



# Fundamental Distributed Algorithms

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

a.bechini@ing.unipi.it

© A. Bechini 2020



## Outline

Introducing distributed algorithms and basic solutions for common problems

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

© A. Bechini 2020

# Classes of Algorithms

## Basic Alg. vs Control Algorithms



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

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



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

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



# Centralized vs Decentralized



In a given algorithm, a process is an **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.

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

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

# Wave Algorithms



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.

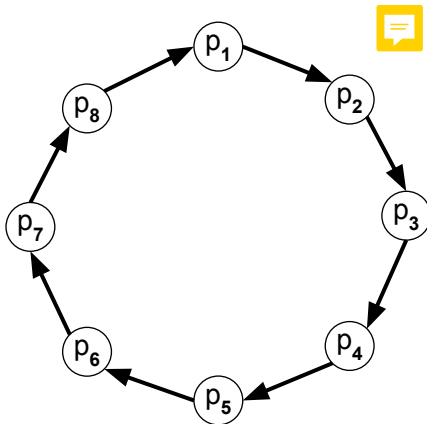


In a **wave algorithm**, a computation satisfies the following properties:

- 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  $p_i$

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

# A Trivial Wave Algorithm: Ring



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)
```

© A. Bechini 2020

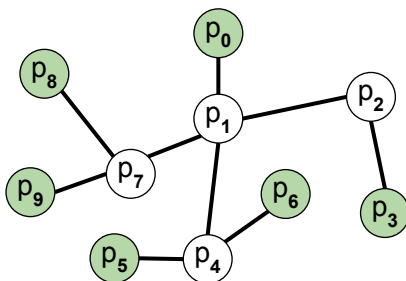
A. Bechini - UniPi

# 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)
```

© A. Bechini 2020

A. Bechini - UniPi



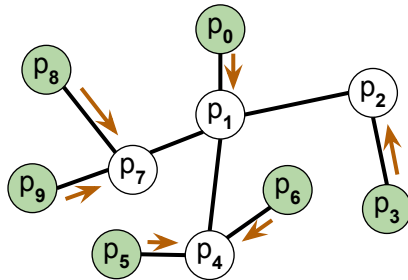
# A More Complex Example: Tree Wave



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)
```

© A. Bechini 2020

A. Bechini - UniPi

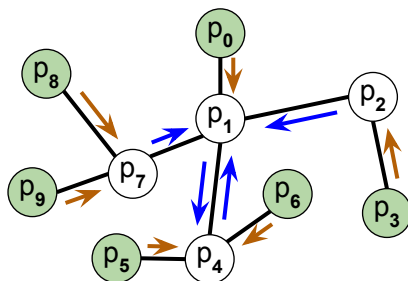


# A More Complex Example: Tree Wave

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

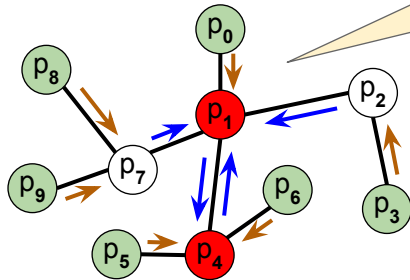
//toNeighbor: the only proc with recv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
recv[toNeighbor] = true //not necessary...
decide()
//this way, only two processes decide. Possibly:
//foreach n in neighbors\toNeighbor:
//    send(n, outToken)
```

© A. Bechini 2020

A. Bechini - UniPi

# 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

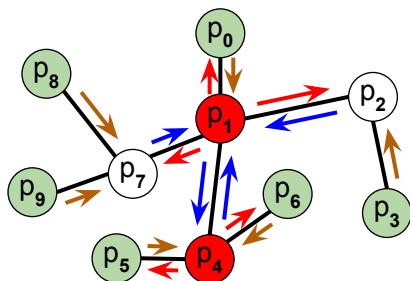
```
//toNeighbor: the only proc with rcv == false
send(toNeighbor, outToken)
receive(toNeighbor, inToken)
rcv[toNeighbor] = true //not necessary...
decide()
//this way, only two processes decide. Possibly:
foreach n in neighbors\toNeighbor:
    send(n, outToken)
```

© A. Bechini 2020

A. Bechini - UniPi

# A More Complex Example: Tree Wave

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



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

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

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

© A. Bechini 2020

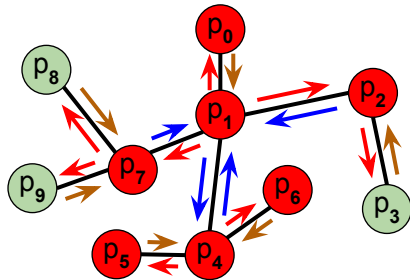
A. Bechini - UniPi

# 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)
```

© A. Bechini 2020

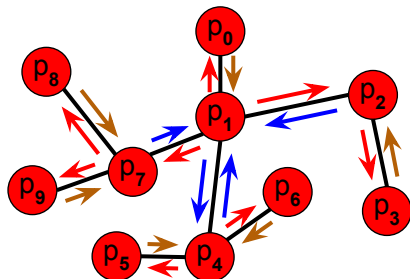
A. Bechini - UniPi

# 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)
```

© A. Bechini 2020

A. Bechini - UniPi



# 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



## Taking Snapshots



# What Do We Mean by “Snapshot”?



Snapshot of the execution of a distributed algorithm:  
configuration consisting of the **local states** of processes,  
along with **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).



## Why Snapshots?

Snapshots can be aimed at:



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



# Snapshot Algorithm (I)

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

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.



# 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.

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.).

# Chandy-Lamport Algorithm (I)

The most popular snapshot algorithm. System hypotheses:

- FIFO channels
- Strongly connected (di)graph

Idea: local recording actions driven by special “marker” messages.

Any process can start a snapshot.

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

## 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>
```

Any process  $p$  receiving a marker msg performs:

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

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
```

# Chandy-Lamport Algorithm: Complexity

Message complexity:



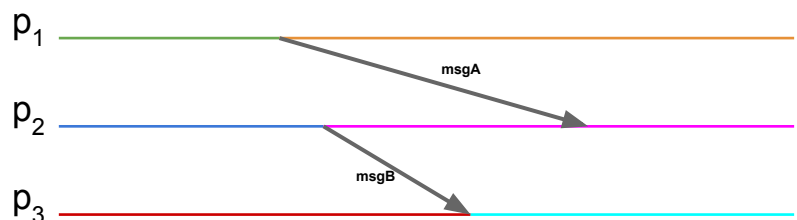
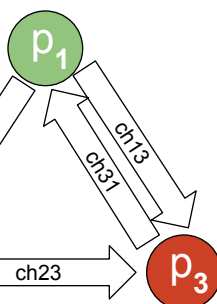
we have just one marker message per FIFO channel,  
so it is  $\Theta(E)$ , with  $E$  as # of channels.

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  $\mathcal{O}(D)$ .

## 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.



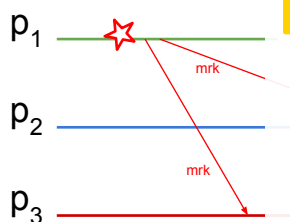
## Chandy-Lamport Alg. Example (II)

$p_1$  starts, recording its state (green), and sends **mrk** to  $p_2$  and  $p_3$ .

$p_3$  receives **mrk** first, so...

*ch12, ch13*

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



© A. Bechini 2020

A. Bechini - UniPi

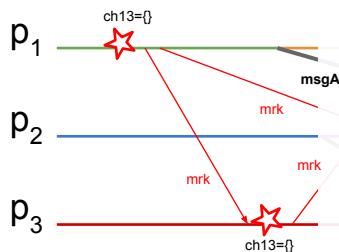
## Chandy-Lamport Alg. Example (III)

$p_3$  records its state (red), and sends **mrk** to  $p_1$ . Meanwhile,

$p_1$  sends basic **msgA** to  $p_2$  and changes state (orange), then...

*ch13* *(ch12, ch13)*

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



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

© A. Bechini 2020

A. Bechini - UniPi

*p2 chmark = [true]*  
*ch23*

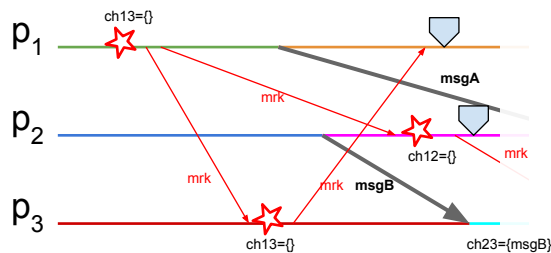
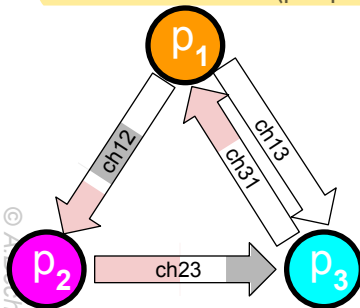
# Chandy-Lamport Alg. Example (IV)

$p_2$  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

*because it received marker messages from all incoming channels*

$p_1$  receives mrk from  $p_3$  and terminates

$p_3$  receives msgB from  $p_2$ , and changes state (cyan).

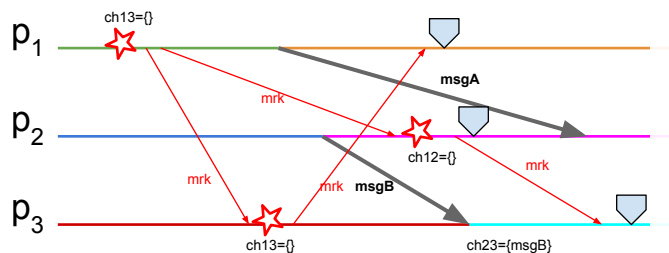
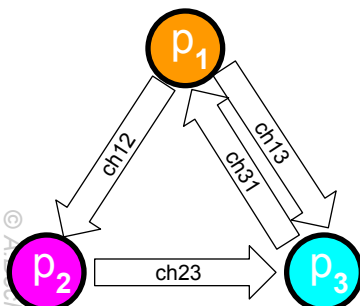


© A. Bechini 2020

# Chandy-Lamport Alg. Example (V)

$p_2$  receives basic msgA from  $p_1$ , but its local snapshot is already terminated;

$p_3$  receives mrk from  $p_2$ , and terminates.



© A. Bechini 2020

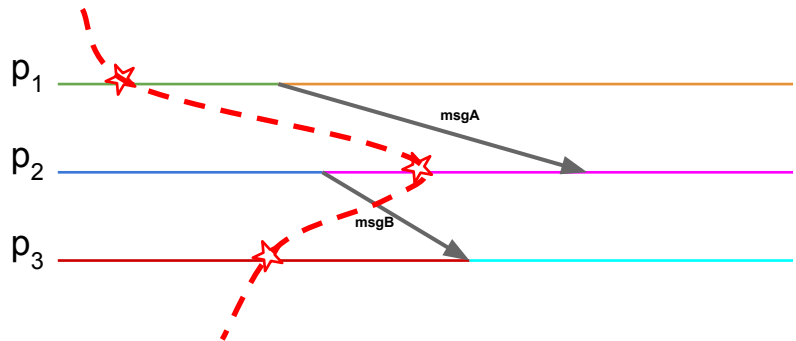
# Chandy-Lamport Alg. Example (VI)



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...

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



# Chandy-Lamport Alg. Correctness



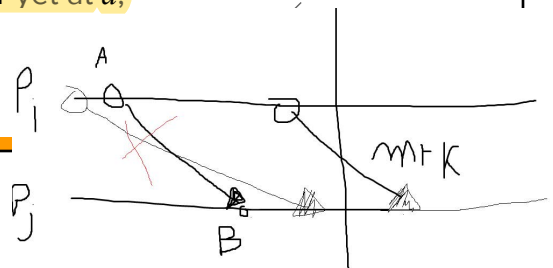
Correctness means: a snapshot corresponds to a *consistent cut*, i.e.:

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

*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 mrk yet at  $a$ ,  
 so  $a$  is pre-snapshot as well.

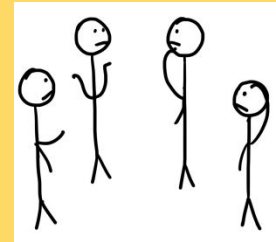
The FIFO hypothesis is crucial!





# Pause for Thought

## Correctness Depends on Hypotheses !



## Removing FIFO Hypothesis...



The FIFO property is crucial for the algorithm correctness.

How to arrange an algorithm to avoid recurring to FIFO?

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

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





# Lai-Yang Algorithm (I)

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

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

© A. Bechini 2020

A. Bechini - UniPi

# Lai-Yang Algorithm (II)



To perform the local snapshot:

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

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
```

© A. Bechini 2020

A. Bechini - UniPi

## 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.

## Performing Broadcast/Convergecast



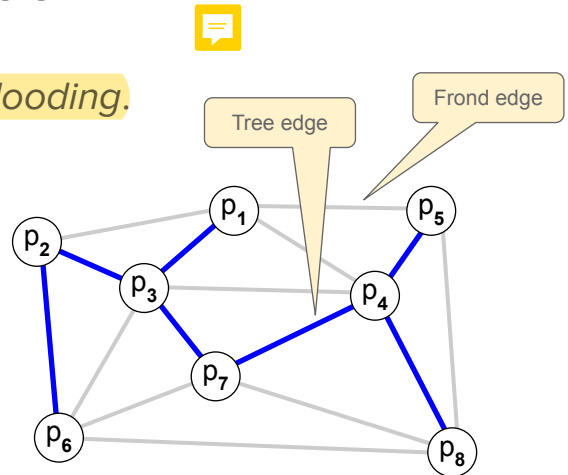


# Building a Spanning Tree

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

An idea could be building up a **spanning tree**,  
to be used later for broadcast/convergecast.

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



## 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.
- 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.

# “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

- 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.

The initiator (the tree root) has just to know its own children.

© A. Bechini 2020

A. Bechini - UniPI

## A Simple Spanning Tree Algorithm

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

First action for the initiator:



```
procedure onForward(proc sender)
  if parent == ⊥ then //first fwd received
    parent=sender; children=empty; expected=n_neighbors-1
    if expected == 0 then //reached a "dead end"
      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"
```

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

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

```
procedure onBackward(proc sender, msg m)
  expected=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>
```

© A. Bechini 2020

A. Bechini - UniPI

# What about Complexity?

## Message complexity:

# of tree edges:  $n-1$  ; 1 fwd + 1 bck per tree edge  $\rightarrow 2(n-1)$

# of fronds:  $E-(n-1)$  ; 2 fwd +2 bck per frond  $\rightarrow 4E - 4(n-1)$

In total,  $4E - 2(n-1)$  so msg complexity is  $O(E)$  with  $E = \#$  edges.

But  $E$  in a connected tree 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  $D = \text{network diameter}$ .

So the time complexity is  $O(D)$ .

# Pause for Thought

## Nondeterministic Behaviors





# Mutual Exclusion: The Distributed Way



## Problem Statement



Informally: Only one process *at a time* can “access a resource.”

What does it mean “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* succeed



# Process States wrt Critical Sections

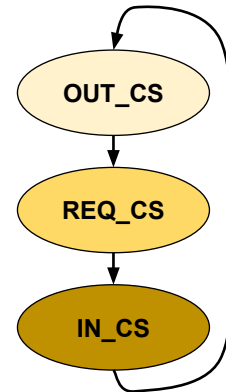
As regards the relationship with a CS,  
a process can be in three different states:



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.

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



# Ricart-Agrawala Algorithm - Basics



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  $(ts_a < ts_b) \vee ((ts_a = ts_b) \wedge (i < j))$

Lamport timestamps...

## Ricart-Agrawala Algorithm



```
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 (pending_oks == 0)
  mystate = IN_CS // then, CS
```

```
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

```
procedure onReq(proc sender, msg m)
  myclock = max(myclock, m.ts)
  my_prior = (mystate != OUT_CS) and (myreq_lt, pid) < m.ts
  if my_prior then req_delayed.append(sender)
  else send(sender, ok)
```

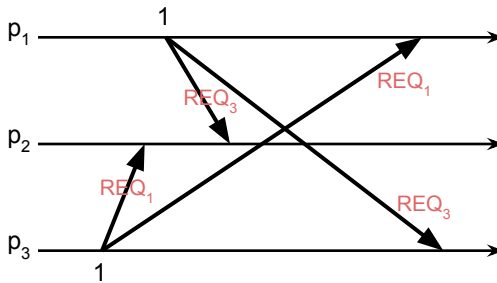
```
procedure onOk(proc sender, msg m)
  pending_oks = pending_oks-1
```





# Ricart-Agrawala Algorithm - Example

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



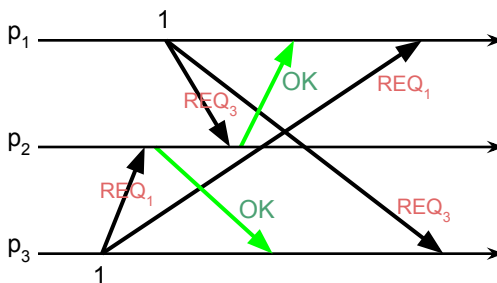
© A. Bechini 2020

A. Bechini - UniPi



# Ricart-Agrawala Algorithm - Example

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



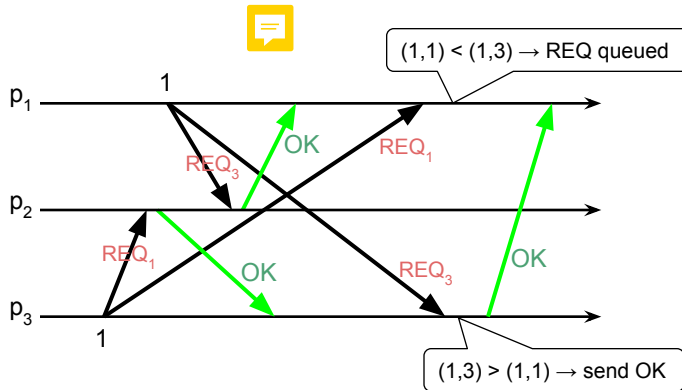
© A. Bechini 2020

A. Bechini - UniPi



# Ricart-Agrawala Algorithm - Example

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

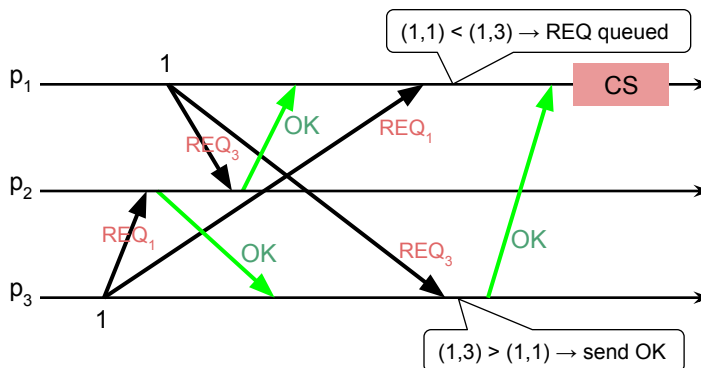


© A. Bechini 2020

A. Bechini - UniPi

# Ricart-Agrawala Algorithm - Example

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

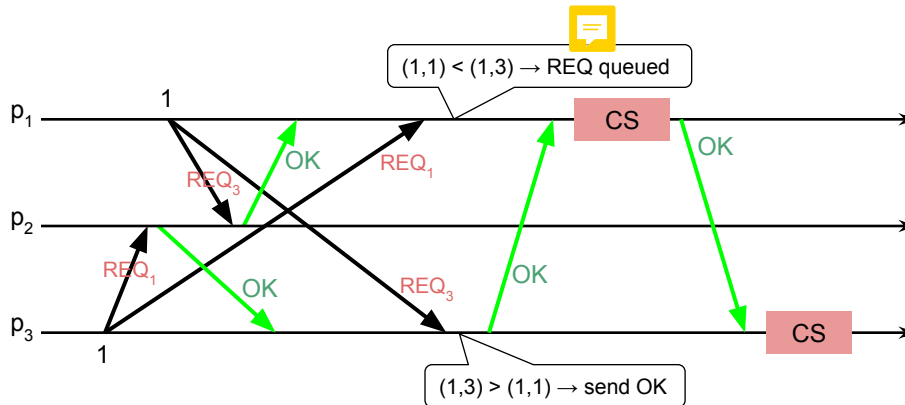


© A. Bechini 2020

A. Bechini - UniPi

# Ricart-Agrawala Algorithm - Example

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



© A. Bechini 2020

A. Bechini - UniPi

## 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  $O(D)$ .

© A. Bechini 2020

A. Bechini - UniPi

# Ricart-Agrawala Alg. Correctness (I)



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

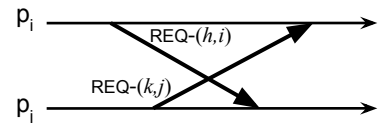
*Proof:* by contradiction; let suppose both  $p_i$  and  $p_j$  are in state IN\_CS. It follows that each of them has sent REQ to the other (with ts  $(h,i)$  and  $(k,j)$ , respectively), and got OK back. Two scenarios are possible:

- 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.

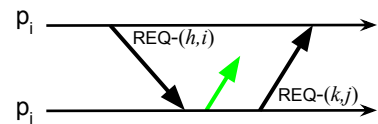
When  $p_j$  receives REQ, it performs

$myclock_{(j)} = \max(myclock_{(j)}, h)$ , 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)

```
<rob> hi
<emily> hey you
<rob> last night was nice
<emily> the best i've had
<rob> yeah it was AMAZING
<emily> ok, i have to ask
<emily> is this for real?
<emily> or is it just sex
<rob> definitely just sex
<emily> holy shit
<emily> are you serious?
<emily> you don't know how much that made
        my stomach hurt
<emily> i want to cry
<rob> i'm sorry
<rob> i wanted to type 'i love you'
<rob> but our line lengths were syncing up
<emily> ...
<rob> and it would have broken the pattern
• emily has disconnected
```



# Token Ring Mutex



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

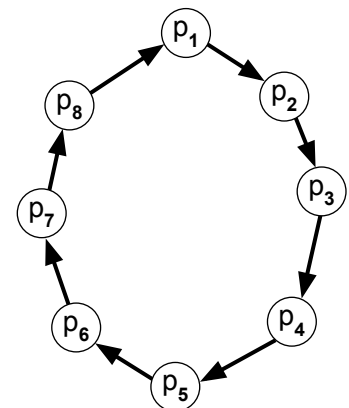
The permit to execute CS is a **token msg** that circulates the ring.

**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.



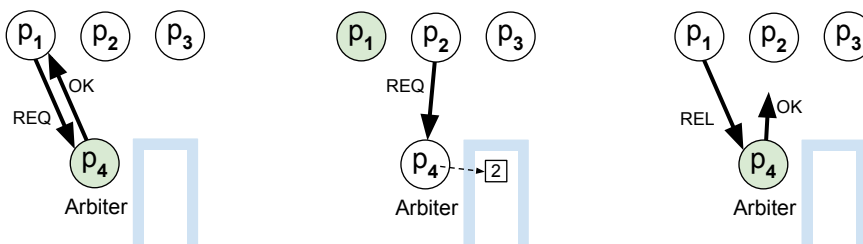
ME2 follows from the ring topology and the token passing rules, which assure that one token will *eventually* reach each node.

# Plain Centralized Mutex



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

© A. Bechini 2020



## 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!

© A. Bechini 2020



# Towards Synchronous Models



A *synchronous model* of computation accounts for a sequence of fixed time-slots according to the global time, and each operation has to be performed at 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 employ 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.

Each process  $p_i$  is associated with a **unique** id :UID <sub>$i$</sub> ; 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

Idea: each process sends its own UID on the ring,  
and stops the incoming UIDs 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.

# Chang and Robert's Election (II)



For a process 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

```

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

```

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

```



# C&R's Election: Complexity



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

Best case: one single initiator, with highest UID  $\rightarrow n$  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 yields  $n$  ELECTION msgs, the one with the second highest UID yields  $n-1$ , and so on; in total,  $n(n+1)/2$  msgs:  $\rightarrow O(n^2)$

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

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.

## 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*.

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!



# 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 have 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 begin the election, must send an ELECTION msg to those processes with higher UID.

Then it awaits for their ANSWER msgs back.

If no answer arrives in 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.



# The Bully Algorithm (III)

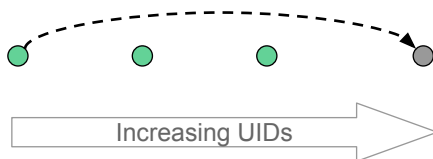


As a process  $p_i$  receives a LEADER msg from a process with higher UID, it sets its variable *leader* to that identifier.

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).



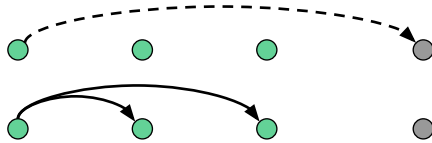
## The Bully Algorithm: Example



A process finds the current leader is down



# The Bully Algorithm: Example

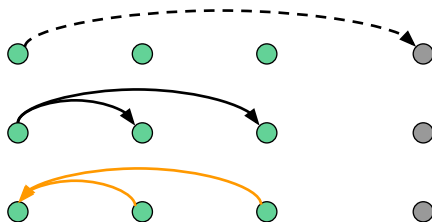


A process finds the current leader is down

Msg ELECTION - "who is the leader?"



# The Bully Algorithm: Example

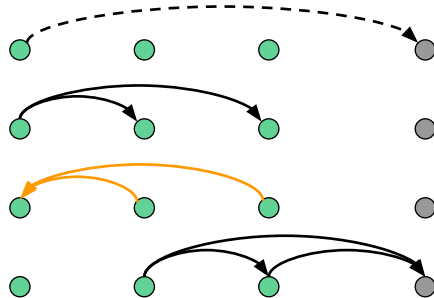


A process finds the current leader is down

Msg ELECTION - "who is the leader?"

Msg ANSWER - "for sure not you!"

# The Bully Algorithm: Example



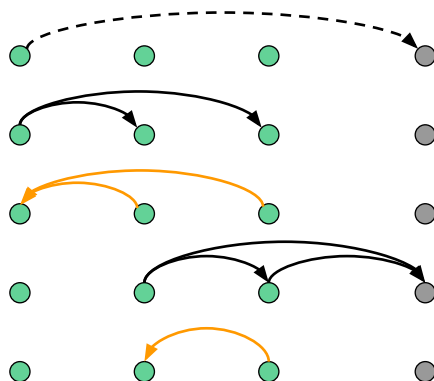
A process finds the current leader is down

Msg ELECTION - "who is the leader?"

Msg ANSWER - "for sure not you!"

Msg ELECTION - "who is the leader?"

# The Bully Algorithm: Example



A process finds the current leader is down

Msg ELECTION - "who is the leader?"

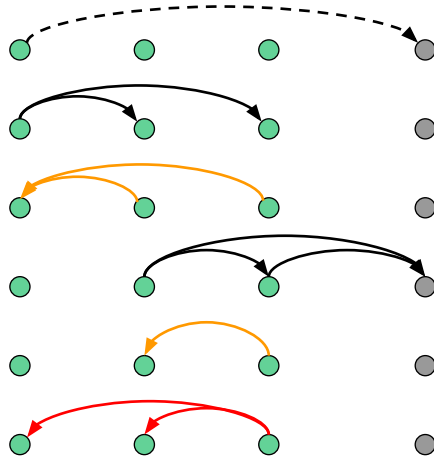
Msg ANSWER - "for sure not you!"

Msg ELECTION - "who is the leader?"

Msg ANSWER - "for sure not you!"



# The 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!"

## But... What about Bullism?



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

If it has 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?





receive → deliver to the app.

# Group Communication



Idea: group up processes, and *address groups instead of single processes*.

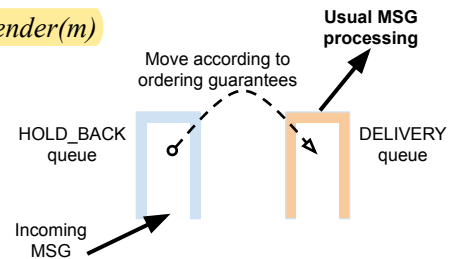
Relative primitives:

$Xmulticast(group\ g, msg\ m)$  - X indicates the specific type, 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**



© A. Bechini 2020

A. Bechini - UniPi

## Basic Multicast



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  $p$ .

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

© A. Bechini 2020

A. Bechini - UniPi

# Reliable Multicast



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

**Integrity** - A correct process delivers a message *m* at most once

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

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



**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 *Bdelivered* it either (in fact, there should be an HB relation from any other *Bdeliver* and the local *Rdeliver*)



# Other Multicast Semantics



**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.

**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.