

Leader Election

▲ Definition:

- each processor has a set of elected states and a set of not-elected states. Once an elected state is entered, the processor always is in an elected state; similarly for non-elected. I.e., irreversible decision.
- In every admissible execution,
 - every processor ^{ALLA FINE} eventually enters either an elected or a not-elected state (liveness)
 - exactly one processor (the **leader**) enters an elected state (safety)

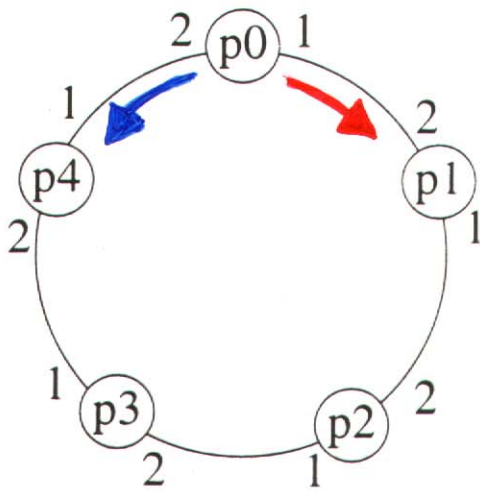
A leader can be used to coordinate future activities of the system. For instance:



- find a spanning tree using the leader as the root
- reconstruct a lost token for a token-ring

We will study leader election in rings.

Rings


In an oriented ring, processors have a consistent notion of left and right:



1 = left = clockwise 
2 = right = counter-clockwise 

For example, if messages are always forwarded on incident channel 1, they will cycle clockwise around the ring.

Why study rings?

- simple starting point, easy to analyze
- abstraction of a token ring 
- lower bounds for ring topology also apply to arbitrary topologies

Anonymous Rings

Intuition is that processors do not have unique identifiers.

Related issue is whether an algorithm A relies on processors knowing the ring size. (NON UNIFORM)

- **uniform** algorithm — does not use the ring size (same algorithm for each size ring)

Formally, every processor in every size ring is modeled with the same state machine A .

- **non-uniform** algorithm — does use the ring size (different algorithm for each size ring; may be only trivially different)

Formally, for every value of n , there is a state machine A_n such that every processor in a ring of size n is modeled with A_n . Thus A is the collection of all the A_n 's.

Leader Election in Anonymous Rings

Theorem 3.2: There is no leader election algorithm for anonymous rings, even if the algorithm knows the ring size (i.e., is non-uniform) and the ring is synchronous.

Proof Sketch: (BY CONTRADICTION)

- Every processor begins in the same state with the same messages originally in transit.
- Every processor receives the same messages and thus makes the same state transition and sends the same messages in round 1.
- Every processor receives the same messages and thus makes the same state transition and sends the same messages in round 2.
- Etc.

^{ALLA FINE} Eventually some processor is supposed to enter an elected state. But then they all would, a contradiction.

■
Consequently, there is no uniform or asynchronous leader election algorithm.

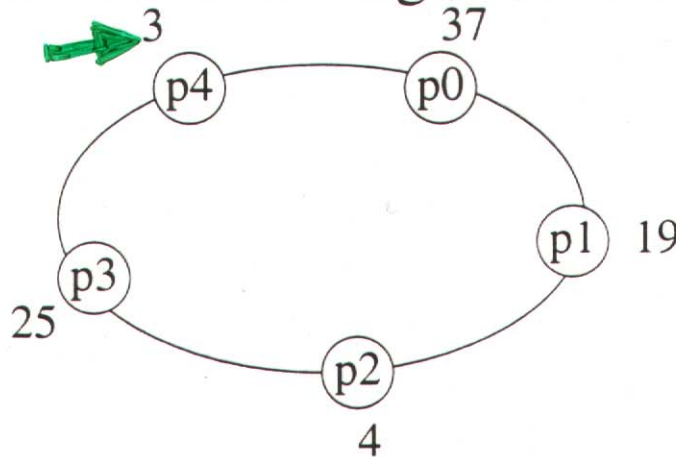
Rings with Identifiers

(NON ANONIMI)

Assume each processor has a unique identifier. Distinguish between indices and identifiers:

- **indices** are 0 through $n - 1$ and are unavailable to the processors; used only for analysis
- **identifiers** are arbitrary nonnegative integers and are available to the processors via a special state component called **id**.

Specify a ring by starting with the smallest id and listing ids in clockwise order. E.g., 3, 37, 19, 4, 25.



Uniform algorithm: There is one state machine for every id, no matter what size ring.

Non-uniform algorithm: There is one state machine for every id and every different ring size.

Overview of Leader Election in Rings with Ids

In this case, there are algorithms. We will evaluate them according to their **message complexity**.

Overview of Upcoming Results:

- asynchronous ring: $\Theta(n \log n)$ messages
- synchronous ring:
 - $\Theta(n)$ messages under certain conditions
 - otherwise $\Theta(n \log n)$ messages

All bounds are asymptotically tight.

ASINCRONO

$O(n^2)$ Messages Leader Election Algorithm

Each processor follows these rules:

- Initially send your id to the left
- When you receive an id (from the right):
 - if it is greater than your id then forward it to the left (you will never be the leader)
 - if it is equal to your id then elect yourself leader
 - if it is smaller than your id then do nothing

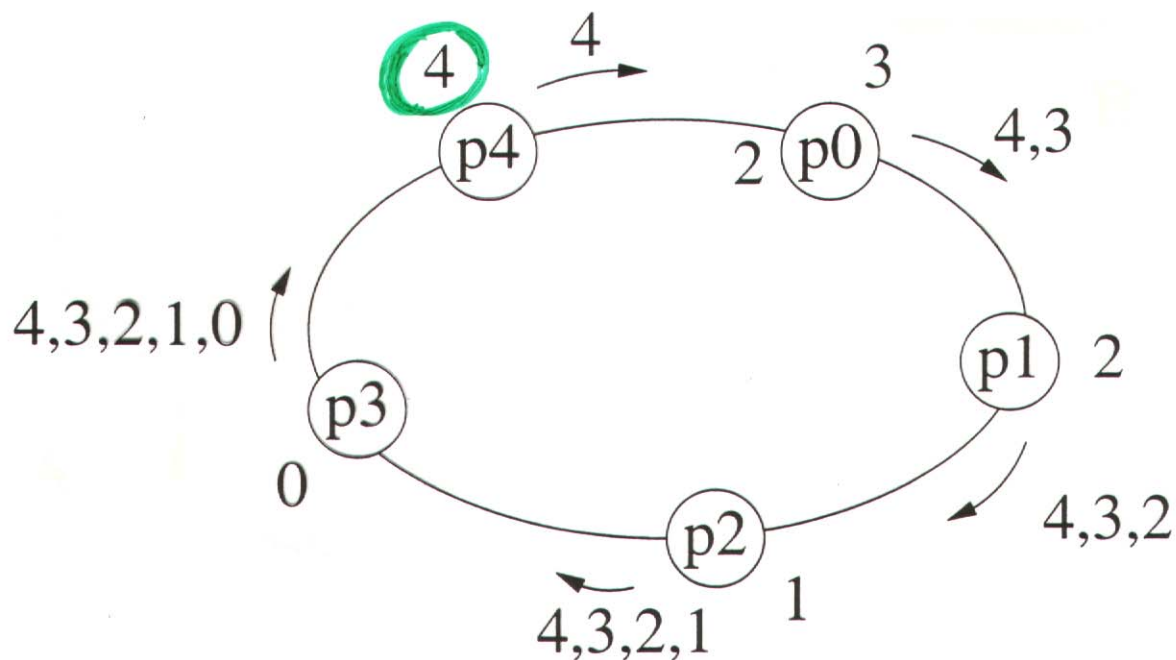
Correctness: Elects processor with largest id. Message containing that id passes through every processor.

Message complexity: Depends how ids are arranged.

- Largest id travels all around the ring, n messages
- Second largest id travels until reaching largest
- Third largest id travels until reaching largest or sec-
ond largest
- Etc.

$O(n^2)$ Messages Algorithm (cont'd)

Worst way to arrange the ids is in decreasing order:



- Second largest id contributes $n - 1$ messages.
- Third largest id contributes $n - 2$ messages.
- Etc.

Total number of messages is

$$\sum_{i=1}^n i = \underline{\Theta(n^2)}.$$

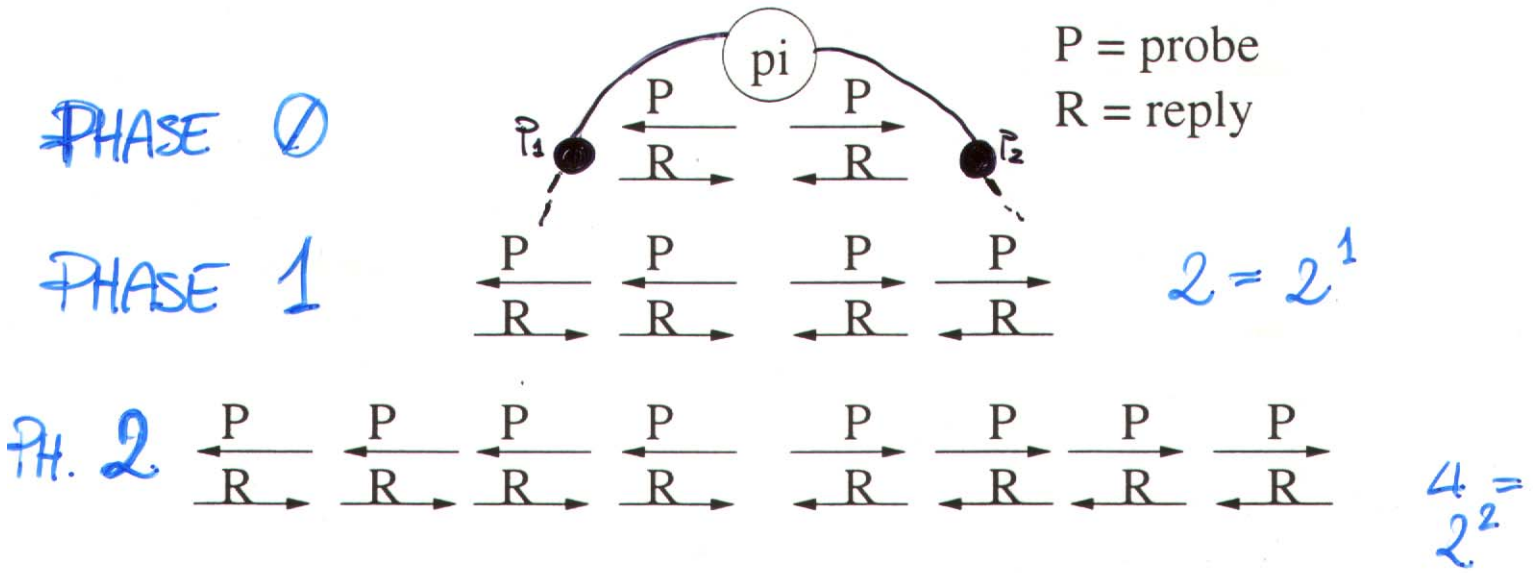
■
APPROFONDIMENTO: QUANTO COSTA NEL CASO MIGLIORE? E NEL CASO MEDIO?

$O(n \log n)$ Messages Leader Election Algorithm

- Each processor tries to probe successively larger neighborhoods. Size of neighborhood doubles in each phase.
SONDARE
K-NEIGHBORHOOD: SET OF PROCESSORS WITHIN DISTANCE K FROM P_i
- A probe is initiated by sending a probe message containing the initiator's id.
- If a processor receives a probe message whose id is larger than its own id, the processor will either forward it on or send back a reply, as appropriate.
- If a processor receives a probe message whose id is smaller than its own id, it does nothing.
- If a processor receives a probe message with its own id, then it becomes the leader.
- If a processor receives a reply message not destined for itself, it forwards it.
- If a processor receives both reply messages destined for itself, it proceeds to the next phase.

→ MESSAGE = \langle id, phase number, hop counter \rangle

$O(n \log n)$ Messages Algorithm (cont'd)



Correctness: Similar to previous algorithm.

Message Complexity:

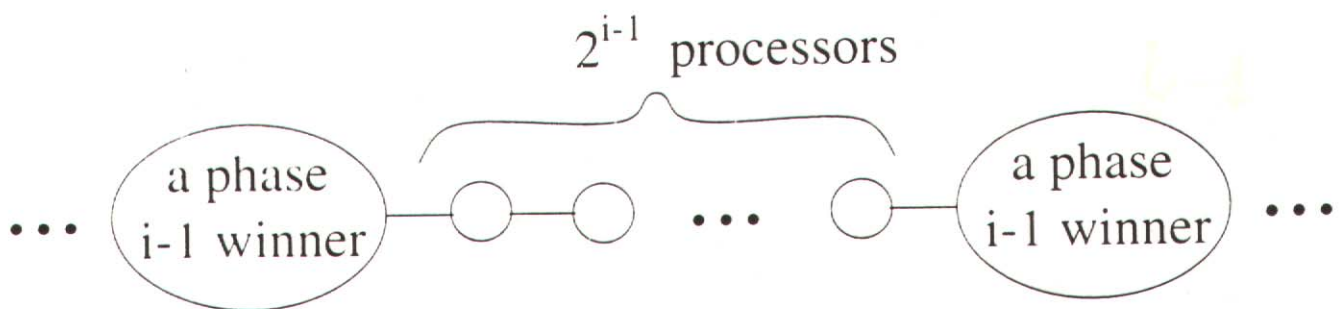
- Each message belongs to a phase and is initiated by a particular processor. GENERATED
- Probe distance in phase i is 2^i .
- The number of messages initiated by a particular processor in phase i is $\leq 4 \cdot 2^i$ (probes and replies in both directions).

$O(n \log n)$ Messages Algorithm (cont'd)

How many processors initiate probes in phase i ?

- For $i = 0$, all n of them do.
- For $i > 0$, every processor that is a "winner" in phase $i - 1$ does (has largest id in its 2^{i-1} neighborhood)

Maximum number of phase $i - 1$ winners occurs when they are packed as densely as possible:



Total number of phase $i - 1$ winners is at most

$$\frac{n}{2^{i-1} + 1}$$

$i \geq 1$

How many phases are there? Phases continue until there is only one winner, so $\log n$ phases suffice.

(IN FACT, $2^{\log n} = n$)

$O(n \log n)$ Messages Algorithm (cont'd)

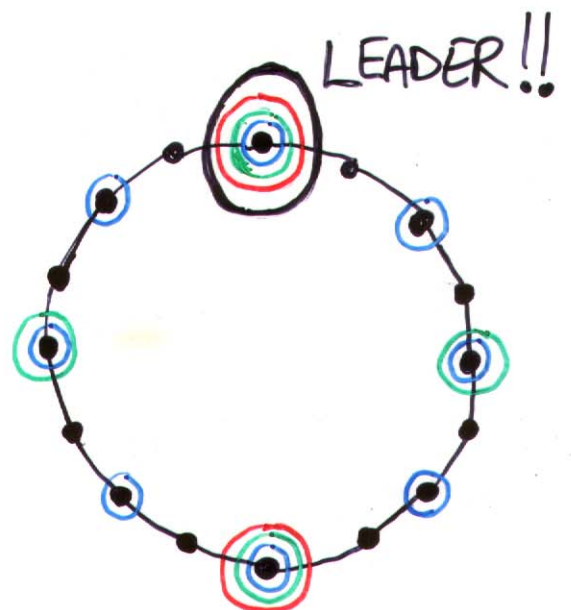
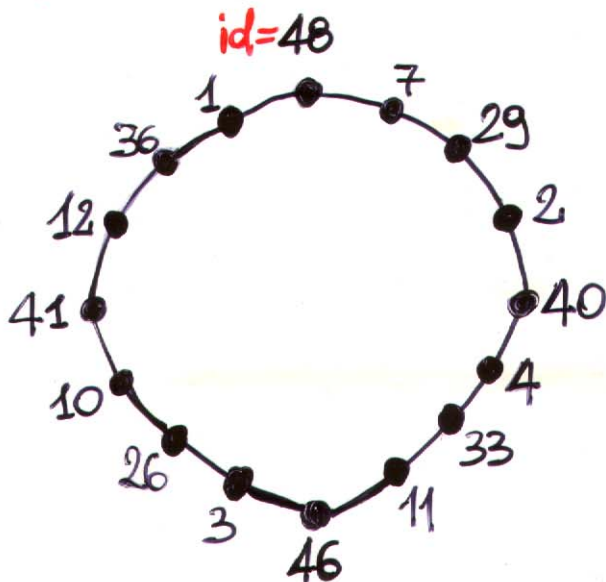
► Total number of messages is

$$\begin{aligned}
 &\leq \underbrace{4 \cdot n}_{\text{PHASE 0}} + \sum_{i=1}^{\log n} 4 \cdot 2^i \cdot \underbrace{\left(\frac{n}{2^{i-1} + 1} \right)}_{\text{MAXIMUM NUMBER OF WINNERS IN PHASE } i-1} + \underbrace{2n}_{\text{LAST PHASE}} \\
 &\leq 6n + 4n \sum_{i=1}^{\log n} \frac{2^i}{2^{i-1}} \\
 &= 6n + 8n \log n \\
 &= \boxed{O(n \log n)}.
 \end{aligned}$$

□

EX

$n = 16$



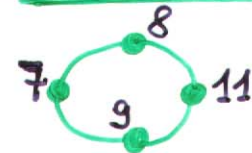
→ NELLA FASE i HO $\frac{n}{2^{i+1}}$ VINCITORI

Leader Election in Synchronous Rings

First, a simple algorithm for the synchronous ^{NON-UNIFORM,} model:

- Group the rounds into **phases** so that each phase contains n rounds.
- In phase i , the processor with id i , if there is one, sends a message around the ring and is elected.

Example: $n = 4$ and 7 is smallest id.



- In phases 0 through 6 (corresponding to rounds 1 through 28), no message is ever sent.
- At beginning of phase 7 (round 29), processor with id 7 sends message which is forwarded around ring.

Note ^{SFRUTTAMENTO} *reliance on synchrony and knowledge of n !*

Correctness: Convince yourself.

Message Complexity: $\Theta(n)$. Note that this is optimal.

Time Complexity: $O(n \cdot m)$, where m is the smallest id in the ring. Not bounded by n .

Another Synchronous LE Algorithm

This algorithm

- works in a slightly weaker model: Processors might not all start at same round; a processor either wakes up spontaneously or when first gets a message.
- is uniform (does not rely on knowing n).

Idea:

- A processor that wakes up spontaneously is active; sends its id in a fast message — 1 edge/round.
- A processor that wakes up when receiving a message is relay; ^{TRASMETTITORE} never in the competition.
- A fast message becomes slow if it reaches an active processor — $1 \text{ edge}/2^m$ rounds (m is msg id)
- Processors (active or relay) only forward a message whose id is smaller than any id this processor has seen so far (ignoring the id of relay processors).
- If a processor gets own id back, leader.

Analysis of Synchronous LE Algorithm

Correctness: Convince yourself that active processor with smallest id is elected.

Message Complexity: Winner's message is the fastest. While it traverses the ring, other messages are slower, so they are ^{SORPASSATI} overtaken and stopped before too many messages are sent.

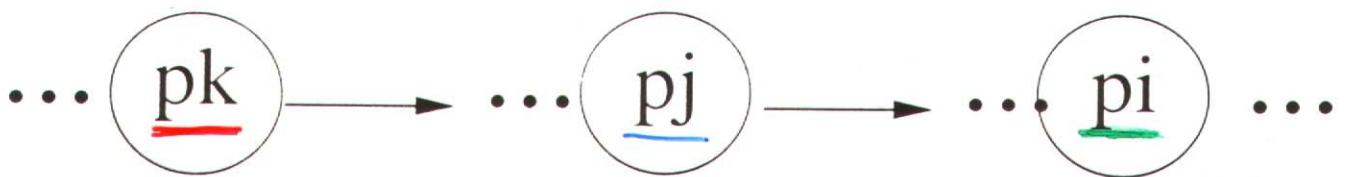
More carefully, divide messages into three kinds:

- ① fast messages
- ② slow messages sent while the leader's message is fast
- ③ slow messages sent while the leader's message is slow

Analysis of Synchronous LE Algorithm (cont'd)

Number of type 1 messages (fast):

Show that no processor forwards more than one fast message. (BY CONTRADICTION)



If $\underbrace{p_i}$ forwards $\underbrace{p_j}$'s fast msg and $\underbrace{p_k}$'s fast msg, then when $\underbrace{p_k}$'s fast message arrives at $\underbrace{p_j}$:

1. either $\underbrace{p_j}$ has already sent its fast message, so $\underbrace{p_k}$'s message becomes slow, or
2. $\underbrace{p_j}$ has not already sent its fast message, so it never will.

► Number of type 1 messages is at most n .

Analysis of Synchronous LE Algorithm (cont'd)

Number of type 2 messages (slow while leader's is fast):

- Leader's message is fast for at most n rounds.
- Slow message i is forwarded $n/2^i$ times in n rounds.
- Worst case (largest number of messages) is when ids are as small as possible, 0 to $n - 1$.

Number of type 2 messages is at most $\sum_{i=1}^{n-1} \frac{n}{2^i} \leq n$.

Number of type 3 messages (slow while leader's is slow):

- Once leader's message x becomes slow, it takes at most $n \cdot 2^x$ rounds to return to leader.
- No messages are sent once leader's message has returned to leader.
- Slow message i is forwarded $n \cdot 2^x / 2^i$ times in $n \cdot 2^x$ rounds.
- Worst case is when ids are 0 to $n - 1$ and $x = 0$.

Number of type 3 messages is at most $\sum_{i=0}^{n-1} n \cdot \frac{2^0}{2^i} \leq 2n$.

□

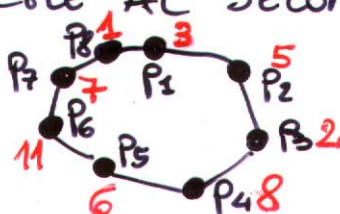
Time Complexity of Synchronous LE Algorithms

Time Complexity: $O(n \cdot 2^x)$, where x is the minimum id. Even worse than the previous algorithm.

Both these algorithms have two potentially undesirable properties:

- rely on the numeric values of the ids to count
- number of rounds bears no relationship to n , but depends on the minimum id

Approfondimento : ESEGUIRE IL SUDDETTO ALGORITMO
SUL SEGUENTE ANELLO, COSTITUITO DA $n=8$ PROCESSORI,
SUPPONENDO CHE AL PRIMO ROUND SI SVEGLINO
 P_1, P_5 e P_8 , E CHE AL SECONDO ROUND SI SVEGLI P_3

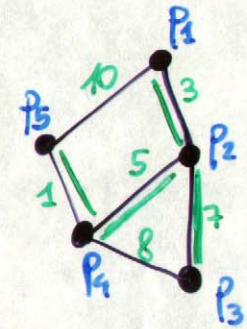


Gallager-Humblet-Spira Minimal Spanning Tree

- Assumption:

- unique weights $w(e)$ for all edges

- ASYNCHRONOUS SYSTEM WITH AN ARBITRARY CONNECTED GRAPH $G=(V,E)$
NON-ANONYMOUS



- Defs:

- Fragment F is a subtree of the MST of G

- Outgoing edge e from F if one of the edge's nodes is in F and the other is not in F .

➔ GHS ALGORITHM REQUIRES $O(|E| + |V| \log |V|)$
MESSAGES

GHS ALGORITHM

FURTHER ASSUMPTION: EDGES FOLLOW A **FIFO** POLICY

LEMMA: IF THE EDGES HAVE DISTINCT WEIGHTS \Rightarrow THE **MST** IS UNIQUE

PROOF: (BY CONTR) T_1, T_2 MST OF G .

LET e BE THE MIN-WEIGHT EDGE BELONGING TO $T_1 \Delta T_2$. W.L.O.G., LET $e \in T_1$.

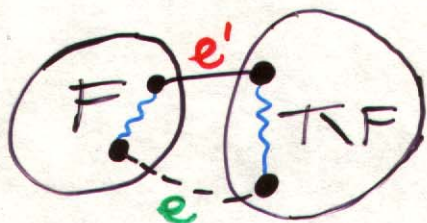
$\rightarrow T_2 \cup \{e\}$ CONTAINS A CYCLE, AND AT LEAST ONE EDGE e' OF THIS CYCLE $\notin T_1$.

$\rightarrow w(e) < w(e')$ AND $T_2 \cup \{e'\} \setminus \{e\}$ IS A SPANNING TREE WITH WEIGHT $<$ THAN T_2 (CONTR!)

LEMMA: LET T BE THE UNIQUE **MST** OF G .

\forall FRAGMENT F OF T , THE **MINIMUM-WEIGHT OUTGOING EDGE** OF $F \in T$.

PROOF: (BY CONTR) LET e BE THE **MWOE** OF F , AND LET $e \notin T$. $\rightarrow T \cup \{e\}$ CONTAINS A CYCLE, AND SUCH A CYCLE CONTAINS AT LEAST ONE ADDITIONAL OUTGOING EDGE OF F , SAY e' .



$$w(e) < w(e')$$

$\rightarrow T \setminus \{e'\} \cup \{e\}$ IS BETTER THAN T . **CONTR!**

GHS (continued)

- Proposition: If F is a fragment and e is the least weight outgoing edge of F , then $F \cup \{e\}$ is a fragment.

Algorithm **MAIN IDEA**

Start with single node fragments and incrementally enlarge them

Global description of GHS

- Maintain for $G = (V, E)$ a set of fragments such that $\cup_i \text{nodes}(F_i) = V$ and $i \neq j \Leftrightarrow F_i \cap F_j = \emptyset$
- start with one-node fragments
- nodes in a fragment cooperate to find the lowest weight outgoing edge
- when ~~the~~^{MWOE} edge is found, combine^{JOIN} with the other fragment
- Terminate when only one fragment remains
- OPERATIONS ARE COORDINATED BY CORE EDGES OF THE FRAGMENTS

FRAGMENT
IDENTITY : (w, L)

w : WEIGHT OF THE CORE EDGE

L : LEVEL OF THE FRAGMENT, WITH $L=0$
IF THE FRAGMENT CONTAINS A SINGLE NODE

FRAGMENT UNION

COMBINATION	$L_1 = L_2$
ABSORPTION	$L_1 < L_2$

A COMBINATION OF TWO FRAGMENTS OF LEVEL $L-1$ PRODUCES A NEW FRAGMENT OF LEVEL L AND WHOSE CORE EDGE IS THE EDGE USED FOR THE UNION

AN ABSORPTION OF A FRAGMENT DOES NOT CHANGE THE IDENTITY OF THE ABSORBING FRAG.

AS SOON AS A UNION TAKES PLACE, THE IDENTITY OF THE RESULTING FRAGMENT IS SENT TO ALL ITS NODES.

Local description of GHS

- Each node p stores
 - the state of its edges e , $state_p[e] \in \{\text{basic, branch, reject}\}$
 - name of its fragment w ? MST MST
 - level of its fragment L
 - best weight of outgoing edges from its fragment
 - father channel (i.e. route towards the core node)
 - its own state, $state[p] \in \{\text{sleeping, Find, Found}\}$

IDENTITY {

(key)



TYPE OF MESSAGES:

INITIATE (w, L, s) : SENT BY CORE NODES RIGHT AFTER CREATION

TEST (w, L) : SENT BY A NODE IN Find STATE OVER ITS MINIMUM-WEIGHT BASIC EDGE TO CHECK WHETHER IS AN OUTGOING EDGE

REJECT, ACCEPT: RESPONSE TO **TEST**

REPORT (w) : USED TO FIND THE MIN-WEIGHT OUTGOING EDGE

CHANGE - CORE: SENT BY CORE NODES TO ACTIVATE UNION

CONNECT (w, L) : REQUEST OF UNION

GHS ALGORITHM

- IDENTIFYING OUTGOING EDGES
- FINDING THE MINIMUM OUTGOING EDGE
- FRAGMENT UNION

IDENTIFYING OUTGOING EDGES

NODE p_i IN $F_1(w_1, L_1)$ PICKS ITS MINIMUM-WEIGHT BASIC EDGE, SAY e AND SENDS ON IT A Test (w_1, L_1)

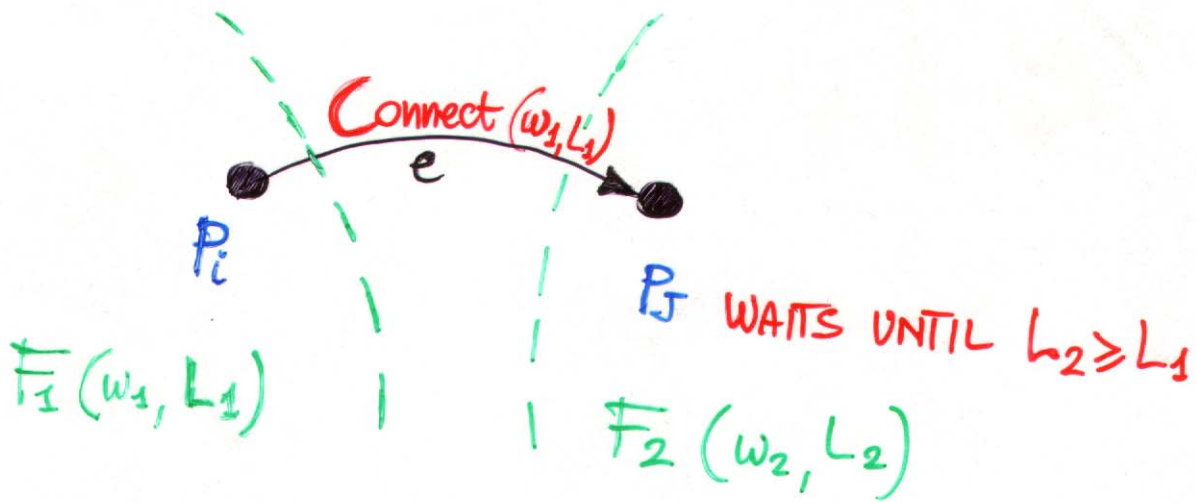


- 1) IF $(w_1, L_1) = (w_2, L_2) \Rightarrow e$ IS NOT AN OUTGOING EDGE $\Rightarrow p_j$ **Reject**; $e :=$ REJECTED
 - 2) IF $(w_1, L_1) \neq (w_2, L_2)$ AND $L_2 \geq L_1$
 $\rightarrow p_j$ **Accept**; p_i MARKS e AS **MIN. OUTGOING EDGE**.
 - 3) IF $(w_1, L_1) \neq (w_2, L_2)$ AND $L_2 < L_1$
 $\rightarrow p_j$ **DOES NOT REPLY** TO p_i UNTIL ONE OF THE ABOVE CONDITIONS IS VERIFIED
- THIS BLOCKS p_j (AND THE WHOLE F_1 !)

FINDING THE MINIMUM OUTGOING EDGE

- PROCESS STARTED BY CORE NODES BY SENDING **Initiate** ($w, L, Find$) TO ALL NODES IN THE FRAGMENT, THROUGH EDGES OF THE FRAGMENT
- A NODE p_i RECEIVING THE **Initiate** CHANGES ITS STATE TO **Find**. THEN:
 - 1) UPDATES ITS INFORMATION ABOUT FRAGMENT, THAT IS (w, L);
 - 2) RECORDS THE DIRECTION TOWARDS THE **CORE**
 - 3) IF p_i HAS OUTWARDS EDGES OF THE FRAGMENT, FORWARDS THE **Initiate** MESSAGE
 - 4) FIND ITS LOCAL MINIMUM OUTGOING EDGE
- IF p_i IS A LEAF OF THE FRAGMENT \rightarrow **Report** (w_i) AND ENTERS INTO A **Found** STATE (NOTE: w_i CAN BE ∞)
- IF p_i IS INTERNAL, WAITS FOR ALL THE REPORTS FROM ITS CHILDREN, AND CHOOSES THE MINIMUM; FINALLY SENDS A **Report** (w_i) AND MARKS THE **BEST EDGE**, AND ENTERS INTO A **Found** STATE
- BASED ON ALL THE **Reports**, CORE NODES SEND A **Change-Core** TO THE CHOSEN NODE, THAT SENDS A **Connect** (w, L) OVER ITS **MIN. OUTGOING EDGE**, AND MARKS IT AS A **Branch**.

COMBINING FRAGMENTS UNION



- ① IF $L_2 = L_1$ AND P_j IS GOING TO SEND (OR ALREADY SENT) A **Connect** ON EDGE e , THEN **COMBINATION** TAKES PLACE

→ $F = F_1 \cup F_2$
 $F(w(e), L_1 + 1)$
 e **CORE EDGE**

- ② IF $L_2 > L_1$, THEN **ABSORPTION** OF F_1 INTO F_2 TAKES PLACE

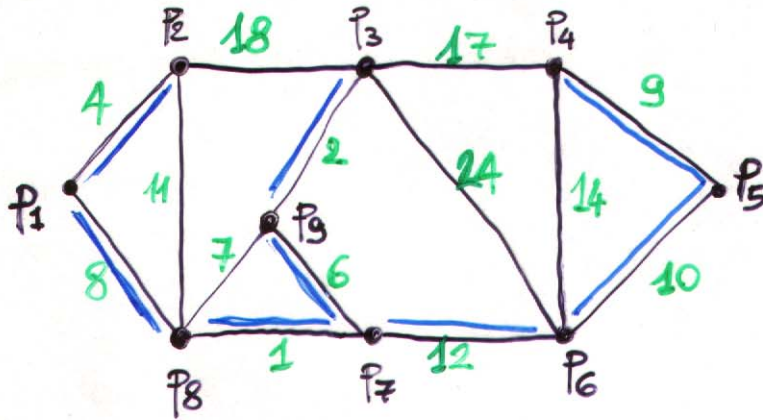
→ $F_2 := F_2 \cup F_1$
 $F_2(w_2, L_2)$

P_j SENDS AN **Initiate** (w_2, L_2, s_j) TO P_i WHERE $s_j \in \{\text{Find, Found}\}$; P_i FORWARDS THE **Initiate** MESSAGE TO NODES OF F_1 .

- ③ $L_1 > L_2$ IS **IMPOSSIBLE** (F_1 IS LOCKED)

Approfondimento

ESEGUIRE GHS SUL SEGUENTE GRAFO:



HP ① L'ALGORITMO INIZIA DA P_1 E P_5

② IL SISTEMA E' **PSEUDOSINCRONO**: I MESSAGGI INVIATI DA PROCESSORI **DISPARI** VENGONO CONSEGNATI IN **1** UNITA' DI TEMPO, QUELLI INVIATI DA PROCESSORI **PARI**, IN **2** UNITA' DI TEMPO

CORRECTNESS OF GHS

• FULL PROOF IS VERY COMPLICATED

→ WE FOCUS ON GENERAL PROPERTIES:

• TERMINATION

• SYNCHRONIZATION

• ABSORPTION WHILE SEARCHING FOR A MOE

TERMINATION

RESPONSE TO **test** AND **connect** ARE SOMETIMES DELAYED ⇒ DEADLOCK IS A PRIORI POSSIBLE

LEMMA FROM ANY CONFIGURATION WITH AT LEAST TWO FRAGMENTS, EVENTUALLY EITHER **ABSORPTION** OR **COMBINATION** TAKES PLACE

Proof: LET **L** BE THE MIN LEVEL IN THIS CONFIGURATION, AND LET **F** BE THE LEVEL-**L** FRAGMENT WHOSE MOE HAS **MIN WEIGHT** AMONG ALL LEVEL-**L** FRAGMENTS.

→ A **test** MESSAGE FROM **F** EITHER REACHES **F'** OF LEVEL $L' \geq L$ OR A **Sleeping** NODE.

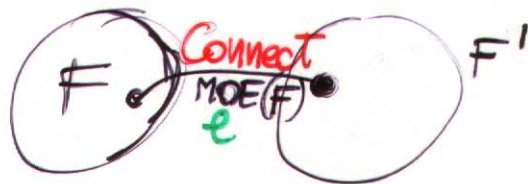
IN THE FIRST CASE, **F** GETS A REPLY IMMEDIATELY.

IN THE SECOND CASE, THE AWAKENED NODE BECOMES

A FRAGMENT OF LEVEL $L=0$ → CHOOSE A NEW **F** AND APPLY RECURSIVELY THE ARGUMENT.

→ **EVENTUALLY**, A CONFIGURATION IN WHICH THE FIRST CASE APPLIES IS REACHED.

→ **EVENTUALLY**, F FINDS ITS MOE:



TWO CASES:

- ① $L(F') > L(F) \rightarrow F'$ ABSORPTS F
- ② $L(F') = L(F) \rightarrow e$ IS ALSO THE MOE OF F' (BY CONSTRUCTION), AND F' CANNOT BE **LOCKED** $\rightarrow F$ AND F' COMBINE.

COROLLARY: GHS TERMINATES.

Proof: IF GHS DOES NOT TERMINATE \rightarrow THERE MUST BE AT LEAST **2** FRAGMENTS (SINCE WITH JUST ONE IT WOULD TERMINATE)

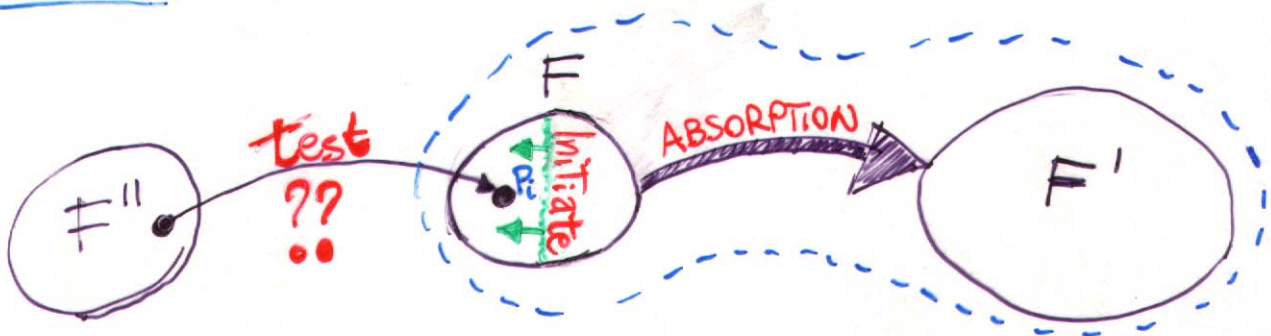
\rightarrow THE ABOVE LEMMA GUARANTEES THAT THE NUMBER OF FRAGMENTS WILL BE PROGRESSIVELY REDUCED UP TO **1** \rightarrow THESIS

SYNCHRONIZATION

MESSAGE TRANSMISSION TIME IS UNBOUNDED

→ A NODE MIGHT HAVE INACCURATE INFO ABOUT ITS OWN FRAGMENT.

EXAMPLE



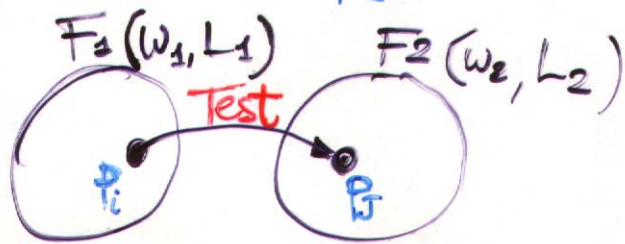
WE WILL SHOW THAT AN INACCURATE ANSWER DOES NOT AFFECT CORRECTNESS.

CLAIM 1 LET e BE THE CORE EDGE OF SOME FRAGMENT F . THEN, e IS NEVER THE CORE OF A FRAGMENT F' SUCH THAT $F \neq F'$.

CLAIM 2 A NODE p_i WHOSE FRAGMENT ID IS CURRENTLY (w, L) BELONGS TO A FRAGMENT WITH LEVEL $L' \geq L$.

Proof IF THE INFO OF p_i IS INACCURATE →
 p_i IS PARTICIPATING IN COMBINATION OR
ABSORPTION → IN BOTH CASES, $L' > L$ ■

REMARK 1 IF P_i SENDS A **test** TO $P_j \rightarrow$ THE FRAGMENT P_i BELONGS TO IS NOT JOINING (NOT COMBINING) (NOT ABSORBED) WITH OTHER FRAGMENTS \rightarrow THE ONLY INCORRECT INFO MIGHT BE IN P_j



REMARK 2 **Reject** MESSAGES ARE ALWAYS **CORRECT**

Claim 3 IF P_j SENDS AN **Accept** MESSAGE $\rightarrow P_i$ AND P_j ARE NOT IN THE SAME FRAGMENT.

Proof **Accept** IS SENT IFF $(w_3, L_3) \neq (w_2, L_2)$ AND $L_2 \geq L_1$.

IF $L_2 > L_1 \rightarrow$ THE REAL L_2 , BY CLAIM 2, CAN ONLY BE $\geq L_2 > L_1 \rightarrow P_i$ AND P_j ARE IN DIFFERENT FRAGMENTS.

IF $L_2 = L_1 \rightarrow$ THE [REAL L_2] $\geq L_2$

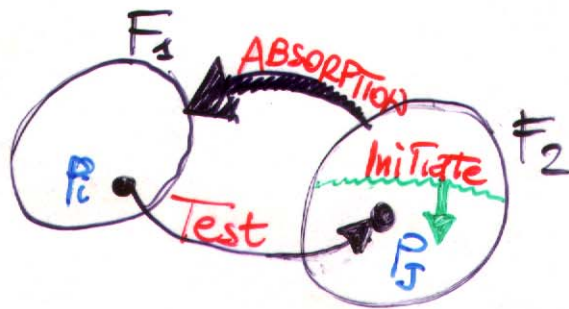
\rightarrow IF IT IS =, THEN $w_2 \neq w_1$ OK

\rightarrow IF IT IS >, THEN SEE ABOVE

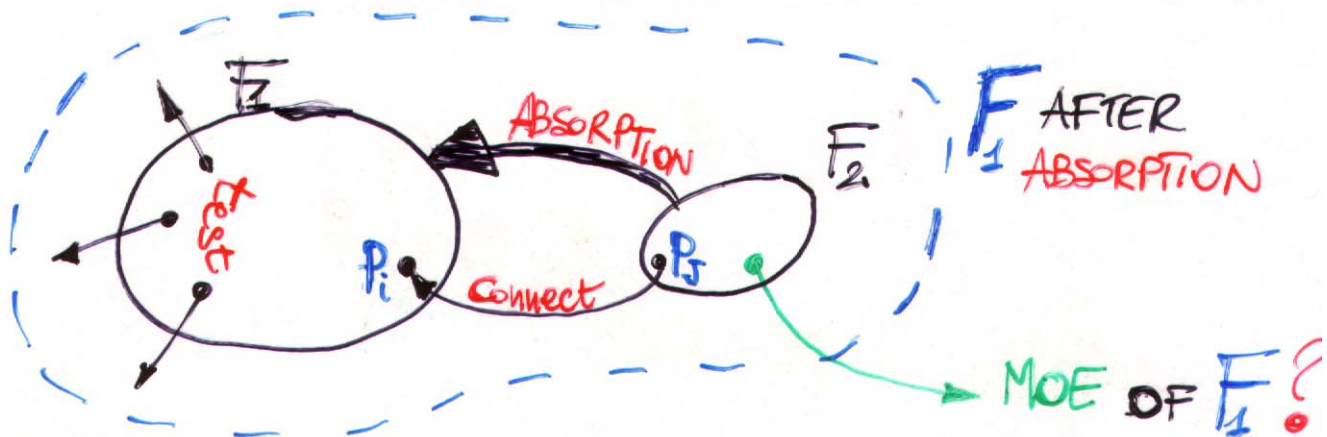
NOTE THAT IF $L_2 < L_1$, THE FOLLOWING IS POSSIBLE: F_2 MIGHT ABSORB F_1 ELSEWHERE, AND P_j IS STILL NOT INITIATED BY F_2

➔ P_i AND P_j ARE IN THE SAME FRAGMENT BUT P_j DOES NOT KNOW IT

➔ NO PROBLEM, SINCE P_j IS NOT REPLYING TO P_i



ABSORPTION WHILE SEARCHING FOR A MOE



$$F_1: (w_1, L_1) \quad F_2: (w_2, L_2) \quad L_2 < L_1$$

AFTER $\text{Connect}(w_2, L_2)$ FROM P_j TO P_i , P_i SENDS TO P_j AN $\text{Initiate}(w_1, L_1, \text{state})$

Two CASES:

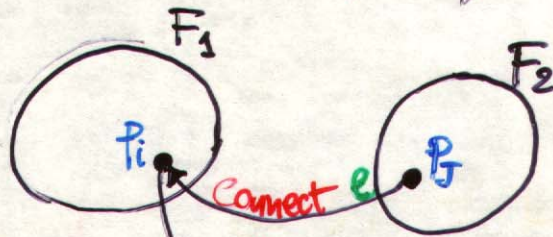
① State = Find \rightarrow NO PROBLEM, SINCE IN THIS CASE F_2 WILL PARTICIPATE IN THE SEARCH OF THE MOE OF F_1 ($= F_1 \cup F_2$).

② State = Found $\rightarrow p_i$ HAS ALREADY SENT A Report MESSAGE.

POTENTIALLY, THIS MIGHT CAUSE A PROBLEM!! BUT:

CLAIM THE MOE OF F_1 IS ALSO THE MOE OF F_1 .
($= F_1 \cup F_2$)

PROOF THE Connect-EDGE CANNOT BE THE MIN-WEIGHT OUTGOING EDGE OF p_i (OTHERWISE p_i WOULD BE LOCKED FROM $p_j \rightarrow$ NO Report).



$w(e') < w(e)$ $\rightarrow e'$: MIN-WEIGHT OUTGOING EDGE OF p_i

$\rightarrow w(\text{MOE}(F_1)) \leq w(e') < w(e) = w(\text{MOE}(F_2)) \leq w(\text{ANY OUTGOING EDGE OF } F_2)$



MESSAGE COMPLEXITY

LEMMA: A FRAGMENT OF LEVEL L CONTAINS AT LEAST 2^L NODES.

Proof: BY INDUCTION.

$L=0 \rightarrow$ TRIVIAL

ASSUME TRUE UP TO FRAGMENTS OF LEVEL $L-1$.

LET $F: (w, L)$

\rightarrow F WAS CREATED

- EITHER AFTER COMBINATION OF F_1 AND F_2 OF LEVEL $L-1$:
 $|F| = |F_1| + |F_2| \geq 2^{L-1} + 2^{L-1} = 2^L$
- OR AFTER ABSORPTION OF A LEVEL- $L' < L$ FRAGMENT F'
 \rightarrow APPLY RECURSIVELY TO $F \setminus F'$

THEOREM: GHS REQUIRES $O(m + n \log n)$ MESSAGES.

Proof: $\left. \begin{array}{l} \text{Connect: } \leq 2 \quad \forall \text{ EDGE} \\ \text{Test-Reject: } \leq 4 \quad \forall \text{ EDGE} \end{array} \right\} O(m)$

ANY NODE SENDS/
RECEIVES

- 1 Initiate
- 1 Test-Accept
- 1 Change Core
- 1 Report

EACH TIME THE LEVEL OF ITS FRAGMENT INCREASES

\rightarrow IT CAN GO THROUGH $\leq \log n$ LEVELS

$\rightarrow O(m + n \log n)$

Shared Memory SYSTEM

(ASYNCHRONOUS)

Processors communicate via a set of shared variables, instead of passing messages. REGISTER

Each shared variable has a **type**, defining a set of operations that can be performed *atomically*. READ/WRITE
READ/MODIFY/WRITE

ACCESS PATTERN: SINGLE/MULTIPLE

Changes to the model from the message-passing case:

- no *inbuf* and *outbuf* state components NO MESSAGES !!
- configuration includes values of shared variables
- only event type is a computation step by a processor.

When p_i takes a step:

↳ NO DELIVERY !!

- p_i 's state in old config specifies which shared variable is to be accessed and with which operation
- operation is done; variable's value in the new config changes according to operation's semantics
- p_i 's state in new config changes according to its old state and result of operation

PRODUCT OF STEPS

SYSTEM: n PROCESSORS P_1, \dots, P_n
 m REGISTERS R_1, \dots, R_m

CONFIGURATION: $C = (q_1, \dots, q_n, r_1, \dots, r_m)$

EVENT: COMPUTATION STEP BY x PROCESSOR ϕ

EXECUTION
SEGMENT: $C_0, \phi_0, C_1, \phi_1, C_2, \phi_2, \dots$

C_{k+1} IS THE RESULT OF APPLYING THE
TRANSITION FUNCTION OF P_{ϕ_k} TO ITS
STATE IN C_k , AND APPLYING P_{ϕ_k} MEMORY
ACCESS OPERATIONS TO THE REGISTERS IN C_k .

$C_k \xrightarrow{\phi_k} C_{k+1}$ STEP OF P_{ϕ_k}

SCHEDULE: $\phi_0, \phi_1, \phi_2, \dots$

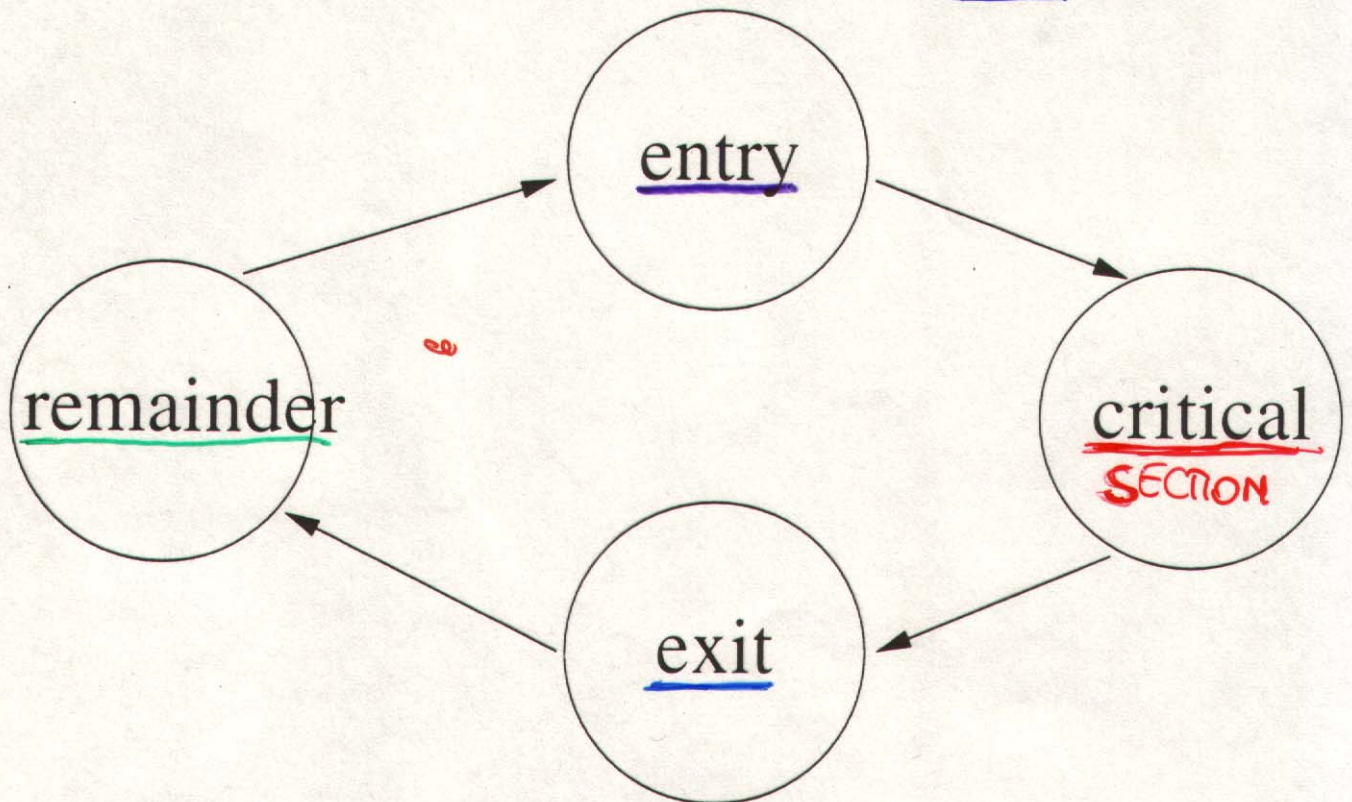
EXECUTION: EXECUTION SEGMENT WITH C_0
AS INITIAL CONFIGURATION OF THE SYSTEM

ADMISSIBLE
EXECUTION: IN AN INFINITE EXECUTION, EACH
PROCESSOR TAKES INFINITE STEPS.

C : CONFIGURATION $\sigma = i_1, i_2, \dots$ SCHEDULE $\rightarrow (C, \sigma)$ APPLICATION
OF σ TO C

Mutual Exclusion Problem




Each processor's code is divided into four sections:



- entry section: synchronize with others to ensure mutually exclusive access to the...
- critical section: use some resource; when done, enter the...
- exit section: clean up, and then enter the...
- remainder section: not interested in the critical section

Mutual Exclusion Algorithms

A *mutual exclusion algorithm* specifies code for entry and exit sections to ensure that:

- **mutual exclusion:** at most one processor is in its critical section at any point, and
- either no deadlock: if a processor is in its entry section at some point, then later some processor is in its critical section, 
- or no lockout: if a processor is in its entry section at some point, then later the same processor is in its critical section,  
- or bounded waiting: no lockout + while a processor is in its entry section, other processors enter the critical section no more than a certain number of times.

LIVENESS CONDITIONS

Algorithm is allowed to assume:

- no processor stays in its critical section forever
- variables used in the entry and exit sections are not accessed during the critical and remainder sections

Overview of Mutual Exclusion Results

The main complexity measure of interest for shared memory mutual exclusion algorithms is the amount of shared space necessary, which is affected by:

- how powerful the type of the shared variables
- how strong the liveness condition to be satisfied

For most powerful shared variables (read-modify-write), number of different states of the shared memory is:

	upper bound	lower bound
ND	2 (T&S alg.)	2 (obvious)
NL	$\frac{n}{2} + c$ (Burns et al.)	$\frac{n}{2}$ (Burns et al.)
BW	n^2 (queue alg.)	n (Burns & Lynch)

Overview of Mutual Exclusion Results (cont'd)

If the shared variables are the weak read/write kind, we measure the number of distinct variables needed.

	upper bound	lower bound
ND		n (Burns & Lynch)
NL	$3n$ boolean (tournament alg.)	
BW	$2n$ unbounded (bakery alg.)	

Overview of Mutual Exclusion Results (cont'd)

If the shared variables are the weak read/write kind, we measure the number of distinct variables needed.

	upper bound	lower bound
ND		n (Burns & Lynch)
NL	$3n$ boolean (tournament alg.)	
BW	$2n$ unbounded (bakery alg.)	

Analysis of Bakery Algorithm

Mutual Exclusion: This follows by showing that processor in critical section has the unique smallest Number (breaking ties with ids) among all contending processors.

No Lockout: This follows by observing that the contending processor p_i with the smallest Number (breaking ties with ids) will be next — every other processor that picks a Number while p_i is waiting will get a larger one.

Space Complexity: Number of shared variables is $2n$; Choosing variables are binary but Number variables are unbounded.

Lemma 1 IF p_i IS IN THE CRITICAL SECTION, AND,
FOR SOME $k \neq i$ $\text{number}[k] \neq 0 \rightarrow$

$$(\text{number}[k], k) > (\text{number}[i], i)$$

Proof SINCE p_i IS IN THE CS, IT PASSED THE SECOND
WAIT STATEMENT FOR $J=k$. THERE ARE 2 CASES:

1) p_i READ $\text{number}[k] = 0$. IN THIS CASE:

EITHER: p_k WAS IN THE REMAINDER SECTION

OR DID NOT FINISH IN CHOOSING ITS NUMBER.

BUT, p_i ALREADY PASSED THE FIRST WAIT STATEMENT
WITH $\text{choosing}[k] = \text{FALSE}$ AND HAD ALREADY CHOSEN
ITS NUMBER.

$\rightarrow p_k$ STARTED READING number AFTER p_i

$$\rightarrow \text{number}[i] < \text{number}[k]$$

2) p_i READ THAT $(\text{number}[k], k) > (\text{number}[i], i)$. IN THIS
CASE, THIS REMAINS TRUE UNTIL p_i EXITS THE CS
OR p_k DOES NOT CHOOSE A NEW NUMBER (THAT
WILL BE, IN ANY CASE, LARGER THAN $\text{number}[i]$).

LEMMA 2 IF p_i IS IN THE CS, THEN $\text{number}[i] > 0$.

Proof BY INDUCTION, IT IS EASY TO PROVE THAT $\text{number}[j] > 0$
FOR ANY p_j . SINCE p_i IS IN THE CS, IN THE ENTRY
SECTION CHOSE A NUMBER $> \text{number}[j]$, $\forall j$, i.e., > 0 .

THEOREM: THE BA PROVIDES MUTUAL EXCLUSION.

Proof: BY CONTRADICTION.

ASSUME p_i AND p_j ARE SIMULTANEOUSLY IN THE CS.

→ FROM LEMMA 2, $\text{number}[i], \text{number}[j] \neq 0$,

AND FROM LEMMA 1, $(\text{number}[i], i) > (\text{number}[j], j)$

AND $(\text{number}[j], j) > (\text{number}[i], i)$, A CONTRAD. ■

THEOREM THE BA PROVIDES NO LOCKOUT.

Proof: BY CONTRADICTION.

ASSUME THERE EXIST STARVED PROCESSORS, AND LET p_i BE THE STARVED PROCESSOR WITH MINIMUM $(\text{number}[i], i)$. ALL PROCESSORS CHOOSING A NUMBER AFTER p_i WILL RECEIVE A LARGER TICKET, AND THEREFORE WILL NOT ENTER THE CS BEFORE p_i . ALL PROCESSORS WITH SMALLER TICKET WILL ENTER THE CS (SINCE NO STARVED) AND EXIT IT.

→ p_i WILL EVENTUALLY PASS THE WAIT STATEMENTS AND ENTER THE CS, A CONTRADICTION. ■

SPACE COMPLEXITY

→ 2-N SHARED UNBOUNDED VARIABLES.

IN FACT, $\text{number}[i] = 0 \forall i$ IFF ALL THE

PROCESSORS ARE IN THE REMAINDER SECTION.

Approfondimento: PROVARE O CONFUTARE LA SEGUENTE

AFFERMAZIONE: THE BA ALGORITHM IS BOUNDED WAITING.

BOUNDED ME ALGORITHM

FIRST STEP: BOUNDED ME ALGORITHM FOR 2 PROCESSORS WHICH **ALLOWS LOCKOUT** P_0, P_1

2 BOOLEAN VARIABLES: $W[i]$

- 0 OTHERWISE
- 1 IF P_i IS INTERESTED IN ENTERING THE CS

WRITTEN BY P_i
READ BY P_{1-i}

THE ALGORITHM IS **ASYMMETRIC**: PRIORITY IS GIVEN TO P_0 : P_1 ENTERS THE CS IFF P_0 IS NOT INTERESTED IN IT AT ALL!

CODE FOR P_0

<Entry>

$W[0] := 1$
WAIT UNTIL ($W[1] = 0$)

<Critical Section>

<Exit>

$W[0] := 0$

CODE FOR P_1

<Entry>

L: $W[1] := 0$
WAIT UNTIL ($W[0] = 0$)
 $W[1] := 1$
IF ($W[0] = 1$) GOTO L

<Critical Section>

<Exit>

$W[1] := 0$

Approfondimento: DIMOSTRARE CHE L'ALGORITMO GARANTISCE LA ME, L'ASSENZA DI STALLO MA **NON** L'ASSENZA DI BLOCCO.

Bounded 2-Processor ME Algorithm - No Lockout

Uses 3 binary shared variables:

- $W[0]$: written by p_0 and read by p_1 , initially 0
- $W[1]$: vice versa, initially 0
- Priority: written and read by both, initially 0

<Entry>: **CODE FOR P_i $i=0,1$**

1. $W[i] := 0$
2. wait until $W[1-i] = 0$ or
Priority = i
3. $W[i] := 1$
4. if (Priority = $1-i$) then
5. if ($W[1-i] = 1$) then goto Line 1
6. else wait until ($W[1-i] = 0$)
7. <Critical Section>

<Exit>:

8. Priority := $1-i$
9. $W[i] := 0$

No Deadlock for 2-Processor Algorithm

(Useful for showing no lockout.)

If one processor (say p_1) ever enters its remainder section for good, then the other processor (say p_0) cannot be starved, since it will keep seeing $W[1] = 0$.

So any deadlock would starve *both* processors.

WLOG, suppose Priority gets stuck at 0 after both processors are stuck in their entry sections.

Thus p_0 is not stuck in Line 2, skips Line 5, and is stuck in Line 6.

Thus p_1 skips Line 6 and is stuck in Line 2 with $W[1]$ stuck at 0.

But then p_0 gets unstuck and enters the critical section.



No Lockout for 2-Processor Algorithm

Suppose in contradiction p_0 (WLOG) is starved. 

Since there is no deadlock, p_1 subsequently goes critical infinitely often.

The first time that p_1 executes Line 8 after p_0 gets stuck in its entry section, Priority gets stuck at 0.

Then p_0 is stuck in Line 6, waiting for $W[1]$ to equal 0, with $W[0]$ = 1.

But the next time p_1 enters its entry section, it gets stuck in Line 2 with $W[1]$ = 0. This contradicts no deadlock.

●
 p_0 in entry

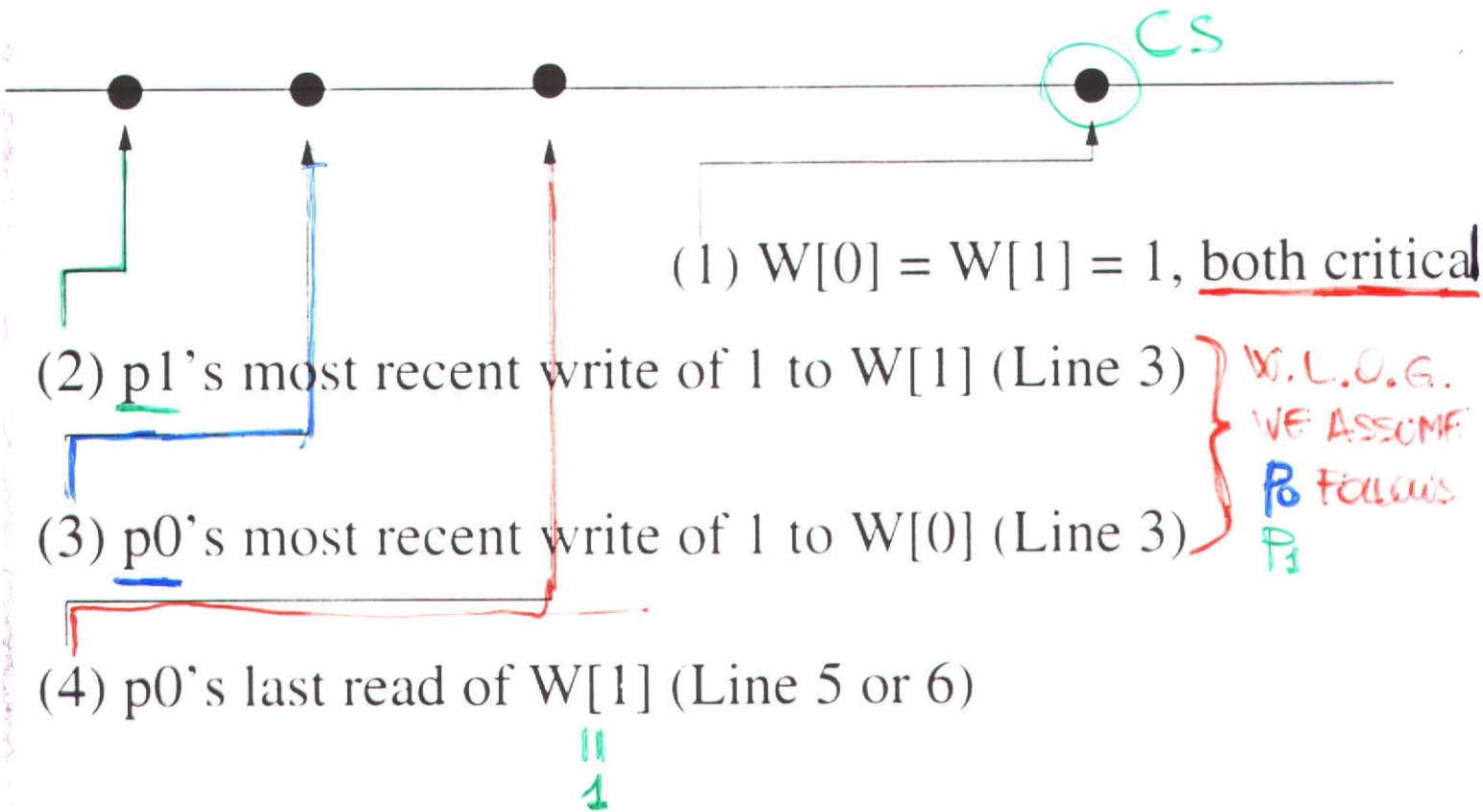
●
 p_1 at Line 8;
Priority = 0
forever after

●
 p_0 stuck in
Line 6 with
 $W[0]$ = 1 forever

●
 p_1 enters entry,
sets $W[1]$ to 0,
stuck in Line 2

Mutual Exclusion for 2-Processor Algorithm

Mutual Exclusion: Suppose in contradiction p_0 and p_1 are simultaneously critical.



WLOG suppose (2) precedes (3). But then in (4), p_0 reads 1, not 0, and thus p_0 cannot be critical at (1).

p_0 CAN ENTER THE CS ONLY IF $W[1] = 0$ (LINES 5 AND 6 OF THE ALGORITHM).

Tournament Algorithm (cont'd)

Pseudocode ~~in book~~ is recursive: HP: $n = 2^{k+1}$

- p_i begins at node $2^k + \lfloor \frac{i}{2} \rfloor$, playing the role of $p_{i \bmod 2}$, where $k = \lceil \log n \rceil - 1$.
- After winning at node v , "critical section" for node v is competition for v 's parent, node $\lfloor \frac{v}{2} \rfloor$, playing role of $p_{v \bmod 2}$ in 2-proc. algorithm.

Correctness: Based on the correctness of the 2-processor algorithm and the tournament structure.

- Projection of an admissible execution of tournament algorithm onto a particular node produces an admissible execution of 2-proc. algorithm.
- ME for tournament algorithm follows from ME for 2-proc. algorithm at the root node.
- NL for tournament algorithm follows from NL for the 2-proc. algorithms at all nodes of tree.

APPROFONDIMENTO:

What about bounded waiting? No.

Space Complexity: $3n$ boolean read/write variables.

procedure NODE (v : integer; side: {0,1})

L: $WANT_{SIDE}^v := \emptyset$

WAIT UNTIL ($WANT_{1-SIDE}^v = \emptyset$ OR $PRIORITY^v = SIDE$)

$WANT_{SIDE}^v := 1$

IF ($PRIORITY^v = 1 - SIDE$) THEN

IF ($WANT_{1-SIDE}^v = 1$) THEN GOTO L

ELSE WAIT UNTIL ($WANT_{1-SIDE}^v = \emptyset$)

IF ($v = 1$) THEN /* AT THE ROOT

< CRITICAL SECTION >

ELSE NODE ($\lfloor v/2 \rfloor, v \bmod 2$)

< Exit >

$WANT_{SIDE}^v := \emptyset$

$PRIORITY^v := 1 - SIDE$

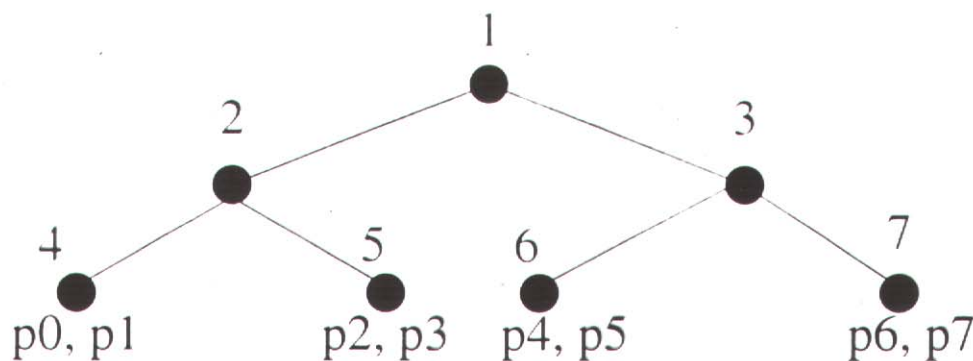
END PROCEDURE

Tournament Algorithm

No lockout mutual exclusion algorithm for n processors using bounded size variables:

- based on a tournament tree, complete binary tree with $n - 1$ nodes
- A copy of the 2-processor algorithm is associated with each node of the tree
- Each proc. begins at a specified leaf, two per leaf
- A proc. proceeds to next level in tree by winning the 2-processor competition for current node:
 - on left side, play role of p_0
 - on right side, play role of p_1
- when processor wins at root, it enters critical section

$n=8$



FAULT-TOLERANCE

FAILURES: $\left\{ \begin{array}{l} \text{BENIGNE (CRASH)} \\ \text{BYZANTINE (ARBITRARY BEHAVIOUR)} \end{array} \right.$

SECOND PART: $\left\{ \begin{array}{l} - \text{COORDINATED ATTACK PROBLEM (CAP)} \\ - \text{CONSENSUS PROBLEM (CP)} \end{array} \right.$

IN SYNCHRONOUS MPS WITH BENIGN FAILURES

SECOND PART: CP IN SYNC MPS WITH BYZANTINE FAILURES

2 ALGORITHMS: $\left\{ \begin{array}{l} \text{OPTIMAL \# OF ROUNDS BUT} \\ \text{EXPONENTIAL MESSAGE COMPLEXITY} \\ \text{DOUBLE \# OF ROUNDS BUT} \\ \text{POLYNOMIAL MESSAGE COMPLEXITY} \end{array} \right.$

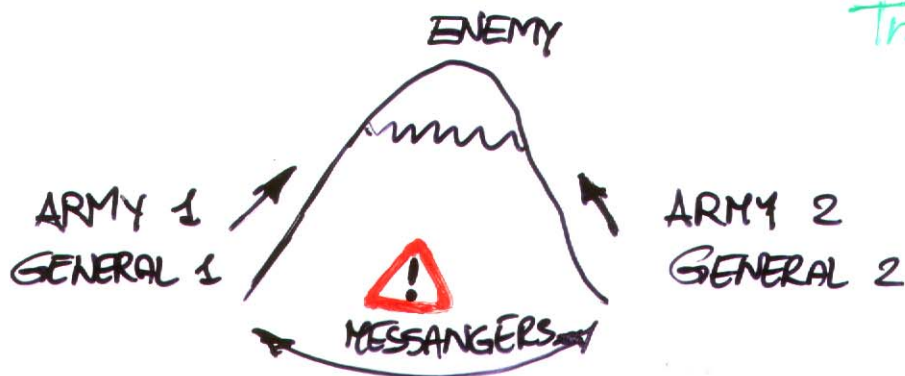
THIRD PART: IMPOSSIBILITY OF SOLVING CP (EVEN IN THE CASE OF ONLY ONE BENIGN FAILURE)

FOR ASYNCHRONOUS SYSTEMS

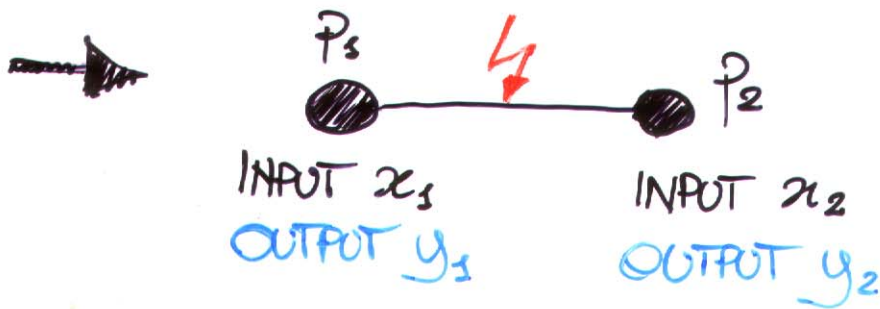
BOTH MPS AND SMS

THE COORDINATED ATTACK PROBLEM (CAP)

SYNC MPS WITH BENIGN FAILURES



THE 2-GENERALS PROBLEM



AGREEMENT: $y_1 = y_2$

VALIDITY: IF $x_1 = x_2 = 0$ AND NO MESSAGES ARRIVE AT P_1 AND P_2 THEN $y_1 = y_2 = 0$

NON-TRIVIALITY: THERE IS AN EXECUTION IN WHICH $y_1 = y_2 = 1$

NO SOLUTION!!

Def: LET α BE AN EXECUTION AND LET P_i BE A PROCESSOR. THE **VIEW** OF P_i IN α , $\alpha|P_i$, IS THE SUBSEQUENCE OF **COMPUTATION** AND **MESSAGE DELIVERY** EVENTS THAT OCCUR IN P_i .

SIMILARITY BETWEEN EXECUTIONS:

α_1, α_2 EXECUTIONS: $\alpha_1 \stackrel{P_i}{\sim} \alpha_2 \iff \alpha_1 / P_i = \alpha_2 / P_i$

REMARK: IF $\alpha_1 \stackrel{P_i}{\sim} \alpha_2$, THEN P_i MAKES THE SAME DECISIONS IN α_1 AND α_2 .

TH: THERE IS NO ALGORITHM THAT SOLVES THE CAP.

Proof: (BY CONTR).

$\exists \beta_1$ EXECUTION S.T. $y_1 = y_2 = 1$. LET k BE THE # OF MESSAGES SENT IN β_1 .

W.L.O.G., ASSUME LAST MESSAGE $M_k = P_1 \rightarrow P_2$.

→ LET α_k BE AN EXECUTION IDENTICAL TO β_1 , EXCEPT M_k IS NOT RECEIVED BY P_2 .

→ $\beta_1 \stackrel{P_1}{\sim} \alpha_k \rightarrow P_1$ DECIDES 1 BOTH IN β_1 AND IN $\alpha_k \rightarrow P_2$ DECIDES 1 AS WELL IN α_k .

THEN, LET α_{k-1} IDENTICAL TO α_k , EXCEPT M_{k-1} IS NOT DELIVERED.

→ $\alpha_{k-1} \stackrel{P_{k-1}}{\sim} \alpha_k$, WHERE $P_{k-1} \in \{P_1, P_2\}$ WAS THE PROCESSOR SENDING M_{k-1} . → ALSO HERE, $y_1 = y_2 = 1$.

⋮
 $\alpha_1 \stackrel{P_1}{\sim} \alpha_2$ WHERE IN α_1 M_1 IS NOT DELIVERED. $y_1 = y_2 = 1$

→ β_0' IDENTICAL TO α_1 , WITH $\alpha_1 = 0 \rightarrow \alpha_1 \stackrel{P_2}{\sim} \beta_0' \rightarrow y_1 = y_2 = 1$

→ β_0 IDENTICAL TO β_0' , WITH $\alpha_2 = 0 \rightarrow \beta_0 \stackrel{P_1}{\sim} \beta_0' \rightarrow y_1 = y_2 = 1$

Fault-Tolerant Consensus

Types of processor failure:

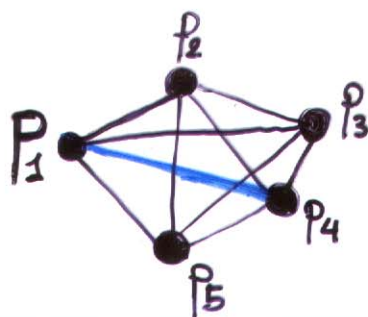
- crash: in middle of step, might only send a subset of messages
- Byzantine: take arbitrary actions

Consensus problem: Every processor has an input. x_i

- Termination: Eventually every nonfaulty processor must decide on a value. y_i
- Agreement: All nonfaulty decisions must be the same. $y_i = y_j \quad \forall p_i, p_j \text{ NON-FAULTY}$
- Validity: If all inputs are the same, then the non-faulty decision must be that input. $y_i \in \{x_1, \dots, x_n\}$

Validity ensures that outputs bear some relationship to inputs (but also rules out easy solutions!).

Background: Collection of armies, all on the same side. Each general begins with an opinion whether to attack. If all attack, they will win, otherwise they will lose. Some generals are traitors and will behave incorrectly.



TOPOLOGY:
CLIQUE

Overview of Consensus Results

Let f be the maximum number of faulty processors.

Tight bounds for synchronous message passing:

	<u>crash failures</u>	<u>Byzantine failures</u>
number of rounds	$f + 1$	$f + 1$
total number of procs	$\geq f + 1$	$\geq 3f + 1$
message size	polynomial	polynomial

- Asynchronous case: impossible in both shared memory and message passing, even if only one crash failure is to be tolerated.

Modeling Processor Failures

For an execution to be **admissible**:

Crash Failures:

All but a set of at most f processors (the **faulty** ones) take an infinite number of steps.

- In synchronous case: once a faulty processor fails to take a step in a round, it takes no more steps.
- In message passing case: In a faulty processor's last step, an arbitrary subset of the processor's outgoing messages make it into the channels. (This is where the difficulties lie.)

Byzantine Failures:

A set of at most f processors (the **faulty** ones) can send messages with arbitrary content and change state arbitrarily (not according to their transition functions).

Consensus Algorithm for Crash Failures

Code for p_i :

$v :=$ my input $\equiv x_i \in \mathbb{N}$

at each round 1 through $f+1$:

{ if I have not yet sent v then
send v to all
 $v :=$ minimum among all received
values and current value of v
in round $f+1$, decide on $v = g_i$

Termination: By the code.

□

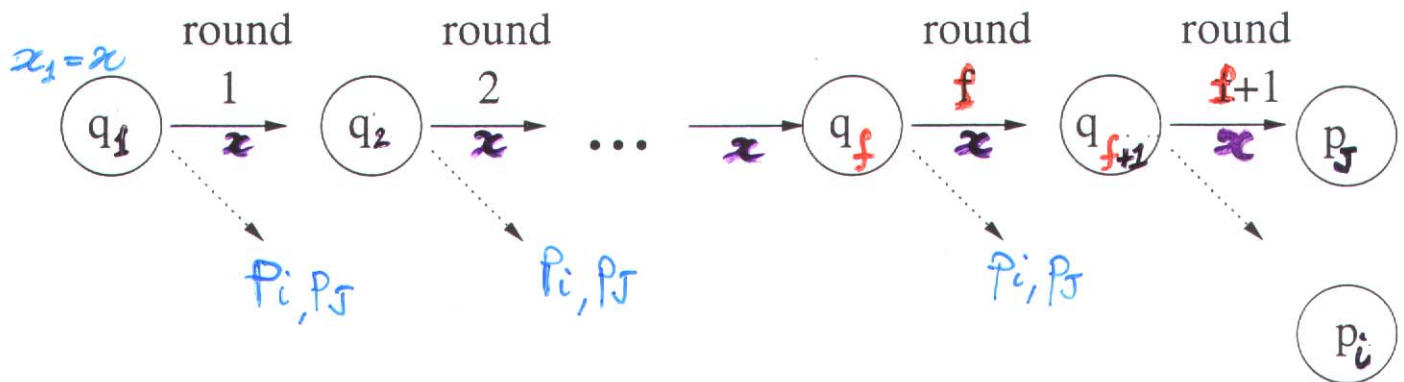
Validity: Holds since processors do not introduce spurious messages: if all inputs are the same, then that is the only value ever in circulation.

□

Analysis of Crash Consensus Algorithm

BY CONTR., ~~p_j < x~~

Agreement: Suppose p_j decides on a smaller value, x , than does p_i . Then x was hidden from p_i by a chain of faulty processors:



There are $f + 1$ faulty processors in this chain, a contradiction.

□

Performance:

- number of processors $n > f$

- $f + 1$ rounds

$O(n^2 \cdot |V|)$ messages each of size $\log |V|$, where V is the input set.

□ \rightarrow # MESSAGES $\leq \frac{n \cdot (n-1)}{2} \cdot \min(|V|, f+1) \leq n^2 |V| = O(n^2 |V|) = O(n^3)$

EDGES

Approfondimento : K-CONSENSUS

$$P = \{p_1, p_2, \dots, p_n\} \quad \text{CLIQUE}$$

$$\text{INPUT: } X = \{x_1, x_2, \dots, x_n\}$$

$$\text{OUTPUT: } Y = \{y_{i_1}, y_{i_2}, \dots, y_{i_m}\}, \quad \text{VALIDITY}$$

$n \geq m \geq n - f$

$y_{i_j} \in X, f < n$

AND THE NUMBER OF DIFFERENT VALUES $\leq K$.

PRESENT A SYNCHRONOUS ALGORITHM WITH

ROUND $\frac{f}{K} + 1$. (ASSUME K DIVIDES f).

CODE FOR p_i

$v := \text{my input}$

at each round 1 through $\frac{f}{K} + 1$

$\left\{ \begin{array}{l} \text{if } \exists \text{ have not yet sent } v, \text{ then} \\ \text{send it to all} \\ v := \text{minimum \{received values, } v \} \\ \text{at last round, decide on } v \end{array} \right.$

MESSAGE COMPLEXITY : $\leq \frac{n \cdot (n-1)}{2} \cdot \min(|V|, \frac{f}{K} + 1) =$

$$O\left(\min\left(\frac{n^3}{K}, n^2 \cdot |V|\right)\right).$$

Byzantine Failures



(LINKS ARE RELIABLE)

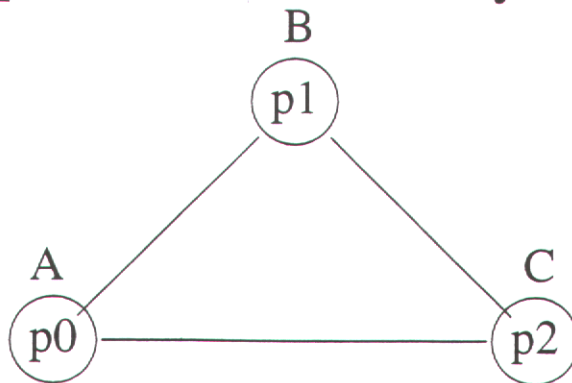
How many processors total are needed to solve consensus when $f = 1$?

- Suppose $n = 2$. If p_0 starts with input 0 and p_1 starts with input 1, then someone has to change, but not both. What if one processor is faulty? How can the other one know?
- Suppose $n = 3$. If p_0 has input 0, p_1 has input 1, and p_2 is faulty, then a tie-breaker is needed, but p_2 might be malicious.

Theorem (5.8): Any consensus algorithm for message passing that tolerates 1 Byzantine failure must have at least 4 processors total.

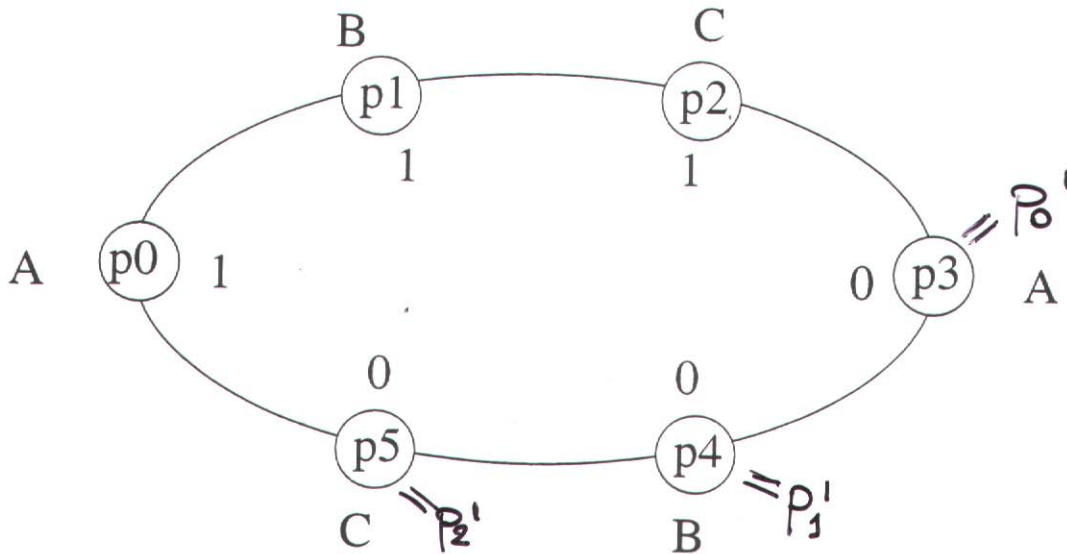
Proof: Suppose there is a consensus algorithm $\mathcal{A} = (A, B, C)$ for 3 processors and 1 Byzantine failure.

→ BY CONTRADICTION



Processor Lower Bound for Byzantine Case

Now consider a ring of six processors running components of A in this fashion:

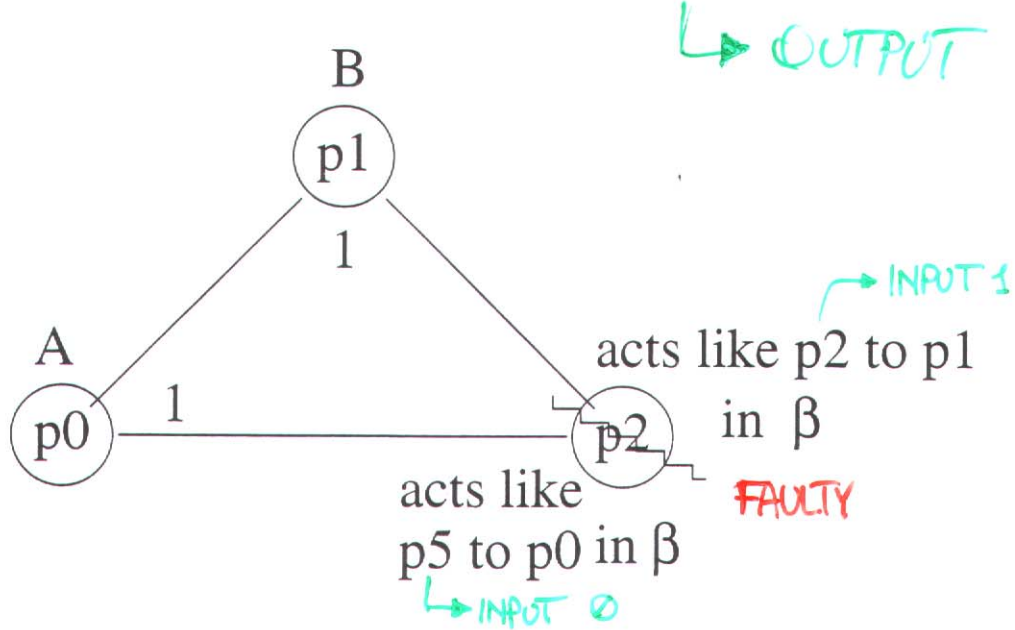


Give each processor the indicated input and let the ring execute. Call the resulting execution β .

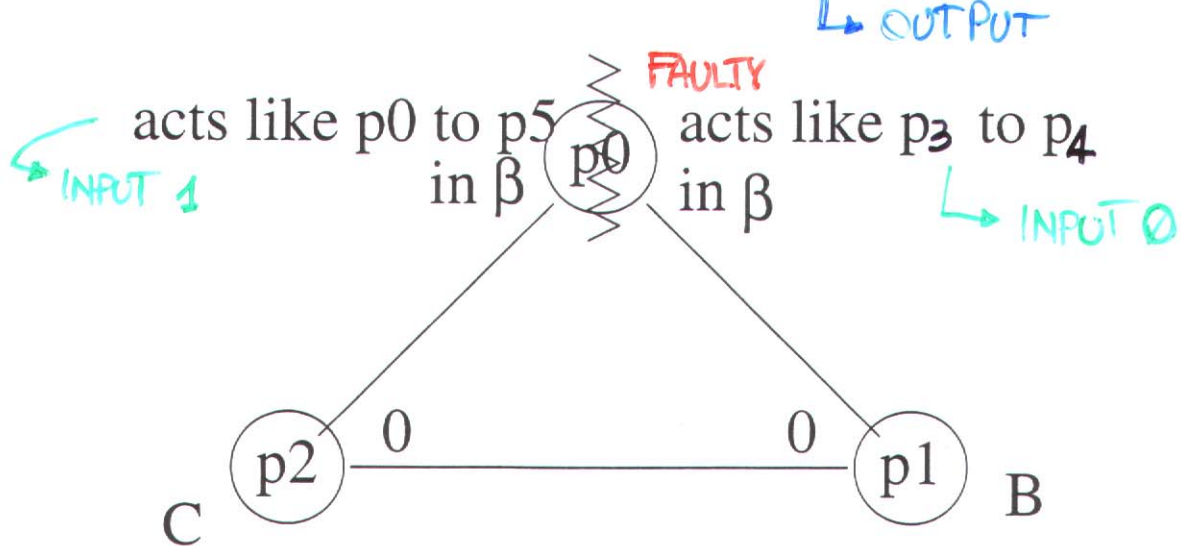
- β does not necessarily solve consensus: it doesn't have to, since the assumptions under which A is supposed to work do not hold.
- However, the processors do something. This behavior will be used to specify the behavior of the faulty processors in certain particularly adversarial executions of A on the triangle.

Processor Lower Bound (cont'd)

Let α_1 be this execution, in which **1** is decided:

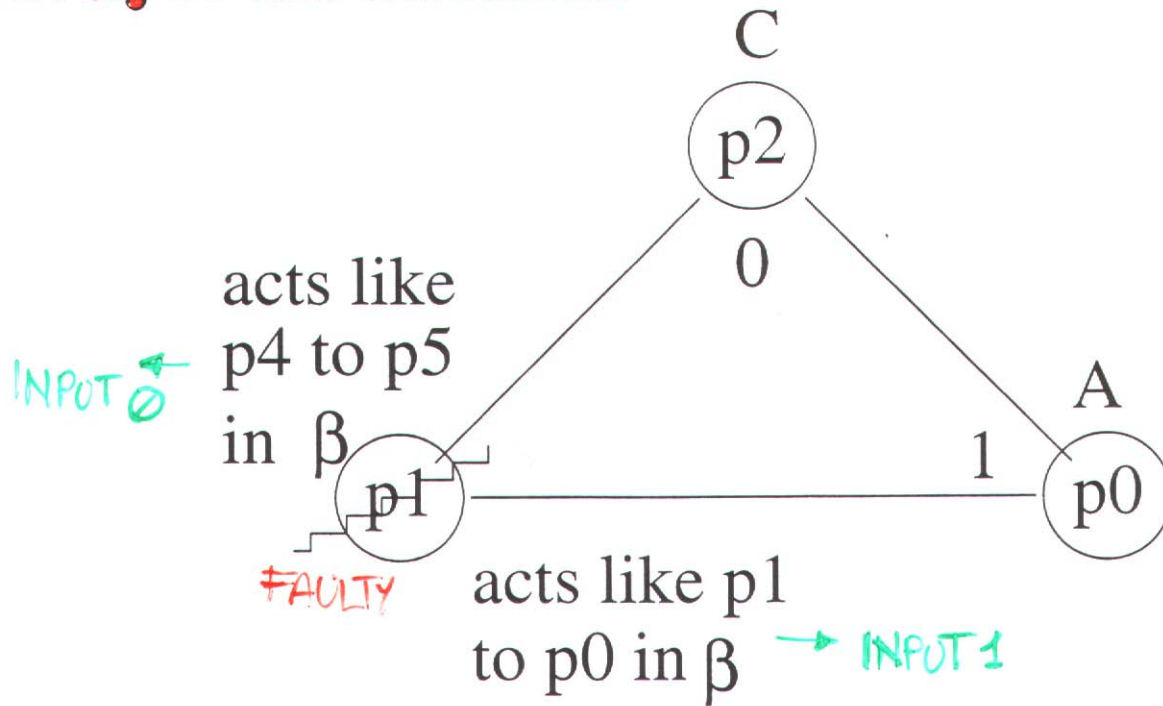


Let α_2 be this execution, in which **0** is decided:



Processor Lower Bound (cont'd)

Let α_3 be this execution:



What is decided in α_3 ?

- p_0 's view in α_3 equals p_0 's view in β , which equals p_0 's view in α_1 . Thus p_0 decides 1 in α_3 .
- p_2 's view in α_3 equals p_2 's view in β , which equals p_2 's view in α_2 . Thus p_2 decides 0 in α_3 .
- But this contradicts agreement.

□

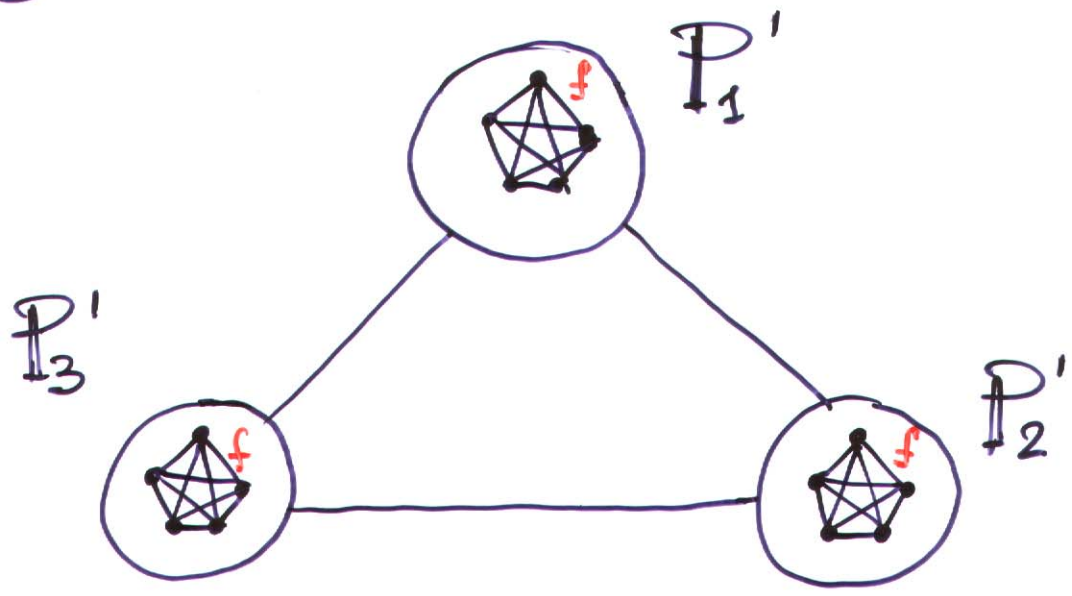
Read reduction in textbook to show $n = 3f$ is impossible for $f > 1$.

THEOREM: IN A SYSTEM WITH n PROCESSORS, ~~WITH~~ WITH AT MOST f OF THEM BYZANTINE ~~PROCESSES~~, THERE IS NO ALGORITHM WHICH SOLVES THE CONSENSUS PROBLEM IF $n \leq 3f$.

PROOF: (BY CONTRADICTION)

FOR THE SAKE OF SIMPLICITY, LET $n = 3f$.

PARTITION THE SET OF PROCESSORS \mathbb{P} INTO 3 SETS $\mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_3$, EACH CONTAINING EXACTLY $\frac{n}{3} = f$ PROCESSORS.



→ if \mathbb{P}_i is FAULTY → AT MOST f PROCESSORS ARE FAULTY IN THE SIMULATED SYSTEM → THE SIMULATED SYSTEM WORKS CORRECTLY → $\mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_3$ WORKS CORRECTLY CONTRADICTION

Consensus Algorithms for Byzantine Failures

Minimum number of rounds is $f + 1$, since crash failures are a special case of Byzantine failures.

Exponential Tree Algorithm HEIGHT: $f + 1$

Each processor maintains a tree data structure in its local state. Each node of the tree is labeled with a sequence of processor indices with no repeats:

- root's label is empty sequence λ (root has level 0)
- root has n children labeled 0 through $n - 1$
- child node labeled i has $n - 1$ children labeled $i : 0$ through $i : n - 1$, skipping $i : i$
- in general, node at level d with label v has $n - d$ children labeled $v : 0$ through $v : n - 1$, skipping any index appearing in v [LENGTH OF THE LABEL: $d+1$]
- nodes at level $f + 1$ are the leaves.

Exponential Tree Algorithm

Each processor fills in the tree nodes with values as the rounds go by:

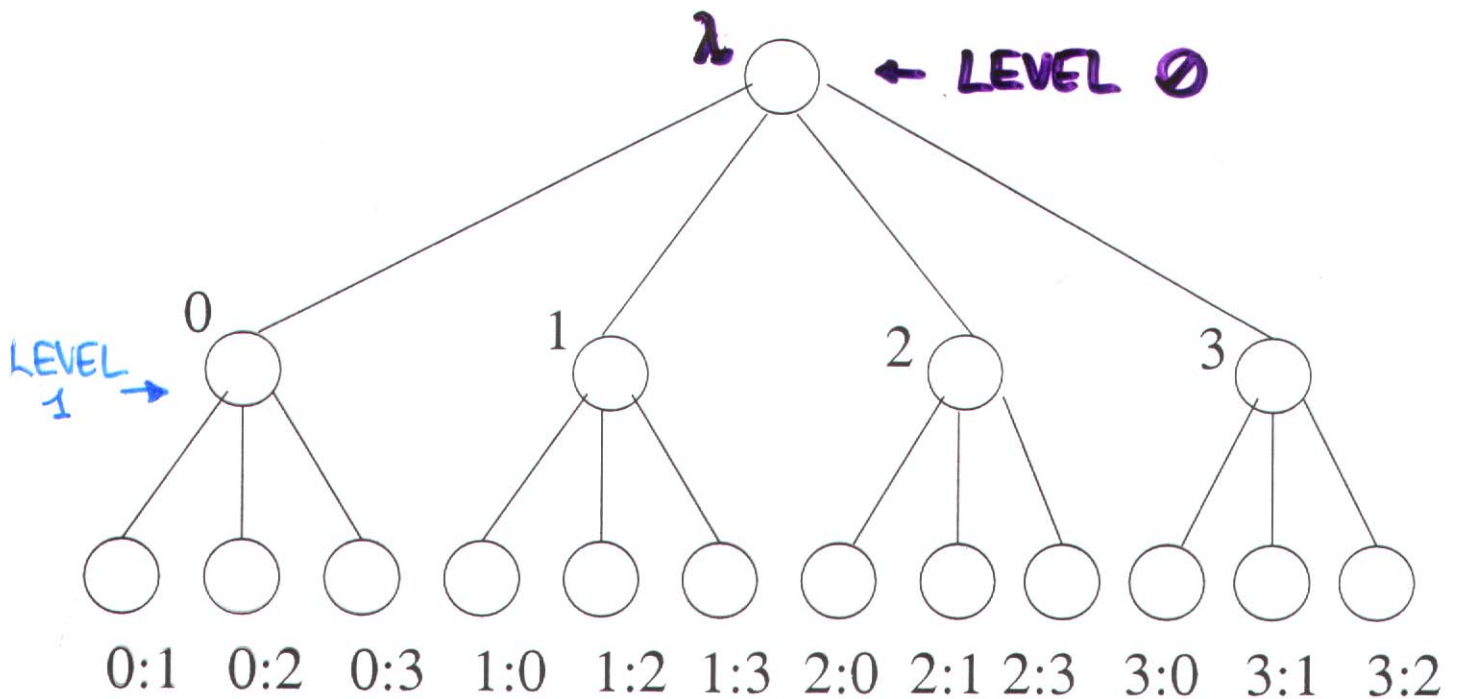
- initially, store your input in the root (level 0)
- round 1: send level 0 of your tree (the root); store value received from p_j in node j (level 1) (default if none)
- round 2: send level 1 of your tree; store value received from p_j for node k in node $k : j$ (level 2) (“the value that p_j told me that p_k told p_j ”) (default if none)
- continue for $f + 1$ rounds

In the last round, each processor uses the values in its tree to compute its decision. The decision is resolve(λ), where resolve(π) equals

- value in tree node labeled π if it is a leaf
- majority { resolve(π') : π' is a child of π } otherwise (default if none).

Example of Exponential Tree

The tree when $n = 4$ and $f = 1$:



→ LEVEL $f+1 = 2$

OF LEVEL d

IN GENERAL, A NODE IN THE TREE IS LABELLED WITH A SEQUENCE:

$$i_1 : i_2 : \dots : i_d$$

" i_d SAYS THAT i_{d-1} SAID THAT i_{d-2} SAID THAT...
 ... THAT i_1 SAID X "

Proof of Exponential Tree Algorithm

LET p_i AND p_j BE NONFAULTY.

Lemma (5.10): Nonfaulty processor p_i 's resolved value for node $\pi = \pi'j$, what p_j reports for π' , equals what p_j has stored for π' .

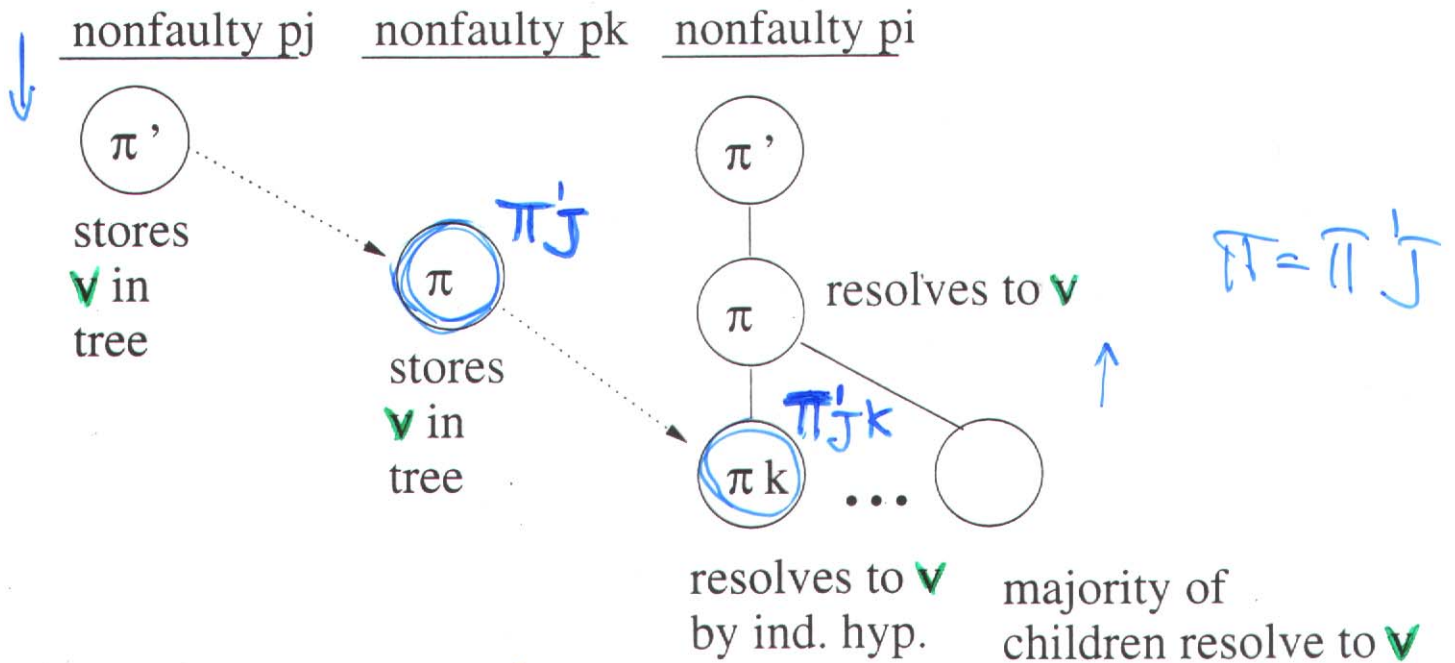
Proof: By induction on the height of π .

Basis: π is a leaf. Then p_i stores in node π what p_j sends it for π' in the last round.. For leaves, the resolved value is the tree value.

Induction: π is not a leaf.

- By tree definition, π has at least $n - f$ children. $> 2f$
- Since $n > 3f$, π has majority of nonfaulty children.
- Let πk be a child of π such that p_k is nonfaulty.
- Since p_j is nonfaulty, p_j correctly reports to p_k that it has some value v in node π' ; thus p_k stores v in node $\pi = \pi'j$.
- By induction, p_i 's resolved value for πk equals the value v that p_k has in its tree node π .
- So all of π 's nonfaulty children resolve to v in p_i 's tree, and thus π resolves to v in p_i 's tree.

Proof of Exponential Tree Algorithm



□

Validity: Suppose all inputs are v .

- Nonfaulty processor p_i decides on $\text{resolve}(\lambda)$, which is the majority among $\text{resolve}(j)$, $0 \leq j \leq n - 1$.
- The previous lemma implies that for each nonfaulty p_i , $\text{resolve}(j)$ is the value stored at the root of p_j 's tree, which is p_j 's input v .
- Thus p_i decides v .

□

Proof of Exponential Tree Algorithm (cont'd)

Agreement: Show that all nonfaulty processors resolve the same value for their tree roots.

F: A node is common if all nonfaulty processors resolve the same value for it. We will show the root is common.

Strategy:

1. Show that every node with a certain property is common.
2. Show that the root has the property.

Lemma (5.11): If every π -to-leaf path has a common node, then π is common.

Proof: By induction on the height of π .

Basis: π is a leaf. Then every π -to-leaf path consists solely of π , and since the path is assumed to contain a common node, that node is π .

Proof of Exponential Tree Algorithm (cont'd)

Induction: π is not a leaf. Suppose in contradiction π is not common.

- Then every child $\pi' = \pi_k$ of π has the property that every π' -to-leaf path has a common node. ~~ASSUME p_k NON-FAULTY~~
(THERE IS A π_k OF NP)
- Since the height of π' is smaller than the height of π , the inductive hypothesis implies that π' is common.
- Therefore all nonfaulty processors compute the same resolved value for π' , and thus π is common.

□ ~~THE ROOT IS COMMON!~~

2) Show every root-to-leaf path has a common node.

- There are $f + 2$ nodes on a root-to-leaf path.
- The label of each non-root node on a root-to-leaf path ends in a distinct processor index: i_1, i_2, \dots, i_{f+1}
- At least one of these indices is that of a nonfaulty processor, say i_k .
- Lemma 5.10 implies that the node whose label ends in i_k is common.

□

Proof of Exponential Tree Algorithm (cont'd)

Complexity:

- $n > 3f$ processors
- $f + 1$ rounds
- Messages in round r contain $n(n-1)(n-2)\cdots(n-(r-2))$ values.

When $r = f + 1$, this is exponential if f is more than constant relative to n .

A Polynomial Algorithm for Byzantine Agreement

We can reduce the message size with a simple algorithm that increases the number of processors to $n > 4f$ and number of rounds to $2(f + 1)$.

Phase King Algorithm

Uses $f + 1$ phases, each taking two rounds.

Code for p_i :

pref := my input x_i

first round of phase k :

send pref to all

receive prefs of others

let maj be the value that occurs $> n/2$ times
among all prefs (0 if none)

let mult be number of times maj occurs

second round of phase k :

if $i = k$ then send maj // I am the phase king
receive tie-breaker from p_k (0 if none)

if mult $> n/2 + f$

then pref := maj

else pref := tie-breaker

if $k = f+1$ then decide pref

Proof of Phase King Algorithm (cont'd)

Agreement:

- Since there are $f + 1$ phases, at least one has a nonfaulty king.
- Lemma 5.14 implies that at the end of that phase, all nonfaulty processors have the same preference.
- Lemma 5.13 implies that from that phase onward, the nonfaulty preferences stay the same.
- Thus the decisions are the same.

□

Performance:

- number of processors $n > 4f$
- $2(f + 1)$ rounds
- $O(n^2 f)$ messages, each of size $\log |V|$.

$O(\log \{\max(V)\})$

V : INPUT SET

Approfondimento: Mostrare un'esecuzione per $n=4$, $f=1$ tale che l'algoritmo PHASE-KING fallisce.

Proof of Phase King Algorithm

VALIDITY

Lemma (5.13): If all nonfaulty processors prefer v at start of phase k then all do at end of phase k .

Proof:

- Each nonfaulty processor receives at least $n - f$ preferences (including its own) for v in the first round of phase k .

→ Since $n > 4f \rightarrow n - f > n/2 + f \rightarrow n - f > n/2$

- Thus the processor still prefers v .

□

Validity: Follows from Lemma 5.13.

□

FIRST ROUND:
CHOOSE v

→ SECOND ROUND:
MAINTAIN v

Proof of Phase King Algorithm (cont'd)

Lemma (5.14): If the king of phase k is nonfaulty, then all nonfaulty processors have the same preference at the end of phase k .

Proof: Consider two nonfaulty processors p_i and p_j .

Case 1: p_i and p_j both use p_k 's tie-breaker. Since p_k is nonfaulty, they agree.

Case 2: p_i uses its majority value and p_j uses the king's tie-breaker.

- p_i 's majority value is v .
- p_i receives more than $n/2 + f$ preferences for v .
- p_k receives more than $n/2$ preferences for v .
- p_k 's tie-breaker is v .

Case 3: p_i and p_j both use their own majority values.

- p_i 's majority value is v .
- p_i receives more than $n/2 + f$ preferences for v .
- p_j receives more than $n/2$ preferences for v .
- p_j 's majority value is also v .

□