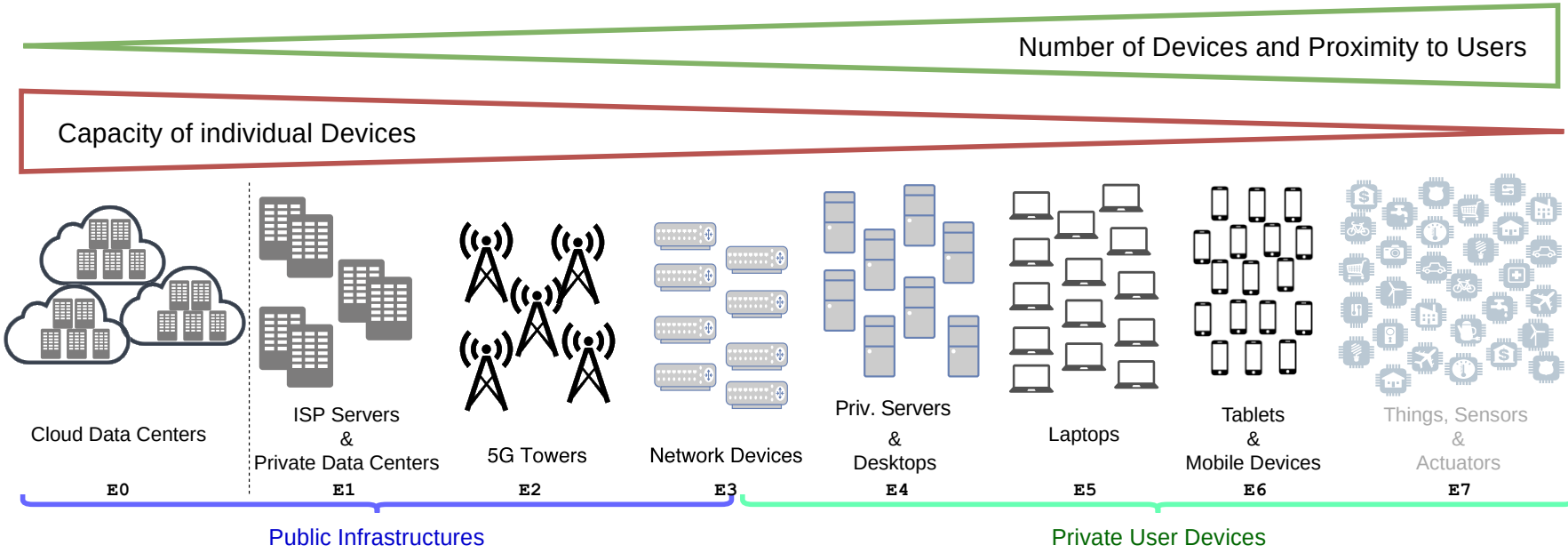
- Problem: Applications that generate lots of data might have problem in shipping all that data to cloud infrastructures and get answers in timely fashion

  (e.g., IoT, Mobile games)

- Solution: Put computations beyond the data center boundaries.

Edge Computing

A high-level perspective:



Leitão et. al. 2018
(European project LightKone)

Some challenges

- Lots of devices to manage and different administrative domains.
- Need to make adequate choices regarding where to execute different computations.
- Computations need data, so data must be also managed on edge devices.
- Security: Data privacy, Data integrity
  (check CSD course)

Some challenges

- Lots of devices to manage and different administrative domains.
- Lots of different challenges to be addressed in the next few years.
- Maybe you will be one of the key people is addressing these challenges! ☺
- Security: Data privacy, Data integrity
  (check CSD course)