

# Algorithms and Distributed Systems 2022/2023 (Lecture Two)

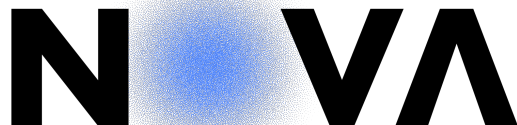
**MIEI - Integrated Master in Computer Science and  
Informatics**

**MEI – Master in Computer Science and  
Informatics**

Specialization block

**Nuno Preguiça** ([nmp@fct.unl.pt](mailto:nmp@fct.unl.pt))

Alex Davidson ([a.davidson@fct.unl.pt](mailto:a.davidson@fct.unl.pt))



NOVA SCHOOL OF  
SCIENCE & TECHNOLOGY

# Acknowledgments

- Slides mostly from João Leitão.

# Lecture structure:

- Homework discussion
- More on Broadcast...
- Membership Protocols.
- The (many) flavors of Gossip.

# Homework 1:

- Write the Pseudo-Code for solving the Reliable Broadcast Problem assuming:
  - A Fail stop model and a synchronous system.
  - Your solution **must** ensure all properties of the Reliable Broadcast Problem.
  - Your solution should ensure that in fault-free executions (i.e., when no process crashes) each process collaborates in the dissemination by sending a single message.

# Homework 1:

- Reliable Broadcast:
  - RB1 (Validity): If a correct process  $i$  broadcasts message  $m$ , then  $i$  eventually delivers the message.
  - RB2 (No Duplications): No message is delivered more than once.
  - RB3 (No Creation): If a correct process  $j$  delivers a message  $m$ , then  $m$  was broadcast to  $j$  by some process  $i$ .
  - RB4 (Agreement): If a message  $m$  is delivered by some correct process  $i$ , then  $m$  is eventually delivered by every correct process  $j$ .

# Homework 1:

- Hints:
  - Since we are in the synchronous environment and in the fail stop model, when a process  $p$  crashes all correct processes will trigger an event:  
***Upon** crash( $p$ ).*
  - You can assume that process identifiers are sequential numbers starting at 1 and going up to  $\#(\pi)$
  - The special Init even can receive arguments, might be useful for you to receive both  $\pi$  the local process identifier  $p$  (which as said above can be interpreted as a number)

**Interface:****Requests:****rBroadcast** (  $m$  )**Indications:****rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message**State:** $myself$  // my own identifier $correct$  // correct processes identifiers $delivered$  // messages already delivered $messages$  // Map that associates to each process  $p$  the messages dependent on it**Upon Init** (  $\Pi$ ,  $self$  ) **do:** $myself \leftarrow self$  $correct \leftarrow \Pi$  $delivered \leftarrow \{\}$ **Foreach**  $p \in correct$  **do:** $messages[p] \leftarrow \{\}$ **Upon rBroadcast** (  $m$  ) **do:****Trigger** **rBcastDeliver** (  $myself, m$  ) $delivered \leftarrow delivered \cup \{m\}$  $p \leftarrow p \in correct: p < p', \forall p' \in correct: p \neq p' \wedge p \neq myself$ **If**  $p \neq \perp$  **Then****Trigger Send**( **BCAST**,  $p, self, m$  ) $messages[p] \leftarrow messages[p] \cup \{(myself, m)\}$ **Upon Receive**( **BCAST**,  $s, p, m$  ) **do:****If**  $m \notin delivered$  **Then** $delivered \leftarrow delivered \cup \{m\}$  $d \leftarrow d \in correct: d > myself \wedge d \neq p \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq p \wedge d' > myself$ **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d, p, m$  ) $messages[d] \leftarrow messages[d] \cup \{(p, m)\}$ **Upon Crash** (  $p$  ) **do:** $correct \leftarrow correct \setminus \{p\}$ **Foreach** (  $s, m$  )  $\in messages[p]$  **do:** $d \leftarrow d \in correct: d > p \wedge d \neq s \wedge d \neq myself \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq s \wedge d' > p$ **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d, s, m$  ) $messages[d] \leftarrow messages[d] \cup \{(s, m)\}$ 

# Homework 1

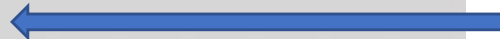
## (possible)

## Solution


**Interface:****Requests:****rBroadcast** (  $m$  )**Indications:****rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message**State:** $myself$  // my own identifier $correct$  // correct processes identifiers $delivered$  // messages already delivered $messages$  // Map that associates to each process  $p$  the messages dependent on it**Upon Init** (  $\Pi$ ,  $self$  ) **do:** $myself \leftarrow self$  $correct \leftarrow \Pi$  $delivered \leftarrow \{\}$ **Foreach**  $p \in correct$  **do:** $messages[p] \leftarrow \{\}$ **Upon rBroadcast** (  $m$  ) **do:****Trigger** **rBcastDeliver** (  $myself, m$  ) $delivered \leftarrow delivered \cup \{m\}$  $p \leftarrow p \in correct: p < p', \forall p' \in correct: p \neq p' \wedge p \neq myself$ **If**  $p \neq \perp$  **Then****Trigger Send**( **BCAST**,  $p, self, m$  ) $messages[p] \leftarrow messages[p] \cup \{(myself, m)\}$ **Upon Receive**( **BCAST**,  $s, p, m$  ) **do:****If**  $m \notin delivered$  **Then** $delivered \leftarrow delivered \cup \{m\}$  $d \leftarrow d \in correct: d > myself \wedge d \neq p \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq p \wedge d' > myself$ **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d, p, m$  ) $messages[d] \leftarrow messages[d] \cup \{(p, m)\}$ **Upon Crash** (  $p$  ) **do:** $correct \leftarrow correct \setminus \{p\}$ **Foreach** (  $s, m$  )  $\in messages[p]$  **do:** $d \leftarrow d \in correct: d > p \wedge d \neq s \wedge d \neq myself \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq s \wedge d' > p$ **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d, s, m$  ) $messages[d] \leftarrow messages[d] \cup \{(s, m)\}$ 

# Homework 1 (possible) Solution

This will pick up the destination of the message as the (correct) process with the lowest identifier (other than myself)





**Interface:****Requests:****rBroadcast** (  $m$  )**Indications:****rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message**State:** $myself$  // my own identifier $correct$  // correct processes identifiers $delivered$  // messages already delivered $messages$  // Map that associates to each process  $p$  the messages dependent on it**Upon Init** (  $\Pi$ ,  $self$  ) **do:** $myself \leftarrow self$  $correct \leftarrow \Pi$  $delivered \leftarrow \{\}$ **Foreach**  $p \in correct$  **do:** $messages[p] \leftarrow \{\}$ **Upon rBroadcast** (  $m$  ) **do:****Trigger** **rBcastDeliver** (  $myself, m$  ) $delivered \leftarrow delivered \cup \{m\}$  $p \leftarrow p \in correct: p < p', \forall p' \in correct: p \neq p' \wedge p \neq myself$ **If**  $p \neq \perp$  **Then****Trigger Send**( **BCAST**,  $p$ ,  $self$ ,  $m$  ) $messages[p] \leftarrow messages[p] \cup \{(myself, m)\}$ **Upon Receive**( **BCAST**,  $s, p, m$  ) **do:****If**  $m \notin delivered$  **Then** $delivered \leftarrow delivered \cup \{m\}$  $d \leftarrow d \in correct: d > myself \wedge d \neq p \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq p \wedge d' > myself$  **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d$ ,  $p$ ,  $m$  ) $messages[d] \leftarrow messages[d] \cup \{(p, m)\}$ **Upon Crash** (  $p$  ) **do:** $correct \leftarrow correct \setminus \{p\}$ **Foreach** (  $s, m$  )  $\in messages[p]$  **do:** $d \leftarrow d \in correct: d > p \wedge d \neq s \wedge d \neq myself \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq s \wedge d' > p$ **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d$ ,  $s$ ,  $m$  ) $messages[d] \leftarrow messages[d] \cup \{(s, m)\}$ 

# Homework 1 (possible) Solution

This will pick up the destination of the message as the (correct) process with identifier immediately after mine that is different from the original sender of the message.

**Interface:****Requests:****rBroadcast** (  $m$  )**Indications:****rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message**State:** $myself$  // my own identifier $correct$  // correct processes identifiers $delivered$  // messages already delivered $messages$  // Map that associates to each process  $p$  the messages dependent on it**Upon Init** (  $\Pi$ ,  $self$  ) **do:** $myself \leftarrow self$  $correct \leftarrow \Pi$  $delivered \leftarrow \{\}$ **Foreach**  $p \in correct$  **do:** $messages[p] \leftarrow \{\}$ **Upon rBroadcast** (  $m$  ) **do:****Trigger** **rBcastDeliver** (  $myself, m$  ) $delivered \leftarrow delivered \cup \{m\}$  $p \leftarrow p \in correct: p < p', \forall p' \in correct: p \neq p' \wedge p \neq myself$ **If**  $p \neq \perp$  **Then****Trigger Send**( **BCAST**,  $p, self, m$  ) $messages[p] \leftarrow messages[p] \cup \{(myself, m)\}$ **Upon Receive**( **BCAST**,  $s, p, m$  ) **do:****If**  $m \notin delivered$  **Then** $delivered \leftarrow delivered \cup \{m\}$  $d \leftarrow d \in correct: d > myself \wedge d \neq p \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq p \wedge d' > myself$ **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d, p, m$  ) $messages[d] \leftarrow messages[d] \cup \{(p, m)\}$ **Upon Crash** (  $p$  ) **do:** $correct \leftarrow correct \setminus \{p\}$ **Foreach** (  $s, m$  )  $\in messages[p]$  **do:** $d \leftarrow d \in correct: d > p \wedge d \neq s \wedge d \neq myself \wedge d < d', \forall d' \in correct: d \neq d' \wedge d' \neq s \wedge d' > p$ **If**  $d \neq \perp$  **Then****Trigger Send**( **BCAST**,  $d, s, m$  ) $messages[d] \leftarrow messages[d] \cup \{(s, m)\}$ 

# Homework 1 (possible) Solution

This will pick up the destination of the message as the (correct) process that is after the failed process and that is diferente from myself and the original sender of the message.

# Lecture structure:

- **More on Broadcast...**
- Membership Protocols.
- The (many) flavors of Gossip.

# Reliable Broadcast Problem (last lecture)

- Asynchronous System under the Crash Fault Model
- Reliable Broadcast:
  - RB1 (Validity): If a correct process  $i$  broadcasts message  $m$ , then  $i$  eventually delivers the message.
  - RB2 (No Duplications): No message is delivered more than once.
  - RB3 (No Creation): If a correct process  $j$  delivers a message  $m$ , then  $m$  was broadcast to  $j$  by some process  $i$ .
  - RB4 (Agreement): If a message  $m$  is delivered by some correct process  $i$ , then  $m$  is eventually delivered by every correct process  $j$ .

# Reliable Broadcast Problem

In all fairness in the Literature: *Eager Reliable Broadcast*

State:

delivered //set of message ids that were already delivered.

**Upon** Init **do**:

delivered  $\leftarrow \{\}$ ;

**Upon** rbBroadcast( m ) **do**

**trigger** rbDeliver(m);

mid  $\leftarrow$  generateUniqueID(m);

delivered  $\leftarrow$  delivered  $\cup$  {mid};

**trigger** bebBroadcast( { mid, m } );

**Upon** bebDeliver(p, { mid, m } ) **do**

**if** ( mid  $\notin$  delivered ) **then**

delivered  $\leftarrow$  delivered  $\cup$  {mid};

**trigger** rbDeliver(m);

**trigger** bebBroadcast(mid, m);

# Different Setting

- Solution for the Reliable Broadcast Problem assuming:
  - A synchronous system and the fail-stop model.
  - This combination, implies that there is a special event that is automatically triggered at each process (and received by your protocol) that notifies that some process  $p$  has crashed.
  - This event is captured by: **Upon Crash (  $p$  ) do:**
  - **Should ensure that the number of redundant messages transmitted in fault-free runs is zero (but processes can send any number of messages for each broadcasted message).**
- **Bonus Goal:** Make sure that when a process fails you minimize the number of messages that must be retransmitted.
- You can use the best effort broadcast protocol as a building block.

# Building Block (optional): Best Effort Broadcast

---

**Algorithm 1:** Best Effort Broadcast (Building Block: Crash / Asynchronous (operates on top of the PP2PLink Abstraction))

---

**Interface:**

**Requests:**

**bebBroadcast** (  $m$  )

**Indications:**

**bebBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message

**State:**

**Upon Init () do:**

**Upon bebBroadcast** (  $m$  ) **do:**

**Forall**  $p \in \Pi$  **do:**

**Trigger** pp2pSend (  $p, m$  )

**Upon pp2pDeliver** (  $s, m$  ) **do:**

**Trigger** bebBcastDeliver (  $s, m$  )

---

# Reliable Broadcast – Synchronous System / Fail-Stop Fault Model (Base Solution)

---

**Algorithm 2:** Reliable Broadcast (Fail-Stop / Synchronous) Base Solution

---

**Interface:**

**Requests:**

**rBroadcast** (  $m$  )

**Indications:**

**rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message

**State:**

delivered //Ids of messages already delivered

messages //Map that associates to each process  $p$  the messages broadcasted by it

**Upon Init () do:**

delivered  $\leftarrow \{\}$

messages  $\leftarrow \{\}$

**Foreach**  $p \in \Pi$  **do:**

messages[ $p$ ]  $\leftarrow \{\}$

**Upon rBroadcast(  $m$  ) do:**

mid  $\leftarrow$  generateUniqueID( $m$ )

**Trigger** rBcastDeliver (  $m$  )

delivered  $\leftarrow$  delivered  $\cup \{mid\}$

**Trigger** bebBroadcast ( {  $mid, m$  } )

**Upon bebDeliver (  $s, \{mid, m\}$  ) do:**

**If** mid  $\notin$  delivered **then**

delivered  $\leftarrow$  delivered  $\cup \{mid\}$

messages[ $s$ ]  $\leftarrow$  messages[ $s$ ]  $\cup \{(mid, m)\}$

**Trigger** rBcastDeliver (  $m$  )

**Upon Crash (  $p$  ) do:**

**Foreach** (mid,  $m$ )  $\in$  messages[ $p$ ] **do:**

**Trigger** bebBroadcast ( {  $mid, m$  } )

---



# Improved Solution (I)

---

**Algorithm 3: Reliable Broadcast (Fail-Stop / Synchronous) With Bonus (I)**

---

**Interface:****Requests:****rBroadcast** (  $m$  )**Indications:****rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message**State:**

delivered //Ids of messages already delivered  
messages //list of tuples containing sender, message identifiers, and message,  
for all messages received (and not confirmed to be delivered to all)  
acks //Map that associates for each message the set of processes that have  
confirmed to deliver it  
correct //Set containing correct processes

**Upon Init () do:**

delivered  $\leftarrow \{\}$   
messages  $\leftarrow \{\}$   
**Foreach**  $p \in \Pi$  **do:**  
    messages[p]  $\leftarrow \{\}$   
acks  $\leftarrow \{\}$   
correct  $\leftarrow \Pi$

**Upon rBroadcast(  $m$  ) do:**

mid  $\leftarrow$  generateUniqueID( $m$ )  
**Trigger** rBcastDeliver (  $m$  )  
delivered  $\leftarrow$  delivered  $\cup \{mid\}$   
**Trigger** bebBroadcast ( {BCAST, mid,  $m$ } )

**Upon bebDeliver (  $s, \{BCAST, mid, m\}$  ) do:**

**If** mid  $\notin$  delivered **then**  
    delivered  $\leftarrow$  delivered  $\cup \{mid\}$   
    messages  $\leftarrow$  messages  $\cup \{(s, mid, m)\}$   
    **Foreach**  $p : p \in \text{correct} \wedge p \neq \{myself, s\}$  **do:**  
        **Trigger** pp2pSend (  $p, \{ACK, mid\}$  )  
        **Trigger** rBcastDeliver (  $m$  )  
acks[mid]  $\leftarrow$  acks[mid]  $\cup \{s\}$   
**Call** checkAcks (  $mid$  )

**Upon pp2pDeliver (  $s, \{ACK, mid\}$  ) do:**

acks[mid]  $\leftarrow$  acks[mid]  $\cup \{s\}$   
**Call** checkAcks (  $mid$  )

**Procedure checkAcks (  $mid$  )**

**If** mid  $\in$  delivered  $\wedge$  acks[mid]  $\supseteq$  ( correct  $\setminus$  myself ) **Then**  
    confirmed  $\leftarrow \{(s, id, m) | (s, id, m) \in \text{messages} \wedge id = mid\}$   
    messages  $\leftarrow$  messages  $\setminus$  confirmed

**Upon Crash (  $p$  ) do:**

correct  $\leftarrow$  correct  $\setminus p$   
**Forall** mid  $\in$  delivered **do:**  
    **Call** checkAcks (  $mid$  )  
**Foreach** (  $s, mid, m$  )  $\in$  messages :  $s = p$  **do:**  
    **Trigger** bebBroadcast ( {BCAST, mid,  $m$ } )

- Intuition:

- Whenever a process receives a message  $m$ , it sends acks to all process except myself and the sender.
- Whenever a process collects acks from everyone else it removes the message from the message set since it knows the message is stable.
- When a process fails, every other process rebroadcasts messages sent by him that are not stable.

# Improved Solution (I')

**Algorithm 4:** Reliable Broadcast (Fail-Stop / Synchronous) With Bonus (I) - remove  
bebBroadcast

**Interface:**

**Requests:**

**rBroadcast** (  $m$  )

**Indications:**

**rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message

**State:**

$messages$  // list of tuples containing sender, message identifiers, and message,  
for all messages received (and not confirmed to be delivered to all)  
 $acks$  // Map that associates for each message the set of processes that have  
confirmed to deliver it  
 $correct$  // Set containing correct processes

**Upon Init** () do:

$delivered \leftarrow \{\}$   
 $messages \leftarrow \{\}$   
**Foreach**  $p \in \Pi$  **do:**  
     $messages[p] \leftarrow \{\}$   
 $acks \leftarrow \{\}$   
 $correct \leftarrow \Pi$

**Upon rBroadcast**(  $m$  ) **do:**

$mid \leftarrow \text{generateUniqueID}(m)$   
**Trigger** **rBcastDeliver** (  $m$  )  
 $delivered \leftarrow delivered \cup \{mid\}$   
**Foreach**  $p \in correct \setminus \{myself\}$  **do:**  
    **Trigger** **pp2pSend** (  $p, \{BCAST, mid, m\}$  )

**Upon pp2pDeliver** (  $s, \{BCAST, mid, m\}$  ) **do:**

**If**  $mid \notin delivered$  **then**  
     $delivered \leftarrow delivered \cup \{mid\}$   
     $messages \leftarrow messages \cup \{(s, mid, m)\}$   
    **Foreach**  $p : p \in correct \wedge p \neq \{myself, s\}$  **do:**  
        **Trigger** **pp2pSend** (  $p, \{ACK, mid\}$  )  
    **Trigger** **rBcastDeliver** (  $m$  )  
     $acks[mid] \leftarrow acks[mid] \cup \{s\}$   
    **Call** **checkAcks** (  $mid$  )

**Upon pp2pDeliver** (  $s, \{ACK, mid\}$  ) **do:**

$acks[mid] \leftarrow acks[mid] \cup \{s\}$   
**Call** **checkAcks** (  $mid$  )

**Procedure checkAcks** (  $mid$  )

**If**  $mid \in delivered \wedge acks[mid] \supseteq (correct \setminus myself)$  **Then**  
     $confirmed \leftarrow \{(s, id, m) | (s, id, m) \in messages \wedge id = mid\}$   
     $messages \leftarrow messages \setminus confirmed$

**Upon Crash** (  $p$  ) **do:**

$correct \leftarrow correct \setminus p$   
**Forall**  $mid \in delivered$  **do:**  
    **Call** **checkAcks** (  $mid$  )  
**Foreach**  $(s, mid, m) \in messages : s = p$  **do:**  
    **Foreach**  $q : q \in correct \wedge q \neq myself$  **do:**  
        **Trigger** **pp2pSend** (  $q, \{BCAST, mid, m\}$  )

- I can express the same algorithm as before without using the best effort broadcast.

# Improved Solution (II)

---

**Algorithm 5:** Reliable Broadcast (Fail-Stop / Synchronous) With Bonus (II) – avoid obvious redundant

---

**Interface:**

**Requests:**

**rBroadcast** (  $m$  )

**Indications:**

**rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message

**State:**

delivered // Ids of messages already delivered  
 messages // list of tuples containing sender, message identifiers, and message,  
           for all messages received (and not confirmed to be delivered to all)  
 acks // Map that associates for each message the set of processes that have  
           confirmed to deliver it  
 correct // Set containing correct processes

**Upon Init () do:**

delivered  $\leftarrow \{\}$   
 messages  $\leftarrow \{\}$   
**Foreach**  $p \in \Pi$  **do:**  
     messages[ $p$ ]  $\leftarrow \{\}$   
 acks  $\leftarrow \{\}$   
 correct  $\leftarrow \Pi$

**Upon rBroadcast(  $m$  ) do:**

mid  $\leftarrow$  generateUniqueID( $m$ )  
**Trigger** rBcastDeliver (  $m$  )  
 delivered  $\leftarrow$  delivered  $\cup \{mid\}$   
**Foreach**  $p \in correct \setminus \{myself\}$  **do:**  
     **Trigger** pp2pSend (  $p, \{BCAST, mid, m\}$  )

**Upon pp2pDeliver (  $s, \{BCAST, mid, m\}$  ) do:**

**If** mid  $\notin$  delivered **then**  
     delivered  $\leftarrow$  delivered  $\cup \{mid\}$   
     messages  $\leftarrow$  messages  $\cup \{(s, mid, m)\}$   
     **Foreach**  $p : p \in correct \wedge p \neq \{myself, s\}$  **do:**  
         **Trigger** pp2pSend (  $p, \{ACK, mid\}$  )  
         **Trigger** rBcastDeliver (  $m$  )  
     acks[mid]  $\leftarrow$  acks[mid]  $\cup \{s\}$   
     **Call** checkAcks (  $mid$  )

**Upon pp2pDeliver (  $s, \{ACK, mid\}$  ) do:**

acks[mid]  $\leftarrow$  acks[mid]  $\cup \{s\}$   
**Call** checkAcks (  $mid$  )

**Procedure checkAcks (  $mid$  )**

**If** mid  $\in$  delivered  $\wedge$  acks[mid]  $\supseteq (correct \setminus myself)$  **Then**  
     confirmed  $\leftarrow \{(s, id, m) | (s, id, m) \in messages \wedge id = mid\}$   
     messages  $\leftarrow$  messages  $\setminus$  confirmed

**Upon Crash (  $p$  ) do:**

correct  $\leftarrow$  correct  $\setminus p$   
**Forall** mid  $\in$  delivered **do:**  
     **Call** checkAcks (  $mid$  )  
**Foreach** (  $s, mid, m$  )  $\in$  messages :  $s = p$  **do:**  
     **Foreach**  $q : q \in correct \wedge q \neq myself \wedge q \notin acks[mid]$  **do:**  
         **Trigger** pp2pSend (  $q, \{BCAST, mid, m\}$  )

- Now, when a process fails, I can retransmit the messages that were sent by that process only to the processes for which I did not get an acknowledgment.

# Improved Solution (III)

**Algorithm 6:** Reliable Broadcast (Fail-Stop / Synchronous) With Bonus (III) – piggyback  
acks

**Interface:**

**Requests:**

**rBroadcast** (  $m$  )

**Indications:**

**rBcastDeliver** (  $s, m$  ) //  $s$  is the sender,  $m$  the message

**State:**

**delivered** // Ids of messages already delivered  
**messages** // list of tuples containing sender, message identifiers, and message,  
for all messages received (and not confirmed to be delivered to all)  
**acks** // Map that associates for each message the set of processes that have  
confirmed to deliver it  
**correct** // Set containing correct processes

**Upon Init** () do:

**delivered**  $\leftarrow \{\}$   
**messages**  $\leftarrow \{\}$   
**Foreach**  $p \in \Pi$  **do:**  
    **messages**[ $p$ ]  $\leftarrow \{\}$   
**acks**  $\leftarrow \{\}$   
**correct**  $\leftarrow \Pi$

**Upon rBroadcast**(  $m$  ) **do:**

**mid**  $\leftarrow \text{generateUniqueID}(m)$   
**Trigger rBcastDeliver** (  $m$  )  
**delivered**  $\leftarrow \text{delivered} \cup \{\text{mid}\}$   
**Foreach**  $p \in \text{correct} \setminus \{\text{myself}\}$  **do:**  
    **Trigger pp2pSend** (  $p, \{\text{BCAST}, \text{mid}, m, \text{delivered}\}$  )

**Upon pp2pDeliver** (  $s, \{\text{BCAST}, \text{mid}, m, s\text{Delivered}\}$  ) **do:**

**If**  $\text{mid} \notin \text{delivered}$  **then**  
    **delivered**  $\leftarrow \text{delivered} \cup \{\text{mid}\}$   
    **messages**  $\leftarrow \text{messages} \cup \{(s, \text{mid}, m)\}$   
    **Trigger rBcastDeliver** (  $m$  )  
**Foreach**  $\text{mid} \in s\text{Delivered}$  **do:**  
    **acks**[ $\text{mid}$ ]  $\leftarrow \text{acks}[\text{mid}] \cup \{s\}$   
    **Call** **checkAcks** (  $\text{mid}$  )

**Procedure checkAcks** (  $\text{mid}$  )

**If**  $\text{mid} \in \text{delivered} \wedge \text{acks}[\text{mid}] \supseteq (\text{correct} \setminus \text{myself})$  **Then**  
    **confirmed**  $\leftarrow \{(s, \text{id}, m) \mid (s, \text{id}, m) \in \text{messages} \wedge \text{id} = \text{mid}\}$   
    **messages**  $\leftarrow \text{messages} \setminus \text{confirmed}$

**Upon Crash** (  $p$  ) **do:**

**correct**  $\leftarrow \text{correct} \setminus p$   
**Forall**  $\text{mid} \in \text{delivered}$  **do:**  
    **Call** **checkAcks** (  $\text{mid}$  )  
**Foreach** (  $s, \text{mid}, m$  )  $\in \text{messages} : s = p$  **do:**  
    **Foreach**  $q : q \in \text{correct} \wedge q \neq \text{myself} \wedge q \notin \text{acks}[\text{mid}]$  **do:**  
        **Trigger pp2pSend** (  $q, \{\text{BCAST}, \text{mid}, m, \text{delivered}\}$  )

- I can avoid to send explicit acknowledgment messages by simply piggybacking my set of already delivered messages on every message that I broadcast.

# Improved Solution (IV)

**Algorithm 7:** Reliable Broadcast (Fail-Stop / Synchronous) With Bonus (III) – aggressive piggybacking

**Interface:**

**Requests:**

**rBroadcast** ( $m$ )

**Indications:**

**rBcastDeliver** ( $s, m$ ) //  $s$  is the sender,  $m$  the message

**State:**

**delivered** // Ids of messages already delivered  
**messages** // list of tuples containing sender, message identifiers, and message, for all messages received (and not confirmed to be delivered to all)  
**acks** // Map that associates for each message the set of processes that have confirmed to deliver it  
**correct** // Set containing correct processes

**Upon Init () do:**

**delivered**  $\leftarrow \{\}$   
**messages**  $\leftarrow \{\}$   
**Foreach**  $p \in \Pi$  **do:**  
    **messages**[ $p$ ]  $\leftarrow \{\}$   
**acks**  $\leftarrow \{\}$   
**correct**  $\leftarrow \Pi$

**Upon rBroadcast(  $m$  ) do:**

**mid**  $\leftarrow$  generateUniqueID( $m$ )  
**Trigger** **rBcastDeliver** ( $m$ )  
**delivered**  $\leftarrow$  **delivered**  $\cup \{mid\}$   
**acks**[ $mid$ ]  $\leftarrow$  **acks**[ $mid$ ]  $\cup \{myself\}$   
**Foreach**  $p \in \text{correct} \setminus \{myself\}$  **do:**  
    **Trigger** **pp2pSend** ( $p, \{\text{BCAST}, mid, m, acks\}$ )

**Upon pp2pDeliver (  $s, \{\text{BCAST}, mid, m, sAcks\}$  ) do:**

**If**  $mid \notin \text{delivered}$  **then**  
    **delivered**  $\leftarrow$  **delivered**  $\cup \{mid\}$   
    **acks**[ $mid$ ]  $\leftarrow$  **acks**[ $mid$ ]  $\cup \{myself\}$   
    **messages**  $\leftarrow$  **messages**  $\cup \{(s, mid, m)\}$   
    **Trigger** **rBcastDeliver** ( $m$ )  
**Foreach**  $mid \in sAcks$  **do:**  
    **acks**[ $mid$ ]  $\leftarrow$  **acks**[ $mid$ ]  $\cup sAcks$   
    **Call** **checkAcks** ( $mid$ )

**Procedure checkAcks (  $mid$  )**

**If**  $mid \in \text{delivered} \wedge \text{acks}[mid] \supseteq (\text{correct} \setminus \text{myself})$  **Then**  
    **confirmed**  $\leftarrow \{(s, id, m) \mid (s, id, m) \in \text{messages} \wedge id = mid\}$   
    **messages**  $\leftarrow$  **messages**  $\setminus \text{confirmed}$

**Upon Crash (  $p$  ) do:**

**correct**  $\leftarrow$  **correct**  $\setminus p$   
**Forall**  $mid \in \text{delivered}$  **do:**  
    **Call** **checkAcks** ( $mid$ )  
**Foreach**  $(s, mid, m) \in \text{messages} : s = p$  **do:**  
    **Foreach**  $q : q \in \text{correct} \wedge q \neq \text{myself} \wedge q \notin \text{acks}[mid]$  **do:**  
        **Trigger** **pp2pSend** ( $q, \{\text{BCAST}, mid, m, acks\}$ )

- I can do even more aggressive piggybacking by sending, on all messages that I broadcast, the map of acknowledgments that I have gathered until that point.
- This forces processes to add themselves to the acknowledgement set whenever they deliver a message.

# Uniform Reliable Broadcast

- Many algorithms have a Uniform Variant.
- The key difference is that Properties also include *something* about imposing Safety conditions regarding processes that have failed.

# Uniform Reliable Broadcast

- Uniform Reliable Broadcast:
  - URB1 (Validity): If a correct process  $i$  broadcasts message  $m$ , then  $i$  eventually delivers  $m$ .
  - URB2 (No Duplications): No message is delivered more than once.
  - URB3 (No Creation): If a correct process  $j$  delivers a message  $m$ , then  $m$  was broadcast to  $j$  by some process  $i$ .
  - URB4 (Agreement): If a message  $m$  is delivered by some process  $i$ , then  $m$  is eventually delivered by every correct process  $j$ .

# Uniform Reliable Broadcast

- Uniform Reliable Broadcast:
  - URB1 (Validity): If a correct process  $i$  broadcasts message  $m$ , then  $i$  eventually delivers  $m$ .
  - URB2 (No Duplications): No message is delivered more than once.
  - URB3 (No Creation): If a correct process  $j$  delivers a message  $m$ , then  $m$  was broadcast to  $j$  by some process  $i$ .
  - **URB4 (Agreement): If a message  $m$  is delivered by some process  $i$ , then  $m$  is eventually delivered by every correct process  $j$ .**



# Uniform Reliable Broadcast

- In general, to solve these problems you must further assume a maximum number of processes that can crash ( $f < N$ ).
- In this case, before you deliver a message, more than  $f$  processes must have a copy of the message.

# Uniform Reliable Broadcast

- In general, to solve these problems you must further assume a maximum number of processes that can crash ( $f < N$ ).
- In this case, before you deliver a message, more than  $f$  processes must have a copy of the message.
- Why is that?

# Uniform Reliable Broadcast

- In general, to solve these problems you must further assume a maximum number of processes that can crash ( $f < N$ ).
- In this case, before you deliver a message, more than  $f$  processes must have a copy of the message.
- Why is that?

To make sure that at least one process will survive that has a copy of the message and there is a way for all other correct processes to eventually deliver it.

# Back to (Regular) Broadcast...

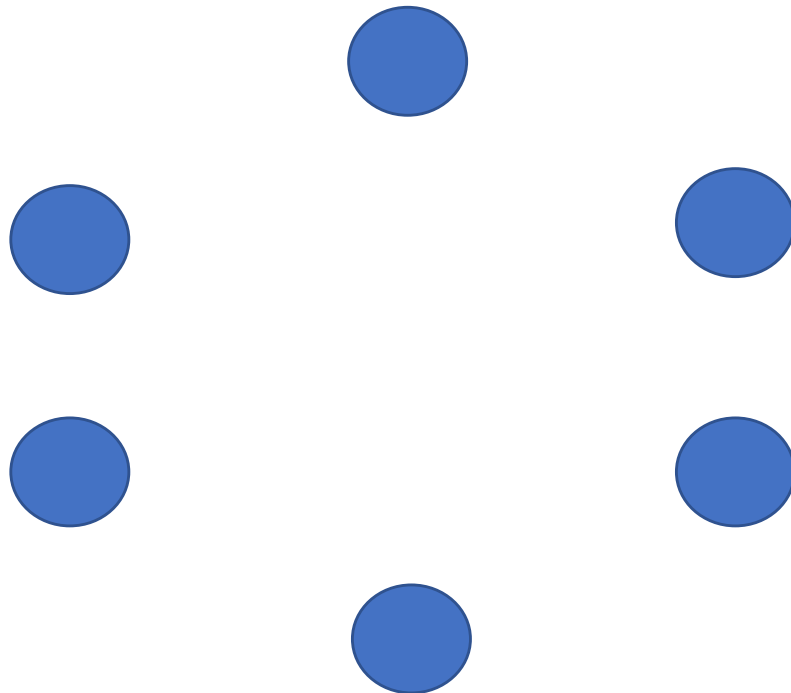
- Are you happy with our broadcast solutions??

# Back to (Regular) Broadcast...

- Are you happy with our broadcast solutions??
- Let's revise how this works in general...

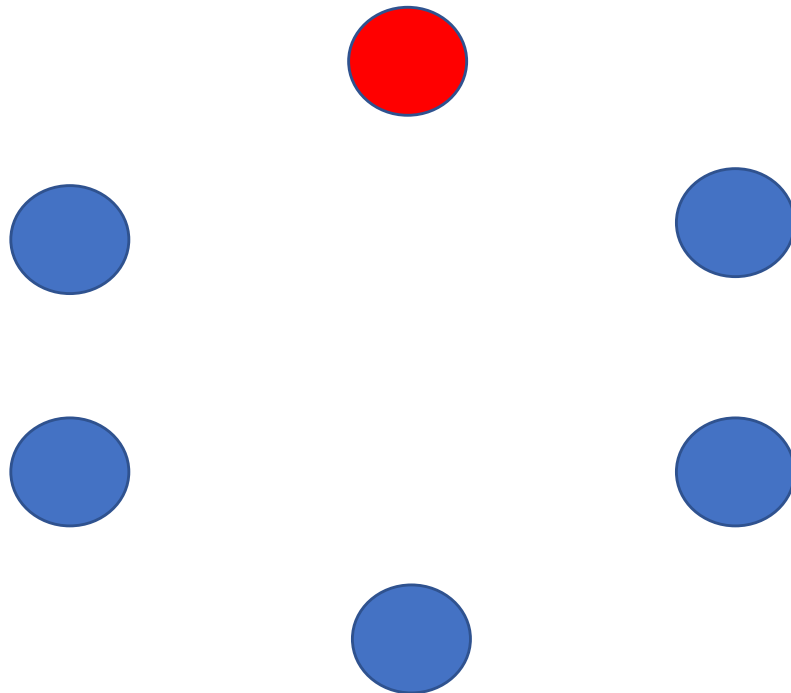
# Back to (Regular) Broadcast...

- Are you happy with our broadcast solutions??
- Let's revise how this works in general...



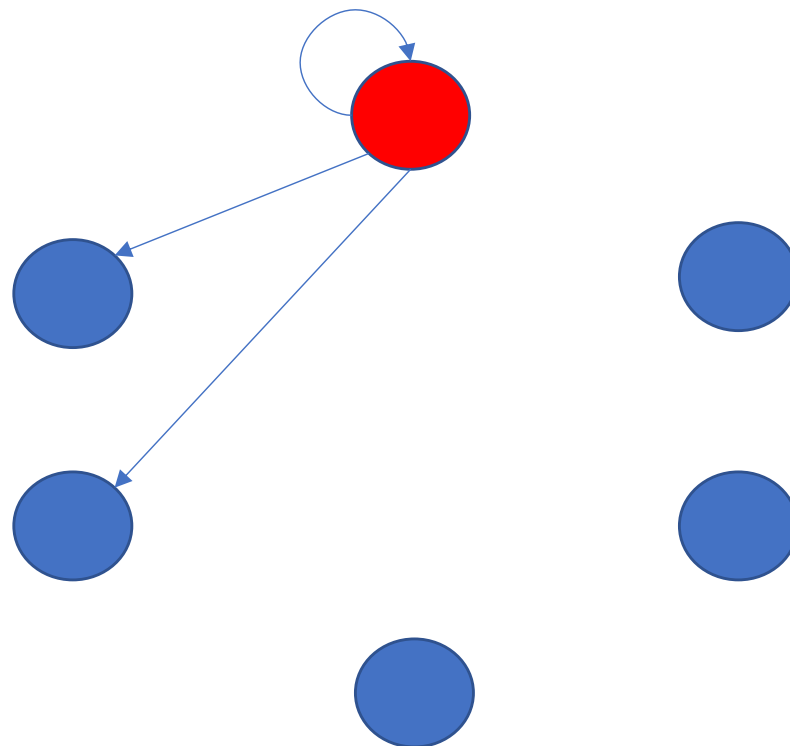
# Back to (Regular) Broadcast...

- Are you happy with our broadcast solutions??
- Let's revise how this works in general...



# Back to (Regular) Broadcast...

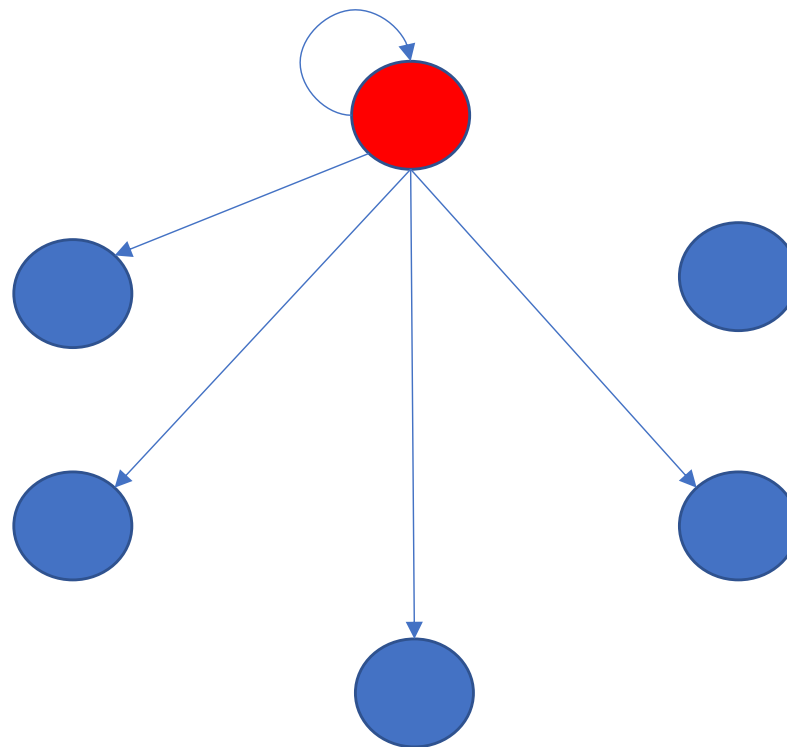
- Are you happy with our broadcast solutions??
- Let's revise how this works in general...





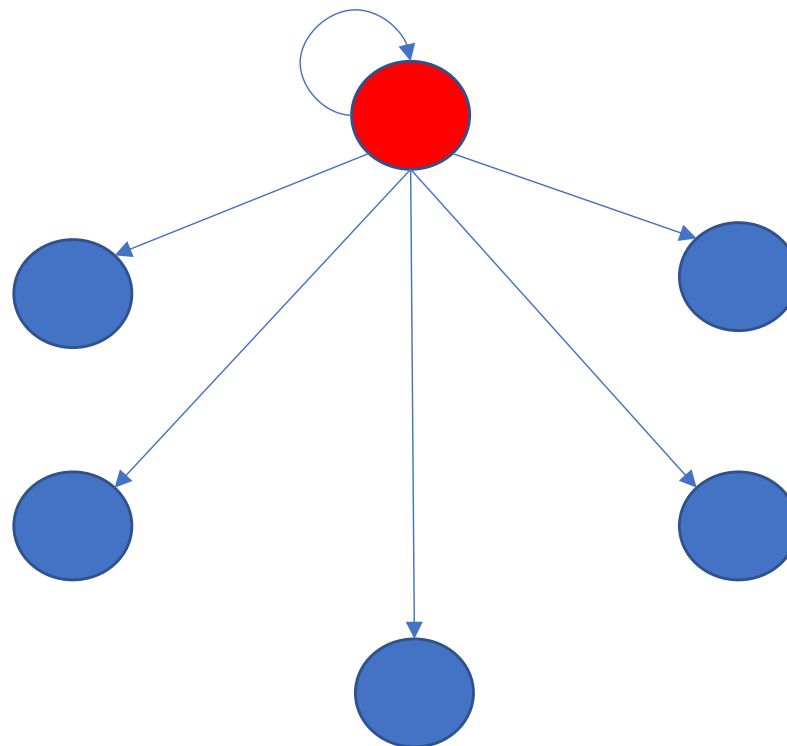
# Back to (Regular) Broadcast...

- Are you happy with our broadcast solutions??
- Let's revise how this works in general...



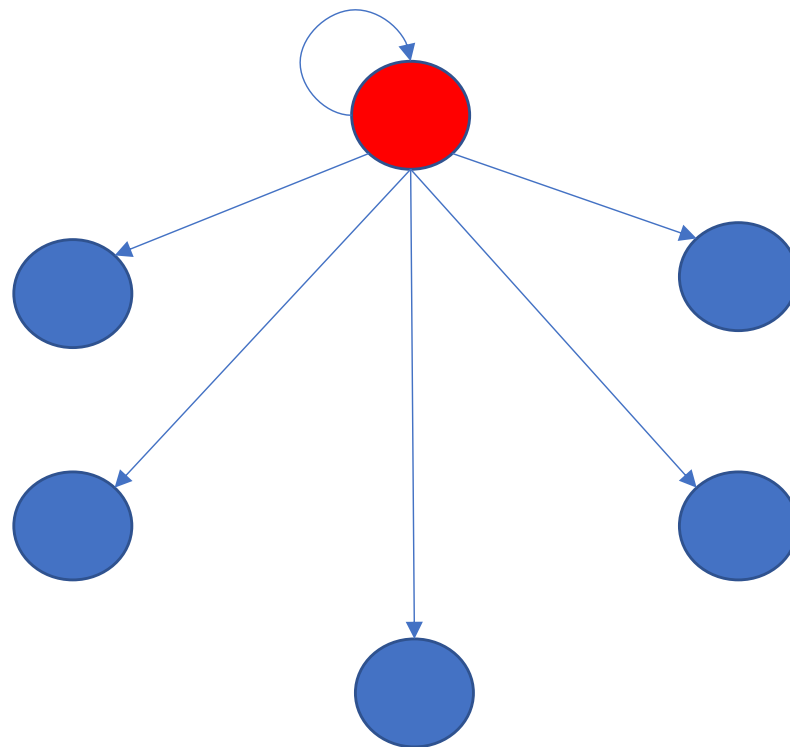
# Back to (Regular) Broadcast...

- Are you happy with our broadcast solutions??
- Let's revise how this works in general...



# Back to (Regular) Broadcast...

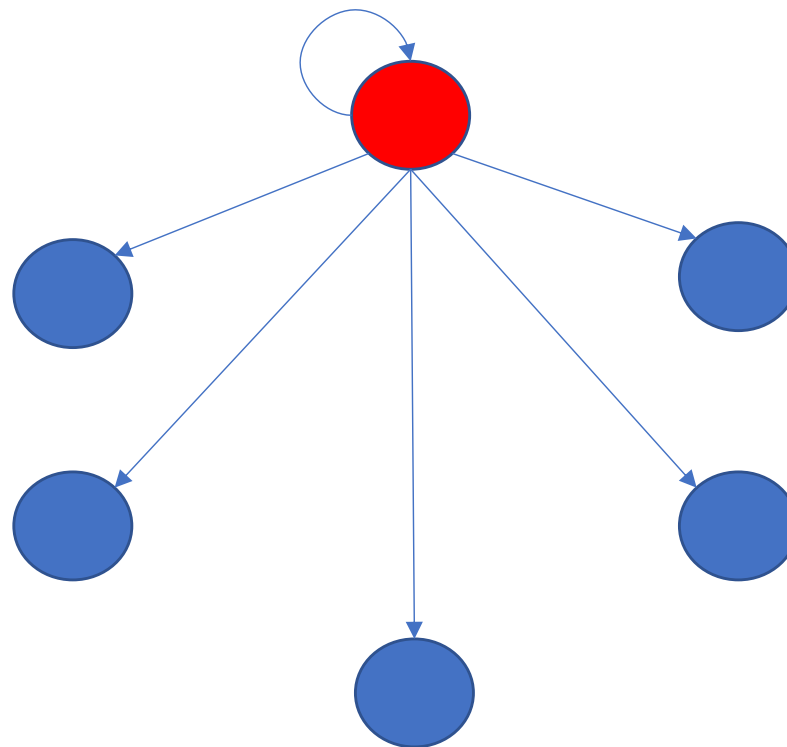
- Are you happy with our broadcast solutions??
- Let's revise how this works in general...



Any  
problem  
here?

# Back to (Regular) Broadcast...

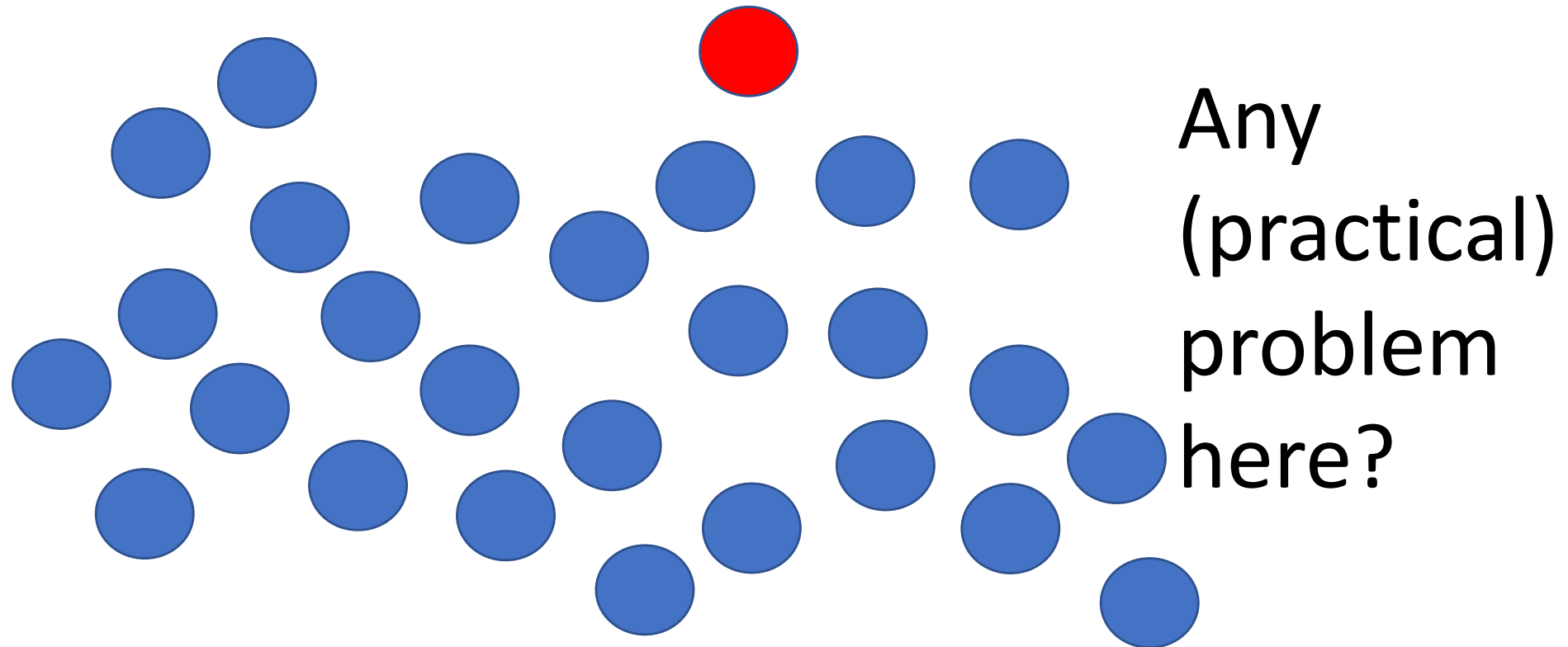
- Are you happy with our broadcast solutions??
- Let's revise how this works in general...



Any  
(practical)  
problem  
here?

# Back to (Regular) Broadcast...

- Are you happy with our broadcast solutions??
- Let's revise how this works in general...



Back to (Re)search broadcast.

Are you happy with our broadcast solution?

- Let's visit which is going

Any  
(practical)  
problem  
here.

# Reliable Broadcast...

- Solutions that we have seen are Reliable which is nice...
- ...but they put a lot of effort (load) on the sender.

# Reliable Broadcast...

- Solutions that we have seen are Reliable which is nice...
- ...but they put a lot of effort (load) on the sender.
- The load of the protocol is not balanced. One node has all the effort, the others not so much (in fault-free runs), and the effort grows with the size of the system.



# More Broadcast...

- So how can we address this issue?

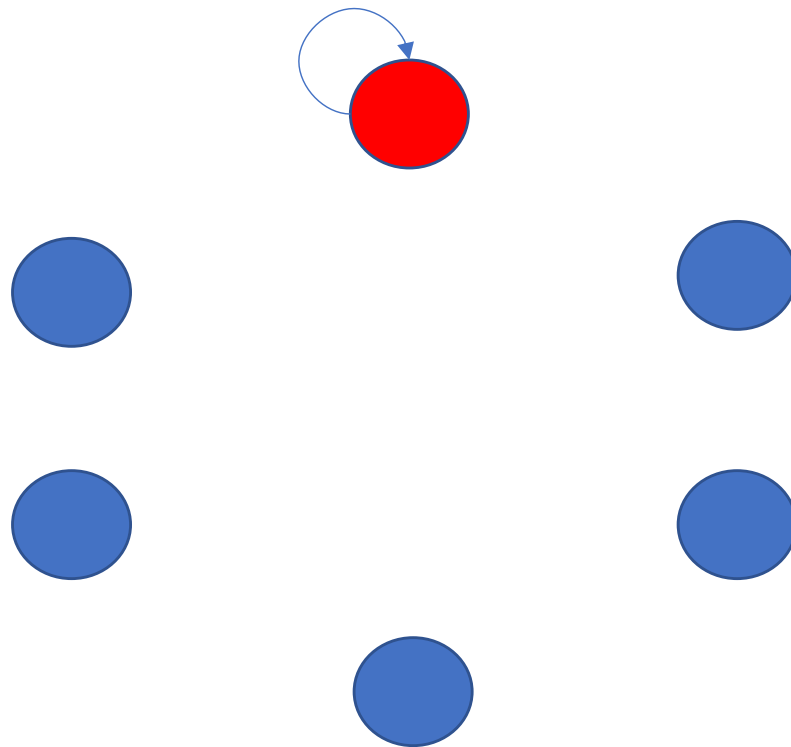
# More Broadcast...

- So how can we address this issue?
- A distributed system is composed by a set of **processes** that are interconnected through some **network** where processes seek to achieve some form of **cooperation** to execute tasks.

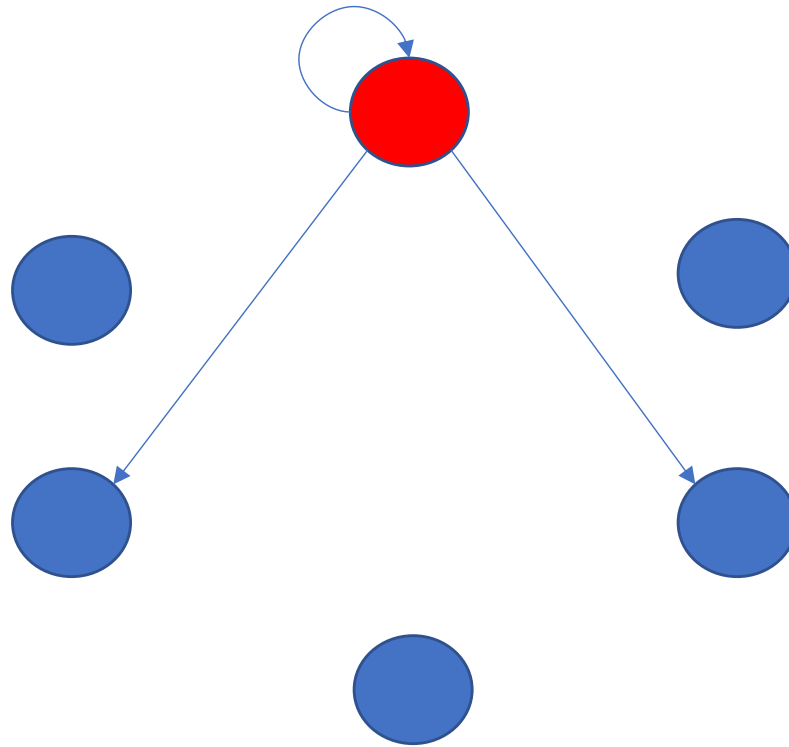
# More Broadcast...

- So how can we address this issue?
- A distributed system is composed by a set of **processes** that are interconnected through some **network** where processes seek to achieve some form of **cooperation** to execute tasks.
- Cooperation and sharing the load across the nodes:  
Welcome to the amazing World of Gossip

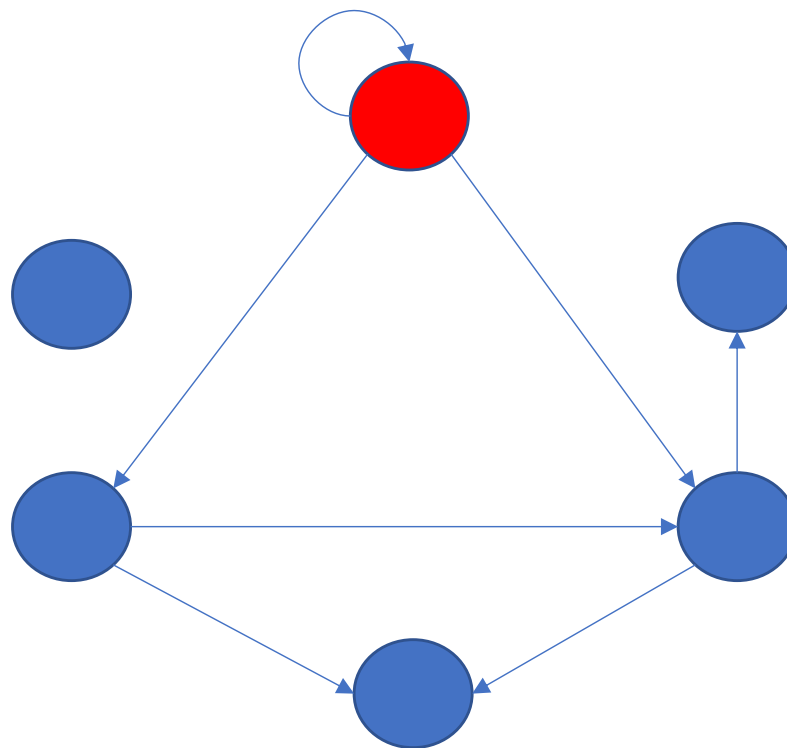
# Gossip in a nutshell...



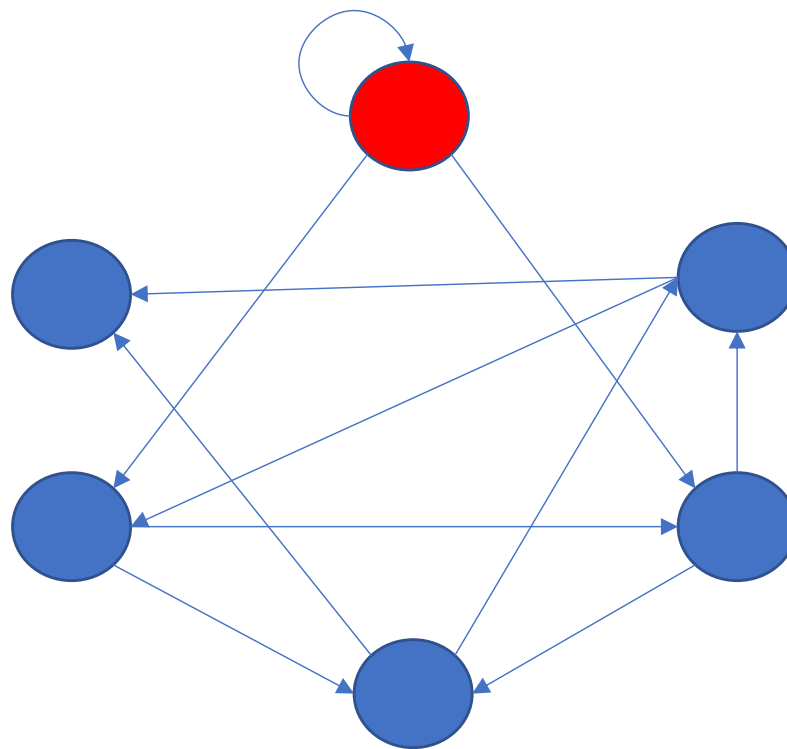
# Gossip in a nutshell...



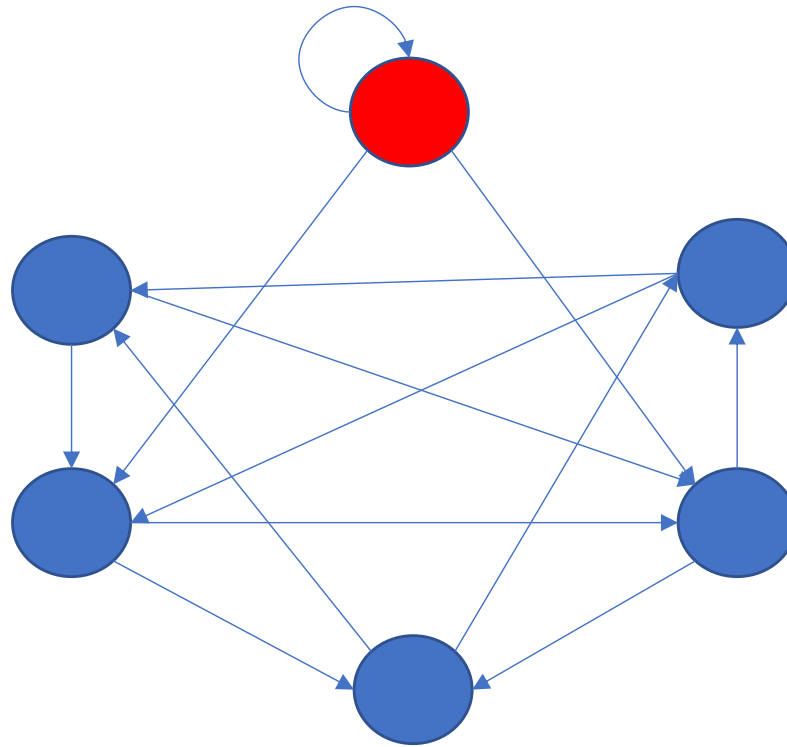
# Gossip in a nutshell...



# Gossip in a nutshell...

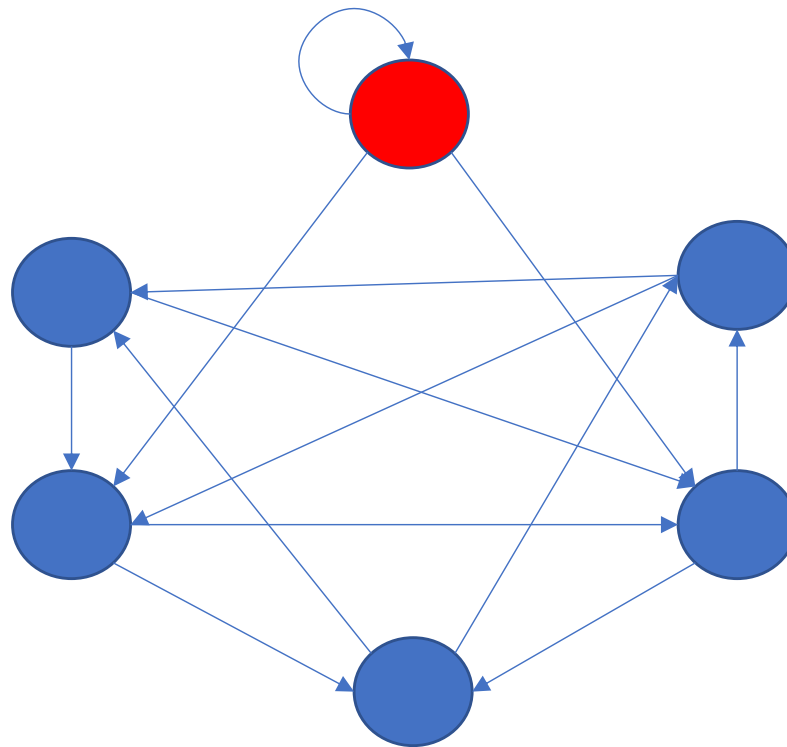


# Gossip in a nutshell...





# Gossip in a nutshell...



We do have some redundant messages...  
but everyone got the broadcasted  
message, so we are happy.

# Why do we call this Gossip:

- Because the algorithm mimics the way a gossip (*rumor*) spreads across a population.



# Why do we call this Gossip:

- Because the algorithm mimics the way a gossip (*rumor*) spreads across a population.



In fact, this also mimics the way diseases spread throughout a population, hence these algorithms/protocols are sometimes called **Epidemic**.

# Why do we call this Gossip:

- Because the algorithm mimics the way a gossip (*rumor*) spreads across a population.



In fact, this also mimics the way diseases spread throughout a population, hence these algorithms/protocols are sometimes called **Epidemic.**

# A simple gossip algorithm:

- When a process  $p$  wants to broadcast a message  $m$  it picks  $t$  other processes from the system.

# A simple gossip algorithm:

- When a process  $p$  wants to broadcast a message  $m$  it picks  $t$  other processes from the system.
- These  $t$  processes are selected **uniformly at random**.
- Process  $p$  then sends  $m$  to these processes.
- When a process receives a message for the first time, it simply repeats this process (eventually, avoiding to send the message back to the sender).

# A simple gossip algorithm:

- How many *gossip targets* should a process pick?
- *More formally*: How do we configure the parameter  $t$ ?

# A simple a gossip algorithm:

- How many *gossip targets* should a process pick?
- *More formally*: How do we configure the parameter  $t$ ?
- The theory of epidemics (this is a real thing) provides an answer to this...
- To ensure high probability that everyone receives the message:  $t \geq \ln(\pi)$
- ( $t$  is usually named the ***fanout*** of the algorithm).



# A simple gossip algorithm:

- Notice something weird here?  
Something that might not match with the specification of Reliable Broadcast?
- The theory of epidemics (this is a real thing) provides an answer to this...
- To ensure high probability that everyone receives the message:  $t \geq \ln(\pi)$
- ( $t$  is usually named the ***fanout*** of the algorithm).

# A simple gossip algorithm:

- Notice something weird here?  
Something that might not match with the specification of Reliable Broadcast?
- The theory of epidemics (this is a real thing) provides an answer to this...
- To ensure **high probability** that everyone receives the message:  $t \geq \ln(\pi)$
- ( $t$  is usually named the *fanout* of the algorithm).

# Probabilistic Reliable Broadcast:

- Probabilistic Reliable Broadcast:
  - PRB1 (Validity): If a correct process  $i$  broadcasts message  $m$ , then  $i$  eventually delivers the message.
  - PRB2 (No Duplications): No message is delivered more than once.
  - PRB3 (No Creation): If a correct process  $j$  delivers a message  $m$ , then  $m$  was broadcast to  $j$  by some process  $i$ .
  - PRB4 (Agreement): If a message  $m$  is delivered by some correct process  $i$ , then  $m$  is eventually delivered by every correct process  $j$  **with a (configurable) high probability.**

---

**Algorithm 1:** Probabilistic Reliable Broadcast (A.K.A. Epidemic Broadcast)

---

**Interface:****Requests:****pBroadcast** (  $m$  )**Indications:****pBcastDeliver** (  $m$  )**State:** $t$  // fanout of the protocol

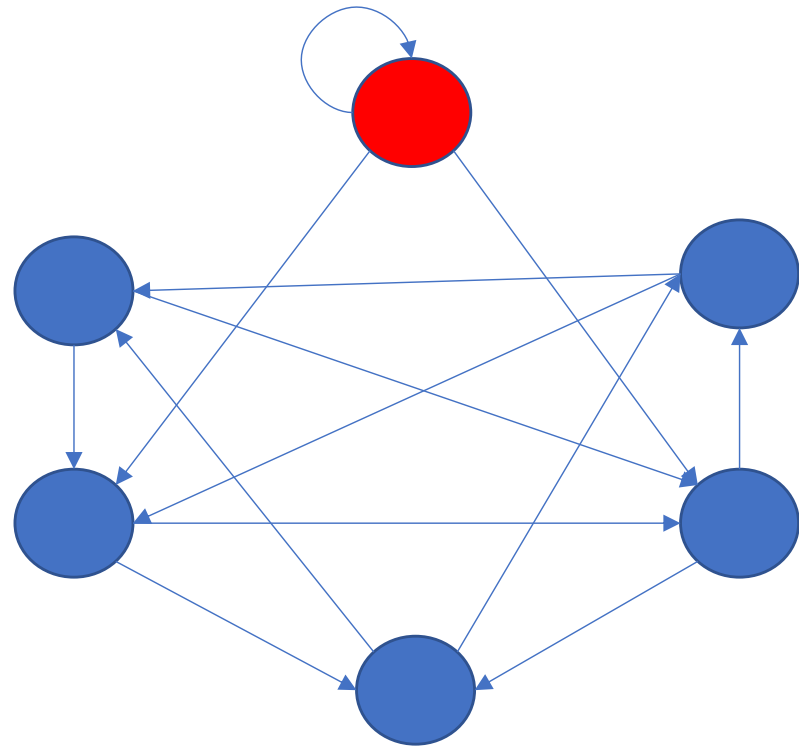
delivered // Ids of messages already delivered

**Upon Init () do:** $t \leftarrow \ln(\Pi);$ delivered  $\leftarrow \{\};$ **Upon pBroadcast(  $m$  ) do:** $mid \leftarrow \text{generateUniqueID}(m);$ **Trigger pBcastDeliver** (  $m$  );delivered  $\leftarrow \text{delivered} \cup \{mid\};$ gossipTargets  $\leftarrow \text{randomSelection}(t, \Pi);$ **Foreach  $p \in \text{gossipTargets}$  do:****Trigger Send( GossipMessage,  $p$ ,  $mid$ ,  $m$  );****Upon Receive (GossipMessage,  $s$ ,  $mid$ ,  $m$  ) do:****If  $mid \notin \text{delivered}$  do:**delivered  $\leftarrow \text{delivered} \cup \{mid\};$ **Trigger pBcastDeliver** (  $m$  );gossipTargets  $\leftarrow \text{randomSelection}(t, (\Pi \setminus s));$ **Foreach  $p \in \text{gossipTargets}$  do:****Trigger Send( GossipMessage,  $p$ ,  $mid$ ,  $m$  );**

---

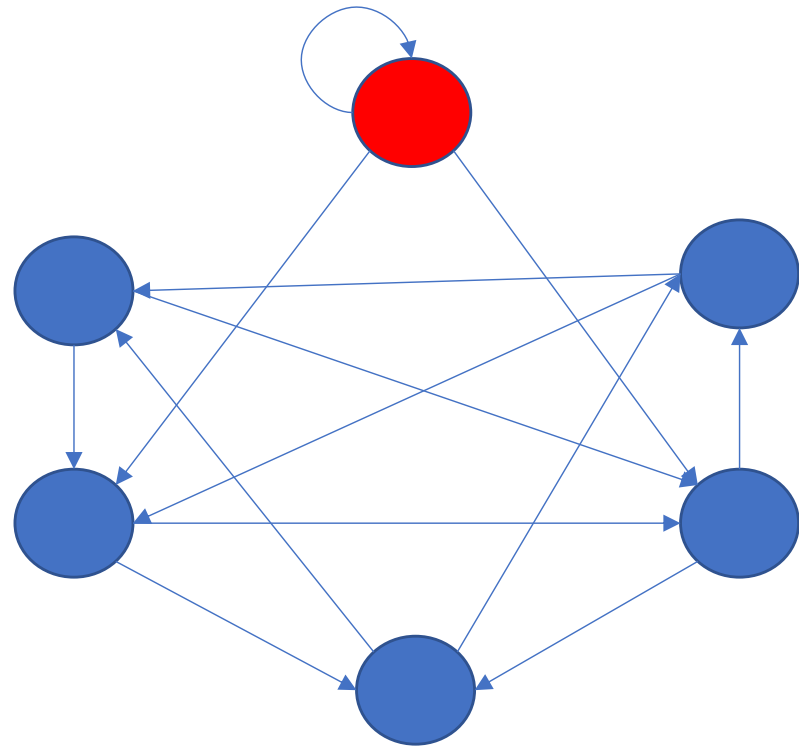
# Gossip Redundancy (the beauty and the beast)

- Notice that there is redundancy here:
  - Which is good, because we are operating on top of fair loss links.
  - On average, each process will receive the message  $t$  times (from different processes).
  - Total cost of messages:  $\# \pi \times t$



# Gossip Redundancy (the beauty and the beast)

- Notice that there is redundancy here:
  - Which is good, because we are operating on top of fair loss links.
  - On average, each process will receive the message  $t$  times (from different processes).
  - Total cost of messages:  $\# \pi \times t$
  - *One day we will tackle this...*



# More practical aspects:



- If a node has a new message, it sends the message to  $t$  other nodes...
- *Is there some (practical) problem here?*

# More practical aspects:



- If a node has a new message, it sends the message to  $t$  other nodes...
- *Is there some (practical) problem here?*
- *What if the message is really big?*



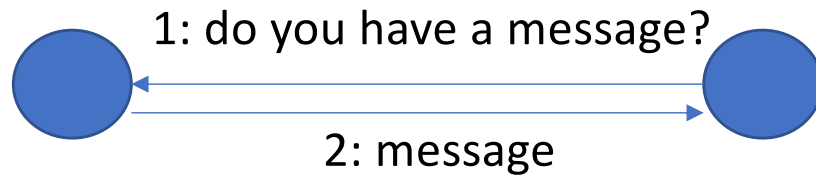
# More ways to gossip:

Sender

Receiver



(Eager) Push Gossip



Pull Gossip

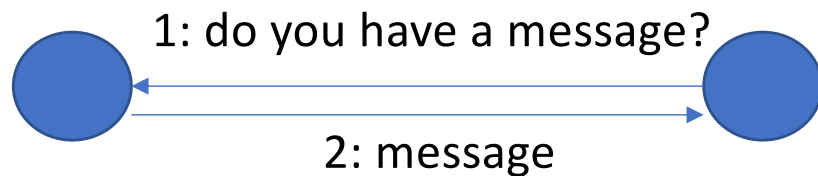
# More ways to gossip:

Sender

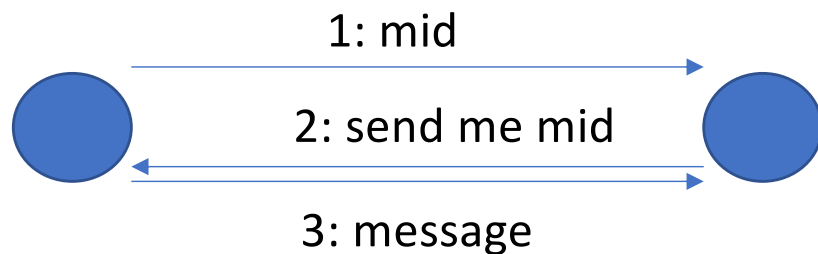
Receiver



(Eager) Push Gossip



Pull Gossip



Lazy Push Gossip

# More ways to gossip:

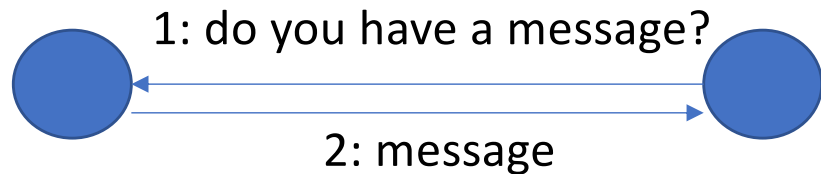
Sender

Receiver



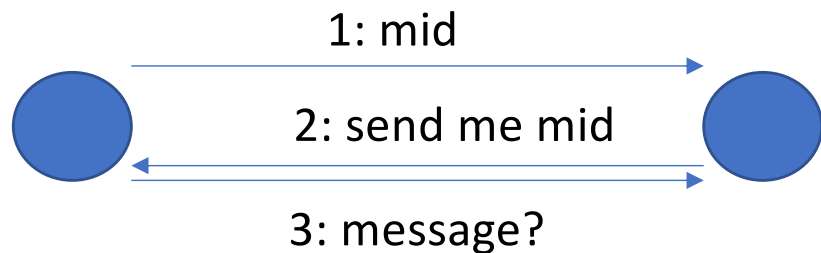
(Eager) Push Gossip

- Fast
- If messages are big -> expensive



Pull Gossip

- If messages are big -> less network traffic
- Slow
- If messages are small more traffic

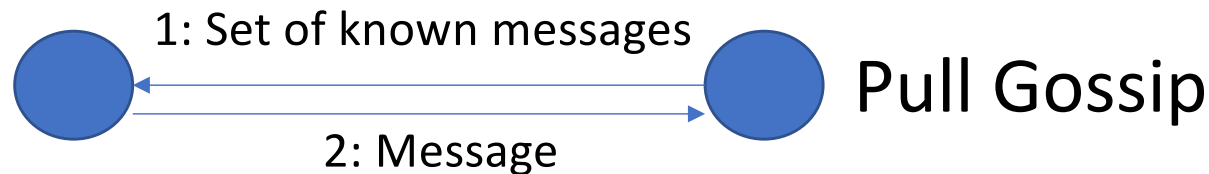


Lazy Push Gossip

- Faster than pull
- More communication steps.
- If messages are small more traffic

# More practical aspects:

- Sometimes, a system does not want to immediately send a message to their peers
  - Batching messages can be more efficient
- In this case, nodes might have multiple message to exchange. How to minimize duplicates?



- It is also possible to do this two-way to synchronize the known messages of the two peers.

# Efficiency

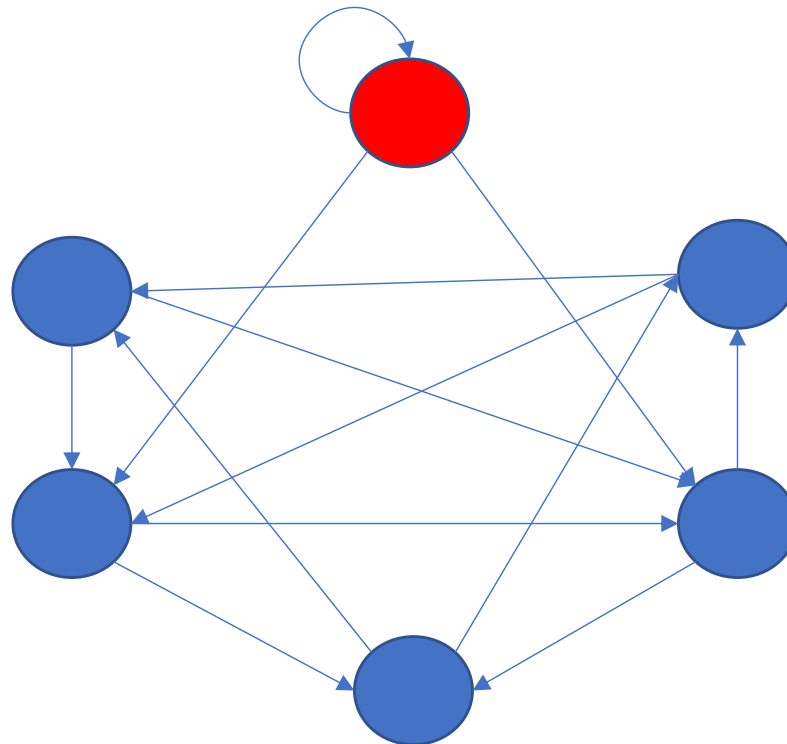
- Sending a set of message ids is inefficient. How to solve it?
- Idea:
  - Assign message id as (process id, counter)
  - Use vector clocks to summarize messages received – entry  $p_i$  has the largest counter for messages originated in  $p_i$

# Efficiency

- If there are many nodes, sending a version vector might be inefficient. How to solve it?
- Idea:
  - Nodes could store the vectors received the last time they communicated and send only the difference in consecutive sessions

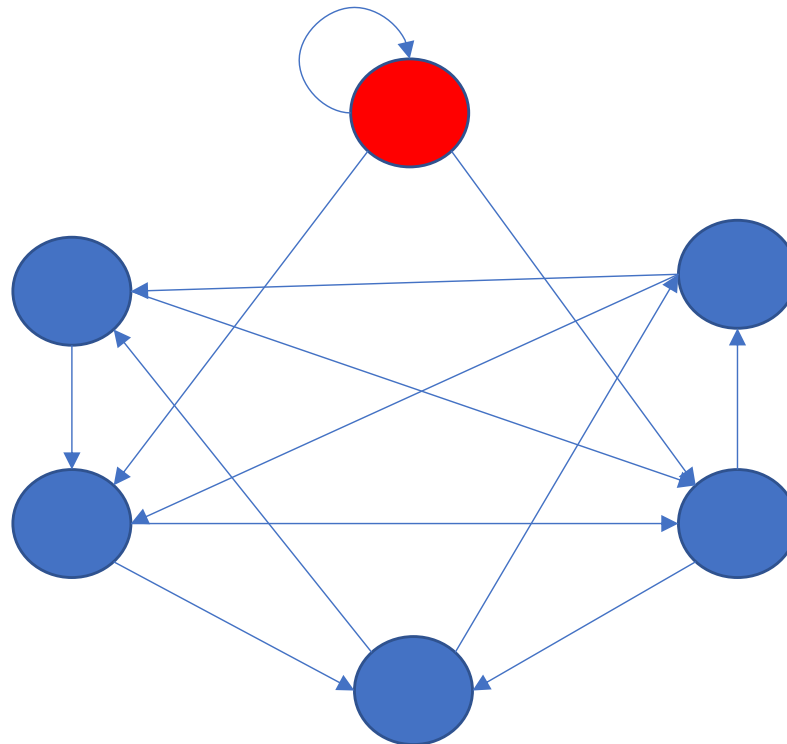
# More practical aspects:

- There might be another hidden problem here that will limit scalability...



# More practical aspects:

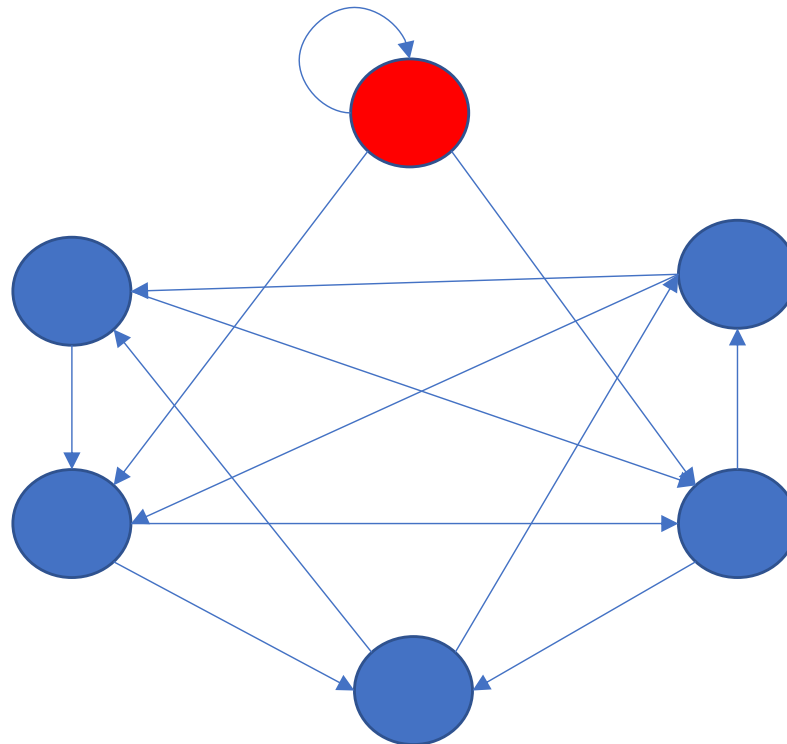
- There might be another hidden problem here that will limit scalability...
- This assumes a **global known membership**.





# More practical aspects:

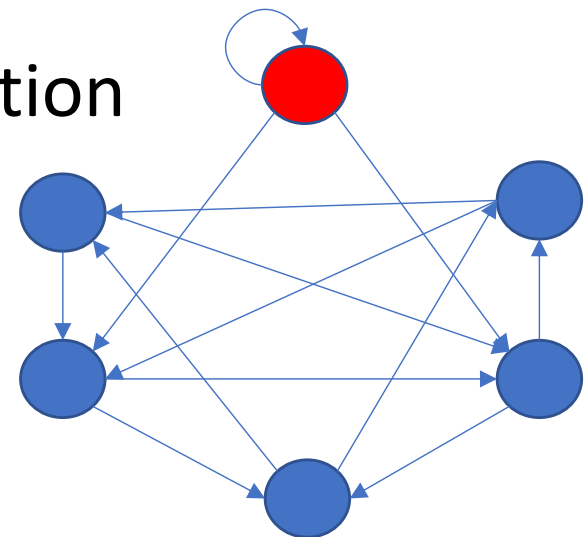
- There might be another hidden problem here that will limit scalability...
- This assumes a **global known membership**.



More formally,  
each process  $p$   
has to know  $\pi$ .

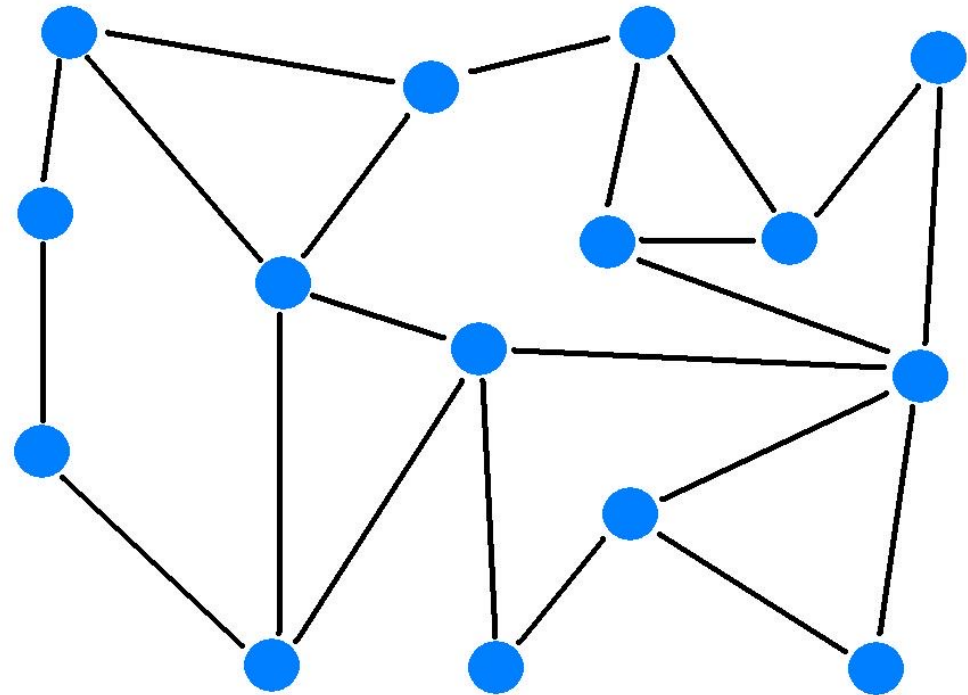
# Why avoid a global known membership.

- In a very large system  $\pi$  is not static.
- New processes might be added (for instance to deal with additional load i.e., achieve elasticity).
- Processes might have to leave, either due to failures (they will happen) or because they are no longer needed.
- The cost to keep all of this information up-to-date might be too expensive.



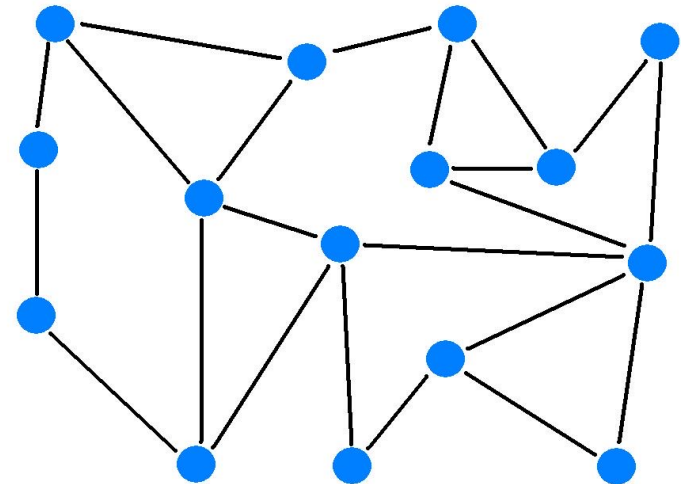
# Partial View Membership System

- Each process in the system knows a (few) other processes in the system.
- This will generate a (virtual) network on top of the physical network (which can also be modeled as a Graph  $G=(V,E)$ ).
- We call this an **overlay network**.



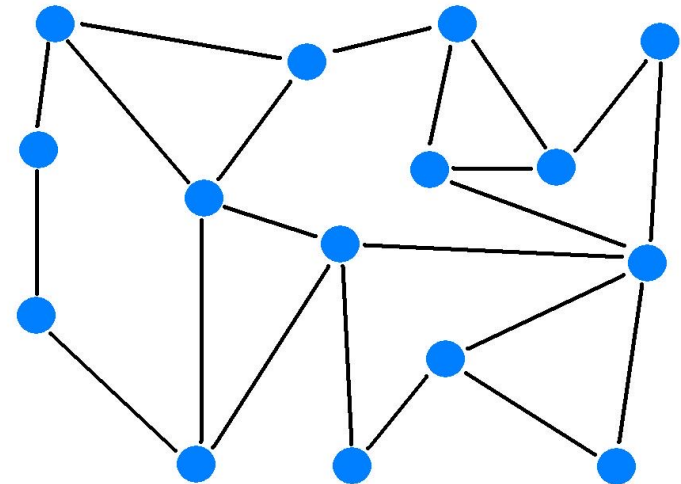
# Overlay Network

- Nodes define (application level/logical) neighboring relationships (materialized by links).
- Correctness:
  - **Connectivity**: There must be a path connecting any correct process  $p$  to every other correct process  $v$ .
  - **Accuracy**: Eventually, no correct process  $p$  will have an overlay link to a failed process.



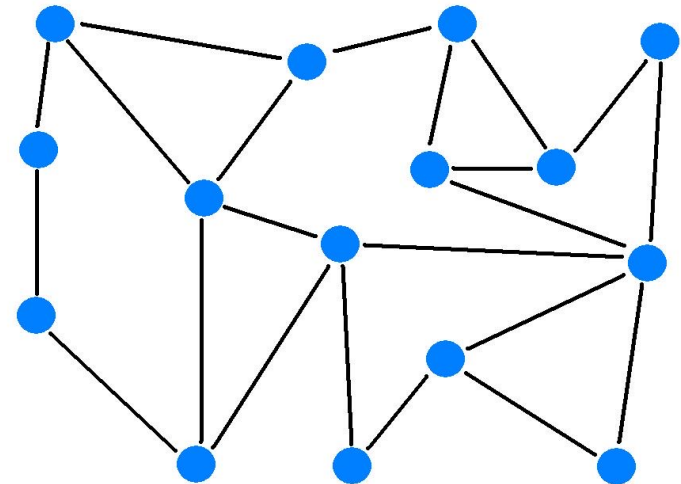
# Overlay Network

- Nodes define (application level/logical) neighboring relationships (materialized by links).
- Efficiency:
  - **Low diameter:** paths between correct processes should be small (measured by the average shortest path).
  - **Low clustering:** the neighbors of each process should be as different as possible from the neighbors of each of its neighbors.
  - **Uniform degree:** All processes should have a similar number of neighbors (degree).



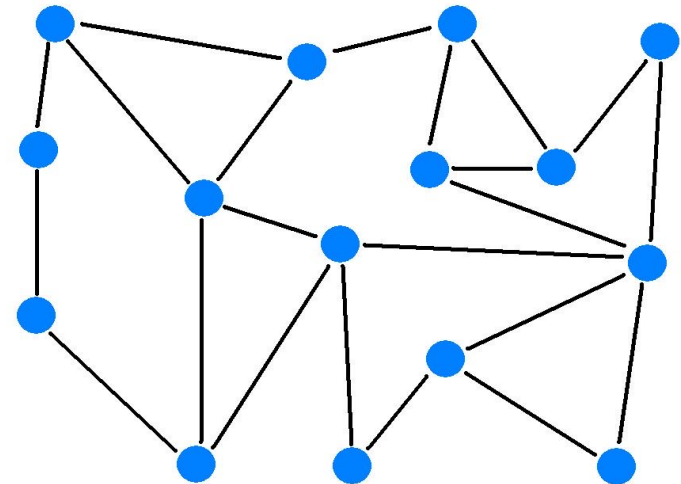
# Random Overlay Network

- Nodes define (application level/logical) **random** neighboring relationships (materialized by links).
- Efficiency:
  - **Low diameter:** paths between correct processes should be small (measured by the average shortest path).
  - **Low clustering:** the neighbors of each process should be as different as possible from the neighbors of each of its neighbors.
  - **Uniform degree:** All processes should have a similar number of neighbors (degree).



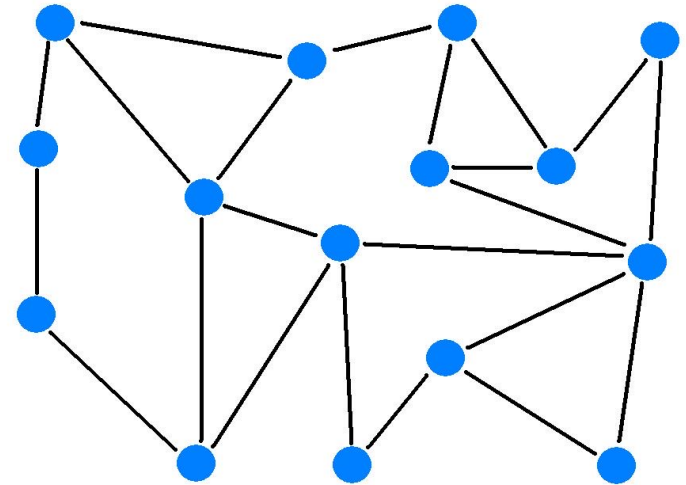
# Unstructured Overlay Network

- Nodes define (application level/logical) **random** neighboring relationships (materialized by links).
- Efficiency:
  - **Low diameter:** paths between correct processes should be small (measured by the average shortest path).
  - **Low clustering:** the neighbors of each process should be as different as possible from the neighbors of each of its neighbors.
  - **Uniform degree:** All processes should have a similar number of neighbors (degree).



# Unstructured Overlay Network

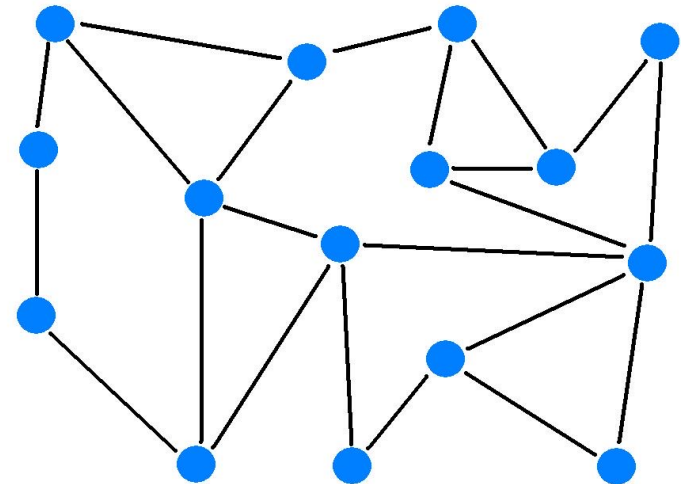
- How do you execute a Gossip algorithm on top of this membership abstraction?





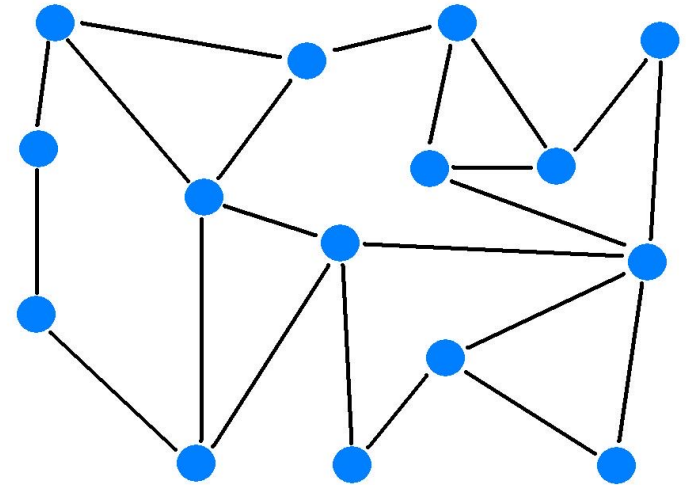
# Unstructured Overlay Network

- How do you execute a Gossip algorithm on top of this membership abstraction?
- The algorithm that maintains the overlay exposes a request whose indication lists the overlay neighbors of the local process .
- Instead of picking up  $t$  random processes out of  $\pi$  you pick  $t$  random processes out of your local logical neighbors.



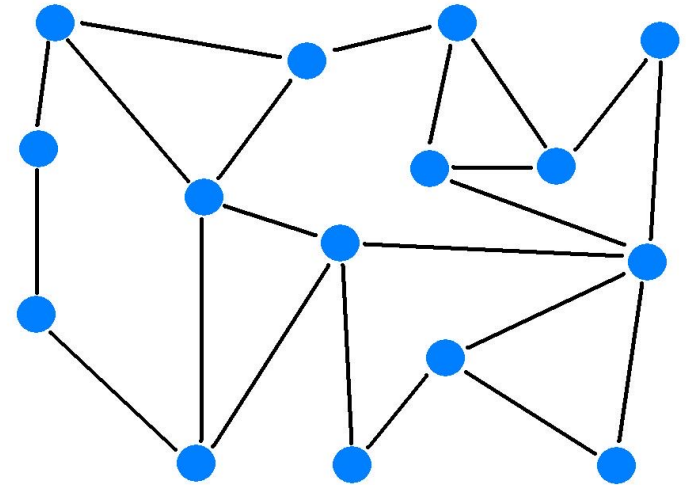
# Unstructured Overlay Network

- How do you build and maintain one of these?



# Unstructured Overlay Network

- How do you build and maintain one of these?
- The answer should be pretty obvious: Gossip



# Unstructured Overlay Network

- A Case study:

CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays.

Voulgaris, S., Gavidia, D. & van Steen, M. Journal Networked Systems Management (2005) 13: 197. <https://doi.org/10.1007/s10922-005-4441-x>

*Journal of Network and Systems Management, Vol. 13, No. 2, June 2005 (© 2005)*  
DOI: 10.1007/s10922-005-4441-x

**CYCLON: Inexpensive Membership Management  
for Unstructured P2P Overlays**

**Spyros Voulgaris,<sup>1,2</sup> Daniela Gavidia,<sup>1</sup> and Maarten van Steen<sup>1</sup>**

# Cyclon Intuition:

- When a new process joins it must know the identifier of another process already in the system (contact).
- Process identifiers are enriched with a counter (age) that state how long ago the identifier was created.
- Periodically, each process picks and removes a process identifier that is among the oldest of its partial view and sends a sample of its neighbors alongside a new identifier for itself (age = zero).
- The other side replies with a sample of its own neighbors.
- Both process integrate the information received from their peer, removing identifiers that it sent to the peer (and random ones if required).

**Requests:**  
     **getNeighbors ( )**

**Indications:**  
     **neighbors (n)** // n is the set of neighbors of the local process

```
neigh//set of neighbors of the process (local partial view)
N//maximum number of neighbors
sample//Sample of neigh sent to the other process in last shuffle
```

```

N ← maxN;
If contact ≠ ⊥ then
    neigh ← {(contact, 0)};
else
    neigh ← {};
sample ← ⊥;
Setup Periodic Timer Shuffle (T); //T is the shuffle period, in the order of seconds

```

```
pview ← neigh; //To avoid the upper layer to modify the neigh set
Trigger neighbors( pview );
```

```

foreach  $(p, age) \in \text{neigh}$  do
   $\text{neigh} \leftarrow (\text{neigh} \setminus (p, age)) \cup (p, age + 1);$ 
 $p \leftarrow \text{pickOldest}(\text{neigh});$ 
If  $p \neq \perp$  then
   $\text{neigh} \leftarrow \text{neigh} \setminus p;$ 
   $\text{sample} \leftarrow \text{randomSubset}(\text{neigh});$ 
  Trigger Send (ShuffleRequest,  $p, \text{sample} \cup \{(myself, 0)\}$ );

```

```
temporarySample ← randomSubset(neigh);
Trigger Send (ShuffleReply, s, temporarySample);
Call mergeViews(peerSample, temporarySample);
```

Call `mergeViews(peerSample, sample);`

```

Foreach  $(p, age) \in \text{peerSample}$  do
  If  $(p', age') \in \text{neigh} \wedge p' = p$  then
    If  $age' > age$  then
       $\text{neigh} \leftarrow (\text{neigh} \setminus (p', age')) \cup \{(p, age)\};$ 
    Else If  $\#\text{neigh} < N$  then
       $\text{neigh} \leftarrow \text{neigh} \cup \{(p, age)\};$ 
    Else
       $(x, age') \leftarrow (x, age') : (x, age') \in \text{neigh} \wedge (x, age'') \in \text{mySample};$  // Pick an element of
                                                                    neigh that is also in mySample
      If  $(x, age') = \perp$  then
         $(x, age') \leftarrow (x, age') : (x, age') \in \text{neigh};$  // Pick a random element of neigh
         $\text{neigh} \leftarrow (\text{neigh} \setminus (x, age')) \cup \{(p, age)\};$ 

```

(A copy of this algorithm can be found in clip under support documentation.)

**Interface:****Requests:**

**getNeighbors ( )**

**Indications:**

**neighbors (  $n$  )** //  $n$  is the set of neighbors of the local process

**State:**

**neigh** // set of neighbors of the process (local partial view)

**N** // maximum number of neighbors

**sample** // Sample of neigh sent to the other process in last shuffle

**Upon Init (  $contact$ ,  $maxN$  ) do:**

$N \leftarrow maxN$ ;

**If**  $contact \neq \perp$  **then**

$neigh \leftarrow \{(contact, 0)\}$ ;

**else**

$neigh \leftarrow \{\}$ ;

$sample \leftarrow \perp$ ;

**Setup Periodic Timer Shuffle (T)**; // T is the shuffle period, in the order of seconds

**Upon getNeighbors( ) do:**

$pview \leftarrow neigh$ ; // To avoid the upper layer to modify the neigh set

**Trigger neighbors( pview );**

**Upon Shuffle( ) do:**

```
foreach  $(p, age) \in \text{neigh}$  do
   $\text{neigh} \leftarrow (\text{neigh} \setminus (p, age)) \cup (p, age + 1);$ 
 $p \leftarrow \text{pickOldest}(\text{neigh});$ 
If  $p \neq \perp$  then
   $\text{neigh} \leftarrow \text{neigh} \setminus p;$ 
   $\text{sample} \leftarrow \text{randomSubset}(\text{neigh});$ 
  Trigger Send (ShuffleRequest,  $p$ ,  $\text{sample} \cup \{(myself, 0)\}$ );
```

**Upon Receive (ShuffleRequest,  $s$ ,  $peerSample$ ) do:**

```
temporarySample  $\leftarrow \text{randomSubset}(\text{neigh});$ 
Trigger Send (ShuffleReply,  $s$ , temporarySample);
Call mergeViews( $peerSample$ , temporarySample);
```

**Upon Receive (ShuffleReply,  $s$ ,  $peerSample$ ) do:**

```
Call mergeViews( $peerSample$ , sample);
```

**Procedure** mergeViews( $peerSample$ ,  $mySample$ )

```
Foreach  $(p, age) \in \text{peerSample}$  do
  If  $(p', age') \in \text{neigh} \wedge p' = p$  then
    If  $age' > age$  then
       $\text{neigh} \leftarrow (\text{neigh} \setminus (p', age')) \cup \{(p, age)\};$ 
    Else If  $\#\text{neigh} < N$  then
       $\text{neigh} \leftarrow \text{neigh} \cup \{(p, age)\};$ 
  Else
     $(x, age') \leftarrow (x, age') : (x, age') \in \text{neigh} \wedge (x, age'') \in mySample; // \text{Pick an element of}$   

 $\text{neigh}$  that is also in  $mySample$ 
    If  $(x, age') = \perp$  then
       $(x, age') \leftarrow (x, age') : (x, age') \in \text{neigh}; // \text{Pick a random element of } \text{neigh}$ 
     $\text{neigh} \leftarrow (\text{neigh} \setminus (x, age')) \cup \{(p, age)\};$ 
```



# Unstructured Overlay Network

- Another Case study (*check this at home*):
- Ganesh, A., Kermarrec, A.-M., & Massoulie, L. (2003, February). Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), 139 – 149.

IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 2, FEBRUARY 2003

139

## Peer-to-Peer Membership Management for Gossip-Based Protocols

Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié

**Abstract**—Gossip-based protocols for group communication have attractive scalability and reliability properties. The probabilistic gossip schemes studied so far typically assume that each group member has full knowledge of the global membership and chooses gossip targets uniformly at random. The requirement of global knowledge impairs their applicability to very large-scale groups. In this paper, we present SCAMP (Scalable Membership protocol), a novel peer-to-peer membership protocol which operates in a fully decentralized manner and provides each member with a partial view of the group membership. Our protocol is self-organizing in the sense that the size of partial views naturally converges to the value required to support a gossip algorithm reliably. This value is a function of the group size, but is achieved without any node knowing the group size. We propose additional mechanisms to achieve balanced view sizes even with highly unbalanced subscription patterns. We present the design, theoretical analysis, and a detailed evaluation of the basic protocol and its refinements. Simulation results show that the reliability guarantees provided by SCAMP are comparable to previous schemes based on global knowledge. The scale of the experiments attests to the scalability of the protocol.

**Index Terms**—Scalability, reliability, peer-to-peer, gossip-based probabilistic multicast, membership, group communication, random graphs.

# Going back to Gossip...

## (or 1001 ways to Gossip)

- Relevant aspects of algorithms:
  - Degree of the overlay (number of neighbors:  $N$ )
  - Fanout ( $t$ ).
  - Communication Mode:
    - Eager Push
    - Pull
    - Lazy Push
- **Epidemic Broadcast** (also known as Gossip):
  - $t \geq \ln(\pi)$  &  $t < N$
  - Eager push

# Going back to Gossip...

## (or 1001 ways to Gossip)

- Relevant aspects of algorithms:
  - Degree of the overlay (number of neighbors:  $N$ )
  - Fanout ( $t$ ).
  - Communication Mode:
    - Eager Push
    - Pull
    - Lazy Push
- **Flood:**
  - $t = N$
  - Eager push

# Going back to Gossip...

## (or 1001 ways to Gossip)

- Relevant aspects of algorithms:
  - Degree of the overlay (number of neighbors:  $N$ )
  - Fanout ( $t$ ).
  - Communication Mode:
    - Eager Push
    - Pull
    - Lazy Push
- **Anti-entropy:**
  - $t = 1$
  - Pull (Executed by everyone periodically)

# Going back to Gossip...

## (or 1001 ways to Gossip)

- Relevant aspects of algorithms:
  - Degree of the overlay (number of neighbors:  $N$ )
  - Fanout ( $t$ ).
  - Communication Mode:
    - Eager Push
    - Pull
    - Lazy Push
- **Timestamped anti-entropy:**
  - $t = 1$
  - Pull (Executed by everyone periodically), using timestamps and vector clocks to ask for new information

# Going back to Gossip...

## (or 1001 ways to Gossip)

- Relevant aspects of algorithms:
  - Degree of the overlay (number of neighbors:  $N$ )
  - Fanout ( $t$ ).
  - Communication Mode:
    - Eager Push
    - Pull
    - Lazy Push
- **Random-Walk** (useful to look for stuff in the overlay):
  - $t = 1$
  - Eager Push (usually with a maximum number of retransmissions aka “time to live” or “TTL”).

# Going back to Gossip... (or 1001 ways to Gossip)

- Relevant algorithms aspects:
  - Degree of the overlay (number of neighbors:  $N$ )
  - Fanout ( $t$ ).
  - Communication Mode:
    - Eager Push
    - Pull
    - Lazy Push
- **There are others... We will see some on another day.**
  - **$t = ?$**
  - **$??$**