Algorithms for UNRELIABLE Distributed Systems:

The consensus problem

Failures in Distributed Systems

Let us go back to the **message-passing model**; it may undergo the following malfunctioning, among others:

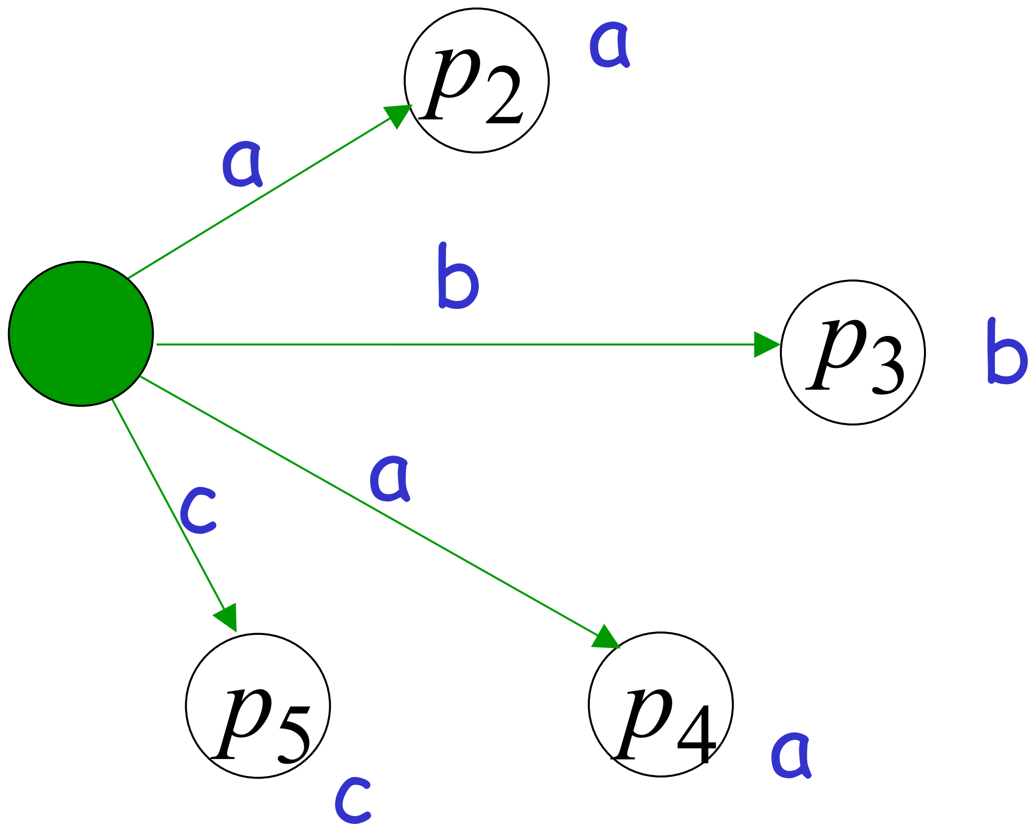
Link failure: A link fails and remains inactive for some time; the network may get disconnected

Processor crash (or benign) failure: At some point, a processor stops forever taking steps; also in this case, the network may get disconnected

Processor Byzantine (or malicious) failure: during the execution, a processor changes state arbitrarily and sends messages with arbitrary content (name dates back to untrustable Byzantine Generals of Byzantine Empire, IV–XV century A.D.); also in this case, the network may get disconnected

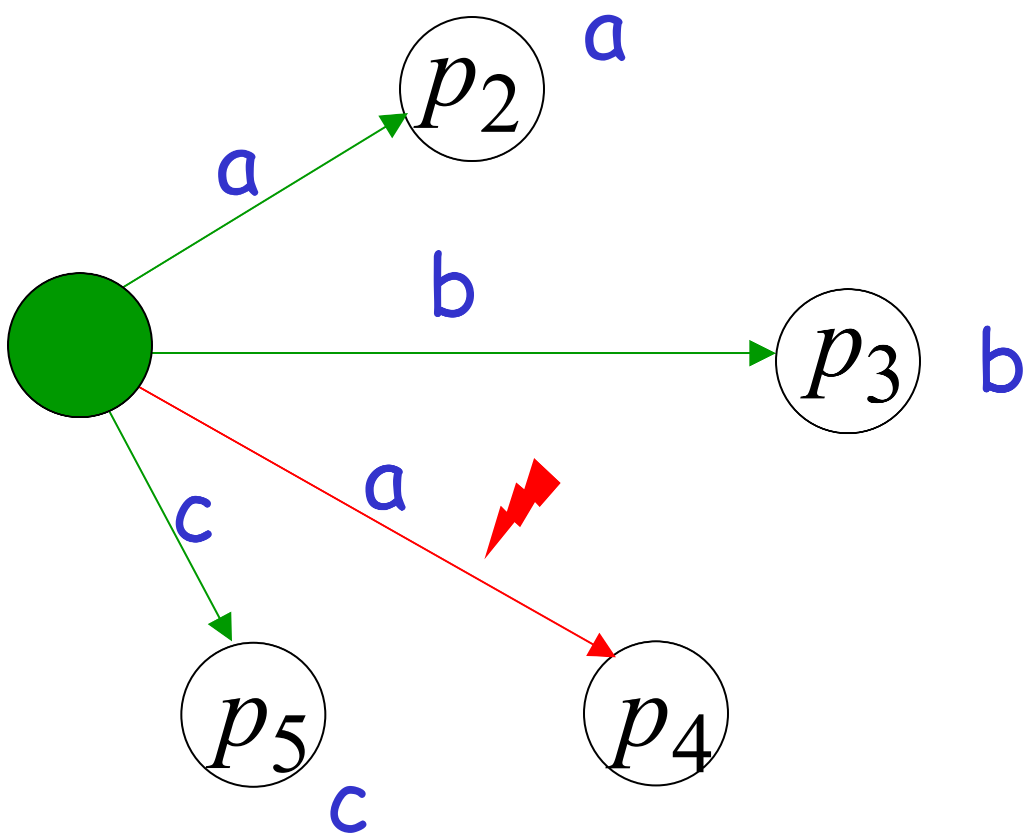
Normal operating

Non-faulty links and nodes



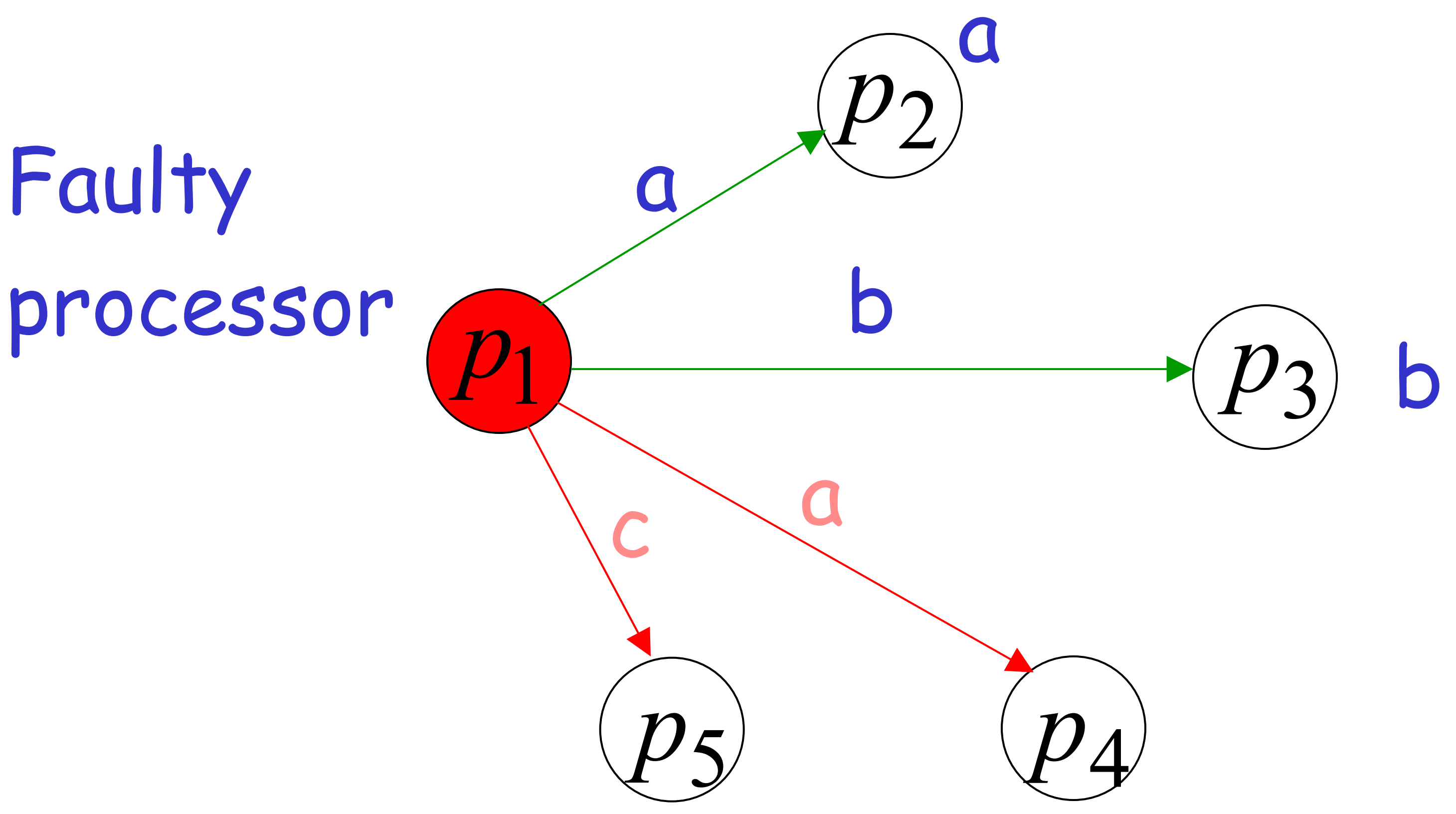
Link (non-permanent) Failures

Faulty
link

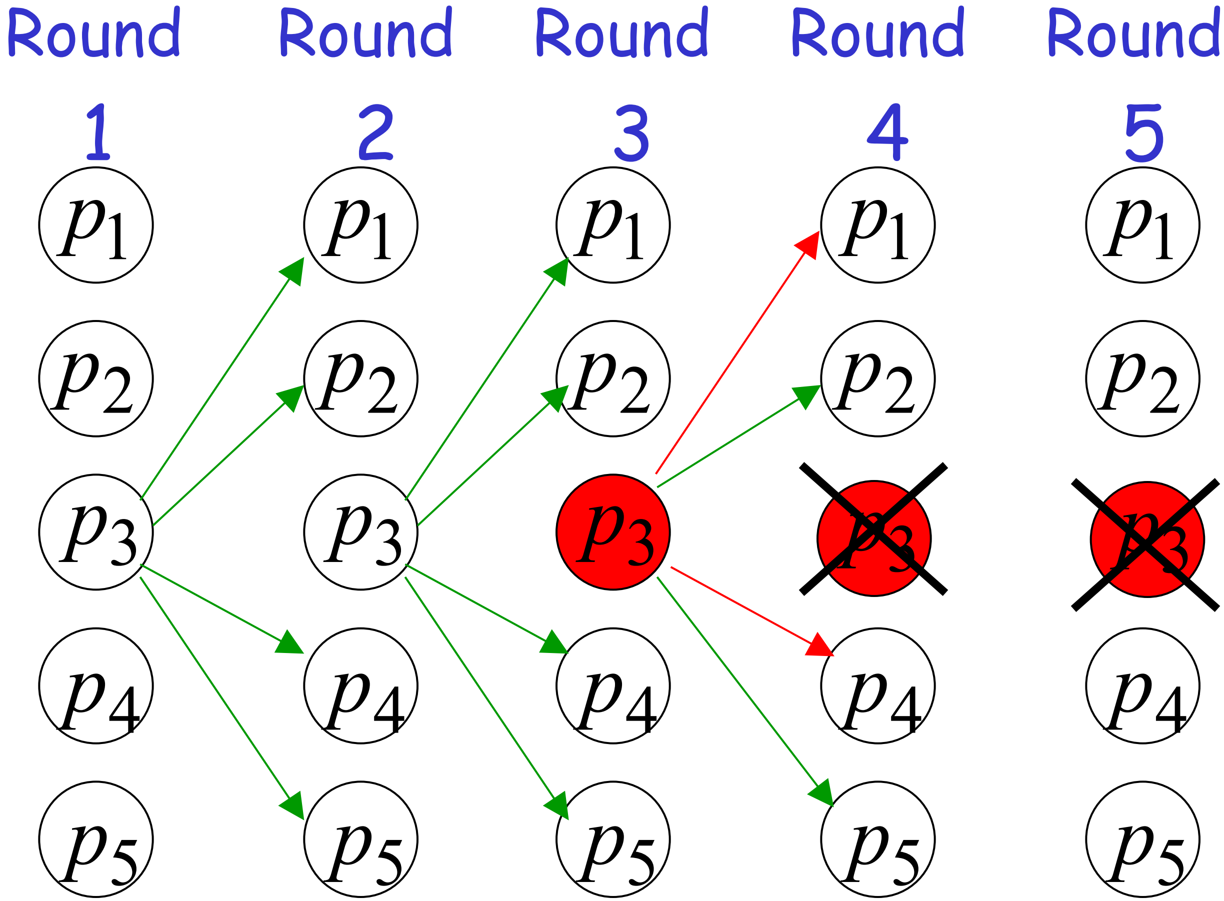


Messages sent on the failed link
are not delivered (for some time), but
they cannot be corrupted

Processor (permanent) crash failure



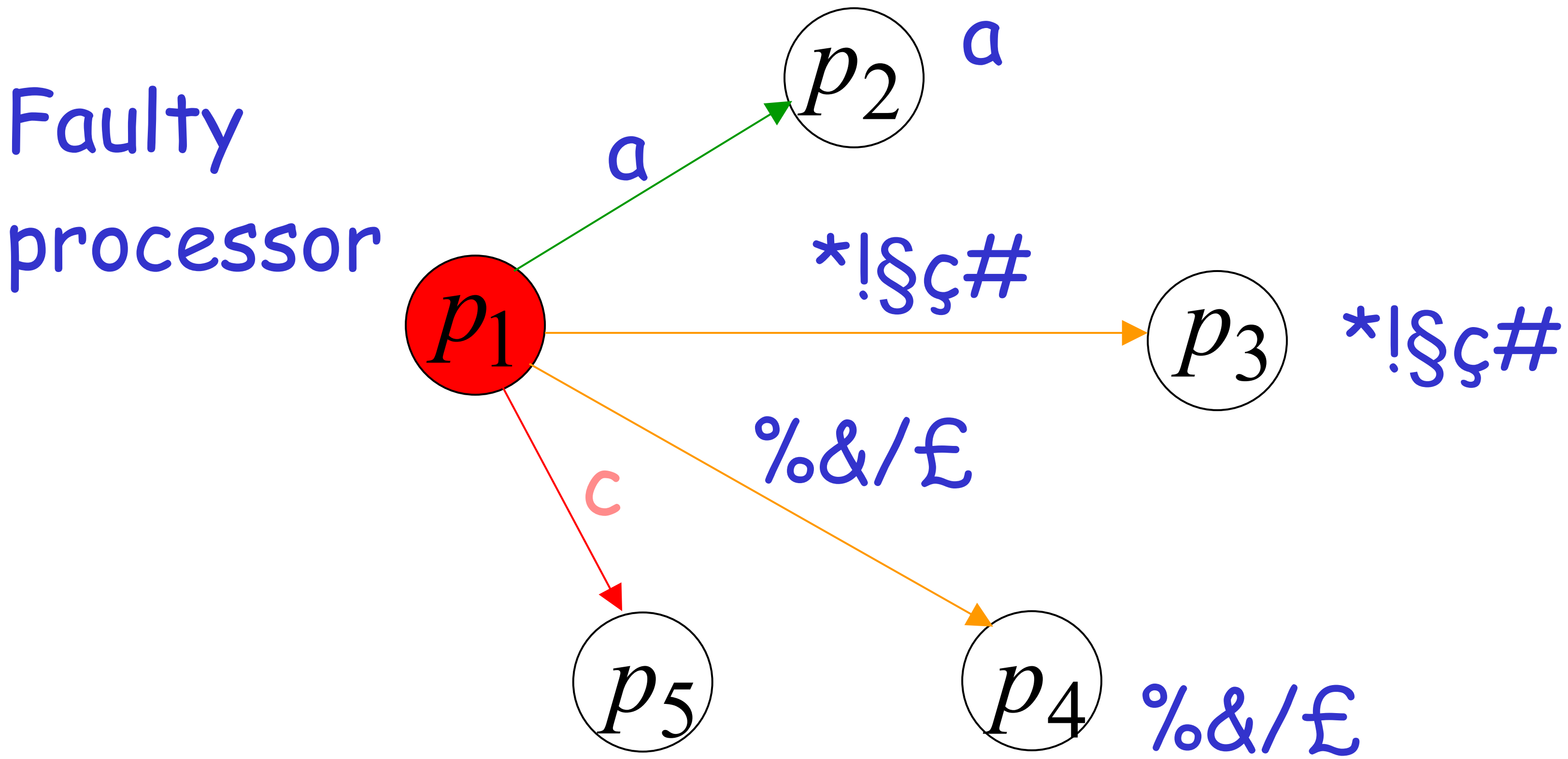
Some of the messages are not sent (forever)



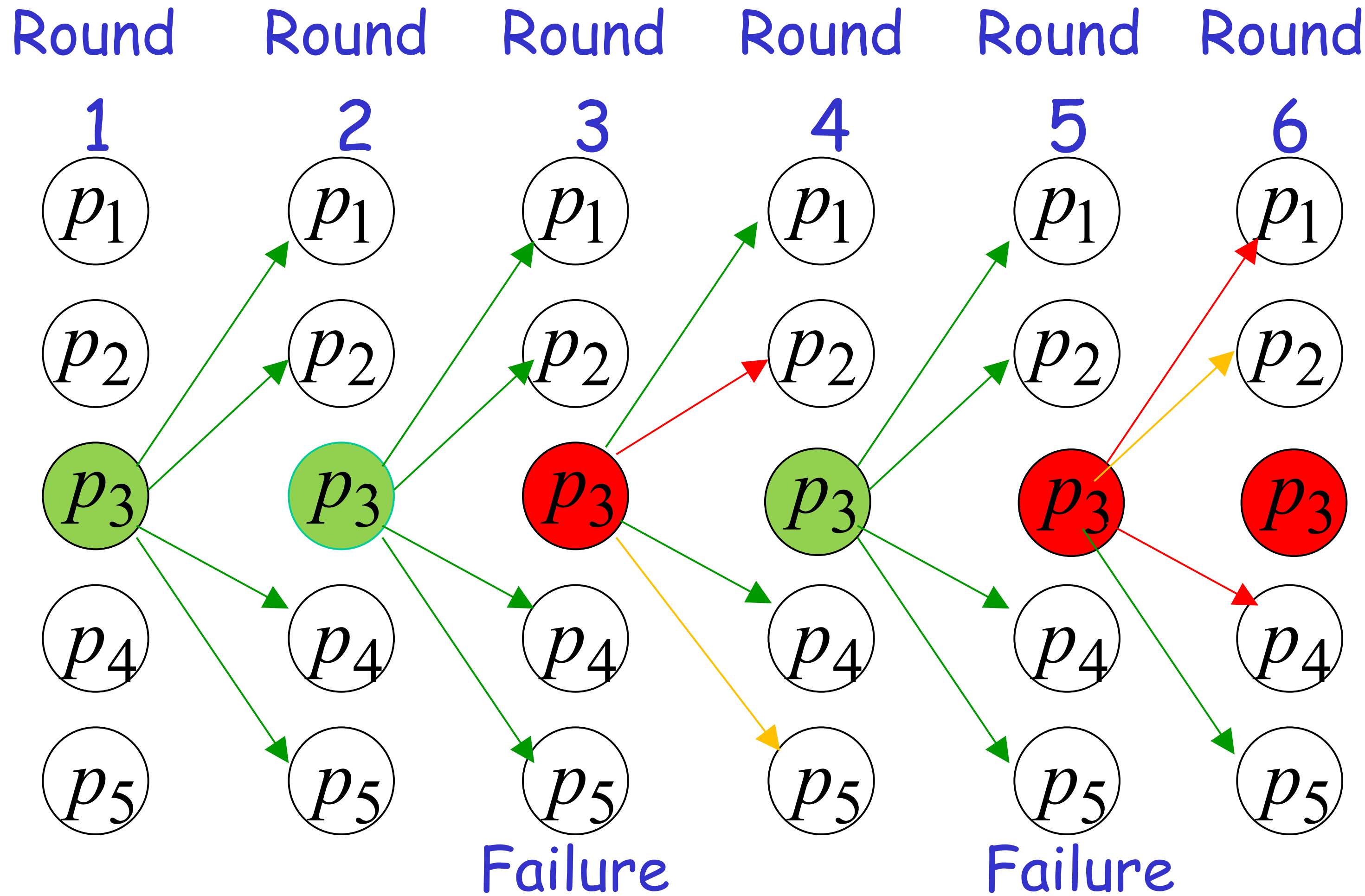
Crash failure in a synchronous MPS

After failure the processor disappears from the network

Processor Byzantine failure



Processor sends arbitrary messages (i.e., they could be either correct or corrupted), plus some messages may be not sent



Byzantine failure in a synchronous MPS

After failure the processor may continue functioning in the network

Consensus Problem

Every processor has an input $x \in X$ (notice that in this way the algorithms running at the processors will depend on their input), and must decide an output $y \in Y$. Assume that link or node failures can possibly take place in the system. Then, design an algorithm enjoying the following properties:

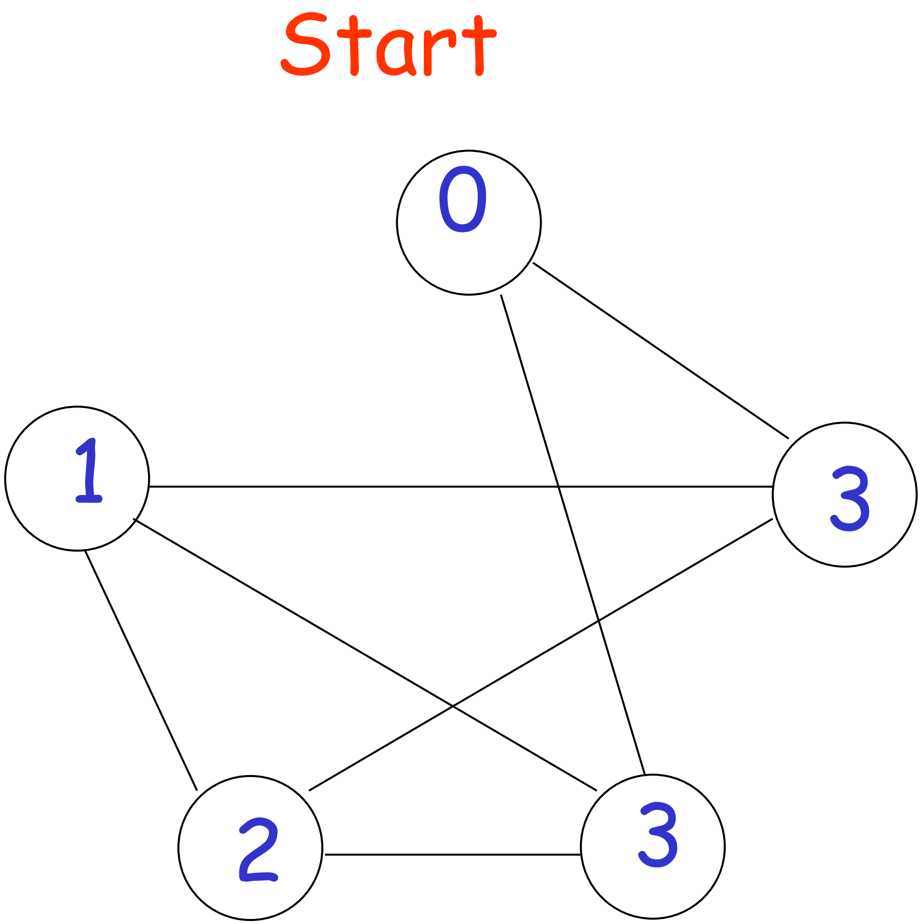
Termination: Eventually, every non-faulty processor decides on a value $y \in Y$.

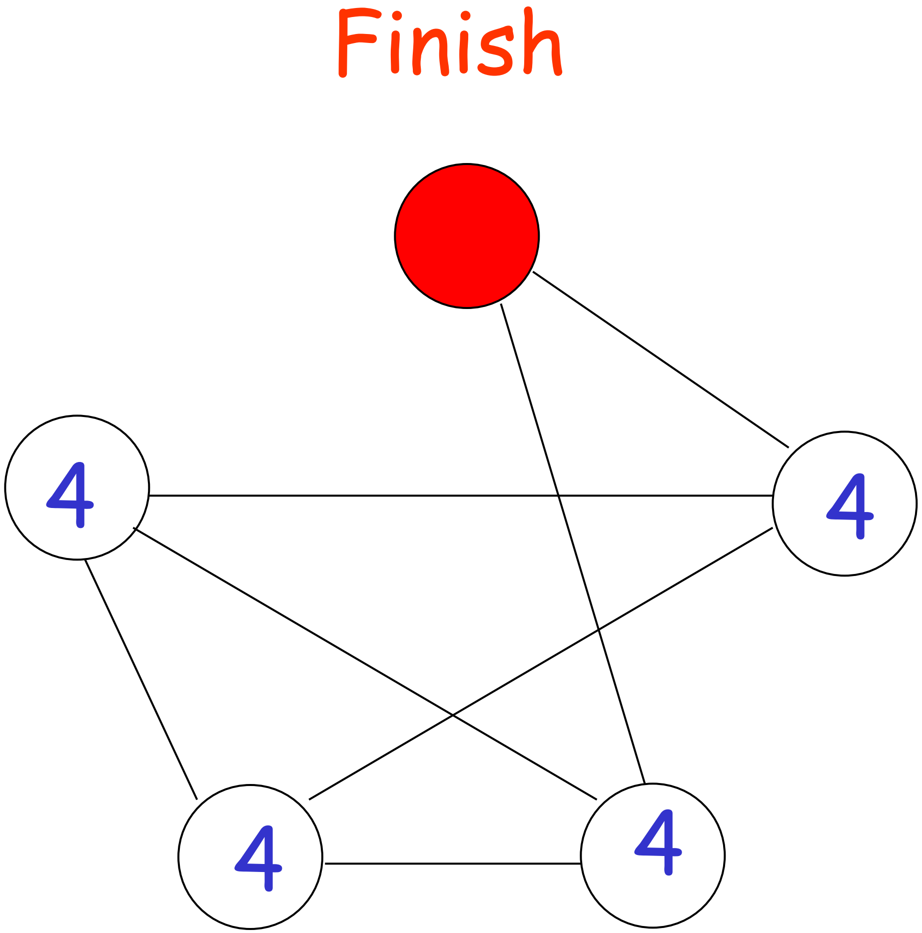
Agreement: All decisions by **non-faulty** processors must be the same.

Validity: If all inputs are the same, then the decision of a non-faulty processor must equal the common input (this avoids trivial solutions).

In the following, we assume that $X=Y=\mathbb{N}$

Agreement





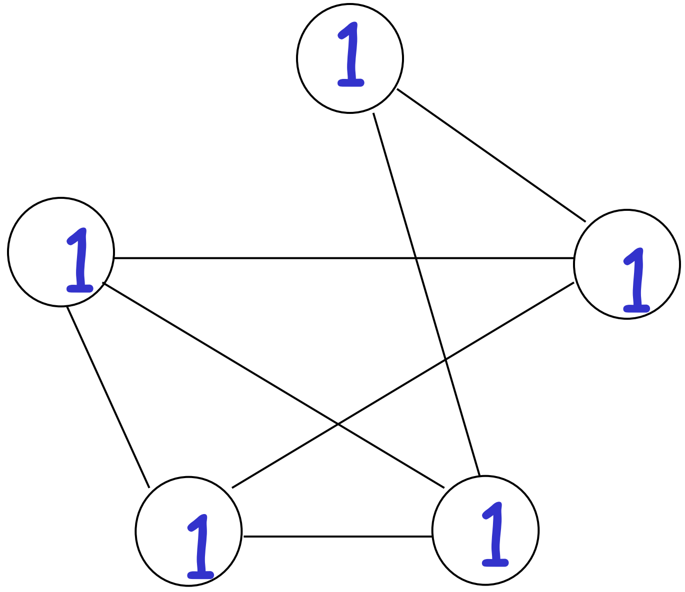
Everybody has an initial value

All non-faulty must decide the same value

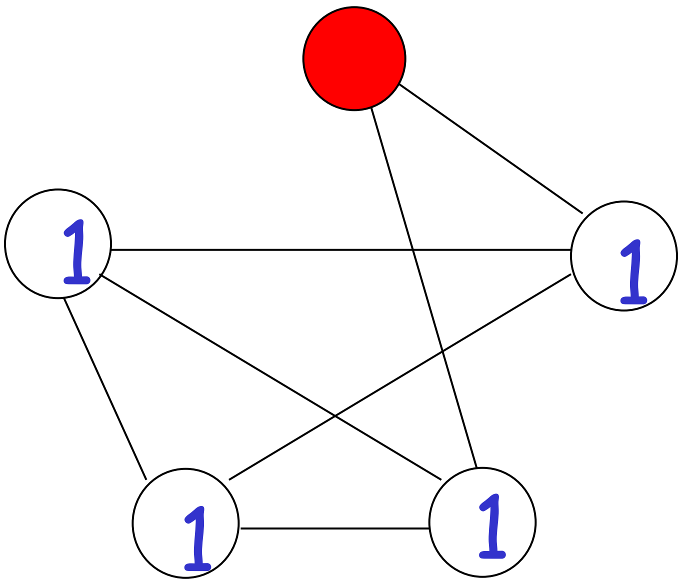
Validity

If everybody starts with the same value, then non-faulty must decide that value

Start



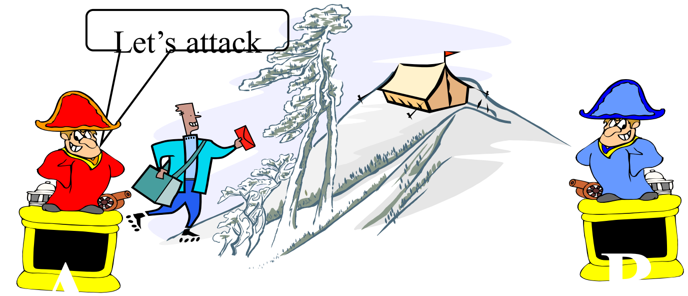
Finish



Negative result for link failures

- Although this is the simplest fault a MPS may face, it may already be enough to prevent consensus
- More formally, there exist input instances for which it is impossible to reach consensus in case of single non-permanent link failures, even in the synchronous non-anonymous case
- To illustrate this negative result, we present the very famous problem of the 2 generals

Consensus under non-permanent link failures: the 2 generals problem



There are two generals of the same army who have encamped a short distance apart. Their objective is to decide on whether to capture a hill, which is possible only if they both attack (i.e., if only one general attacks, he will be defeated, and so their common output should be either "not attack" or "attack"). However, they might have different opinion about what to do (i.e., their input). The two generals can only communicate (synchronously) by sending messengers, which could be captured (i.e., link failure), though. Is it possible for them to reach a common decision?

More formally, we are talking about consensus in the following MPS:

Impossibility of consensus under link failures

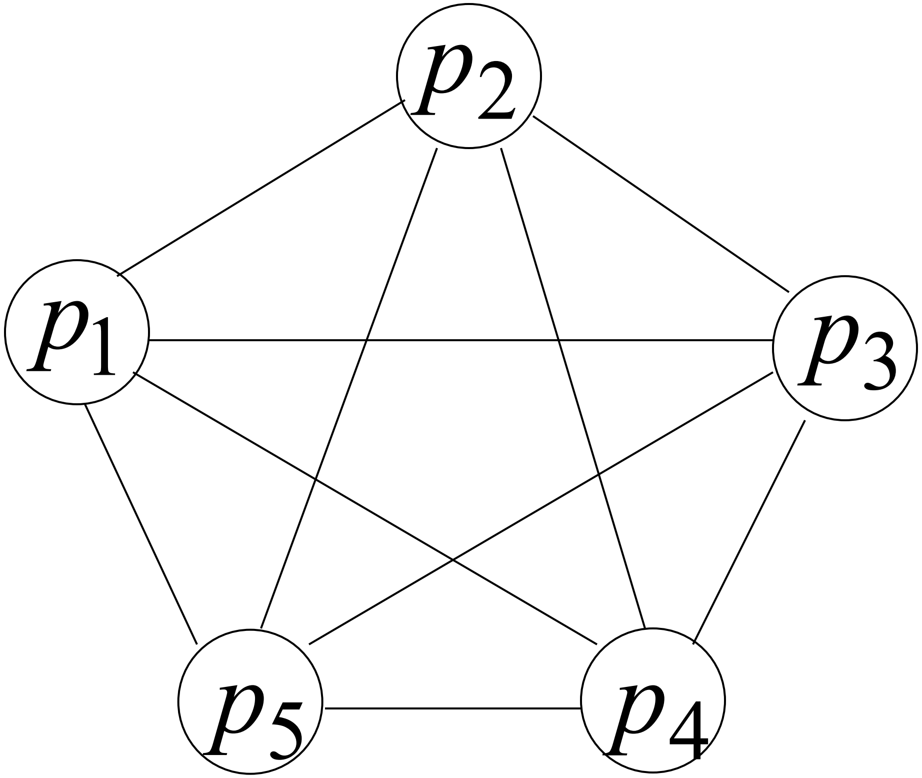
- First of all, notice that it is needed to exchange messages to reach consensus (as we said, generals might have different opinions in mind!)
- Assume the problem can be solved, and let Π be the shortest protocol (i.e., a solving algorithm with the minimum number of messages) for a given input configuration.
- Since this protocol is **deterministic**, for such a fixed input configuration, there will be a **sequence** of messages to be exchanged, which however may not be all successfully delivered, due to the possible link failure.
- In particular, suppose now that the last message in Π does not reach the destination (i.e., a link failure takes place). Since Π is correct independent of link failures, consensus must be reached in any case. This means, the last message was useless, and then Π could not be shortest!

Negative result for processor failures in asynchronous systems

- It is not hard to see that a processor failure (both permanent crash and byzantine) is at least as difficult as a non-permanent link failure, and then also in this case not for all the input instances it will be possible to solve the consensus problem
- **Negative result**: in the asynchronous case it can be proven that it is impossible to reach consensus for any system topology and already for a single crash failure!

$\Rightarrow$ in search of some positive result, we focus on the synchronous case and we look at the powerful clique topology

Positive results: Assumption on the communication model for crash and byzantine failures



- Complete undirected graph (in a sense, this implies non-uniformity)
- Synchronous network, synchronous start: w.l.o.g., we assume that rounds are now organized as follows: messages are sent at the beginning of a round, and then delivered and read in the very same round

Overview of Consensus Results

f-resilient consensus algorithms (i.e., algorithms solving consensus for **at most** f faulty processors)

	Crash failures	Byzantine failures
Number of rounds	$f+1$ (tight)	$2(f+1)$ $f+1$ (tight)
Total number of processors	$n \geq f+1$ (tight)	$n \geq 4f+1$ $n \geq 3f+1$ (tight)
Message complexity	$O(n^3)$	$O(n^3)$ $O(n^{O(n)})$ (exponential)

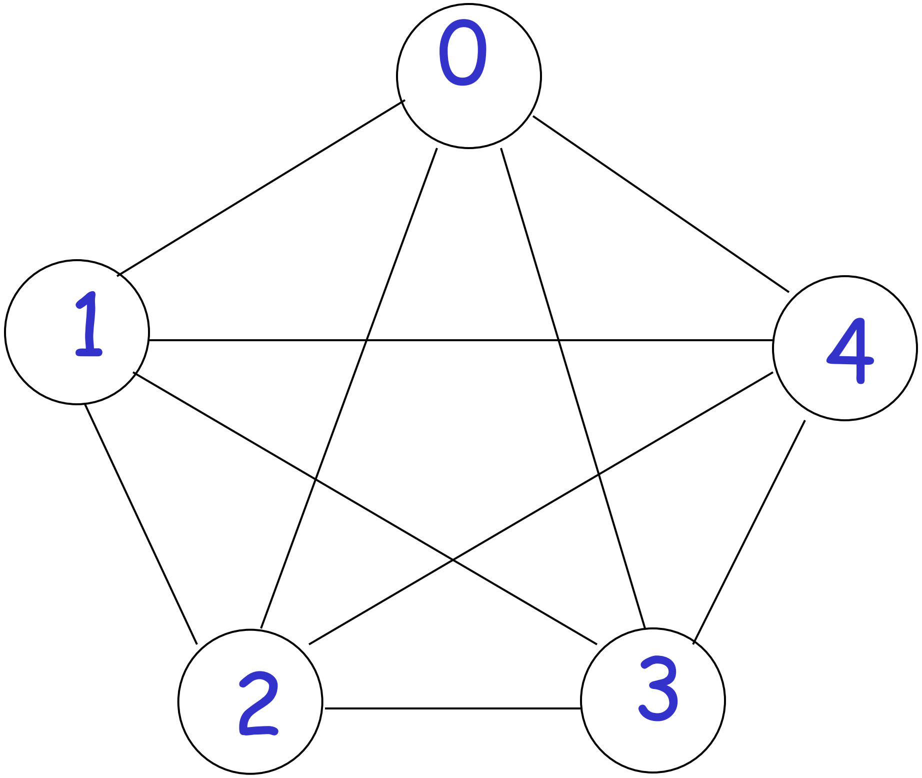
A simple algorithm for fault-free consensus

Each processor:

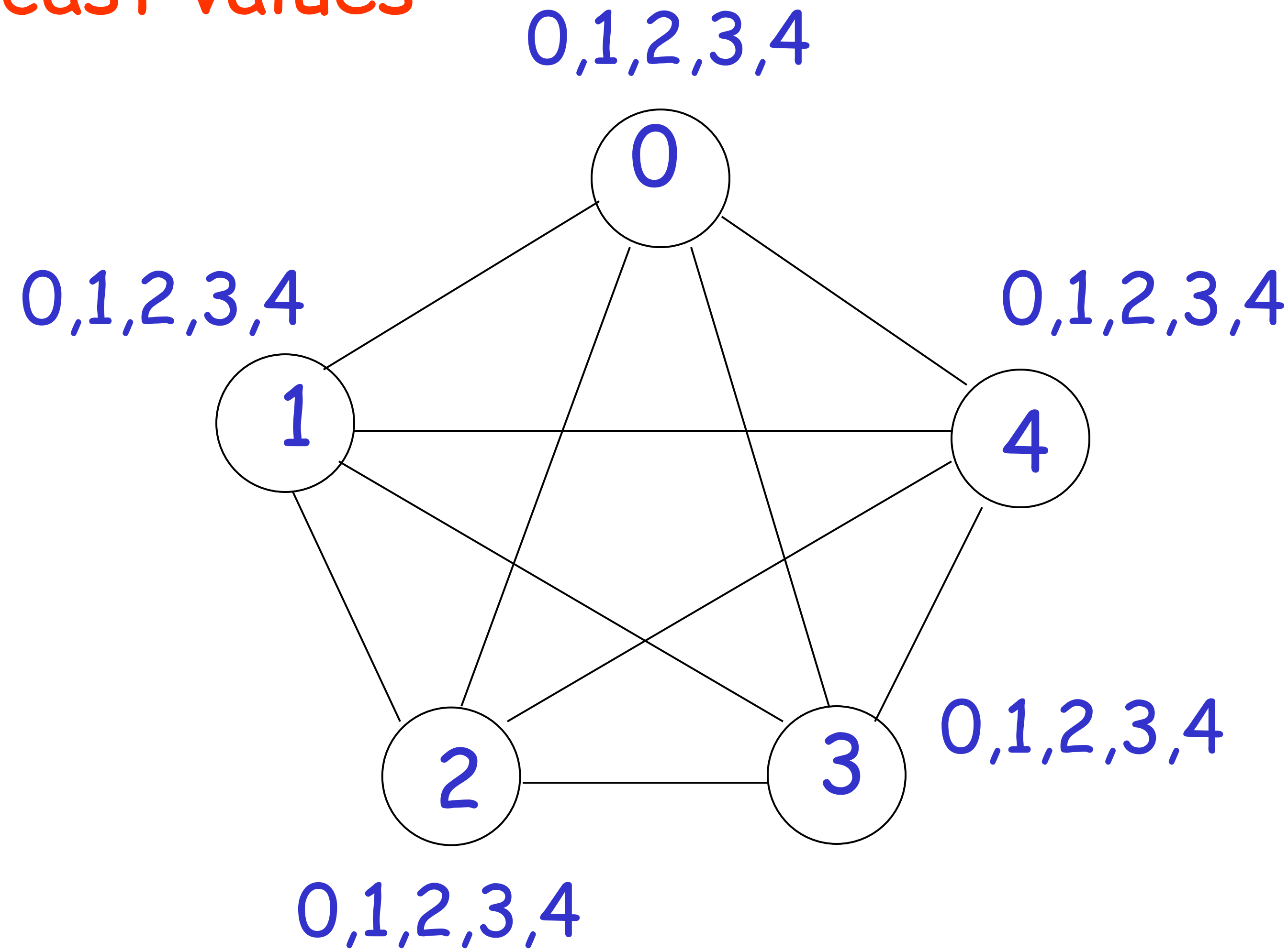
1. Broadcasts its input to all processors (including itself)
2. Reads all the incoming messages
3. Decides on the minimum received value

(only one round is needed, since the graph is complete)

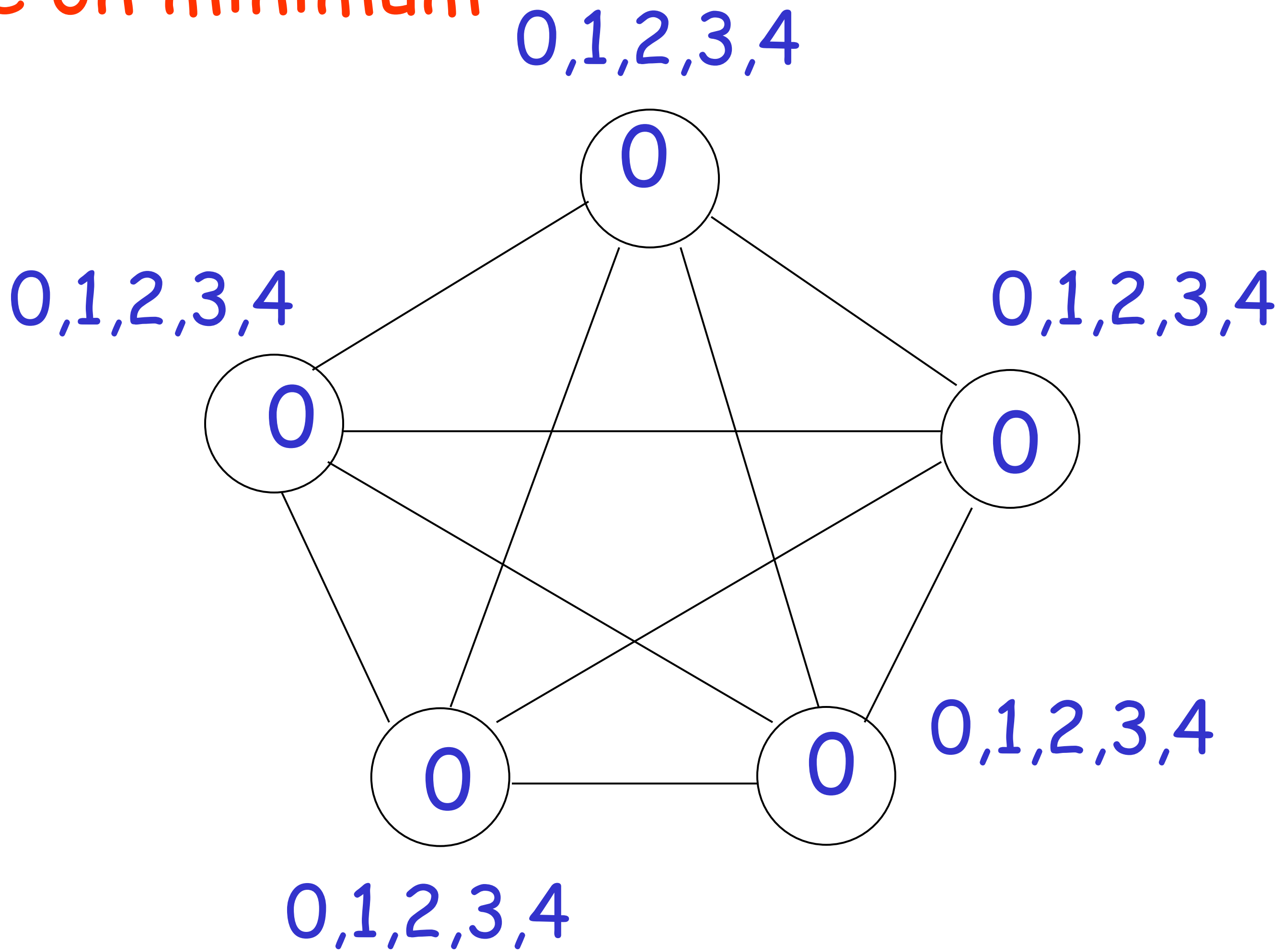
Start



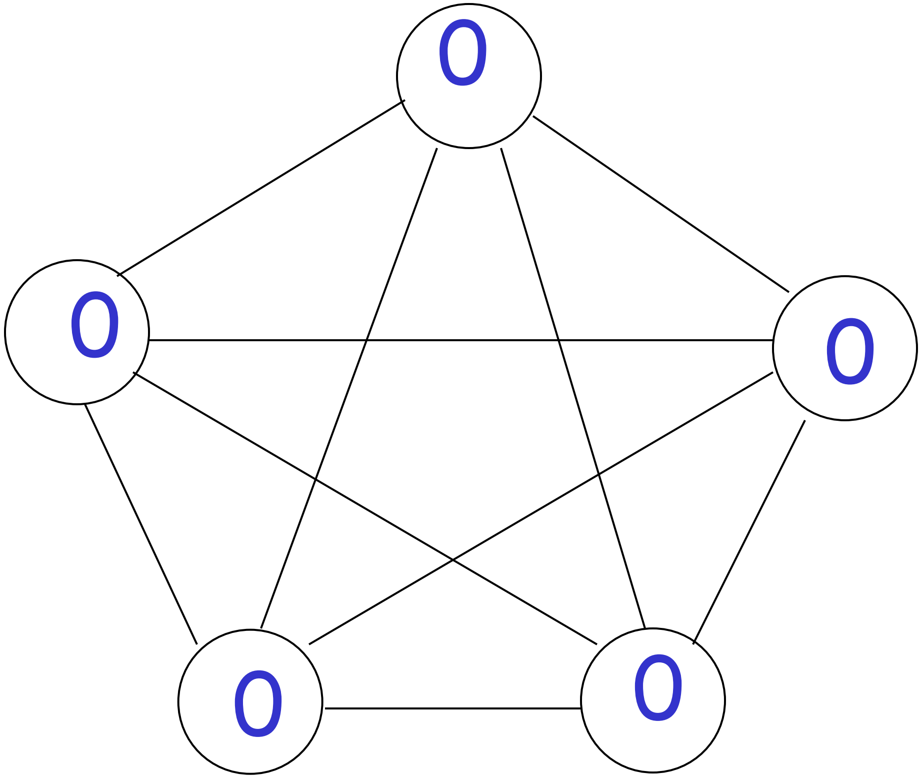
Broadcast values



Decide on minimum

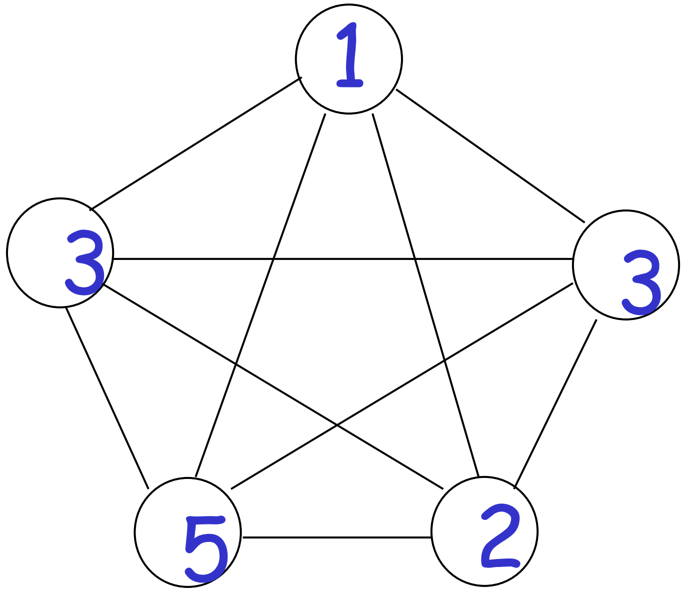


Finish

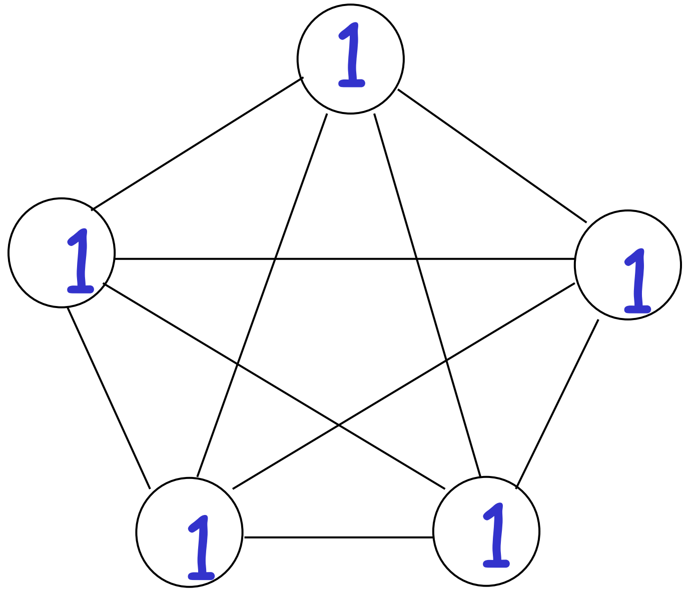


This algorithm satisfies the agreement

Start



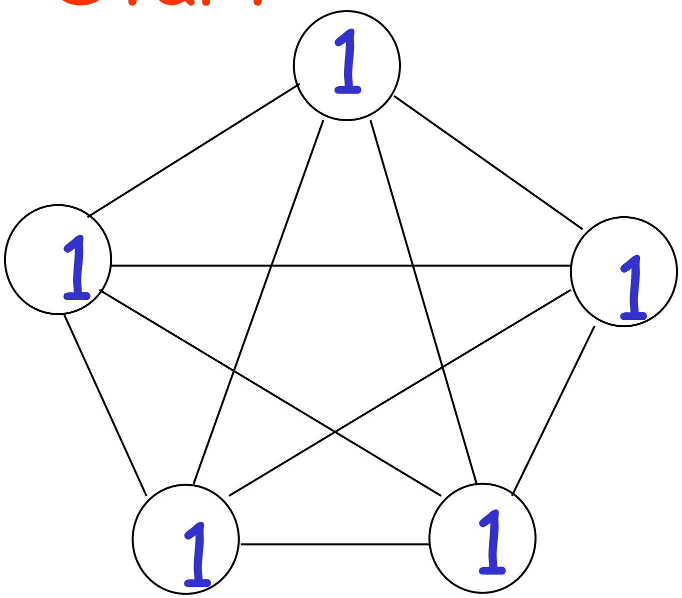
Finish



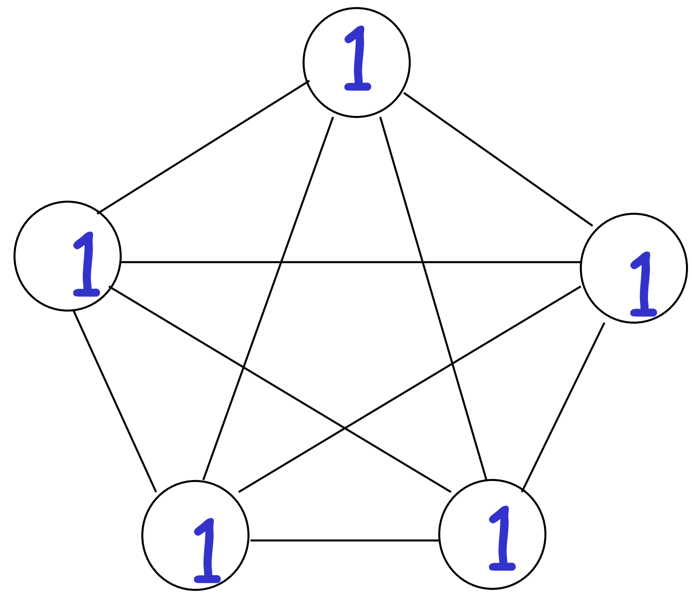
All the processors decide the minimum exactly over the same set of values

This algorithm satisfies the validity condition

Start



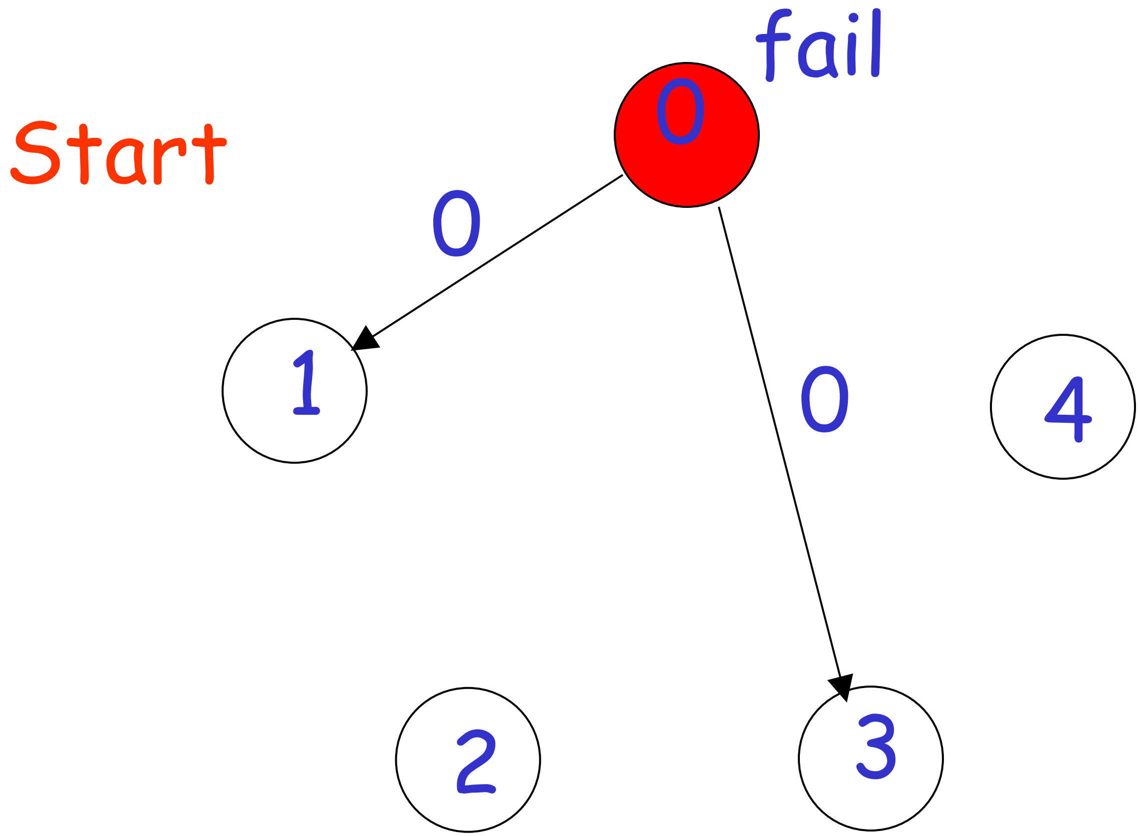
Finish



If everybody starts with the same initial value, everybody decides on that value (minimum)

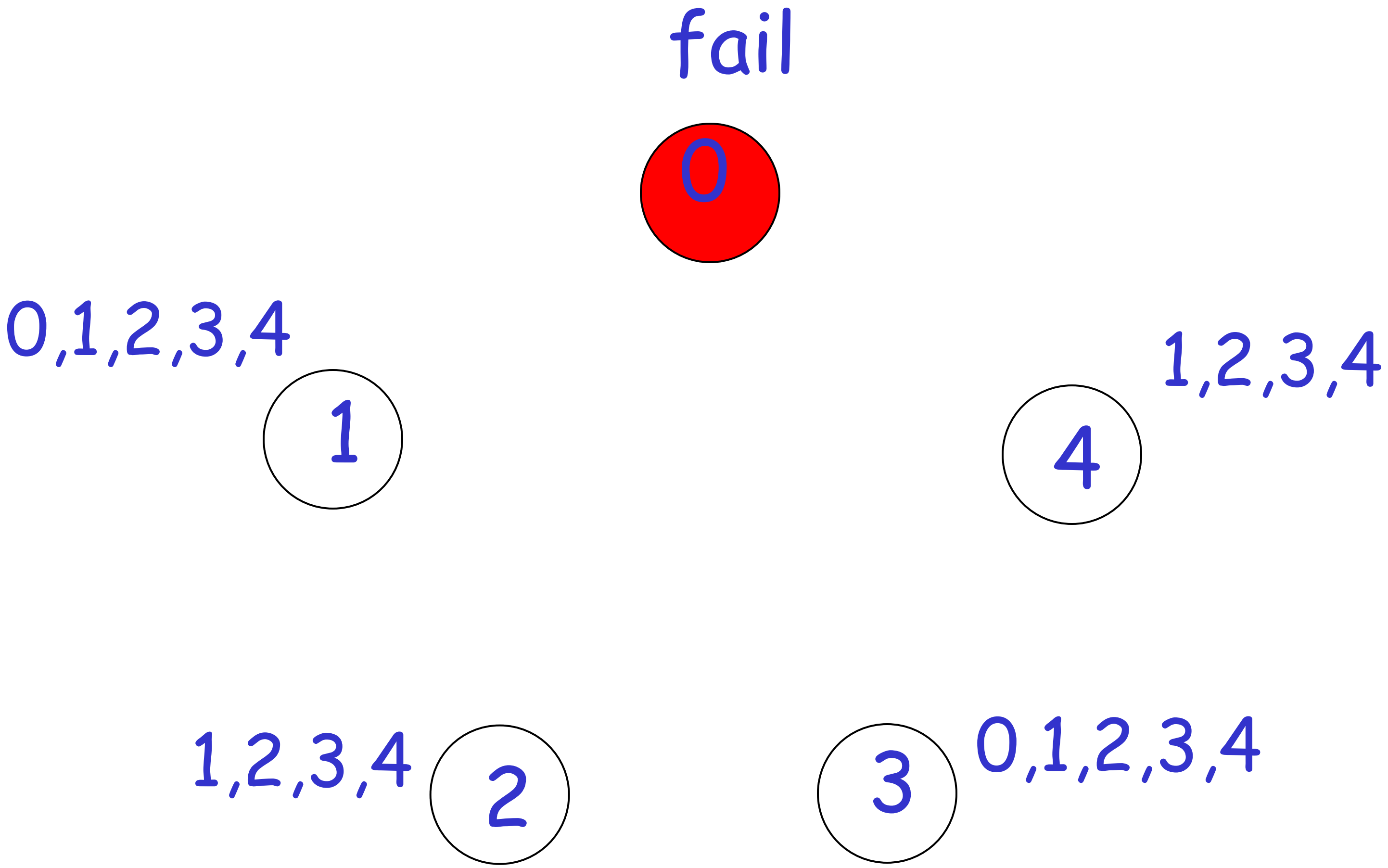
Consensus with Crash Failures

The simple algorithm doesn't work

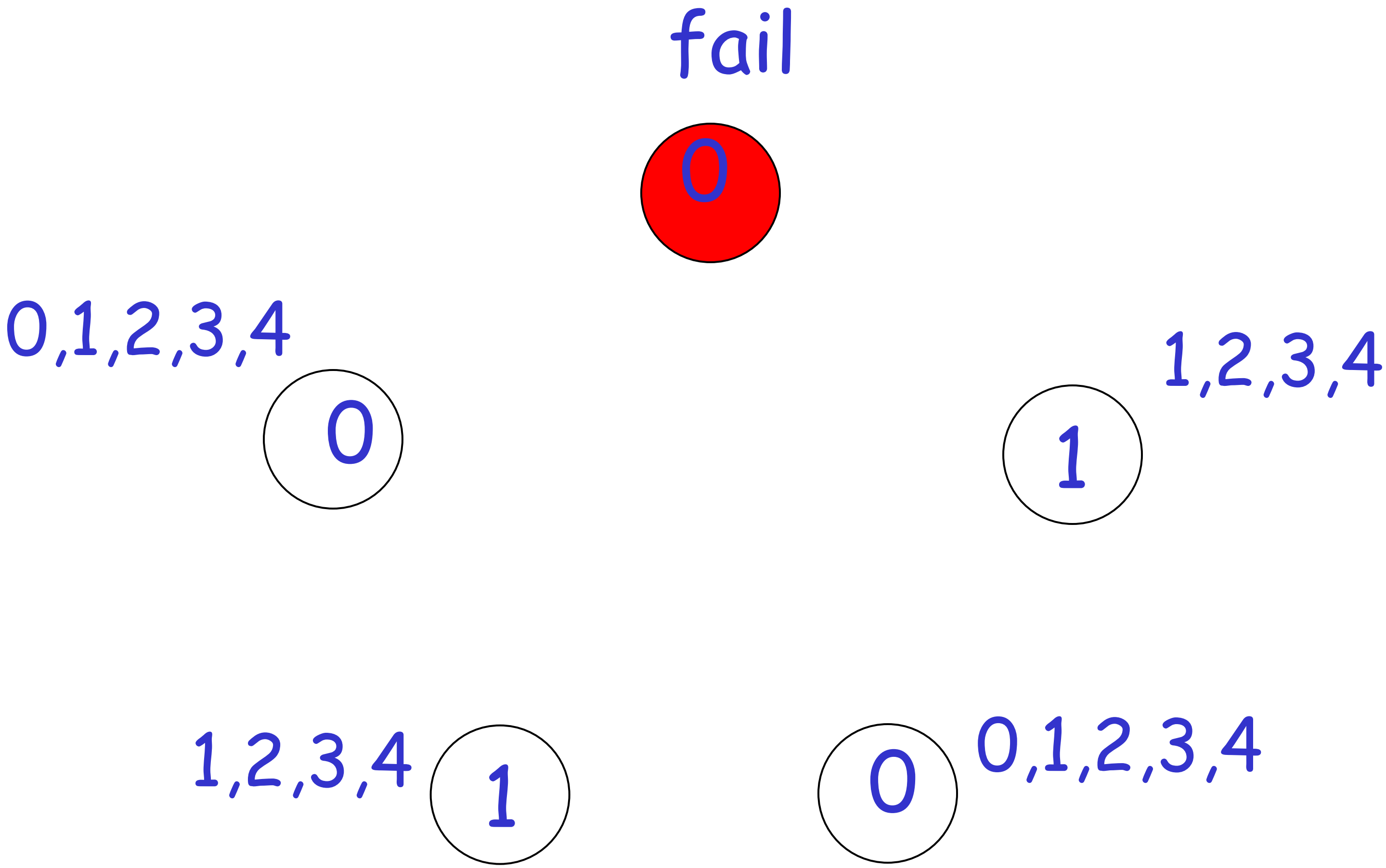


The failed processor doesn't broadcast its value to all processors

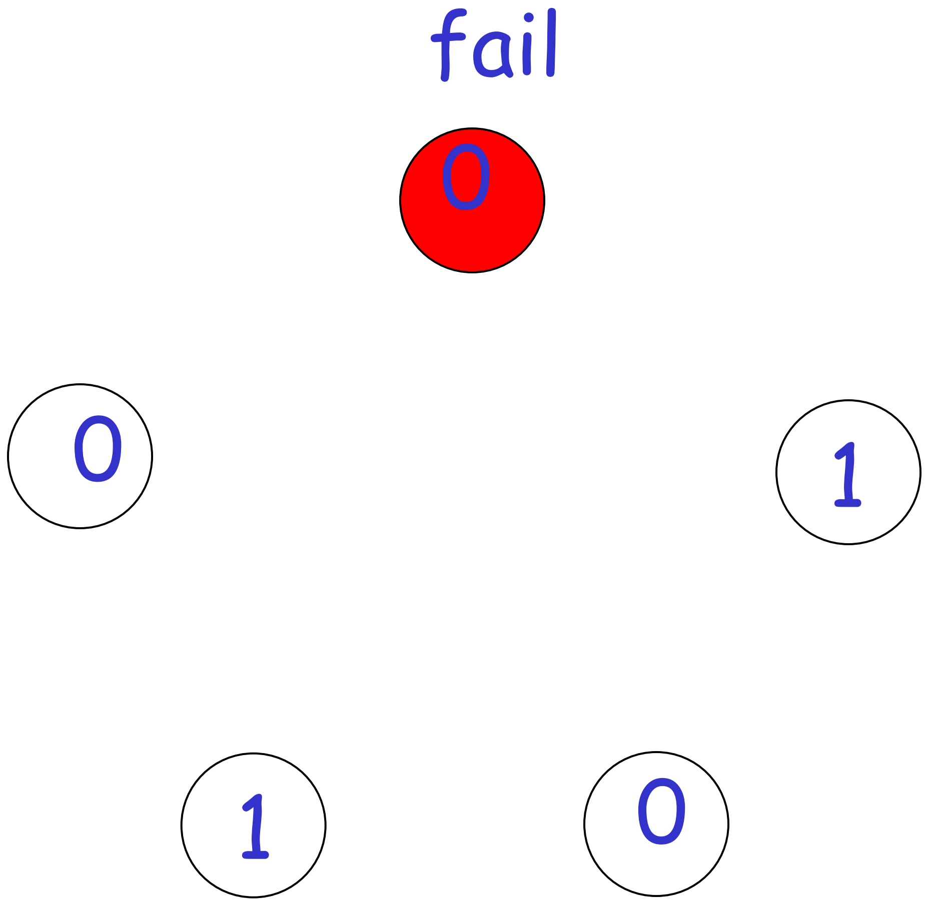
Broadcasted values



Decide on minimum



Finish



No agreement!!!

An f-resilient to crash failures algorithm

Each processor:

Round 1:

Broadcast to all (including myself) my value;

Read all the incoming values;

Round 2 to round f+1:

Broadcast to all (including myself) any new received values (one message for each value):

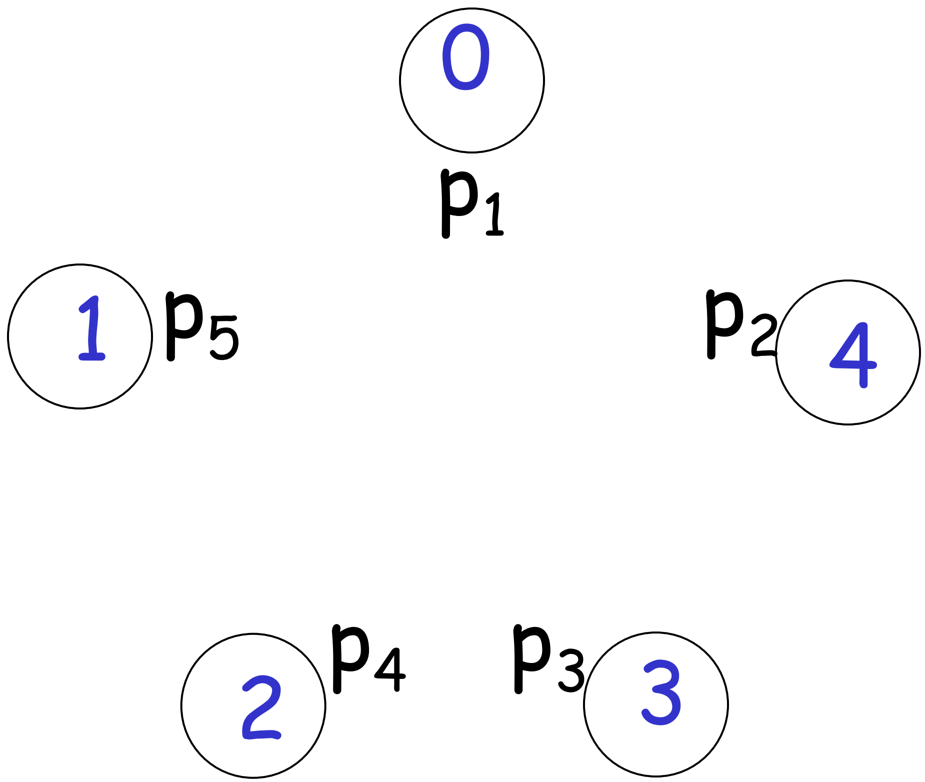
Read all the incoming values;

End of round f+1:

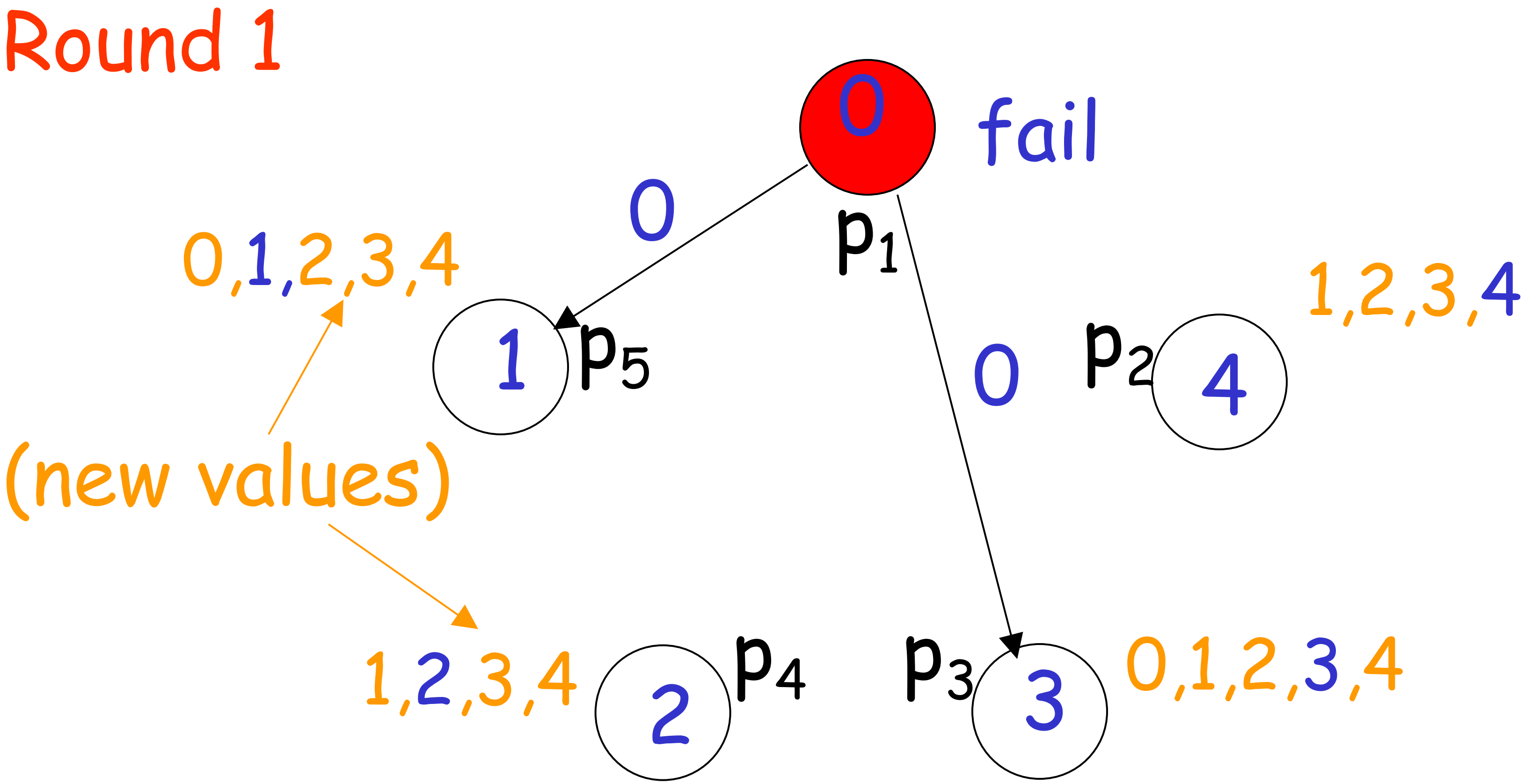
Decide on the minimum value ever received.

Example 1: f=1 failures, f+1 = 2 rounds needed

Start



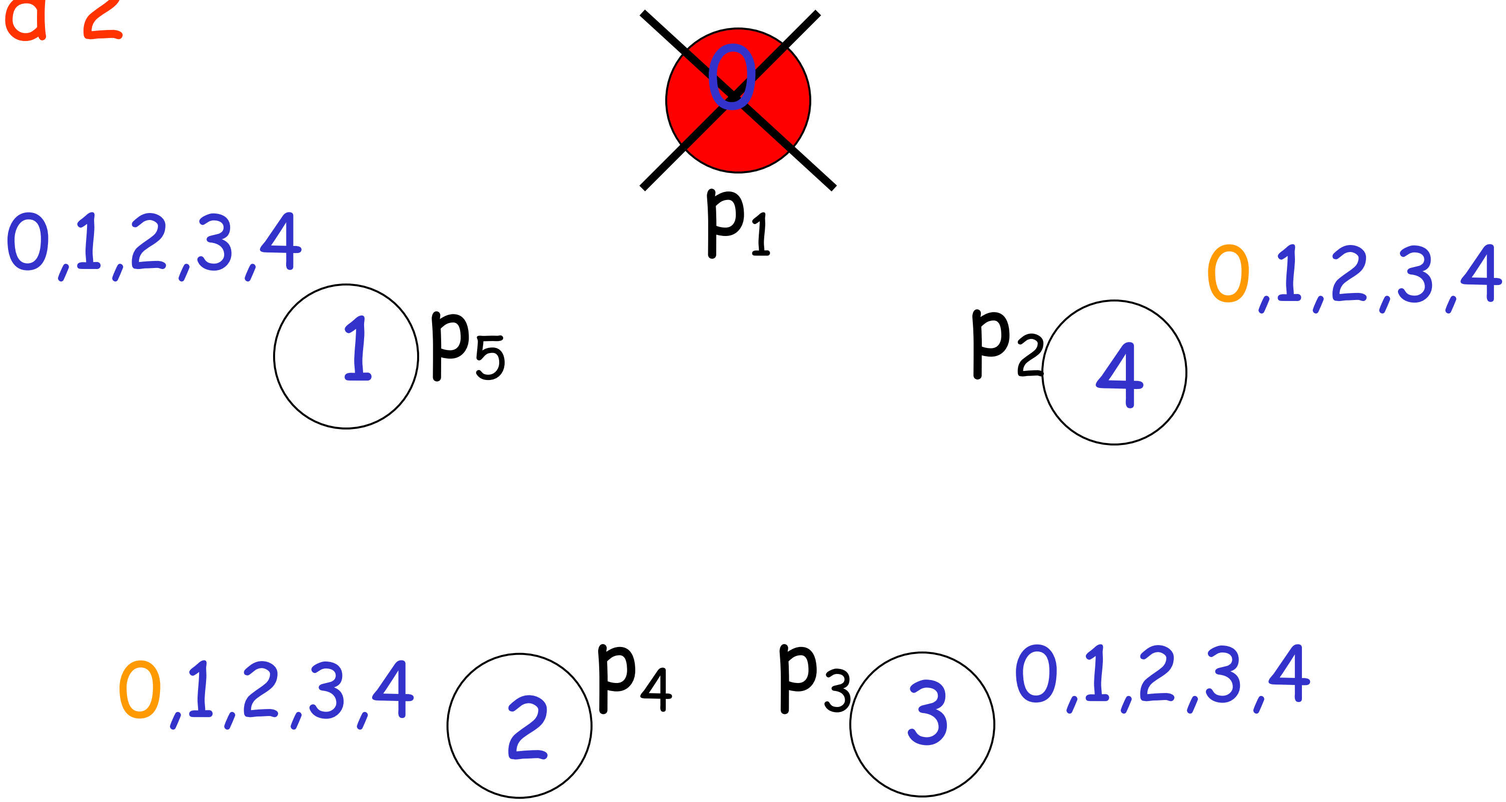
Example 1: f=1 failures, f+1 = 2 rounds needed



Broadcast all values to everybody

Example 1: f=1 failures, f+1 = 2 rounds needed

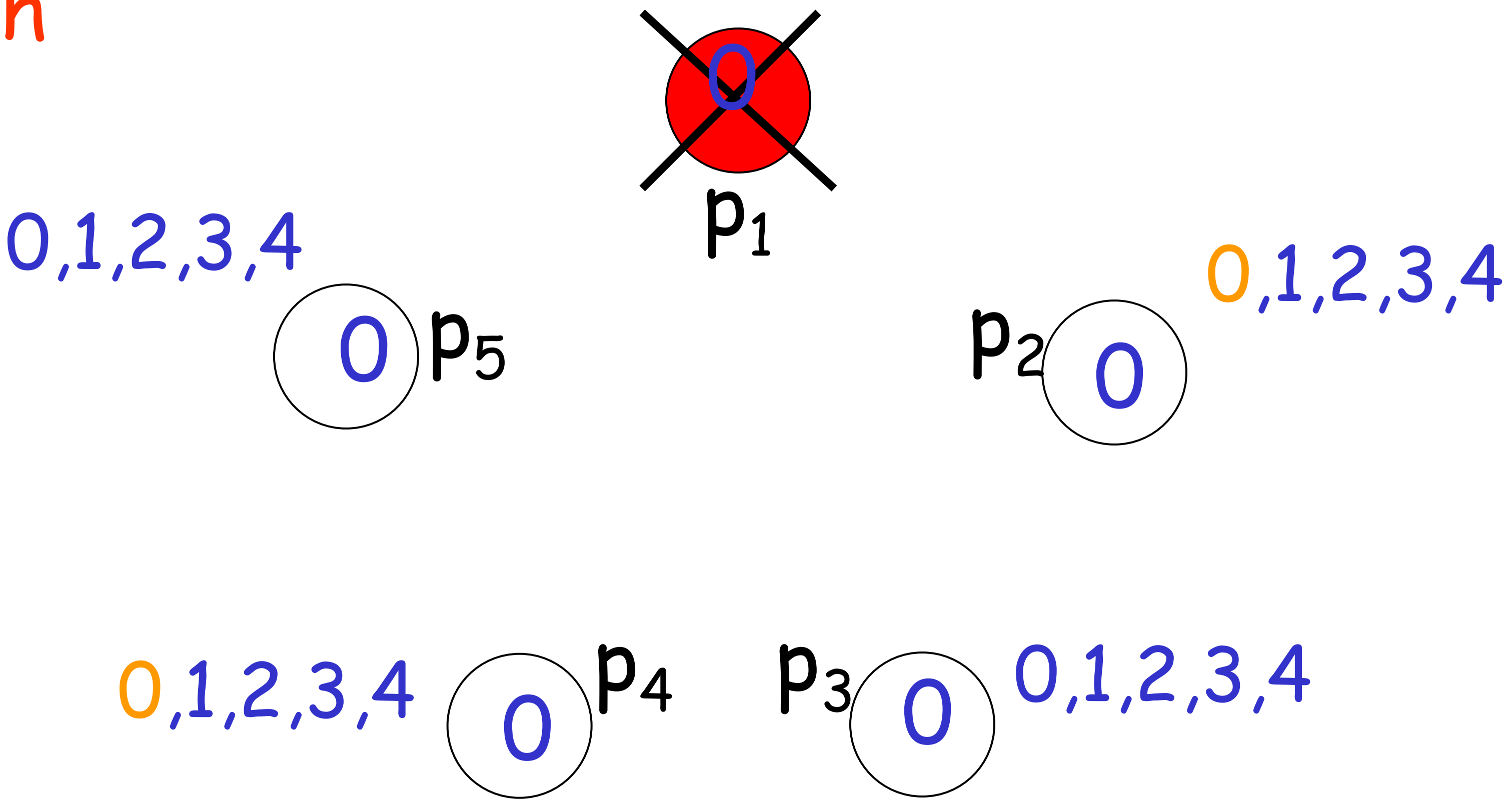
Round 2



Broadcast all new values to everybody

Example 1: f=1 failures, f+1 = 2 rounds needed

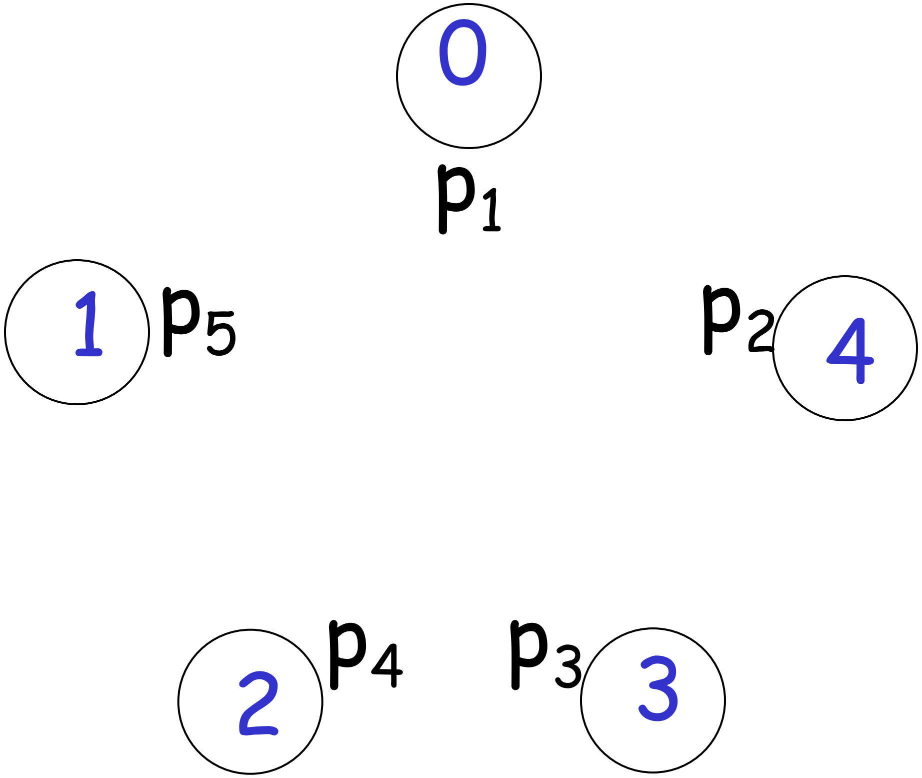
Finish



Decide on minimum value

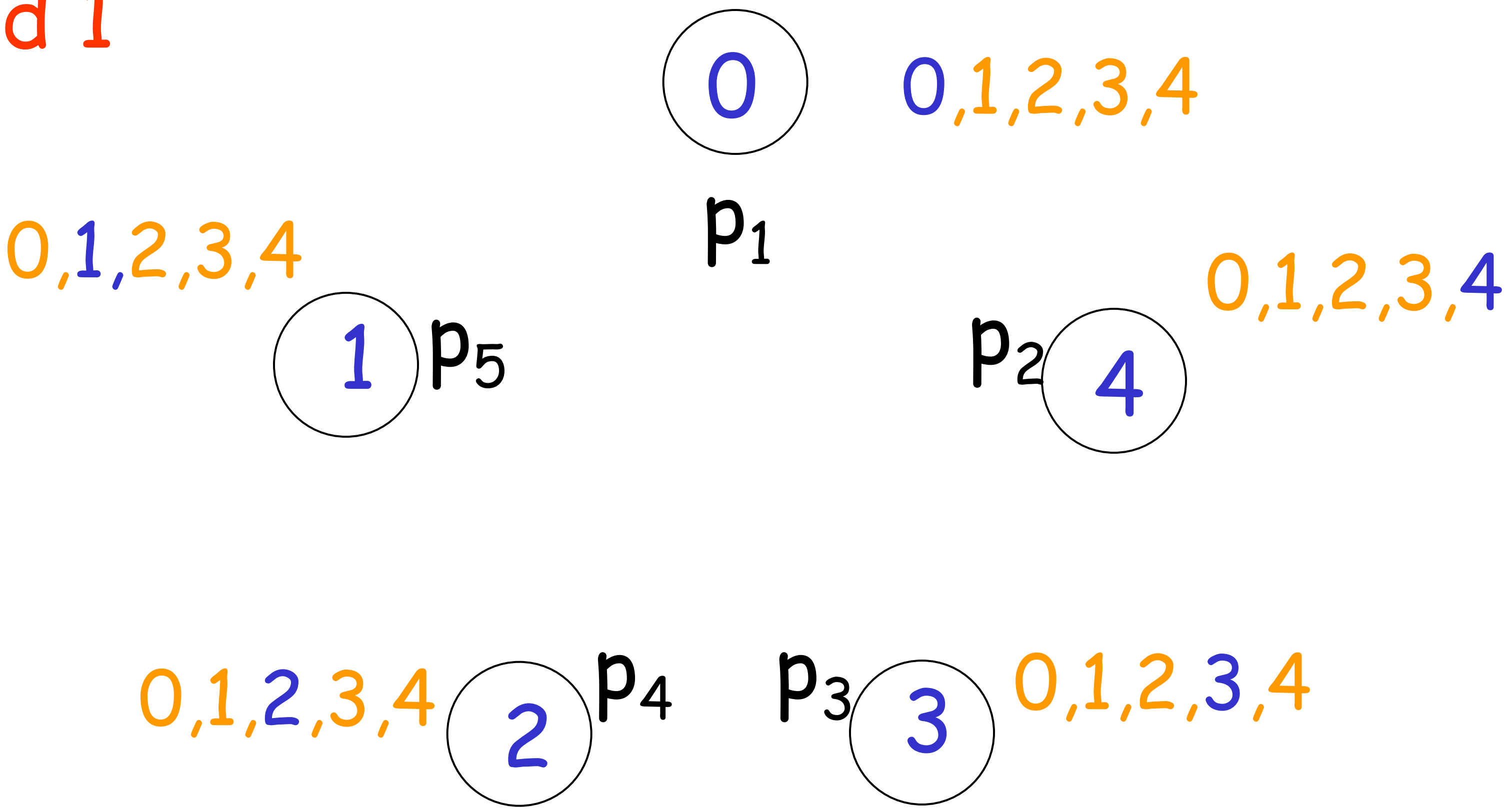
Example 2: f=1 failures, f+1 = 2 rounds needed

Start



Example 2: f=1 failures, f+1 = 2 rounds needed

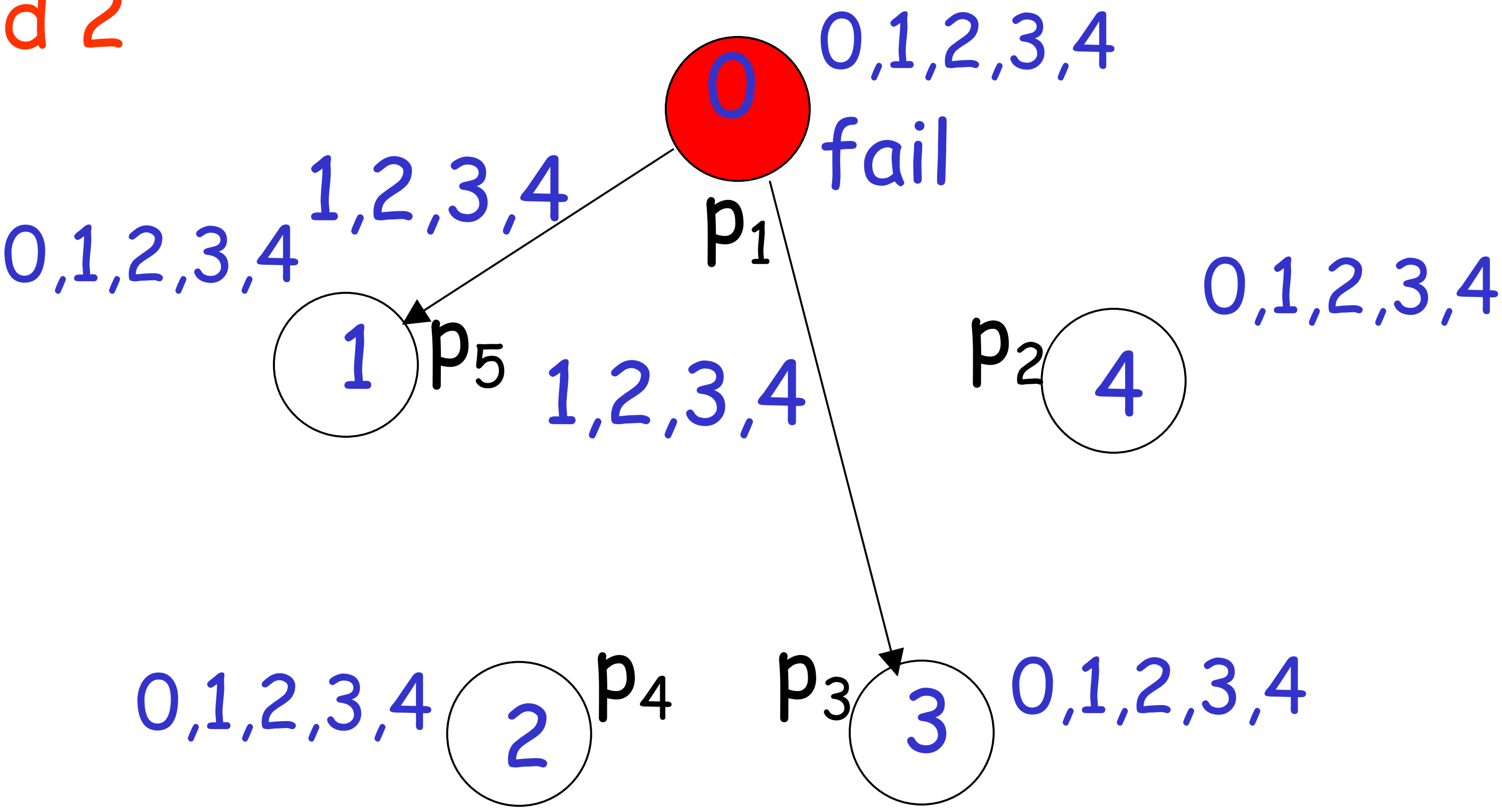
Round 1



No failures: all values are broadcasted to all

Example 2: f=1 failures, f+1 = 2 rounds needed

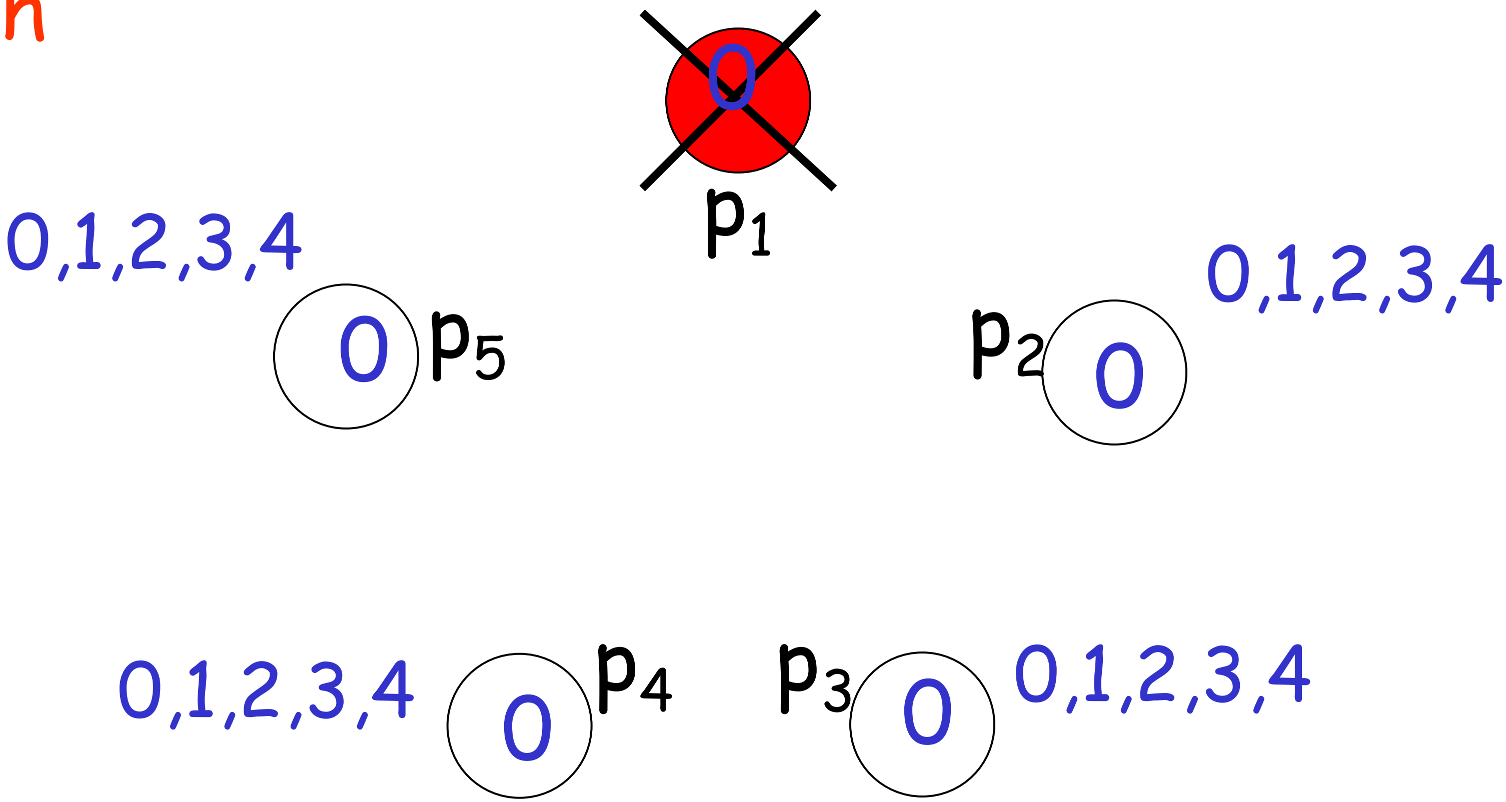
Round 2



No problem: processors p_2 and p_4 have already seen 1,2,3 and 4 in the previous round

Example 2: f=1 failures, f+1 = 2 rounds needed

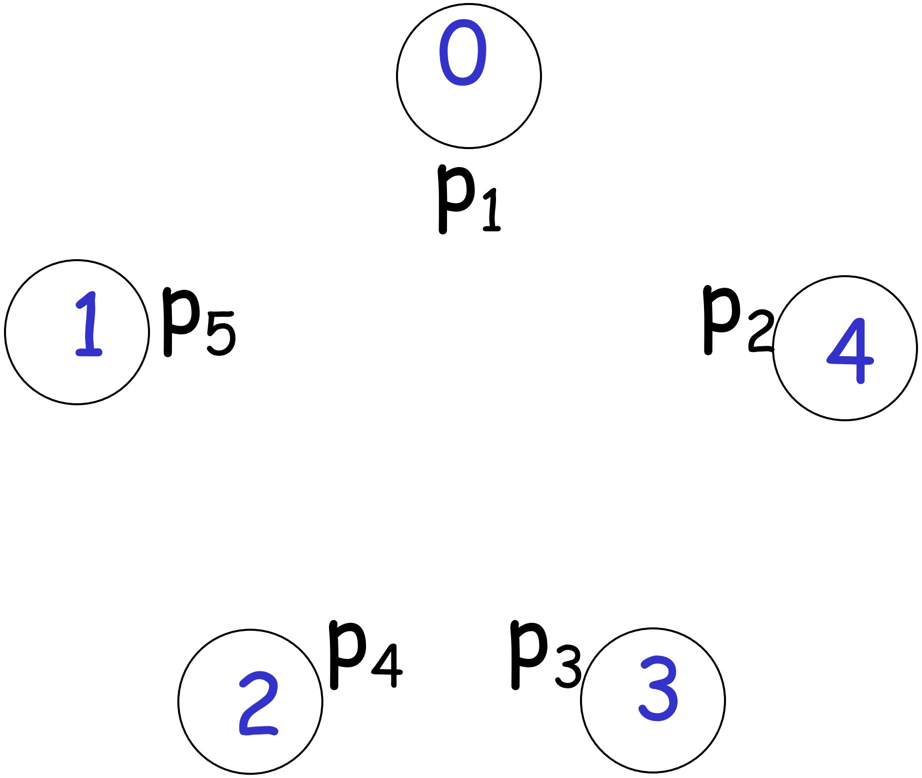
Finish



Decide on minimum value

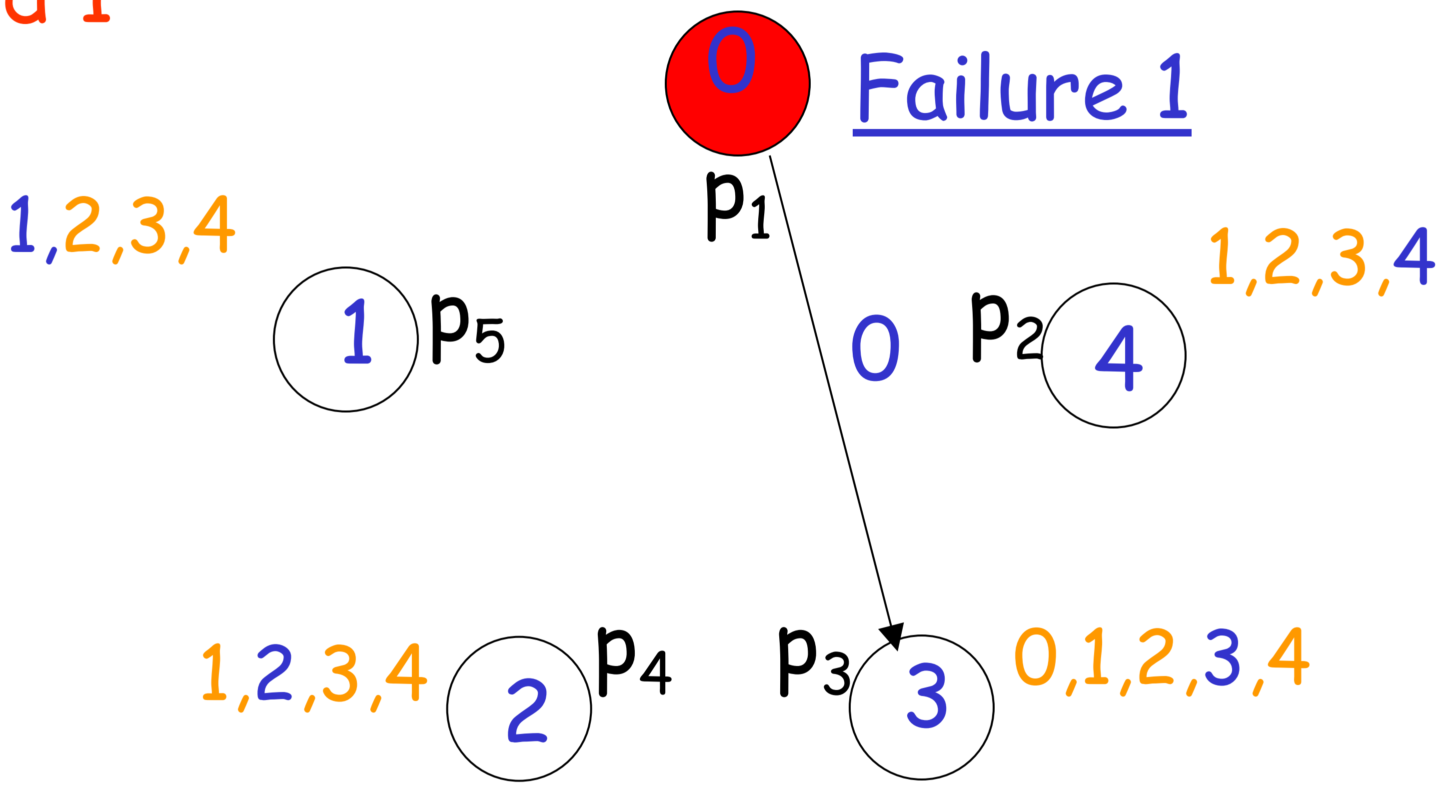
Example 3: f=2 failures, f+1 = 3 rounds needed

Start



Example 3: f=2 failures, f+1 = 3 rounds needed

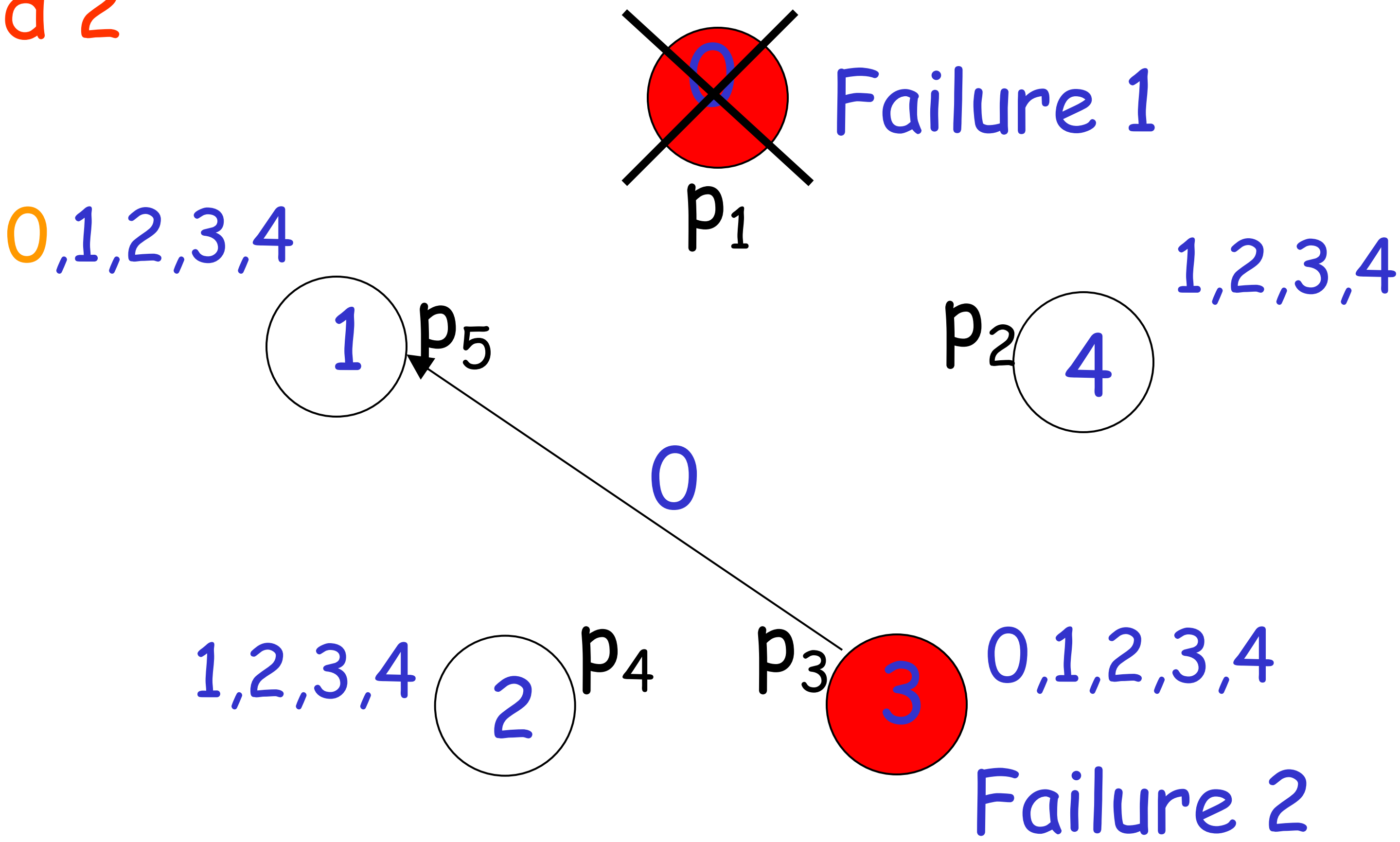
Round 1



Broadcast all values to everybody

Example 3: f=2 failures, f+1 = 3 rounds needed

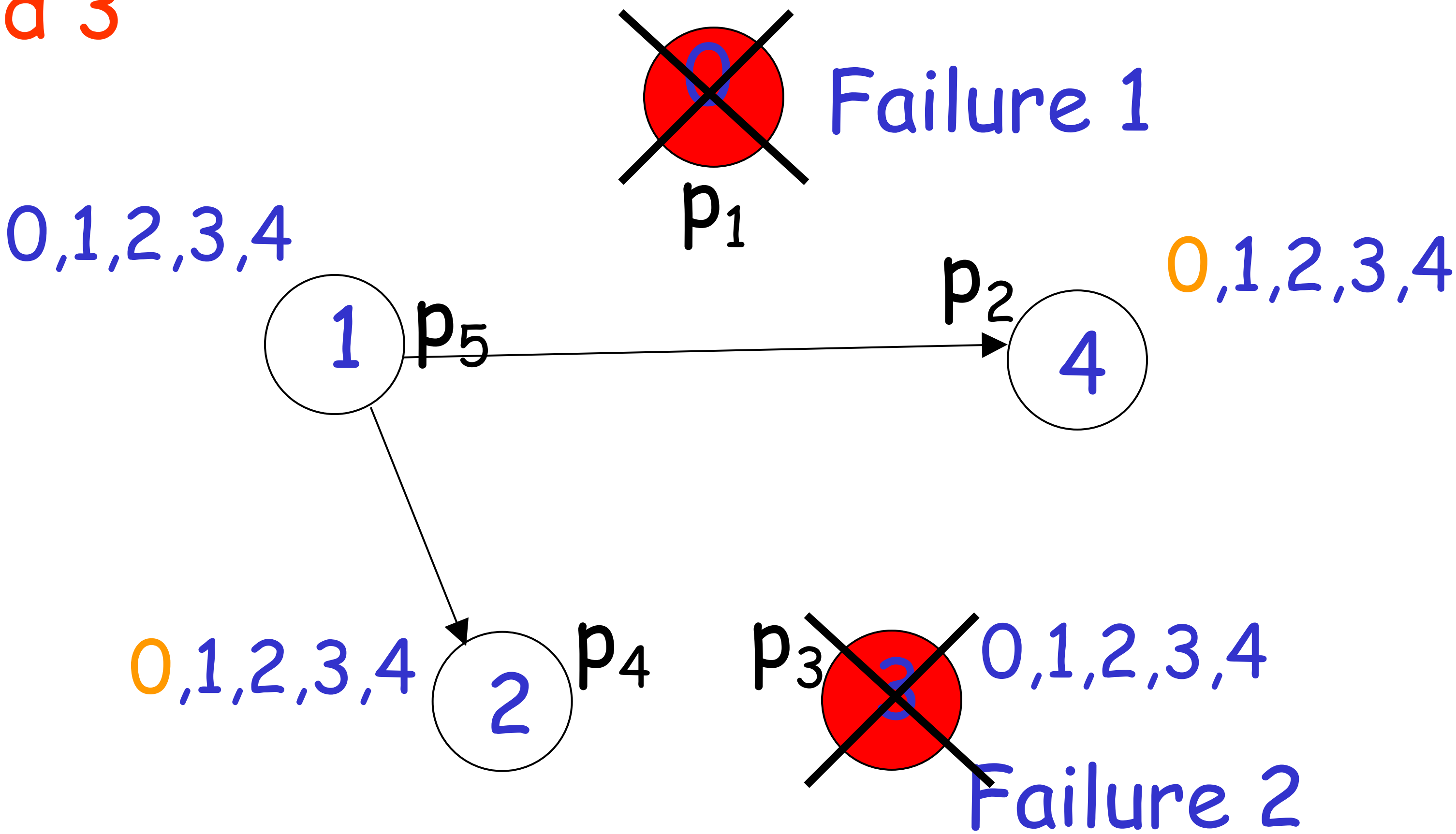
Round 2



Broadcast new values to everybody

Example 3: f=2 failures, f+1 = 3 rounds needed

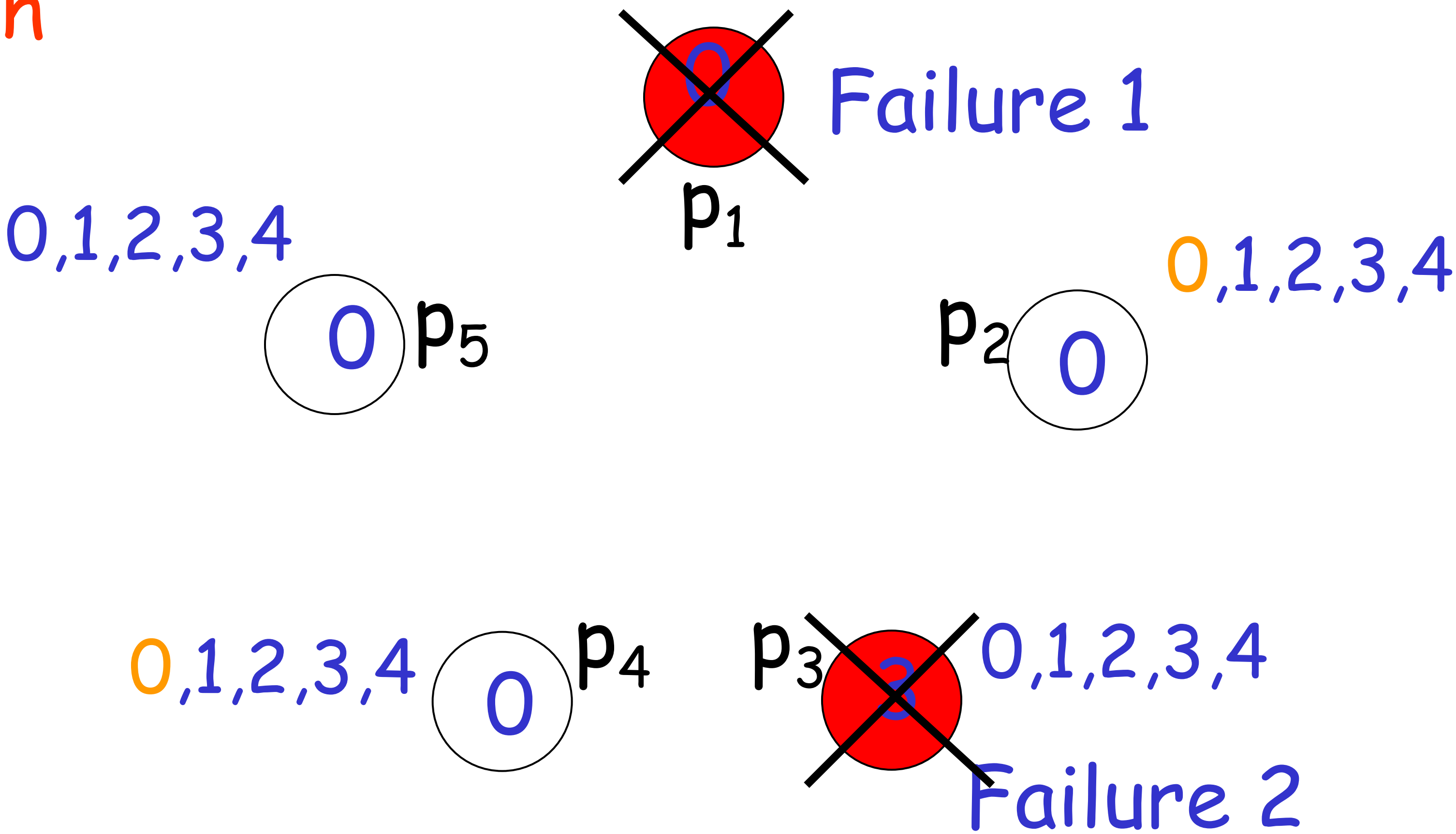
Round 3



Broadcast new values to everybody

Example 3: f=2 failures, f+1 = 3 rounds needed

Finish



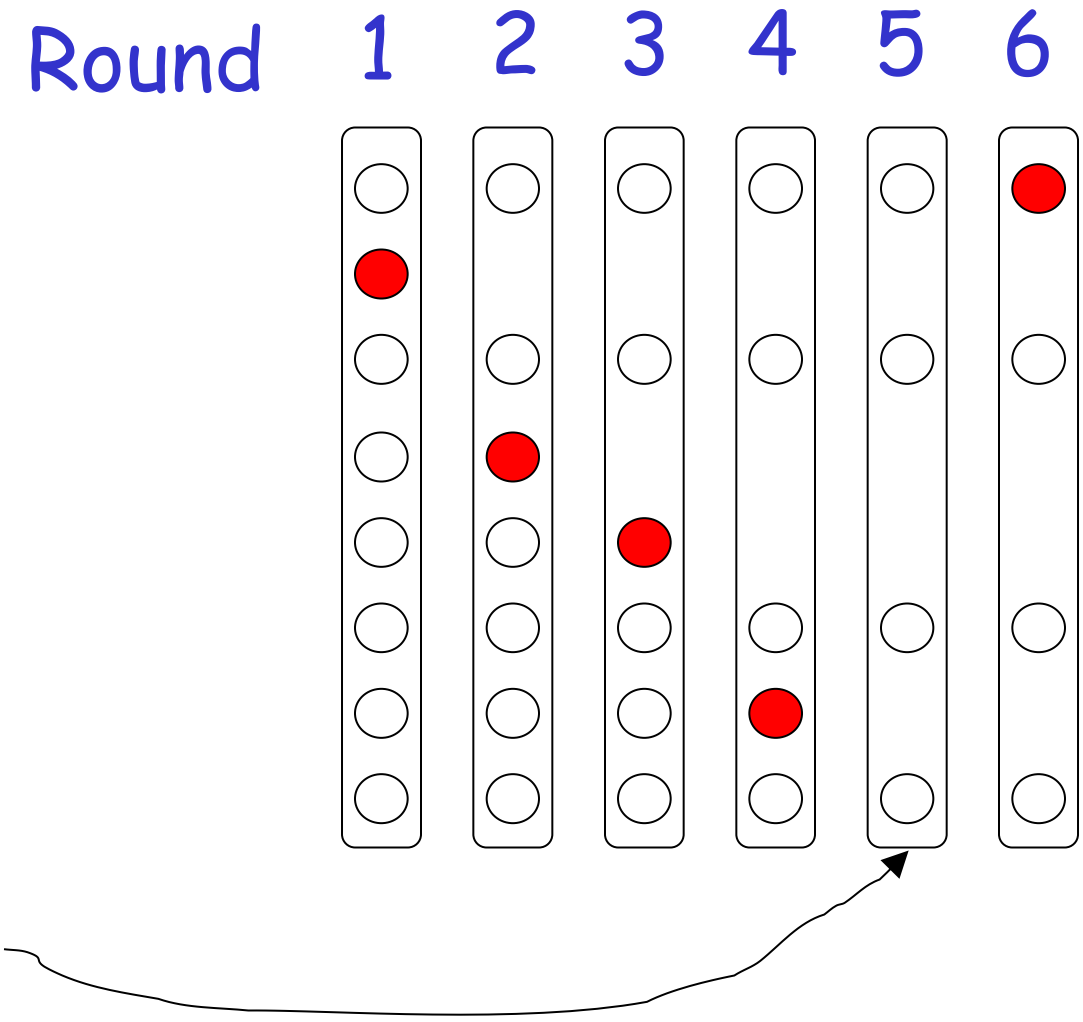
Decide on the minimum value

In general, since there are f failures and f+1 rounds, then there is at least a round with no new failed processors:

Example:

5 failures,

6 rounds



No failure

Correctness (1/2)

Lemma: In the algorithm, at the end of the round with no new failures, all the non-faulty processors know the same set of values.

Proof: For the sake of contradiction, assume the claim is false. Let x be a value which is known only to a subset of non-faulty processors at the end of the round with no failures. Observe that any such processors cannot have known x for the first time in a previous round, since otherwise it had broadcasted x to all. So, the only possibility is that it received it right in this round, otherwise all the others should know x as well. But in this round there are no failures, and so x must be received and known by all, a contradiction.

QED

Correctness (2/2)

Agreement: this holds, since at the end of the round with no failures, every (non-faulty) processor has the same knowledge, and this doesn't change until the end of the algorithm (no new values can be introduced, since we assumed synchronous start) $\Rightarrow$ eventually, everybody will decide the same value!

Remark: we don't know the exact position of the free-of-failures round, so we have to let the algorithm execute for f+1 rounds

Validity: this holds, since the value decided from each processor is some input value (no exogenous values are introduced)

Performance of Crash Consensus Algorithm

- Number of processors: $n > f$
- $f+1$ rounds
- $O(n^2 \cdot k) = O(n^3)$ messages, where $k = O(n)$ is the number of different inputs. Indeed, each processor sends $O(n)$ messages (one for each processor) containing a given seen input value

A Lower Bound

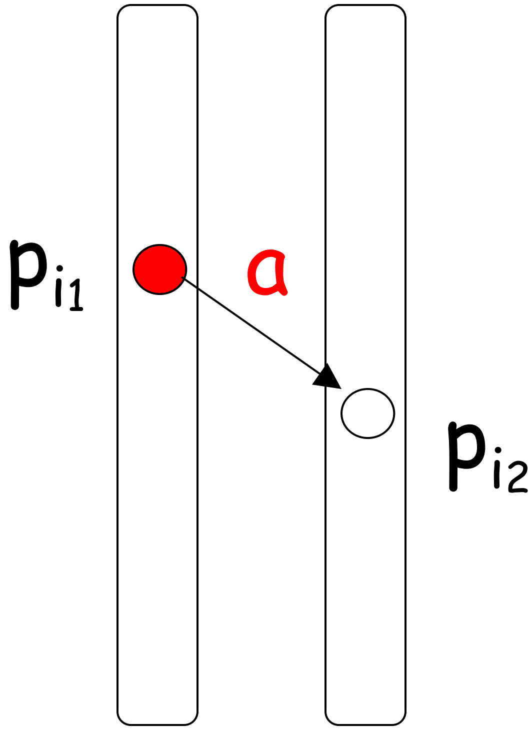
Theorem: Any f-resilient to crash failures consensus algorithm requires at least f+1 rounds

Proof sketch: Assume by contradiction that f or less rounds are enough. Clearly, every algorithm which solves consensus requires that eventually non-faulty processors have the very same knowledge

Worst case scenario:
There is a processor that fails in each round

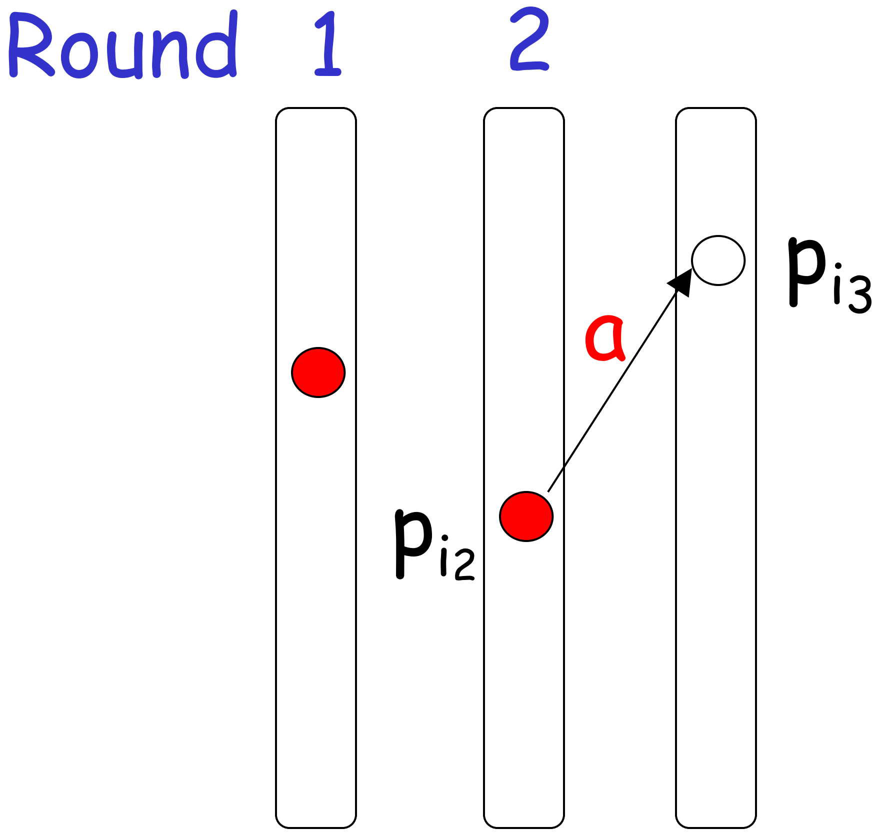
Worst case scenario

Round 1



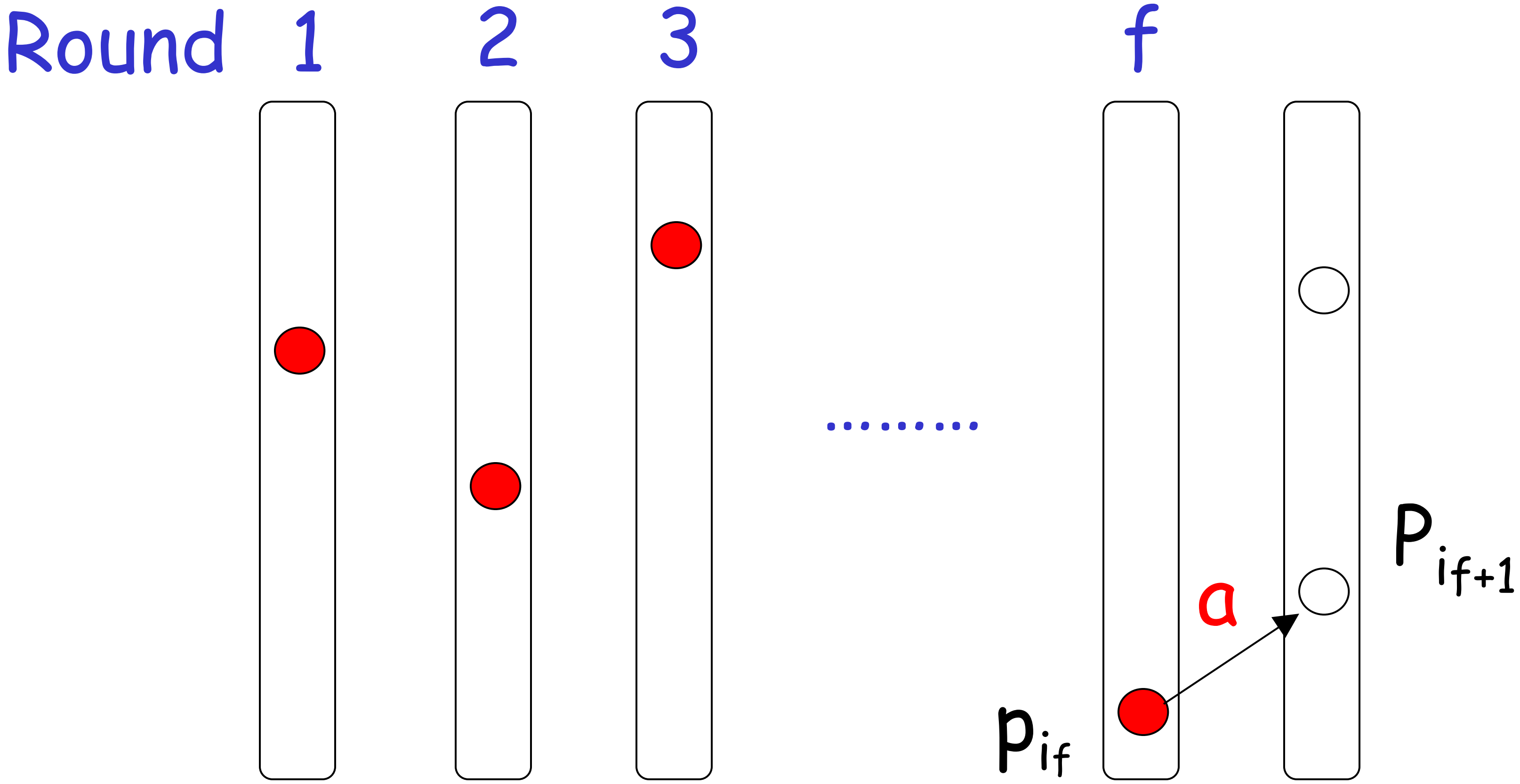
before processor p_{i_1} fails, it sends its value a to only one processor p_{i_2}, and so at the end of round 1 only p_{i_2} knows a

Worst case scenario



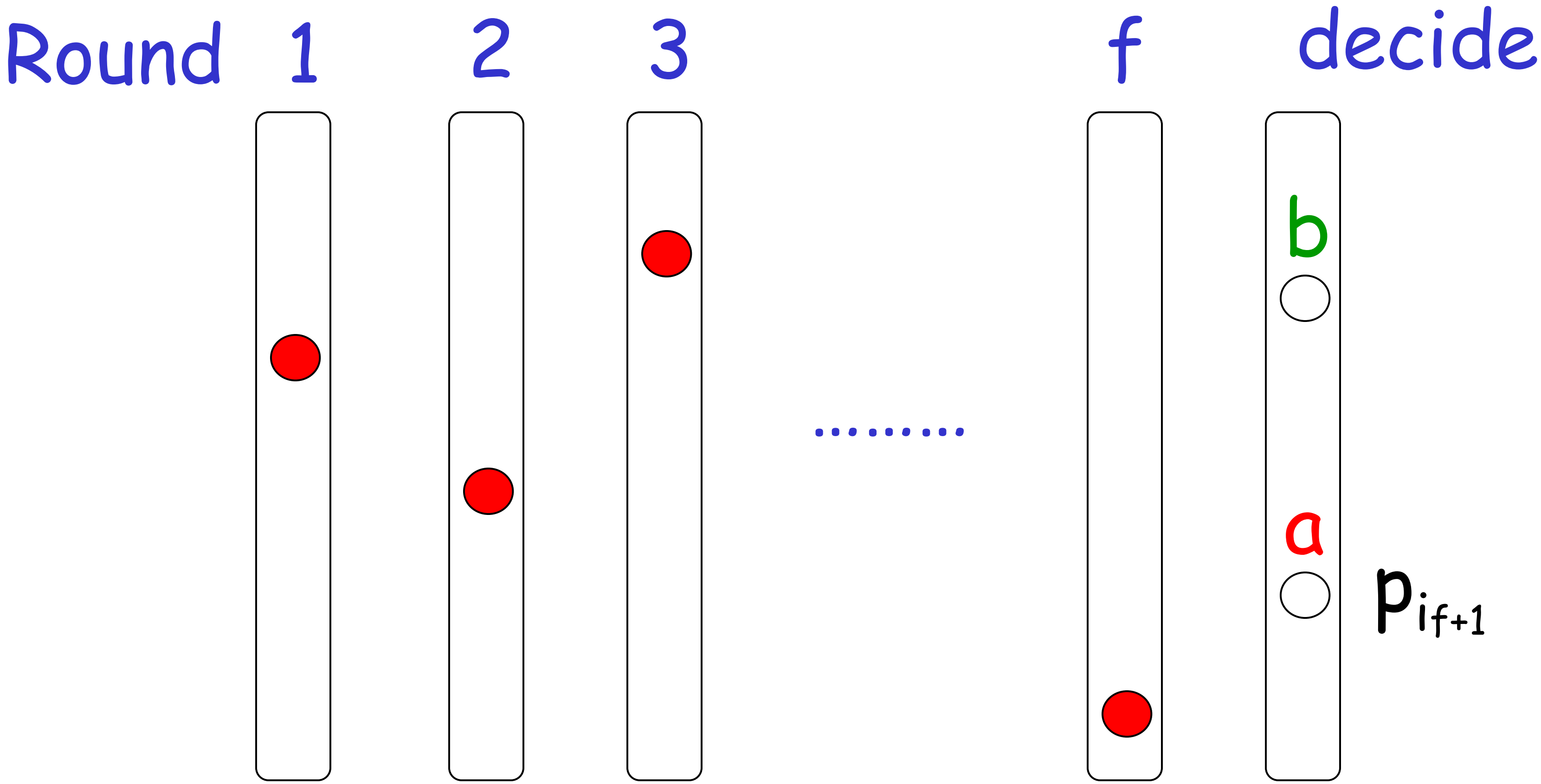
before processor p_{i_2} fails, it sends its value a to only one processor p_{i_3}, and so at the end of round 2 only p_{i_3} knows a

Worst case scenario



Before processor p_{i_f} fails, it sends its value a to only one processor $p_{i_{f+1}}$. Thus, at the end of round f only processor $p_{i_{f+1}}$ knows about a

Worst case scenario



No agreement: Processor $p_{i_{f+1}}$ has a different knowledge, i.e., it may decide a, and all other processors may decide another value, say b>a $\Rightarrow$ contradiction, f rounds are not enough. QED

Consensus with Byzantine Failures

f-resilient to byzantine failures consensus algorithm:

solves consensus for **at most** f byzantine processors

Lower bound on number of rounds

Theorem: Any f-resilient to byzantine failures consensus algorithm requires at least f+1 rounds

Proof:

follows from the crash failure lower bound

An f-resilient to byzantine failures algorithm

The **King** algorithm (P. Berman, J.A. Garay, and K.J. Perry, *FOCS 1989*)

Solves consensus in $2(f+1)$ rounds for n processors out of which at most $n/4$ can be byzantine, namely $f<n/4$ (i.e., $n\geq 4f+1$)

Assumption: The system is non-uniform and processors have (distinct) ids in $\{1,\ldots,n\}$ (and so the system is non anonymous), and we denote by p_i the processor with id i; this is common knowledge, i.e., processors cannot cheat about their ids (namely, p_i cannot behave like if it was p_j, $i\neq j$, even if it is byzantine!)

The King algorithm

There are $f+1$ phases; each phase has 2 rounds, used to update in each processor p_i a preferred value v_i. At the beginning, the preferred value is set to the input value

In each phase there is a different king

$\Rightarrow$ There is a king that is non-faulty!

The King algorithm

Phase $k=1,\ldots,f+1$

Round 1, every processor p_i:

- Broadcast to all (including myself) its preferred value v_i
- Let a be the most frequent received value (including v_i, in case of tie pick an arbitrary value), a.k.a. majority value, and let $1 \leq m_i \leq n$ be its number of occurrences (or majority)
- Set v_i:=a

The **King** algorithm

Phase k=1,…,f+1

Round 2, king p_k:

Broadcast (to the others) its current preferred value v_k

Round 2, processor p_i:

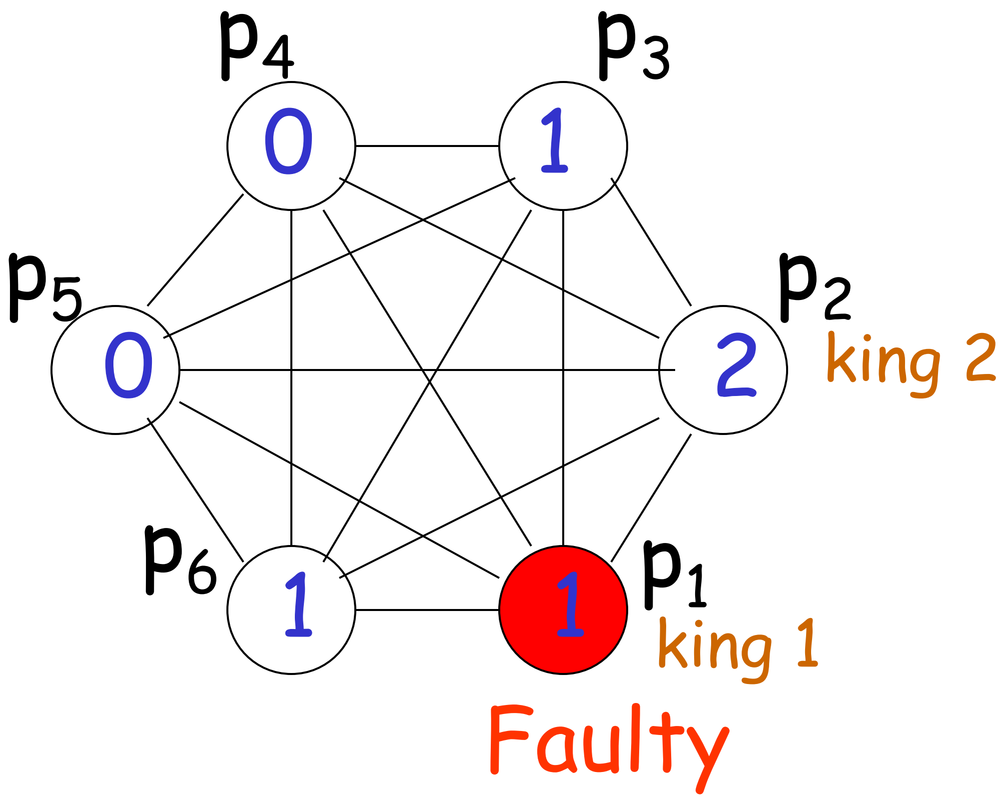
After receiving v_k, if p_i selected in Round 1 a preferred value v_i with a weak majority, i.e., $m_i < n/2+f+1$ (here non-uniformity is required), then set v_i:=v_k, otherwise maintain your preferred value v_i

The **King** algorithm

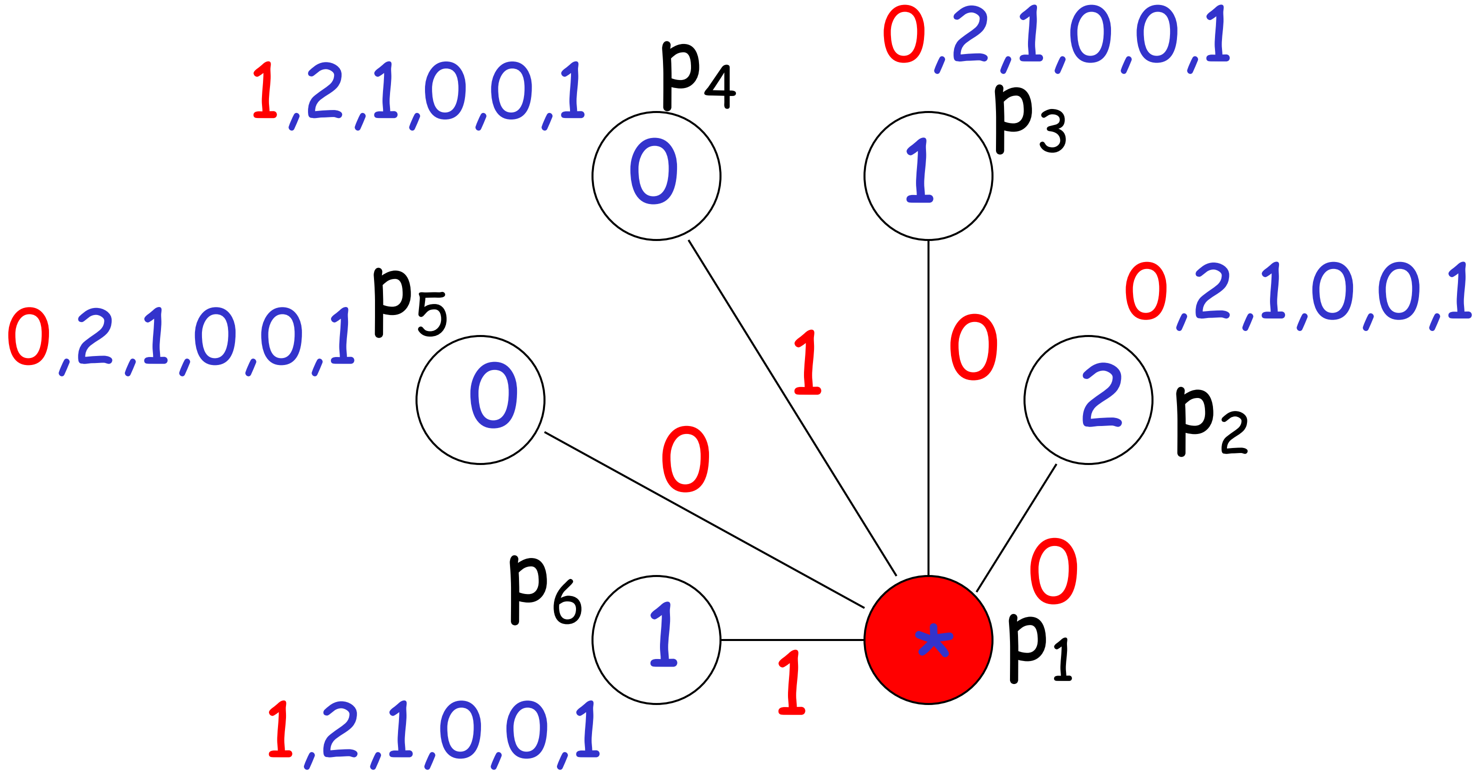
End of Phase f+1:

Each non-faulty processor decides on its preferred value

Example 1: 6 processors, 1 fault $\Rightarrow$ 2 phases



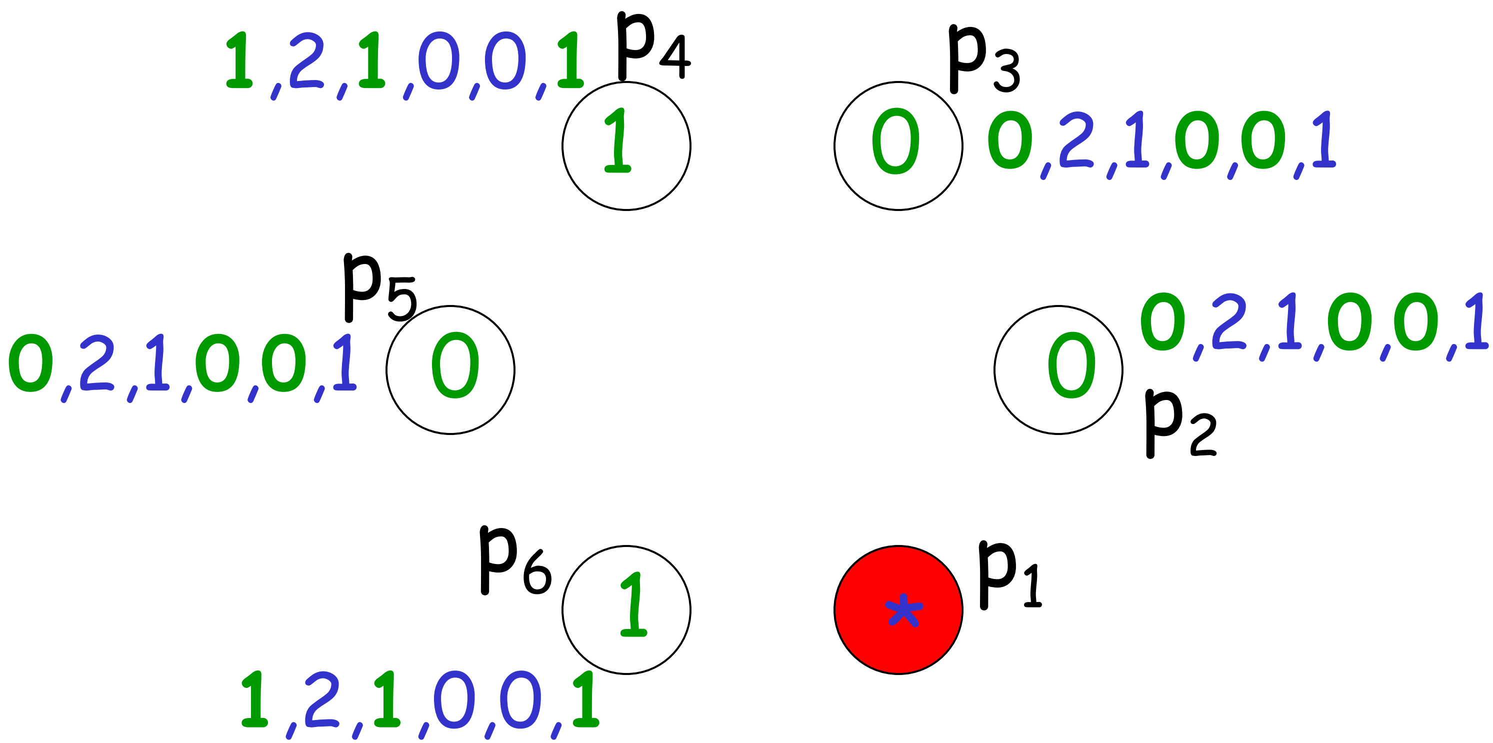
Phase 1, Round 1



Everybody broadcasts, and faulty p_1 sends arbitrary values

Phase 1, Round 1

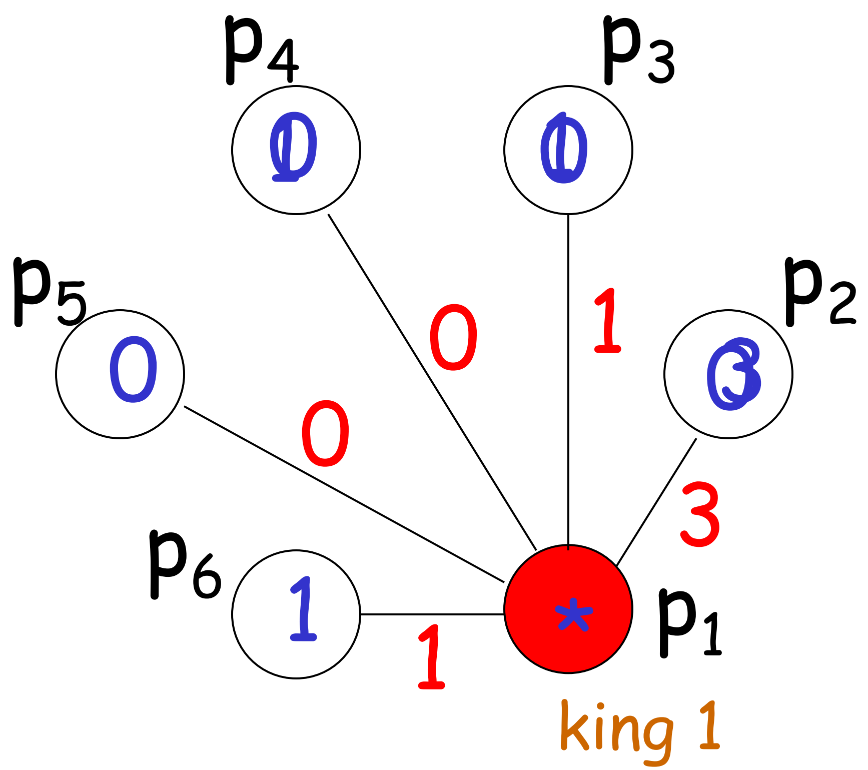
Choose the majority



Each (weak) majority is equal to $3 < \frac{n}{2} + f + 1 = 5$

$\Rightarrow$ On round 2, everybody will choose the king's value

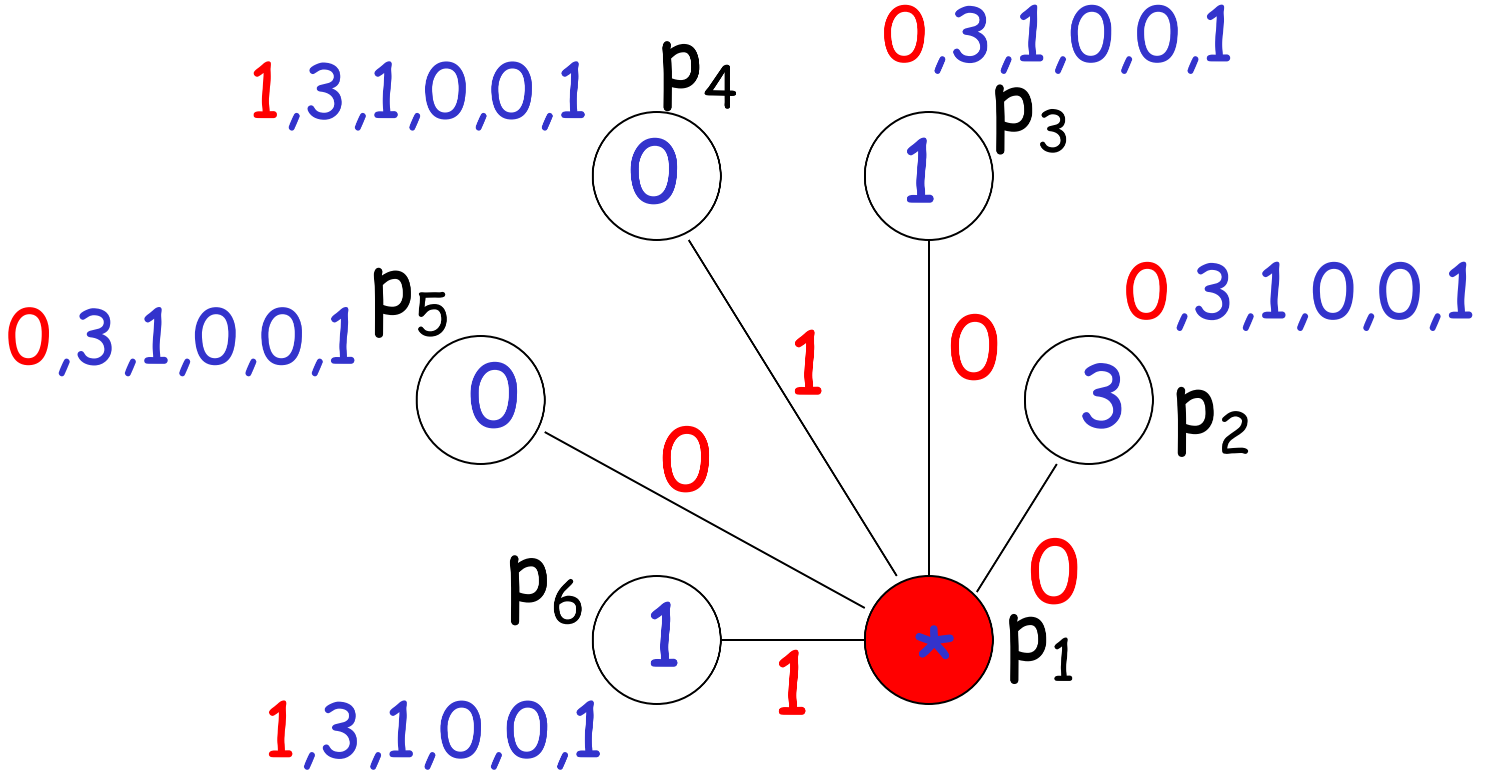
Phase 1, Round 2



The faulty king broadcasts arbitrary values

⇒ Everybody chooses the king's value

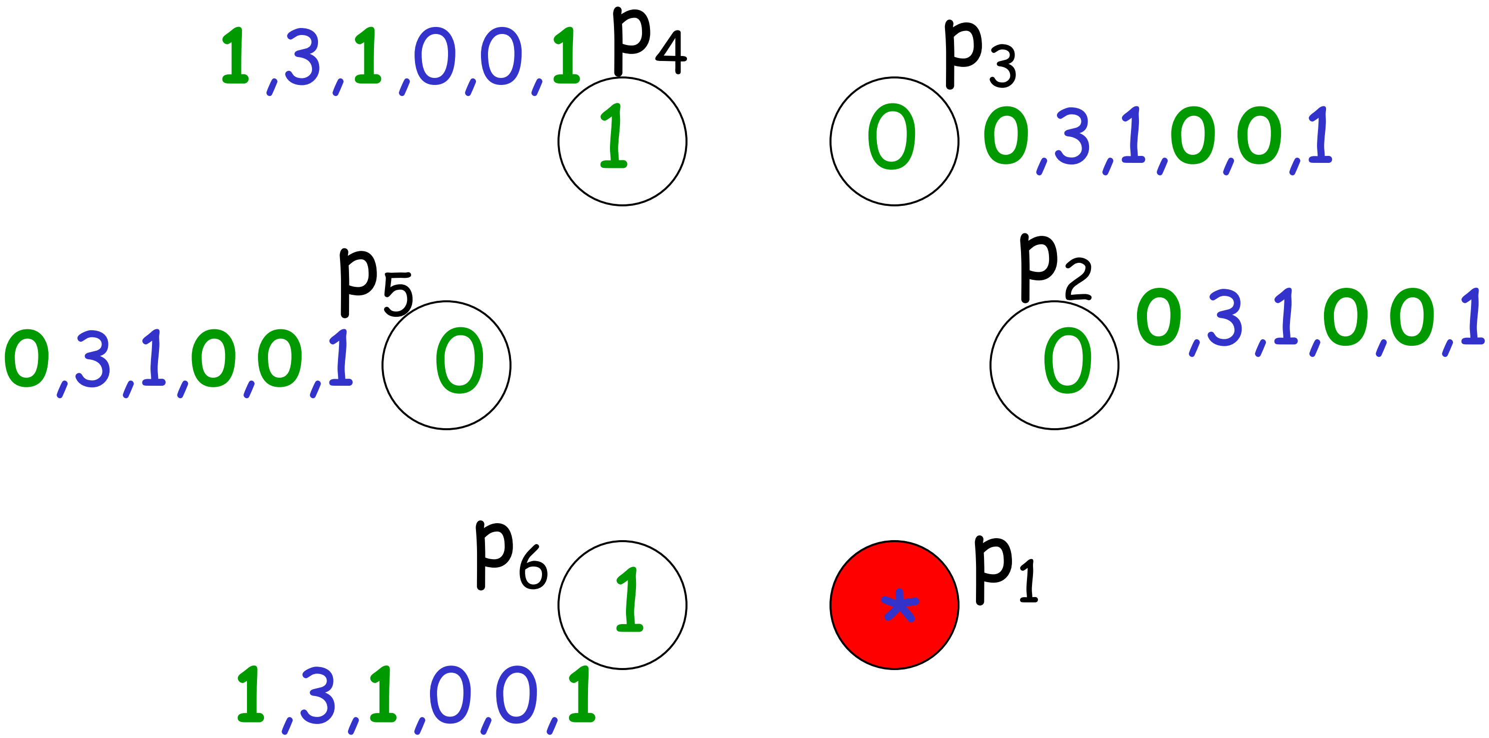
Phase 2, Round 1



Everybody broadcasts, and faulty p_1 sends arbitrary values

Phase 2, Round 1

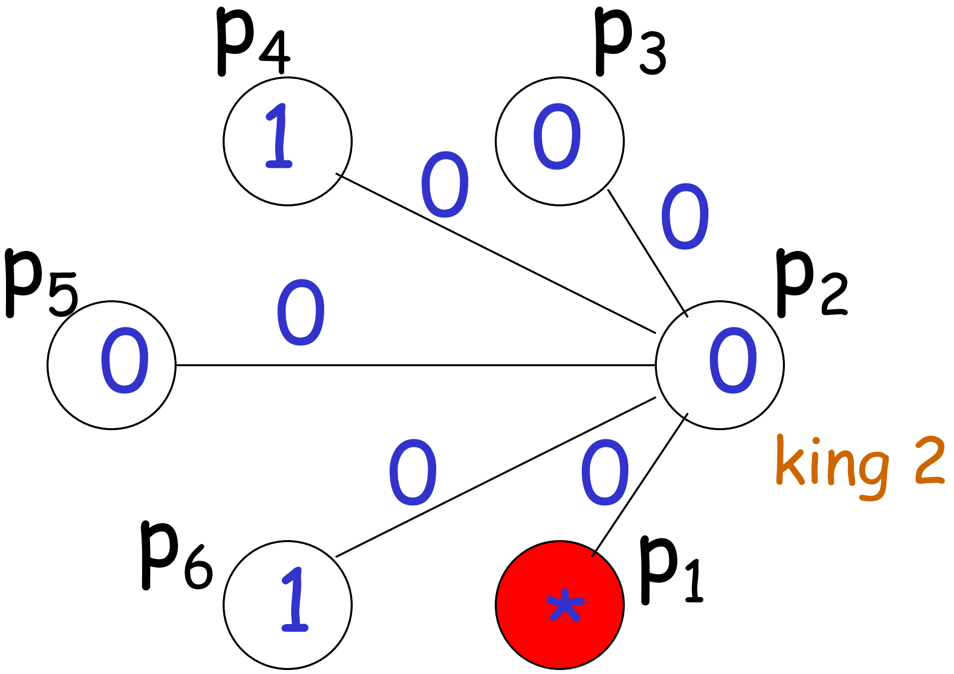
Choose the majority



Each (weak) majority is equal to $3 < \frac{n}{2} + f + 1 = 5$

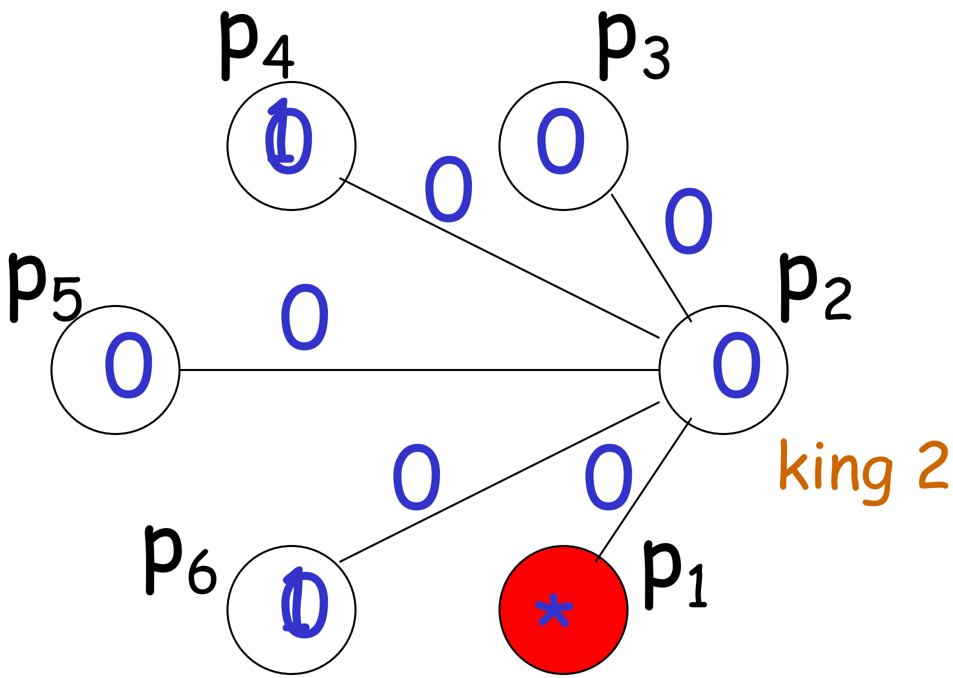
$\Rightarrow$ On round 2, everybody will choose the king's value

Phase 2, Round 2



The non-faulty king p_2 broadcasts its 0

Phase 2, Round 2

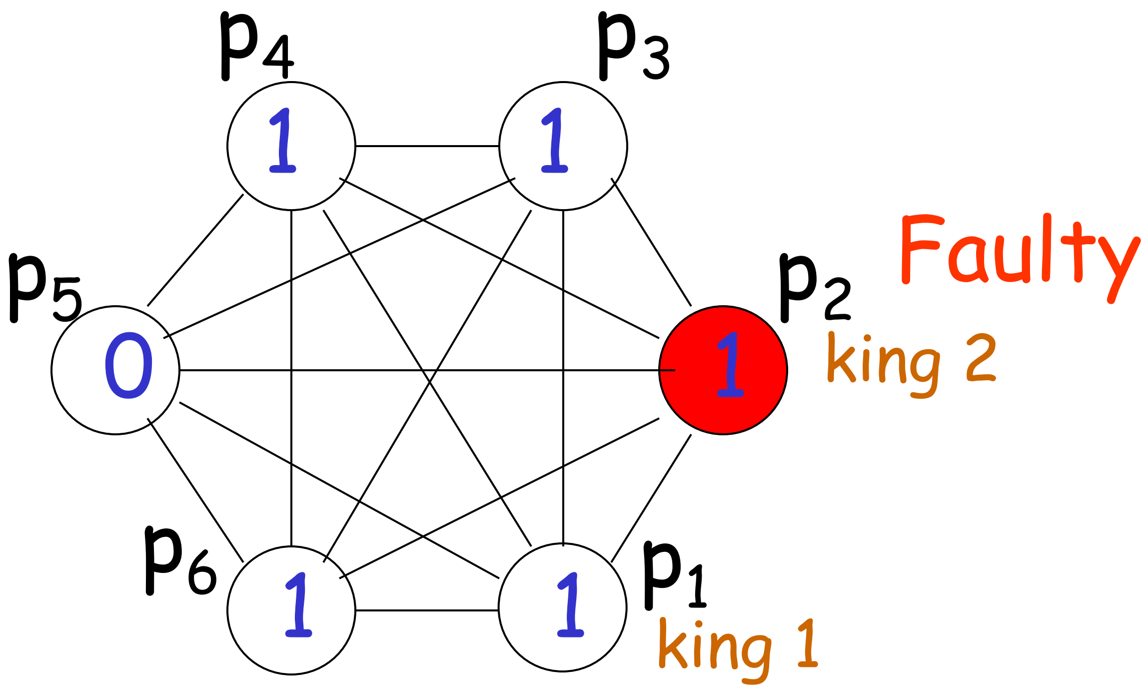


The non-faulty king p_2 broadcasts its 0

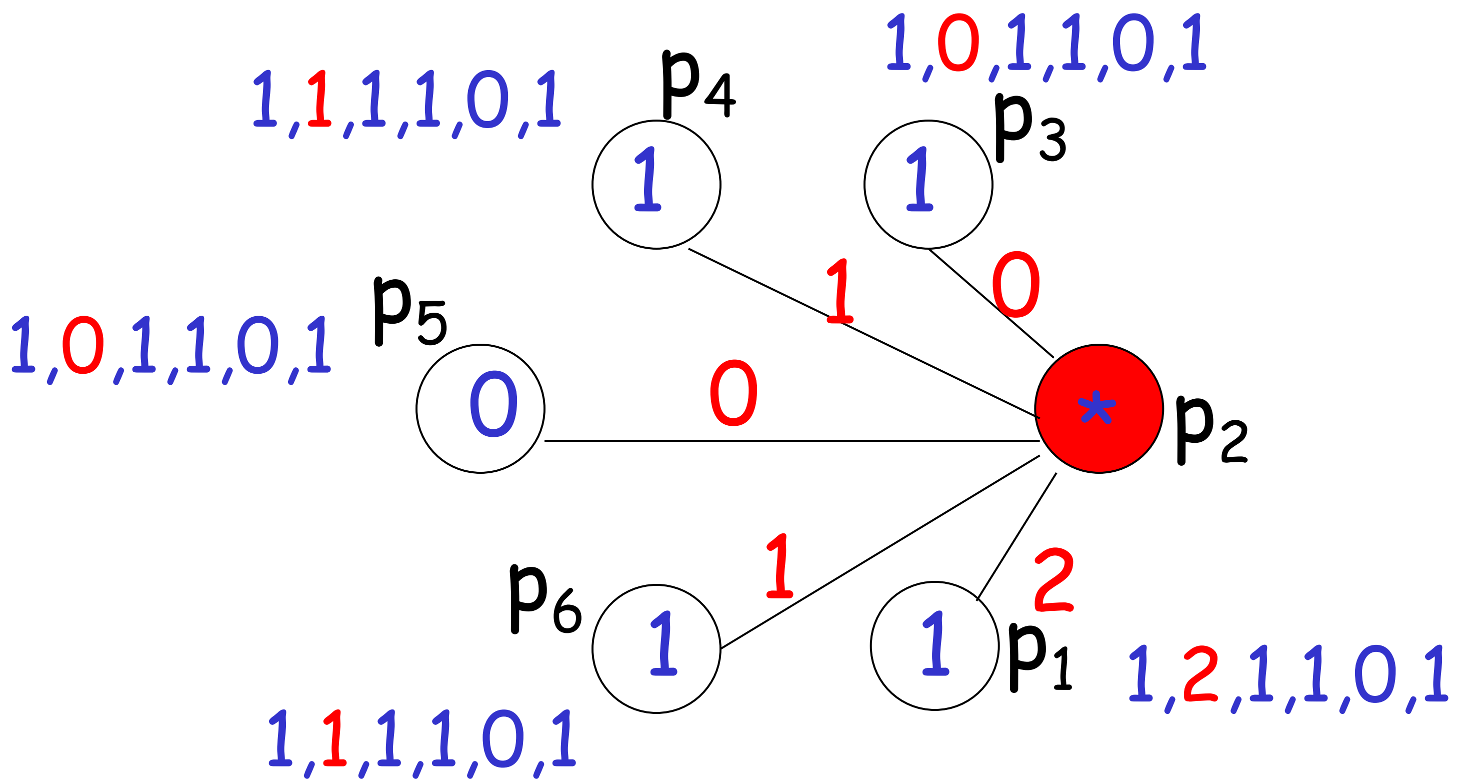
$\Rightarrow$ Everybody chooses the king's value

$\Rightarrow$ Final decision and agreement on 0

Example 2: 6 processors, 1 fault $\Rightarrow$ 2 phases



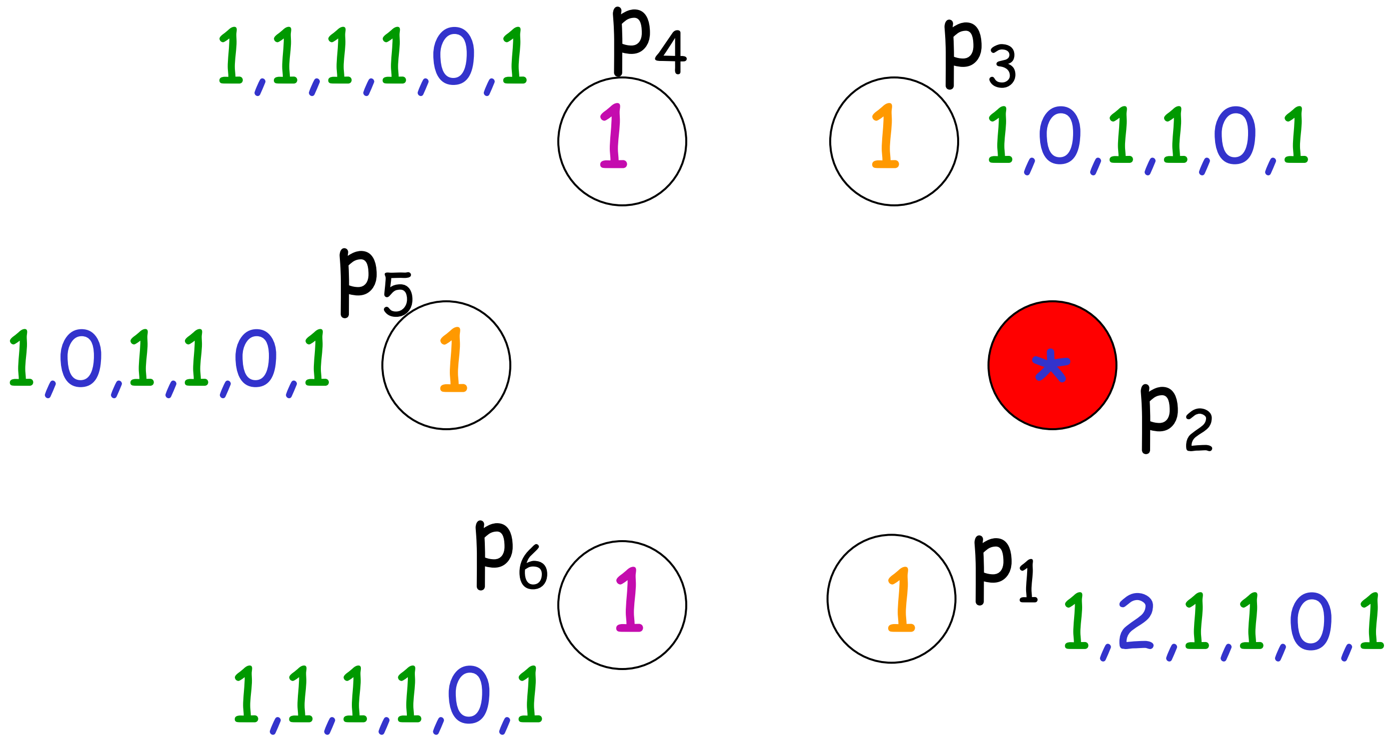
Phase 1, Round 1



Everybody broadcasts, and faulty p_2 sends arbitrary values

Phase 1, Round 1

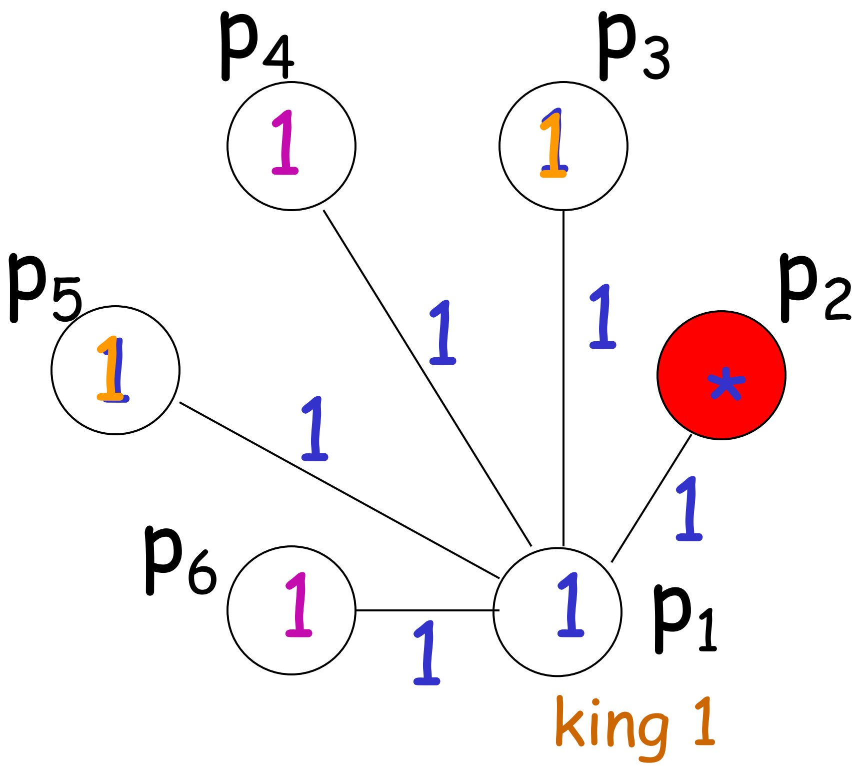
Choose the majority



Some majorities are strong (at least 5 votes), others are weak (less than 5 votes)

⇒ On round 2, somebody will choose the king's value, someone else will keep its own value

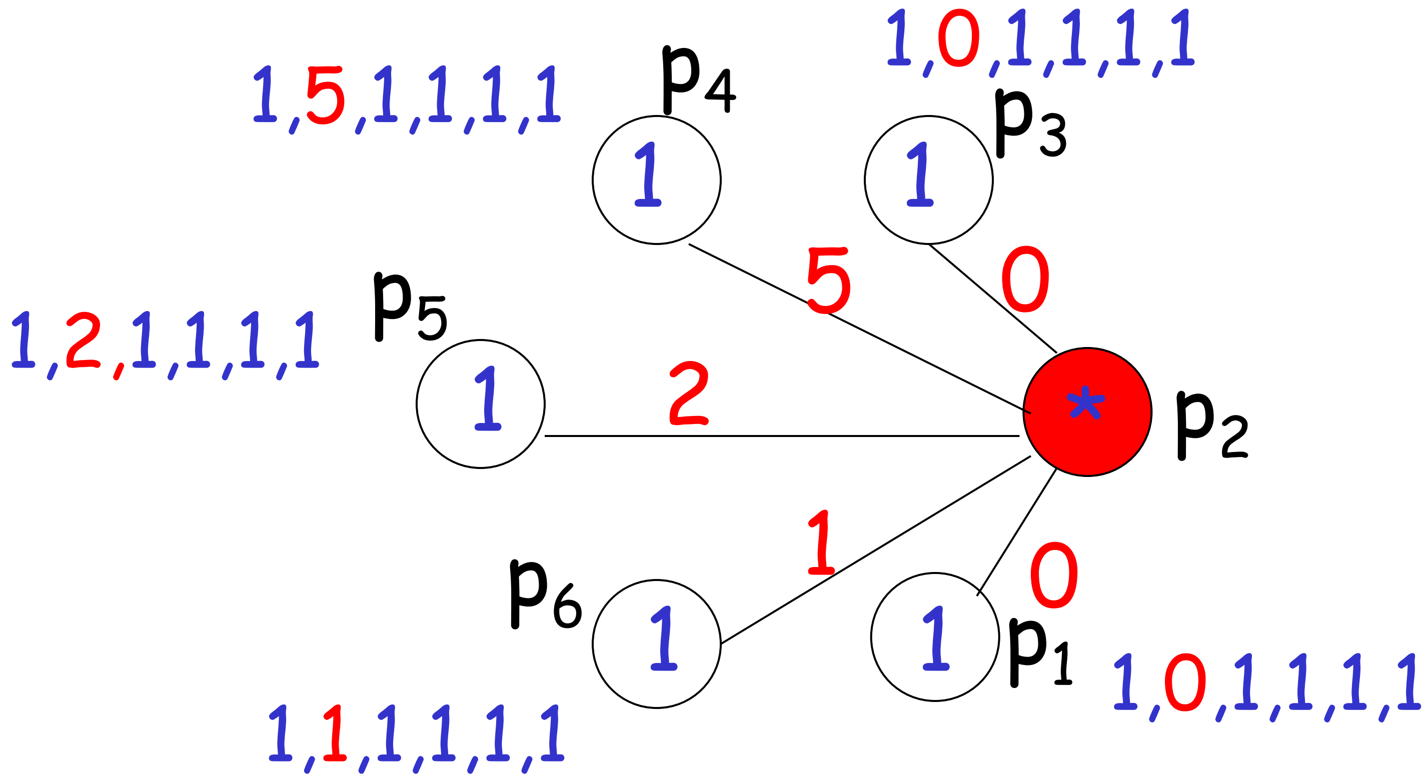
Phase 1, Round 2



The non-faulty king p_1 broadcasts its 1

$\Rightarrow$ Some processors switch to the king's value, but they will still selects 1!

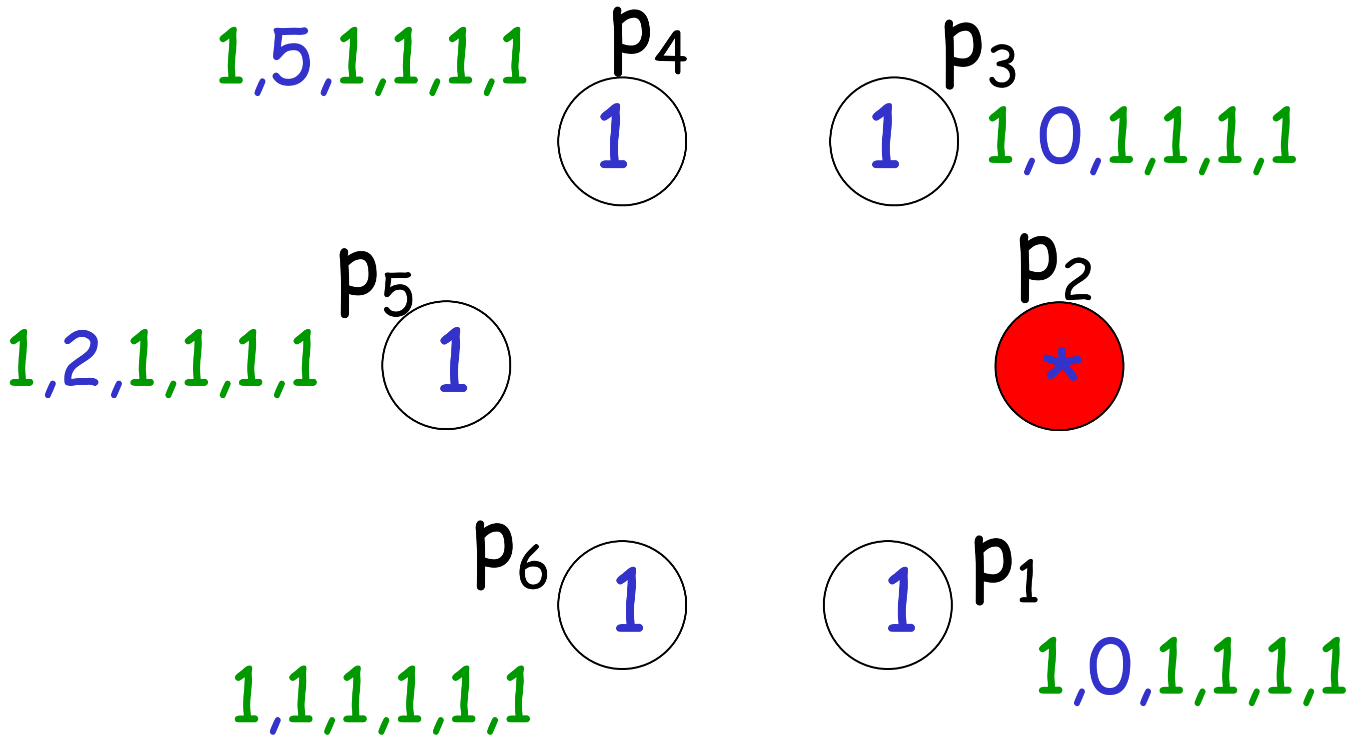
Phase 2, Round 1



Everybody broadcasts, and faulty p_2 sends arbitrary values

Phase 2, Round 1

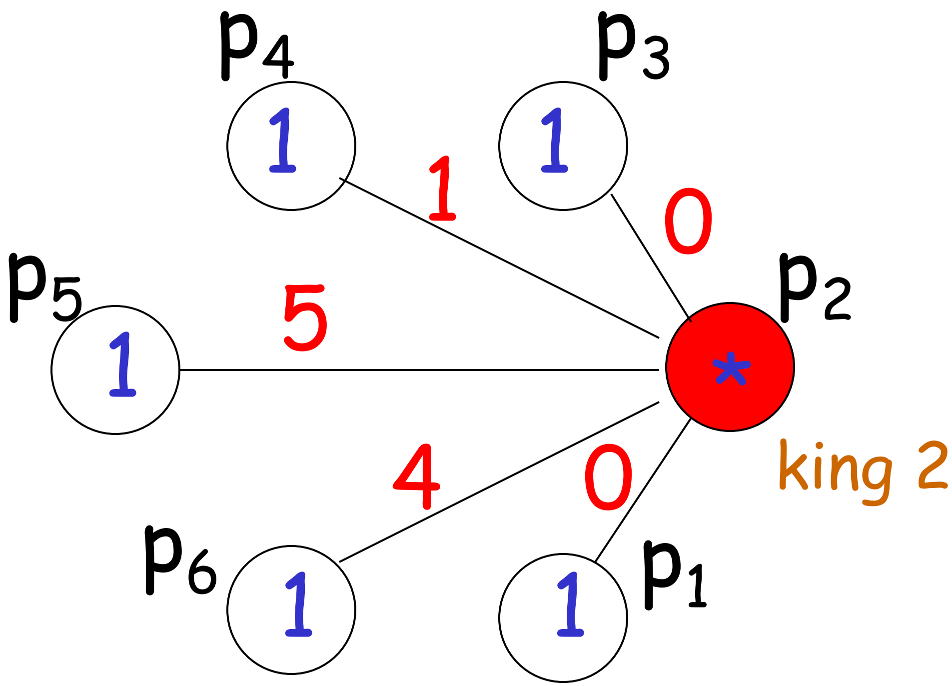
Choose the majority



Each majority is at least $5 = \frac{n}{2} + f + 1$ i.e., it's **strong**!

$\Rightarrow$ On round 2, nobody will choose the king's value

Phase 2, Round 2



The faulty king p_2 broadcasts arbitrary values, but nobody changes its preferred value

$\Rightarrow$ Final decision and agreement on 1

Correctness of the King algorithm

Lemma 1: At the end of a phase ϕ where the king is non-faulty, every non-faulty processor prefers the same value

Proof: Consider the end of round 1 of phase ϕ.

There are two cases:

Case 1: All non-faulty processors have chosen their preferred value with **weak** majority (i.e., $< n/2+f+1$ votes) [see phase 2 of Example 1]

Case 2: Some non-faulty processor has chosen its preferred value with **strong** majority (i.e., $\geq n/2+f+1$ votes) [see phase 1 of Example 2]

Case 1: All non-faulty processors have chosen their preferred value at the end of round 1 of phase ϕ with **weak** majority (i.e., $< n/2+f+1$ votes)

$\Rightarrow$ Every non-faulty processor will adopt the value broadcasted by the king during the second round of phase ϕ, thus all of them will prefer the same value, since the king is non-faulty

Case 2: Suppose a non-faulty processor p_i has chosen its preferred value a at the end of round 1 of phase ϕ with **strong** majority ($\geq n/2+f+1$ votes)

$\Rightarrow$ This implies that at least $n/2+1$ non-faulty processors must have broadcasted a at start of round 1 of phase ϕ, and then at the end of that round, every other non-faulty processor (**including the king**) must have received value a with an absolute majority of at least $n/2+1$ votes, and so such a value becomes preferred in these processors

At end of round 2, there are 2 cases:

1. If a non-faulty processor keeps its own value due to strong majority, then it maintains a
2. Otherwise, if a non-faulty processor adopts the value of the non-faulty king, then it prefers a as well, since the king has broadcasted a

Therefore: Every non-faulty processor prefers a

END of PROOF

Lemma 2: Let a be a common value preferred by non-faulty processors at the end of a phase ϕ. Then, a will be preferred until the end.

Proof: First of all, notice that the system contains at most f byzantine processors, and then at least $n-f$ non-faulty processors. But since $f<n/4$, it follows that $n-f>n/2+f$, since

$$f < \frac{n}{4} \Rightarrow 2f < \frac{n}{2} \Rightarrow 2f < n - \frac{n}{2} \Rightarrow n - 2f > \frac{n}{2} \Rightarrow n - f > \frac{n}{2} + f$$

This means, after ϕ, a will always be preferred with strong majority (i.e., $>n/2+f$), and so, until the end of phase $f+1$, every non-faulty processor will keep on preferring a. QED

Agreement in the King algorithm

Follows from Lemma 1 and 2, observing that since there are f+1 phases and at most f failures, there is al least one phase in which the king is non-faulty (and thus from Lemma 1 at the end of that phase all non-faulty processors prefer the same value, and from Lemma 2 this preference will be maintained until the end).

Validity in the King algorithm

Follows from the fact that if all (non-faulty) processors have a as input, then in round 1 of phase 1 each non-faulty processor will receive a at least n-f times, i.e., with strong majority, since as we observed in Lemma 2:

$$n-f > \frac{n}{2}+f$$

and so in round 2 of phase 1 this will be the preferred value of all non-faulty processors, independently of the king's broadcasted value. From Lemma 2, this will be maintained until the end, and will be exactly the decided output!

QED

Performance of King Algorithm

- Number of processors: $n > 4f$ (we will see it is not tight)
- $2(f+1)$ rounds (we will see it is not tight)
- $\Theta(n^2 \cdot f) = O(n^3)$ messages. Indeed, each non-faulty node sends n messages in the first round of each phase, each containing a given preference value, and each non-faulty king sends $n-1$ messages in the second round of each phase. Notice that we are not considering the fact that a byzantine processor could in principle generate an unbounded number of messages!

Homework

Show an execution with $n=4$ processors and $f=1$ for which the King algorithm fails.

Discuss the 3 possible cases:

1) Neither p_1 nor p_2 is faulty
2) p_1 is faulty
3) p_2 is faulty

An Impossibility Result (M.C. Pease, R.E. Shostak, and L. Lamport, *JACM 1980*)

Theorem: There is no f-resilient to byzantine failures algorithm for n processors when

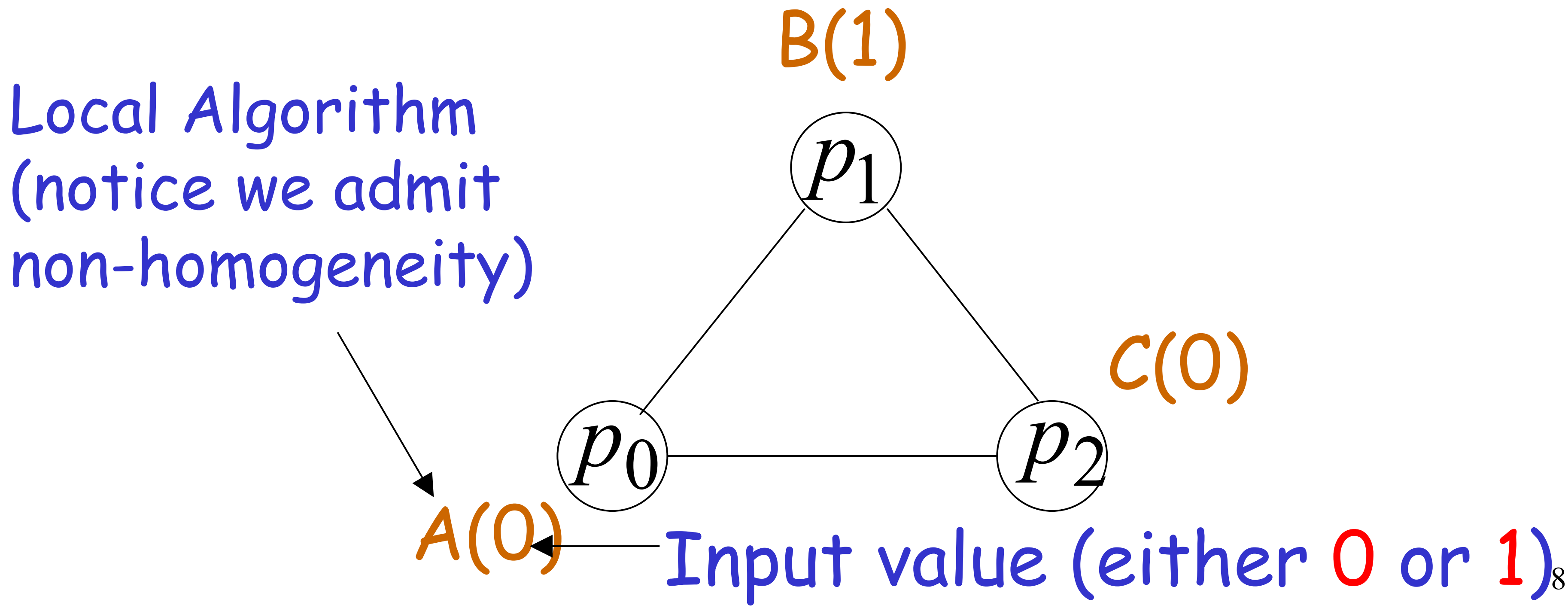
$$f \geq \frac{n}{3}$$

Proof: First we prove the 3 processors case, and then the general case

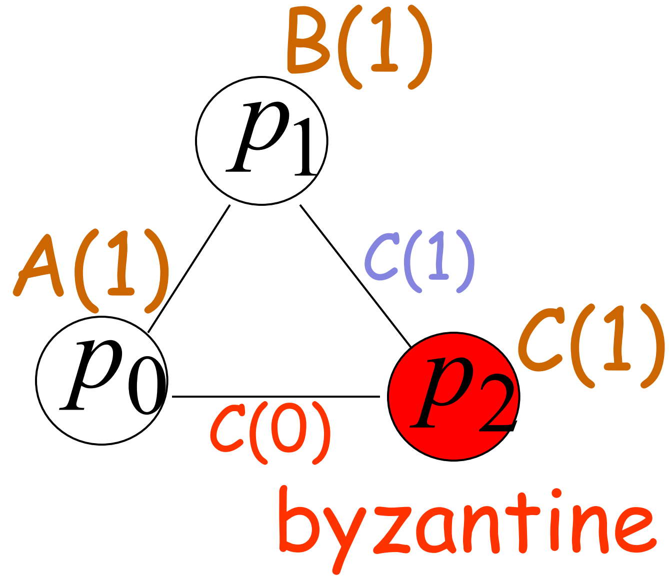
The 3 processors case

Lemma: There is no 1-resilient to byzantine failures algorithm for 3 processors

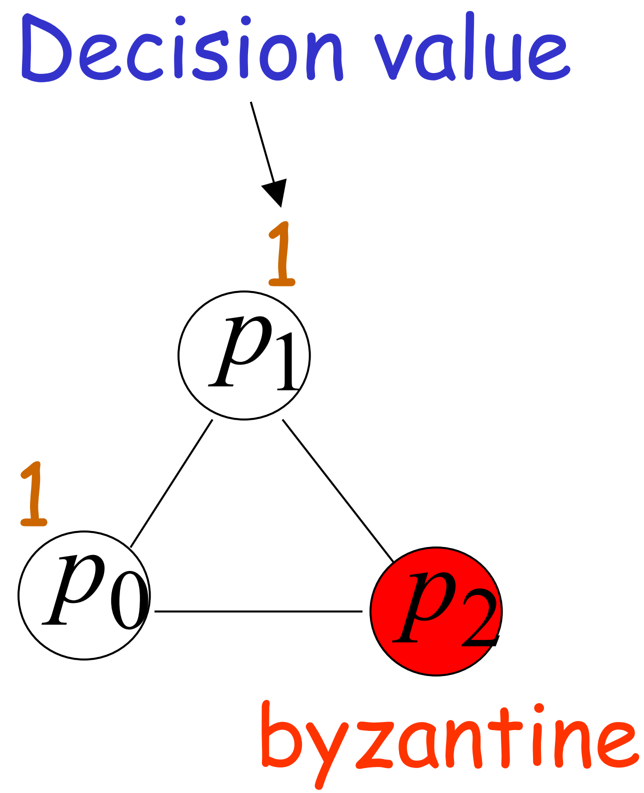
Proof: Assume by contradiction that there is a 1-resilient algorithm for 3 processors



A first execution



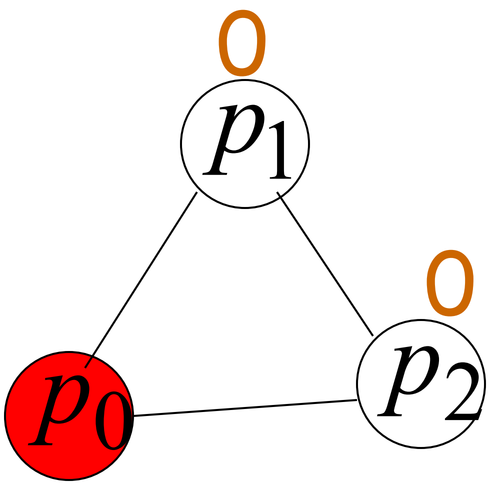
p_2 behaves (we don't know exactly what it will do) towards p_0 (resp., p_1) has if it had input 0 (resp., 1)



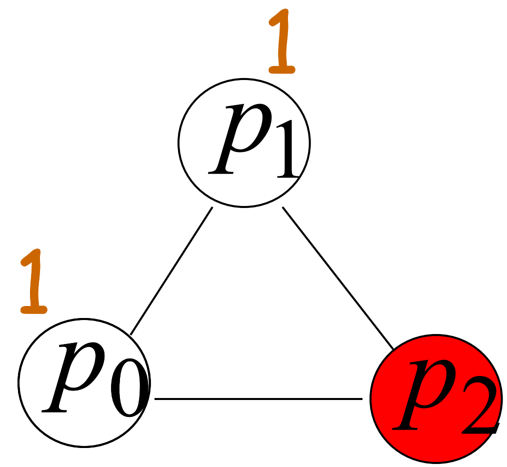
(validity condition)

A second execution

p_0 behaves towards p_1 (resp., p_2) has if it had input 0 (resp., 1)



byzantine

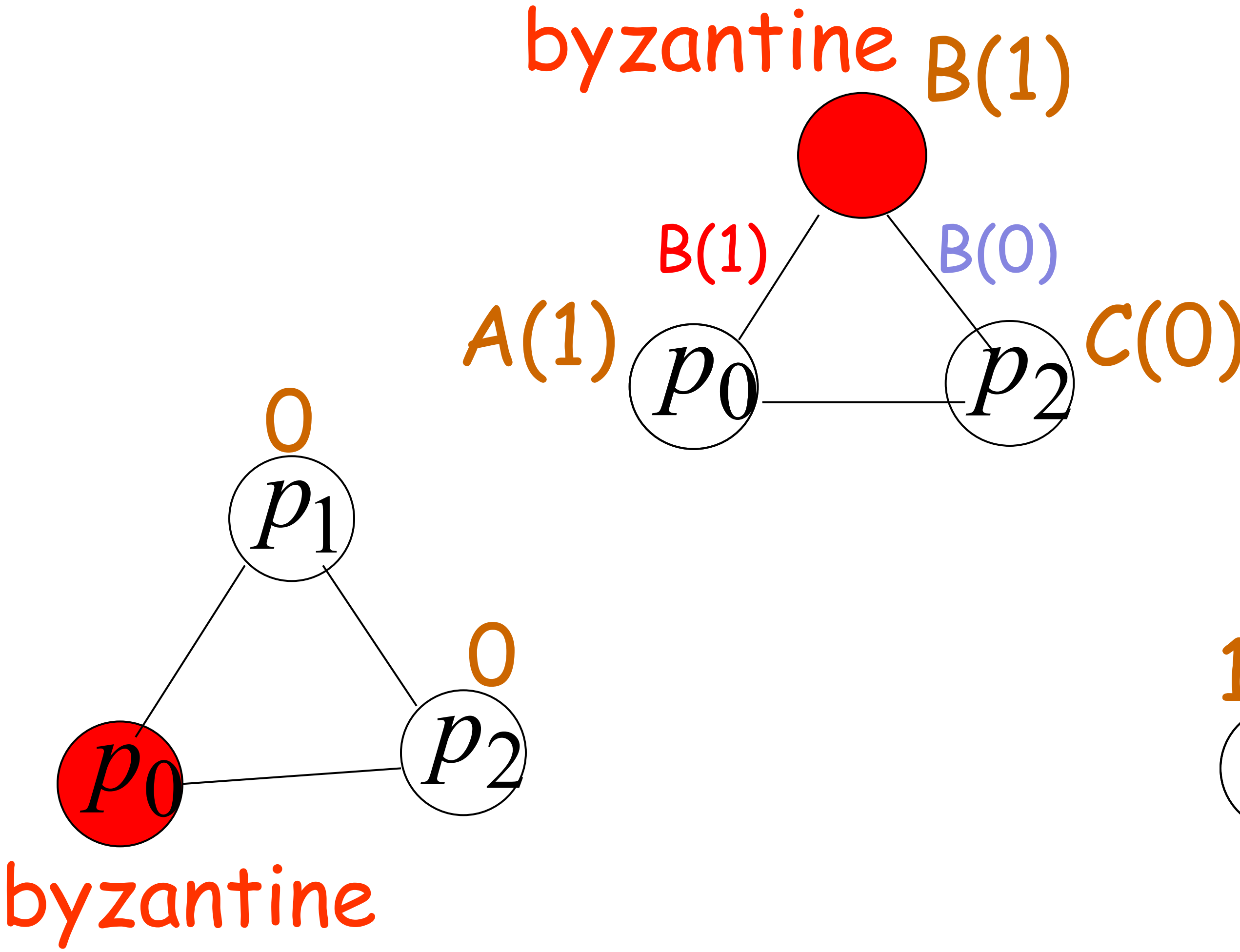


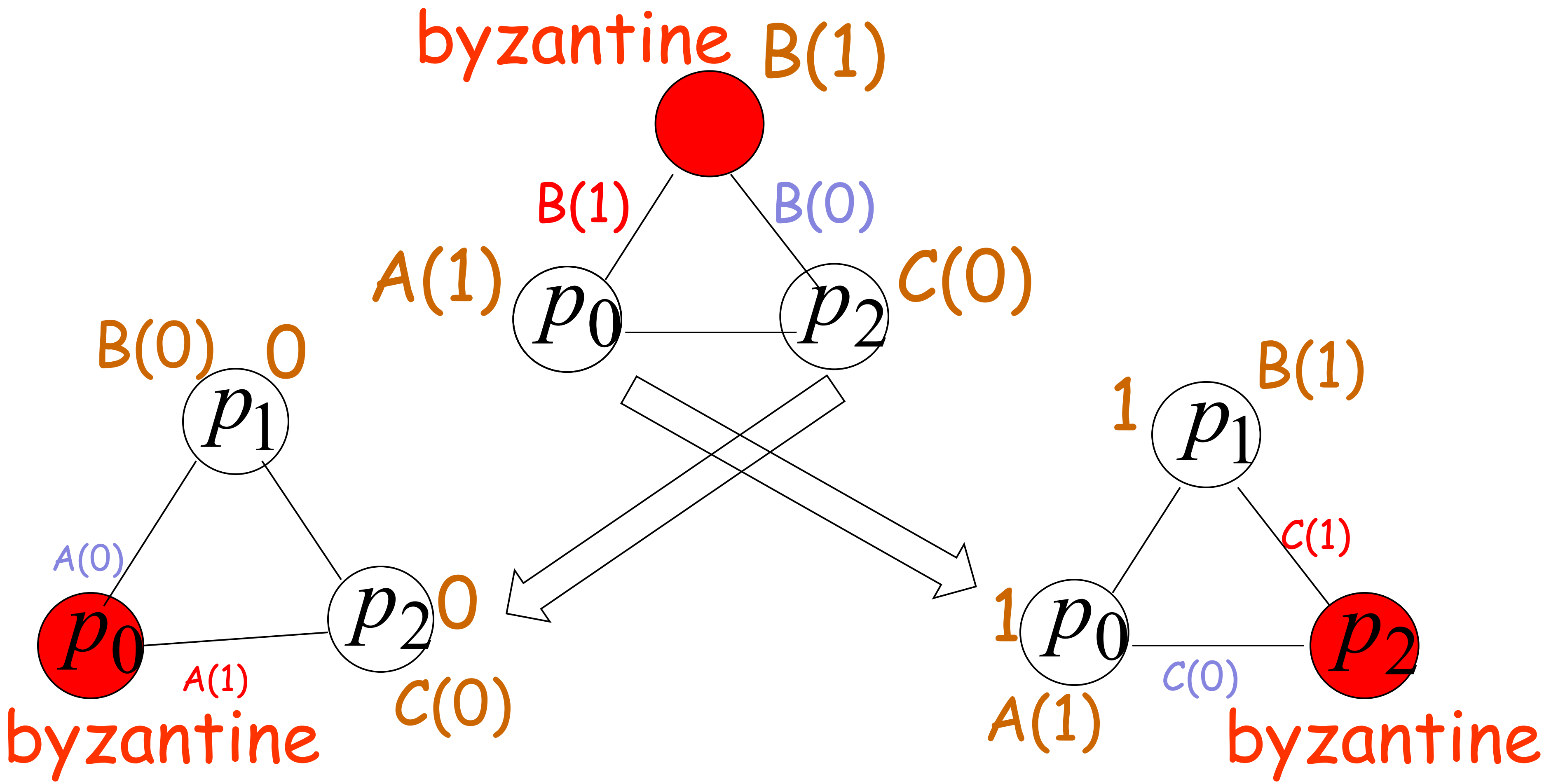
byzantine

(validity condition)

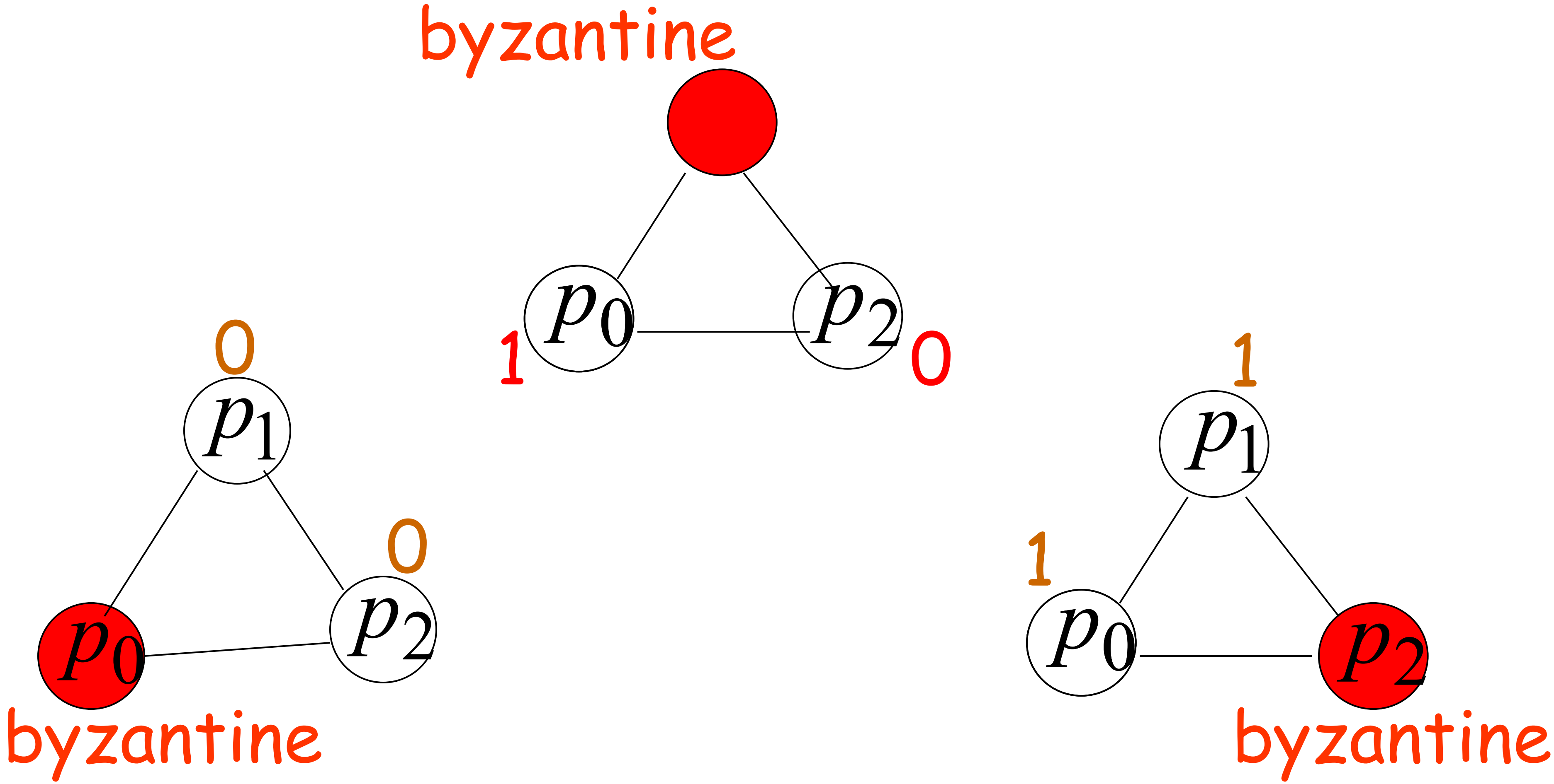
A third execution

p_1 behaves towards p_2 (resp., p_0) has if it had input 0 (resp., 1)





The **view** of p_2 (resp., p_0) in the third execution, namely the **behavior** of p_0 and p_1 (resp., p_1 and p_2) it observes, and thus its own behavior, is exactly the same as in the second (resp., the first) execution, so it must take the same decision as before!



No agreement!!! Contradiction, since the algorithm was supposed to be 1-resilient

Therefore:

There is no algorithm that solves
consensus for 3 processors
in which 1 is a byzantine!

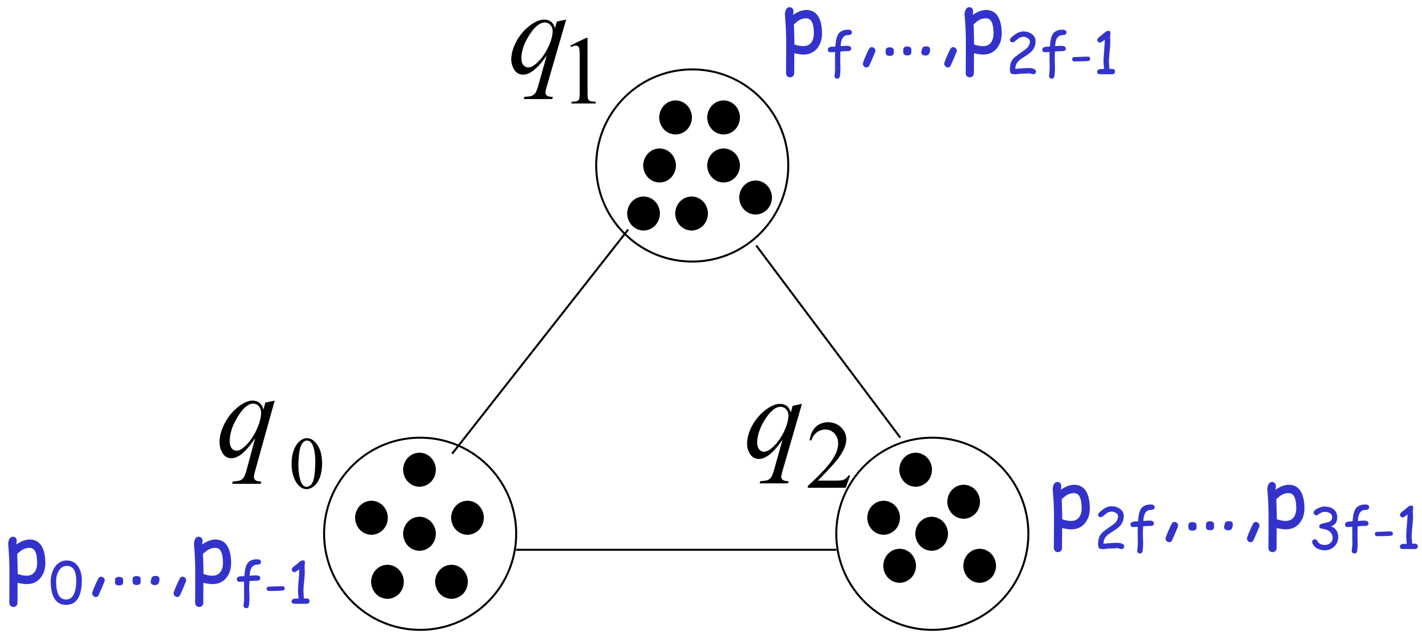
The n processors case

Assume by contradiction that there is an f-resilient distributed algorithm *A* for n>3 processors for $f \geq \frac{n}{3}$

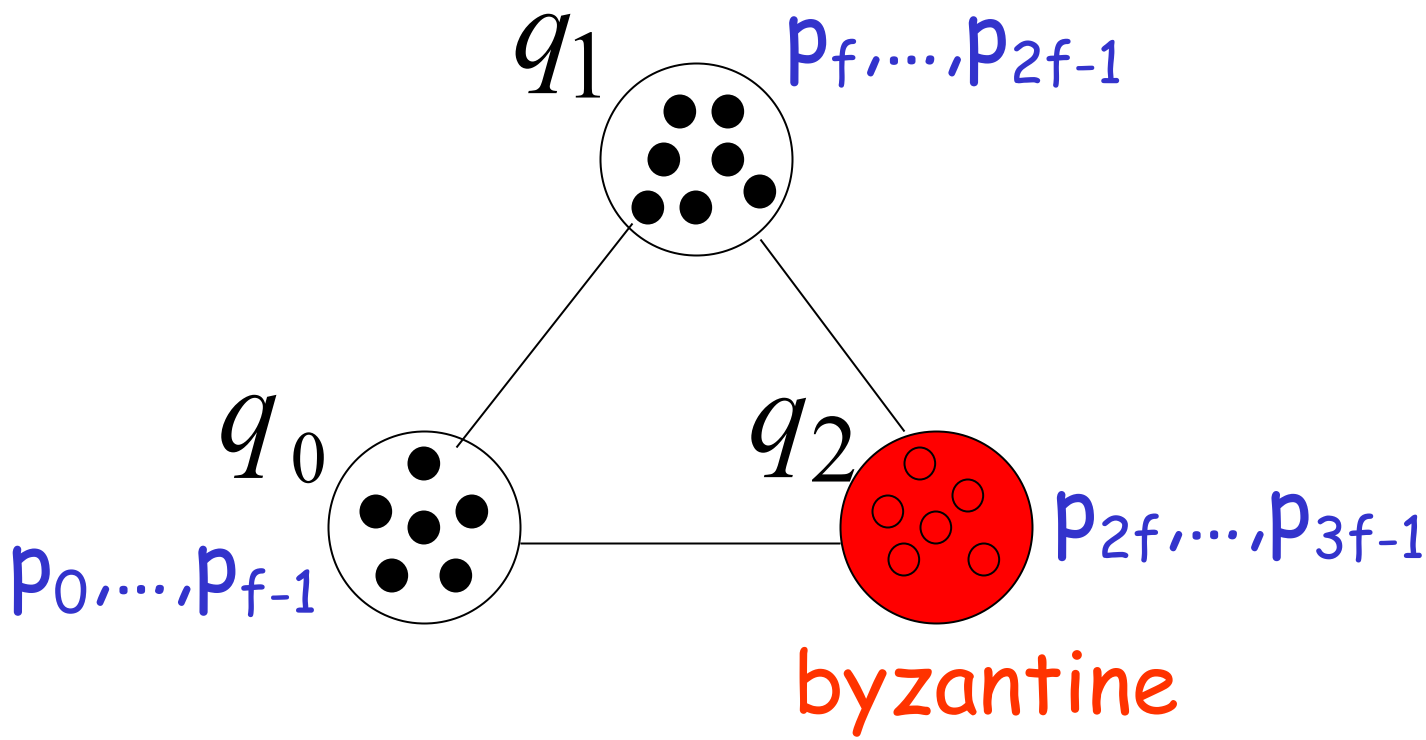
We will use *A* to solve consensus for 3 processors and 1 byzantine failure

(contradiction)

W.l.o.g. let $n=3f$, and let $P=\langle p_0,p_1,\ldots,p_{3f-1}\rangle$ be the n-processor system. We partition arbitrarily the n processors in 3 sets P_0,P_1,P_2, each containing $n/3$ processors; then, given a 3-processor system $Q=\langle q_0,q_1,q_2\rangle$, we associate each q_i with P_i

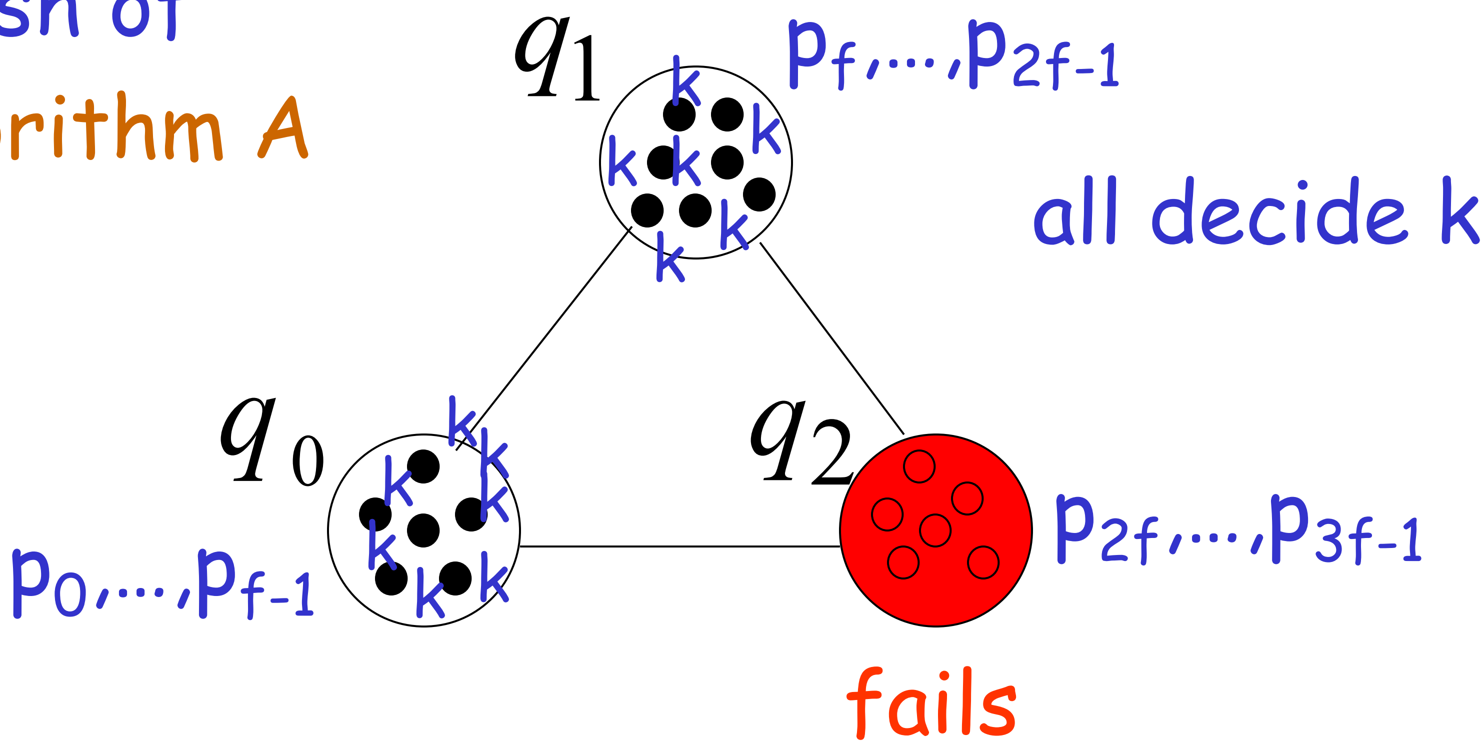


Each processor q_i simulates the execution of algorithm *A* once restricted to the set P_i of $n/3$ processors. In particular, q_i decides k if the majority of its processors decides k.



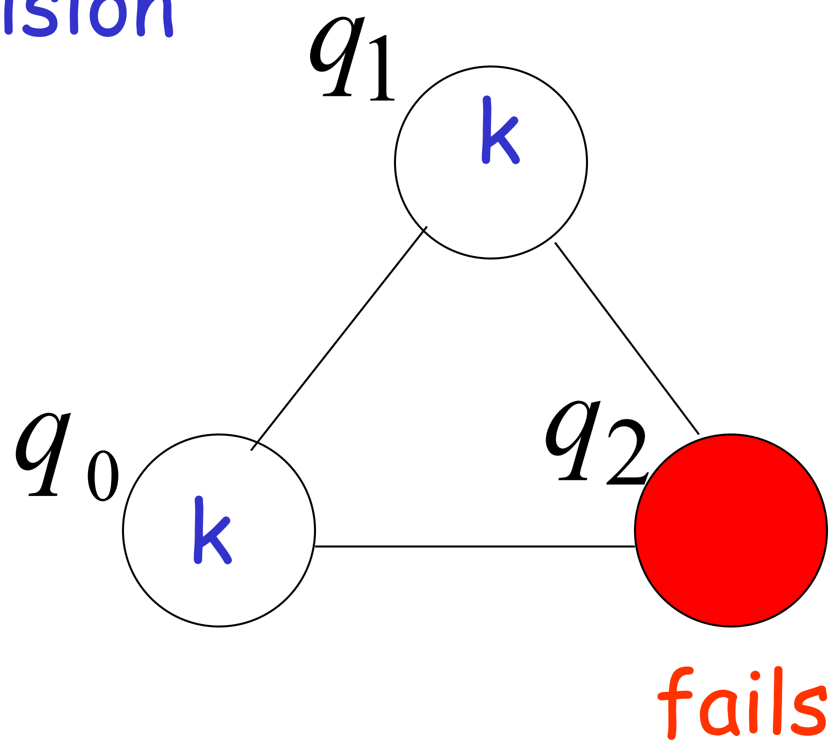
When a processor in Q fails, then at most n/3 original processors in the original n-processor system P are affected

Finish of algorithm A



But we were assuming that the original algorithm A tolerates at most $f=n/3$ failures, so the remaining $2f$ processors must agree!

Final decision



We reached consensus with 1 failure

Impossible!!!

Therefore:

There is no f-resilient to byzantine failures algorithm for n processors in case

$$f \geq \frac{n}{3}$$

Question:

Is there an f-resilient to byzantine failures algorithm for n processors if $f<n/3$, namely for $n \geq 3f+1$?

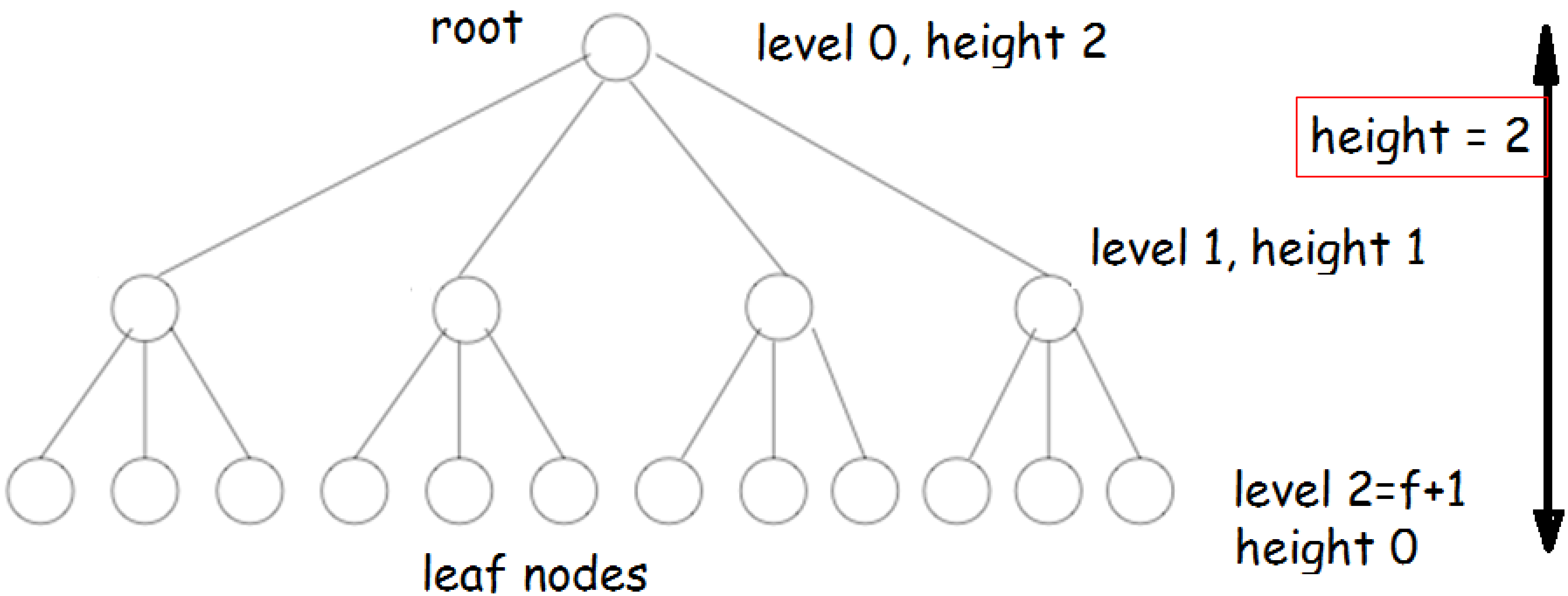
For $n \geq 4f+1$, YES (King algorithm), but what about $3f+1 \leq n < 4f+1$, and in particular $n=3f+1$?

Exponential Tree Algorithm (a.k.a. *Exponential Information Gathering (EIG)* algorithm, M.C. Pease, R.E. Shostak, and L. Lamport, JACM 1980)

- This algorithm uses
  - $n=3f+1$ processors (optimal)
  - $f+1$ rounds (optimal)
  - exponential number of messages (sub-optimal, the King algorithm was using only $O(n^3)$ msgs)
- Each processor keeps a rooted tree data structure in its local state
- From a topological point of view, all the trees are identical: they have height $f+1$, each root has n children, the number of children decreases by 1 at each level, and all the leaves are at the same level
- Values are filled top-down in the tree during the $f+1$ rounds; more precisely, during round i, level i of the tree is filled (the root is at level 0)
- At the end of round $f+1$, the values in the tree are used to compute bottom-up the decision.

Example of Local Tree

The tree when n=4 and f=1:

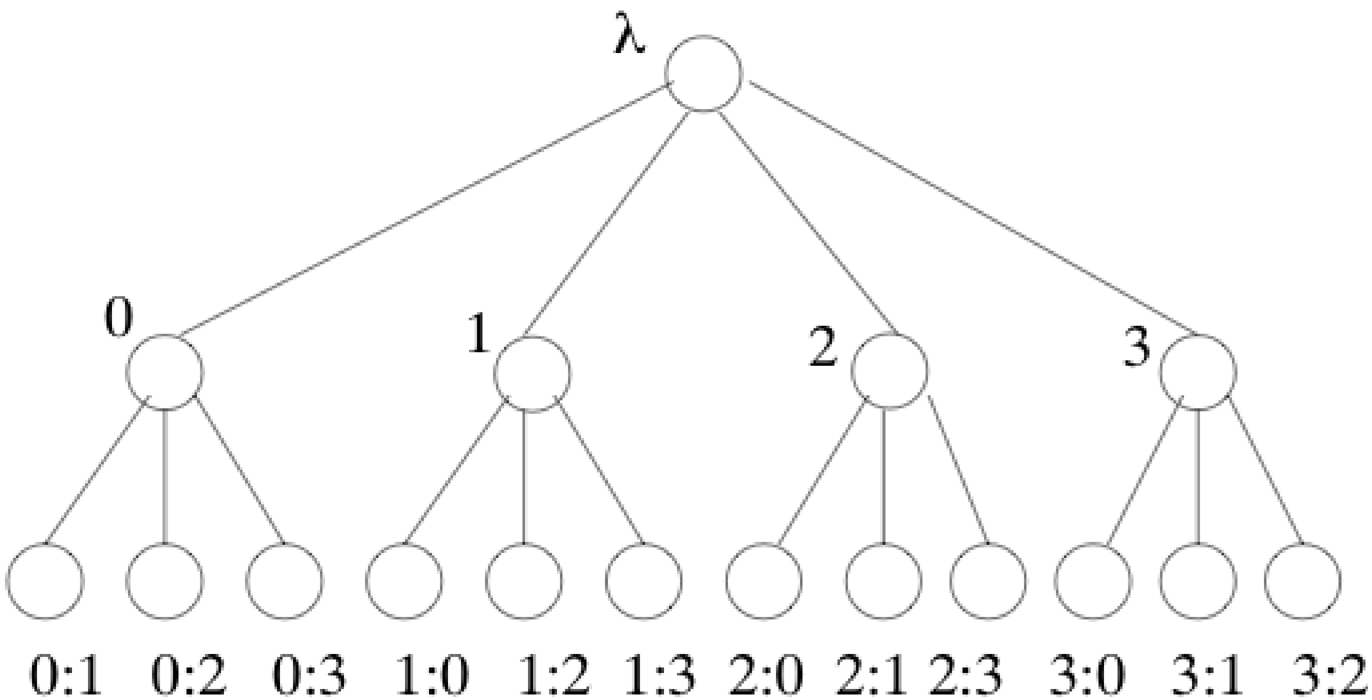


Local Tree Data Structure

- **Assumption**: Similarly to the King algorithm, processors have (distinct) ids (now in {0,1,…,n-1}), and we denote by p_i the processor with id i; this is common knowledge, i.e., processors cannot cheat about their ids;
- Each tree node is labeled with a sequence of unique processor ids in 0,1,…,n-1 defined recursively as follows:
  - Root's label is the empty sequence λ (the root has level 0 and height f+1);
  - Root has n children, labeled 0 through n-1
  - The child node of the root (level 1) with label i has n-1 children, labeled i:0 through i:n-1 and skipping i:i;
  - A node at level d>1 has a label made up of d **distinct** indexes, say $i_1:i_2:\ldots:i_{d-1}:i_d$, where $i_1:i_2:\ldots:i_{d-1}$ is the label of its parent, and i_d is a value in 0,1,…,n-1; morover, if d<f+1, such a node has n-d children, labeled $i_1:i_2:\ldots:i_d:0$ through $i_1:i_2:\ldots:i_d:n-1$, skipping any index $i_1,i_2,\ldots,i_d$;
  - Nodes at level f+1 are leaves with label $i_1:i_2:\ldots:i_{f+1}$ and have height 0.

Labels of the Sample Local Tree

The tree when n=4 and f=1:

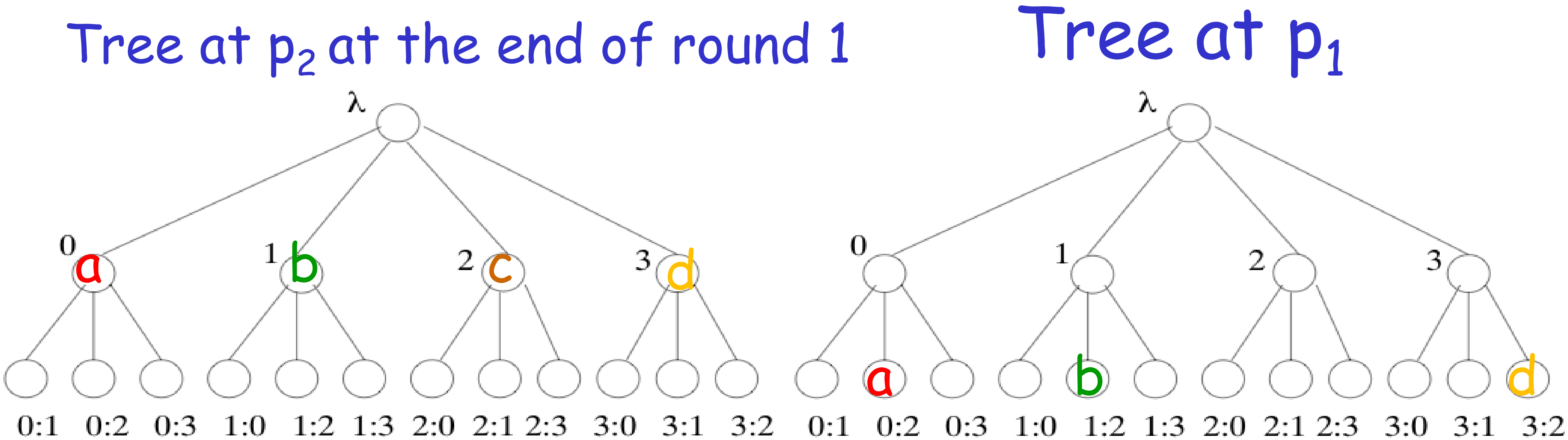


Filling-in the Tree Nodes

- **Round 1:**
  - Initially store your input in the root (level 0)
  - send level 0 of your tree (i.e., your input) to all (including yourself)
  - store value x received from p_j, j=0,…,n-1, in tree node labeled j (level 1); use a default value "*" (known to all!) if necessary (i.e., in case a value is not received or it is unfeasible)
  - node labeled j in the tree associated with p_i now contains what "p_j told to p_i" about its input (assuming p_i is non-faulty)
- **Round 2:**
  - send level 1 of your tree to all, including yourself (this means, send n messages to each processor)
  - let $\{x_0,\dots,x_{n-1}\}$ be the set of values that p_i receives from p_j; then, p_i discards x_j, and stores each remaining x_k in level-2 node labeled k:j (and use default value "*" if necessary)
  - node k:j in the tree associated with p_i now contains "p_j told to p_i that "p_k told to p_j that its input was x_k""

Example: filling the Local Tree at round #2

As before, n=4 and f=1, and assume that non-faulty p_2 tells to non-faulty p_1 that the first level of its local tree contains {a,b,c,d}; then, p_1 stores in the local tree:



⇒ The value c is not stored in the tree at p_1 since there is no node with label 2:2

Filling-in the Tree Nodes (2)

- .
- .
- .
- Round d>2:
  - send level d-1 of your tree to all, including yourself (this means, send $n(n-1)\ldots(n-(d-2))$ messages to each processor, one for each node on level $d-1$)
  - Let x be the value that p_i receives from p_j for node of level d-1 labeled $i_1:i_2:\ldots:i_{d-1}$, with $i_1,i_2,\ldots,i_{d-1} \neq j$; then, p_i stores x in tree node labeled $i_1:i_2:\ldots:i_{d-1}:j$ (level d), using default value "*" if necessary
- Continue for f+1 rounds

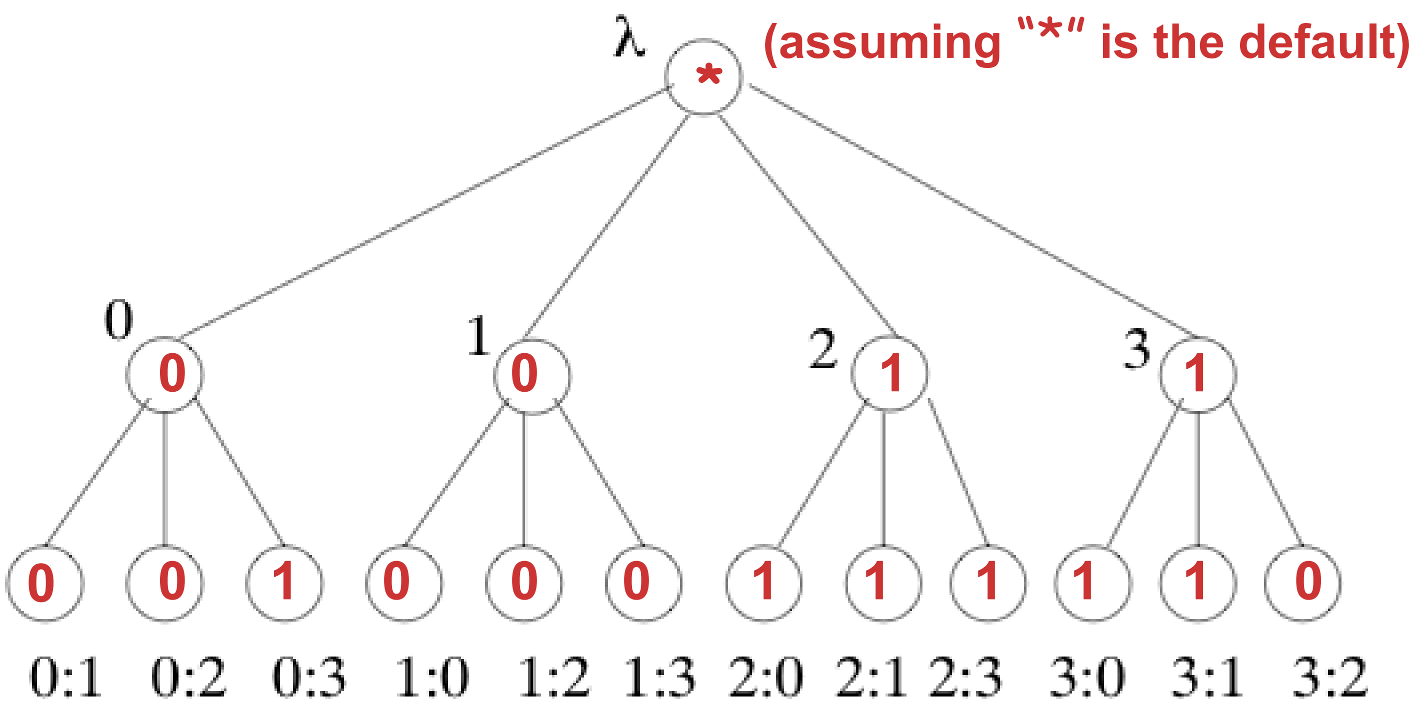
Calculating the Decision

- In round f+1, each processor uses the values in its tree to compute its final decision (output)
- Recursively compute the "resolved" value for the root of the tree, **`resolve`**(λ), based on the "resolved" values for the other tree nodes:

$$\texttt{resolve}(\pi) = \begin{cases} \text{value in tree node labeled } \pi \text{ if it is a leaf} \\ \text{majority}\{\texttt{resolve}(\pi') : \pi' \text{ is a child of } \pi\} \text{ otherwise (use default “*” if tied)} \end{cases}$$

Example of Resolving Values

The tree when n=4 and f=1:



Resolved Values are consistent

Lemma 1: If p_i and p_j are non-faulty, then p_i's resolved value for tree node labeled $\pi=\pi'j$ is equal to what p_j stores in its node π' during the filling-up of the tree (and so the value stored in π by p_i is the same value which is resolved in π by p_i, i.e., the resolved value is consistent with the stored value). (Notice this lemma **does not hold** for the root)

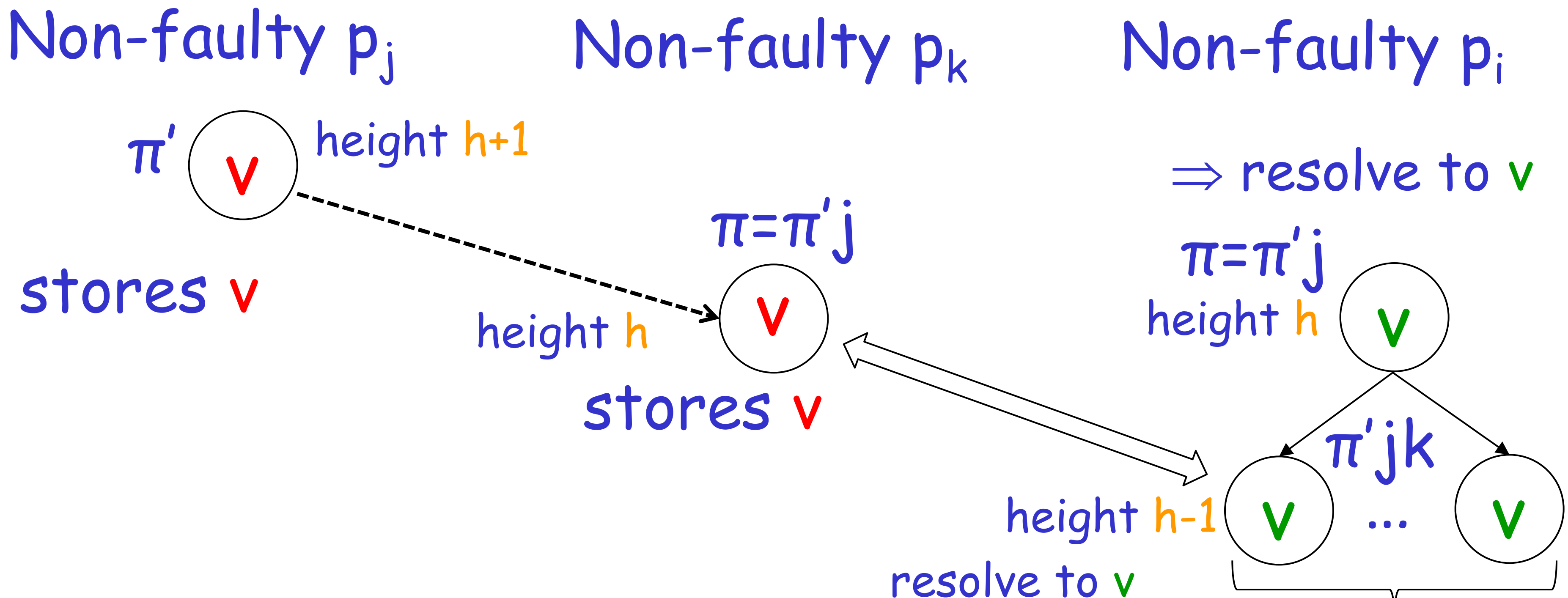
Proof: By induction on the height h of tree node π.

- **Basis**: π is a leaf, i.e., has h=0. Then, p_i stores in node $\pi=\pi'j$ what p_j sends to it for π' in the last round (i.e., round f+1). By definition, this is the resolved value by p_i for π.

- **Induction**: π is not a leaf, i.e., has height $h>0$;
  - By construction, π has at least n-f children, and since $n>3f$, this implies $n-f>2f$, i.e., it has a majority of non-faulty children (i.e., whose last digit of the label corresponds to a non-faulty processor)
  - Let $\pi k=\pi' jk$ be a child of π of height h-1 such that p_k is non-faulty.
  - Since p_j is non-faulty, it correctly reports a value v stored in its π' node; thus, p_k stores it in its $\pi=\pi' j$ node.
  - By induction, p_i's resolved value for πk equals the value v that p_k stored in its π node.
  - So, all of π's non-faulty children resolve to v in p_i's tree, and thus π resolves to v in p_i's tree.

END of PROOF

Inductive step by a picture



≥ n-f children, and since n>3f, i.e., n-f>2f, it follows there are more than 2f nodes here, and so there is a majority of (at least f+1) non-faulty nodes which will resolve to v by the inductive hypothesis

Corollary: all the non-faulty processors will resolve the very same value in $\pi=\pi' j$, namely v

Validity

- Suppose all inputs of (non-faulty) processors are v
- Non-faulty processor p_i decides resolve(λ), which is the majority among resolve(j), $0 \leq j \leq n-1$, based on p_i's tree.
- Since by Lemma 1 resolved values are consistent, if p_j is non-faulty, then p_i's resolved value for tree node labeled j, i.e., resolve(j), is equal to what p_i stores in the tree node labeled j, which in turn is equal to what p_j stores in its root, namely p_j's input value, i.e., v.
- Since there is a majority of non-faulty processors (indeed, $n>3f$, and so at level 1 there are more than $2f$ nodes associated with non-faulty processors), and their inputs are all equal to v, then p_i decides v.

Agreement: Common Nodes and Frontiers

Definition 1: A tree node π is **common** if all non-faulty processors compute the same value of `resolve`(π).

Notice that Lemma 1 told to us that all the nodes whose label ends with an index associated with a non-faulty processor are common. However it cannot be used to establish that the root is common, as we already pointed out, since the label of the root is the empty string.

$\Rightarrow$ To prove agreement, we have now to show that also the root is common; to do that we need to show that there exist other common nodes, besides those captured by Lemma 1.

Definition 2: A tree node π has a **common frontier** if every path from π to a descending leaf contains at least a common node.

Observation: If π is common, then it has a common frontier.

Lemma 2: If π has a common frontier, then π is common.

Proof: By induction on the height h of π:

- **Basis** (π is a leaf, i.e., h=0): then, since the only path from π to a leaf consists solely of π, the common node of such a path can only be π, and so π is common;
- **Induction** (π is not a leaf): By contradiction, assume π has height h>0 and has a common frontier but is not common; then:
  - Every child π' of π has a common frontier (this is not true, in general, if π would be common);
  - Since every child π' of π has height h-1 and has a common frontier, then by the inductive hypothesis, it is common;
  - Then, all non-faulty processors resolve the same value for every child π' of π, and thus all non-faulty processors resolve the same value for π, i.e., π is common (contradiction!).

END of PROOF

Agreement: the root has a common frontier

- There are $f+2$ nodes on any root-leaf path
- The label of each non-root node on a root-leaf path ends in a distinct processor index: $i_1,i_2,\ldots,i_{f+1}$
- Since there are at most f faulty processors, at least one of such nodes has a label ending with a non-faulty processor index
- This node, say $i_1{:}i_2{:},\ldots,i_{k-1}{:}i_k$, by Lemma 1 is **common** (more precisely, in all the trees associated with non-faulty processors, the resolved value in $i_1{:}i_2{:},\ldots,i_{k-1}{:}i_k$ equals the value stored by the non-faulty processor p_{i_k} in node $i_1{:}i_2{:},\ldots,{:}i_{k-1}$)

$\Rightarrow$ Thus, the root has a common frontier, since on any root-leaf path there is at least a common node, and so the root is common (by previous lemma)

$\Rightarrow$ Therefore, agreement is guaranteed!

Complexity

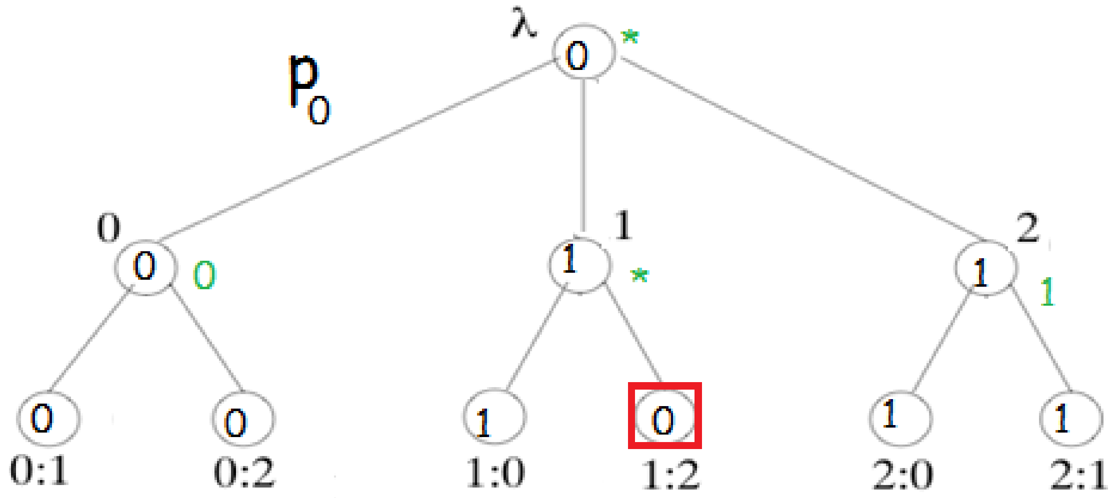
- Exponential tree algorithm uses $f+1$ rounds, and $n=3f+1$ processors are enough to guarantee correctness (see Lemma 1)
- Exponential number of messages:
  - In round 1, each (non-faulty) processor sends n messages $\Rightarrow O(n^2)$ total messages
  - In round $2 \leq d \leq f+1$, each of the $O(n)$ (non-faulty) processors broadcasts to all (i.e., n processors) the level $d-1$ of its local tree, which contains $n(n-1)(n-2)\ldots(n-(d-2))$ nodes $\Rightarrow$ this means, for round d, a total of

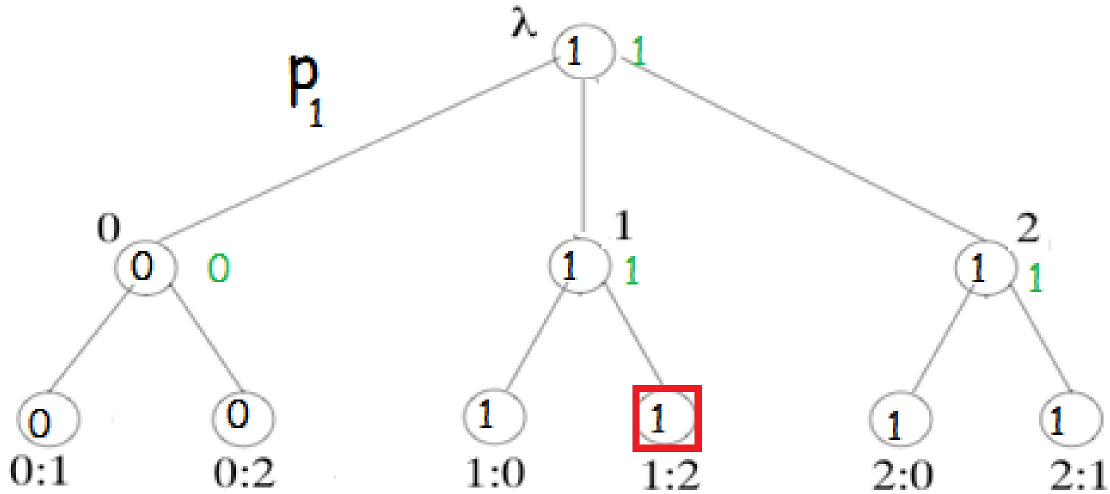
    $O(n \cdot n \cdot n(n-1)(n-2)\ldots(n-(d-2)))=O(n^{d+1})$ messages
  - This means a total of $O(n^2)+O(n^3)+\ldots+O(n^{f+2})=O(n^{f+2})$ messages, and since $f=O(n)$, this number is exponential in n if f is more than a constant relative to n

Homework

Show an execution with $n=3$ processors and $f=1$ for which the exp-tree algorithm fails.

Sketch of solution: p_2 byzantine (green values are the resolved ones)





Randomized Byzantine Consensus

- This algorithm uses
  - $n>8f$ processors (sub-optimal)
  - $O(\log n)$ rounds (w.h.p., this is notable, since for $f=\Theta(n)$ it means breaking the lower bound barrier of $f+1$ rounds)
  - $O(n^2 \log n)$ number of messages (w.h.p., remind that the King algorithm was using $O(n^3)$ msgs)

There is a **trustworthy** processor q which at every round tosses a random coin and informs every other processor

Coin = heads (probability 1/2)

Coin = tails (probability 1/2)

Each processor p_i has a preferred value v_i

In the beginning,
the preferred value is set to the initial value

Assume that initial value is binary

$$v_i \in \{0,1\}$$

The algorithm tolerates $f < \frac{n}{8}$
Byzantine processors

There are three threshold values:

$$L = \frac{5n}{8} \qquad H = \frac{6n}{8} \qquad G = \frac{7n}{8}$$

In each round, processor p_i executes:

Broadcast v_i;

Receive values from all processors;

$maj_i \leftarrow$ majority value;

$tally_i \leftarrow$ occurrences of maj;

Receive coin from the trustworthy processor;

If coin=head then $threshold \leftarrow L = \frac{5n}{8}$

else $threshold \leftarrow H = \frac{6n}{8}$

If $tally_i \geq theshold$ then $v_i \leftarrow maj_i$

else $v_i \leftarrow 0$

If $tally_i > G = \frac{7n}{8}$ then decision is reached

Analysis: Examine cases in a round

Termination: There is a processor p_i with $tally_i > G = \frac{7n}{8}$

Other cases:

Case 1: Two processors p_i and p_k have different $maj_i \neq maj_k$

Case 2: All processors have same maj_i

Termination: There is a processor p_i with $tally_i > G = \frac{7n}{8}$

Since faulty processors are at most $f < \frac{n}{8}$

processor p_i received at least

$$tally_i - f > \frac{6n}{8}$$

votes for maj_i from good processors

Therefore, every good processor p_k

will have $maj_i = maj_k$

with $tally_k > H = \frac{6n}{8}$

Consequently, at the end of the round **all the good processors** will have the same preferred value:

$$v_k = maj_k = maj_i$$

Observation:

If at the beginning of a round all the good processors (remind they are at least $\frac{7n}{8}$) have the same preferred value, then the algorithm terminates (and solves correctly the consensus problem) in that round

This holds since for every processor p_i the termination condition $tally_i > G = \frac{7n}{8}$ will be true in that round

Notice that this observation implies **validity**

Therefore, if the termination condition is true for one processor at a round, then, the termination condition will be true for **all processors** at next round.

Case 1: Two processors p_i and p_k have different $maj_i \neq maj_k$

We now show that it has to be that

$$tally_i < L = \frac{5n}{8}$$

and

$$tally_k < L = \frac{5n}{8}$$

And therefore $v_i = v_k = 0$

Thus, every processor chooses 0, and the algorithm terminates correctly in next round

Proof: Suppose (for sake of contradiction) that

$$tally_i \geq L = \frac{5n}{8}$$

Then at least

$$tally_i - f > \frac{4n}{8} = \frac{n}{2}$$

good processors have voted maj_i

Consequently, we would have $maj_i = maj_k$

Contradiction!

Case 2: All processors have same maj_i

Then for any two processors p_i and p_k it holds that $|tally_i - tally_k| \leq f$

Since otherwise, the number of faulty processors would exceed f

Let $p_{\min}$ be the processor with

$$tally_{\min} = \min_i \{tally_i\}$$

We have 4 possible subcases:

2.1 $tally_{min} < L = \frac{5n}{8}$ and $threshold = H = \frac{6n}{8}$ good

2.2 $tally_{min} \geq L = \frac{5n}{8}$ and $threshold = L = \frac{5n}{8}$ good

2.3 $tally_{min} < L = \frac{5n}{8}$ and $threshold = L = \frac{5n}{8}$ bad

2.4 $tally_{min} \geq L = \frac{5n}{8}$ and $threshold = H = \frac{6n}{8}$ bad

We do not know the exact probability each of the 4 possible subcases will occur, but good and bad cases will occur with probability 1/2

Sub-case 2.1: $tally_{min} < L = \frac{5n}{8}$

and $threshold = H = \frac{6n}{8}$

then, for any processor p_k it holds

$$tally_k \leq tally_{min} + f < L + f \leq \frac{6n}{8} = H$$

And therefore $v_i = v_k = 0$

Thus, every processor chooses 0,
and the algorithm terminates in next round

Sub-case 2.2: $$tally_{min} \geq L = \frac{5n}{8}$$

and $$threshold = L = \frac{5n}{8}$$

then, for any processor p_k it holds

$$tally_k \geq tally_{min} \geq L$$

And therefore $v_k = v_{min} = maj_{min}$

Thus, every processor chooses v_{min},
and the algorithm terminates in next round

- In other words, subcases 2.1 and 2.2 will make the algorithm terminate in the next round, while the remaining two subcases will be bad (i.e., the algorithm will not stop in next round)
- From the above analysis, it follows that the algorithm will terminate w.h.p. within $O(\log n)$ rounds, since at each round **it will terminate in the next round** with probability at least ½ (remember we are in Case 2, which is one out of the three cases we analyzed); thus, the probability **it will not terminate** within $\log n$ rounds will be at most $(1/2)^{\log n}=1/n$, and so the probability **it will terminate** within $\log n$ rounds will be at least $1-1/n$
- Concerning the message complexity, in each round circulate $O(n^2)$ messages, and so w.h.p. the total number of messages will be $O(n^2 \log n)$

Homework

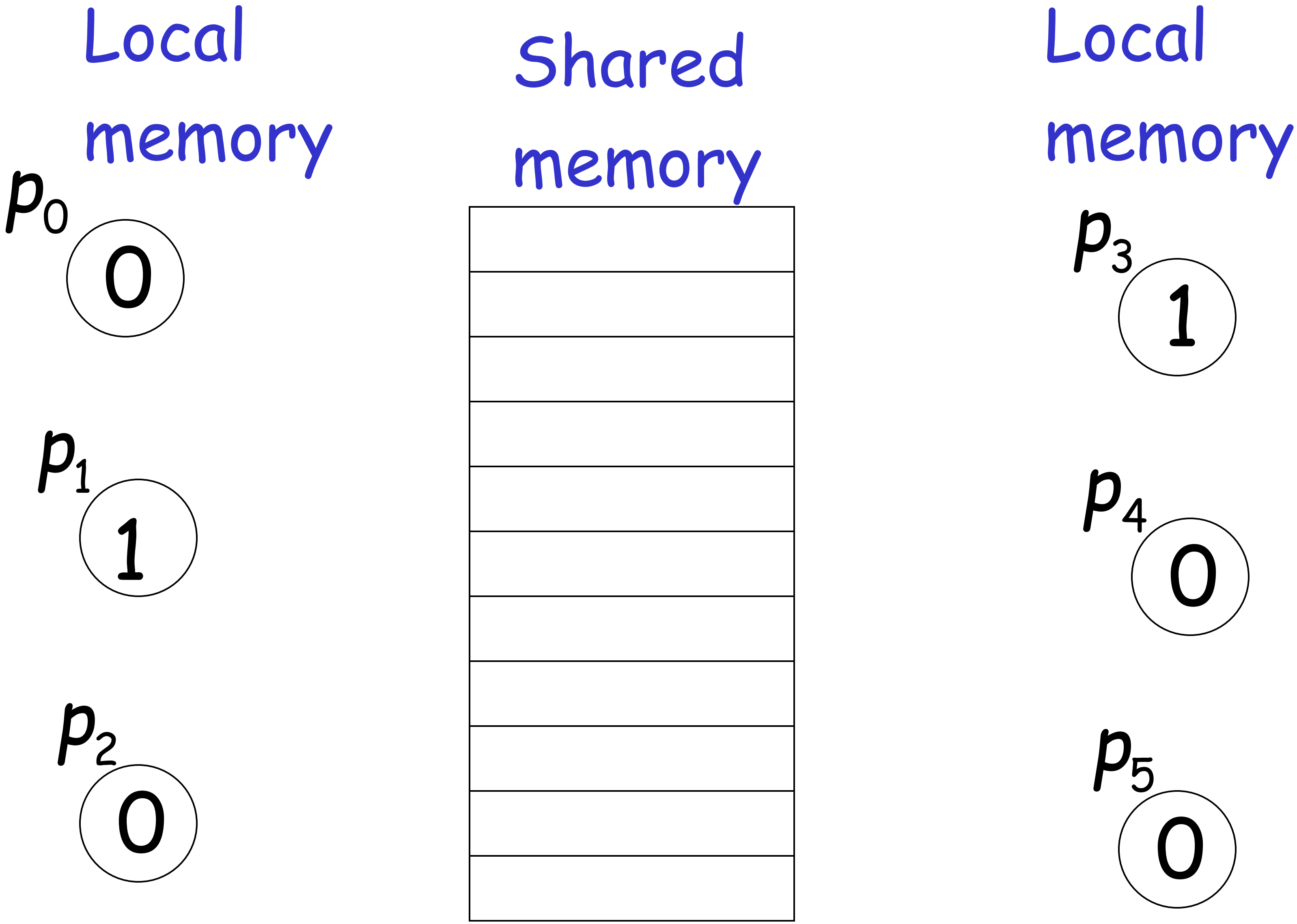
- Show an execution with n=9 processors and f=1 for which the randomized algorithm does not converge.

Consensus in the Shared Memory Model

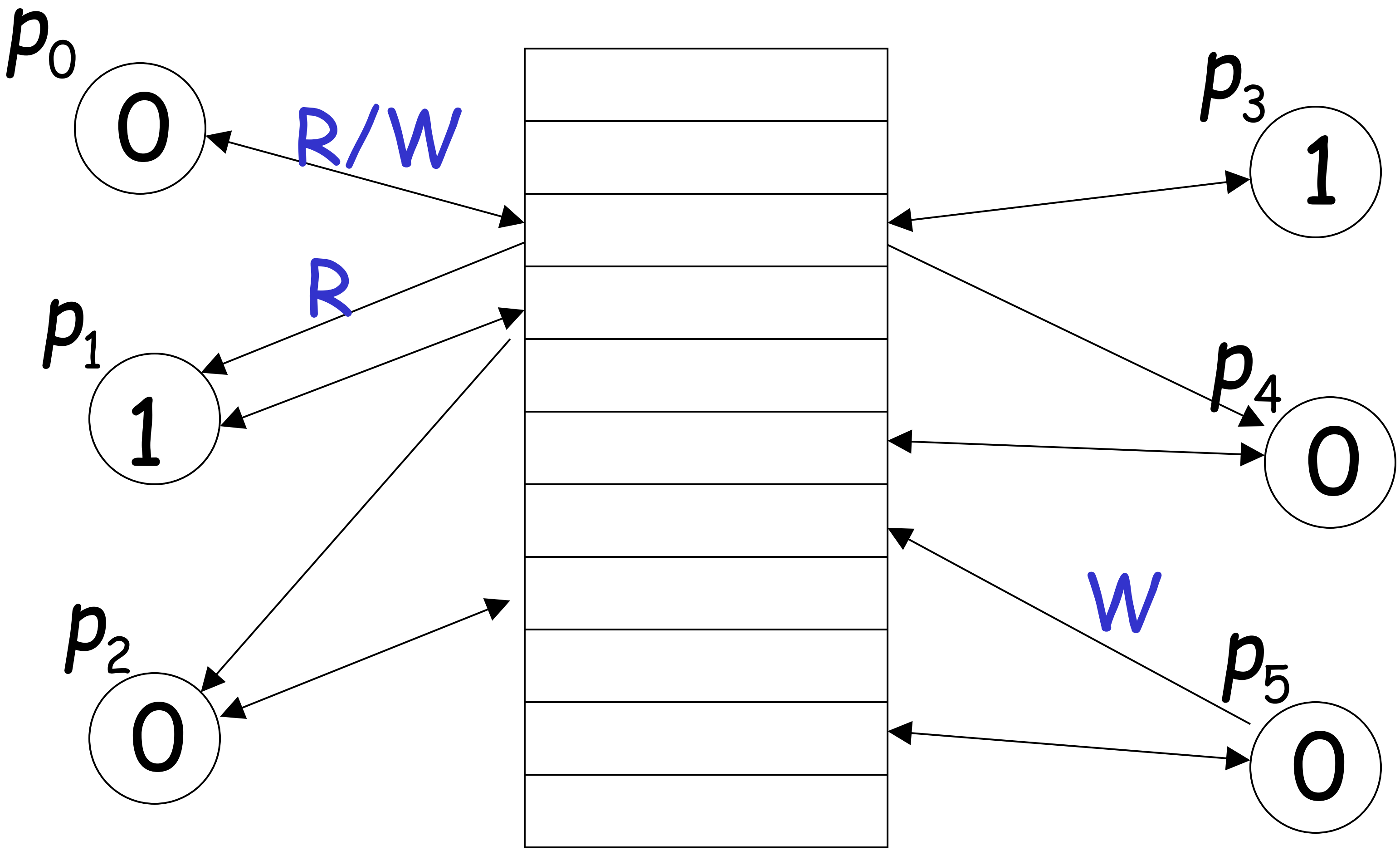
Consider n processors in shared memory:

$$p_0, \ldots, p_{n-1}$$

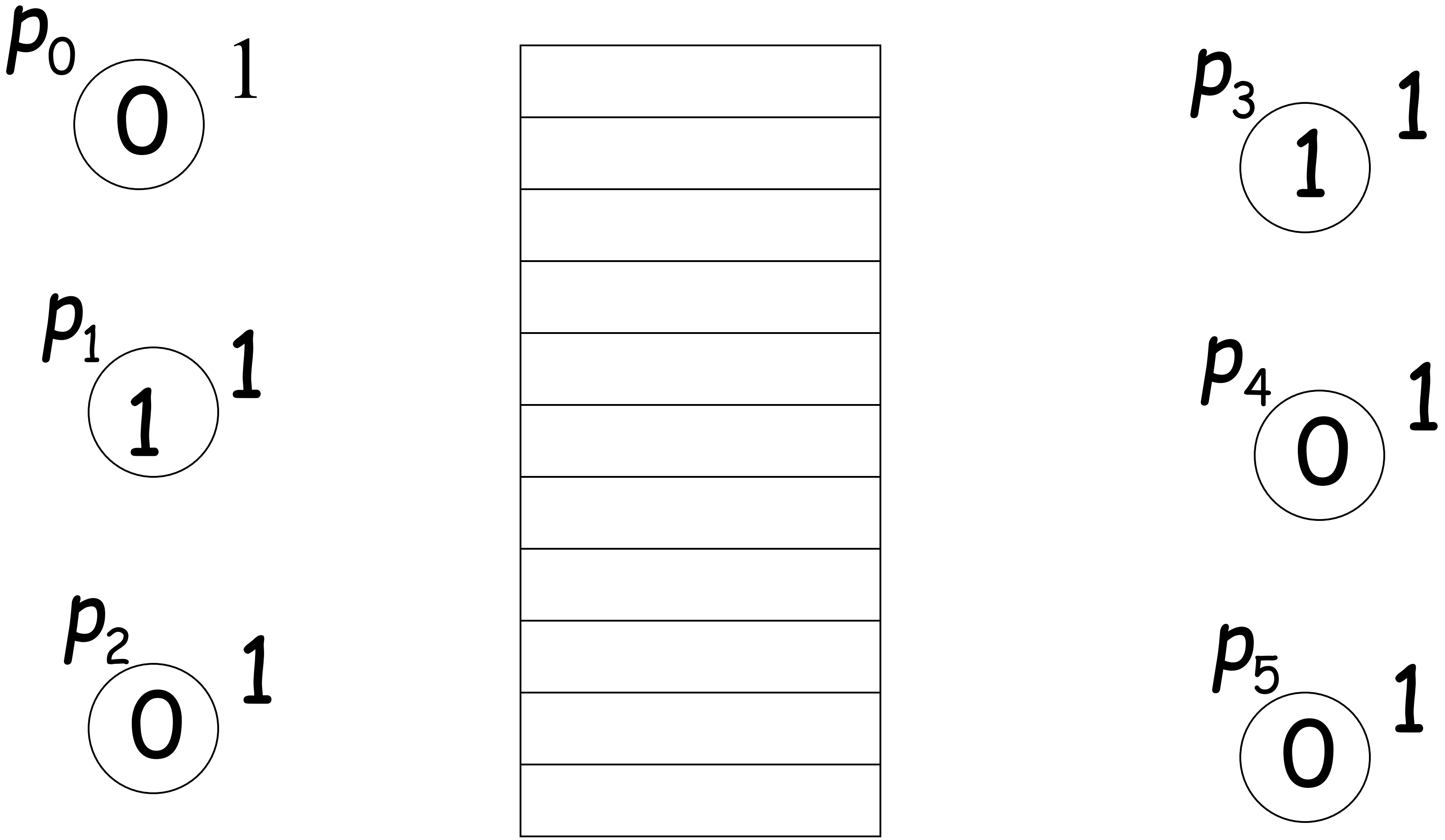
which try to solve the consensus problem, but they can **crash**



Every processor starts with an initial value stored in local memory (w.l.o.g., 0 or 1)



communication through shared memory



At the end of execution, every processor has to decide the same value (0 or 1, **agreement**), and if every processor starts with the same value, then every processor should decide that value (**validity condition**)

Wait-freedom in asynchronous systems:

A processor should be able to finish execution of an algorithm even if all other processors fail

Wait-freedom captures:

- Asynchronous executions
- Crash failures

Consensus Number

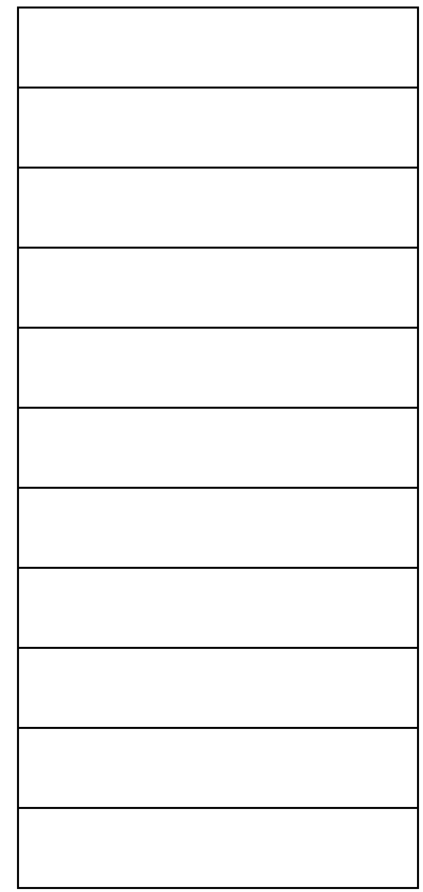
Consensus Number of a shared-variable type:

The **maximum** number of processors for which a shared-variable type can be used to solve the wait-free consensus problem

Shared-variable Type	Consensus Number
Read/Write	1
Test&Set	2
Compare&Swap	∞ (infinity)

Read/Write

Shared Memory



Suppose that the shared memory can only be accessed through Read or Write operations

Theorem: The consensus number of the Read/Write shared-variable type is 1

Proof of Theorem:

Trivially, a system with only 1 processor using read/write (shared) variables enjoys wait-free consensus.

It remains to show:

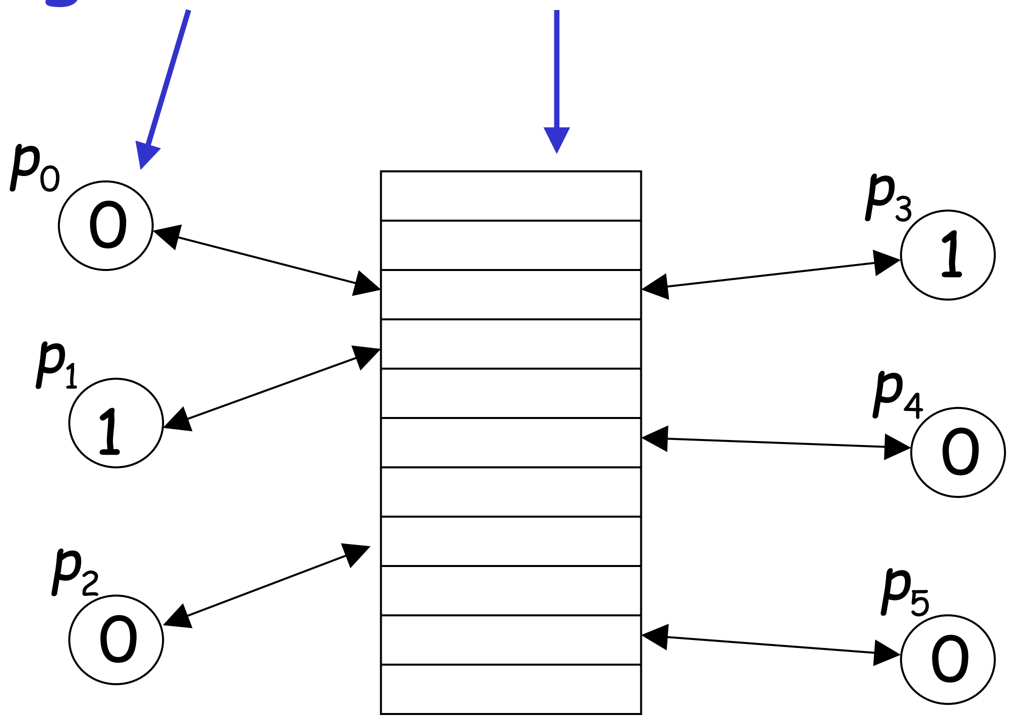
Wait-free consensus cannot be solved using only read/write shared variables for $n \geq 2$ processors

Approach:

We will show that any algorithm that solves wait-free consensus for $n \geq 2$ has an execution that never terminates

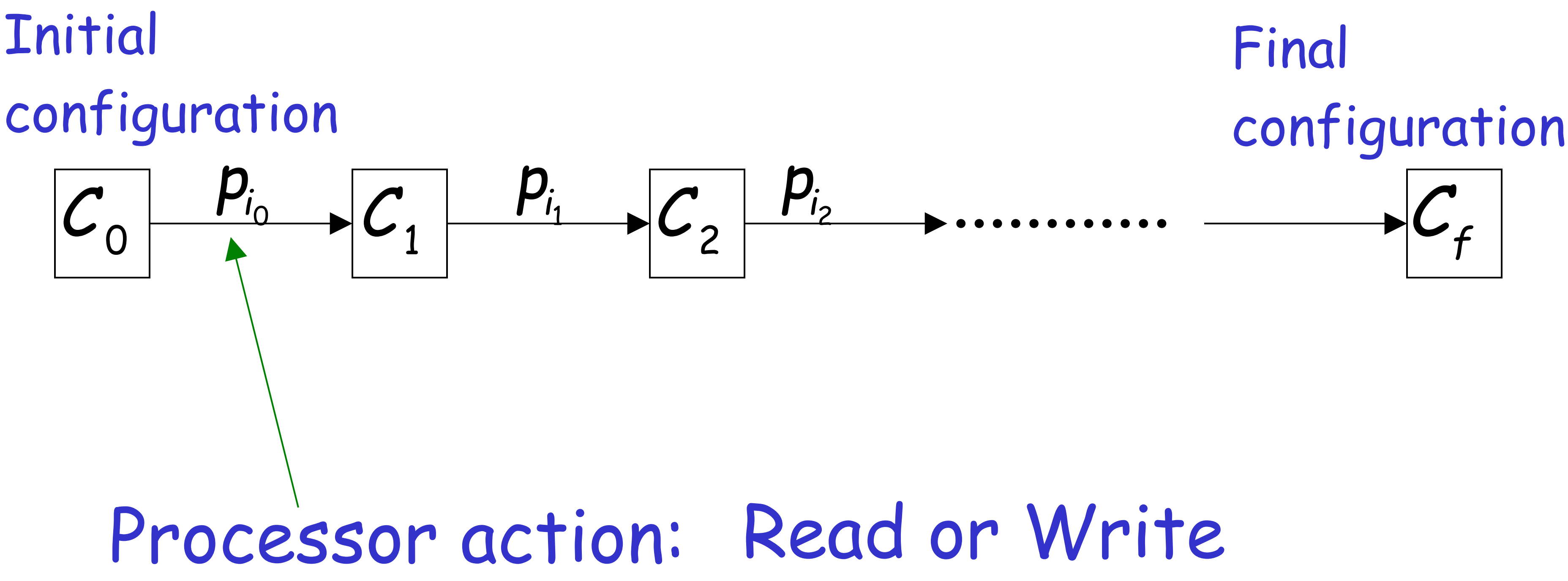
System configuration: C

Is the set of all variables in the system, including local and shared

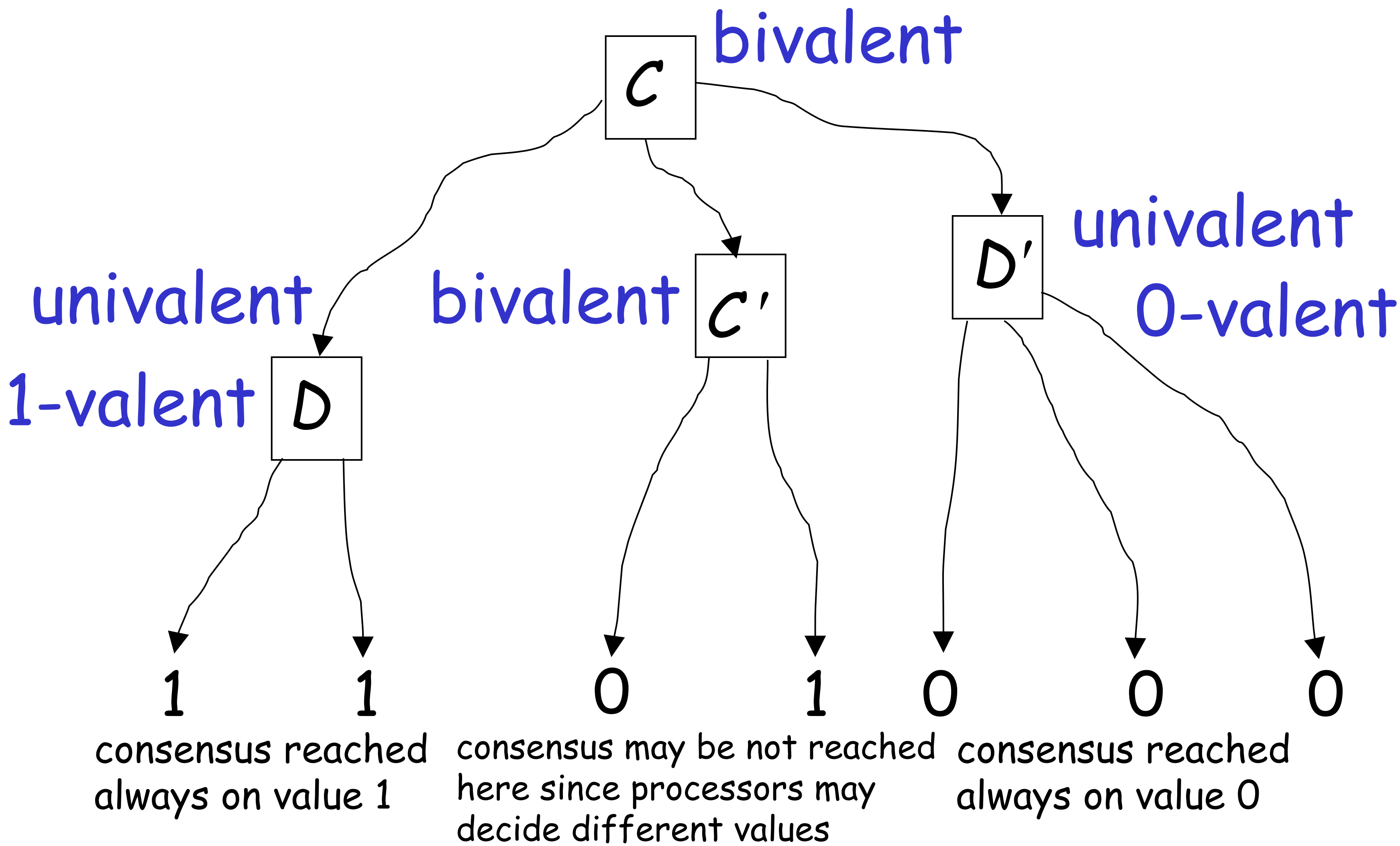


A distributed system execution can be always be viewed as a:

sequence of configurations

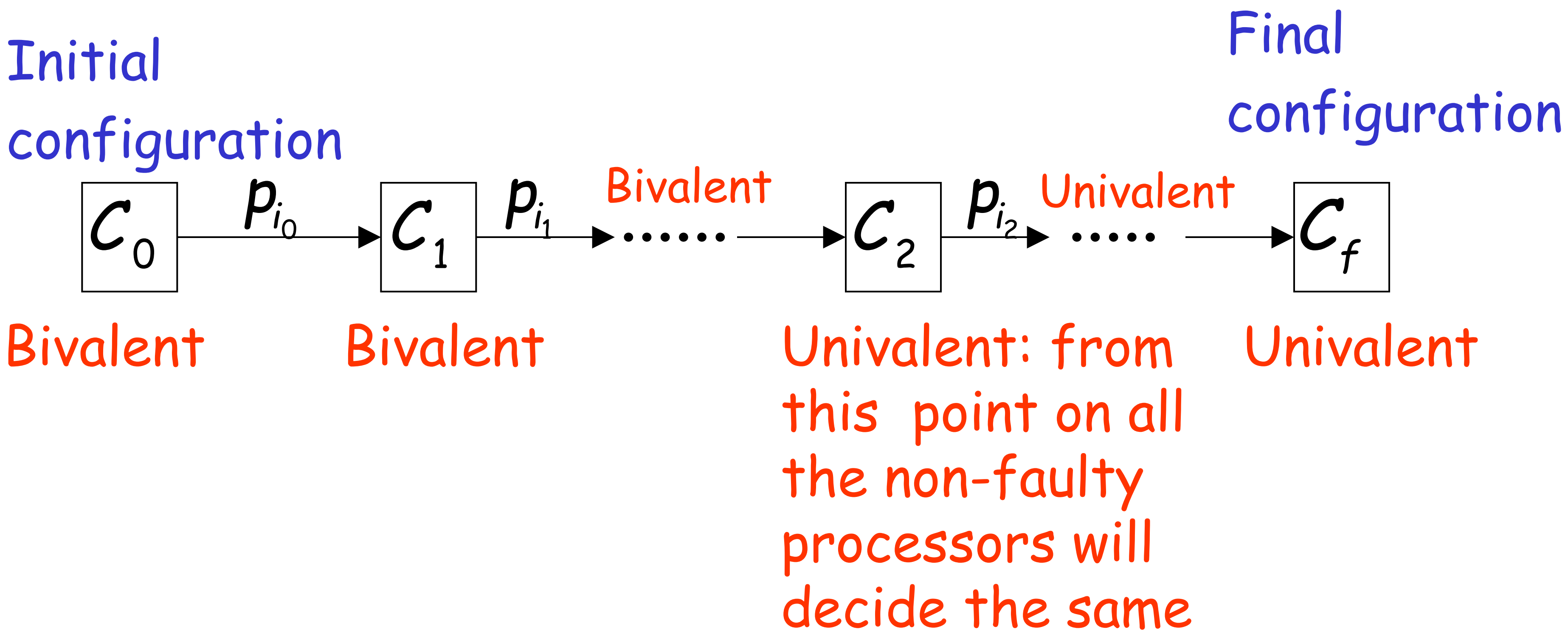


Valence of a system configuration C: set of set of all values decided by a nonfaulty processor in some configuration reachable from C by an admissible execution.

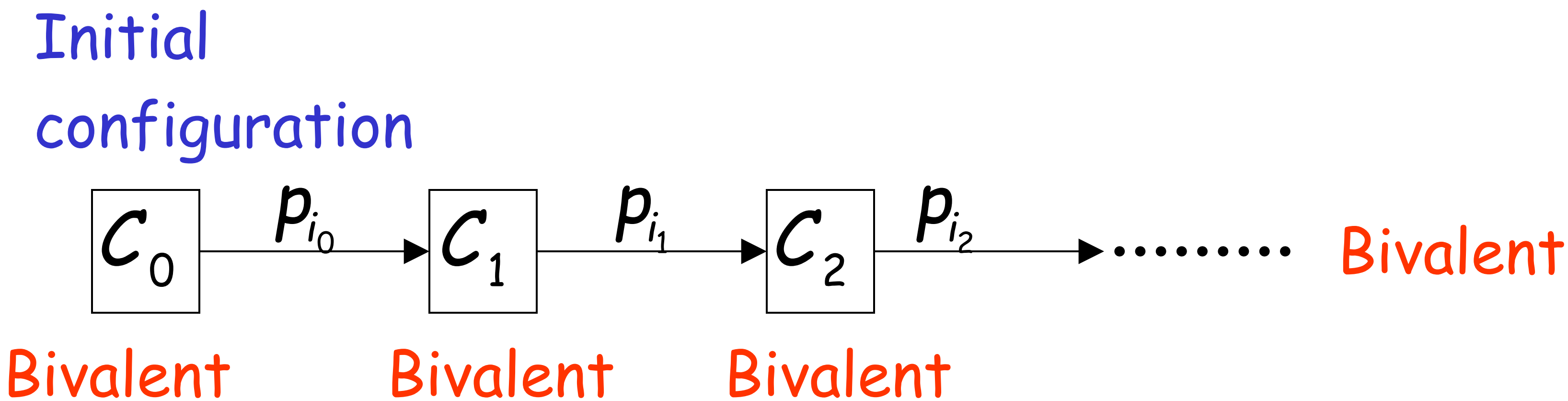


Output value at possible execution paths

A terminating execution:



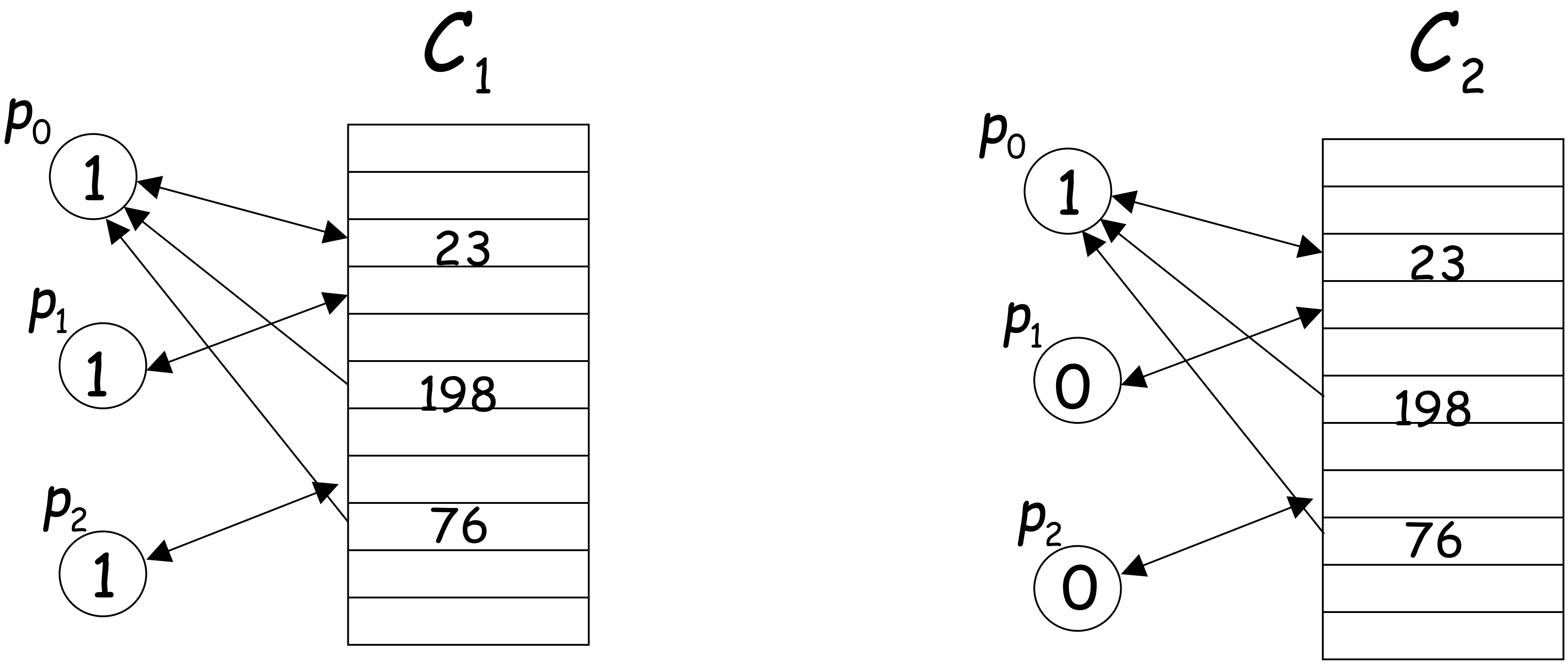
To prove the theorem, we will show that there is always an execution where every configuration is bivalent



Never-ending execution

Similar configurations for processor p_0

$$C_1 \overset{p_0}{\approx} C_2$$



Same shared variables

Local variables of others may differ

Lemma: If there exist univalent configurations C_1 and C_2 such that $C_1 \overset{p_i}{\approx} C_2$

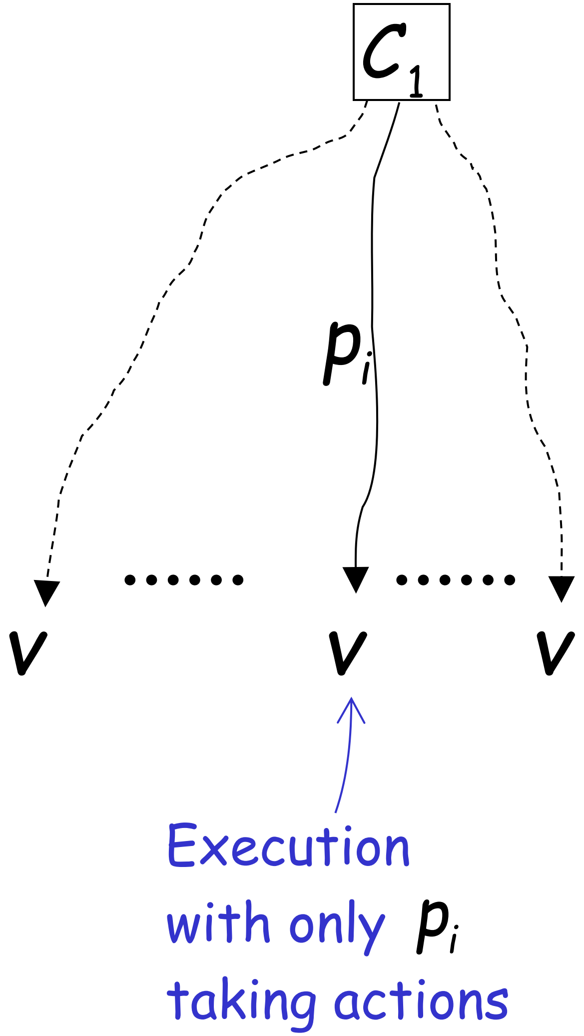
then if C_1 is v-valent

then C_2 is v-valent too

$(v = 0 \text{ or } 1)$

Proof of Lemma:

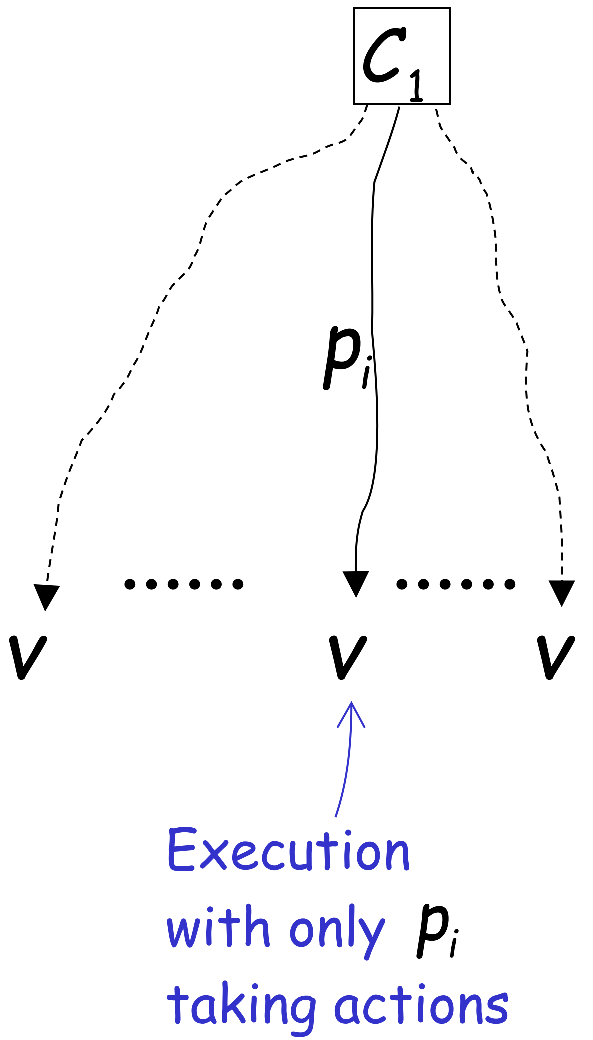
Univalent



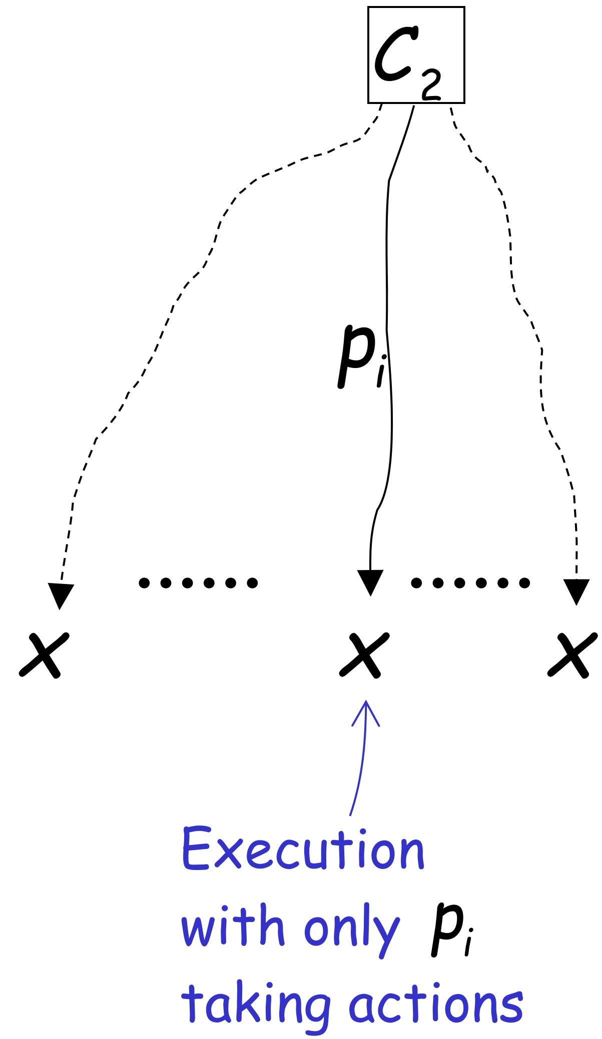
All possible executions from c_1

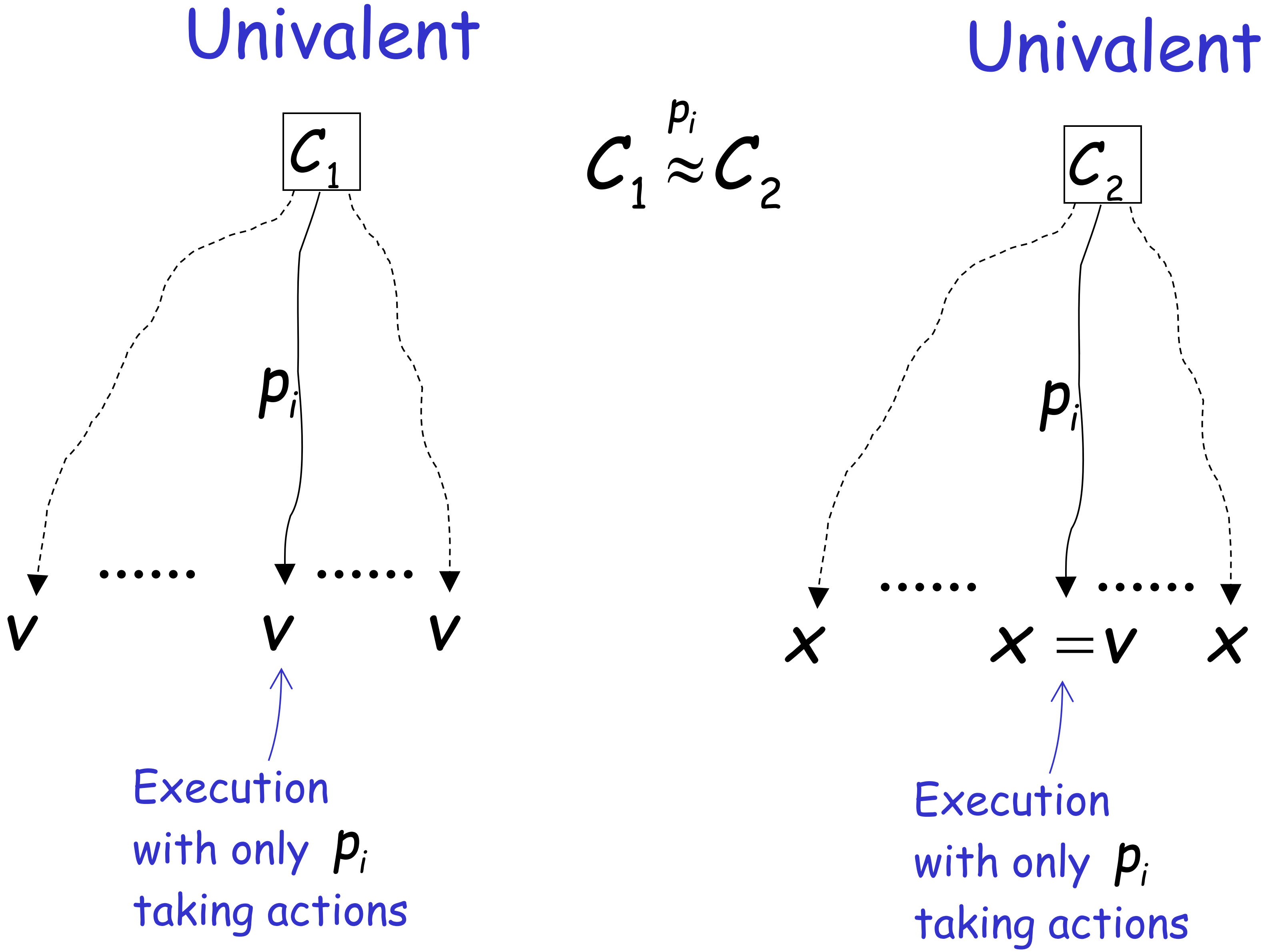
final decision for each Possible execution

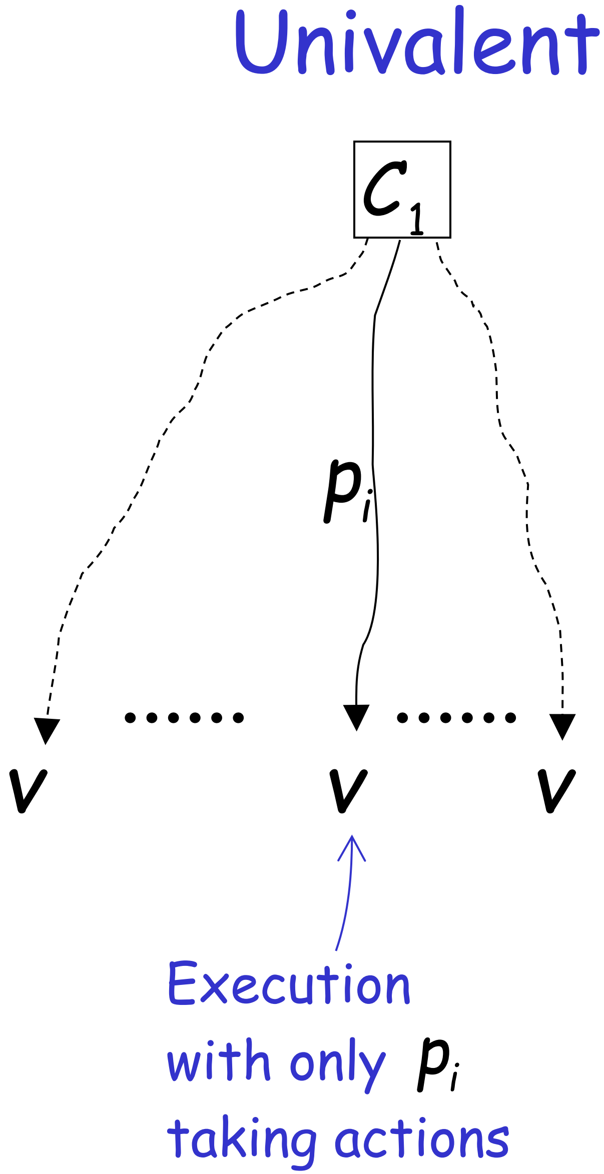
Univalent



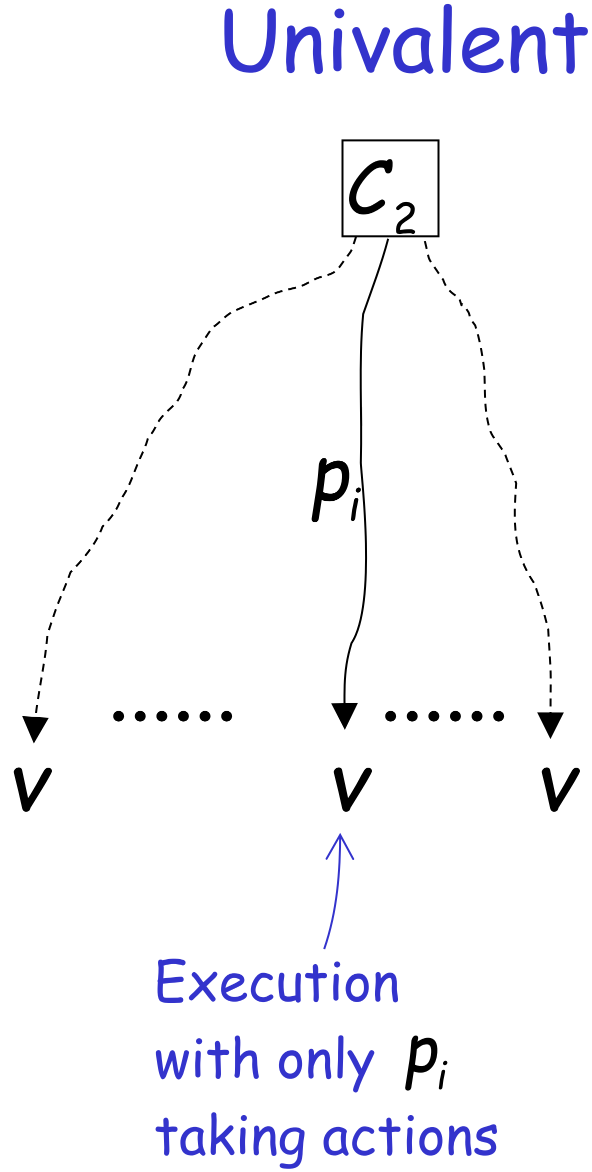
Univalent




Univalent
Univalent
$C_1 \overset{p_i}{\approx} C_2$
C_1
p_i
v ...... v v
Execution with only p_i taking actions
C_2
p_i
x ...... $x = v$ x
Execution with only p_i taking actions



$$C_1 \overset{p_i}{\approx} C_2$$



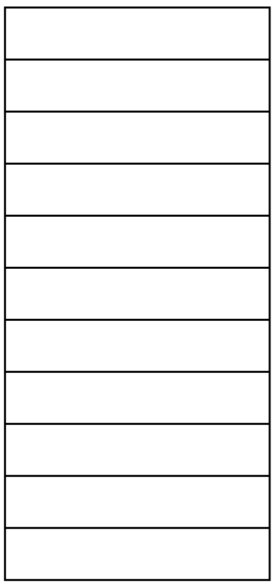
End of Lemma Proof

Lemma: There exists a bivalent initial configuration

Proof of Lemma:

Possible Initial Configurations

Shared Memory



Empty

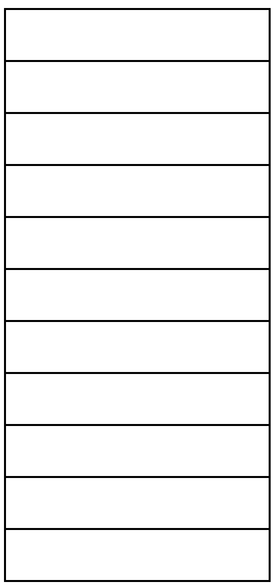
Local Memory

Initial Configuration

	I_0	I_{01}	I_1
p_0	0	0	1
p_1	0	1	1
	$\vdots$	$\vdots$	$\vdots$
p_{n-1}	0	1	1

Possible Initial Configurations

Shared Memory



Empty

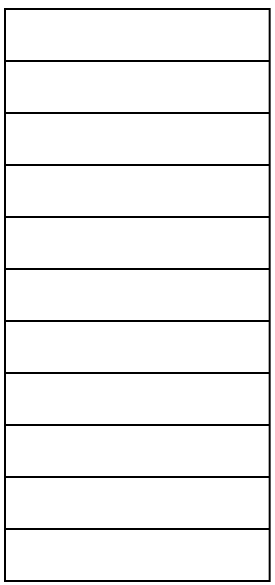
Local Memory

Initial Configuration

	I_0	I_{01}	I_1
p_0	0	0	1
p_1	0	1	1
	⋮	⋮	⋮
p_{n-1}	0	1	1
	0-valent	?	1-valent

Possible Initial Configurations

Shared Memory



Empty

Local Memory

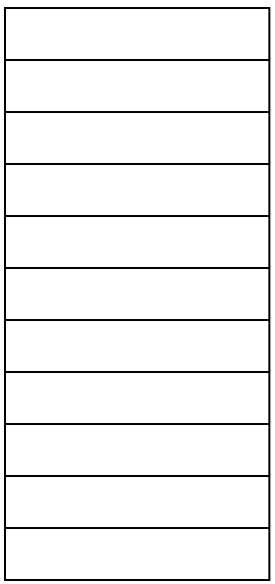
Initial Configuration

	I_0	I_{01}	I_1
p_0	0	0	1
p_1	0	1	1
⋮	⋮	⋮	⋮
p_{n-1}	0	1	1
	0-valent	1-valent?	1-valent

No, because $I_0 \overset{p_0}{\approx} I_{01}$

Possible Initial Configurations

Shared Memory



Empty

Local Memory

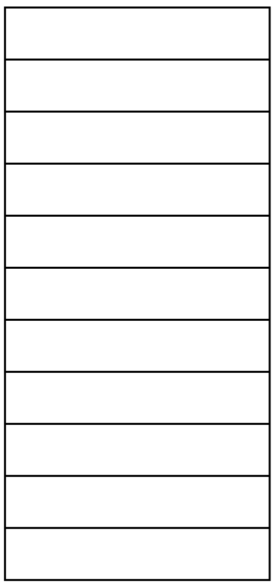
Initial Configuration

	I_0	I_{01}	I_1
p_0	0	0	1
p_1	0	1	1
⋮	⋮	⋮	⋮
p_{n-1}	0	1	1
	0-valent	0-valent?	1-valent

No, because $I_{01} \overset{p_1}{\approx} I_1$

Possible Initial Configurations

Shared Memory



Empty

Local Memory

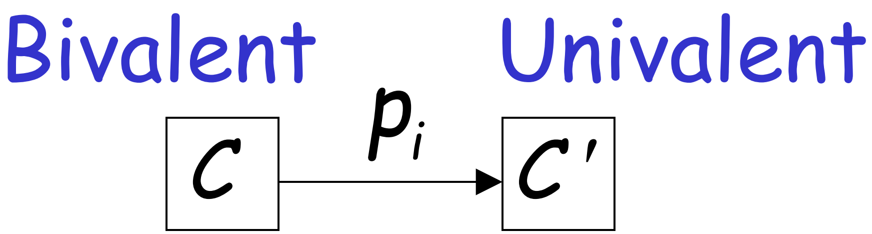
Initial Configuration

	I_0	I_{01}	I_1
p_0	0	0	1
p_1	0	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
p_{n-1}	0	1	1
	0-valent	bivalent	1-valent

End of Lemma Proof

Critical processor for a configuration:

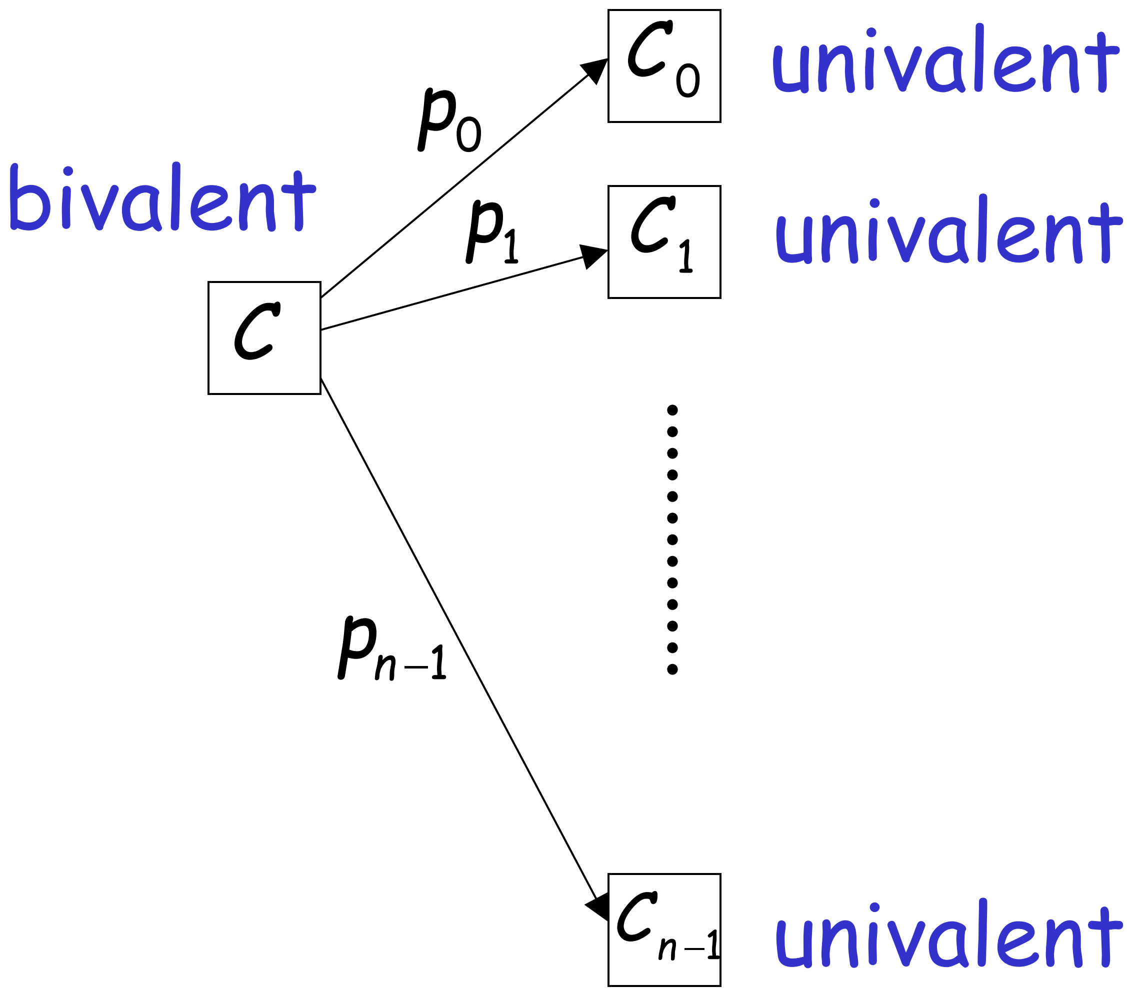
the configuration is bivalent,
and after the processor takes step
the configuration becomes univalent



Lemma: If C is a bivalent configuration then, there is at least one processor which is not critical

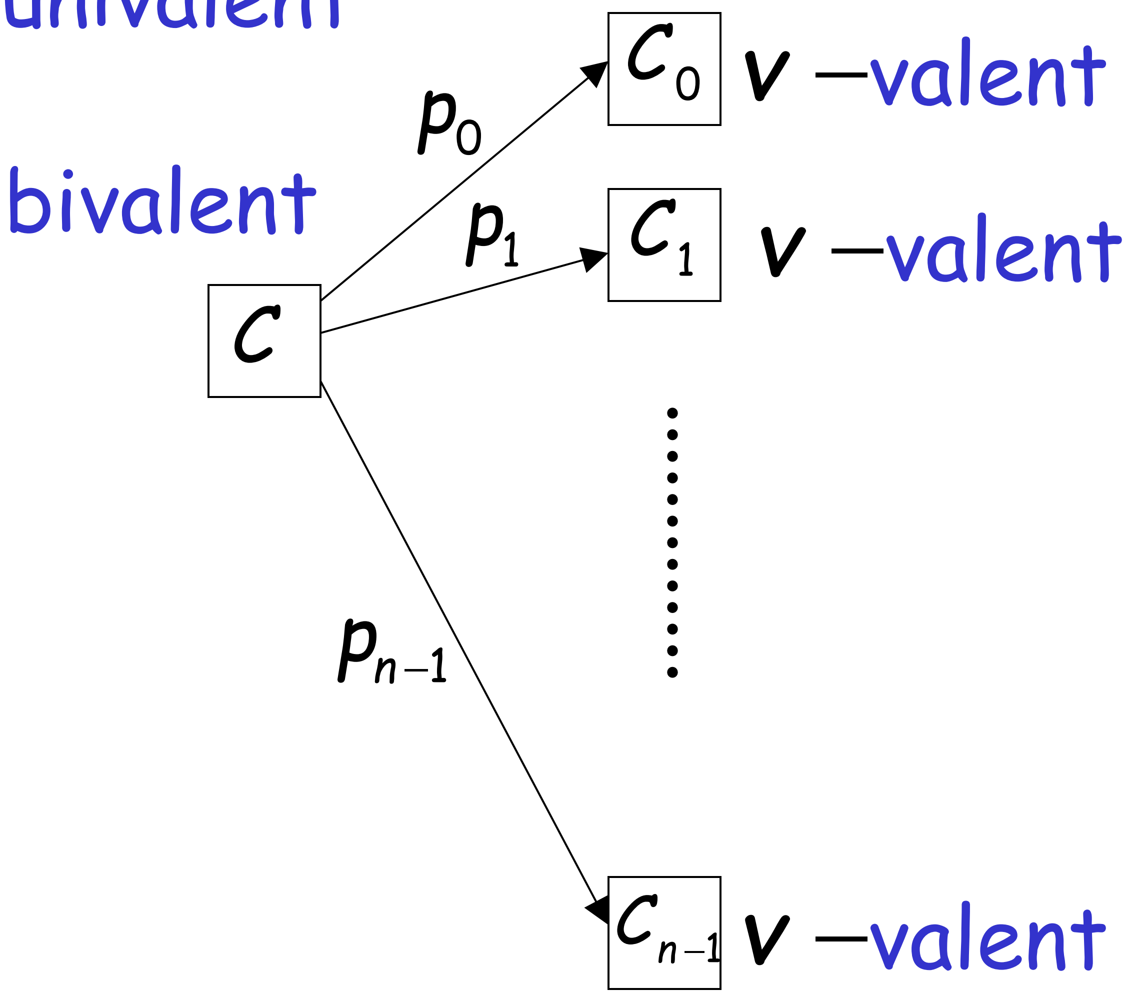
Proof of Lemma:

Assume for contradiction that all processors are critical

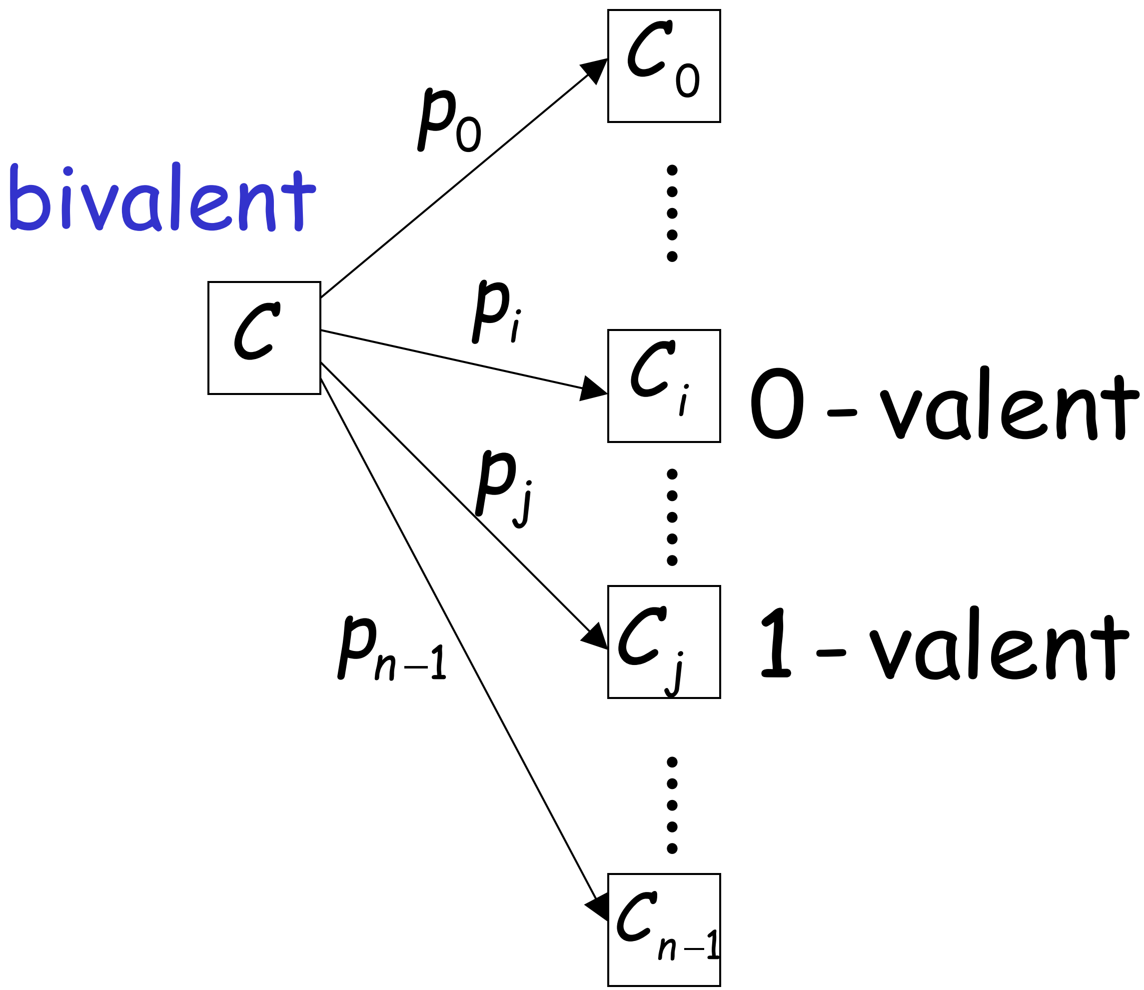


Possible executions

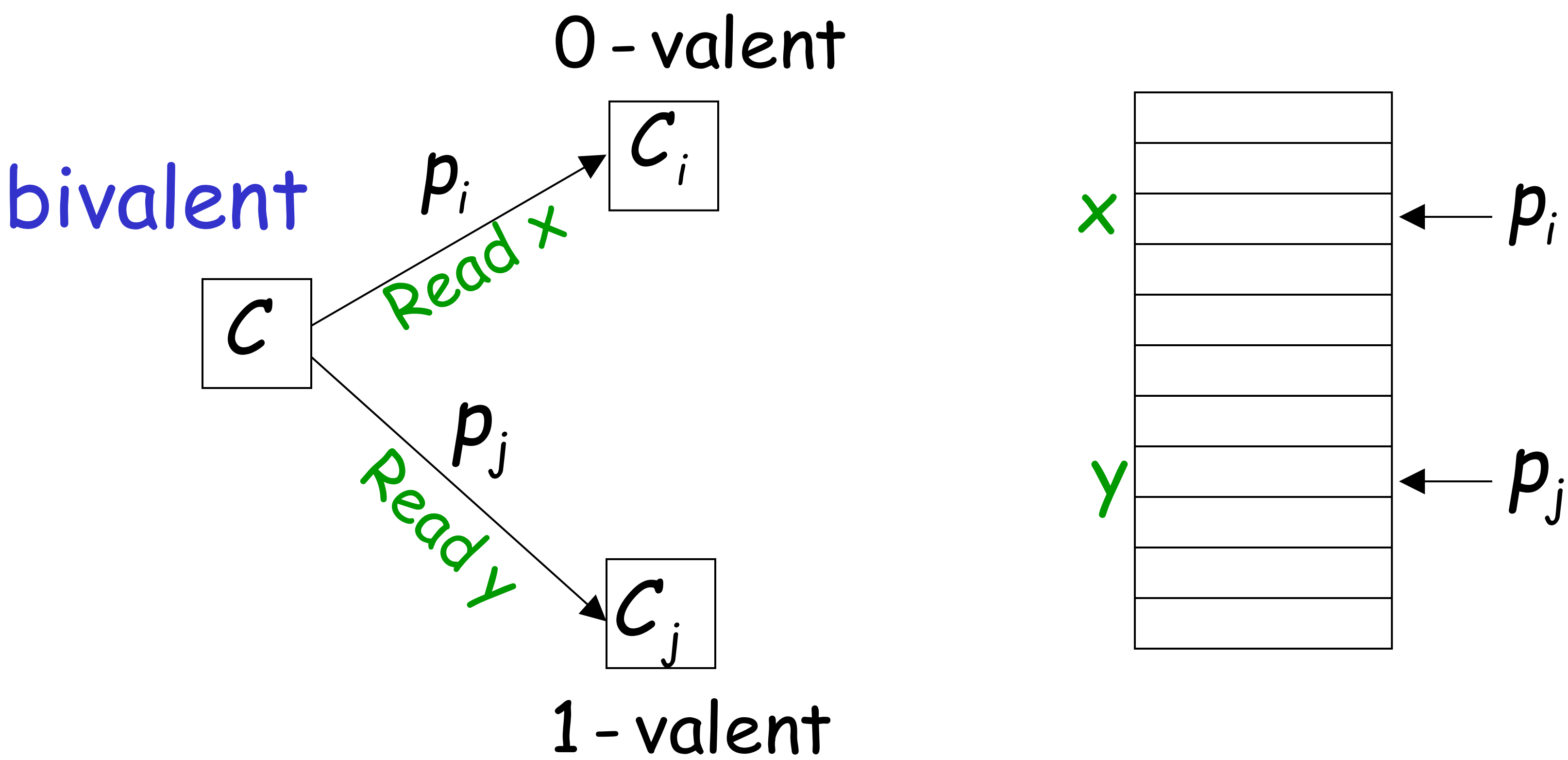
It cannot be that all have the same valence ($v = 0$ or 1) otherwise C would be univalent



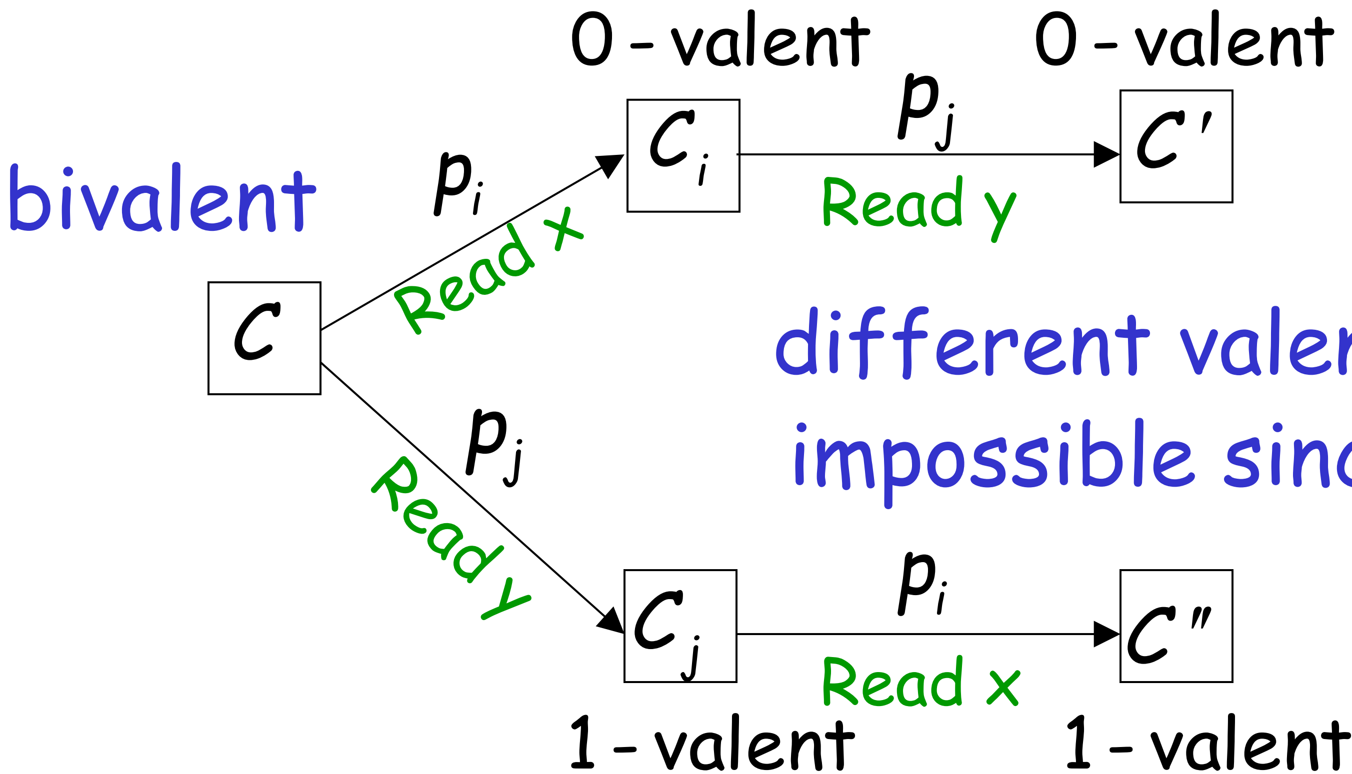
There must exist two processors with different valences



Case 1: suppose that they access different shared variables



two possible executions



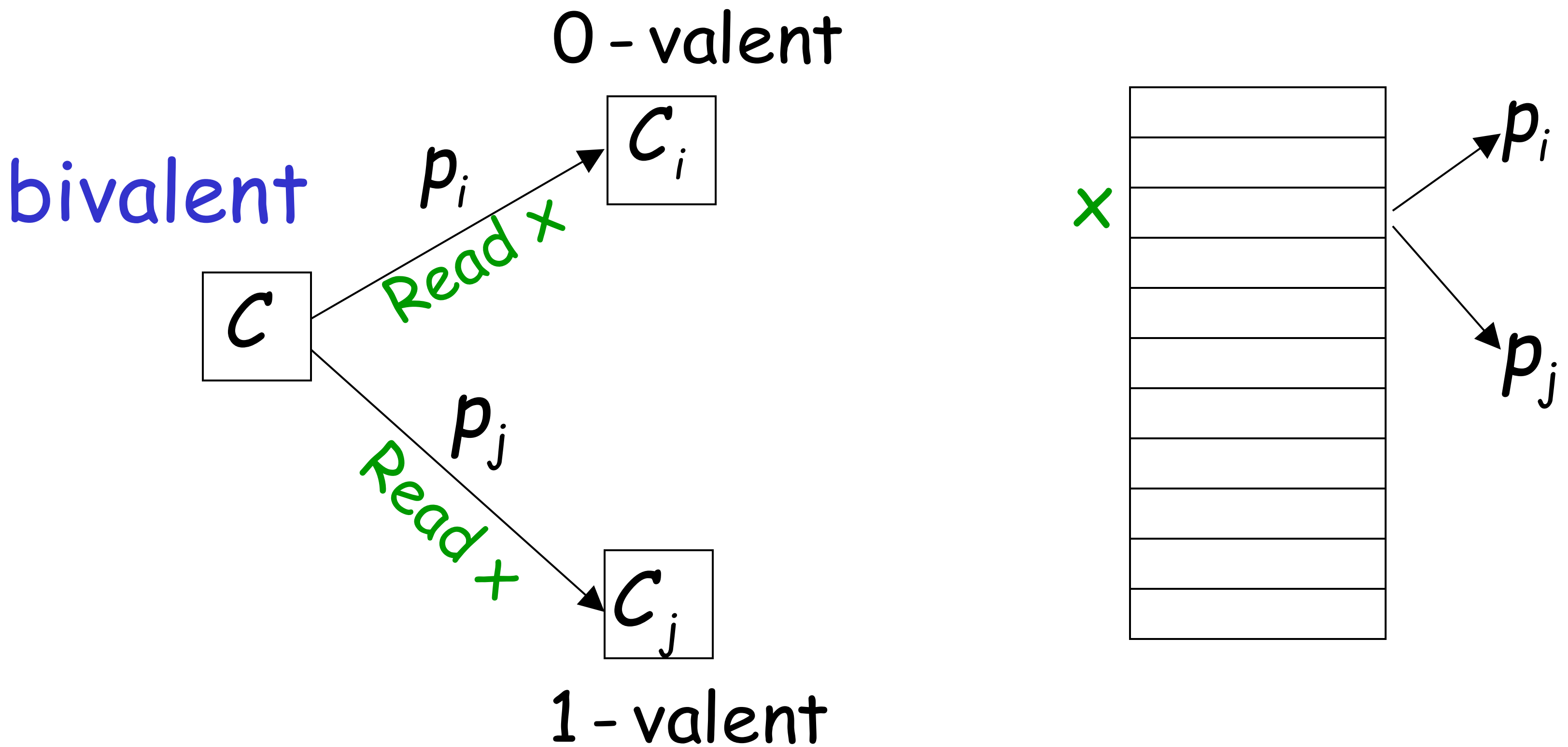
different valence

impossible since $C' = C''$

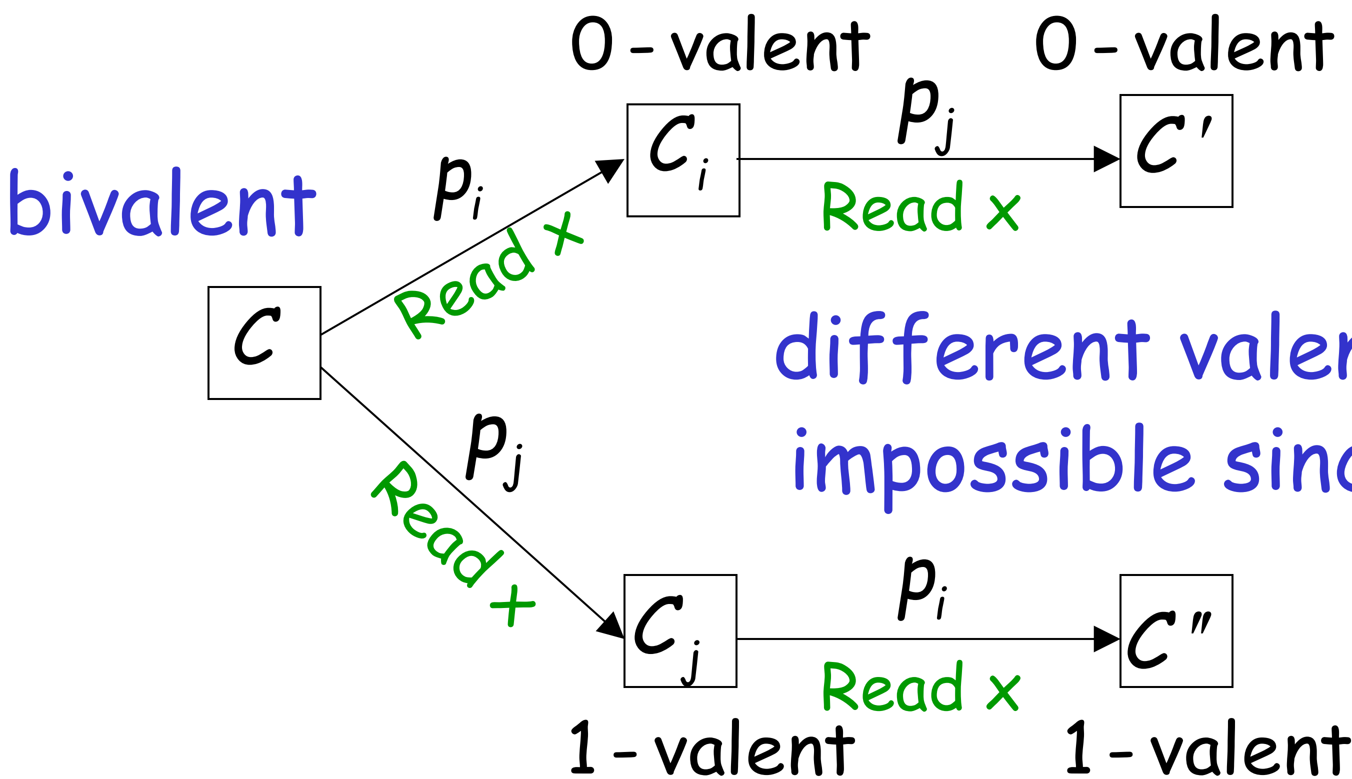
same result holds for any kind
of operation (Read or Write)
that the processors apply to x and y

Case 2: suppose that they access the same shared variable

subcase: read/read



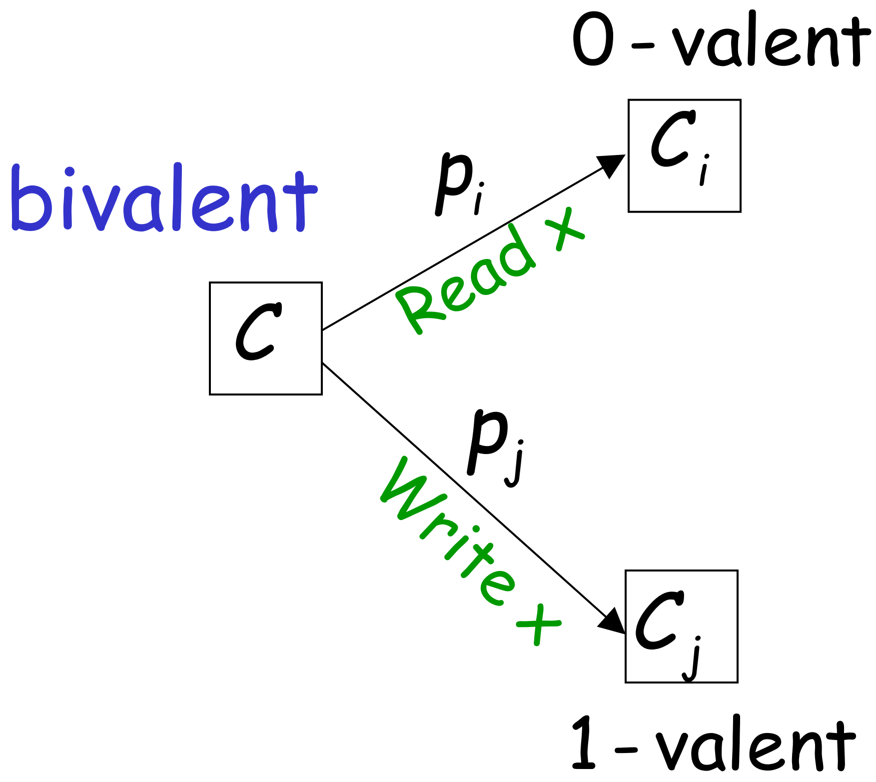
two possible executions



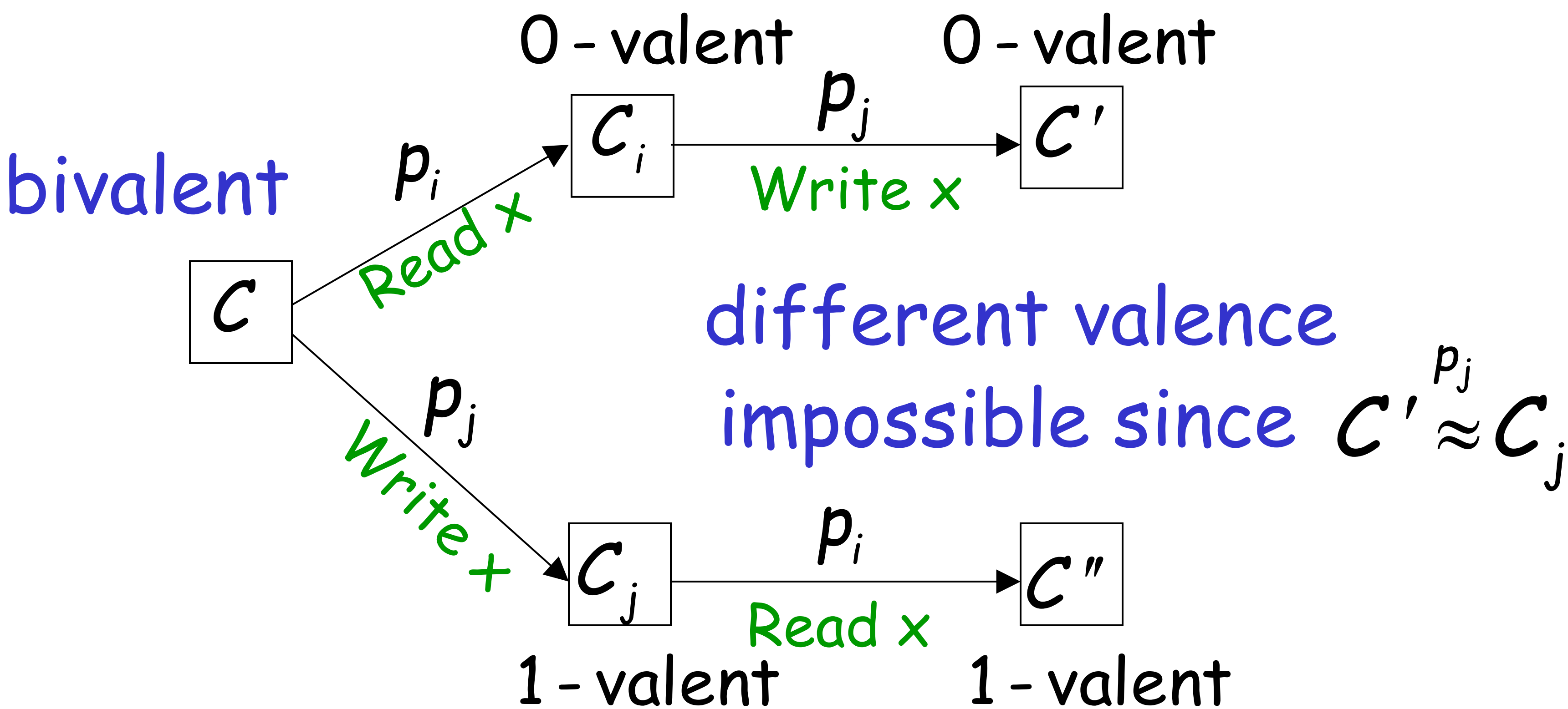
different valence

impossible since $C' = C''$

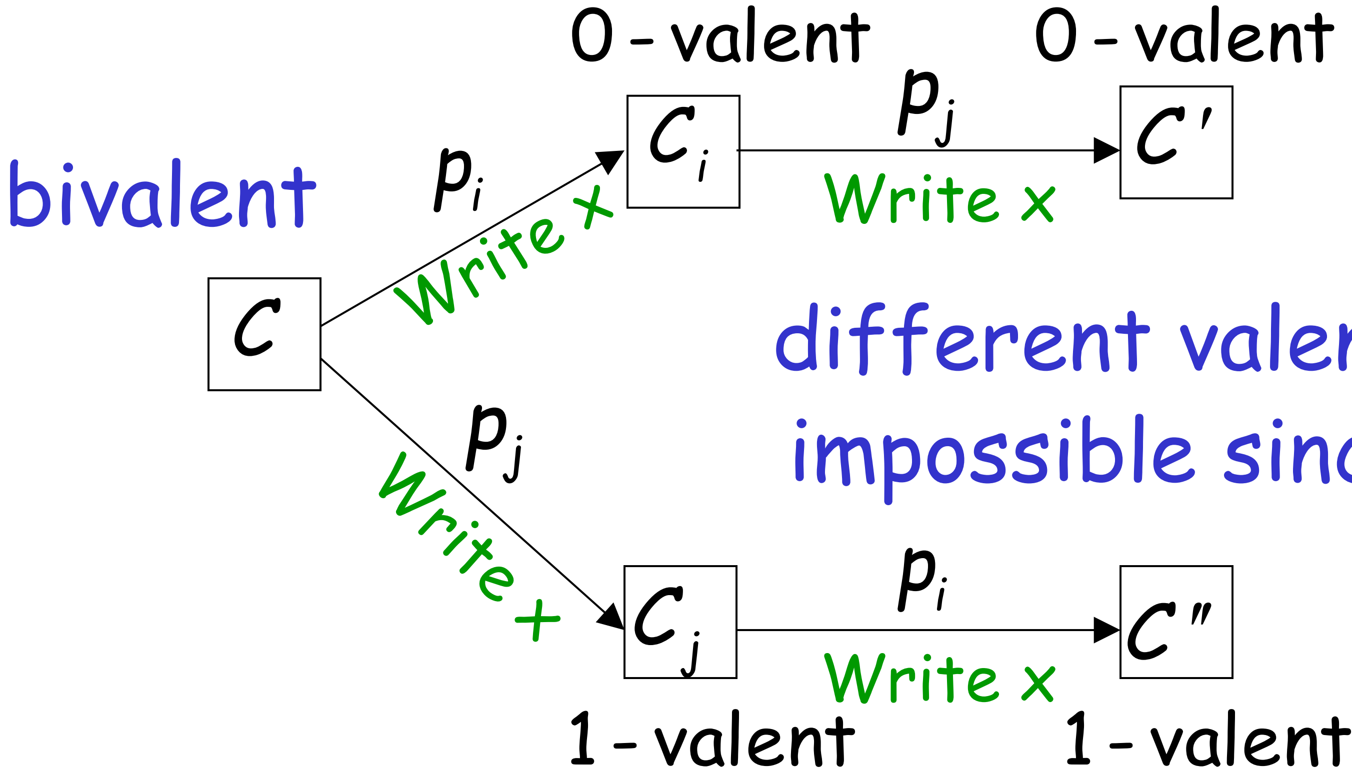
subcase: read/write



two possible executions



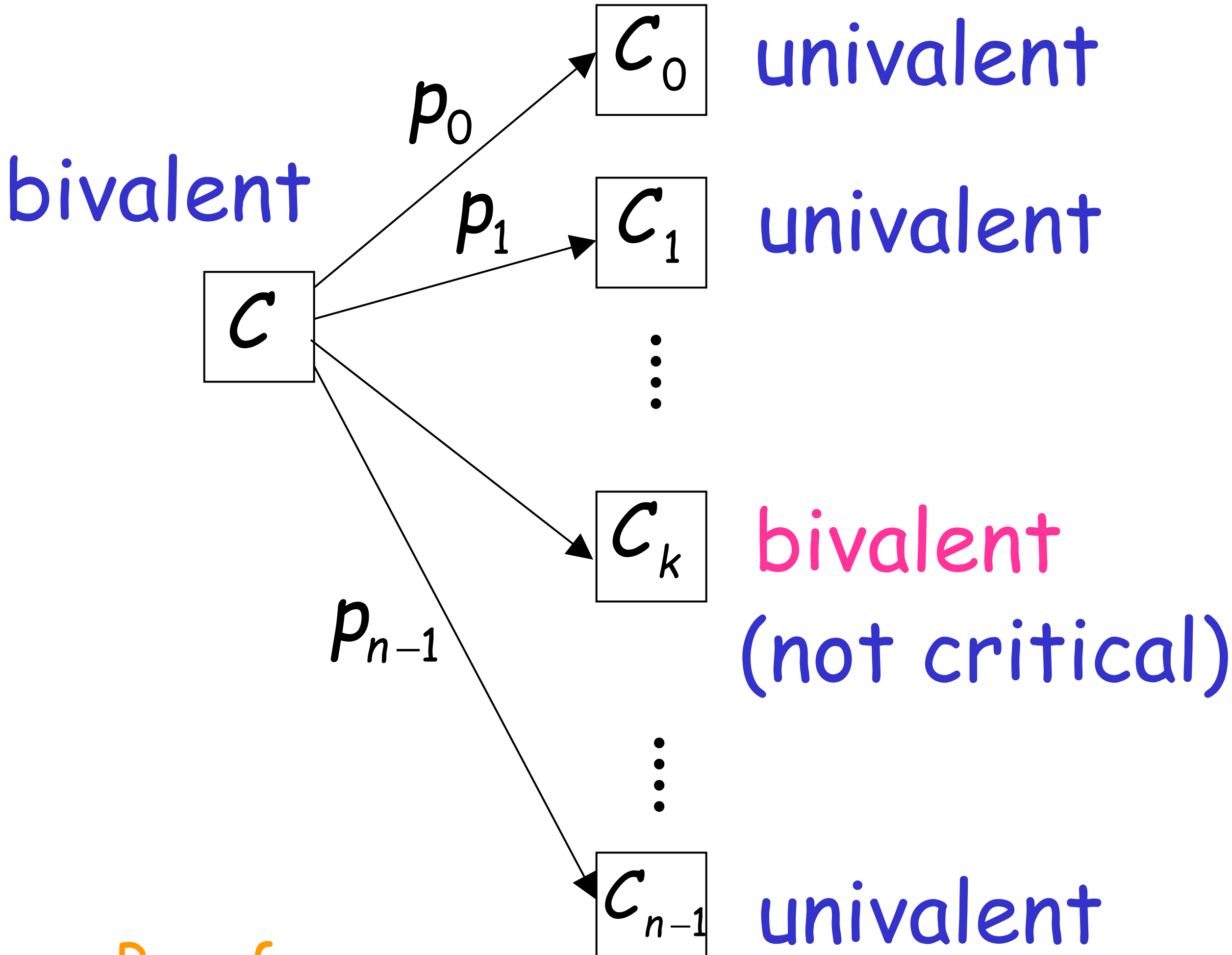
subcase: write/write



different valence impossible since $C' \overset{p_j}{\approx} C_j$

In all cases we obtained contradiction

Therefore, there exists a processor which is not critical



End of Lemma Proof

Therefore, we can construct an execution

bivalent bivalent bivalent

$$C_0 \xrightarrow{p_{i_0}} C_1 \xrightarrow{p_{i_1}} C_2 \xrightarrow{p_{i_2}} \cdots\cdots\cdots\cdots$$

Never ends

Initial configuration

Consensus can never be reached

End of Theorem Proof