Fundamental Distributed Algorithms

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

alessio.bechini@unipi.it

Introduction:
"If you program your application using this kind of support, you don't have to worry about all the underlying communication and related details; everything is handled by the underlying platform. It's up to the platform to use algorithms to ensure the overall system works correctly. The software in the middleware layer addresses situations where various possible errors may occur. This is why it's crucial to be familiar with distributed algorithms.

The approach to dealing with distributed algorithms is different from that used for ordinary sequential algorithms because, in this case, we work with a model of the system. We assert that the algorithm will work properly only under specific assumptions. If these assumptions are not met, the algorithm will not function correctly. So, developing distributed algorithms is particularly challenging."

Outline

Introducing distributed algorithms and basic solutions for common problems

- Classes of algorithms
- Snapshots
- Mutual exclusion
- Working in faulty settings...
- Election

Classes of Algorithms

Basic Alg. vs Control Algorithms

Often a target *distributed* algorithm is supposed to run *at the same time* the overall system is performing a given job.

The algorithm run to perform such a job is named **Basic Algorithm**.

è l'algoritmo che realizza il lavoro principale per cui il sistema è stato creato.

The other, i.e. our target algorithm, is named **Control Algorithm**.

Il suo obiettivo non è quello di eseguire il lavoro principale del sistema, ma di assicurarsi che il Basic Algorithm funzioni efficacemente e correttamente, gestendo aspetti come la sincronizzazione, il coordinamento delle risorse, il controllo degli errori, o la manutenzione dello stato del sistema.

Sometimes it may be convenient considerare regarding a control algorithm as executed by a *controller process* cp_i that pairs to p_i at node i

che si abbina

p_i is the basic algorithm

Centralized vs Decentralized

In a given algorithm, a process is said "**initiator**" if it can start performing events *without any input from other processes*.

The first event of an initiator is either an internal one, or a send.
(il primo evento non può certo essere una receive in quanto è lui il primo processo ad attivarsi)

An algorithm is **centralized** if it has one single initiator.

An algorithm is **decentralized** if it admits multiple initiators.

NB: It is considered centralized if there is a node with different functionalities from the others that is responsible for initiating the algorithm and is the only one that can do so.

! ALL THE ALGORITHMS WE WILL SEE IN THIS COURSE ARE DECENTRALIZED !

Wave Algorithms (categoria di algoritmi)

"This category is crucial because a central concern is gaining control over the spread of information across all participating nodes in the application. Often, deciding whether to take a certain action or not relies on collecting information present in the different nodes of the application."

Common need: collection of information spread across processes.

Typically, a "request" originates at one node, and then involves all the nodes, which in turn must provide back the required information.

In a **wave algorithm**, a computation satisfies the following properties:

- It is finite
- It contains one or more *decision* events
- For each decision event d and process p_i, $c \xrightarrow{HB} d$ holds for some c at p_i

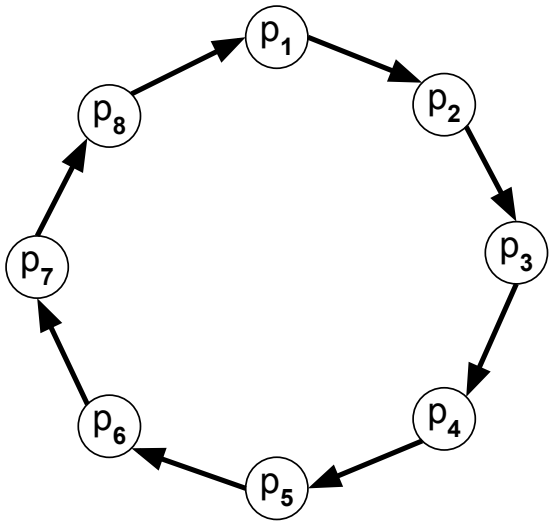
This means that 'd' should somehow be able to obtain information from the process 'p_i'. This implies that a decision point can take into account what happened in all the processes so far.

Note that, after the definition, each process must take part in the computation

It is not possible for one process to be disconnected from the decision event or events that we have identified.

A Trivial Wave Algorithm: Ring

Ogni nodo rappresenta un processo, le frecce rappresentano una comunicazione tra due processi.
In un anello, P2 non può comunicare con P6 se non facendo tutto il giro.



Network topology: directed ring

The initiator (whatever p_x) sends out a token msg:

```
send(toNeighbor, OutToken)
receive(fromNeighbor, inToken)
decide()
```

Any process p that receives a token msg performs:

```
receive(fromNeighbor, inToken)
send(toNeighbor, outToken)
```

In performing the decision action, you can take into account all the information you received from the other nodes that make up the ring, accumulating it message by message.

▲ IN THIS EXAMPLE, ALL NODES RUN EXACTLY THE SAME CODE.

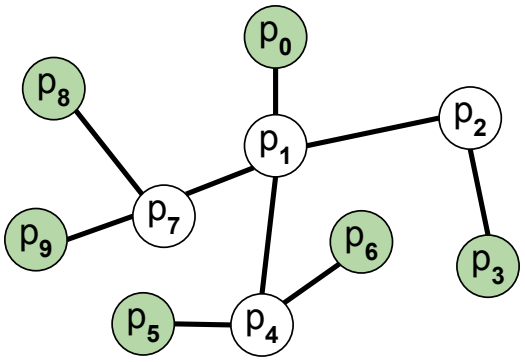
A More Complex Example: Tree Wave

è un albero, senza però una radice. Possiamo solo distinguere le foglie: hanno un solo vicino.

Undirected Tree Topology.

Each process sends only one msg.

Initial msgs: (viene da) out of leaves.




```
bool recv[n_neigh] //array initialized [false]

// loop executed n_neigh - 1 times
while len(filter(false, recv)) > 1:
     receive(fromNeighbor, inToken)
     recv[fromNeighbor] = true

//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary…
decide()
//this way, only two processes decide. Possibly:
//foreach n in neighbors\toNeighbor:
//     send(n, outToken)
```

le foglie avendo un solo vicino skippano questa parte

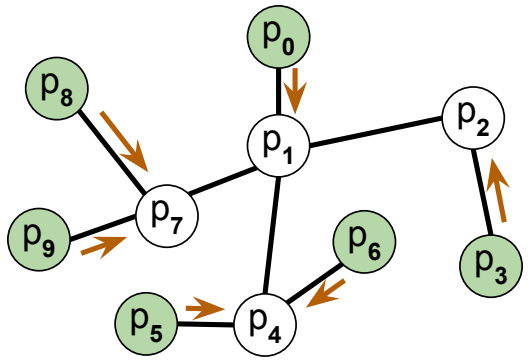
filter := filtra l'array recv per trovare tutti gli elementi che sono false

scritto in questo modo, ogni nodo invia e ogni nodo riceve. Quindi avremo N decide() quanti sono i nodi.

i messaggi vengono propagati dalle foglie verso l'interno fino a che tutti i nodi hanno ricevuto i messaggi necessari per prendere una decisione finale.

A More Complex Example: Tree Wave

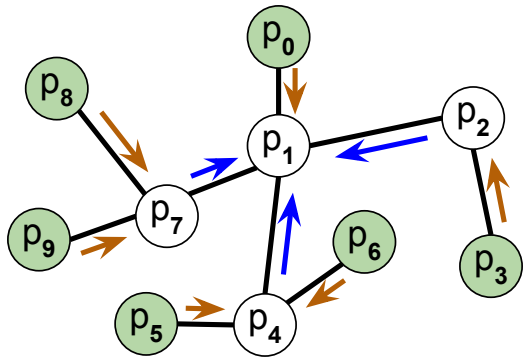
Undirected Tree Topology.
Each process sends only one msg.
Initial msgs: out of leaves.




```
bool recv[n_neigh] //array initialized [false]

// loop executed n_neigh - 1 times
while len(filter(false, recv)) > 1:
     receive(fromNeighbor, inToken)
     recv[fromNeighbor] = true

//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary…
decide()
//this way, only two processes decide. Possibly:
//foreach n in neighbors\toNeighbor:
//     send(n, outToken)
```


A More Complex Example: Tree Wave

Undirected Tree Topology.
Each process sends only one msg.
Initial msgs: out of leaves.




```
bool recv[n_neigh] //array initialized [false]

// loop executed n_neigh - 1 times
while len(filter(false, recv)) > 1:
     receive(fromNeighbor, inToken)
     recv[fromNeighbor] = true

//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary…
decide()
//this way, only two processes decide. Possibly:
//foreach n in neighbors\toNeighbor:
//     send(n, outToken)
```

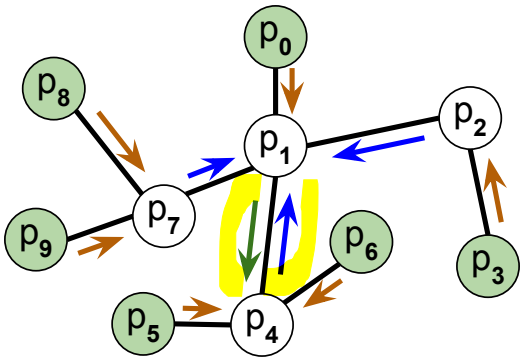
P1 received messages from all its neighbors, so it will be able to decide.
It can send out its own message before receiving one of them.

© A.Bechini 2023

A. Bechini - UniPi

A More Complex Example: Tree Wave

Undirected Tree Topology.
Each process sends only one msg.
Initial msgs: out of leaves.



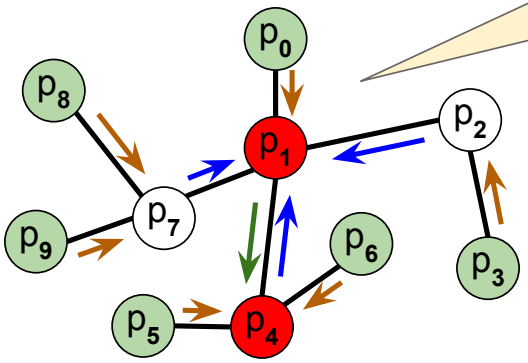

```
bool recv[n_neigh] //array initialized [false]

// loop executed n_neigh - 1 times
while len(filter(false, recv)) > 1:
     receive(fromNeighbor, inToken)
     recv[fromNeighbor] = true

//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary…
decide()
//this way, only two processes decide. Possibly:
//foreach n in neighbors\toNeighbor:
//     send(n, outToken)
```

The nodes able to decide are just two, as only those two nodes have executed the receive present in the second part of the code.
So, you will have two nodes that have (potentially) received the state of all the other nodes in the system.

A More Complex Example: Tree Wave

Undirected Tree Topology.
Each process sends only one msg.
Initial msgs: out of leaves.

Processes getting
to the decision point
have (potentially) received information
from all the other nodes

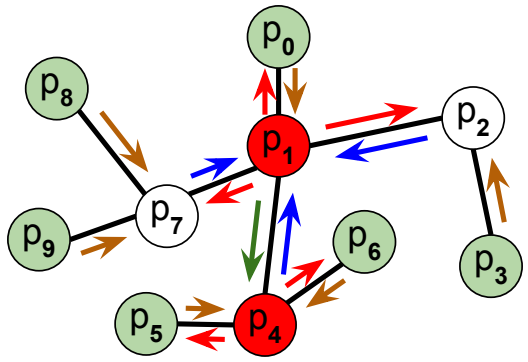


Secondo me, non per forza p1 invia a p4.
p1 invierà al nodo che fra p2, p4 e p7 invia il messaggio a lui (p1) per ultimo.

© A.Bechini 2023

A More Complex Example: Tree Wave

Undirected Tree Topology.
Each process sends only one msg.
Initial msgs: out of leaves.



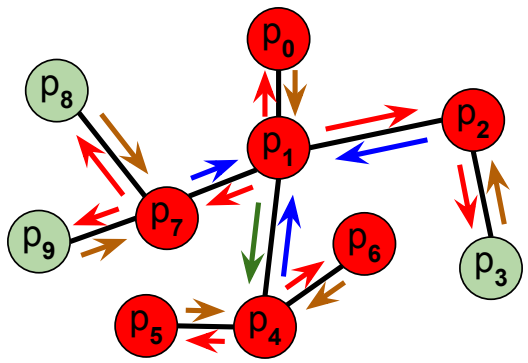

```
bool recv[n_neigh] //array initialized [false]

// loop executed n_neigh - 1 times
while len(filter(false, recv)) > 1:
     receive(fromNeighbor, inToken)
     recv[fromNeighbor] = true

//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary…
decide()
//this way, only two processes decide. Possibly:
foreach n in neighbors\toNeighbor:
     send(n, outToken)
```

tutti i vicini tranne toNeighbor

After deciding, we can also add the final part of the code that spreads the information about the decision taken to the other nodes. In this way, all nodes (including leaves) can make a decision based on the information received from all the other nodes.

A More Complex Example: Tree Wave

Undirected Tree Topology.
Each process sends only one msg.
Initial msgs: out of leaves.



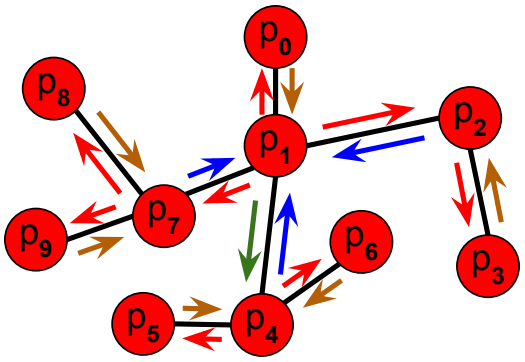

```
bool recv[n_neigh] //array initialized [false]

// loop executed n_neigh - 1 times
while len(filter(false, recv)) > 1:
     receive(fromNeighbor, inToken)
     recv[fromNeighbor] = true

//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary…
decide()
//this way, only two processes decide. Possibly:
foreach n in neighbors\toNeighbor:
     send(n, outToken)
```

© A.Bechini 2023

A. Bechini - UniPi

A More Complex Example: Tree Wave

Undirected Tree Topology.
Each process sends only one msg.
Initial msgs: out of leaves.




```
bool recv[n_neigh] //array initialized [false]

// loop executed n_neigh - 1 times
while len(filter(false, recv)) > 1:
     receive(fromNeighbor, inToken)
     recv[fromNeighbor] = true

//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary…
decide()
//this way, only two processes decide. Possibly:
foreach n in neighbors\toNeighbor:
     send(n, outToken)
```


Traversal Algorithms

A **traversal algorithm** is a *centralized* wave algorithm.
In a computation, the initiator sends out token(s), and:

- All the processes receive the token
- Finally, the token(s) returns back to the initiator, which performs a decision event

Traversal algorithm is another type of algorithm which is just a special form of wave algorithm.
A traversal algorithm is a centralized wave algorithm, meaning that every node is permitted to be the initiator.
Thus, we can identify one special node with a special role.

Taking Snapshots

Nei sistemi distribuiti, mantenere una visione coerente dello stato del sistema rappresenta una sfida cruciale. Questa necessità emerge in vari contesti, come il debugging, la tolleranza ai guasti e il monitoraggio delle prestazioni.

Uno snapshot di un sistema distribuito cattura lo stato globale del sistema in un determinato momento, comprendendo lo stato di tutti i processi e i messaggi in transito. Riuscire a ottenere uno snapshot coerente è complesso a causa dell'assenza di un orologio globale e della natura asincrona dei sistemi distribuiti.

What Do We Mean by “Snapshot”?

Snapshot of the execution of a distributed algorithm: configuration consisting of the **local states** of processes, along with (insieme a) **messages in transit** (i.e. channel states).

A snapshot is **consistent** if it is taken just after a **consistent cut** (corresponding to a consistent global state).

consistent cut := insieme di eventi in un sistema distribuito che mantiene la coerenza temporale e causale delle azioni all'interno del sistema. In altre parole, se un evento e1 avviene prima di un evento e2 in un processo, e e2 è incluso nel cut, allora e1 deve essere incluso nel cut.

Why Snapshots?

Snapshots can be aimed at:

- Check properties
  (in particular, **stable predicates**, those that will keep on being true):
  - Deadlock
  - Termination
  - Etc.
- Checkpointing

sono predicati che, una volta che diventano veri, rimangono veri

Snapshot Algorithm (I)

A Snapshot Algorithm is a **Control Algorithm** that must obtain a picture of what it is going on *with a basic distributed algorithm* at a given progression point.

è un algoritmo che ha il compito di monitorare l'esecuzione di un algoritmo base.

di ciò che sta accadendo

con un

Actually, it "monitors" executions...

This operation in a distributed system
is not so trivial as it is in a centralized system.

A snapshot algorithm is asked to run "on-the-fly," along with the basic one, with no need to stop the system to inspect its state.

(to run at the same time the basic algorithm is run)

It is required that the controlled basic algorithm must not be stopped simply for the purpose of taking a snapshot.

Snapshot Algorithm (II)

A Snapshot Algorithm is required to take a **consistent snapshot**, i.e. a *possible* configuration of the ongoing execution. The related state-recording operations can occur at different points in time.

this means that the state recording operation at the different nodes can occur at different points in time, meaning different points of the global clock. However, they have to provide a vision of a progression point of the application that is consistent.

In general, a snapshot algorithm can be initiated by any process; each process is in charge of locally recording its own portion of the snapshot. Later, all the portions can be collected (e.g. by a traversal algorithm, etc.).

This is not a limitation because we have seen that we are able to develop algorithms to collect information from all the nodes, such as traversal algorithms.

Please note that each distributed algorithm, will function correctly only if the assumptions under which it was designed are met. If these assumptions are not met, the algorithm may fail to provide the required data.
This is fundamentally different from ordinary sequential algorithms.

Chandy-Lamport Algorithm (I)

The most popular snapshot algorithm. System hypotheses:

- FIFO channels — So in the communication between process A (Alice) and process B (Bob), if Alice sends two messages to Bob over the same channel, message one will always arrive before message two.
- Strongly connected (di)graph ➜ all the processes can be reached

It is not necessary for each process to have a direct link with every other process; it is sufficient that each process can communicate with the others using the communication network.

Idea: local recording actions driven by special "marker" messages.

Thus, when a message reaches a certain node, the node can start and perform actions related to the snapshot.

Any process can start a snapshot.

A channel state (corresponding to *basic* msgs in transit through it) is recorded by the destination process.

Each node is responsible for recording its own internal state, which is straightforward. However, recording the state of the channels is more complex. We must determine which process will be responsible for recording the state of each channel.

The channel state, which includes the messages in transit, is recorded by the destination process. Therefore, each process is responsible for recording the state of its incoming channels.

Each node will maintain an array (chmark[·]) to mark the state of the channels to which it is connected, indicating the reception of marker messages from these channels.

Chandy-Lamport Algorithm (II)

The initiator performs:

```
procedure takeSnapshot()
   if(not marked)then
      marked = true
      foreach(c in outchannels){send(c, marker)}
      <record local state + incoming channels' state>
```

marked is just a flag

At this point, other nodes may receive this marker message.

Any process p receiving a marker msg performs:
(msg received on channel C)

```
procedure onMarker(channel c)
   takeSnapshot()
   chmark[c] = true
   if(chmark[ic]  for each incoming channel  ic)then
      <local snapshot termination>
```

I check if for each incoming channel, there is a corresponding true in the chmark array, indicating that a message has been received from each incoming channel.
(Here, "ic" is a variable I use in the loop to iterate through the entire array.)

Qui è giusto l'if, e non il while come pensavo io. Se ad una ennesima esecuzione di questa funzione l'if dà false, quando eseguirò nuovamente questa funzione prima o poi avrò true.

Any process p receiving a basic msg performs:

```
procedure onBasicMsg(channel c, msg m)
   if(marked and not chmark[c]) then
      chstate[c].append(m) //state for chann. c
```

Each node will also have a data structure to keep track of the state of the incoming channels, specifically recording which messages were in transit at the snapshot instant.

The complexity of an algorithm is typically evaluated with respect to a specific resource, often time. In a sequential algorithm might consider memory as another critical resource. In the case of distributed algorithms, an important resource to consider is the number of messages exchanged.

Chandy-Lamport Algorithm: Complexity

Message complexity:

we have just one marker message per FIFO channel, so it is $\Theta(E)$, with E as # of channels.

E stands for edges (archi)

Time complexity:

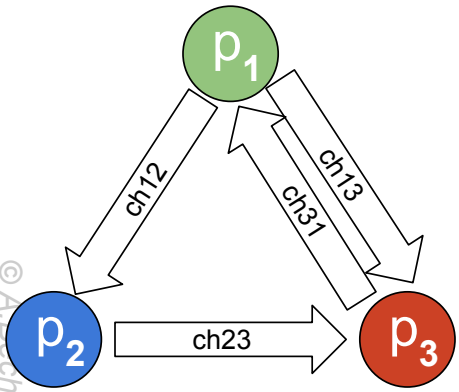
all the processes have to be reached, thus in the worst case it depends on the longest of the paths (# of hops) between those that "link" two generic processes in the network, i.e. the *network diameter* D. The time complexity is $O(D)$.

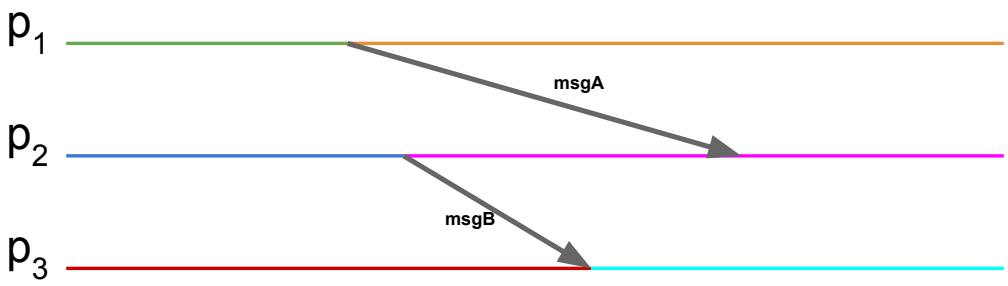
The evaluation of time complexity is more intricate because it involves ensuring all processes are reached. In the worst-case scenario, the time taken for the algorithm to complete depends on the placement of nodes within the network graph, especially where the initiator node is located.
If the initiator node is on one edge of the graph and needs to reach a distant node elsewhere in the network, the maximum number of hops required determines the time complexity. Therefore, in the worst case, the time complexity is determined by the longest path in terms of the number of message transmissions.

network diameter := is the longest path that connects two distinct nodes within the graph.

Chandy-Lamport Alg. Example (I)

Three processes; in the basic algorithm, p_1 sends msgA to p_2 and p_2 sends msgB to p_3, as shown.

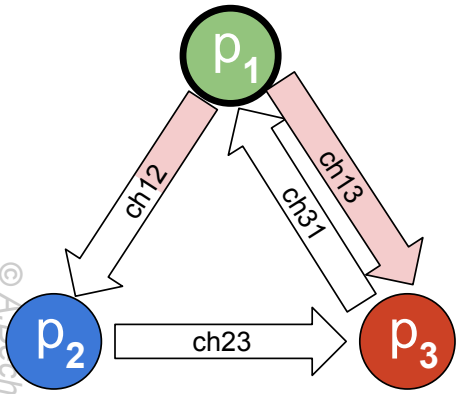


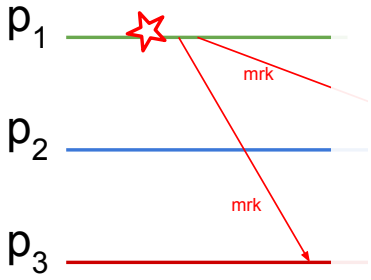


Chandy-Lamport Alg. Example (II)

p1 dà il via all'algoritmo di snapshot

p_1 starts, recording its state (green), and sends mrk to p_2 and p_3.
p_3 receives mrk first, so...

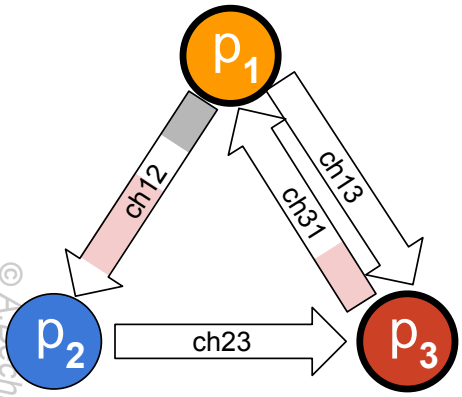


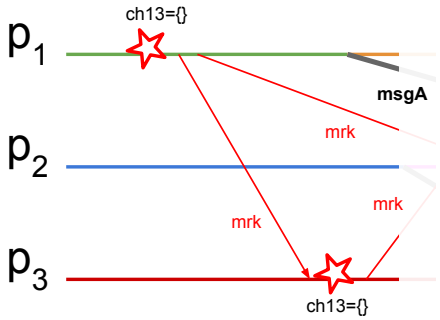


note: p3 receives the marker before p2

Chandy-Lamport Alg. Example (III)

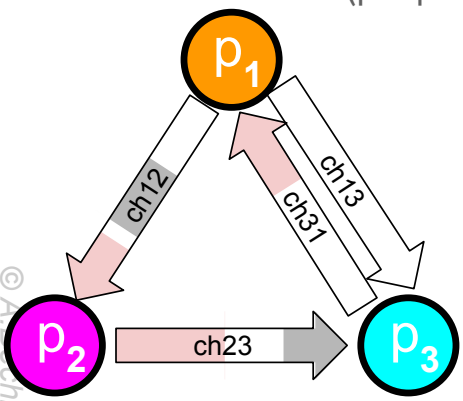
p_3 records its state (red), and sends mrk to p_1. Meanwhile,
p_1 sends basic msgA to p_2
and changes state (➔orange). Then…



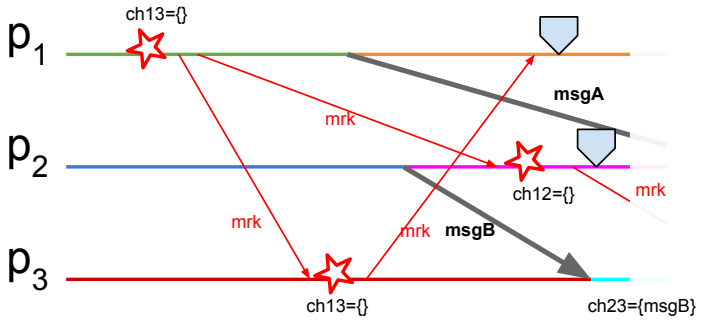


Chandy-Lamport Alg. Example (IV)

p_2 sends msgB to p_3, changes state (➔purple); then, it receives mrk from p_1,
records its state (purple), sends mrk to p_3, and terminates its local snapshot.
p_1 receives mrk from p_3, and terminates.
p_3 receives msgB from p_2, and changes state (➔cyan).



(i processi "cambiano colore" all'invio o alla ricezione di un BasicMsg)

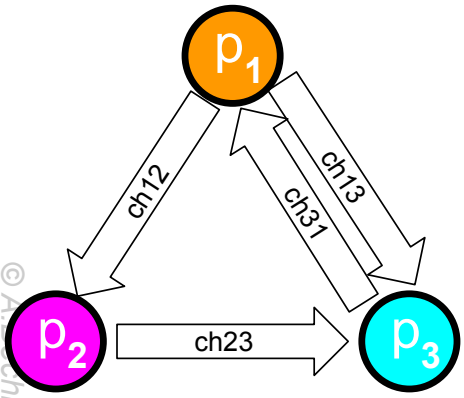


La stellina indica il momento in cui il nodo (che esegue un determinato processo) "capisce" di dover eseguire il local snapshot usando la procedure onMarker()

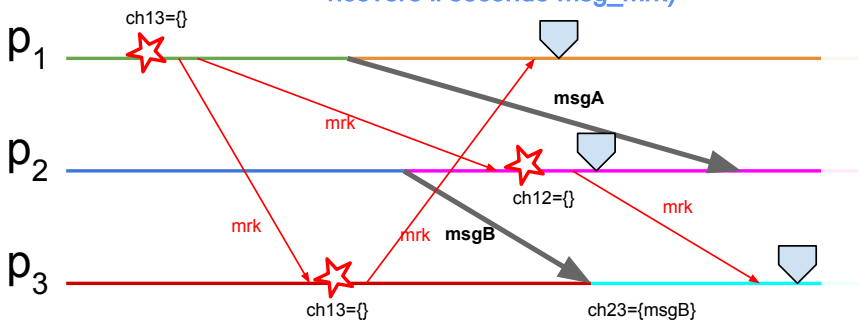
Chandy-Lamport Alg. Example (V)

non verrà quindi salvato (in questo snapshot)

p_2 receives basic msgA from p_1, but its local snapshot is already terminated;
p_3 receives mrk from p_2, and terminates.



Il msg_23 viene invece salvato, in quanto arriva a p3 prima che il suo local snapshot termini.
(p3 non è ancora terminato perché deve ancora ricevere il secondo msg_mrk)



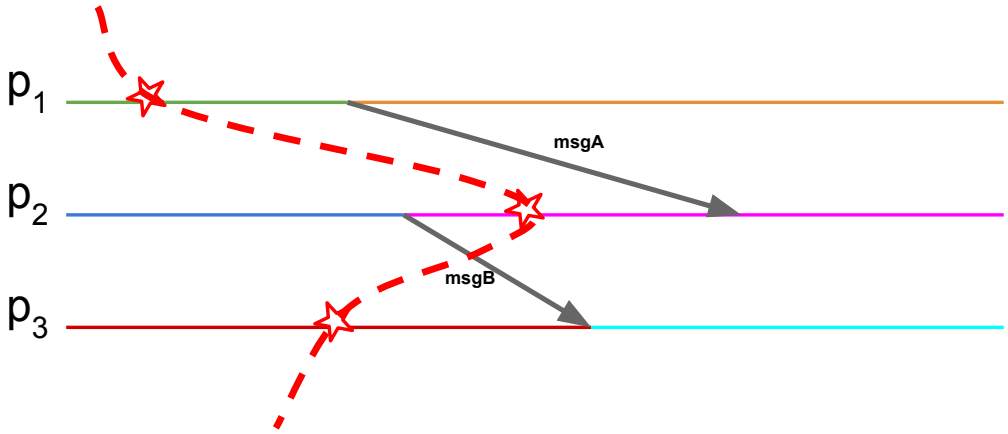
NB: il processo viene salvato al momento della prima ricezione di un msg_mrk.

It is a consistent cut because in this cut, every receive event has a corresponding send event that also belongs to the cut (this is simply the definition).
The concept is that arrows can only extend outwards from the cut; it's not possible for an arrow to enter the cut if its starting event is outside of the cut.

Chandy-Lamport Alg. Example (VI)

The obtained snapshot corresponds to the consistent cut shown below;
anyway, the overall system has not experienced such a state in any specific point in (global) time...

p_1
p_2
p_3
$ch_{12} = \{\}$
$ch_{13} = \{\}$
$ch_{23} = \{msgB\}$
$ch_{31} = \{\}$



We need to formally prove the correctness of this algorithm, where correctness implies that a snapshot corresponds to a consistent cut.

Chandy-Lamport Alg. Correctness

Correctness means: a snapshot corresponds to a *consistent cut*, i.e.:

For any two events a, b such that $a \rightarrow b$, if b is pre-snapshot, so is a.

(esempi di eventi: l'invio o la ricezione di un messaggio)

Proof:

1) a and b on same process: trivial
2) a in p_i, b in p_j, $a = send(m_x)$, $b = receive(m_x)$
   In this case, as b is pre-snapshot, p_j has not received mrk yet at b.
   Because of FIFO channels, p_i has not sent mkr yet at a,
   so a is pre-snapshot as well. □

The FIFO hypothesis is crucial!

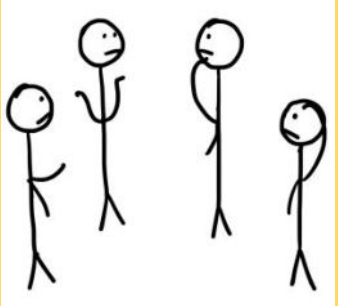
Se (per ipotesi) 'b' rientra nel cut, questo implica che il processo p_j non ha ancora ricevuto msg_mrk da p_i. Poichè i canali sono FIFO, se p_i non ha ancora inviato msg_mrk, siamo certi che a sia stato inviato prima.

It's important to note that this proof relies on the assumption that channels are FIFO. Without this FIFO property, we cannot guarantee that p_i has not sent the marker at event 'a', thus 'a' would not be considered a pre-snapshot event.

Pause for Thought

Correctness Depends on Hypotheses !

Therefore, whenever you design a distributed algorithm, the first thing to consider are your assumptions about the underlying system. These assumptions define the conditions under which your algorithm will be correct. Without clear assumptions, you cannot ensure the correctness of your algorithm under varying conditions.



Removing FIFO Hypothesis...

The FIFO property is crucial for the algorithm correctness.
How to arrange an algorithm to avoid recurring to FIFO?

we should

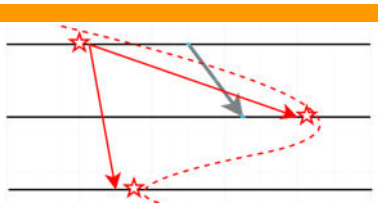
Avoid that any message *sent after* a local snapshot will be *received before* the local snapshot at the receiving side.*

This means that there is no possibility of having arrows going inside the consistent cuts.

Possible solution: *piggybacking*, letting a (basic) message carry information on the state of the sending process.
The receiver *will possibly delay the actual receive*, just performing the snapshot before it.

This information is inserted into all the messages that are part of the basic algorithm. Specifically, in the message, we will indicate if the sender has already taken the snapshot.

* questo perché voglio evitare:



PERÒ SECONDO ME, QUESTO PROBLEMA VIENE GIÀ EVITATO CON IL CODICE ATTUALE DEL CHANDY-LAMPORT... in particolare dall' if nella onBasicMsg()

Lai-Yang Algorithm (I)

adotta tale trucco

This algorithm adopts such a trick to get consistent snapshots, no matter the ordering in message delivery.

Yet another issue: how to obtain channel states?
It is sufficient to keep the log of outgoing/ingoing messages for any channel at any process;
then, the state for channel c_{ij} from p_i to p_j can be computed as

$$state(c_{ij}) = sent\_msgs_i[j] \setminus received\_msg_j[i]$$

But of course, this means that if you need to collect this information, you may end up with a massive amount of data to gather for determining the channel state. For instance, you might eventually find that these two sets are exactly the same, and the channel state is an empty set. This could pose a problem.

! Another problem will become clear later on.

MI SEMBRA DI CAPIRE (dal codice) CHE I msg_mrk ORA NON CI SONO PIÙ e che ora si usano solo i basicMsg.
Per esserne sicuro vorrei chiederlo al prof...

Lai-Yang Algorithm (II)

To perform the local snapshot:

```
procedure takeSnapshot()
    marked = true
    <record local_state considering also all logged
     outgoing and incoming messages >
```

In this case, messages belonging to the basic algorithms are responsible for carrying information about the local state. Instead of using the ordinary send, we will use what we call wrappedSend.

Any process p sending out **any** msg m from ch. c:

```
procedure wrappedSend(channel c, msg m)
    m.color = green if not marked else red
    send(c,m)
    log(c,m) // keep record of outgoing msgs
```

Any process p receiving **any** msg m on ch. c:

```
procedure wrappedReceive(channel c, msg m)
    if msg.color == red then
        takeSnapshot()
    receive(c,m)
    log(c,m) // keep record of incoming msgs
```

"if the sender has already executed a snapshot, then I will execute it as well before receiving the msg"

- wrappedSend(·)
The message M is also associated with a particular attribute, which we call "color." If the sender is not marked, "color" is set to green. Otherwise, it is marked as red.
After the message is sent, we need to keep a record of any outgoing message, so we must log this outgoing message.

- wrappedReceive(·)
The color attribute of the message (which is the piggyback information) is checked. In this case, a local snapshot will be taken, and then the ordinary receive operation will be performed.
Again, we need to keep a record of this incoming message as well.

So now we can see that one additional problem may arise. For example, what if one particular node is reached by a message not immediately, or rather, after an indefinite amount of time? Depending on the communication pattern between the pairs of processes belonging to the application, the finalization of the snapshot may take a lot of time.
This is because, even if a channel exists between two nodes, depending on the specific application, maybe no message will be sent over the channel for a long period of time. Or maybe, depending on the application, no message will be sent over the channel at all.
This delay in message transmission can significantly postpone the finalization of the snapshot, which is a problem.

Lai-Yang Algorithm (III)

The global snapshot will be computed upon the collection of all the local snapshots.

Further complication: in case of no outgoing basic msg after the first snapshot, we should wait all the other processes take their own when they please...

Possible solution: introduction of a **"trigger" control msg** to be sent through all outgoing channels, just after taking a local snapshot. This will "urge" the completion of the algorithm.

subito dopo

We introduce an additional control message to ensure that all nodes of the system will be reached within a reasonable time, to finalize the snapshot.

Now, the information about the snapshot is distributed across all the nodes. To analyze it, we need to collect this information from each node in the system.
We can establish a communication infrastructure that allows one node to gather information from all the other nodes and disseminate information back to them. This involves using broadcast for spreading information and convergecast for collecting information. Convergecast means that the information sent by all nodes is directed to a single node, which we can refer to as the "sink" node.

sink node := è un nodo designato in una rete che funge da punto di raccolta per i dati. Questo nodo riceve informazioni da altri nodi della rete

Performing Broadcast/Convergecast

Broadcast e Convergecast sono due operazioni fondamentali negli algoritmi distribuiti, spesso utilizzate per la comunicazione e il coordinamento tra nodi in una rete distribuita.
In un ambiente distribuito, dove le informazioni devono essere scambiate tra diversi nodi per raggiungere un obiettivo comune, queste operazioni consentono di diffondere dati (Broadcast) e di raccoglierli (Convergecast) in modo efficiente e affidabile.

Cos'è un albero di copertura?
Un albero di copertura di un grafo connesso è un sottoinsieme degli archi del grafo che collega tutti i nodi senza formare cicli e include tutti i nodi del grafo originale.

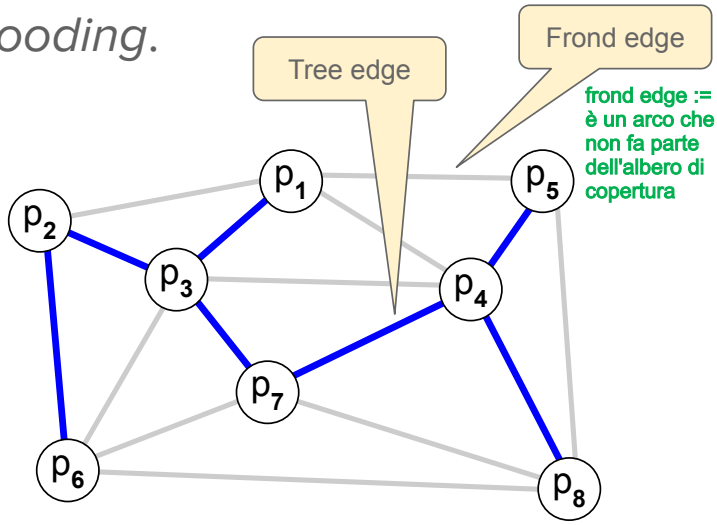
albero di copertura

Building a Spanning Tree

Cos'è il Flooding?
Il flooding è un processo in cui un messaggio viene inviato da un nodo sorgente a tutti i suoi vicini, che a loro volta inoltrano il messaggio a tutti i loro vicini, e così via, finché il messaggio non ha raggiunto ogni nodo nella rete. In altre parole, il messaggio si "allaga" attraverso la rete.

A trivial approach to broadcasting is *flooding*.
But what if the initiator wants back some information from each process?

Let's build up a **spanning tree**, to be used later for broadcast/convergecast.



We assume here to work with undirected, FIFO channels (and connected network).

"un albero è un grafo senza cicli"

On Spanning Trees

- Every connected (and undirected) graph *G* with *n* nodes and *E* edges has *at least* one spanning tree.
- All the possible spanning trees for *G* have *n* vertices and *n-1* edges. un albero di copertura DEVE avere esattamente n−1 archi
- A spanning tree is *minimally connected*, i.e. the removal of an edge makes it disconnected.
- A spanning tree is *maximally acyclic*, i.e. adding one edge always creates a loop.

L'algoritmo di Kruskal è un algoritmo utilizzato per trovare l'albero di copertura minimo (MST) in un grafo connesso non orientato. Funziona ordinando tutti gli archi del grafo in base al peso crescente e aggiungendo gli archi uno alla volta all'MST, assicurandosi di non formare cicli.

"Distributed" Spanning Tree Info

The structure of the spanning tree can be described keeping at each process information about:

1. The process that is the parent (variable **`parent`**)
2. A list with the children (list **`children`**)

The structure can be built in a distributed fashion with a *traversal* algorithm:
"forward" msgs are propagated ahead to explore the network "il primo che ti chiede di essere suo figlio sarà tuo padre"
- the parent of a process is the one the first forward msg is received from;
"backward" msgs keep collected info and state parent-child relationships.
contengono le informazioni raccolte e lo stato delle relazioni genitore-figlio

The initiator (the tree root) has just to know its own children, nothing else.

We have two different phases for setting up the spanning tree.

Once all the nodes have been reached, we can implement another phase with backward messages. These backward messages gather the collected information and maintain the parent-child relationships established during the information dissemination phase.

A Simple Spanning Tree Algorithm

- children is a list.
- expected is a numeric variable that indicates how many replies-back I am expecting.
- bck() sends a 'backward' type message to the parent of the current node.

```
procedure startSpanning()
    parent=pid //pid: local proc. index
    children=empty; expected = n_neighbors
    foreach p in neighbors send(p, fwd)
```

We recognize the root process by the fact that it has itself as its parent.

First action for the initiator:

Any process *p* receiving a msg **<fwd>**:

Any process *p* receiving a msg **<bck>**:

```
procedure onForward(proc sender)
    if parent == ⊥ then  //first fwd received (if the parent is not yet set)
        parent=sender; children=empty; expected=n_neighbors-1
        if expected == 0 then //reached a "dead end" (it is a leaf)
            send(parent, bck(list( (pid, myval) ))
        else
            foreach p in neighbors\sender send(p, fwd)
    else send(sender, bck(empty)) //ie "I'm not your child"
```

I respond by sending my pid, and if we want to collect some values, I put it in the field myval

```
procedure onBackward(proc sender, msg m)
    expected -= 1; val_list.append(m.val_list)
    if m.val_list not empty then children.append(sender)
    if expected == 0 then //no more bck from children
        val_list.append( (pid,myval) )
        if parent != pid then send(parent, bck(val_list))
        else <termination; use val_list>
```

In short, if I have finished collecting data on my sub-tree, I also save my data and send it to my parent. Finally, if I am the root node, as the last operation, I use the collected information (tree structure and node values).

The idea is to first explore the graph to set up the parent variable for all nodes.
In the second phase, we determine the children of each node.

▲ What we obtain is just one of the possible spanning trees. By rerunning the algorithm, you may build up different styles of spanning trees because of possible races between forward messages.

What about Complexity?

Perchè nei Frond Edges vengono inviati 4 msg?
Perchè, per come è scritto l'algoritmo, prima un nodo A chiederà a B di essere suo figlio (richiesta che verrà rifiutata), successivamente anche B chiederà ad A di essere suo figlio (anche questa rifiutata).
Abbiamo quindi 2 fwd (richieste) e 2 bck (risposte).
Nei Tree Edges invece, poichè la richiesta di A viene accettata, B questa volta non invia una nuova richiesta.

Message complexity:

- # of *tree* edges: $n-1$; 1 fwd + 1 bck per tree edge ➔ $2(n-1)$
- # of fronds: $E-(n-1)$; 2 fwd +2 bck per frond ➔ $4E - 4(n-1)$
- In total, $4E - 2(n-1)$ so msg complexity is $O(E)$ with E = # edges. # edges of the original graph, NOT of the spanning tree
- But E in a connected graph is in the interval $[(n-1), n(n-1)/2]$,
- so in the worst case the msg complexity is also $O(n^2)$

Time complexity:

- all the processes are reached, and a message comes back to the initiator, so $2D$ hops, with D = *network diameter.*
- So the time complexity is $O(D)$.

network diameter (D) := is the longest path that connects two distinct nodes within the graph.

Pause for Thought

Nondeterministic Behaviors

The algorithm we just saw is NONDETERMINISTIC because, even knowing the algorithm and the graph on which we will apply it, the result is not predictable. Because the final result will depend on the relative speeds of the messages that are sent over the network.

Mutual Exclusion: The Distributed Way

Nel contesto dei sistemi distribuiti, la mutua esclusione è un problema cruciale che riguarda la gestione dell'accesso a risorse condivise. In un sistema distribuito, diversi processi possono trovarsi su nodi separati, e la necessità di sincronizzare l'accesso a risorse critiche senza causare conflitti diventa fondamentale.
A differenza dei sistemi centralizzati, dove un singolo punto di controllo può gestire la mutua esclusione, nei sistemi distribuiti la coordinazione deve essere ottenuta attraverso la comunicazione tra nodi indipendenti.

Problem Statement

Informally: Only one process *at a time* can "access a resource."
What does it means "at a time" ?

Actual access to the resource: through a set of operations said *critical section*;
a CS is ideally delimited by the operations *req_access()* and *rel_access()*

Properties:

ME1 - safety - at most one process may execute in the critical section

ME2 - liveness - requests to enter/exit the critical section *eventually* (prima o poi) succeed

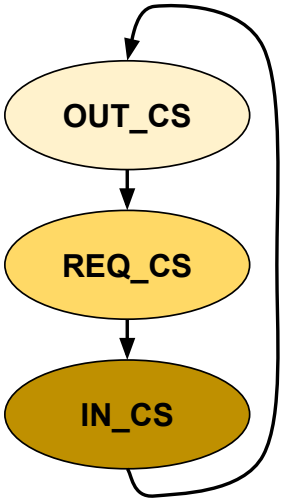
Process States wrt Critical Sections

As regards the relationship with a CS,
a process can be in three different states:

OUT_CS - not interested in executing CS

REQ_CS - started *req_access()*,
not yet executing CS

IN_CS - executing CS



A variable in each process can be dedicated to encode such a state.

Types of Distributed Mutex Algorithms

They are necessarily **decentralized** algorithms: anyone can be the initiator.

We may have many different initiators. We don't know which node will be the initiator; any node can be an initiator (it can request access to the shared resource).

- **Individual Permissions**: Any single process grants a permission just on its own behalf.

i processi collaborano tra loro per gestire l'accesso alla sezione critica. Ogni processo non agisce in modo completamente autonomo, ma interagisce con gli altri processi per coordinare l'accesso alla risorsa condivisa.

- **Token-based** - a unique token is shared across all processes, and CS is executed only when the token is held.
- **Arbiter-Permissions**: there exists at least one process that resolves conflicts between any couple of requesting processes; permissions are returned at CS exit.
  (general case: ~~"Quorum-based approach"~~) (we will not talk about it)

Ricart-Agrawala Algorithm - Basics

belongs to the "Individual Permissions" category

Classically, the execution of CS requires

- An enter protocol - **`req_access()`**
- An exit protocol - **`rel_access()`**

Ideas for dealing with "individual permissions":

- To enter CS, a process must request the permit to each other process, and wait for all the relative "OKs."
- Possible request conflicts are resolved using totally-ordered timestamps: *the earliest request gets the priority.*
  Remember: $(ts_a, i) < (ts_b, j)$ *iff* (if and only if) $(ts_a < ts_b) \vee ((ts_a = ts_b) \wedge (i < j))$ (OR, AND)

Lamport timestamps...

This is just a formal way to verify if the timestamp of A is strictly less than the timestamp of B, and consequently, A will have priority and will be chosen as the winner. If the timestamps of A and B are exactly the same, we then look at the second part of the timestamp, which is the PID associated with the timestamp. In this scenario, the process with the lower PID will be chosen.

In this case, since we need all requests to be totally ordered, the timestamps consist of two parts: the ordinary Lamport timestamp and the PID associated with the process issuing the request.

Ricart-Agrawala Algorithm

- req_set := the set of all the other processes to which the request should be sent

```
procedure req_access() // pid: local pid
   mystate = REQ_CS
   pending_oks = N-1 //all the others
   myreq_lt = myclock+1 // l.timest. for my req
   req.ts = (myreq_lt, pid) //set msg timest.
   foreach p in req_set send(p, req)
   wait_until (pending_oks == 0 )
   mystate = IN_CS // then, CS
```

Perché "myclock+1" ?
Boh, lui a lezione non lo dice... CREDO sia una conseguenza del fatto che qui viene usato il Lamport Timestamp

```
procedure rel_access()
   mystate = OUT_CS
   foreach p in req_delayed send(p, ok)
   req_delayed = empty
```

Enter protocol Exit protocol

Deal with: OK, REQ

sent by a process to ask to enter in the critical section

```
procedure onReq(proc sender, msg m)
   myclock = max(myclock, m.ts)
 * my_prior =(mystate != OUT_CS) and (myreq_lt,pid)<m.ts
   if not my_prior then send(sender, ok)
   else req_delayed.append(sender)
```

Here, only the part of the timestamp containing the Lamport timestamp will be considered.

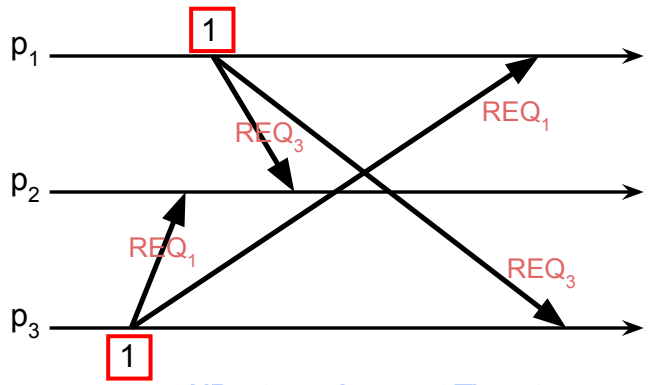
my_prior := booleano che indica se "io ho priorità" rispetto alla richiesta ricevuta

```
procedure onOk(proc sender, msg m)
   pending_oks = pending_oks-1
```

* There, I must consider whether there is a conflict between myself and the incoming request. This is possible only if I am currently in a critical section or if I am requesting to enter a critical section.

Ricart-Agrawala Algorithm - Example

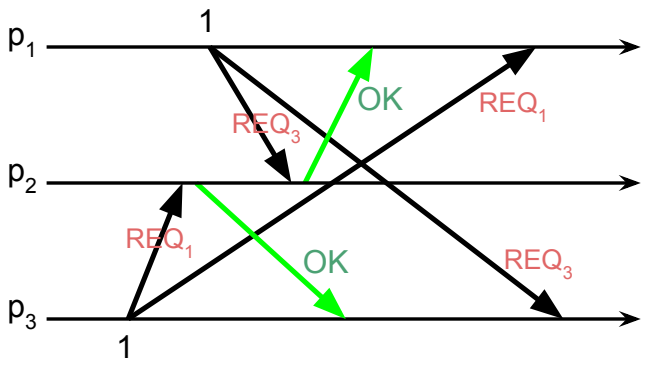
In the example shown here, both p_1 and p_3 try to enter the CS "at the same time"



NB: stesso Lamport Timestamp

Ricart-Agrawala Algorithm - Example

In the example shown here, both p_1 and p_3 try to enter the CS "at the same time"

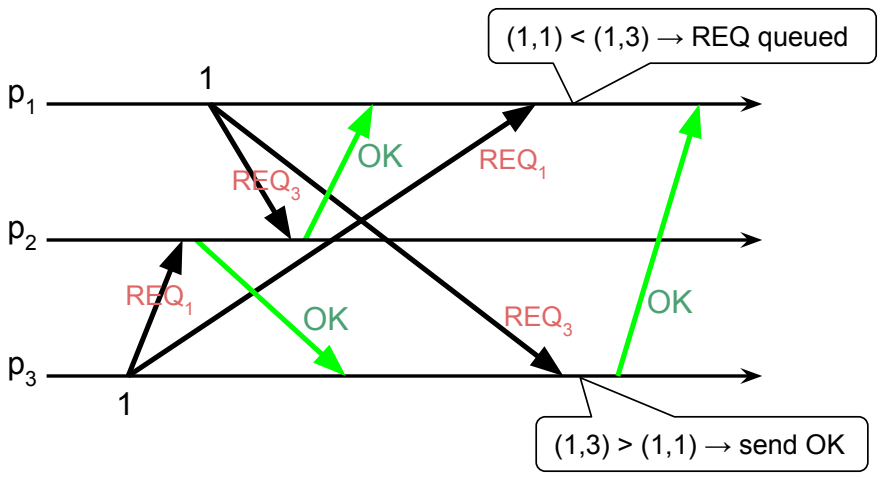


p2 will respond OK to both p1 and p3 because p2 is not interested in entering the critical section and is not currently within a critical section.

Now, you see that p1 received the permit from p2, but it is not allowed to proceed until the permit from p3 is also received.

Ricart-Agrawala Algorithm - Example

In the example shown here, both p_1 and p_3 try to enter the CS "at the same time"



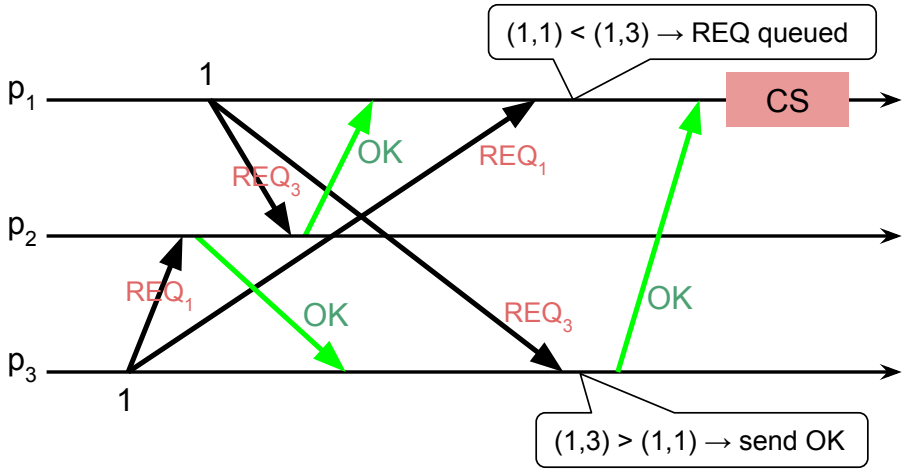
P1 will have to check whether its own request and the incoming requests received are conflicting or not, and of course they are. Consequently, the process has to decide whether it holds the priority for the execution of the critical section or not.

In this case, P1 will have the priority over P3 to execute the critical section. So the request coming from P3 will be queued. Similarly, P3 will perform the same comparison, but for it, the priority is not for the local process. So P3 will send out the OK message to P1. This OK message will reach P1, allowing P1 to execute the critical section.

We can see that the two concurrent requests from P1 and P3 will result in the serialization of the execution of the two critical sections. This serialization is determined by the priority based on the timestamps associated with the requests.

Ricart-Agrawala Algorithm - Example

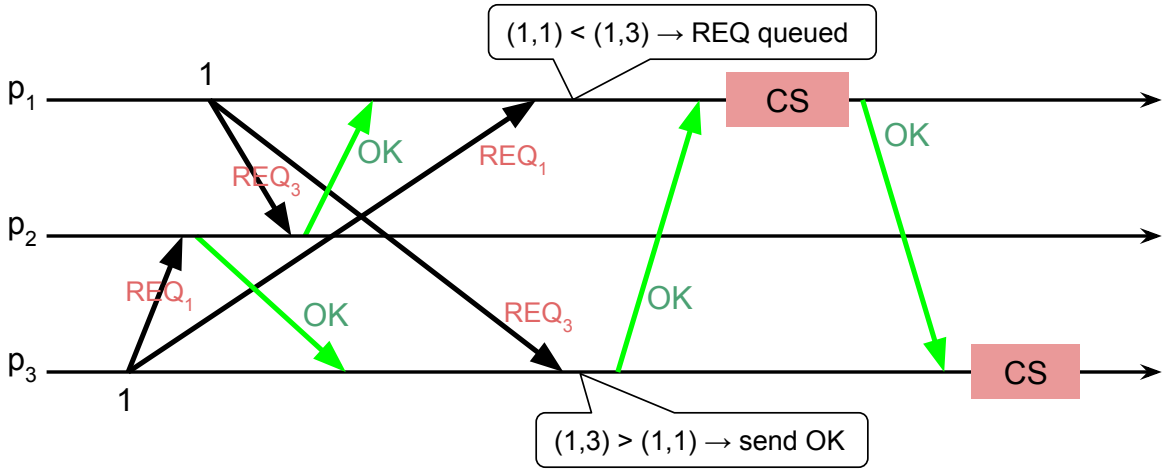
In the example shown here, both p_1 and p_3 try to enter the CS "at the same time"



Ricart-Agrawala Algorithm - Example

In the example shown here, both p_1 and p_3 try to enter the CS "at the same time"

As the critical section is over, it's up to P1 to execute the exit protocol. In the exit protocol, P1 has to send back the (delayed) OK to P3, allowing P3 to enter the critical section.



Ricart-Agrawala Algorithm: Complexity

Message complexity:

The use of a CS for one process involves the exchange of $n-1$ REQs and $n-1$ OKs, so $2(n-1)$ in total.

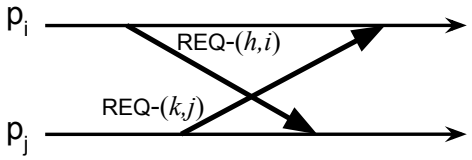
Time complexity:

all the processes have to be reached, thus in the worst case it depends on the longest possible path (# of hops) between two generic processes in the network, i.e. the *network diameter* D. The time complexity is $O(D)$.

Ricart-Agrawala Alg. Correctness (I)

The alg. satisfies ME1, i.e. at most one process may execute in CS.

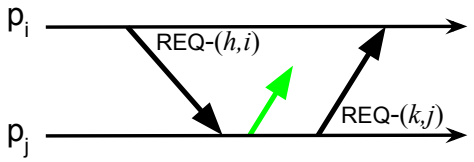
Proof: by contradiction; let suppose both p_i and p_j are in state IN_CS. It follows that each of them has sent REQ to the other (with ts (h,i) and (k,j), respectively), and got OK back. Two scenarios are possible:



1) each process has sent its REQ before receiving the other's REQ.
   Let's assume e.g. $(h,i) < (k,j)$, thus p_j was in the condition to send OK to p_i; on the other hand, under the same condition, p_i didn't send OK to p_j, thus p_j cannot be in CS $\Rightarrow$ contradiction
2) *... (cont.)*

Ricart-Agrawala Alg. Correctness (II)

1) ...
2) One process, say p_j, has sent the other (say p_i) its OK before sending its REQ.
When p_j receives REQ, it performs $myclock_{(j)} = max(myclock_{(j)}, h)$, yielding $myclock_{(j)} \geq h$.
When subsequently p_j send REQ, the relative ts is set to $k = myclock_{(j)} + 1 > h$.
As this REQ is received by p_i, its state is not OUT_CS, and $(h,i) < (k,j)$ and thus p_i is not in the condition to send back OK to p_j, which thus cannot enter CS $\Rightarrow$ contradiction. □



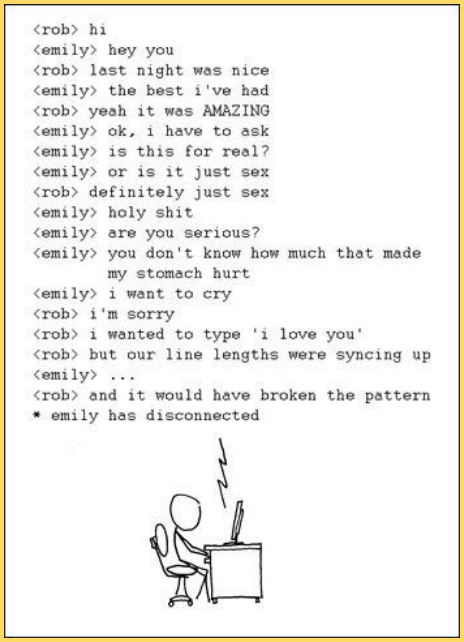
ME2 (liveness) can be proven by showing absence of deadlock and starvation.

Pause for Thought

Adhere to Patterns

(enter/exit protocols)

To ensure that everything runs properly, we must adhere to patterns. The enter and exit protocols have to be executed at the appropriate times. Otherwise, we cannot guarantee that the basic properties will be satisfied.



Token Ring Mutex

belongs to the "Token-based" category

W.r.t. the algorithm's purposes, processes must be organized in a directed ring topology - an "overlay network."

The permit to execute CS is a **token msg** that circulates the ring.

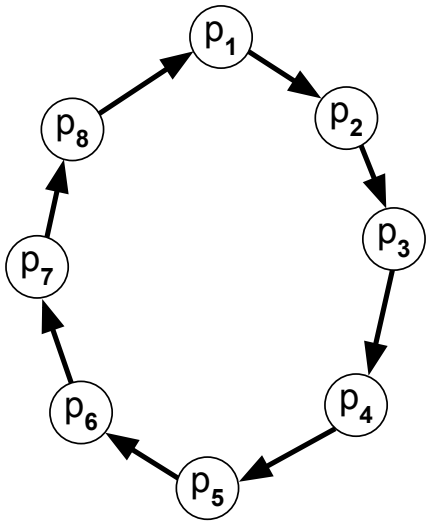
The token is placed in a message, which circulates around the ring.

`req_access()` means to set the state to REQ_CS

`onToken()`: if mystate = REQ_CS, then execute CS, otherwise send the token to next process.

`rel_access()` means to set my state to OUT_CS, and then send the token to the next process.

Here, state IN_CS is practically meaningless.



Token Ring Mutex: Correctness

Informally:

ME1 follows from the unicity of the token.

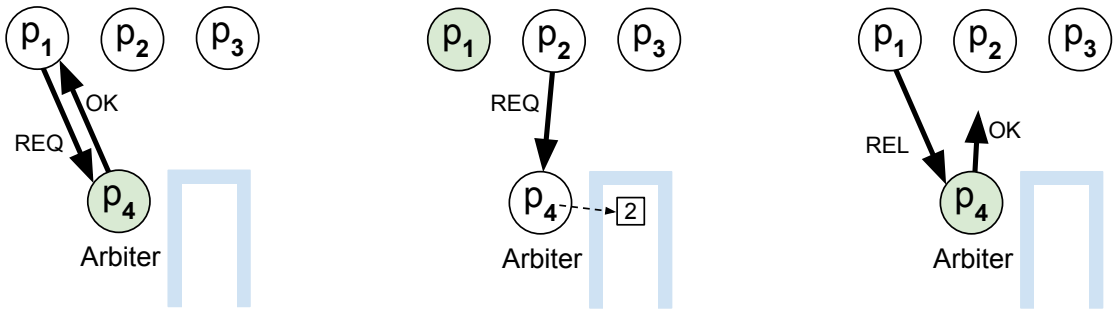
ME2 follows from the ring topology and the token passing rules, which assure that one token will *eventually* reach each node.

Plain Centralized Mutex

belongs to the "Arbiter-Permissions" category

A simple solution makes use of one arbiter process, which handles one single permit that, after each use, has to be sent back, or “released.”

The arbiter usually keeps a queue for processes waiting to access CS.



Failures Right Around The Corner

Nel contesto dei sistemi distribuiti, i fallimenti sono sempre dietro l'angolo. I nodi del sistema possono andare in crash, oppure potrebbero sorgere problemi sulle linee di comunicazione.
La sfida diventa allora sviluppare algoritmi distribuiti in grado di far funzionare il sistema correttamente anche in presenza di fallimenti.

Classifying Failures

Class of Failure	Affects	Description
Fail-stop	process	Process halts and remains halted. Others may detect this state.
Crash	process	Ditto, but others may not be able to detect this state.
Omission	channel	Message got lost in the way
Send-omission	process	Message "sent", but not inserted in out-buffer
Receive-omission	process	Message arrived to in-buffer, but not received
Byzantine	proc/channel	Any arbitrary behavior! Questo è il tipo di fallimento più difficile da gestire, poiché il processo o il canale può comportarsi in modo completamente imprevedibile, inviando messaggi falsi o corrompendo i dati.

Before moving on to discuss algorithms that work in the case of failures, I want to introduce a specific model of execution known as the synchronous model of computation. In this model, we account for a sequence of time slots according to the global time. Each operation by all processes has to be performed within a time slot. This approach allows us to work with evolving time as if it were a discrete variable.
In this case, we have a discrete notion of time. This means that algorithms under this model can be developed in a different way.

But why am I presenting the synchronous model at this point in my exposition of different algorithms?
Because, of course, this model is not a realistic one. However, whenever I want to deal with faults, I can imagine relying on upper bounds of reply times. If I consider upper bounds of reply times, I will be able to employ timeouts to check for possible faults.

Towards Synchronous Models

A *synchronous model* of computation accounts (prevede una sequenza) for a sequence of fixed time-slots according to the global time, and each operation has to be performed at (da ogni) each process *within a timeslot*. This assumption let us deal with a "discrete" notion of time, developing algorithms in a different way.

This "bare" model is clearly unrealistic; anyway, in dealing with faulty settings, we can imagine to count on *upper bounds on replying times*.
In this way, we can employ **timeouts** to check for possible faults.

Often, the system model letting us legally use timeouts is said *synchronous*.

Leader Election Algorithms

Leader Election: Problem Statement

Informally: All the processes choose one of them (or out a set of candidates) to play a special role in the system (coordination/control).
Leader election is a form of ***symmetry-breaking*** in distributed systems.
(because we will have one process that behaves differently from all the others)

Each process p_i is associated with a **unique** "UID_i"; the process to be elected is the one (in the set of candidates) with an *extremal* (max, min) UID.

Possible states for a process during an election run: participant/nonparticipant.

Required local state vars:
- bool (or UID type) $leader_i$, to indicate "I'm the leader" (or the UID of the leader)
- (possibly) $done_i$, to indicate that a leader has been elected.

Leader Election: Properties

LE1 - safety -

at most one (non-crashed) process is elected;
$elected_i$ and $done_i$ are stable;
termination implies that one process has been elected.

LE2 - liveness - (termination)

a process is eventually elected;
the election is eventually known by all the processes (non-crashed ones).

Note: it has been proven that in an “Anonymous” system, with no personal ids and a regular network topology, the leader election problem has no solution.

Chang and Robert’s Election (I)

An *unidirectional ring topology* is assumed (not necessarily FIFO channels).

Two phases:

1. **Leader identification**
2. **Leader announcement**

Possibly, multiple concurrent initiators

This is because each process acts independently of the others, meaning that we might have two nodes concurrently starting the election algorithm separately at the same time.

! process with the highest UID wins the election

Each process sends its own UID on the ring, and stops any incoming UID lower than its own.
The only UID that can pass all the way around the ring is the leader’s one.

Two types of exchanged messages: ELECTION and LEADER.

- election messages, is used to determine who will be the leader
- leader messages, is used to spread the news about who has been elected as the leader

Chang and Robert's Election (II)

The "election" message contains a UID slot, which is set to myUID. It is important to note that the recipient is not specified because, in a ring topology, a process can send a message only to its next neighbor.

For a process (initiator) to start election:

```
procedure startElection()
   mystate = participant
   election.UID = myUID
   send(election)
```

What to do upon receiving ELECTION msg:

```
procedure onElection(msg m)
   if   myUID < m.UID then
      mystate = participant
      send(m)
   elif myUID > m.UID then
      if mystate == nonparticipant
         mystate = participant
         election.UID = myUID
         send(election)
   elif myUID == m.UID then
      leader.UID = myUID
      send(leader); elected = true
```

poichè gli UID sono unici, se trovo un UID uguale al mio vuol dire che ho vinto l'elezione

To spread the word about the elected process via LEADER msg:

function triggered when a leader message is received:

```
procedure onLeader(msg m)
   leader = m.UID;  done = true
   if myUID != m.UID then
      elected = false
      send(m)
```


- nonparticipant:
All'inizio, tutti i processi si trovano nello stato di nonparticipant.
Quando un processo riceve un messaggio di elezione (ELECTION msg) e si trova in stato nonparticipant, questo stato indica che il processo non ha avviato o partecipato a nessun'altra elezione. Pertanto, il processo diventa un partecipante all'elezione, cambia il suo stato in participant e inoltra il messaggio di elezione.

- participant:
Un processo entra in questo stato quando inizia un'elezione o quando riceve un messaggio di elezione mentre si trovava nello stato nonparticipant.

Questi stati, servono per evitare che un processo partecipi più volte alla stessa elezione (in caso di più initiator nello stesso momento)

C&R's Election: Complexity

Message cost: in any case, n LEADER msgs are used.

Best case: one single initiator, with highest UID ⟶ n ELECTION msgs

Worst case: all initiate at the same time, and processes on the ring are ordered with decreasing UIDs. The process with the highest UID genera n ELECTION msgs, the one with the second highest UID genera $n-1$, and so on; in total, $n(n+1)/2$ msgs: ⟶ $O(n^2)$

Average case (not proven here): $O(n \log n)$

Per comprendere n, n-1, ecc...
Basta pensare che qui è scritto in modo confuso.
Vedila così: il messaggio generato dal processo con UID più alto verrà trasmesso n volte, la prima volta dal processo stesso e le altre volte dagli altri nodi.

Time cost: Best case: one initiator, with highest UID, as before, $O(n)$.

Worst case: the only initiator is the process with the second highest UID, and follows the one with the highest; $(n-1)+n$ ELECTION msgs are used.

Se consideriamo il tempo di esecuzione dell'algoritmo su un nodo come 'molto breve', il costo temporale corrisponde al tempo necessario per far transitare tutti i messaggi. In questo algoritmo, i messaggi (di una stessa iniziativa di elezione) non viaggiano contemporaneamente ma uno dopo l'altro. Pertanto, il tempo totale necessario può essere stimato in base al numero di messaggi che devono essere trasmessi.

Election with Possible Crashes

Assumptions: Reliable delivery, synchronous system, i.e. an *upper bound* exists *on the msg latency*.

Processes can crash even during election turns, but process failures can be detected with *timeouts*.

premesse

Premises for the next algorithm:

- Each process knows what processes have other UIDs, and is also able to communicate with them.
- The leader has to be the process with the current highest UID: but this depends on what processes are currently up/down!

Very different from the algorithms seen so far because, previously, each process didn't know about the UID of all the other processes. Now, they do.

However, they do not know which processes are currently alive because they could have crashed.

The Bully Algorithm (I)

Types of msgs:

- ELECTION (to announce an election),
- ANSWER (to response to an election),
- LEADER (to announce the elected leader).

The election is typically triggered by any process that, by using timeouts, notices that *the current leader is no more alive*.

The process that knows to hold the highest UID in the group can just send LEADER msgs to the others to inform them that it is the leader.

The Bully Algorithm (II)

One process, to start the election, must send an ELECTION msg to those processes with higher UID.

I contact all the processes with a higher UID than mine because I don't know whether they are still alive or not.

Then, it awaits for their ANSWER msgs back.

If no answer arrives within time T, it takes itself as the leader, and sends LEADER msgs to the processes with lower UIDs; Otherwise, it waits for an additional T' for a LEADER msg to come from the new coordinator; in case the msg does not arrive, the process starts a new election round.

(because it means that all processes with UIDs higher than mine have crashed)

So all the others will communicate among themselves, exchanging messages, and at a certain point, the leader will be found. Once the leader is found, I just have to wait for an additional time to receive a leader message.

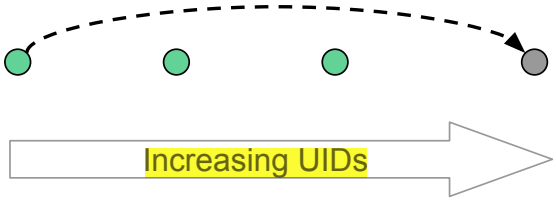
The Bully Algorithm (III)

As a process p_i receives a LEADER msg from a process with higher UID, it sets its variable *leader* to that identifier.

As a process p_i receives an ELECTION msg, it sends back an ANSWER and begins another election (unless it has not done this yet).

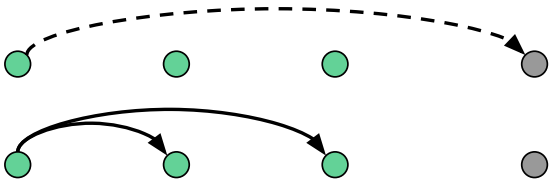
1. There's no need to check if my UID is higher because it definitely is (otherwise, I wouldn't have received the message).
2. In turn, I need to check if there are other nodes with UIDs higher than mine, which I do by sending them an election message.

The Bully Algorithm: Example



A process finds the current leader is down

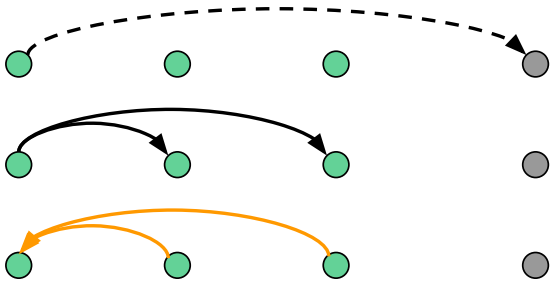
The Bully Algorithm: Example



A process finds the current leader is down

Msg ELECTION - “who is the leader?”

The Bully Algorithm: Example



A process finds the current leader is down

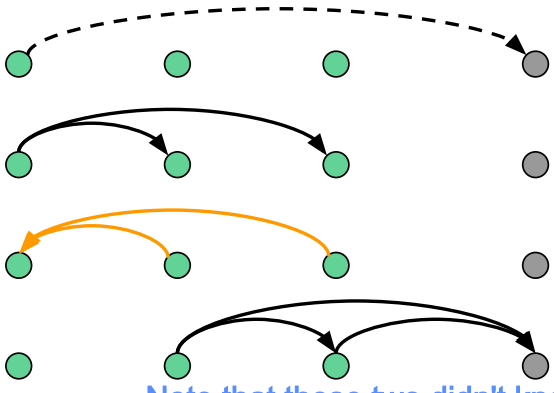
Msg ELECTION - “who is the leader?”

Msg ANSWER - “for sure not you!”

perché ANSWER non viene inviato anche al nodo grigio (dal momento che i due verdi non sanno ancora che è crashato)?
Perché ANSWER viene inviato solo in risposta ad un msg ELECTION.

Poi, magari, i nodi verdi, ricevendo un messaggio ELECTION, capiscono che il vecchio leader è crashato. Tuttavia, anche se possono dedurlo, invieranno comunque un messaggio ELECTION al nodo grigio. Lo fanno per vari motivi, tra cui il fatto che il nodo grigio potrebbe essere tornato online.

The Bully Algorithm: Example



A process finds the current leader is down

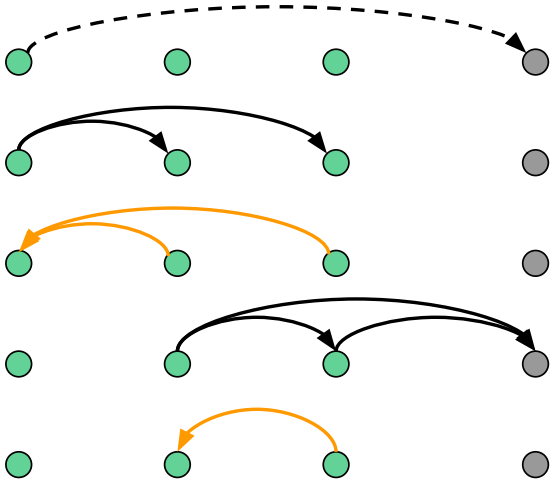
Msg ELECTION - “who is the leader?”

Msg ANSWER - “for sure not you!”

Msg ELECTION - “who is the leader?”

Note that these two didn't know that the last process was. So they also sent a message to the green process.

The Bully Algorithm: Example



A process finds the current leader is down

Msg ELECTION - “who is the leader?”

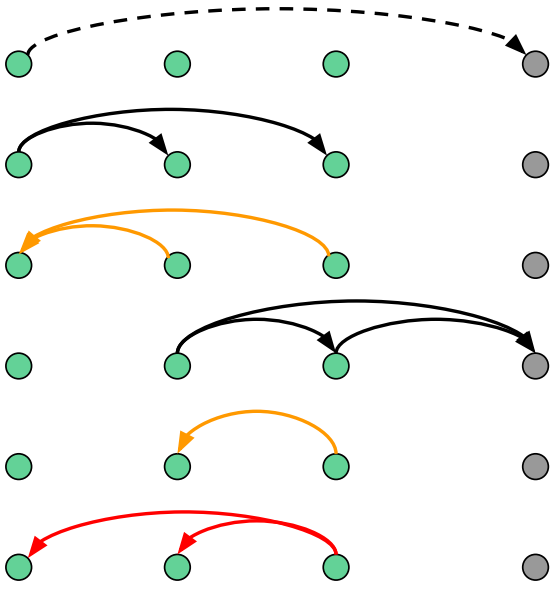
Msg ANSWER - “for sure not you!”

Msg ELECTION - “who is the leader?”

Msg ANSWER - “for sure not you!”

The last green process will know that it is the leader because it didn't receive any response message.

The Bully Algorithm: Example



A process finds the current leader is down

Msg ELECTION - “who is the leader?”

Msg ANSWER - “for sure not you!”

Msg ELECTION - “who is the leader?”

Msg ANSWER - “for sure not you!”

Msg LEADER - “I’m the leader!”

Now all the processes will know who the leader is

But... What about Bullism?

perché lo chiamiamo così? Supponiamo che un processo crashi e che poi rientri a far parte del gruppo, in questo modo non conosce gli UID degli altri e quindi quello che fa è iniziare una nuova election round, provando quindi a sostituire l'attuale leader.

When a process is started to replace a crashed process, it initiates a new election round.

If it holds the highest possible UID, it just decides it is the leader, and announces this to the others: no matter if the current leader is still up!

Such an “impolite” behavior inspired the name for the algorithm.

The Bully Algorithm: Properties

About LE1 (safety): at most one (non-crashed) process is elected;
In fact, it is not possible to have two leaders,
because the process with lower UID would discover the other,
and would defer leadership to it.

About LE2 (liveness): guaranteed by the assumed reliable delivery.

Message Complexity:
worst case - initiator with the *lowest* UID ➔ $O(n^2)$;
best case - initiator with the *highest* UID ➔ $n-1$ LEADER msgs, $O(n)$.

Multicast: What Multicast?

Il multicast è una tecnica di comunicazione utilizzata negli algoritmi distribuiti per inviare messaggi simultaneamente a un gruppo di destinatari.

Let's go on with another type of communication in the setting where processes may crash.

Group Communication

If I want to send a message from one process to an entire group, I will use a primitive called "multicast".

Group up processes, and *address groups* *instead of single processes*.

Relative primitives:

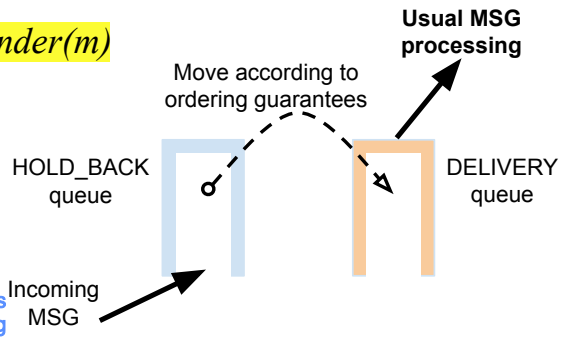
Xmulticast(group g, msg m) - **X** indicates the specific type, leggi giù...
according to different possible semantics about ordering guarantees

Xdeliver(msg m) - getting *m*, it must be possible to obtain *sender(m)*

Why “deliver” and not “receive”?
Possible reorderings of msgs at destination require an **hold-back queue**.

Since we want to implement this primitive with different semantics, we may need to manage the reordering of messages upon arrival at the destination. For this reason, instead of having a single receive queue and the typical input buffer, it would be prudent to maintain what is known as a hold-back queue.

Similar trick already seen in Lai-Yang alg.



Instead of having a single incoming queue, we will have two: the hold back queue and the delivery queue. The actual delivery of a message involves extracting it from the delivery queue. When a message arrives at its destination, it is initially kept in the hold back queue. A message is only moved from the hold back queue to the delivery queue when it is possible to do so according to the ordering guarantees associated with the specific semantics of the used multicast primitive.

Generally, there are many different multicast primitives. We denote it as X multicast towards a group G of a message M. The 'X' indicates the specific type of this primitive, allowing for various semantics regarding the ordering guarantees of the messages across different multicast primitives.

By 'different semantics,' we mean that the function Xmulticast(group G, msg M) can exhibit various behaviors related to the ordering of messages. These behaviors define how messages are ordered and delivered to the processes within the group.
For example, FIFO Multicast, Total Order Multicast, etc...

Basic Multicast

This is a type of multicast with no guarantee and no checks on the ordering of delivery.

For the time being, let us suppose processes may fail *only by crashing*.

The most straightforward implementation for a multicast:

Bmulticast(g,m) ➔ for each process p in g, *send*(p,m)

On *receive*(m) at p: *Bdeliver*(m) at p. Bdeliver(m) := il messaggio m viene consegnato al livello applicativo del processo p (ricorda che *deliver* qui significa *receive*)

> "Basic" multicast guarantees that a correct process will *eventually* deliver the message, as long as the multicaster does not crash.

So, here, the problem is the crash of the process during the operation of sending out all the messages in the loop.

Reliable Multicast

A reliable multicast (*Rmulticast*) satisfies the following properties:

Integrity - A correct process delivers a message m at most once (no duplicati)

Validity - If a correct process multicasts msg m, it will eventually deliver it
(this is self-validity! It's meaningful only along insieme a with the next property).

Agreement (a.k.a. *"Atomicity"*) - If a correct process delivers msg m, then all the other correct processes in *group*(m) will eventually deliver it.

"It may happen that either all the processes deliver the message or none of the processes deliver it."

Trivial Implementation of Rmulticast

Rmulticast can be simply implemented on top of *Bmulticast* in the following way:

- *Rmulticast* corresponds exactly to *Bmulticast*
- *Rdeliver* corresponds to: *Bdeliver* + (*only once*) *Bmulticast* of the same msg to the group; it is assumed that only one copy of the msg is actually delivered, and the (possible) others are discarded.

duplicate messages that may reach a recipient will be discarded.

Agreement comes from observing that, after the *Bdeliver*, each processes issues a *Bmulticast*: if a correct process does not *Rdeliver m*, this may happen only because no other process *Bdeliver*ed it either (in fact, there should be an HB relation from any other *Bdeliver* and the local *Rdeliver*)

Ogni nodo che riceve un messaggio esegue una Bmulticast dello stesso messaggio, il che significa che lo stesso messaggio verrà inviato nuovamente a tutti gli altri nodi. Tuttavia, questo comportamento è parte del meccanismo di ridondanza per assicurare che il messaggio sia consegnato correttamente, anche in presenza di fallimenti.

Tradeoff between stronger guarantee and msg complexity

We have to employ many messages to support these semantic guarantees. This creates a trade-off between stronger guarantees and the message complexity of the algorithm.

Other Multicast Semantics

We refer to a "correct process" because our statements apply to processes that did not crash.

FIFO: if on one correct process $Fmulticast(g,m_1)$ and then $Fmulticast(g,m_2)$ occur,

at any destination in g, m_1 is *Fdeliver*-ed before m_2

Causal: if, on correct processes, $COmulticast(g,m_1)$ H.B. $COmulticast(g,m_2)$, at any destination in g, m_1 is *COdeliver*-ed before m_2. Causal implies FIFO.

(FIFO mc is a special case of Causal mc)

Total: if a correct process *Tdeliver* m_1 first and m_2 later, the same order of delivering will be experienced by any process.

COmulticast is implemented using vector timestamps; messages are kept at destination inside the hold-back queue until the checks on their precedence relation with the local vector clock would allow the actual deliver operation.

These types of communications, such as multicast and others, are typically employed within the middleware layers. It is quite rare to use this type of communication directly in your distributed applications. This approach is generally employed to allow the middleware to implement specific guarantees for the applications built on top of it.