Algorithms and Distributed Systems 2019/2020 (Lecture Two)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Leitão (jc.leitao@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Lecture structure:

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

Home Work 1:

- Write the Pseudo-Code for solving the Reliable Broadcast Problem assuming:
  - A Crash fault model and an asynchronous system.
  - Your solution **must** ensure all properties of the Reliable Broadcast Problem.

  - Remember that if a process crashes he is no longer correct, and hence messages sent by him can never be delivered (as long as no-one delivered those messages).

Home Work 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 (Aggrement): If a message m is delivered by some correct process i, them m is eventually delivered by every correct process j.

  - Remember that if a process crashes he is no longer correct, and hence messages sent by him can never be delivered (as long as no-one delivered those messages).

Building Block: Best Effort Broadcast

State: (could be omitted)

Upon Init do: (could be omitted)

Upon bebBroadcast(m) **do**
    forall p ∈ **π**
        trigger pp2pSend(p, m);

Upon pp2pDeliver (p , m) **do**
    trigger bebDeliver(p, m);

Reliable Broadcast Problem

State:

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

Upon Init **do**:

delivered ⟵ {};

Upon rbBroadcast(m) **do**

trigger rbDeliver(m);

mid ⟵ generateUniqueID(m);

delivered ⟵ delivered U {mid};

trigger bebBroadcast(mid, m);

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

if (mid ∉ delivered) **then**

delivered ⟵ delivered U {mid};

trigger rbDeliver(m);

trigger bebBroadcast(mid, m);

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 ⟵ {};

Upon rbBroadcast(m) **do**

trigger rbDeliver(m);

mid ⟵ generateUniqueID(m);

delivered ⟵ delivered U {mid};

trigger bebBroadcast(mid, m);

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

if (mid ∉ delivered) **then**

delivered ⟵ delivered U {mid};

trigger rbDeliver(m);

trigger bebBroadcast(mid, m);

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 U {mid};

trigger bebBroadcast(mid, m);

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

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

delivered $\leftarrow$ delivered U {mid};

trigger rbDeliver(m);

trigger bebBroadcast(mid, m);

Why is this solution correct?

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 $\longleftarrow$ {};

Upon rbBroadcast(m) **do**

trigger rbDeliver(m);

mid $\longleftarrow$ generateUniqueID(m);

delivered $\longleftarrow$ delivered U {mid};

trigger bebBroadcast(mid, m);

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

if (mid ∉ delivered) **then**

delivered $\longleftarrow$ delivered U {mid};

trigger rbDeliver(m);

trigger bebBroadcast(mid, m);

This do consume a lot of messages (not being nice for the network nor processors)!

HW1: Reliable Broadcast Problem

- 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 (Aggrement): If a message m is delivered by some correct process i, them m is eventually delivered by every correct process j.
- Solve it in the fail-stop failure model & synchronous system model

Algorithm 1: Reliable Broadcast (Fail-Stop / Synchronous) `I`

```
Interface:
    Requests:
        rBroadcast ( m )
    Indications:
        rBcastDeliver ( m )

State:
    delivered //Ids of messages already delivered
    messages //Map that associates to each process p the messages broadcasted by it

    Upon Init () do:
        delivered ⟵ {};
        messages ⟵ {};
        Foreach p ∈ Π do:
            messages[p] ⟵ {};

    Upon rBroadcast( m ) do:
        mid ⟵ generateUniqueID(m);
        Trigger rBcastDeliver (m);
        delivered ⟵ delivered ∪ {mid};
        Trigger bebBroadcast ( mid, m );

    Upon bebDeliver ( s, mid, m ) do:
        If mid ∉ delivered then
            delivered ⟵ delivered ∪ {mid};
            messages[s] ⟵ messages[s] ∪ {(mid, m)};
            Trigger rBcastDeliver (m);

    Upon Crash ( p ) do:
        Foreach (mid, m) ∈ messages[p] do:
            Trigger bebBroadcast (mid, m);
```

Algorithm 2: Reliable Broadcast (Fail-Stop / Synchronous) `II`

```
Interface:
    Requests:
        rBroadcast ( m )
    Indications:
        rBcastDeliver ( m )

State:
    round //Current round
    delivered //Ids of messages already delivered
    messages //Map that associates to each process p the messages broadcasted by it

    Upon Init () do:
        round ⟵ 0;
        delivered ⟵ {};
        messages ⟵ {};
        Foreach p ∈ Π do:
            messages[p] ⟵ {};
        Setup Periodic Timer NextRound (T); //T is the max time to deliver a message

    Upon rBroadcast( m ) do:
        mid ⟵ generateUniqueID(m);
        Trigger rBcastDeliver (m);
        delivered ⟵ delivered ∪ {mid};
        Trigger bebBroadcast ( mid, m );

    Upon bebDeliver ( s, mid, m ) do:
        If mid ∉ delivered then
            delivered ⟵ delivered ∪ {mid};
            messages[s] ⟵ messages[s] ∪ {(mid, m, round)};
            Trigger rBcastDeliver (m);

    Upon NextRound() do:
        round ⟵ round + 1;
        Foreach p ∈ Π do:
            Foreach (mid, m, r) ∈ messages[p] do:
                If (r < round − 1) then
                    messages[p] ⟵ messages[p] \ (mid, m, r);

    Upon Crash ( p ) do:
        Foreach (mid, m, r) ∈ messages[p] do:
            Trigger bebBroadcast (mid, m);
```

Algorithm 3: Reliable Broadcast (Fail-Stop / Synchronous) `III`

```
Interface:
    Requests:
        rBroadcast ( m )
    Indications:
        rBcastDeliver ( m )

State:
    delivered //Ids of messages already delivered
    messages //Map that associates to each process p the messages broadcasted by it

    Upon Init () do:
        round ⟵ 0;
        delivered ⟵ {};
        messages ⟵ {};
        Foreach p ∈ Π do:
            messages[p] ⟵ {};

    Upon rBroadcast( m ) do:
        mid ⟵ generateUniqueID(m);
        Trigger rBcastDeliver (m);
        delivered ⟵ delivered ∪ {mid};
        Trigger bebBroadcast ( mid, m );

    Upon bebDeliver ( s, mid, m ) do:
        If mid ∉ delivered then
            delivered ⟵ delivered ∪ {mid};
            messages[s] ⟵ messages[s] ∪ {(mid, m)};
            Trigger rBcastDeliver (m);
            Setup Timer StableMessage(T, s, mid, m);//T is the max time to deliver a message

    Upon StableMessage(p, mid, m) do:
        messages[p] ⟵ messages[p] \ (mid, m);

    Upon Crash ( p ) do:
        Foreach (mid, m, r) ∈ messages[p] do:
            Trigger bebBroadcast (mid, m);
```

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 the message.
  - 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 (Aggrement): If a message m is delivered by some process i, them 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 the message.
  - 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 (Aggrement): If a message m is delivered by some process i, them m is eventually delivered by every correct process j.**

Uniform Reliable Broadcast

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

Theory versus Practice

- An algorithm such as the ones we have seen can actually be implemented (and become a protocol).
- Obviously, when implementation is performed some aspects that to be taken into consideration:
  - You cannot expect to have infinite memory.
  - You cannot expect to have infinite bandwidth.
  - You should not have an absurdly high number of timers running.
- This is where the engineering comes into play.

Theory versus Practice

- For instance, when you consider the link abstractions that we have studied last week…
- ...are they practical?

Theory versus Practice

- For instance, when you consider the link abstractions that we have studied last week…
- ...are they practical?

- In fact TCP uses very similar techniques, but it solves the practical aspects by a combination of engineering strategies:
  - Infinite transmission of messages: Acks, transmitted in piggyback when possible.
  - Infinite memory: Through the use of the transmission and reception windows.
  - (This is more sophisticated than this...)

Back to Broadcast…

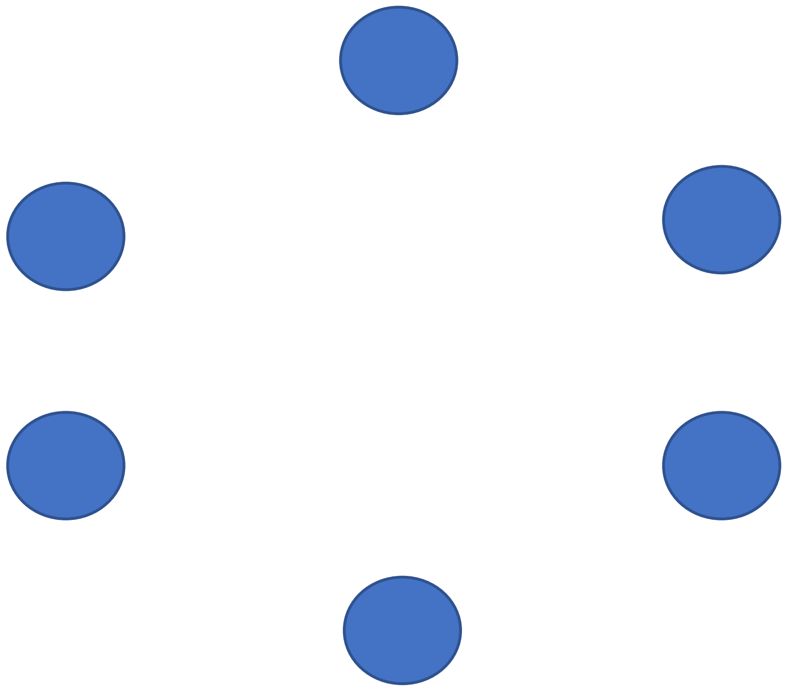
- Are you happy with our broadcast solutions??

Back to Broadcast…

- Are you happy with our broadcast solutions??
- Lets revise how this works in general…

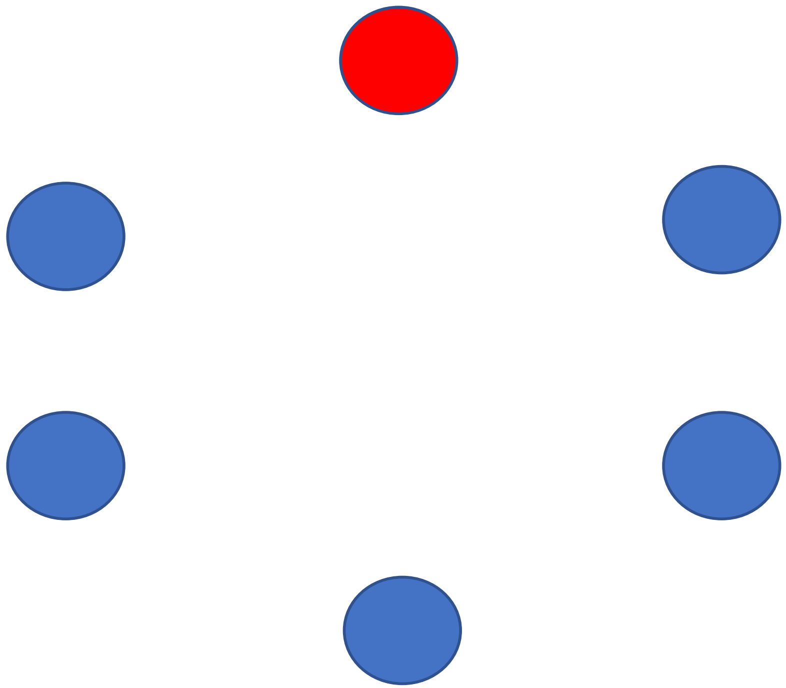
Back to Broadcast…

- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



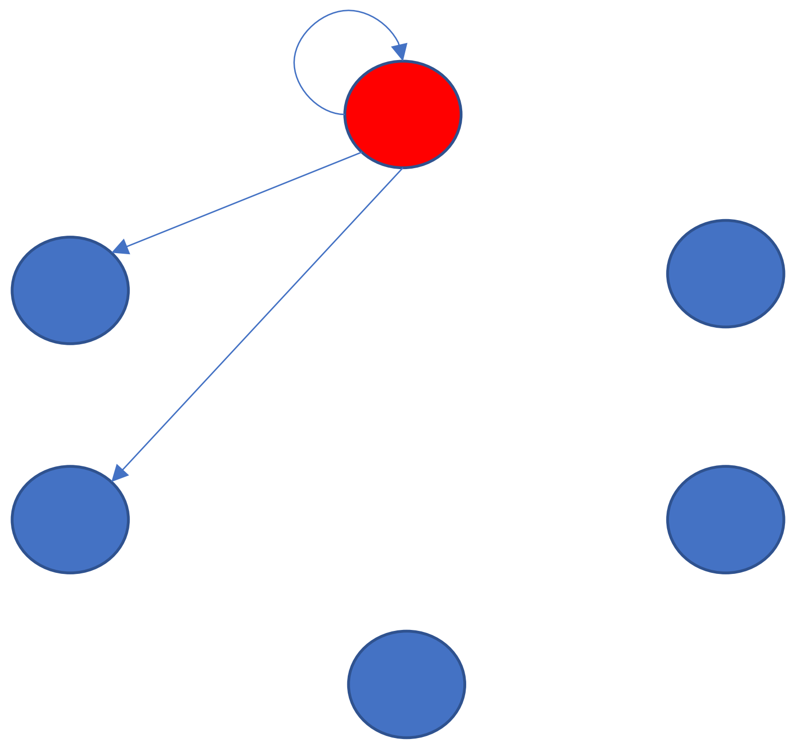
Back to Broadcast…

- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



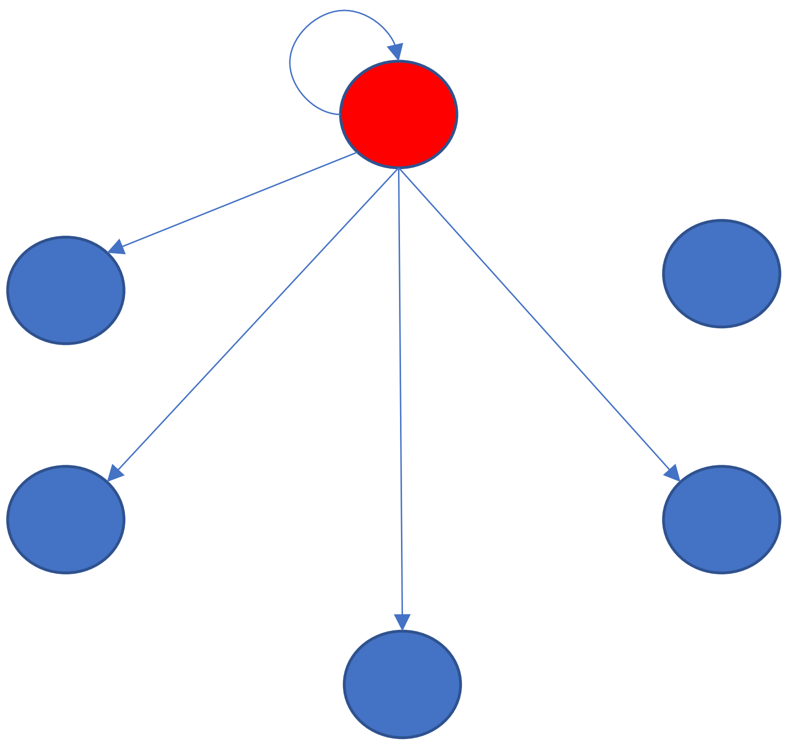
Back to Broadcast…

- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



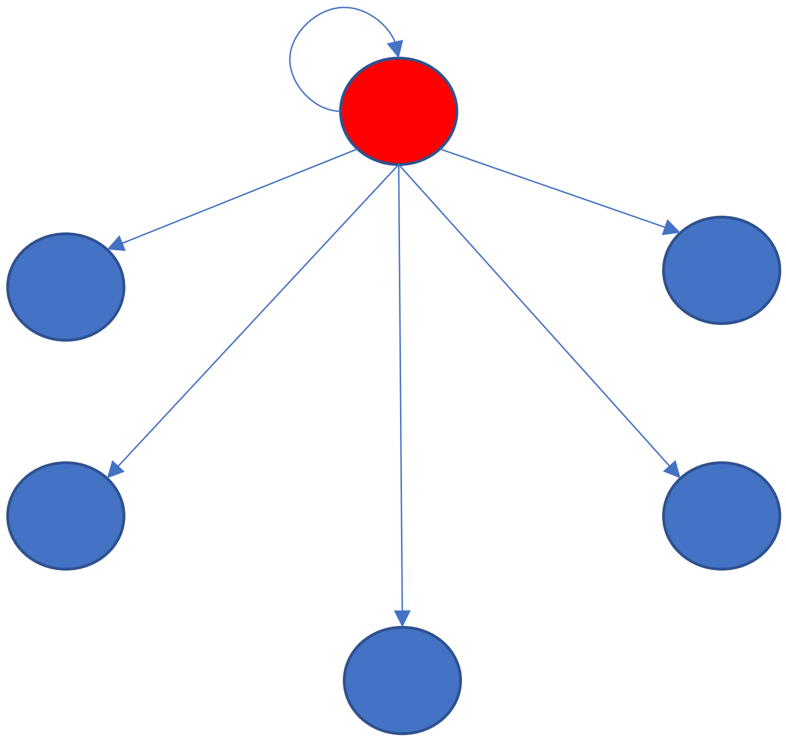
Back to Broadcast…

- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



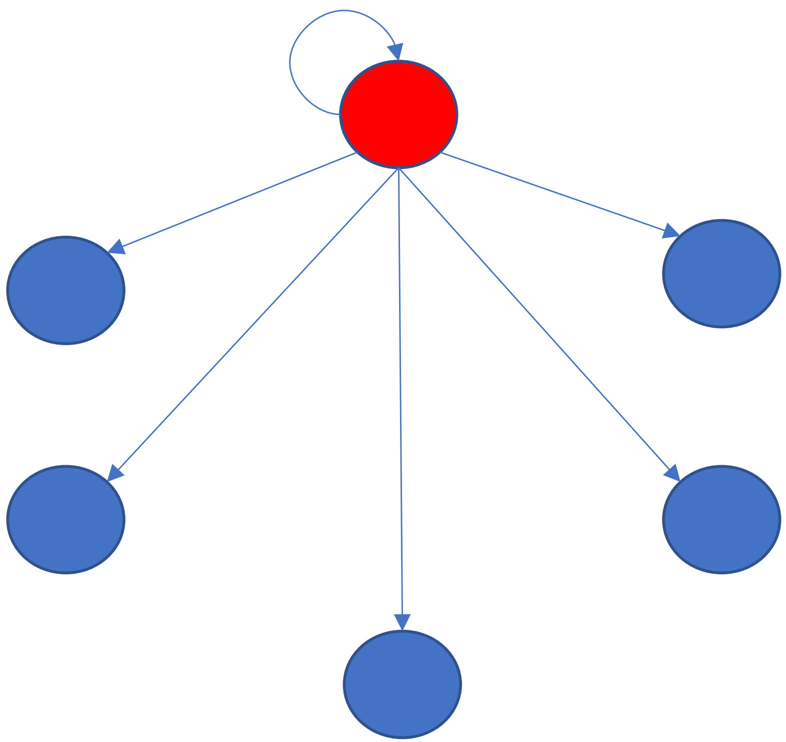
Back to Broadcast…

- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



Back to Broadcast…

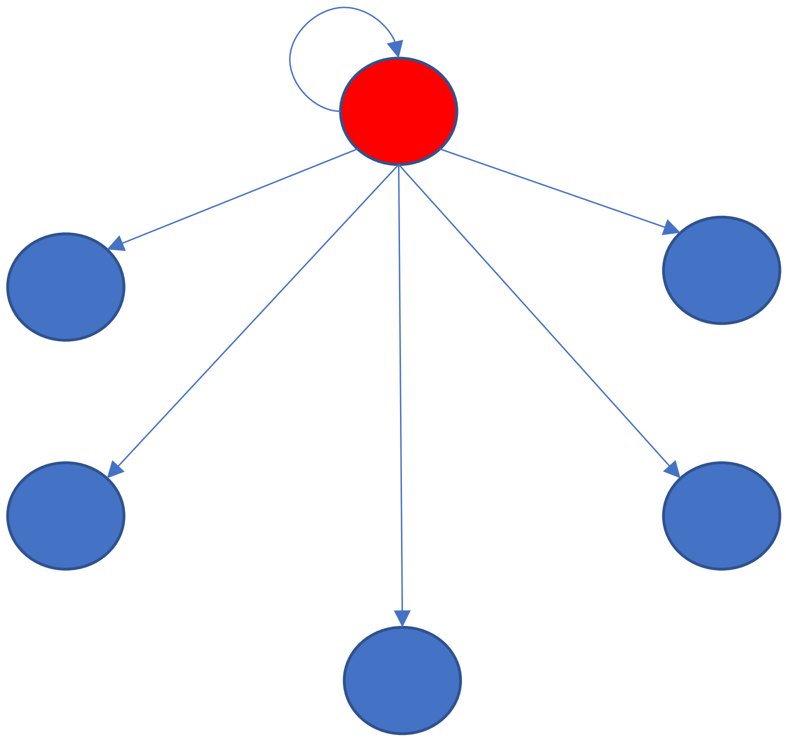
- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



Any problem here?

Back to Broadcast…

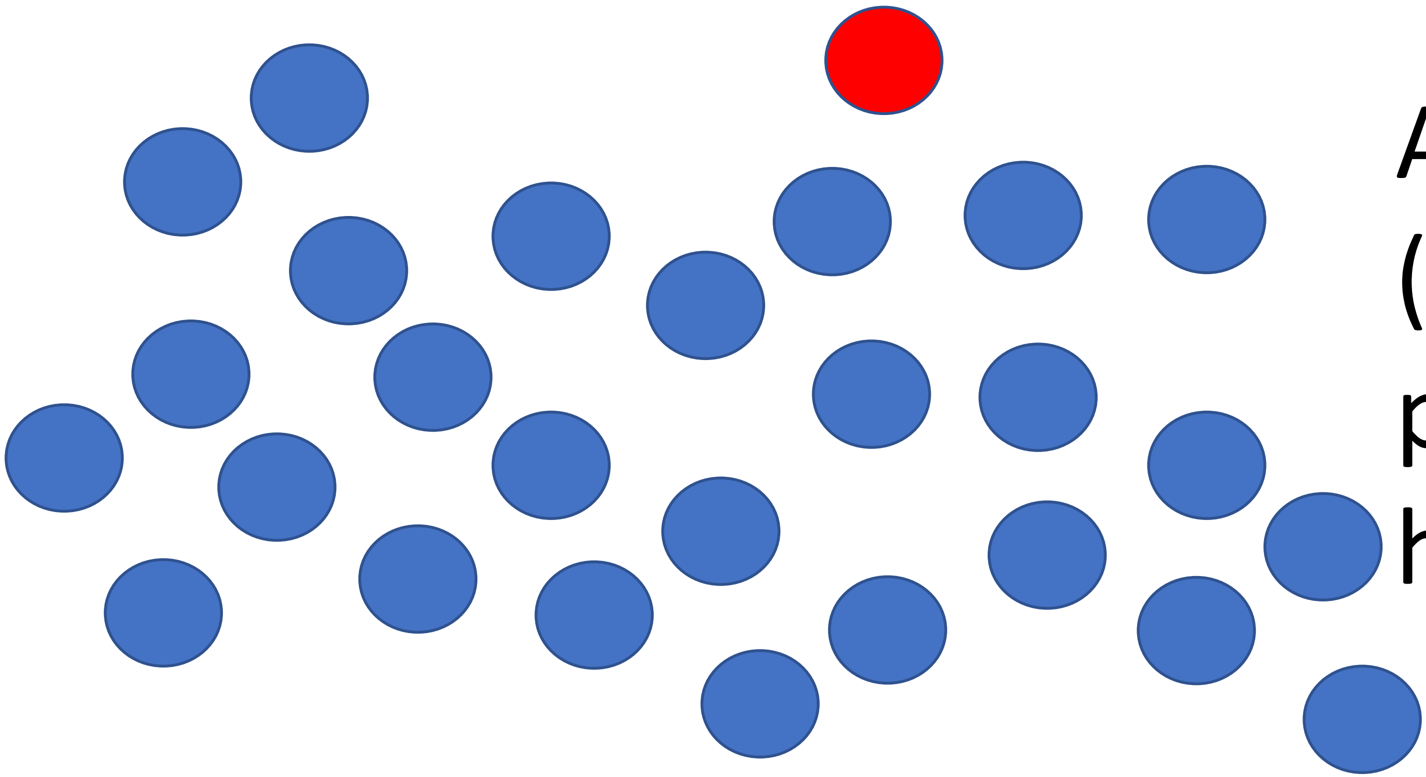
- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



Any (practical) problem here?

Back to Broadcast…

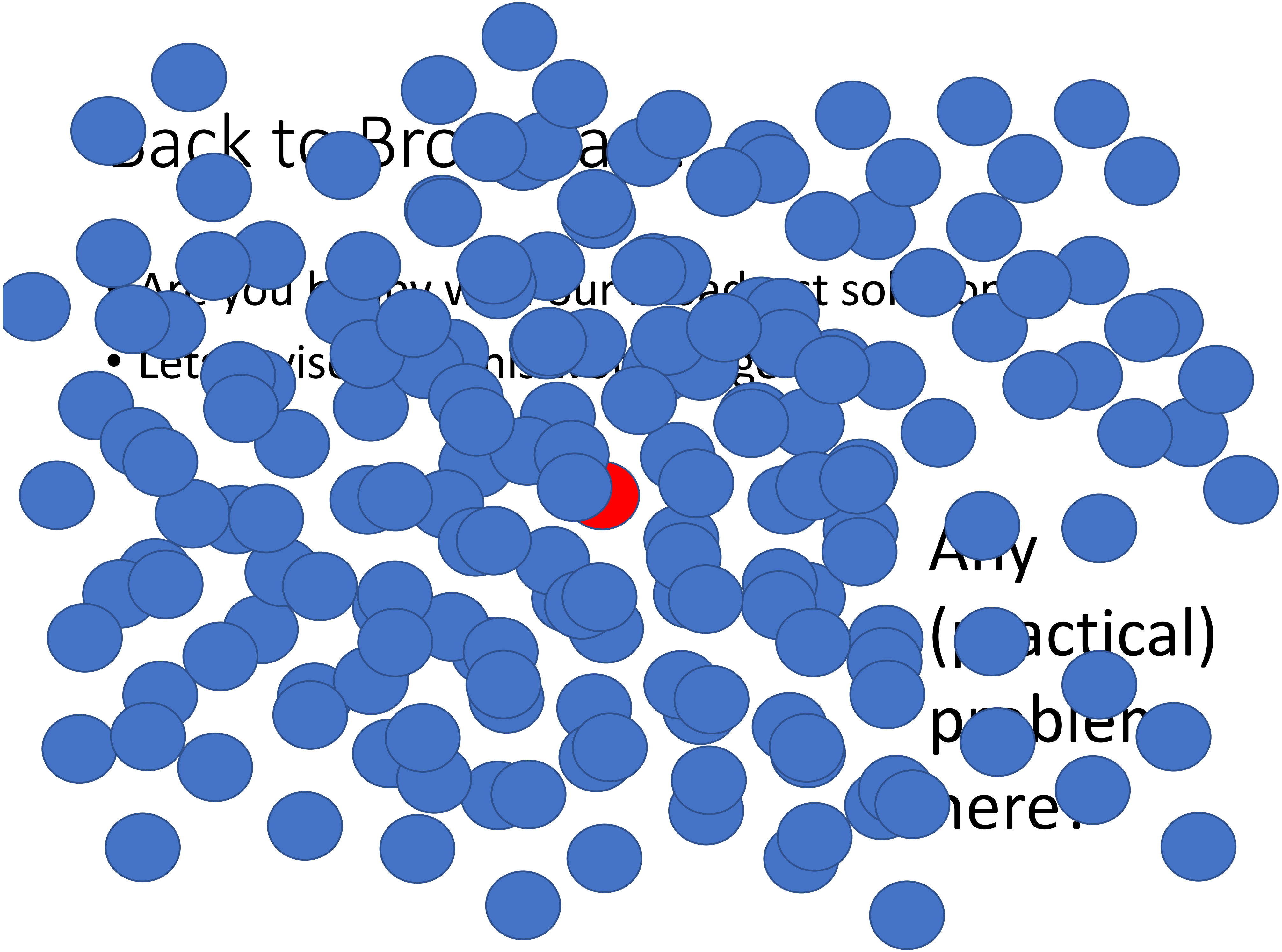
- Are you happy with our broadcast solutions??
- Lets revise how this works in general…



Any (practical) problem here?

Back to Bro... a...

- Are you h...y w... our ...ad...t sol...on
- Let... vis... his... g...



Any (practical) problem here?

Reliable Broadcast...

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

Reliable Broadcast...

- Solutions that we have seen are Reliable which is nice…
- ...but they put a lot of effort 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.

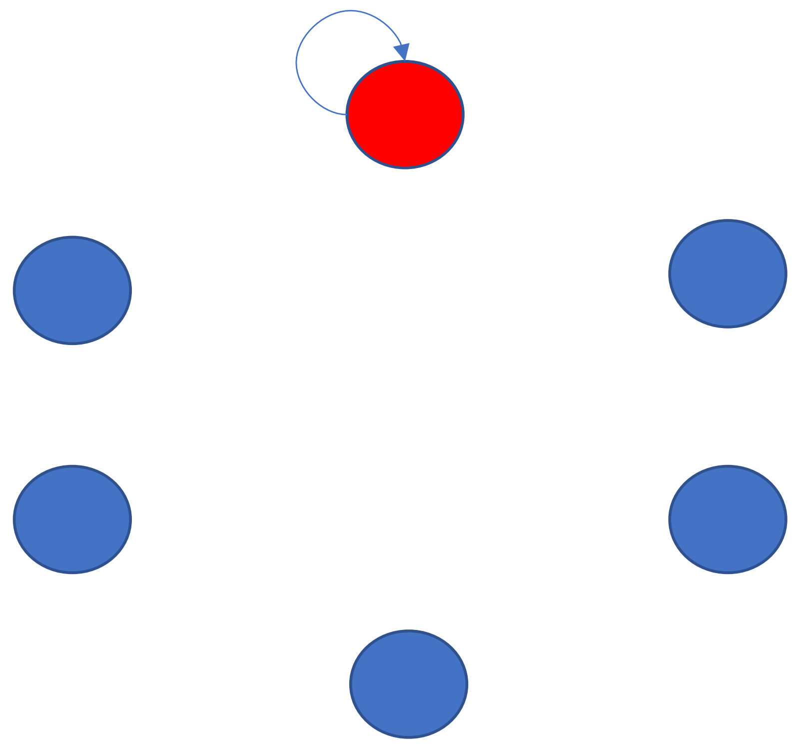
More Broadcast…

- So how can we address this issue?

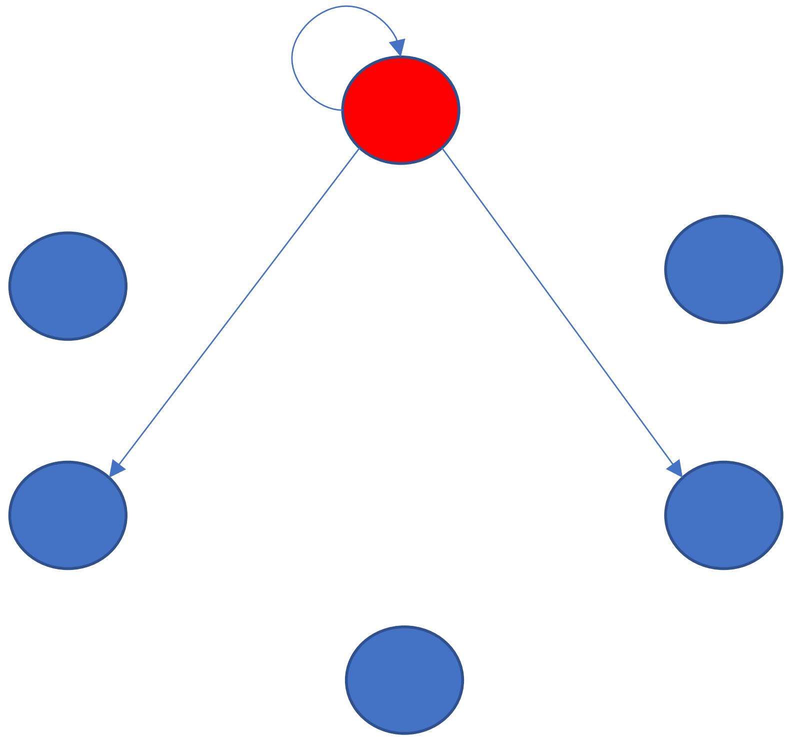
- A distributed system is composed by a set of **processes** that are interconnected through some **network** where processes seek to achieve some form of **cooperation** to execute tasks.

- Cooperation and sharing the load across the nodes: Welcome to the amazing World of Gossip

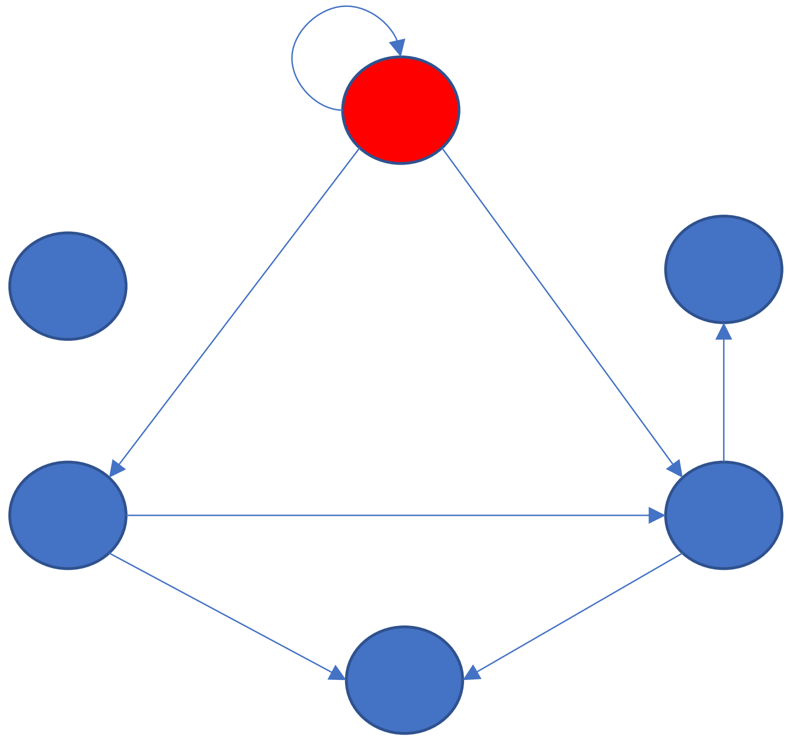
Gossip in a nutshell...



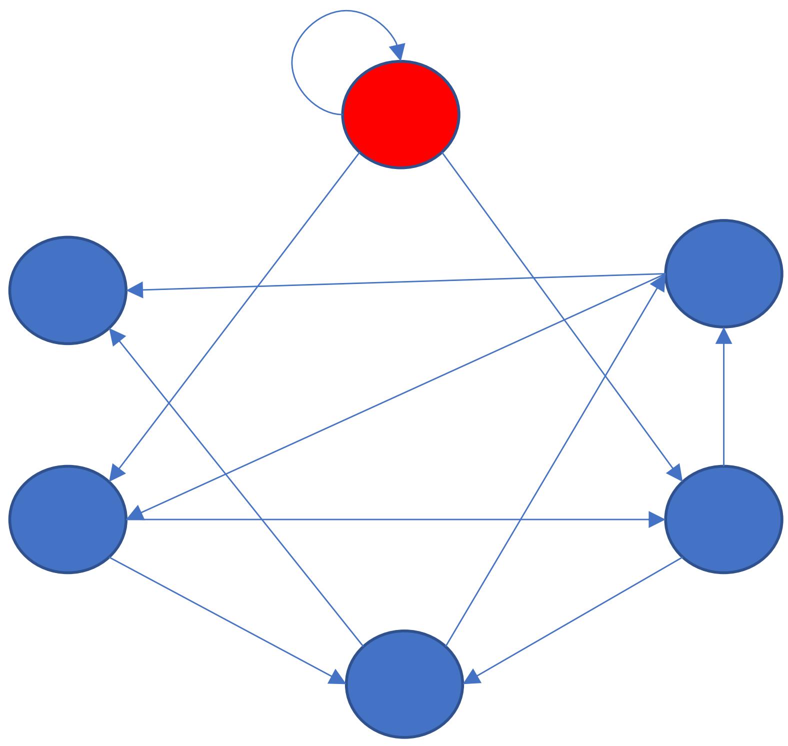
Gossip in a nutshell…



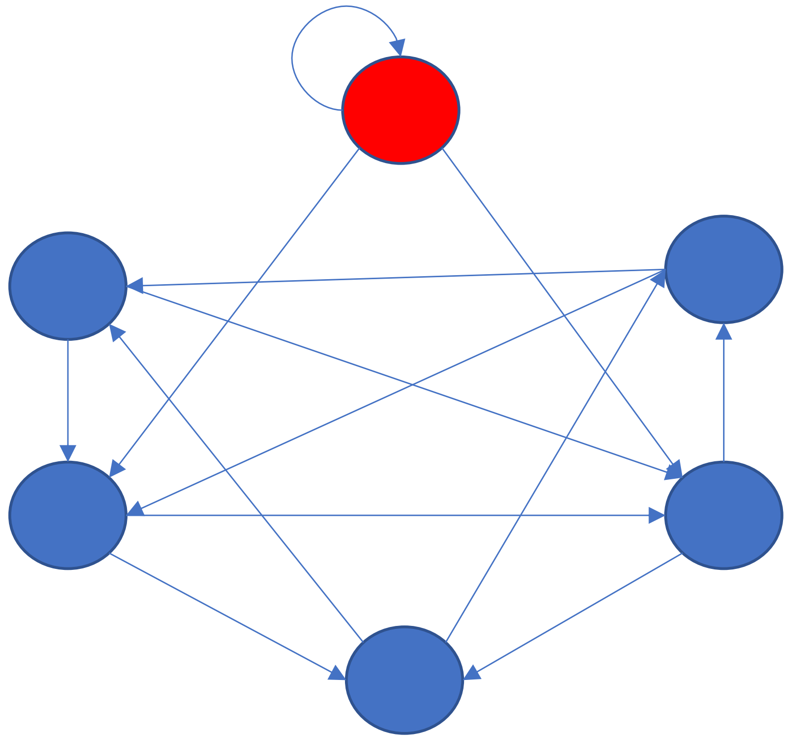
Gossip in a nutshell...



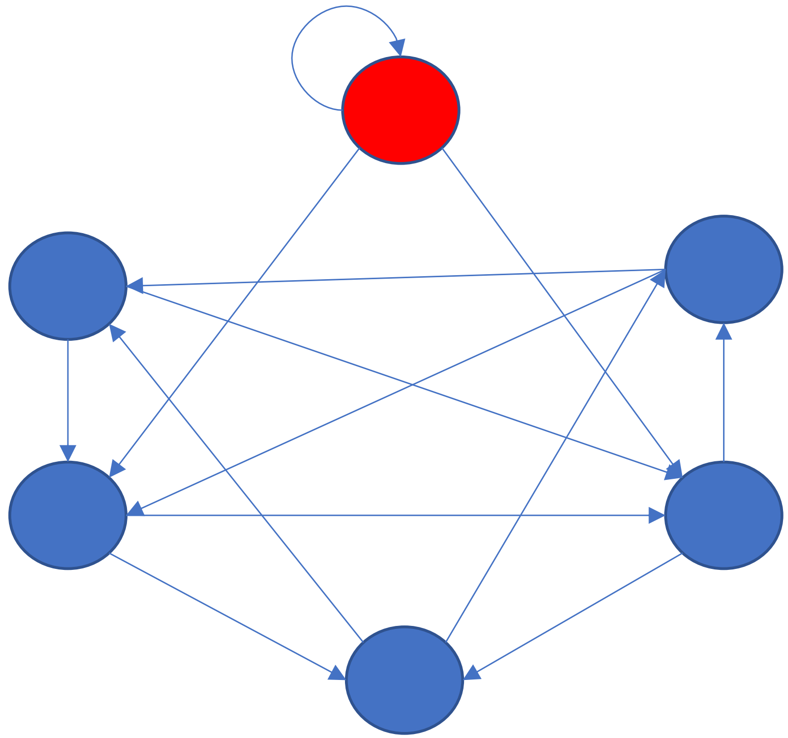
Gossip in a nutshell...



Gossip in a nutshell…



Gossip in a nutshell…



We do have some redundant messages… but everyone got the 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**.

A simple gossip algorithm:

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

A simple gossip algorithm:

- When a process wants to broadcast a message it picks *t* other processes from the system.
- These processes are selected **uniformly at random**.
- It then sends the message 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 a 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) actually provides an answer to this…
- To ensure high probability that everyone receives the message: t >= ln (# π)
- t is usually named the ***fanout*** of the algorithm).

A simple a 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) actually provides an answer to this…
- To ensure high probability that everyone receives the message: t >= ln (# π)
- t is usually named the ***fanout*** of the algorithm).

A simple a gossip algorithm:

Notice something weird here? Something that might not match with the specification of Reliable Broadcast?

- H ?
- N eter *t*

- The theory of epidemics (this is a real thing) actually provides an answer to this…
- To ensure **high probability** that everyone receives the message: t >= ln (# π)
- 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 (Aggrement): If a message m is delivered by some correct process i, them 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 $\longleftarrow \ln(\Pi)$;

        delivered $\longleftarrow \{\}$;

    Upon pBroadcast(m **) do:**

        mid $\longleftarrow$ generateUniqueID(m);

        Trigger pBcastDeliver (m**)**;

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

        gossipTargets $\longleftarrow$ randomSelection(t, Π);

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

            Trigger Send(GossipMessage, p, mid**,** m **)**;

    Upon Receive (GossipMessage, s**,** mid**,** m **) do:**

        If mid $\notin$ delivered **do:**

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

            Trigger pBcastDeliver (m**)**;

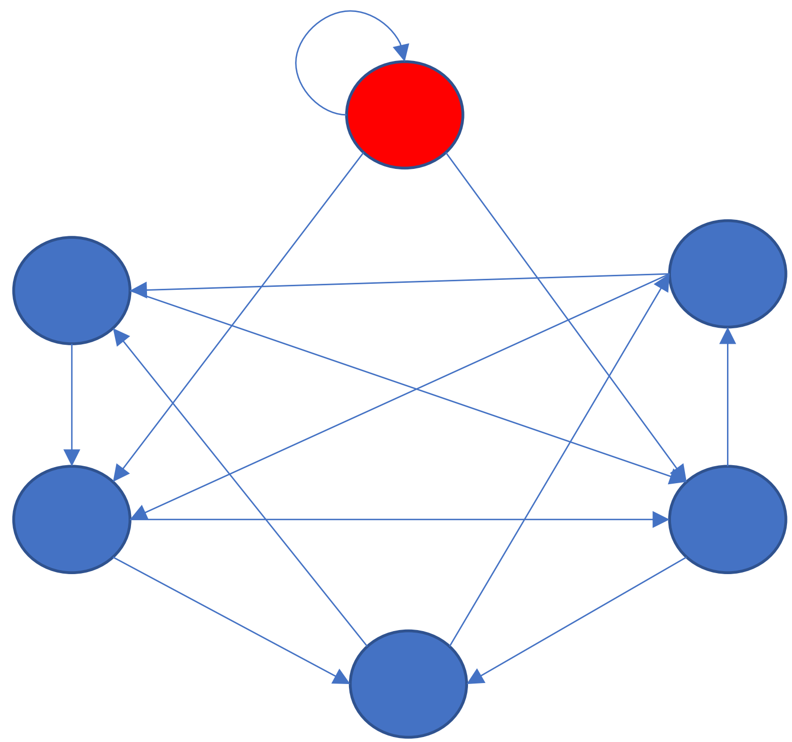
            gossipTargets $\longleftarrow$ randomSelection(t, ($\Pi \setminus$ s);

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

                Trigger Send(GossipMessage, p, mid**,** m **)**;

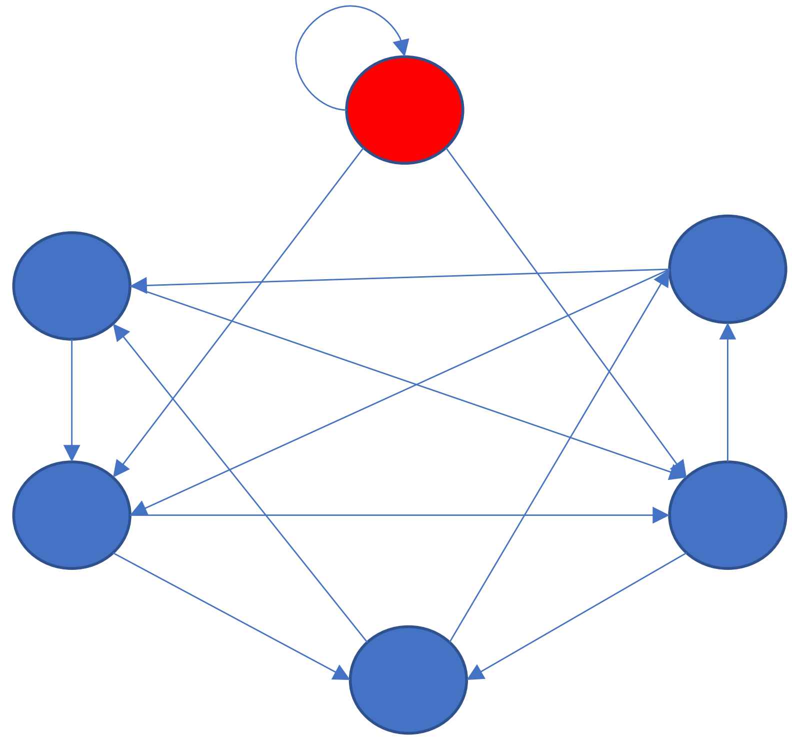
Gossip Redundancy
(the beauty and the beast)

- 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?*

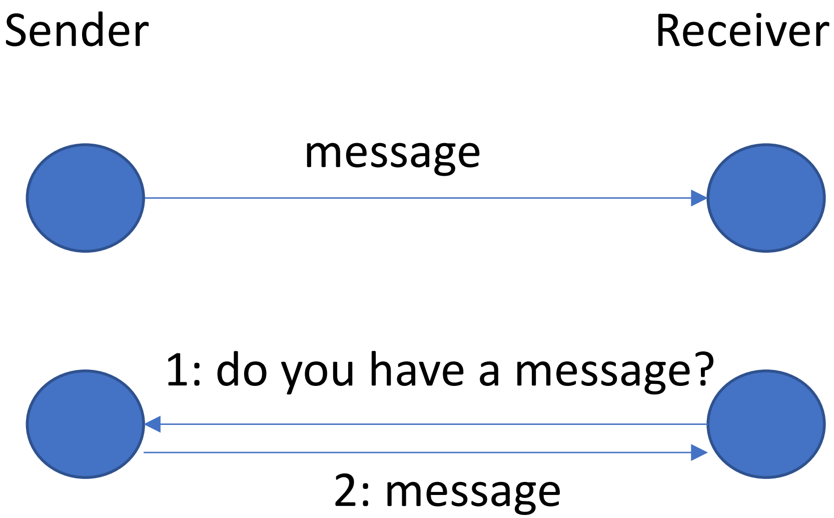
More practical aspects:

- 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 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?*

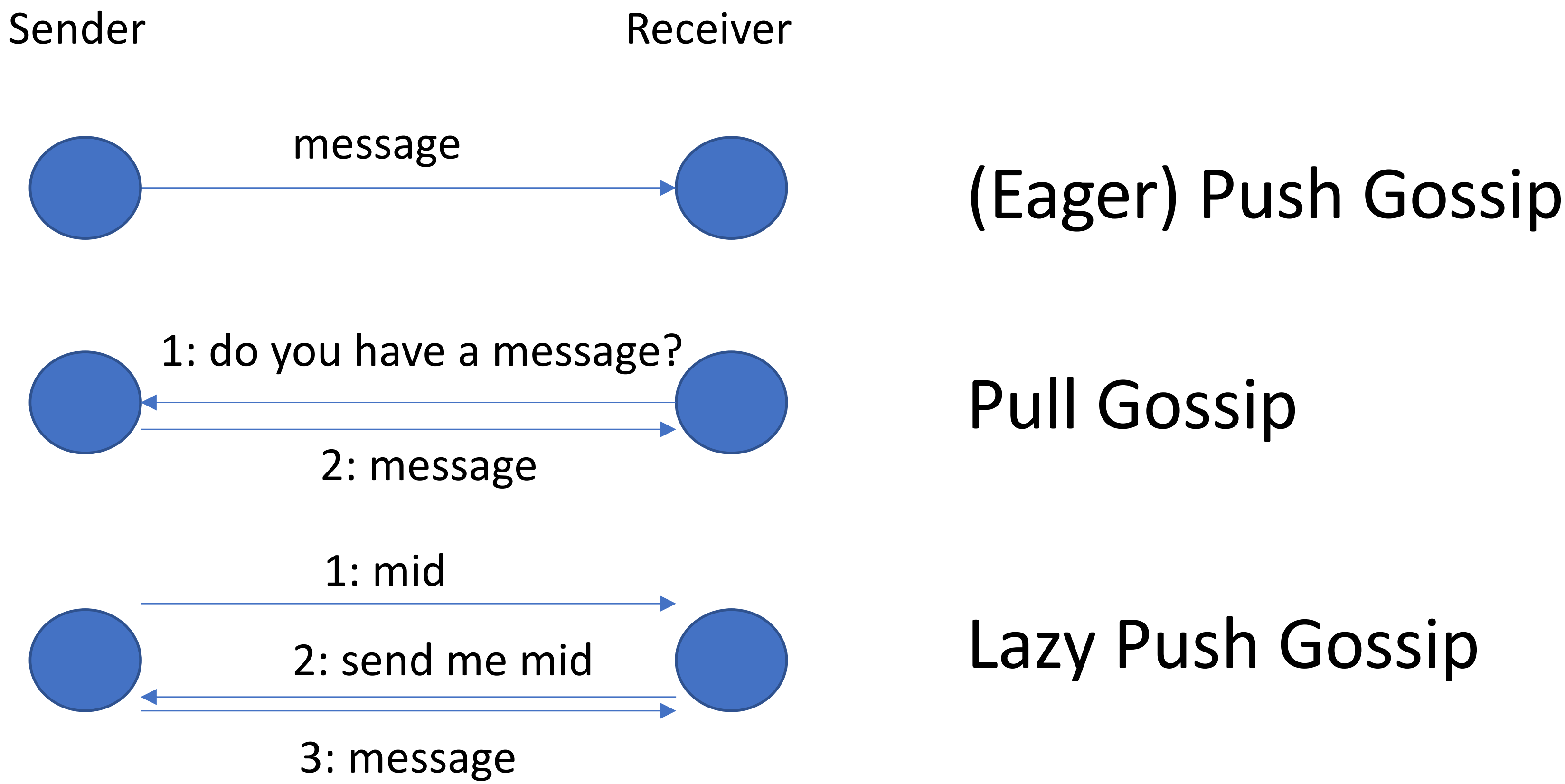
More ways to gossip:



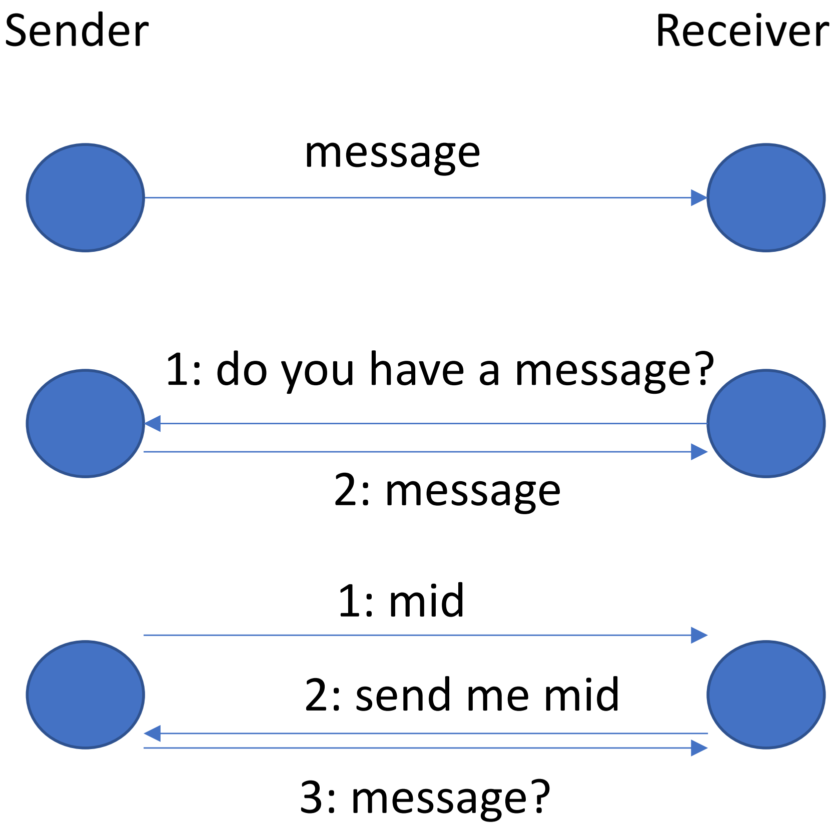
(Eager) Push Gossip

Pull Gossip

More ways to gossip:



More ways to gossip:



(Eager) Push Gossip

- Fast
- If messages are big -> expensive

Pull Gossip

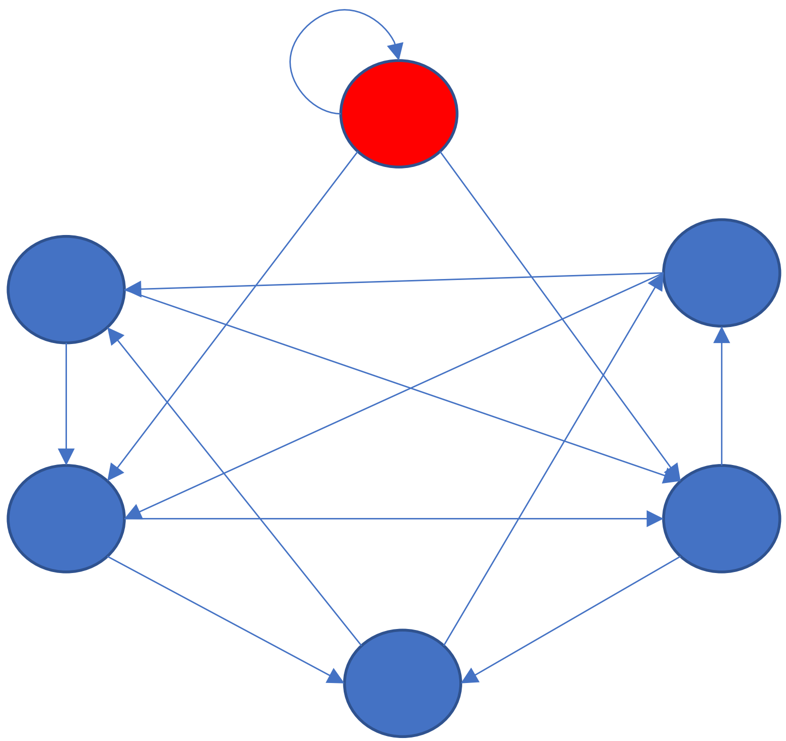
- If messages are big -> less network traffic
- Slow
- If messages are small more traffic

Lazy Push Gossip

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

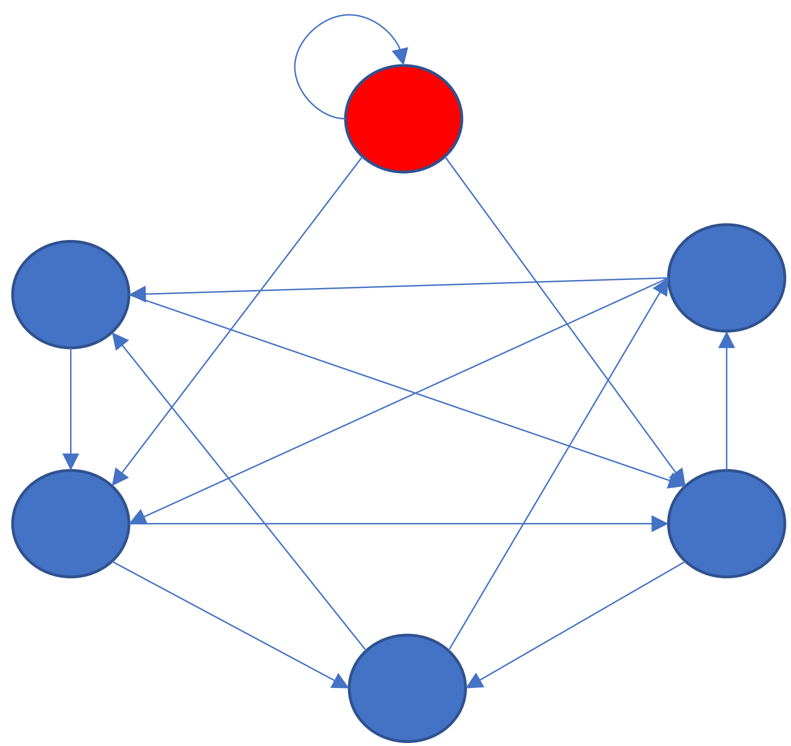
More practical aspects:

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



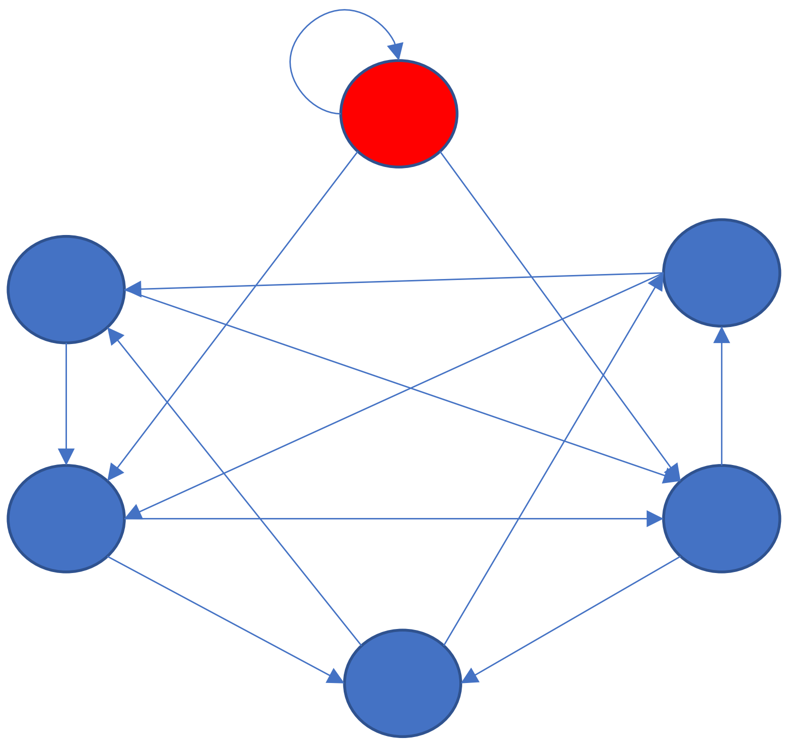
More practical aspects:

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



More practical aspects:

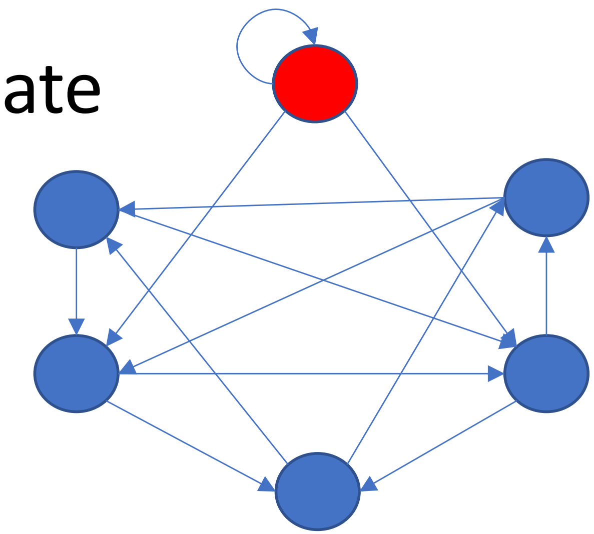
- There might be another hidden problem here that will limit scalability…
- This assumes a global **known membership**.



More formally, each process p has to know π.

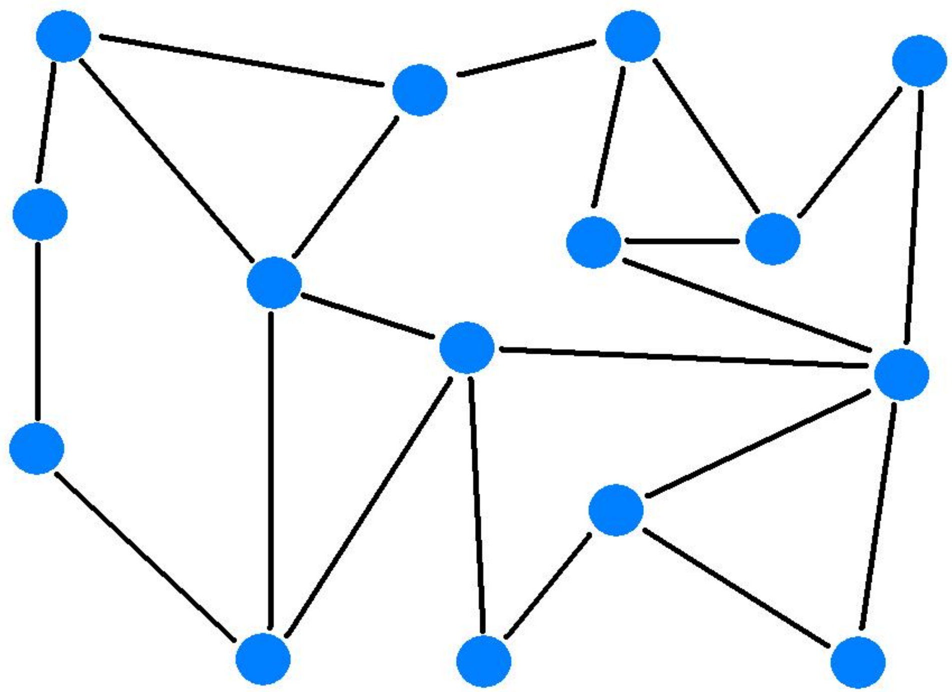
Why avoid a global known membership.

- In a very large system π 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 up-to-date might be too expensive.



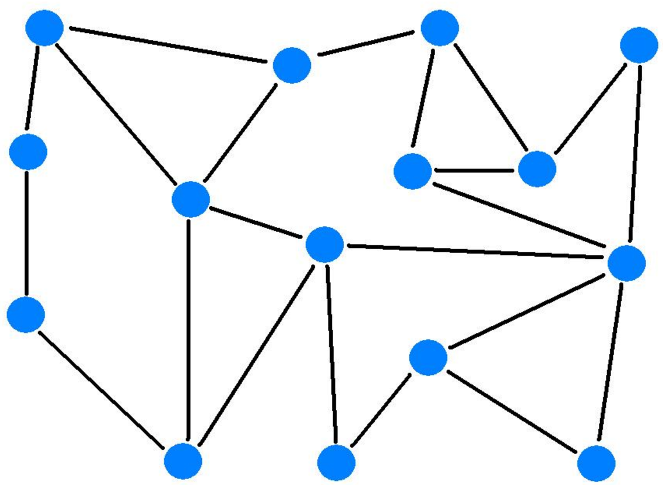
Partial View Membership System

- 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 na **overlay network**.



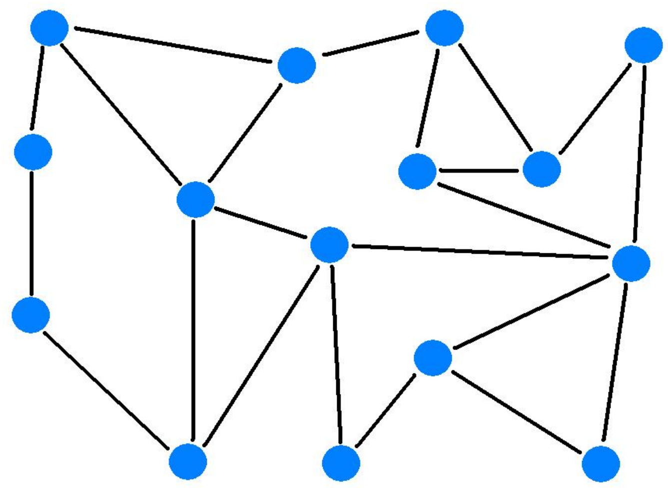
Overlay Network

- 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.
  - **Acuracy**: Eventually, no correct process p will have an overlay link to a failed process.



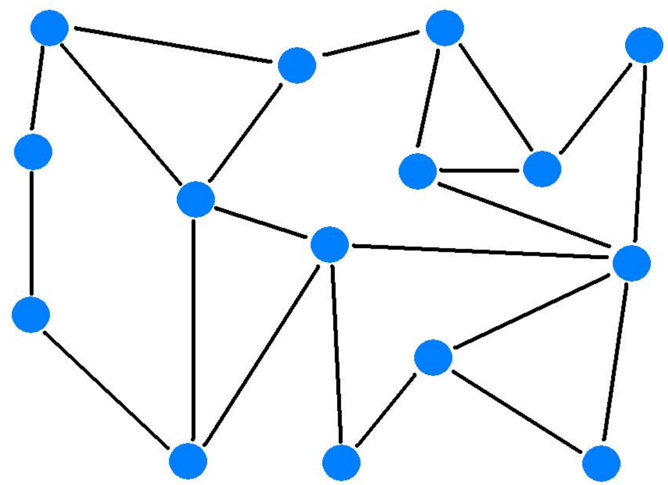
Overlay Network

- Nodes extablish (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).



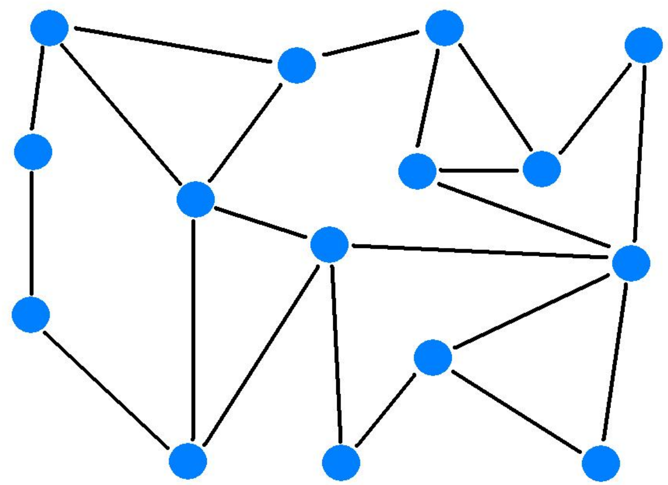
Random Overlay Network

- Nodes extablish (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).



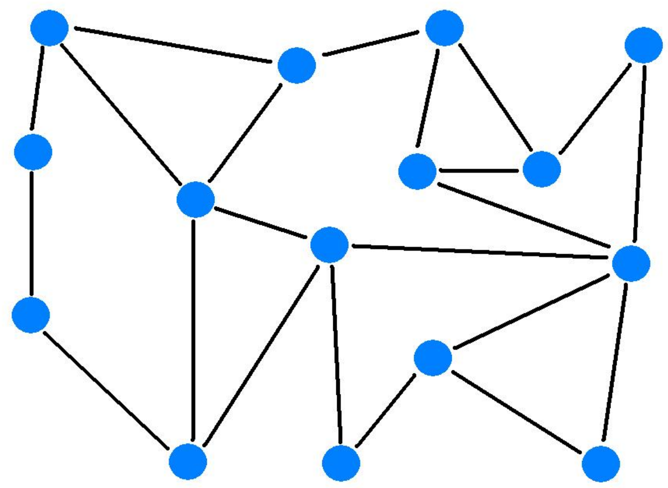
Unstructured Overlay Network

- Nodes extablish (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).



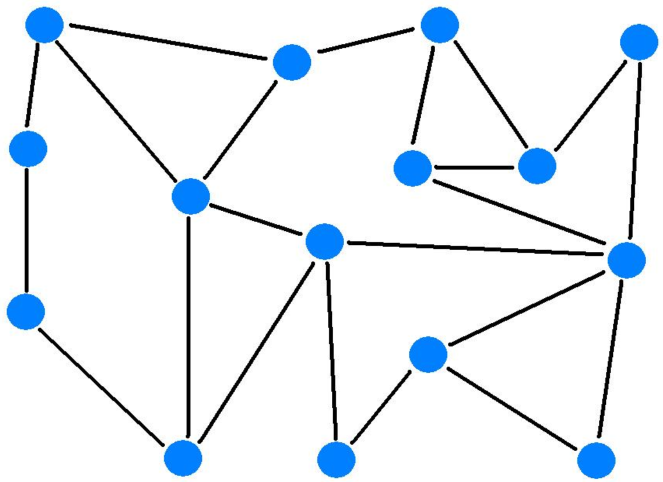
Unstructured Overlay Network

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



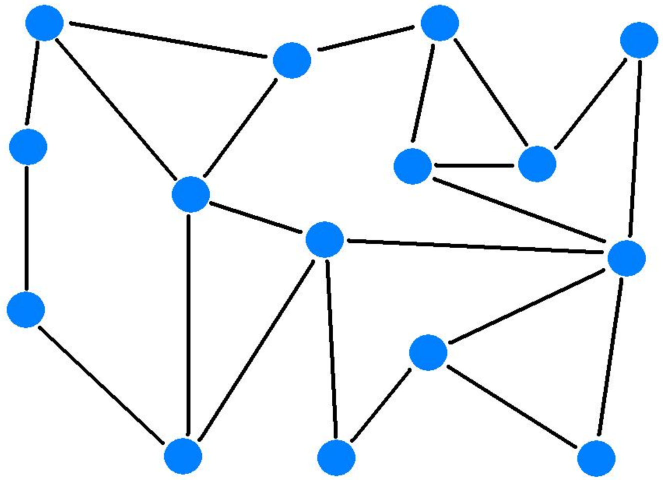
Unstructured Overlay Network

- 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 π you pick t random processes out of your logical neighbors.



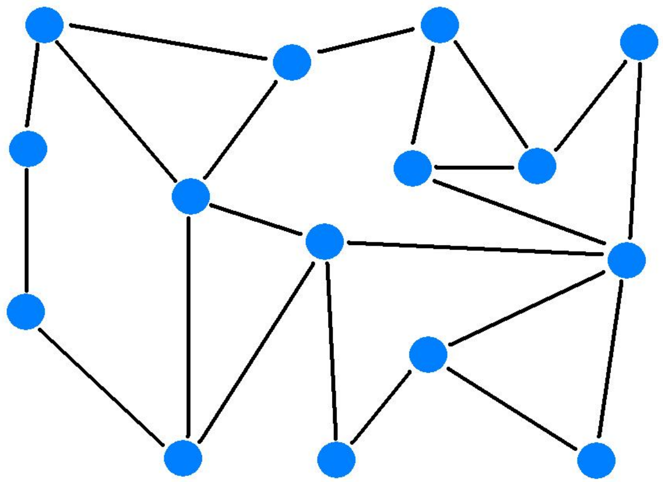
Unstructured Overlay Network

- 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,[1,2] Daniela Gavidia,[1] and Maarten van Steen[1]

Cyclon Intuition:

- When a new process joins it has to 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 the 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).

```
Algorithm 1: Cyclon (Unstructured Overlay Management)
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 ⟵ 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

  Upon getNeighbors( ) do:
    pview ⟵ neigh; //To avoid the upper layer to modify the neigh set
    Trigger neighbors( pview );

  Upon Shuffle( ) do:
    foreach (p, age) ∈ neigh do
      neigh ⟵ (neigh \(p, age)) ∪ (p, age + 1);
    p ⟵ pickOldest(neigh);
    If p ≠ ⊥ then
      neigh ⟵ neigh \ p;
      sample ⟵ randomSubset(neigh);
      Trigger Send (ShuffleRequest, p, sample ∪ {(myself, 0)});

  Upon Receive (ShuffleRequest, s, peerSample) do:
    temporarySample ⟵ randomSubset(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) ∈ peerSample do
      If (p′, age′) ∈ neigh ∧ p′ = p ∧ age′ > age then
        neigh ⟵ (neigh \(p′, age′)) ∪ {(p, age)};
      Else If #neigh < N then
        neigh ⟵ neigh ∪ {(p, age)};
      Else
        (x, age’) ⟵ (x,age’) : (x,age’) ∈ neigh ∧ (x,age”) ∈ mySample; //Pick an element of
                                                                  neigh that is also in mySample
        If (x, age’) = ⊥ then
          (x, age’) ⟵ (x,age’) : (x, age’) ∈ neigh; //Pick a random element of neigh
        neigh ⟵ (neigh \(x, age′)) ∪ {(p, age)};
```

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

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):
  - $t >= \ln(\pi)$ && $t < N$
  - 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

- **Random-Walk** (usefull to look for stuff in the overlay):
  - t = 1
  - Eager Push (usally with a maximum number of retrasnmissions).

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 = ?**
  - **??**