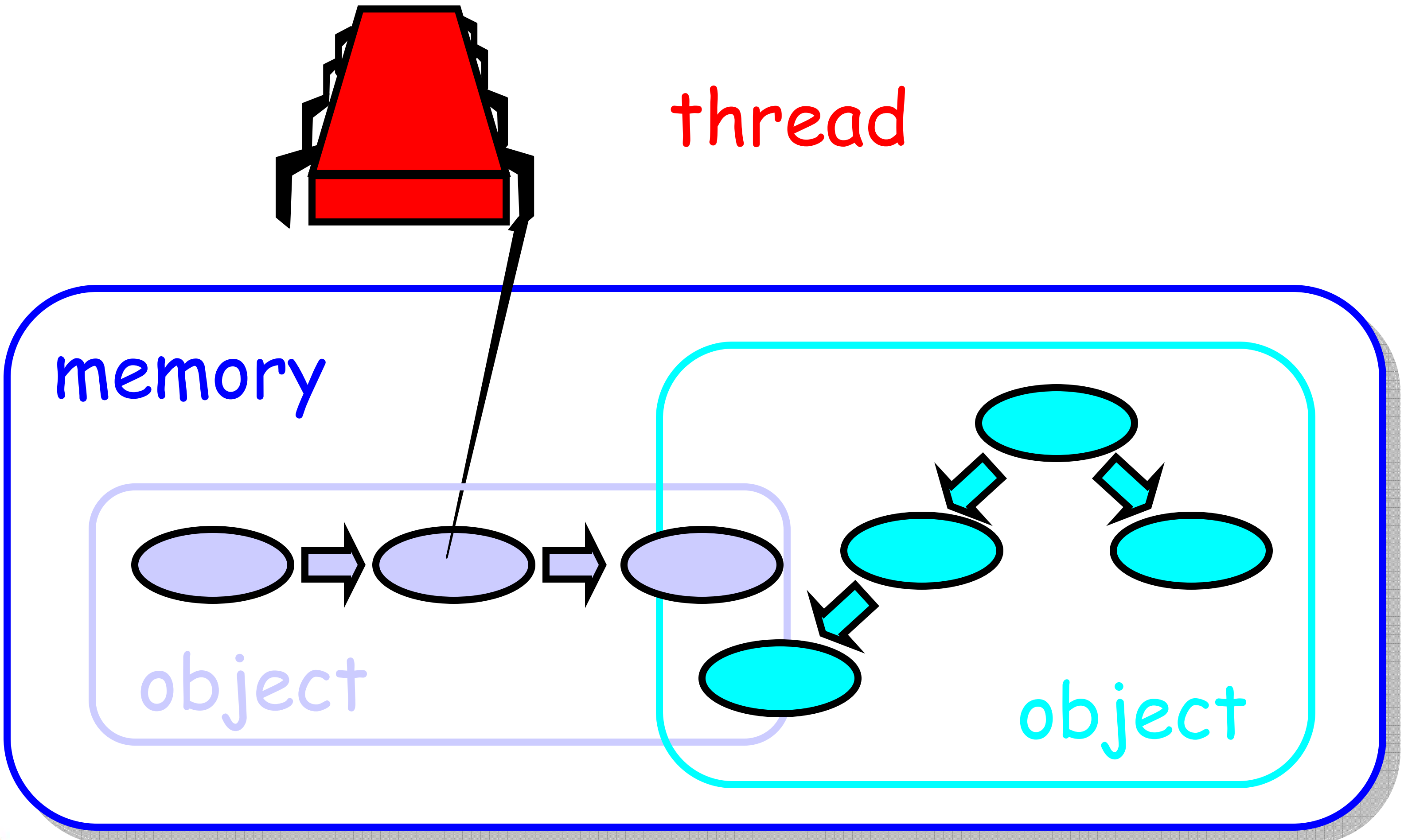
The Consensus Problem

Roger Wattenhofer

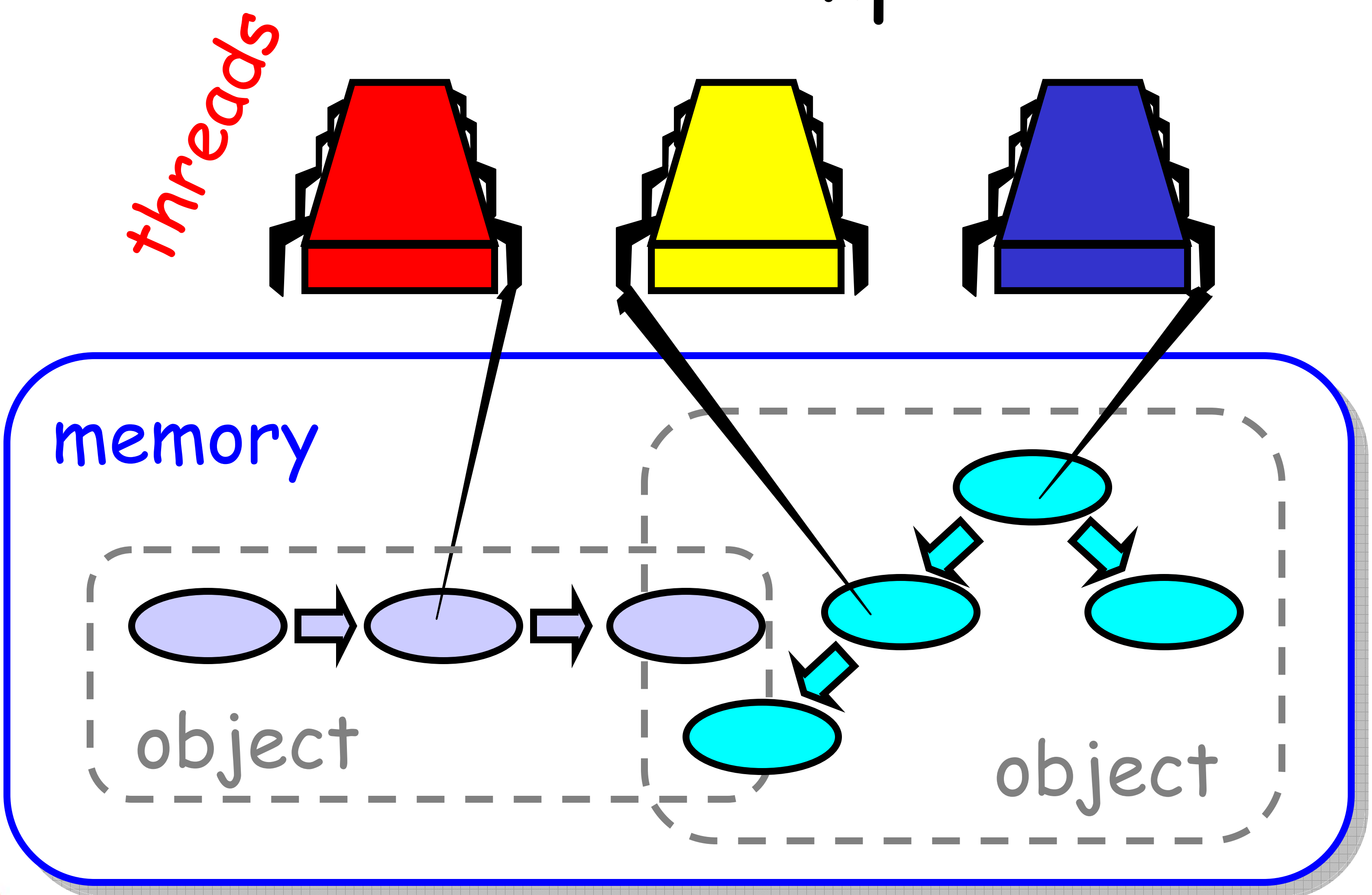
Distributed Computing Group

a lot of kudos to
Maurice Herlihy
and Costas Busch
for some of
their slides

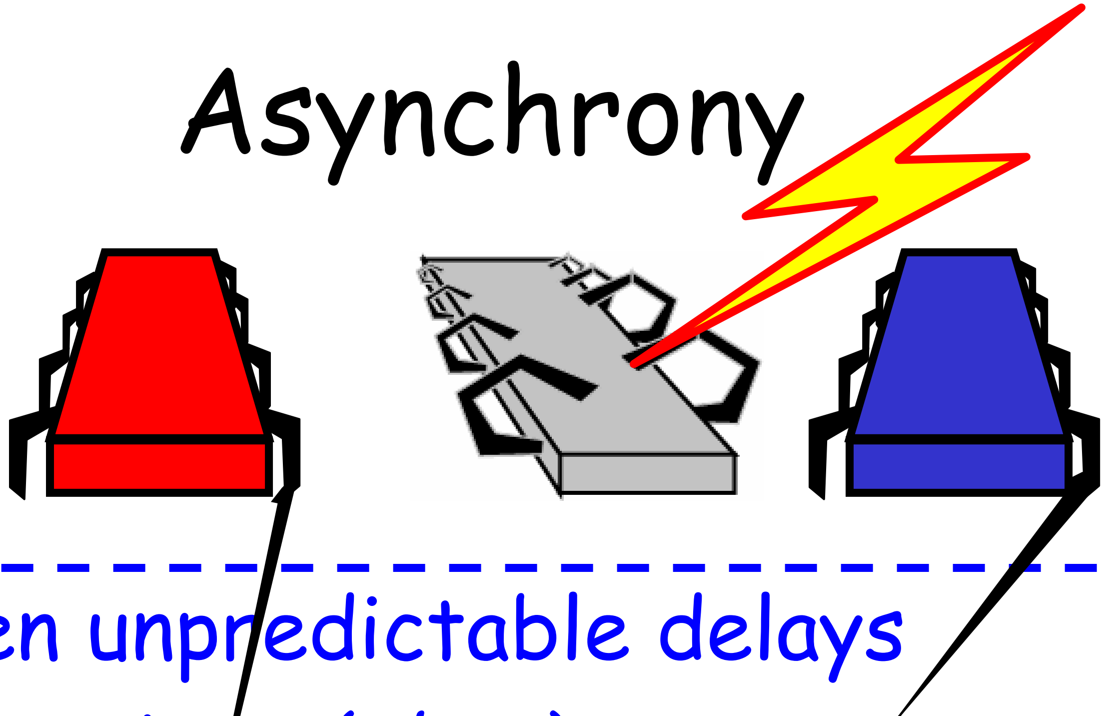
Sequential Computation



Concurrent Computation



Asynchrony



- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)

Model Summary

- Multiple *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays

Road Map

- We are going to focus on principles
  - Start with idealized models
  - Look at a simplistic problem
  - Emphasize correctness over pragmatism
  - "Correctness may be theoretical, but incorrectness has practical impact"

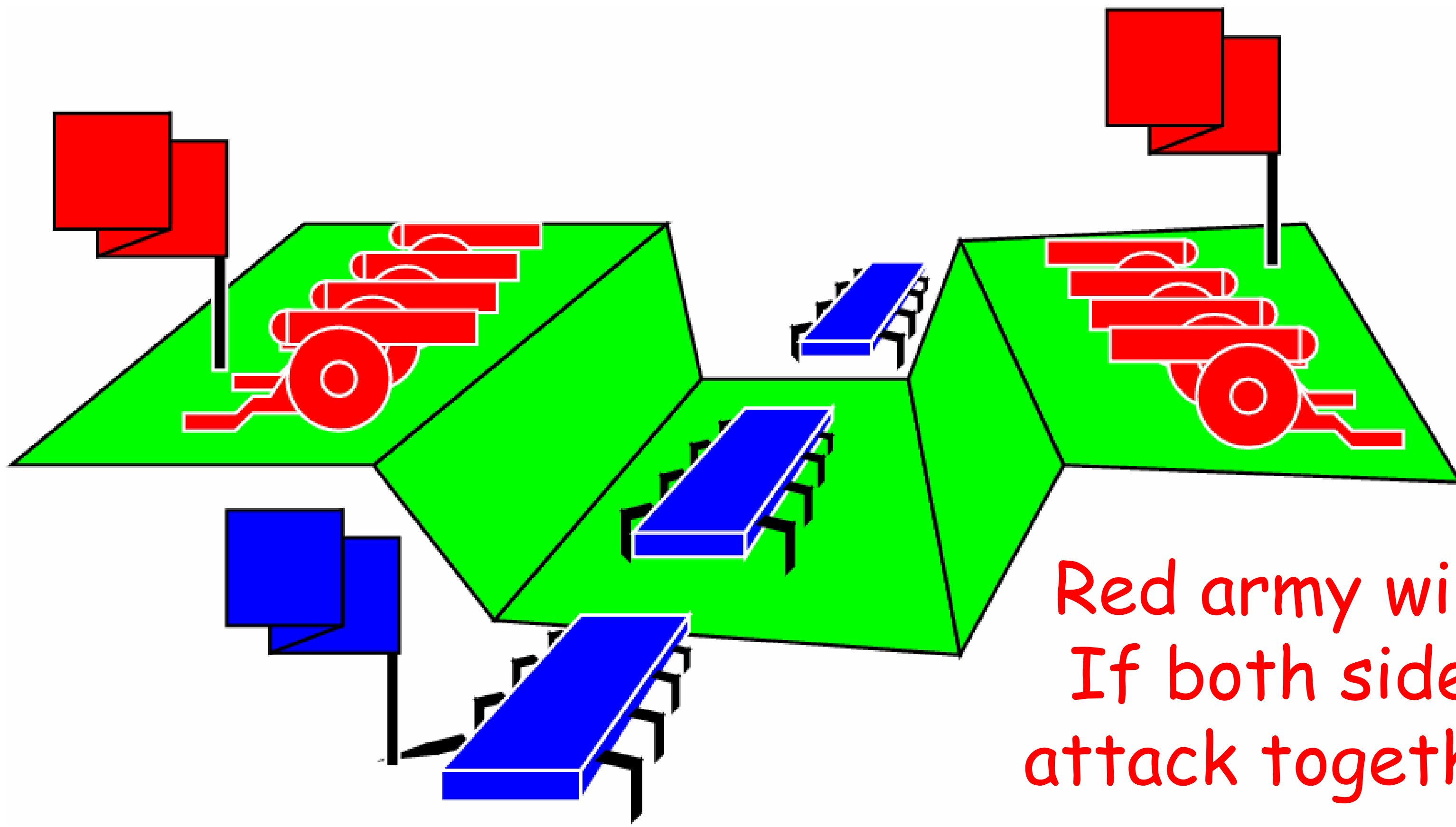
You may ask yourself …

I'm no theory weenie - why all the theorems and proofs?

Fundamentalism

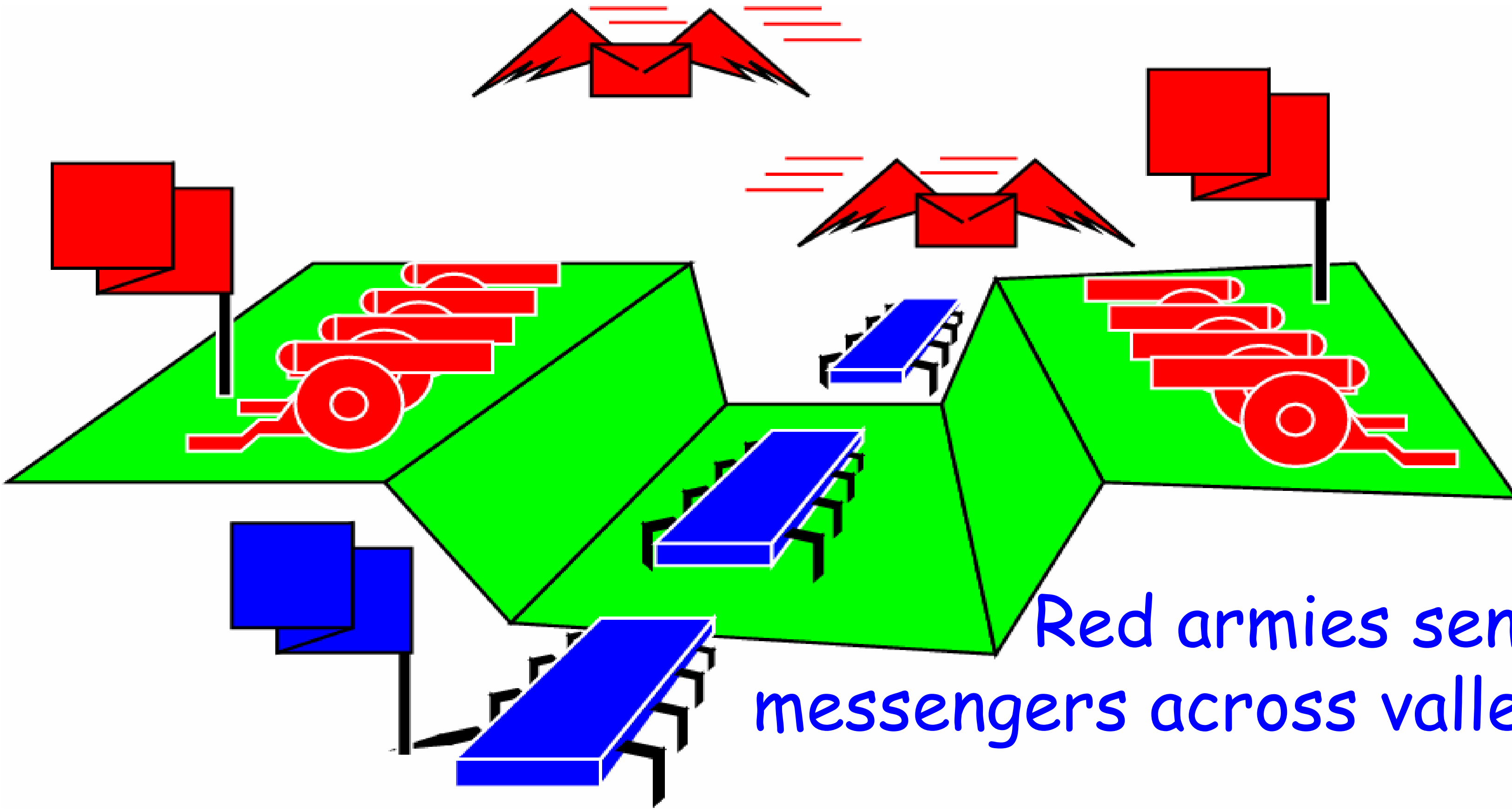
- Distributed & concurrent systems are *hard*
  - Failures
  - Concurrency
- Easier to go from theory to practice than vice-versa

The Two Generals



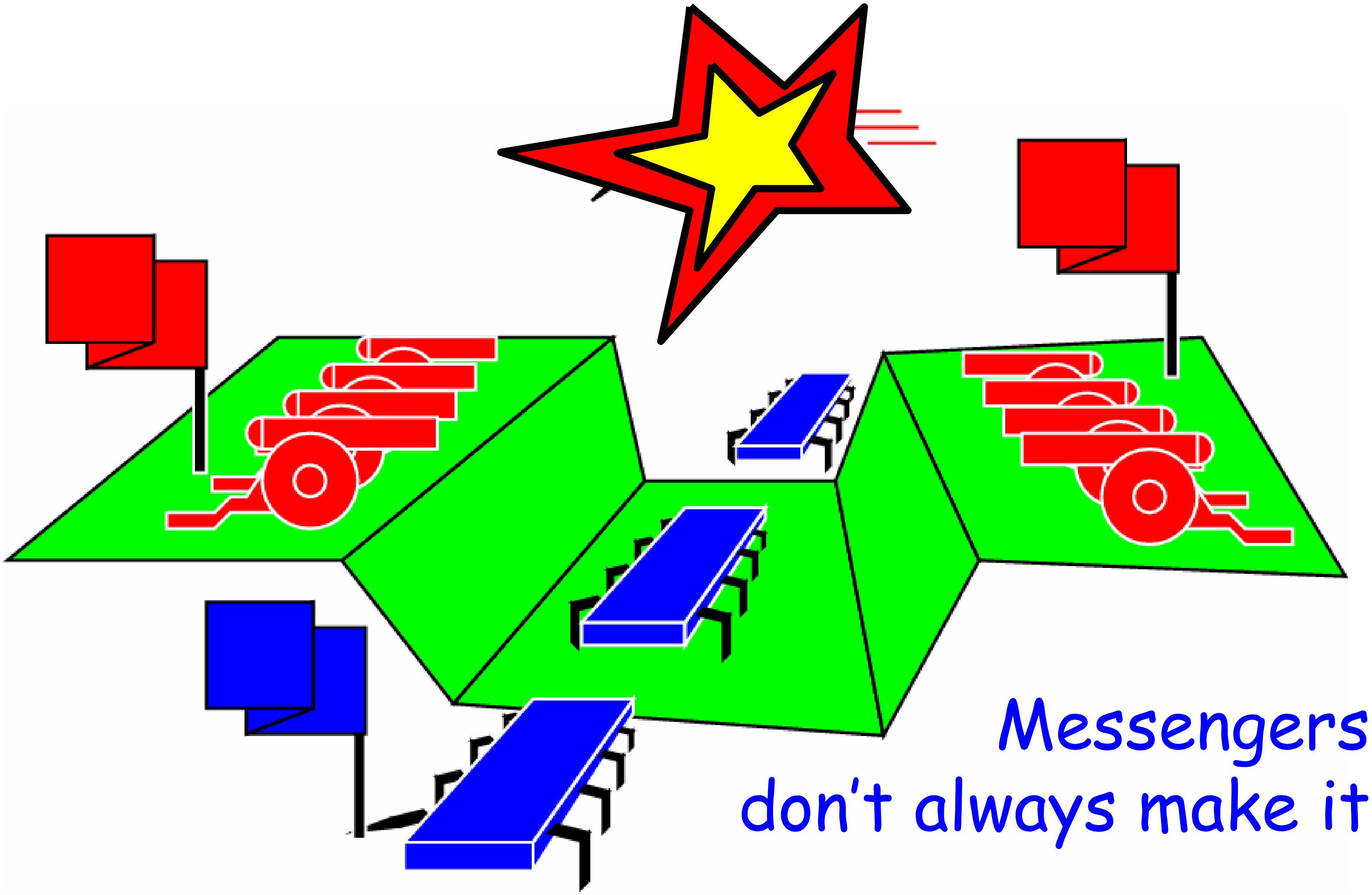
Red army wins
If both sides
attack together

Communications



Red armies send messengers across valley

Communications



Your Mission

Design a protocol to ensure that red armies attack simultaneously

Real World Generals

Date: Wed, 11 Dec 2002 12:33:58 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Vorlesung

Sie machen jetzt am Freitag, 08:15 die Vorlesung Verteilte Systeme, wie vereinbart. OK? (Ich bin jedenfalls am Freitag auch gar nicht da.) Ich uebernehme das dann wieder nach den Weihnachtsferien.

Real World Generals

Date: Mi 11.12.2002 12:34
From: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
To: Friedemann Mattern <mattern@inf.ethz.ch>
Subject: Re: Vorlesung

OK. Aber ich gehe nur, wenn sie diese Email nochmals bestaetigen... :-)

Gruesse -- Roger Wattenhofer

Real World Generals

Date: Wed, 11 Dec 2002 12:53:37 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Naechste Runde: Re: Vorlesung ...

Das dachte ich mir fast. Ich bin Praktiker und mache es schlauer: Ich gehe nicht, unabhaengig davon, ob Sie diese email bestaetigen (beziehungsweise rechtzeitig erhalten). (:-)

Real World Generals

Date: Mi 11.12.2002 13:01
From: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
To: Friedemann Mattern <mattern@inf.ethz.ch>
Subject: Re: Naechste Runde: Re: Vorlesung ...

Ich glaube, jetzt sind wir so weit, dass ich diese Emails in der Vorlesung auflegen werde...

Real World Generals

```
Date: Wed, 11 Dec 2002 18:55:08 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Re: Naechste Runde: Re: Vorlesung ...

Kein Problem. (Hauptsache es kommt raus, dass der
Prakiker am Ende der schlauere ist... Und der
Theoretiker entweder heute noch auf das allerletzte
Ack wartet oder wissend das das ja gar nicht gehen
kann alles gleich von vornherein bleiben laesst...
(:-))
```

Theorem

There is no non-trivial protocol that ensures the red armies attacks simultaneously

Proof Strategy

- Assume a protocol exists
- Reason about its properties
- Derive a contradiction

Proof

1. Consider the protocol that sends fewest messages
2. It still works if last message lost
3. So just don't send it
   - Messengers' union happy
4. But now we have a shorter protocol!
5. Contradicting #1

Fundamental Limitation

- Need an unbounded number of messages
- Or possible that no attack takes place

You May Find Yourself …

I want a real-time YAFA compliant Two Generals protocol using UDP datagrams running on our enterprise-level fiber tachyion network ...

You might say

I want a real-time YAFA

p

ru

fiber tachyion netwo

Yes, Ma'am, right away!

You might say

Advantage:

- Buys time to find another job
- No one expects software to work anyway

fiber tachyion netwo

You might say

Advantage:

- Buys time to find another job
- No
any

Disadvantage:

- You're doomed
- Without this course, you may not even know you're doomed

You might say

I can't find a fault-tolerant algorithm, I guess I'm just a pathetic loser.

fiber tachyion netw

You might say

Advantage:

- No need to take course

olerant algorithm, I guess I'm just a pathetic loser.

fiber tachyion netw

You might say

Advantage:

- No need to take course

Disadvantage:

- Boss fires you, hires University St. Gallen graduate

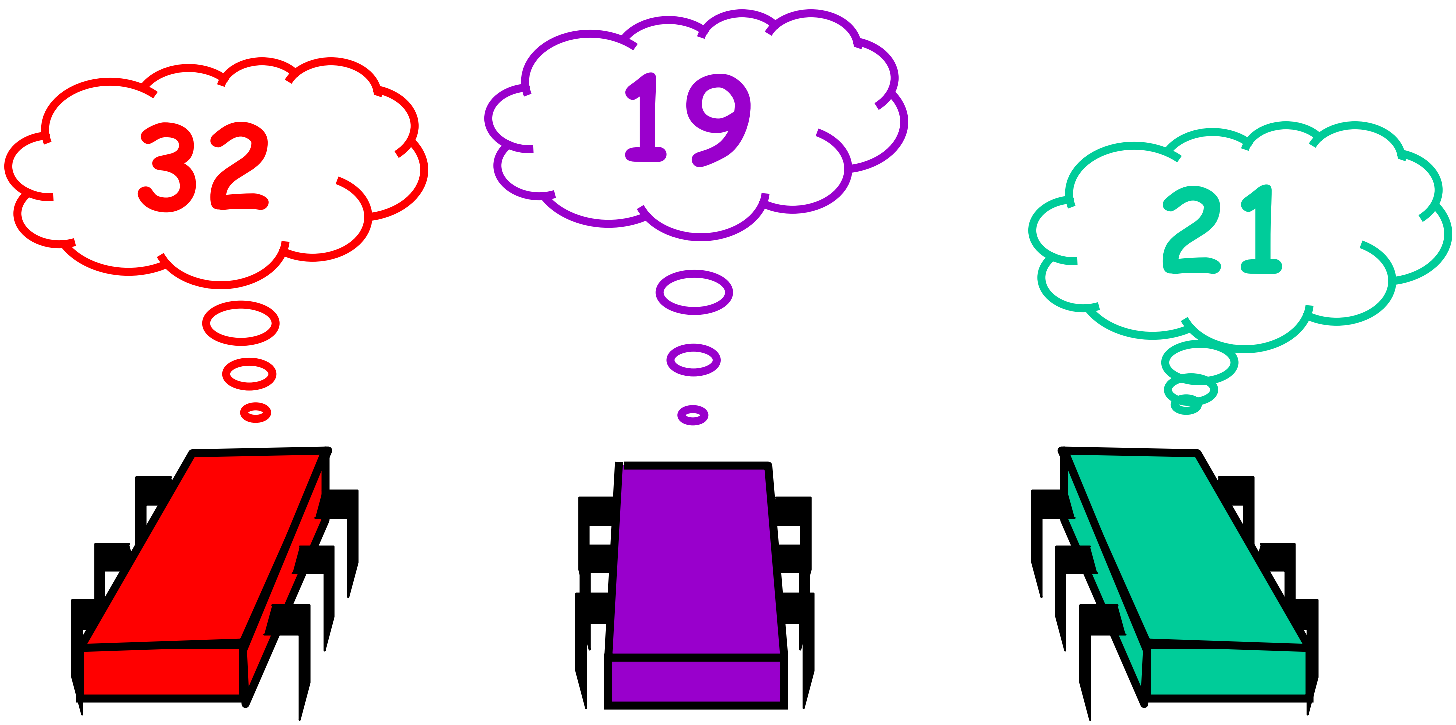
You might say

I want a real-time YAFA

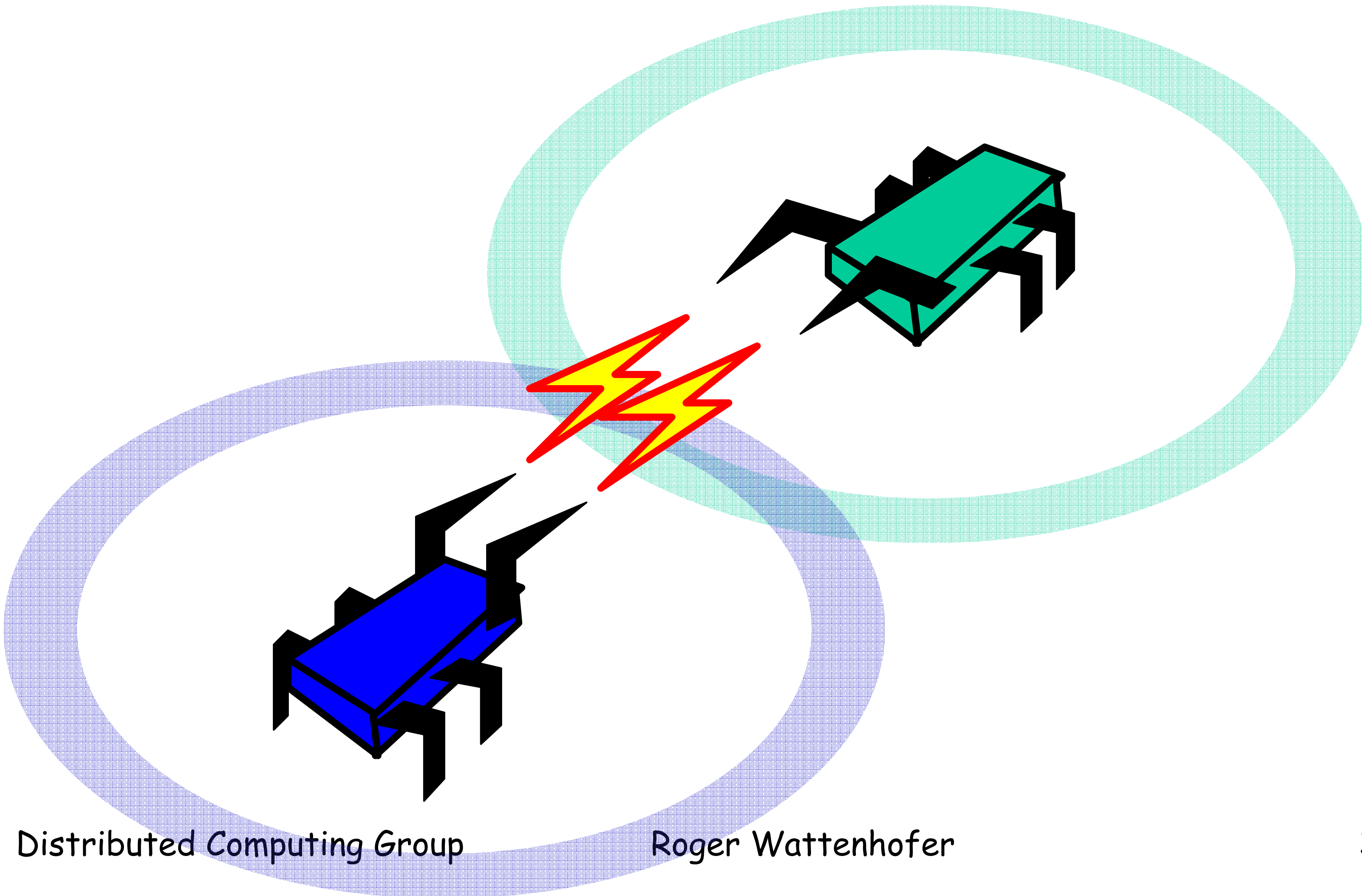
Using skills honed in course, I can avert certain disaster!

- Rethink problem spec, or
- Weaken requirements, or
- Build on different platform

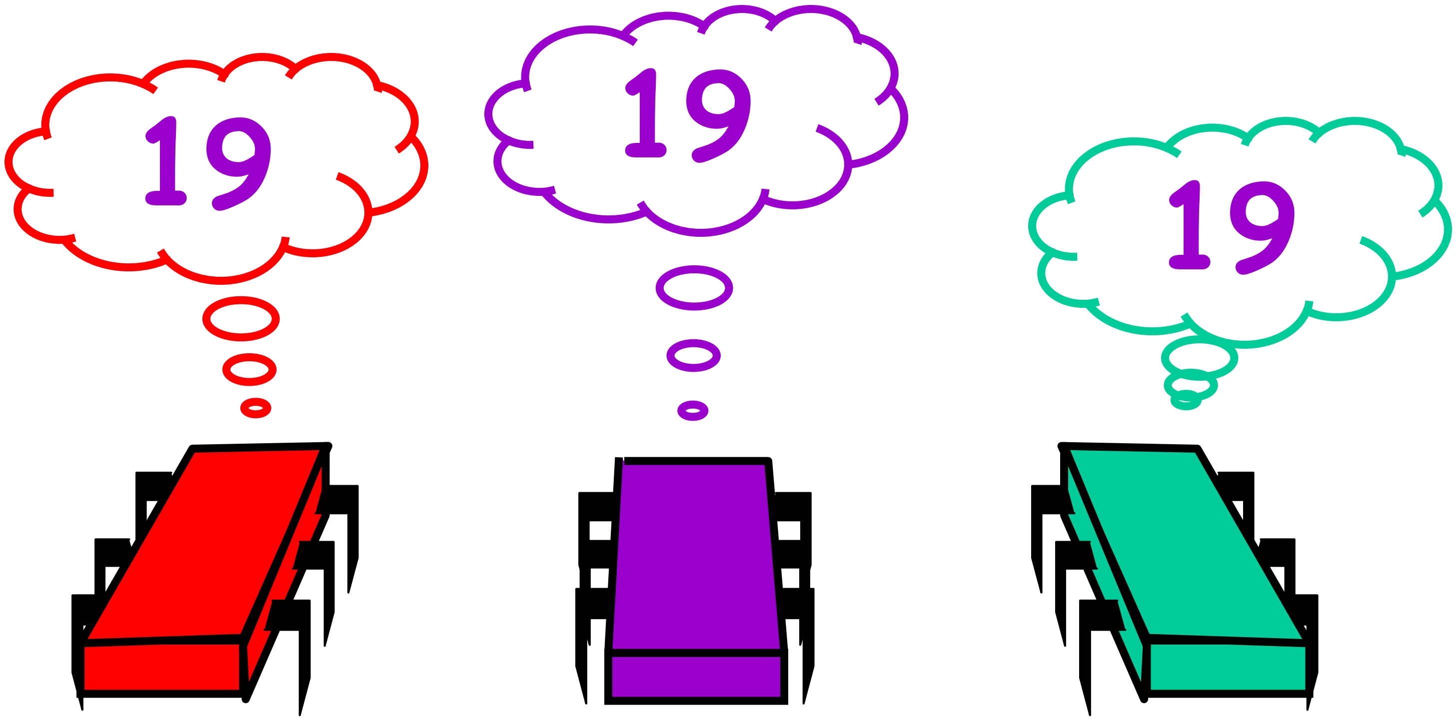
Consensus: Each Thread has a Private Input



They Communicate



They Agree on Some Thread's Input



Consensus is important

- With consensus, you can implement anything you can imagine…
- Examples: with consensus you can decide on a leader, implement mutual exclusion, or solve the two generals problem

You gonna learn

- In some models, consensus is possible
- In some other models, it is not

- Goal of this and next lecture: to learn whether for a given model consensus is possible or not ... and prove it!

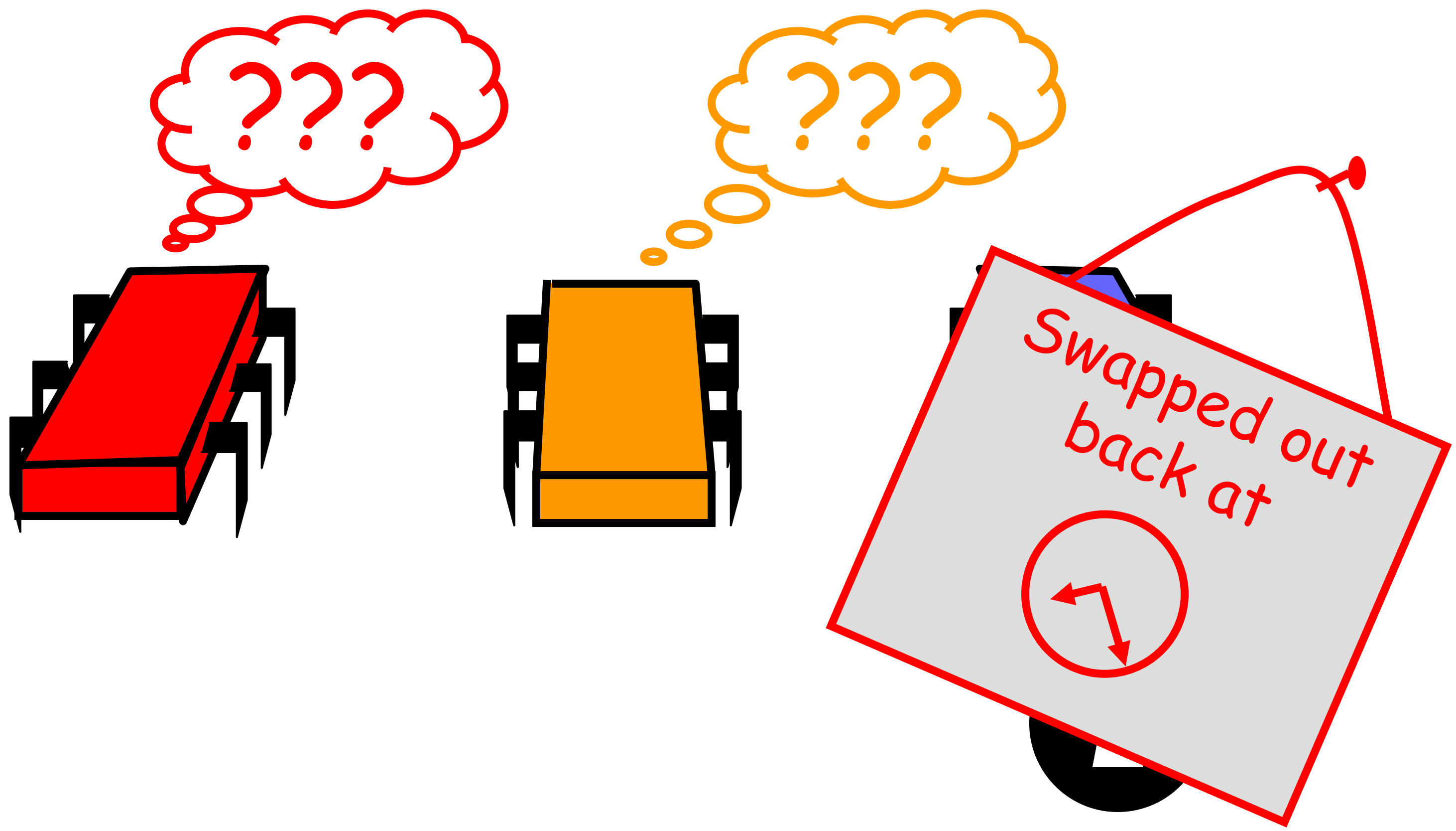
Consensus #1
shared memory

- n processors, with n > 1
- Processors can atomically *read* or *write* (not both) a shared memory cell

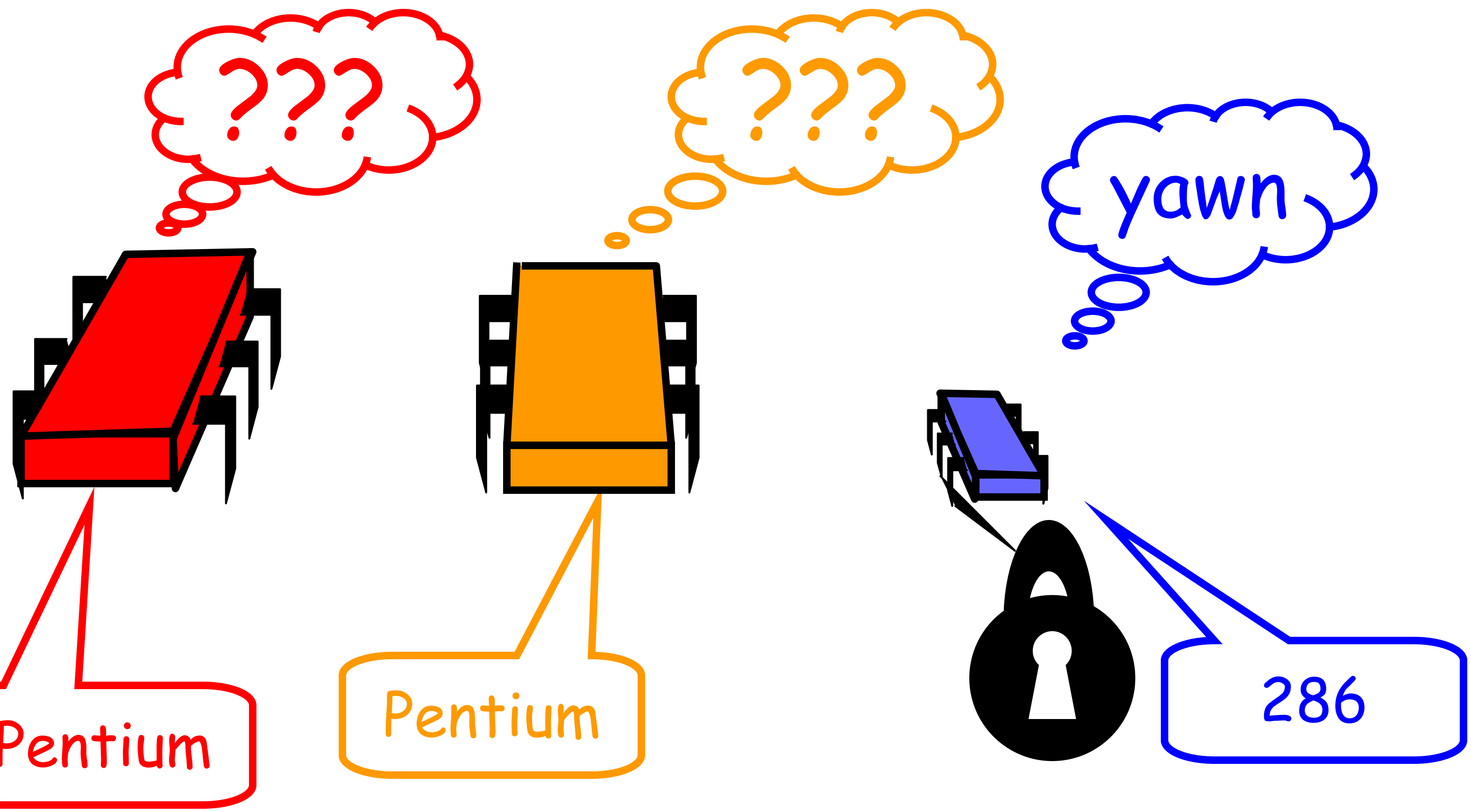
Protocol (Algorithm?)

- There is a designated memory cell c.
- Initially c is in a special state "?"
- Processor 1 writes its value v_1 into c, then decides on v_1.
- A processor j (j not 1) reads c until j reads something else than "?", and then decides on that.

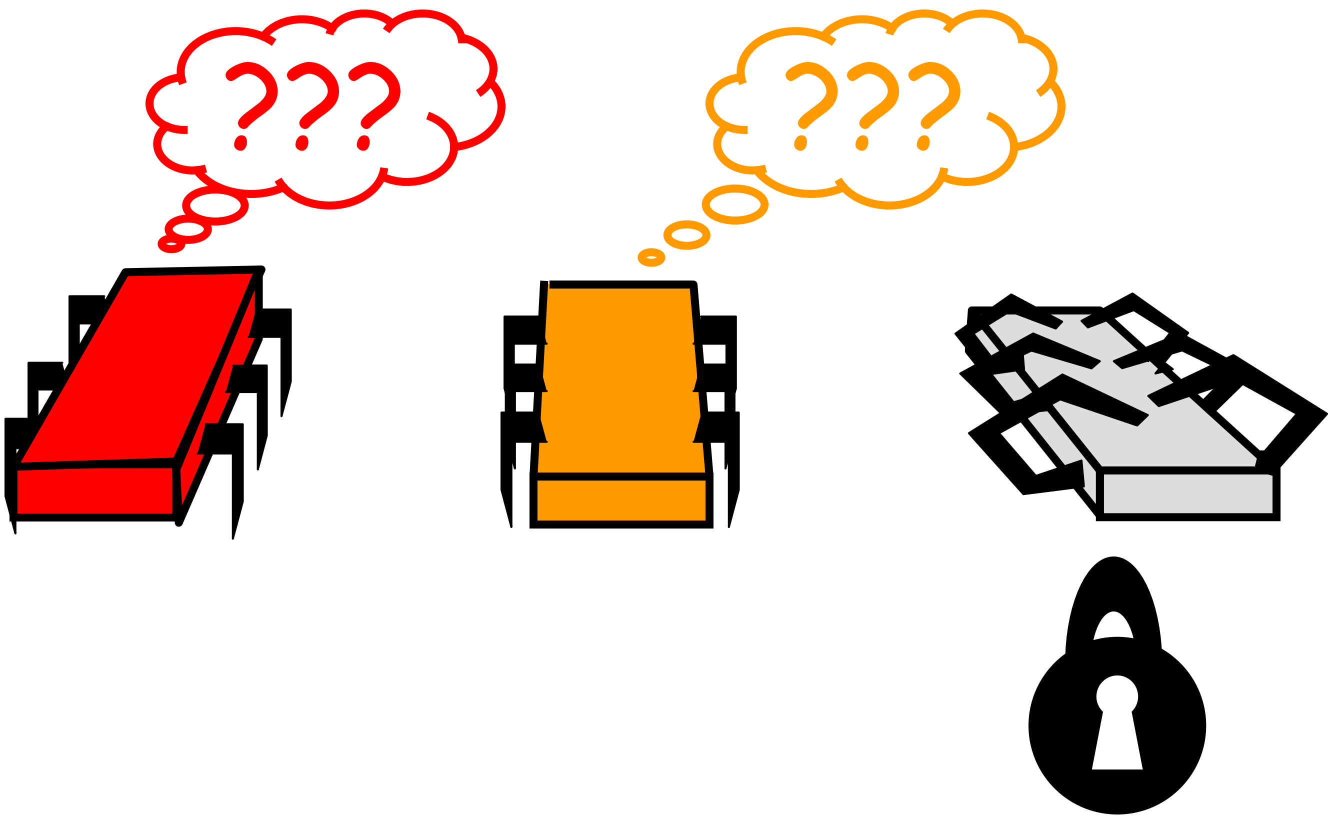
Unexpected Delay



Heterogeneous Architectures



Fault-Tolerance



Consensus #2
wait-free shared memory

- n processors, with n > 1
- Processors can atomically *read* or *write* (not both) a shared memory cell
- Processors might crash (halt)
- Wait-free implementation… huh?

Wait-Free Implementation

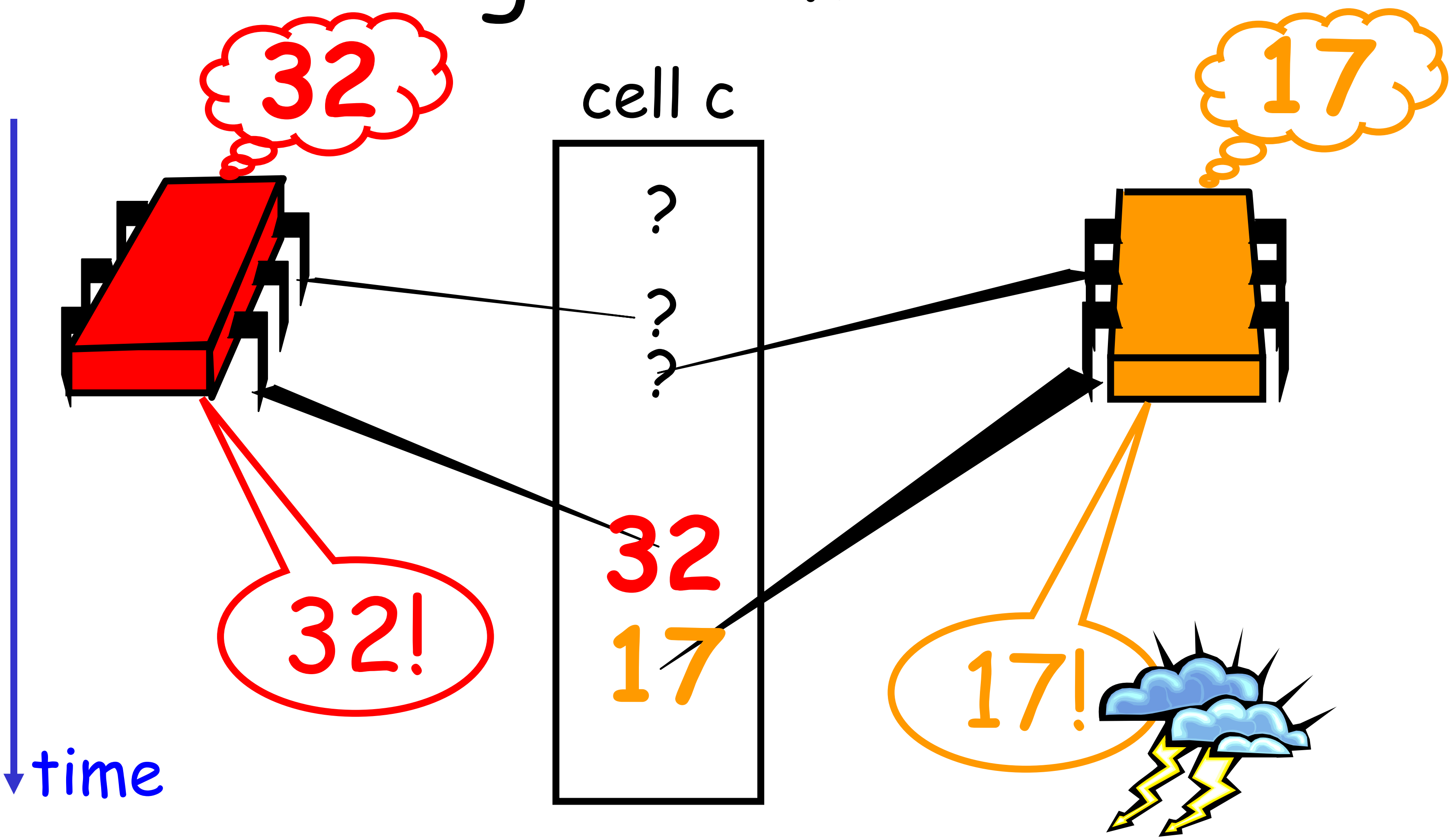
- Every process (method call) completes in a finite number of steps
- Implies no mutual exclusion
- We assume that we have wait-free atomic registers (that is, reads and writes to same register do not overlap)

A wait-free algorithm…

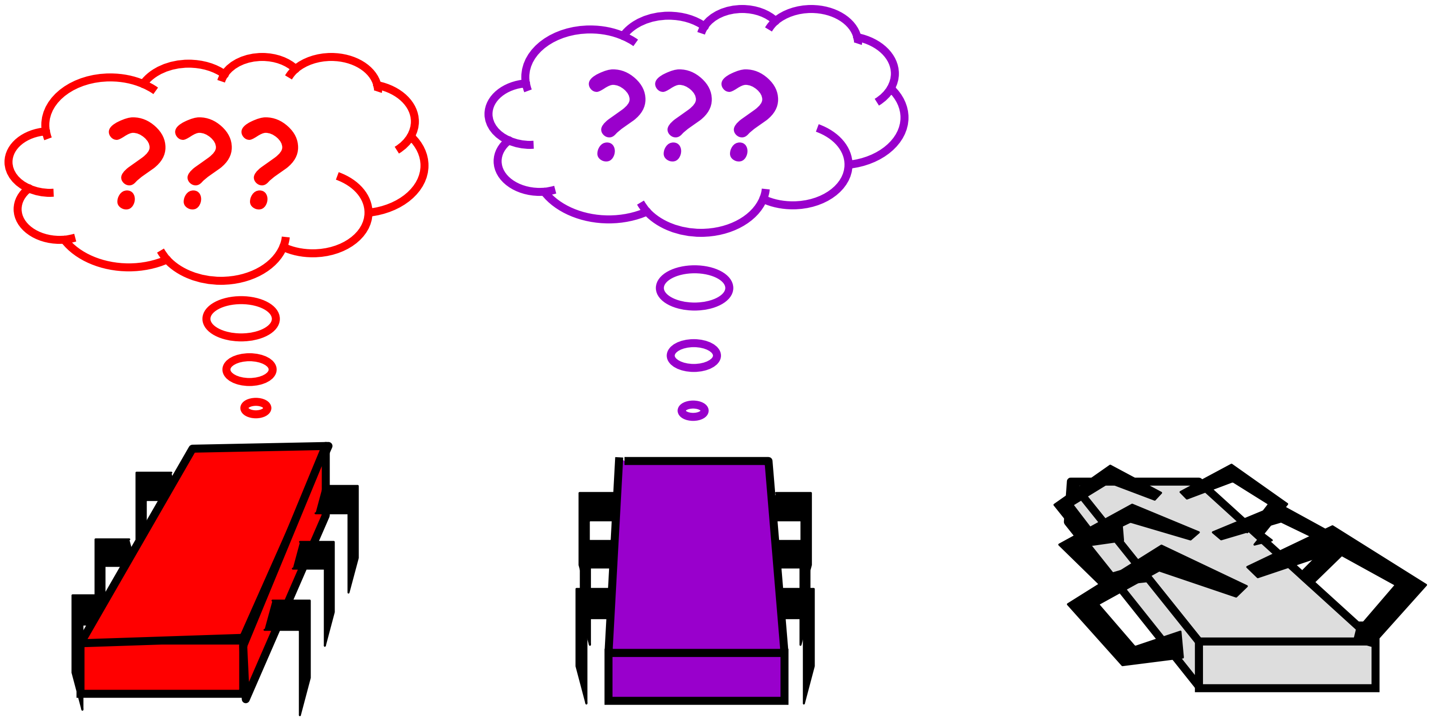
- There is a cell c, initially c="?"
- Every processor i does the following

```
r = Read(c);
if (r == "?") then
      Write(c, vi); decide vi;
else
      decide r;
```

Is the algorithm correct?



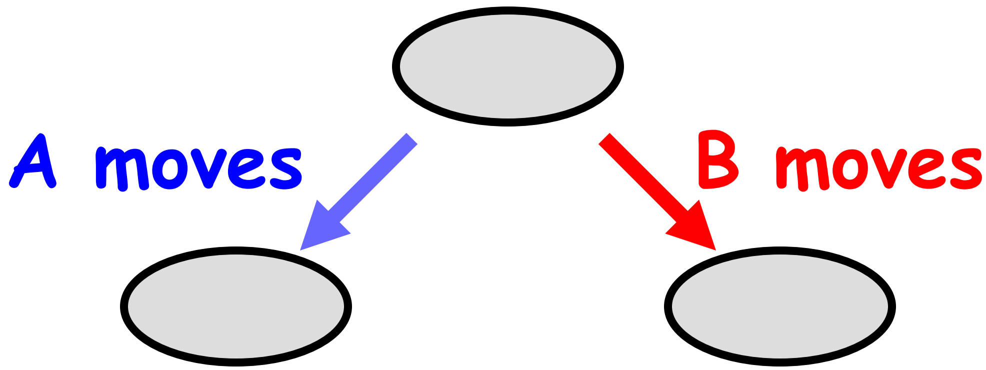
Theorem:
No wait-free consensus



Proof Strategy

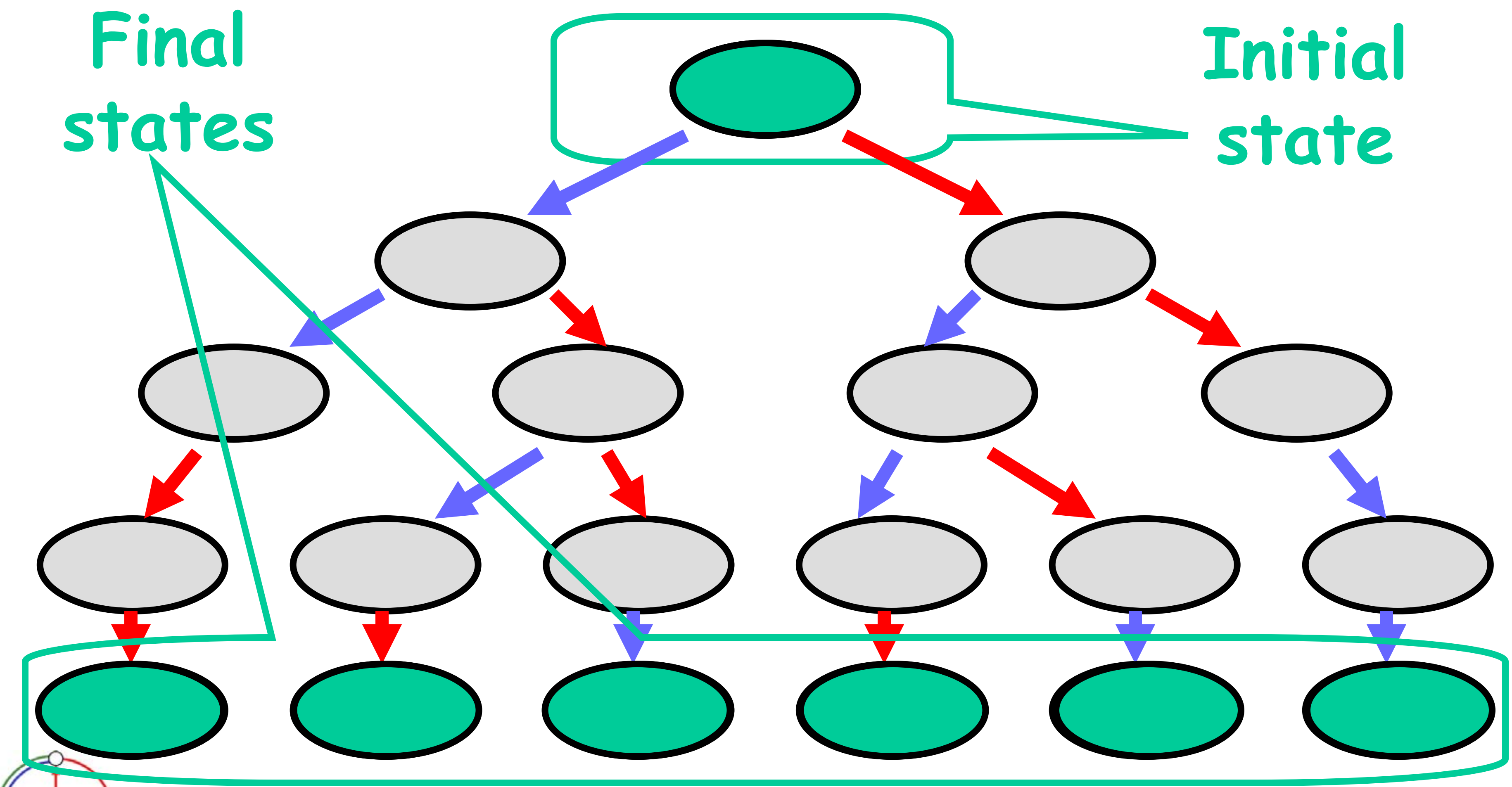
- Make it simple
  - $n = 2$, binary input
- Assume that there is a protocol
- Reason about the properties of any such protocol
- Derive a contradiction

Wait-Free Computation

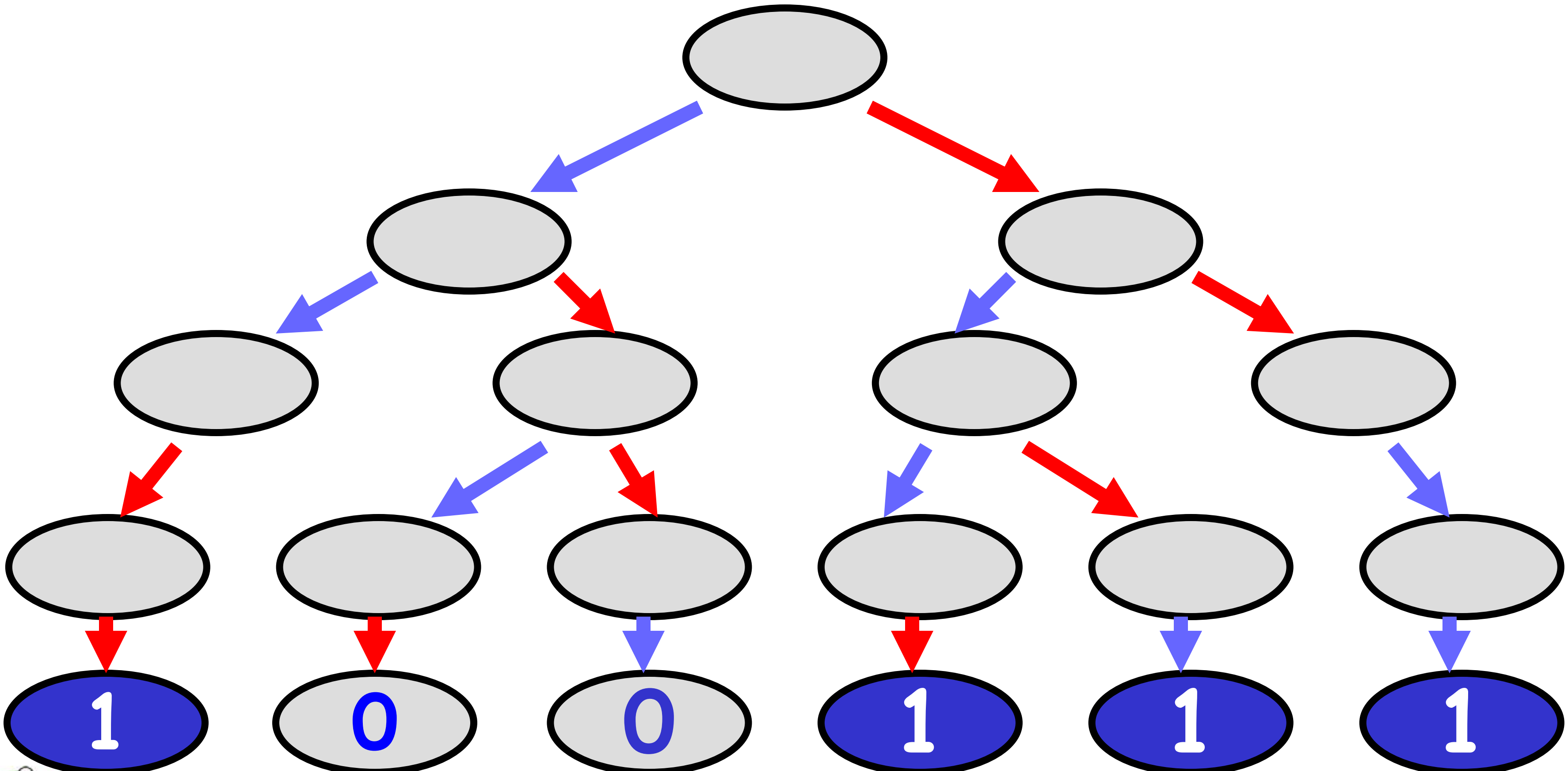


- Either A or B "moves"
- Moving means
  - Register read
  - Register write

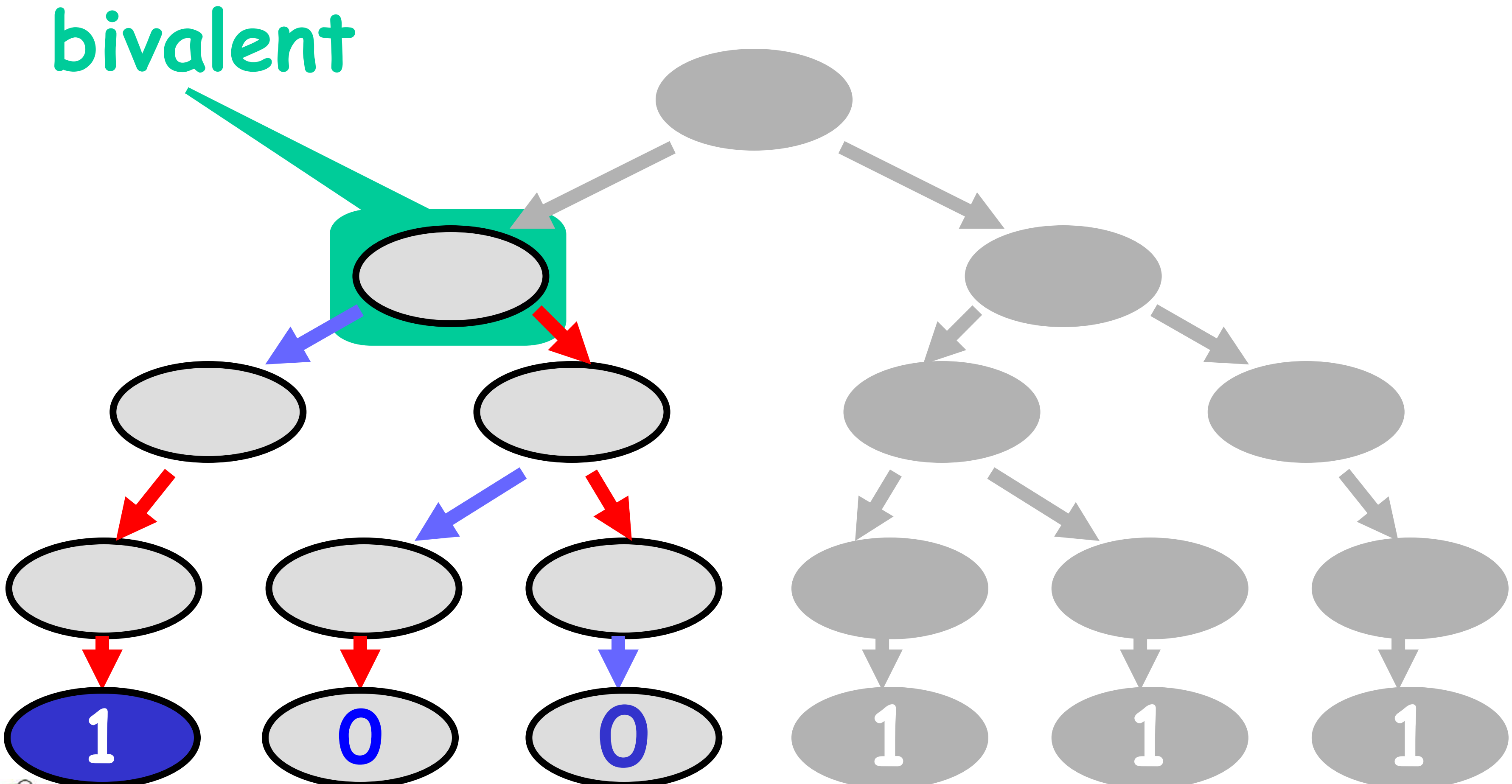
The Two-Move Tree



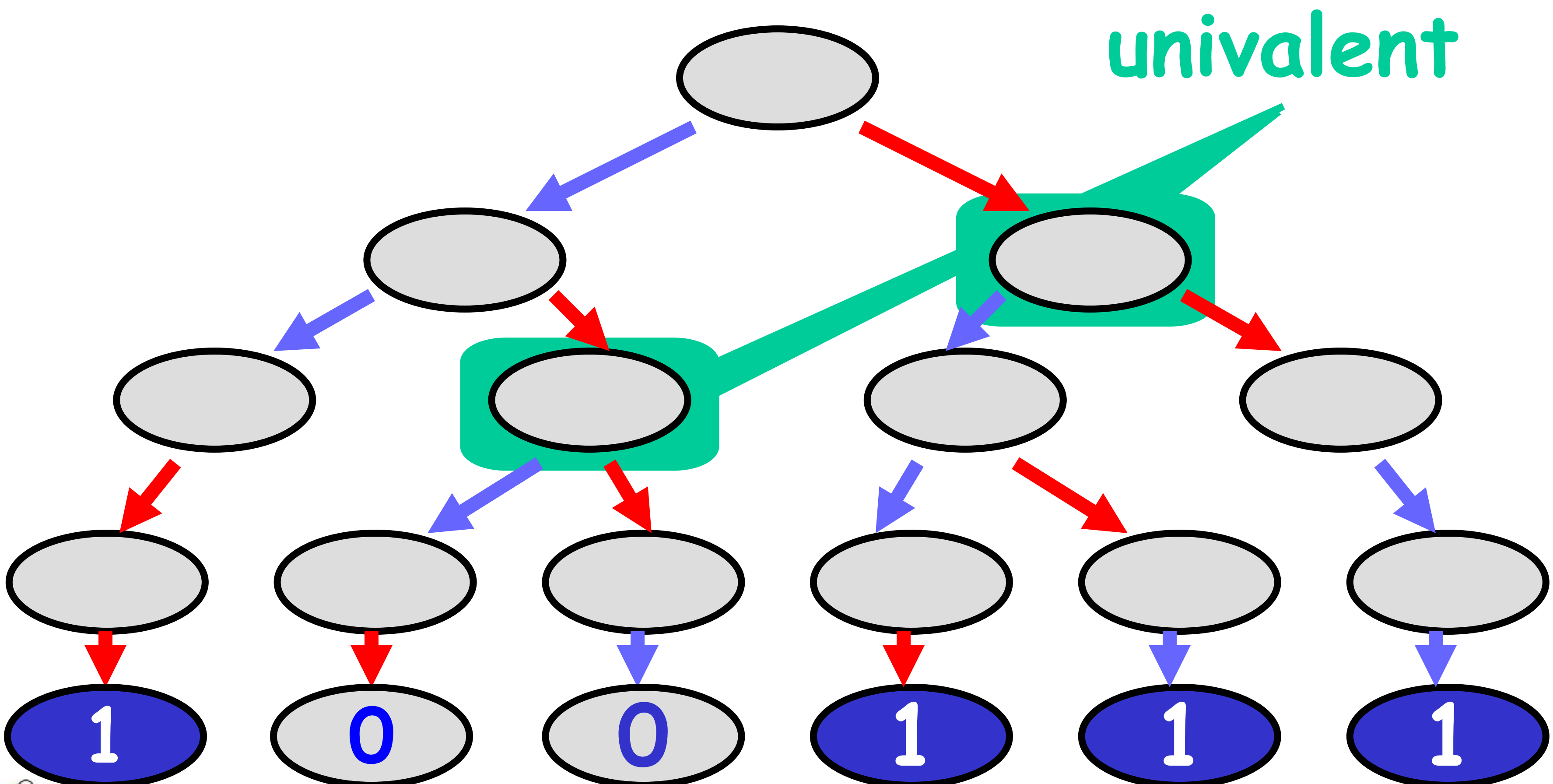
Decision Values



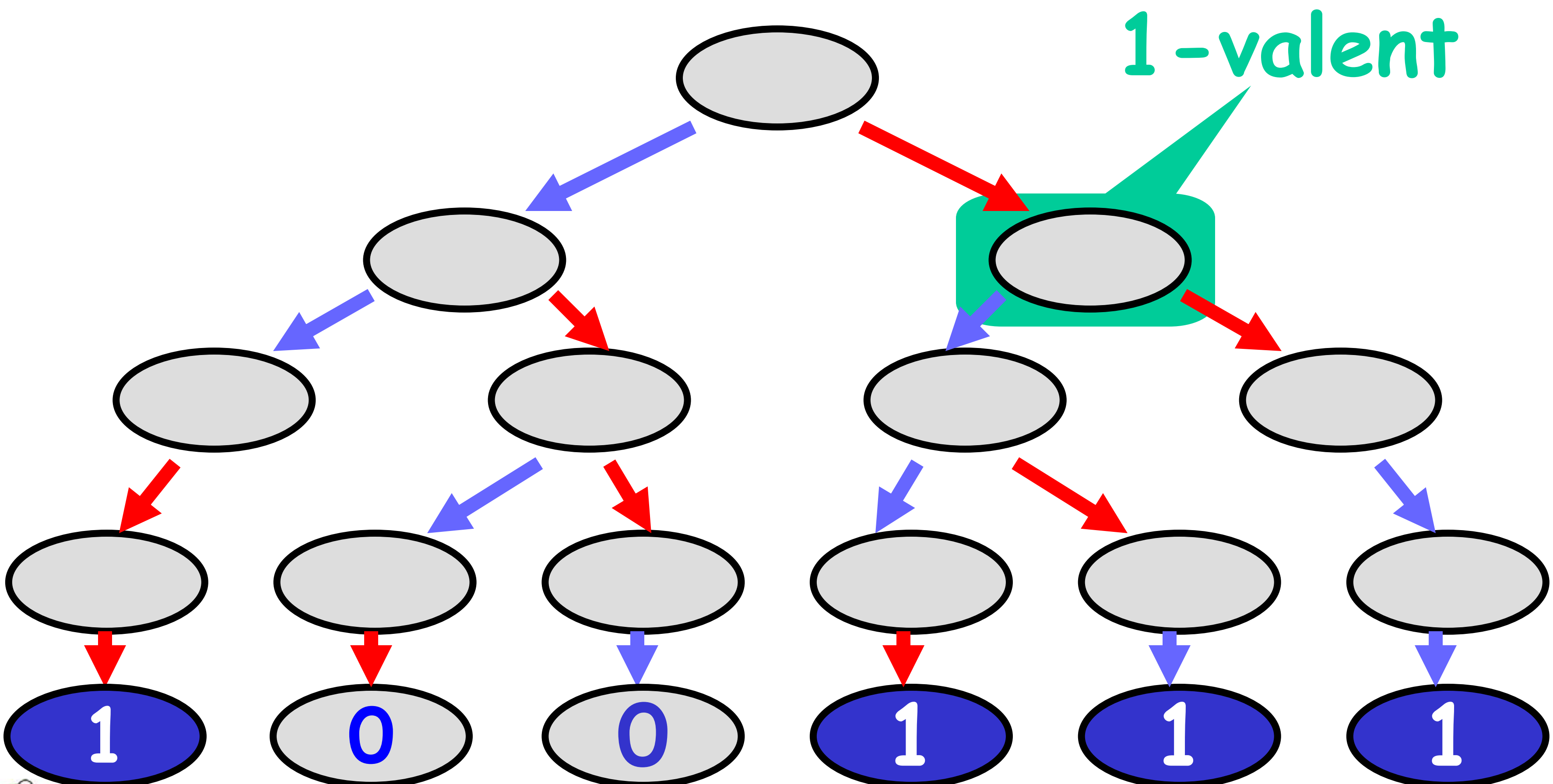
Bivalent: Both Possible



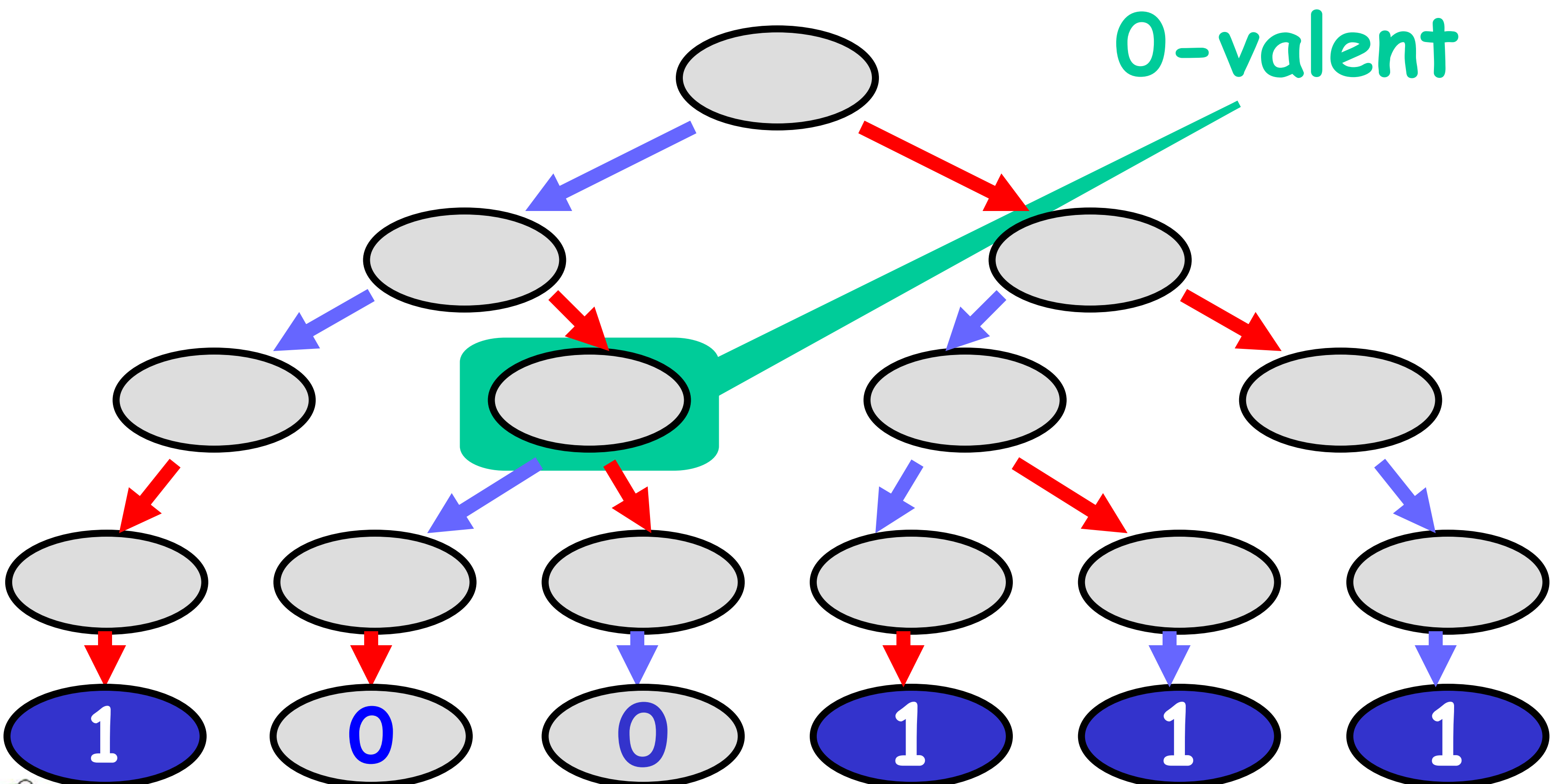
Univalent: Single Value Possible



1-valent: Only 1 Possible



0-valent: Only 0 possible



Summary

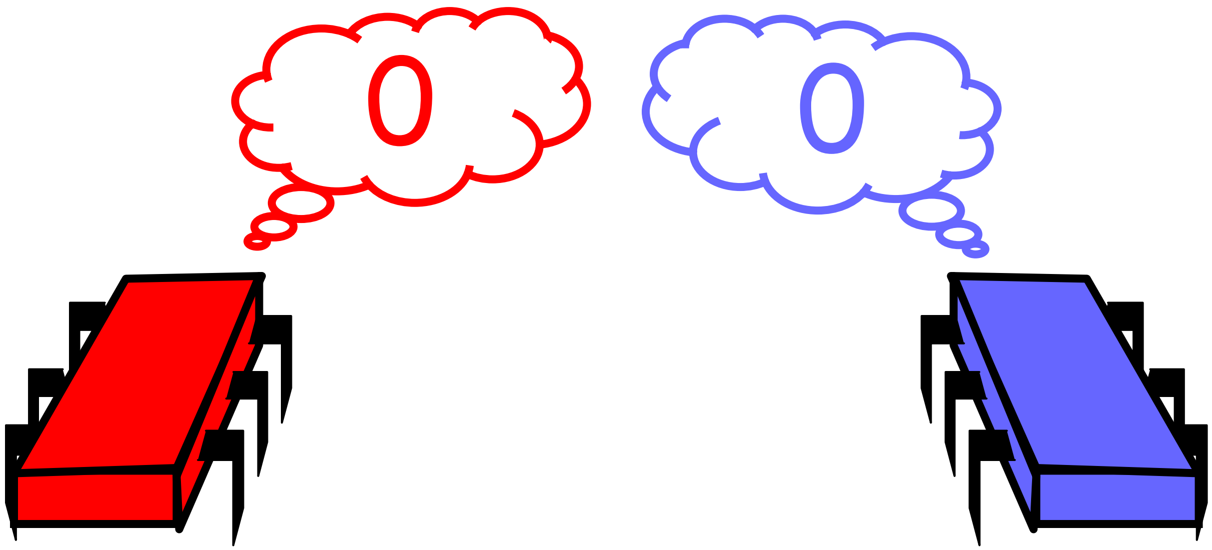
- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed
- Univalent states
  - Outcome is fixed
  - Maybe not "known" yet
  - 1-Valent and 0-Valent states

Claim

Some initial system state is bivalent

(The outcome is not always fixed from the start.)

A 0-Valent Initial State

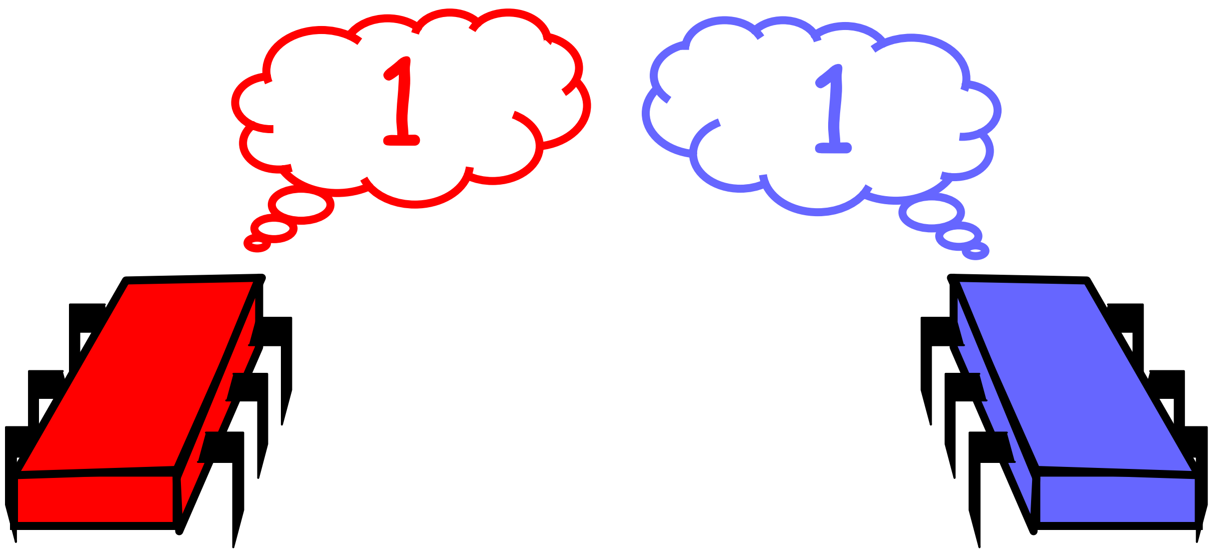


- *All executions lead to decision of* 0

A 0-Valent Initial State

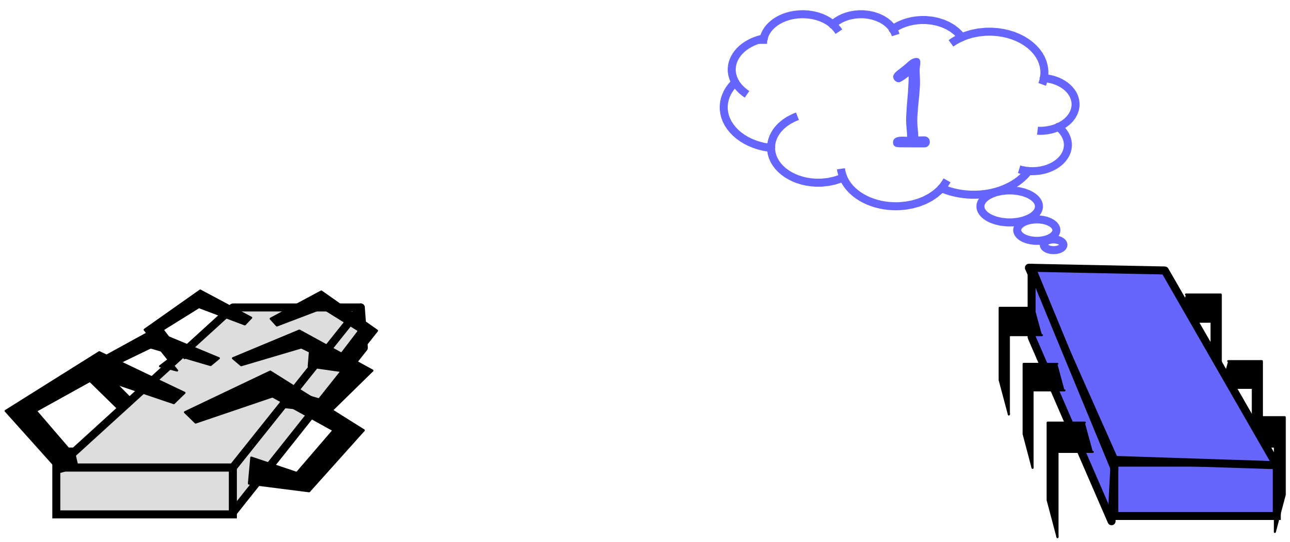
- Solo execution by *A* also decides 0

A 1-Valent Initial State



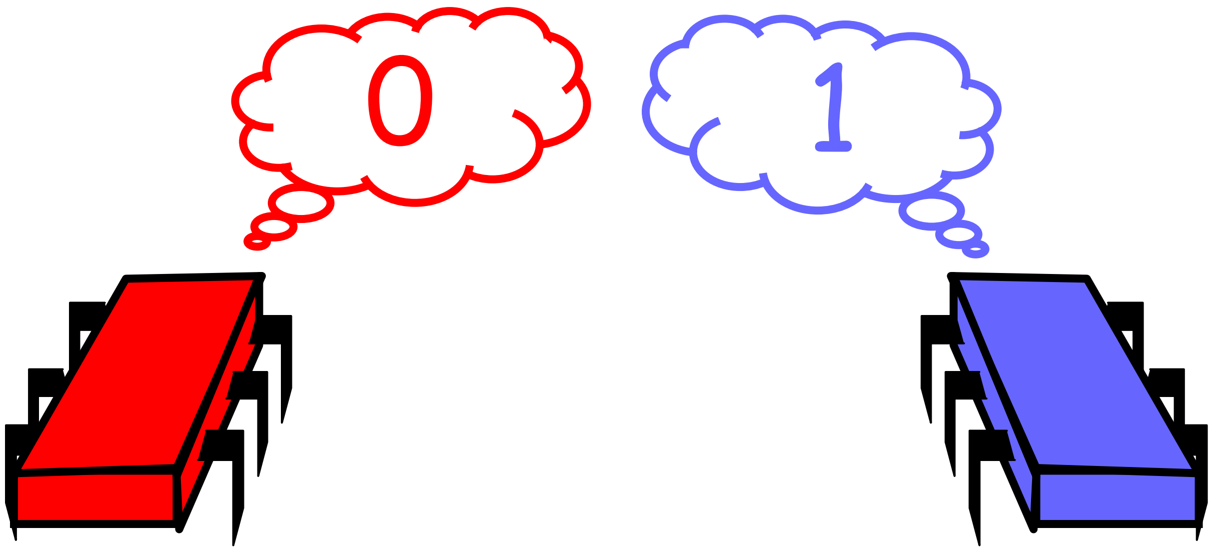
- All executions lead to decision of 1

A 1-Valent Initial State



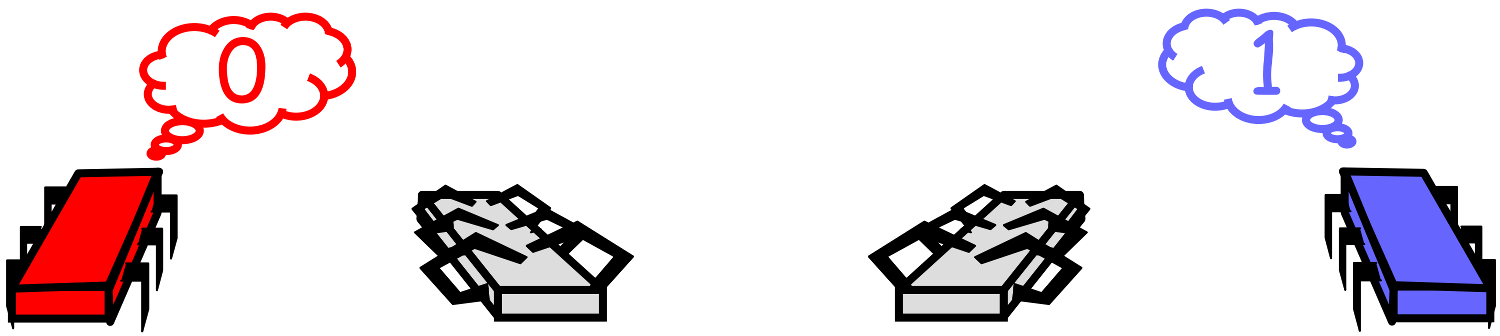
- Solo execution by B also decides 1

A Univalent Initial State?



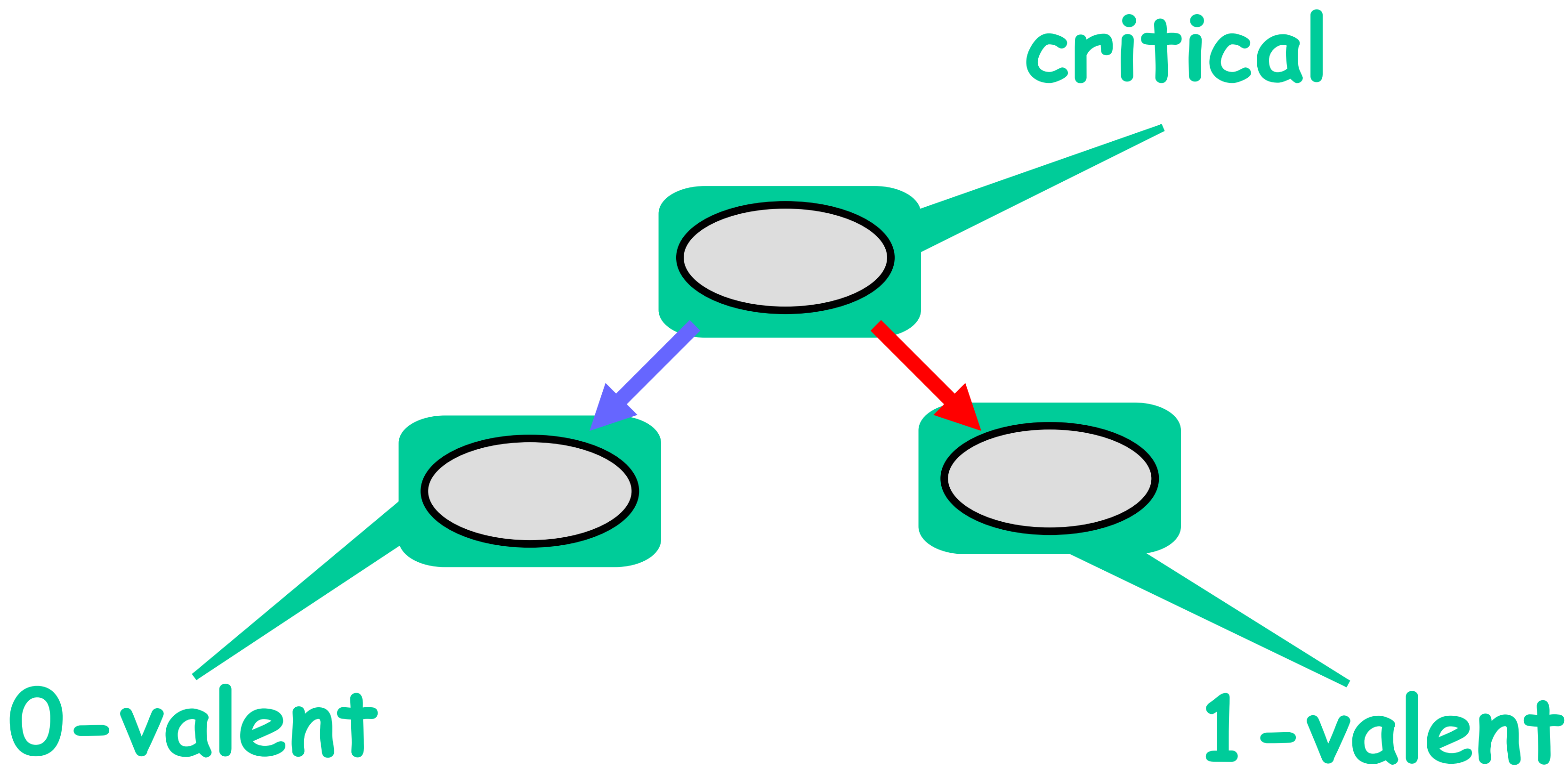
- Can all executions lead to the same decision?

State is Bivalent



- Solo execution by A must decide 0
- Solo execution by B must decide 1

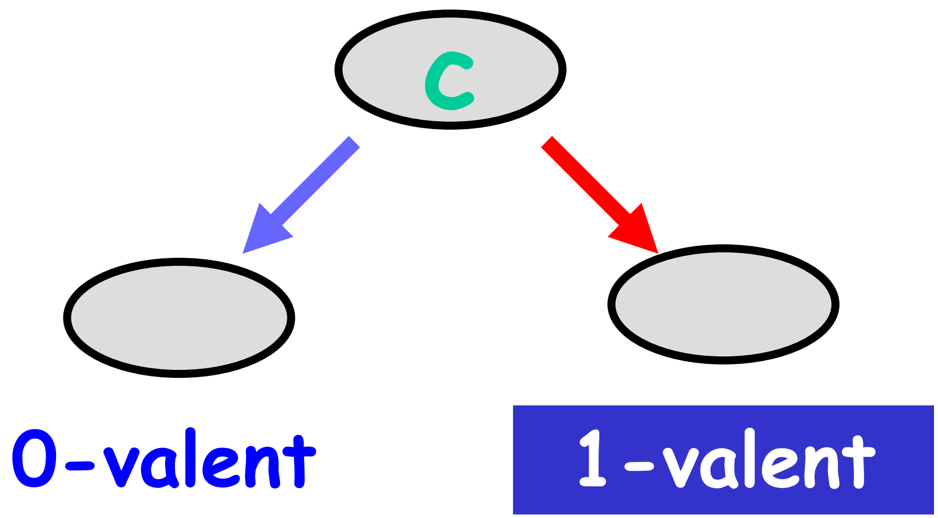
Critical States



Critical States

- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free

From a Critical State



If A goes first, protocol decides 0

If B goes first, protocol decides 1

Model Dependency

- So far, memory-independent!
- True for
  - Registers
  - Message-passing
  - Carrier pigeons
  - Any kind of asynchronous computation

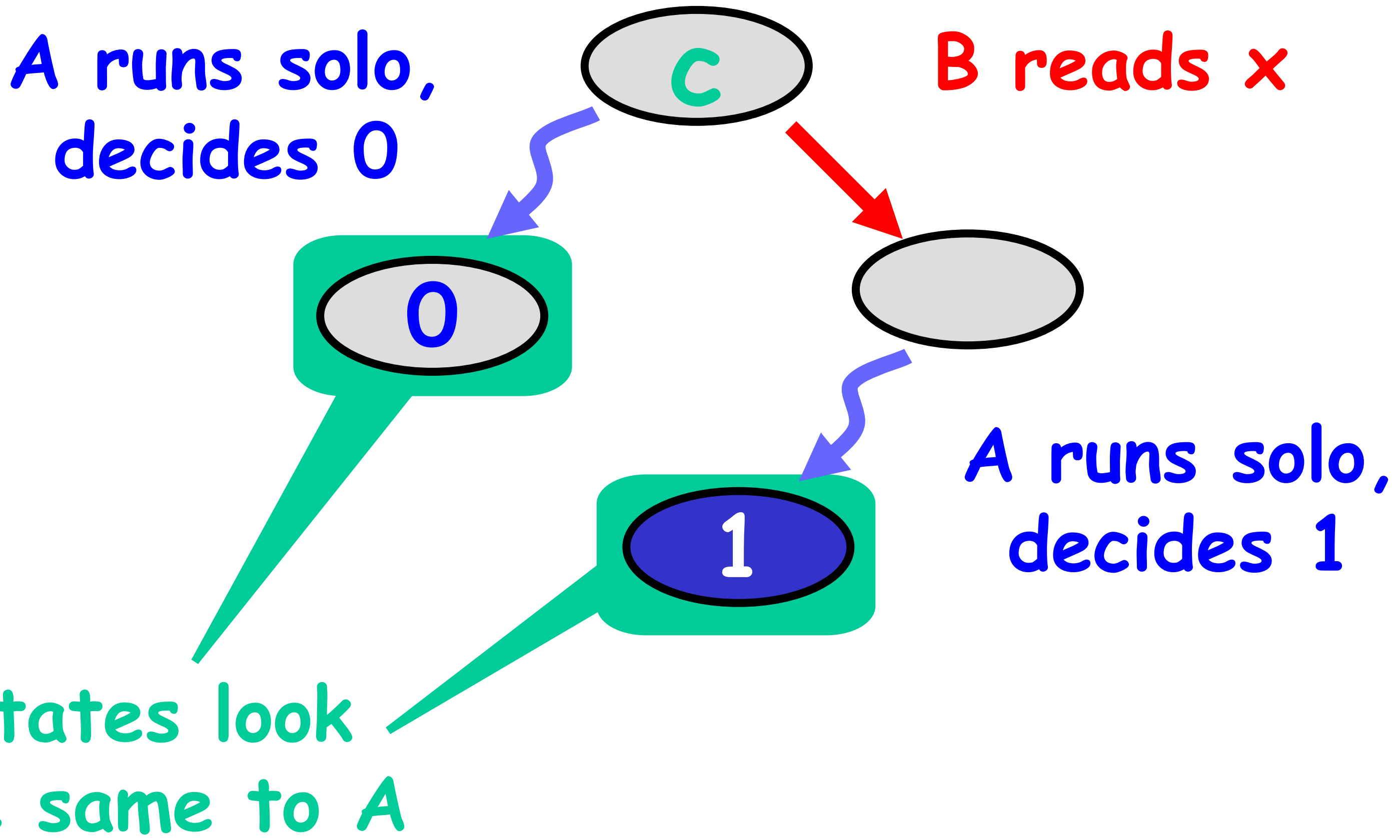
What are the Threads Doing?

- Reads and/or writes
- To same/different registers

Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	?	?	?	?
y.read()	?	?	?	?
x.write()	?	?	?	?
y.write()	?	?	?	?

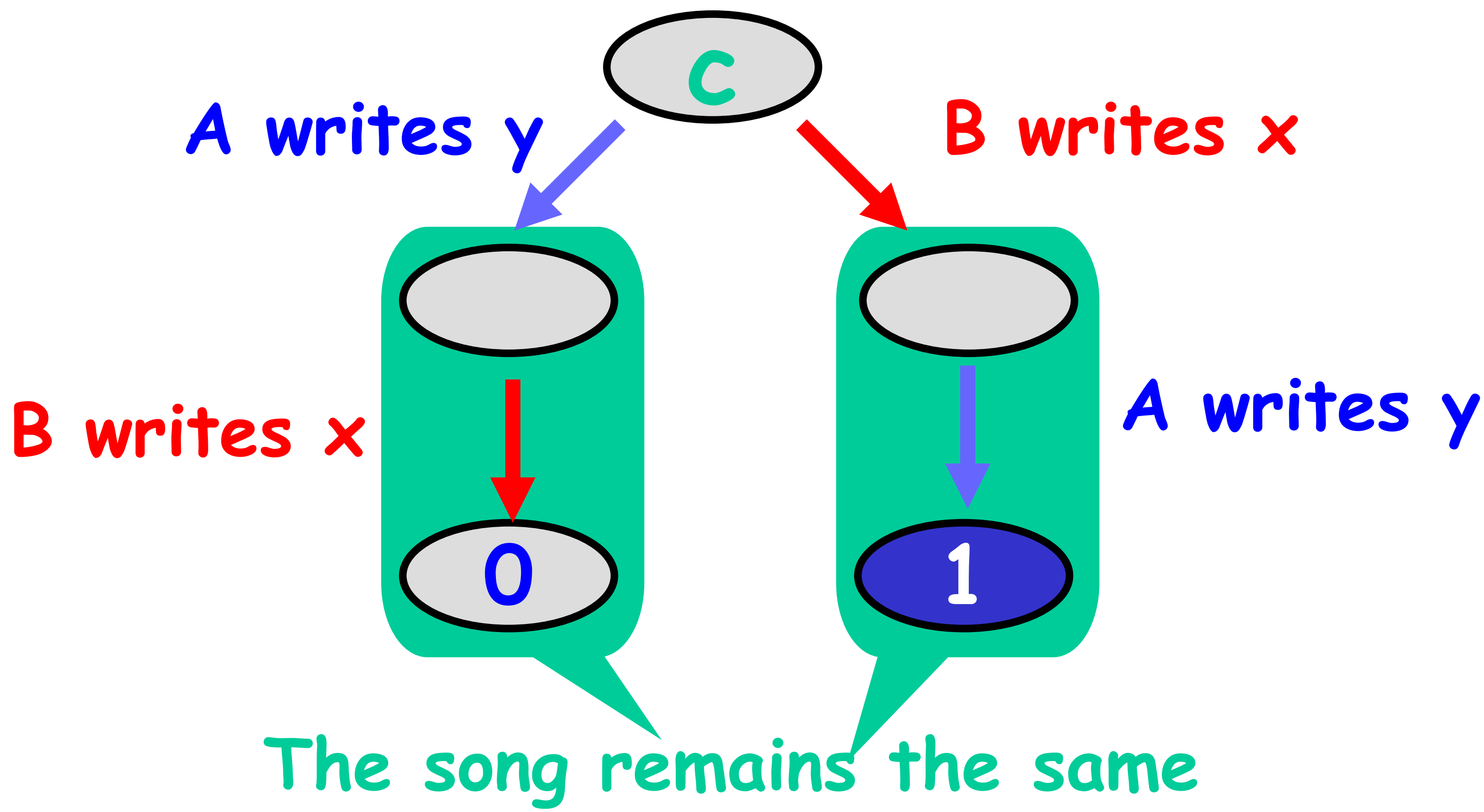
Reading Registers



Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	?
y.write()	no	no	?	?

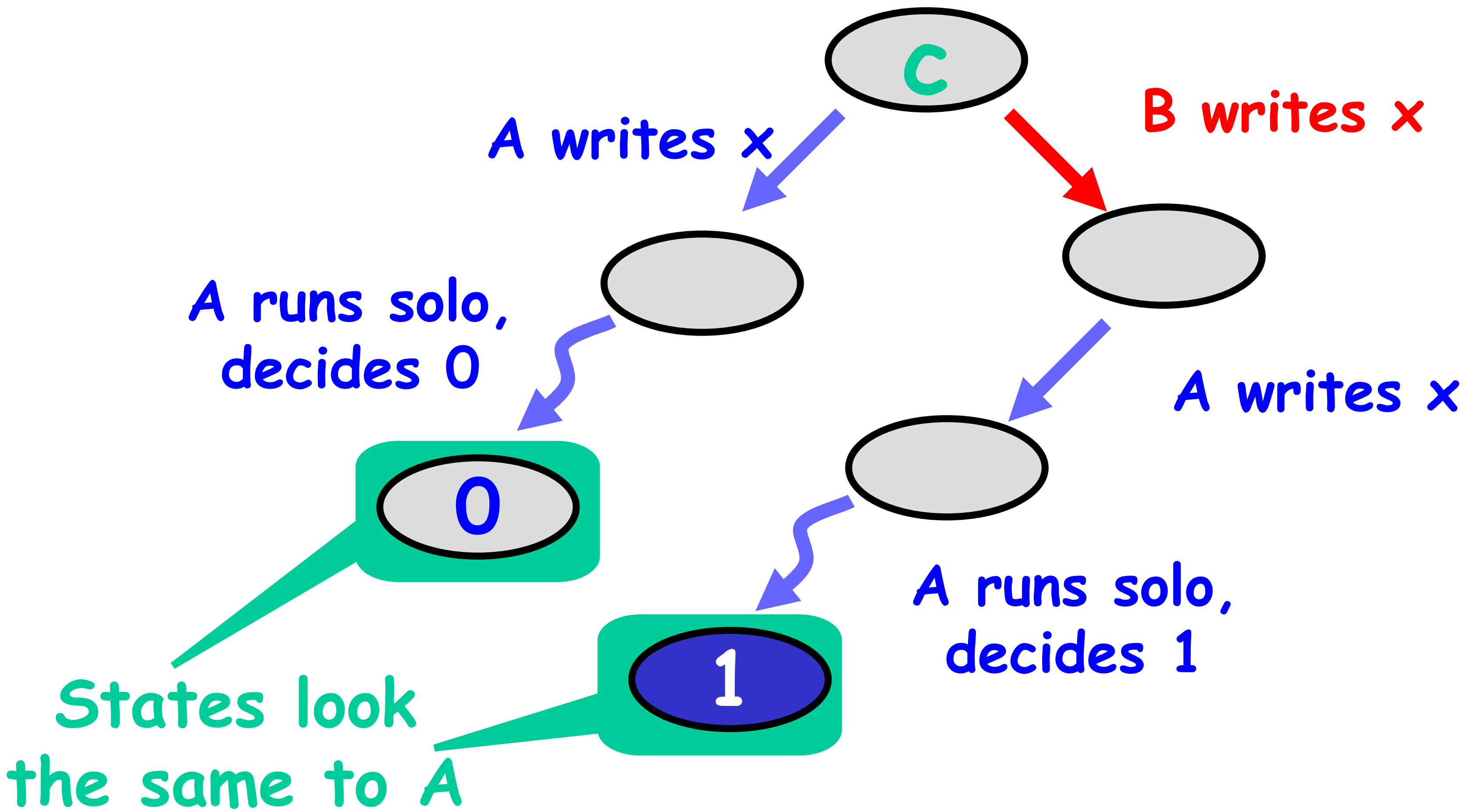
Writing Distinct Registers



Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	no
y.write()	no	no	no	?

Writing Same Registers



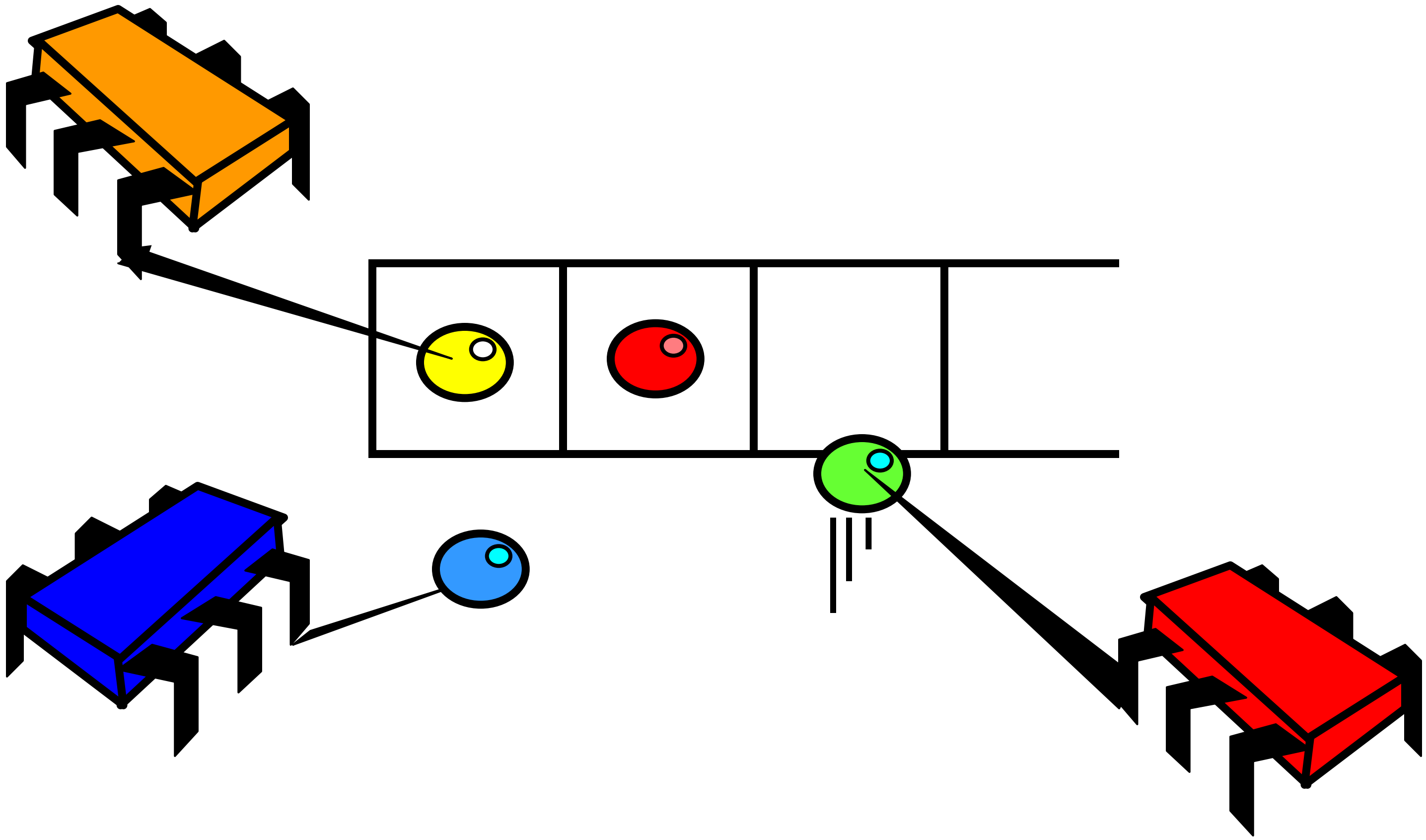
That's All, Folks!

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	no	no
y.write()	no	no	no	no

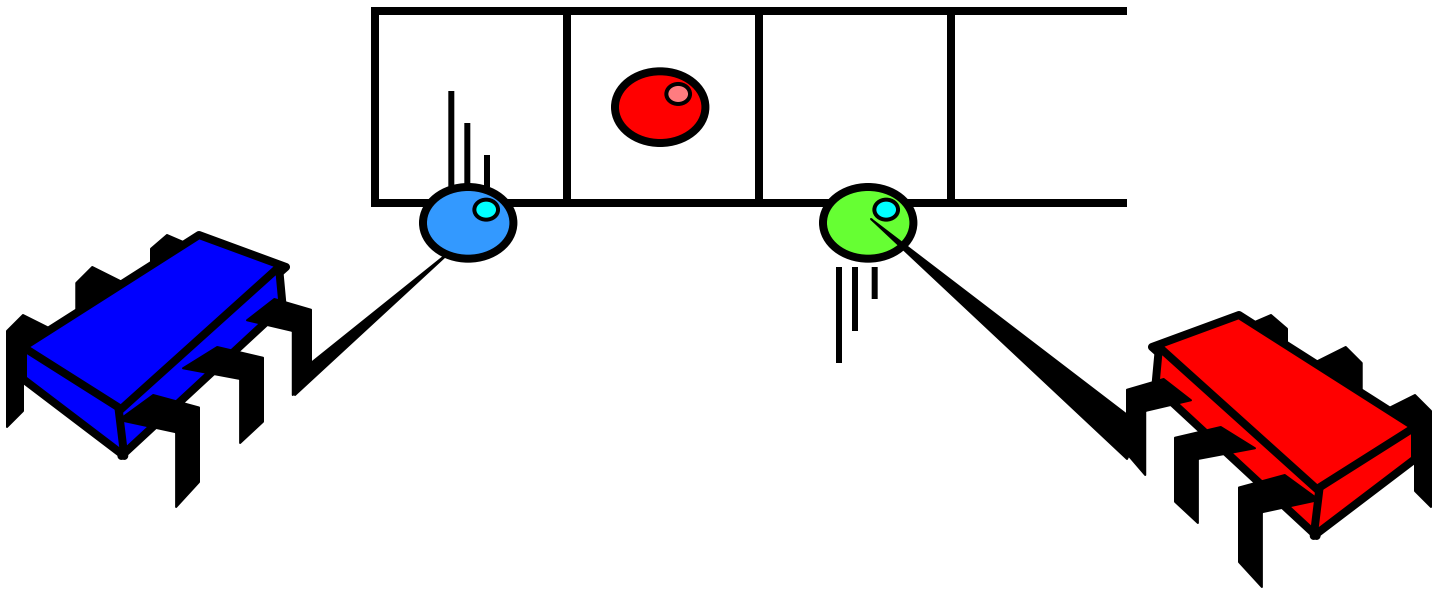
Theorem

- It is impossible to solve consensus using read/write atomic registers
  - Assume protocol exists
  - It has a bivalent initial state
  - Must be able to reach a critical state
  - Case analysis of interactions
    - Reads vs others
    - Writes vs writes

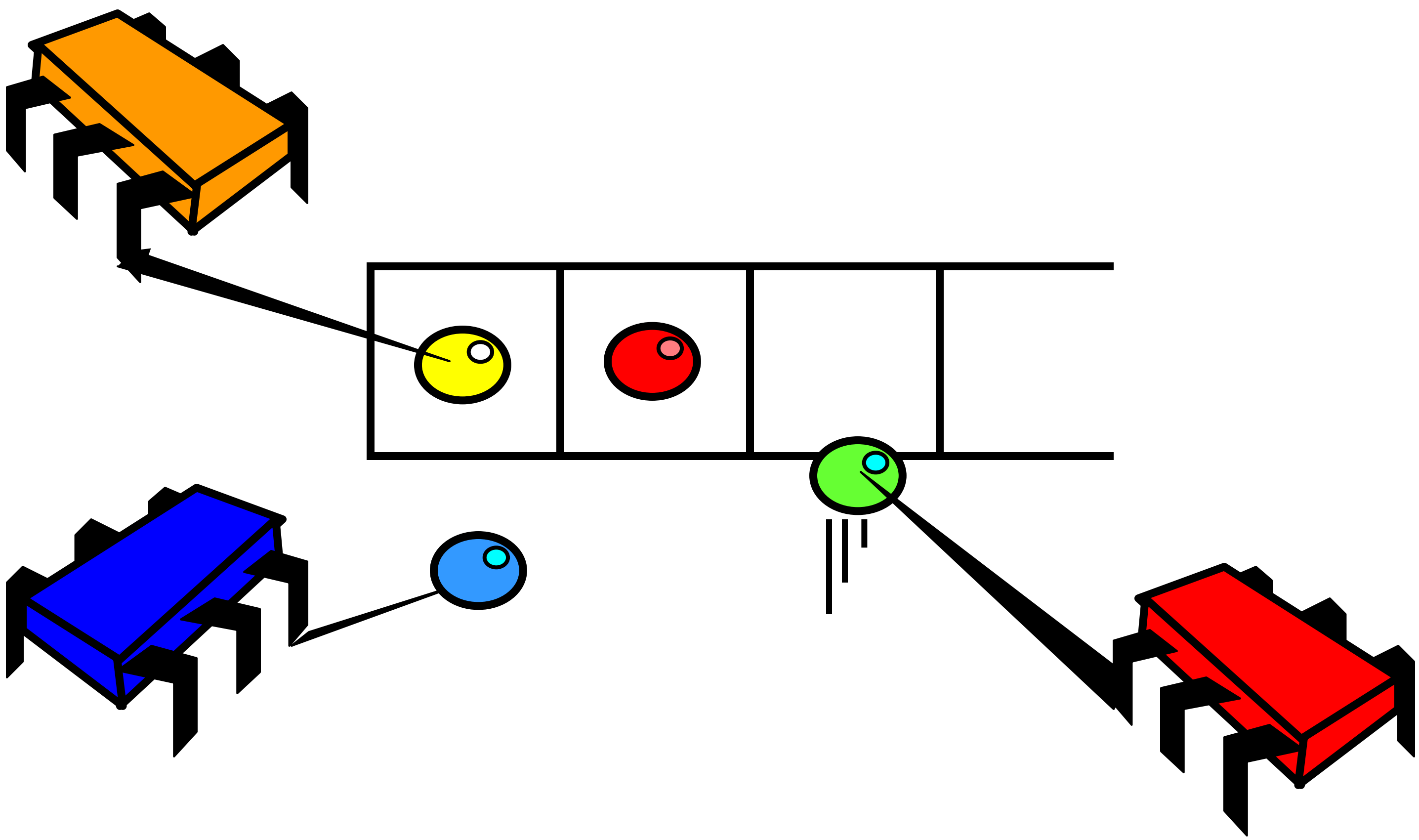
What Does Consensus have to do with Distributed Systems?



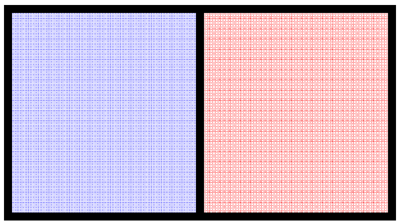
We want to build a Concurrent FIFO Queue



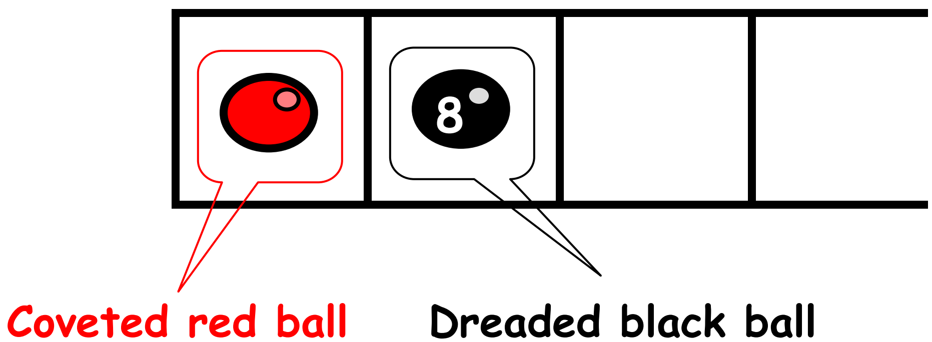
With Multiple Dequeuers!



A Consensus Protocol

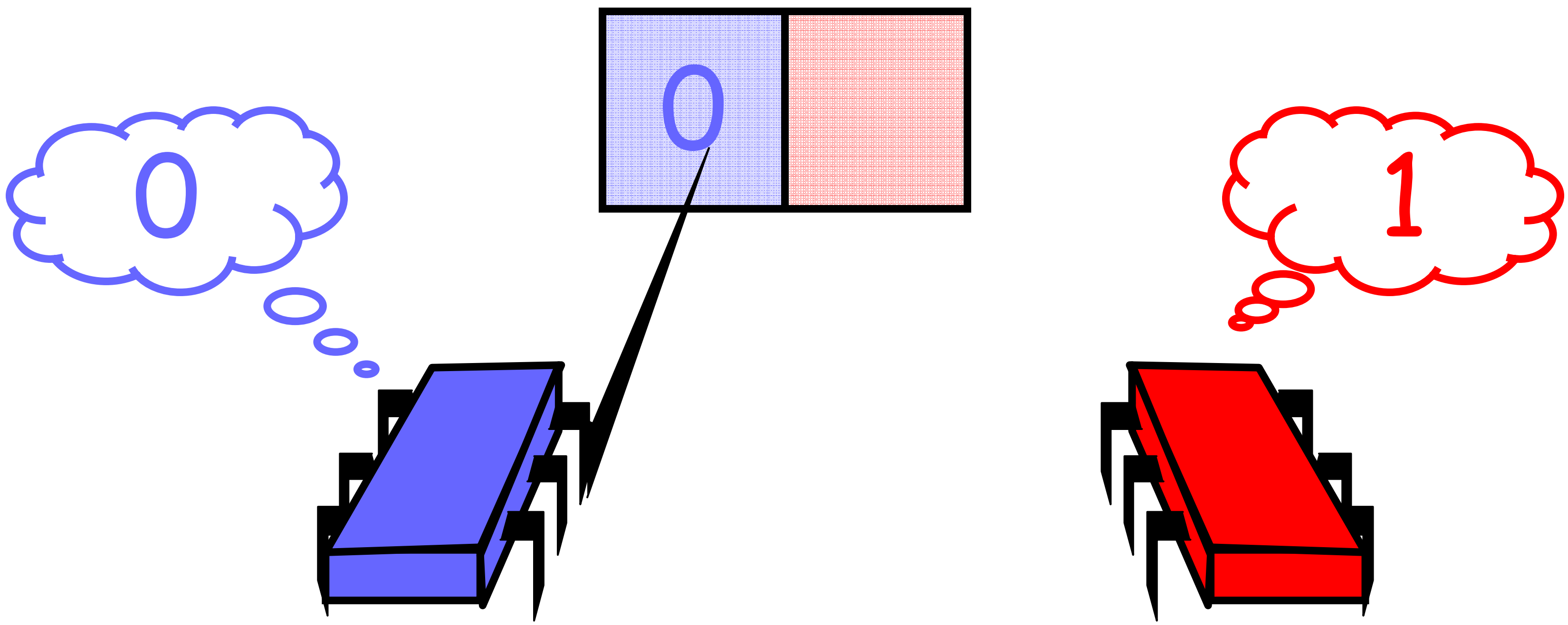


2-element array

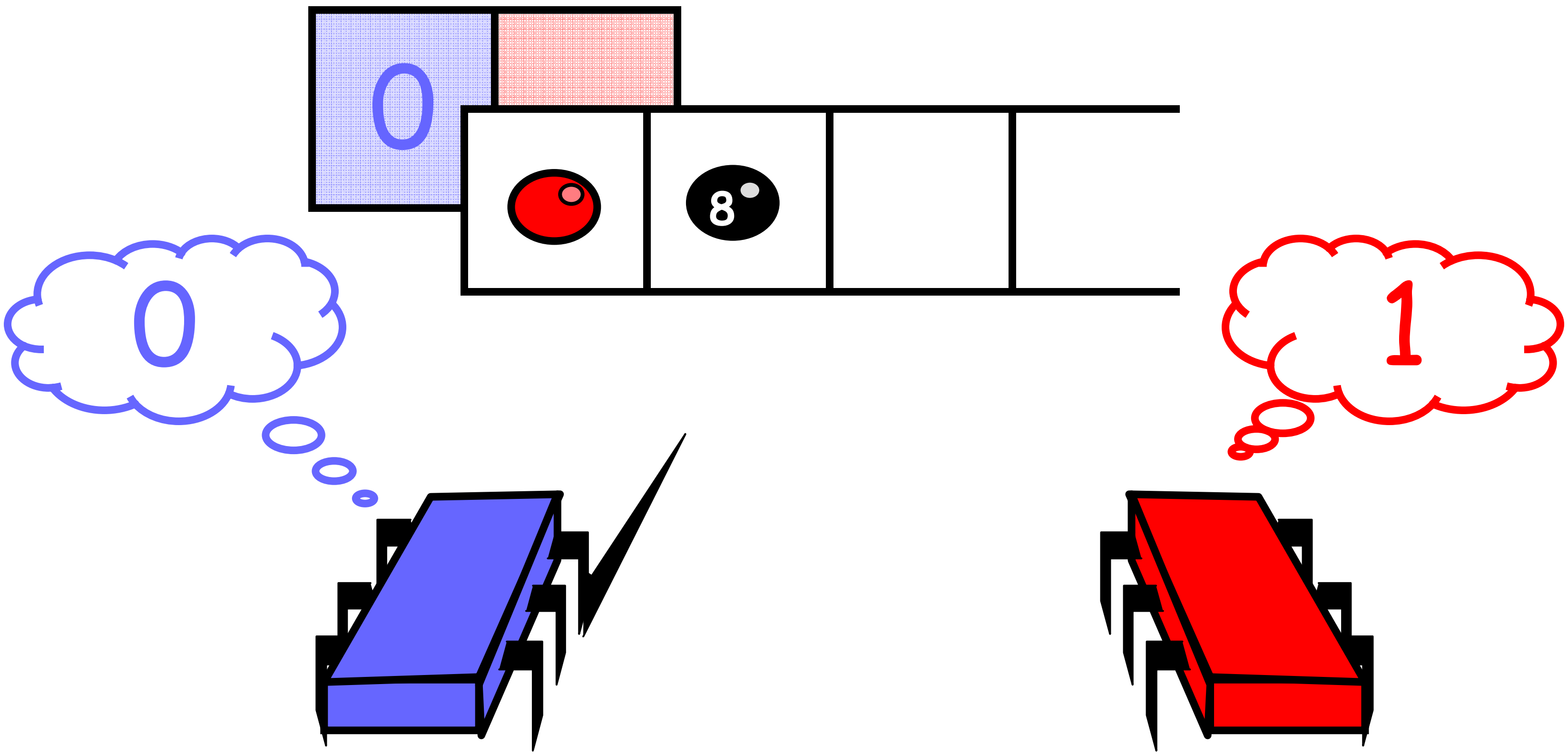


FIFO Queue with red and black balls

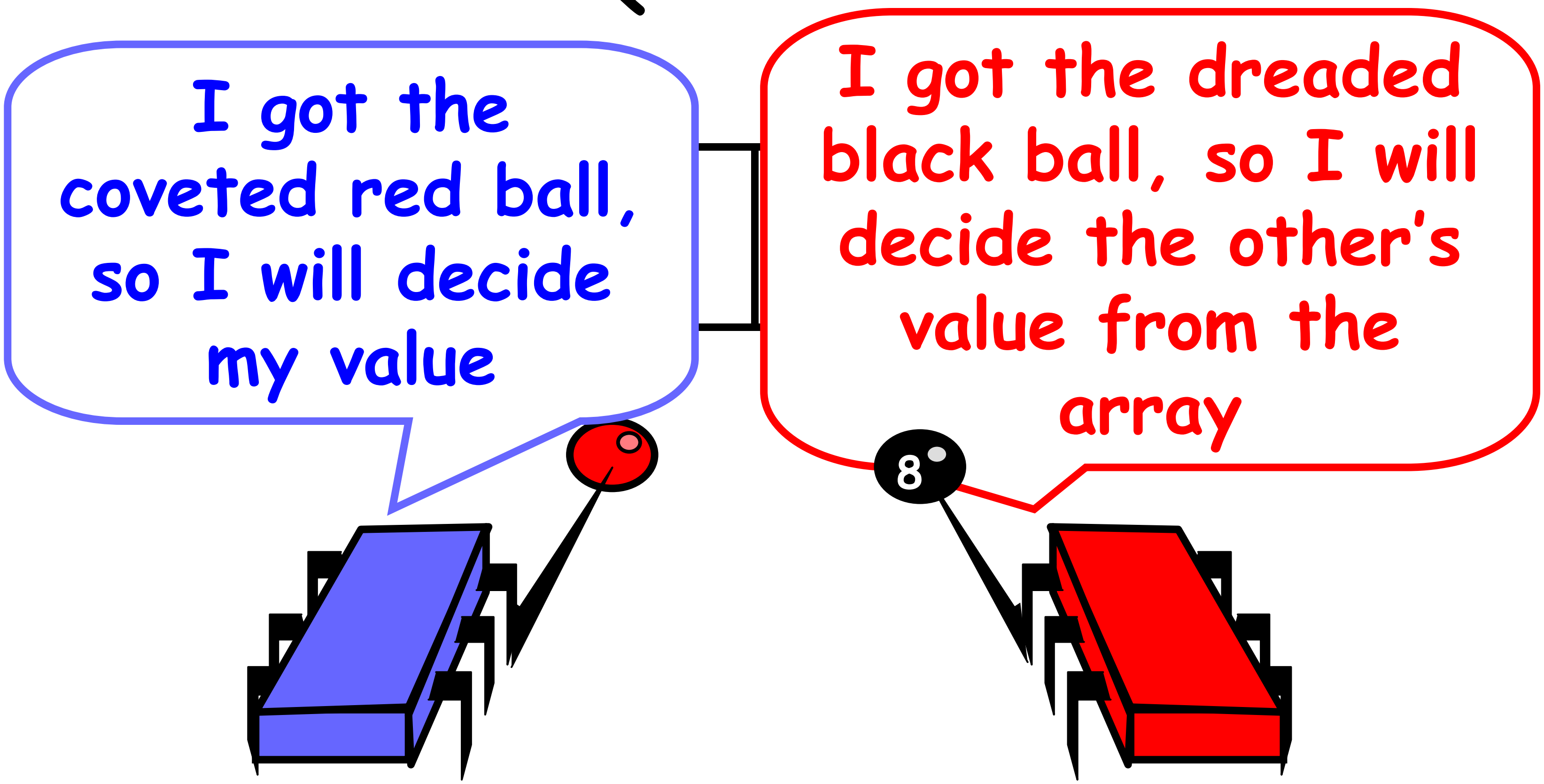
Protocol: Write Value to Array



Protocol: Take Next Item from Queue



Protocol: Take Next Item from Queue



Why does this Work?

- If one thread gets the red ball
- Then the other gets the black ball
- Winner can take her own value
- Loser can find winner's value in array
  - Because threads write array before dequeuing from queue

Implication

- We can solve 2-thread consensus using only
  - A two-dequeuer queue
  - Atomic registers

Implications

- Assume there exists
  - A queue implementation from atomic registers
- Given
  - A consensus protocol from queue and registers
- Substitution yields
  - A wait-free consensus protocol from atomic registers

contradiction

Corollary

- It is impossible to implement a two-dequeuer wait-free FIFO queue with read/write shared memory.
- This was a proof by reduction; important beyond NP-completeness…

Consensus #3
read-modify-write shared mem.

- n processors, with n > 1
- Wait-free implementation
- Processors can atomically read *and* write a shared memory cell in one atomic step: the value written can depend on the value read
- We call this a RMW register

Protocol

- There is a cell c, initially c="?"
- Every processor i does the following

```
RMW(c), with
  if (c == "?") then
       Write(c, v_i); decide v_i;
  else
       decide c;
```

atomic step

Discussion

- Protocol works correctly
  - One processor accesses c as the first; this processor will determine decision
- Protocol is wait-free
- RMW is quite a strong primitive
  - Can we achieve the same with a weaker primitive?

Read-Modify-Write more formally

- Method takes 2 arguments:
  - Variable x
  - Function f
- Method call:
  - Returns value of x
  - Replaces x with f(x)

Read-Modify-Write

```
public abstract class RMW {
  private int value;

  public void rmw(function f) {
    int prior  = this.value;
    this.value = f(this.value);
    return prior;
  }

}
```

Return prior value

Apply function

Example: Read

```
public abstract class RMW {
  private int value;

  public void read() {
    int prior  = this.value;
    this.value = this.value;
    return prior;
  }

}
```

identity function

Example: test&set

```
public abstract class RMW {
  private int value;

  public void TAS() {
    int prior  = this.value;
    this.value = 1;
    return prior;
  }

}
```

constant function

Example: fetch&inc

```
public abstract class RMW {
  private int value;

  public void fai() {
    int prior  = this.value;
    this.value = this.value+1;
    return prior;
  }

}
```

increment function

Example: fetch&add

```
public abstract class RMW {
  private int value;

  public void faa(int x) {
    int prior  = this.value;
    this.value = this.value+x;
    return prior;
  }

}
```

addition function

Example: swap

```
public abstract class RMW {
  private int value;

  public void swap(int x) {
    int prior  = this.value;
    this.value = x;
    return prior;
  }

}
```

constant function

Example: compare&swap

```
public abstract class RMW {
  private int value;

  public void CAS(int old, int new) {
    int prior = this.value;
    if (this.value == old)
      this.value = new;
    return prior;
  }

}
```

complex function

“Non-trivial” RMW

- Not simply read
- But
  - test&set, fetch&inc, fetch&add, swap, compare&swap, general RMW
- Definition: A RMW is non-trivial if there exists a value v such that $v \neq f(v)$

Consensus Numbers (Herlihy)

- An object has consensus number n
  - If it can be used
    - Together with atomic read/write registers
  - To implement n-thread consensus
    - But not (n+1)-thread consensus

Consensus Numbers

- Theorem
  - Atomic read/write registers have consensus number 1

- Proof
  - Works with 1 process
  - We have shown impossibility with 2

Consensus Numbers

- Consensus numbers are a useful way of measuring synchronization power

- Theorem
  - If you can implement X from Y
  - And X has consensus number c
  - Then Y has consensus number at least c

Synchronization Speed Limit

- Conversely
  - If X has consensus number c
  - And Y has consensus number $d < c$
  - Then there is no way to construct a wait-free implementation of X by Y
- This theorem will be very useful
  - Unforeseen practical implications!

Theorem

- Any non-trivial RMW object has consensus number at least 2
- Implies no wait-free implementation of RMW registers from read/write registers
- Hardware RMW instructions not just a convenience

Proof

```
public class RMWConsensusFor2
    implements Consensus {
  private RMW r;

  public Object decide() {
    int i = Thread.myIndex();
    if (r.rmw(f) == v)
      return this.announce[i];
    else
      return this.announce[1-i];
  }}
```

Initialized to v

Am I first?

Yes, return my input

No, return other's input

Proof

- We have displayed
  - A two-thread consensus protocol
  - Using any non-trivial RMW object

Interfering RMW

- Let F be a set of functions such that for all f_i and f_j, either
  - They commute: $f_i(f_j(x))=f_j(f_i(x))$
  - They overwrite: $f_i(f_j(x))=f_i(x)$
- Claim: Any such set of RMW objects has consensus number exactly 2

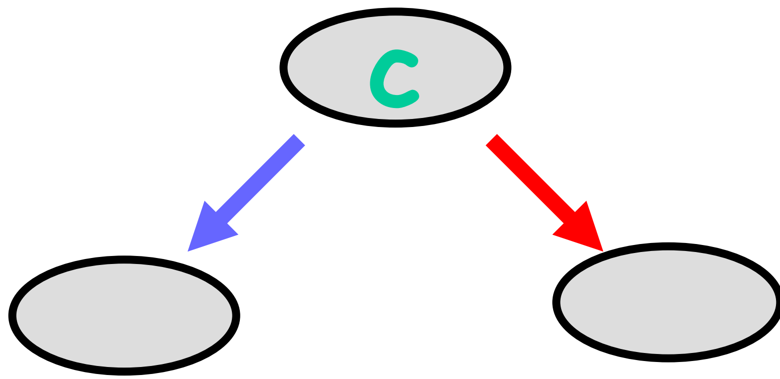
Examples

- Test-and-Set
  - Overwrite
- Swap
  - Overwrite
- Fetch-and-inc
  - Commute

Meanwhile Back at the Critical State

A about to apply f_A

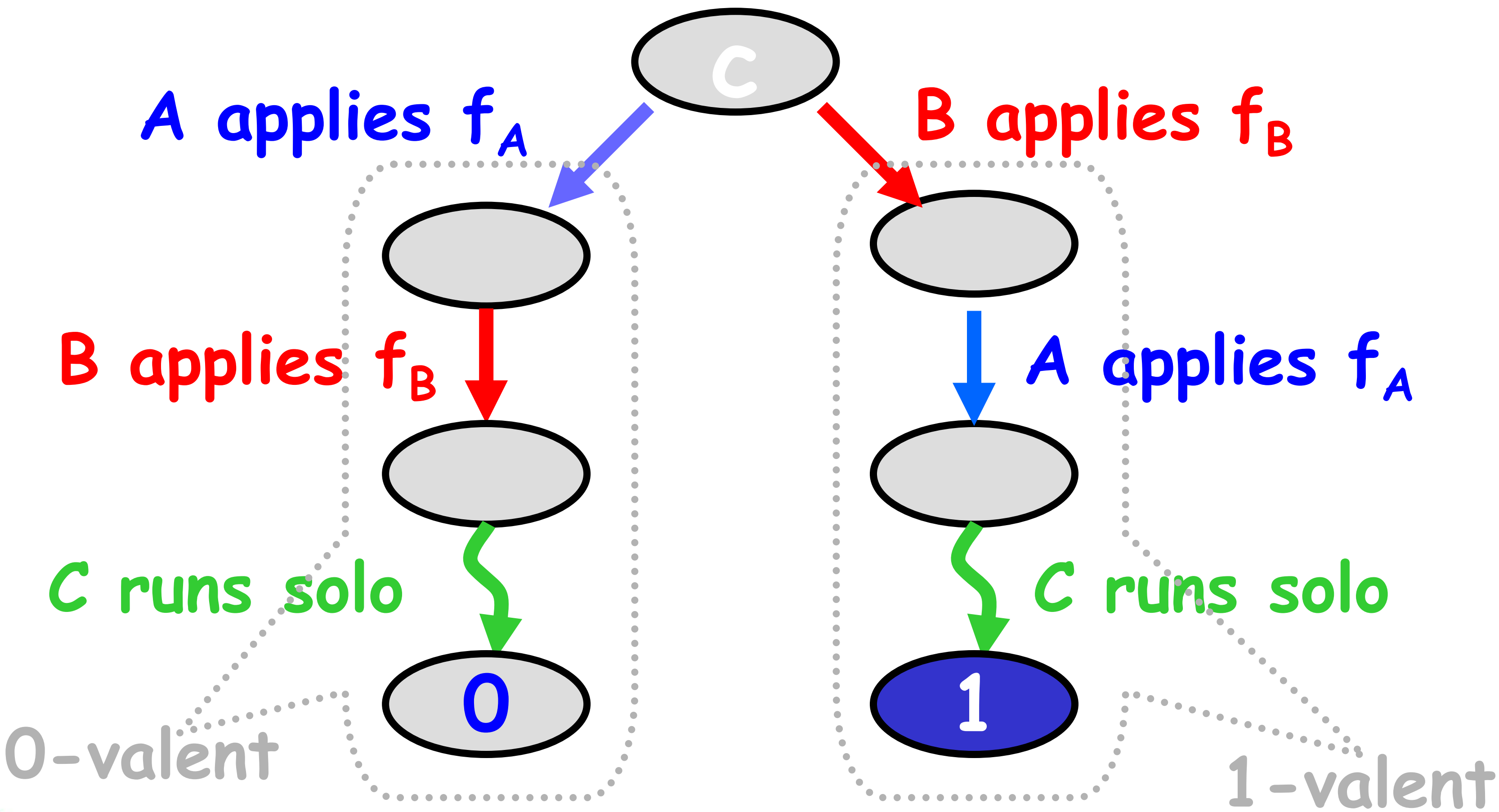
B about to apply f_B



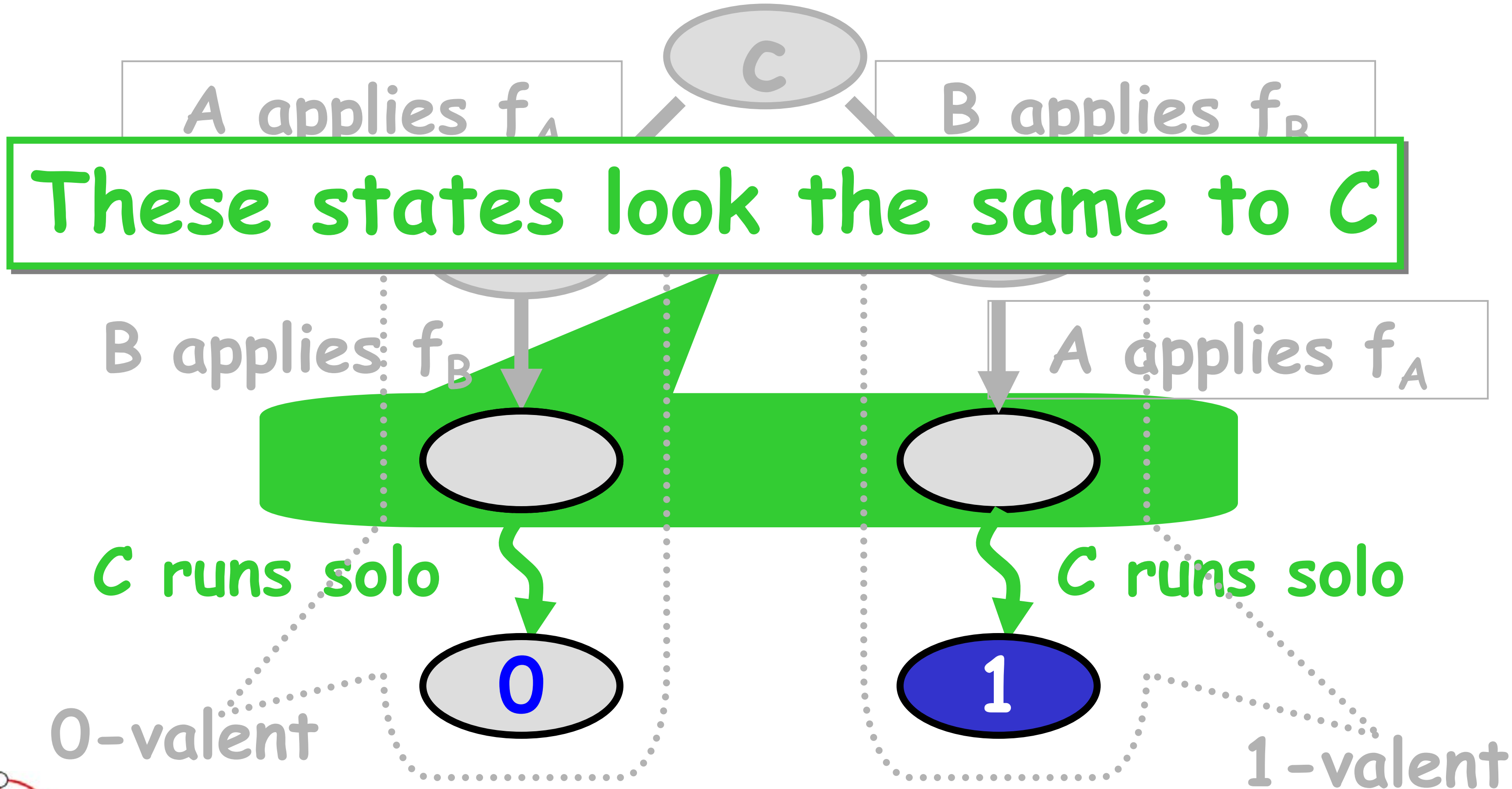
0-valent

1-valent

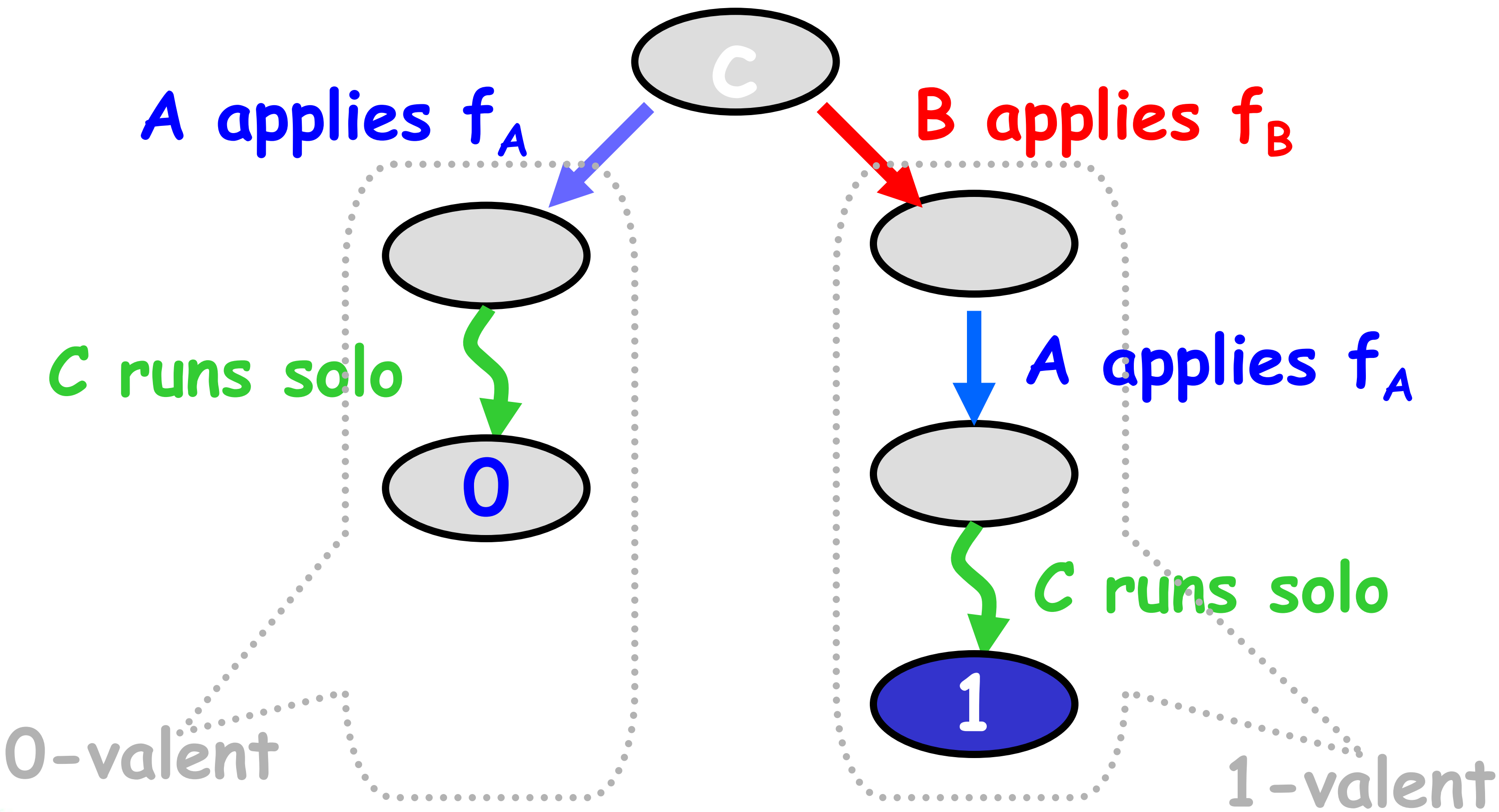
Maybe the Functions Commute



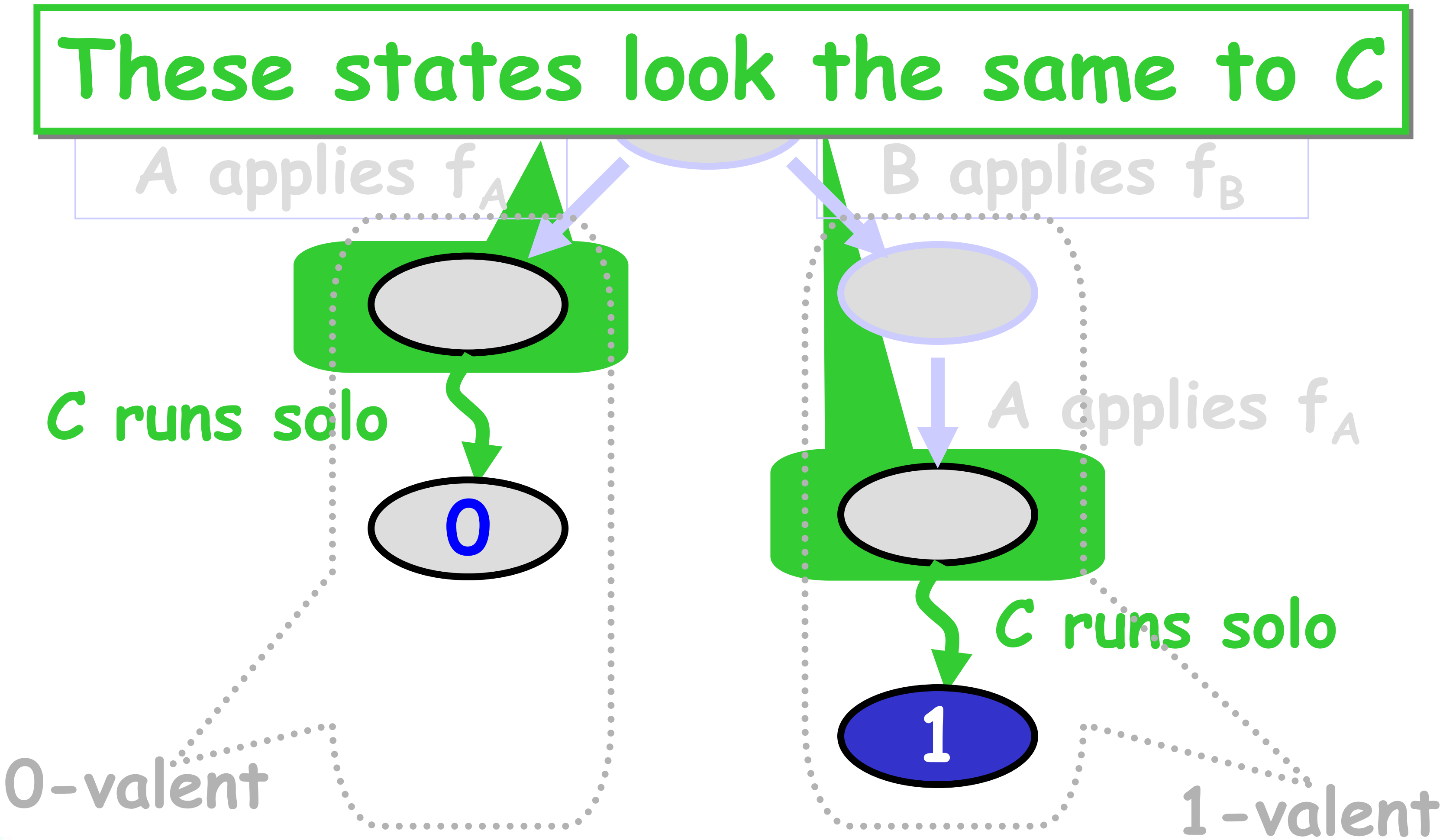
Maybe the Functions Commute



Maybe the Functions Overwrite



Maybe the Functions Overwrite



Impact

- Many early machines used these "weak" RMW instructions
  - Test-and-set (IBM 360)
  - Fetch-and-add (NYU Ultracomputer)
  - Swap
- We now understand their limitations
  - But why do we want consensus anyway?

CAS has Unbounded Consensus Number

```
public class RMWConsensus
    implements Consensus {
  private RMW r;

  public Object decide() {
    int i = Thread.myIndex();
    int j = r.CAS(-1,i);
    if (j == -1)
      return this.announce[i];
    else
      return this.announce[j];
  }}
```

Initialized to -1

Am I first?

Yes, return my input

No, return other's input

The Consensus Hierarchy

1	Read/Write Registers, …
2	T&S, F&I, Swap, …
.	
.	
.	
∞	CAS, …

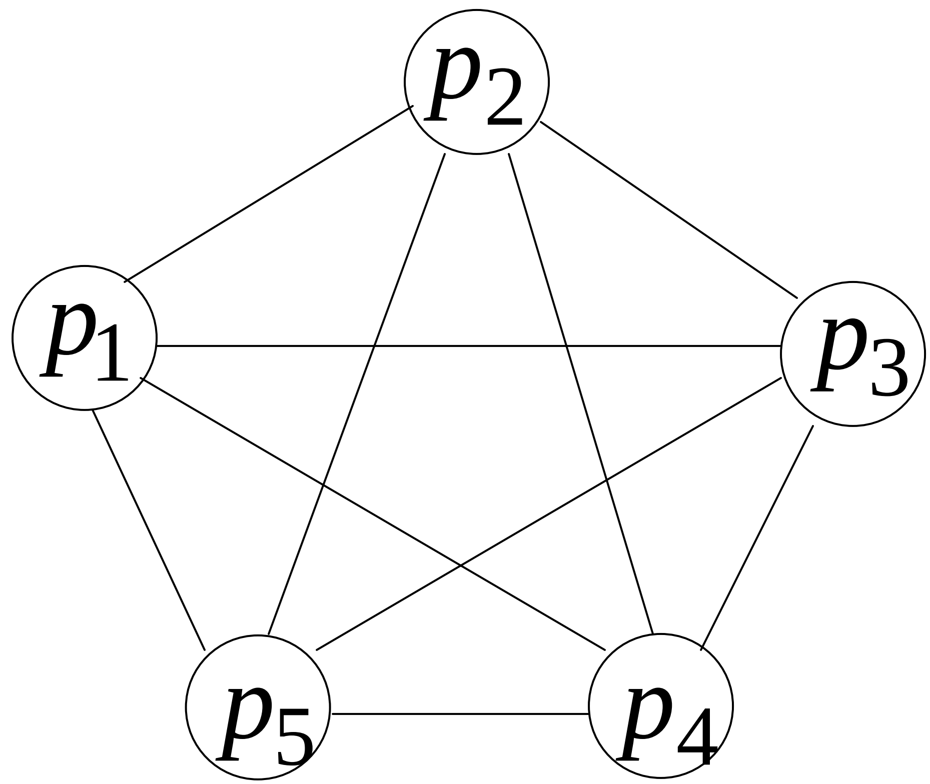
Consensus #4
Synchronous Systems

- In real systems, one can sometimes tell if a processor had crashed
  - Timeouts
  - Broken TCP connections

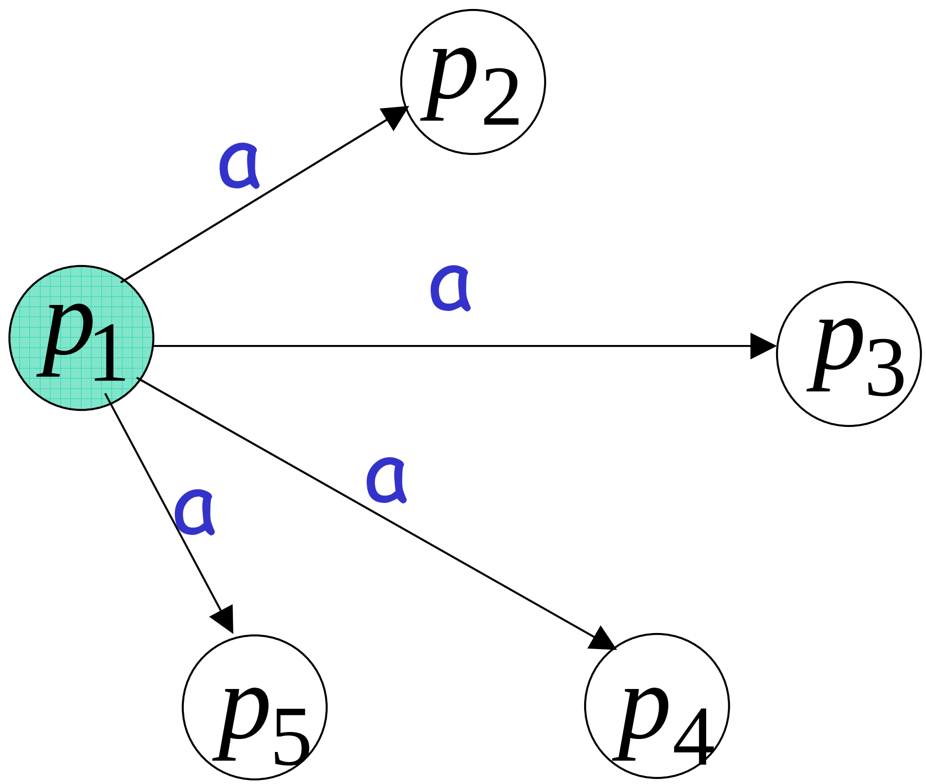
- Can one solve consensus at least in synchronous systems?

Communication Model

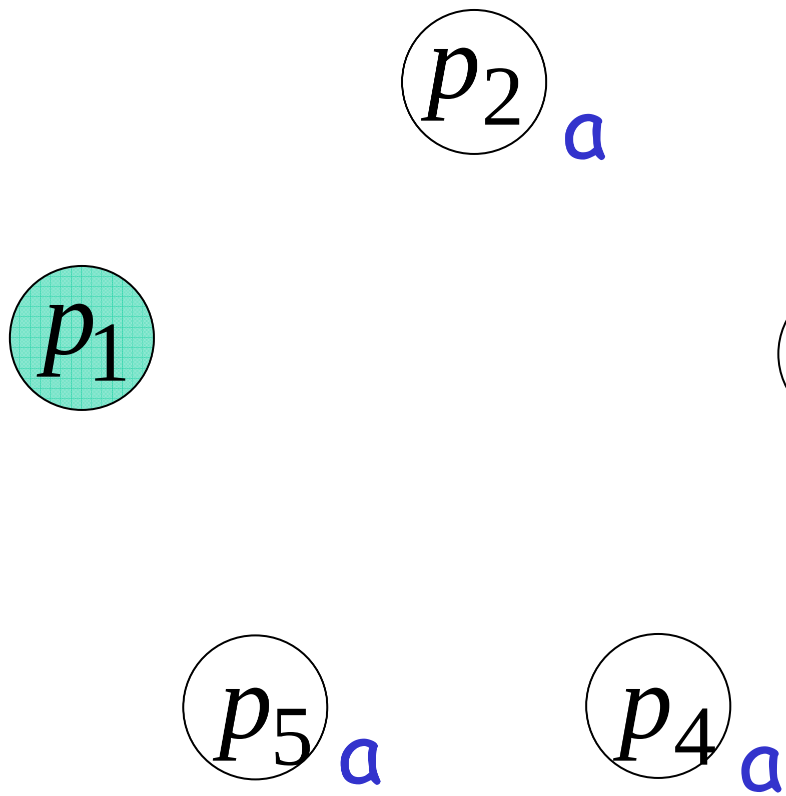
- Complete graph
- Synchronous



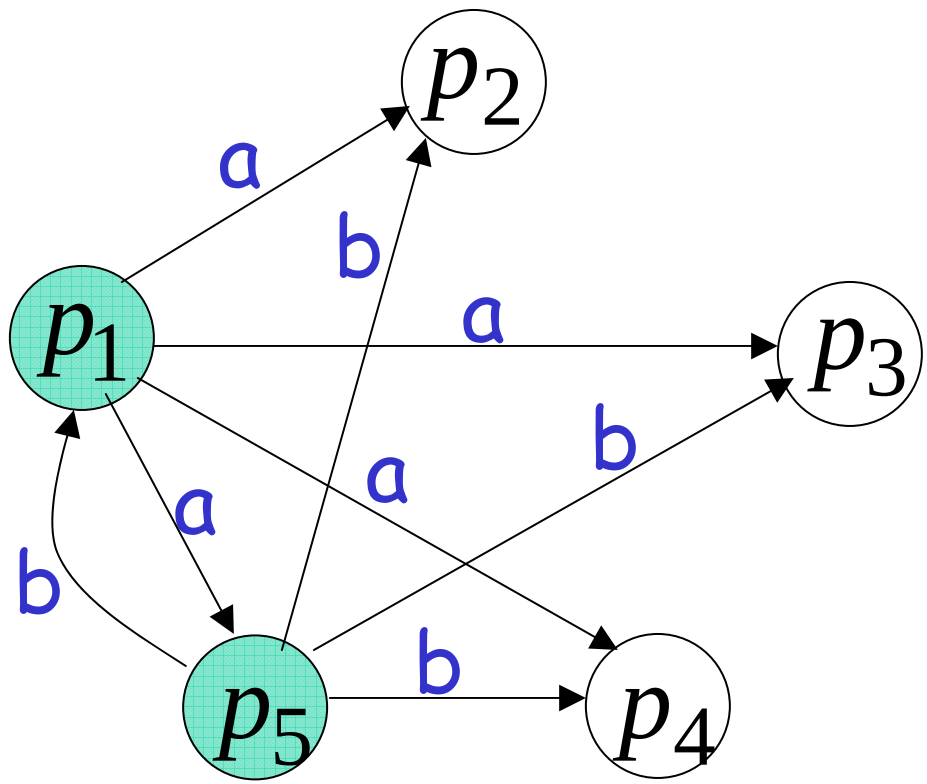
Send a message to all processors in one round: Broadcast



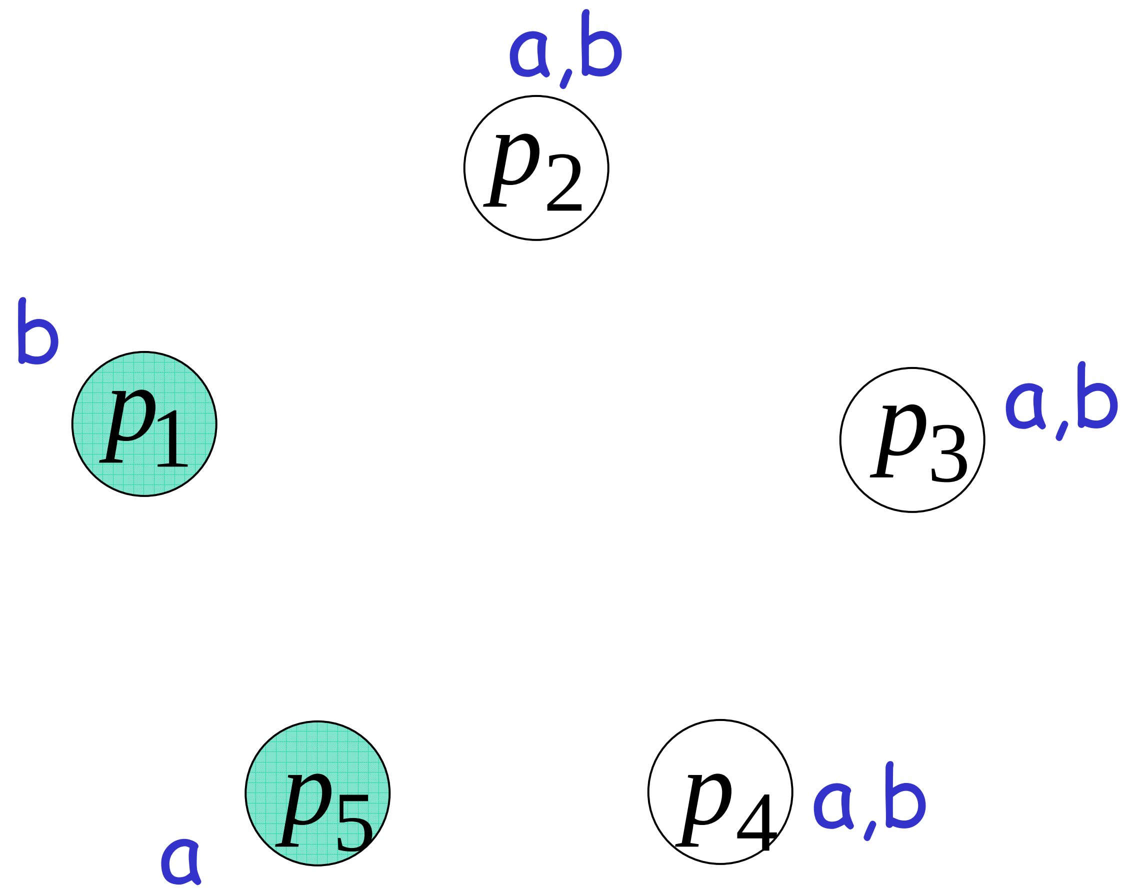
At the end of the round: everybody receives a



Broadcast: Two or more processes can broadcast in the same round

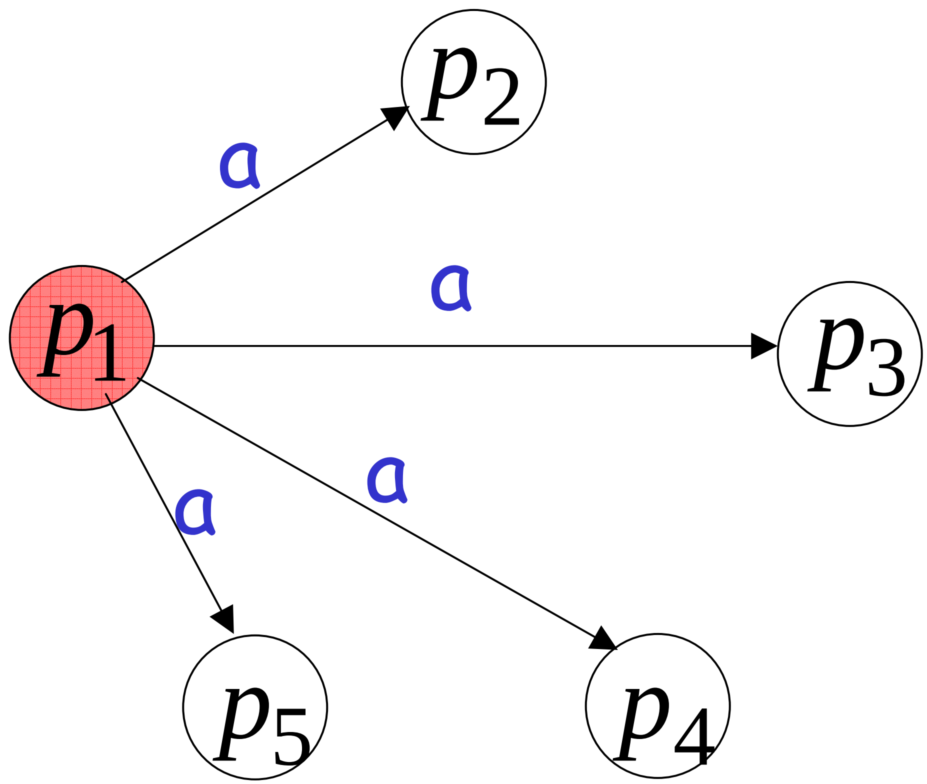


At end of round...

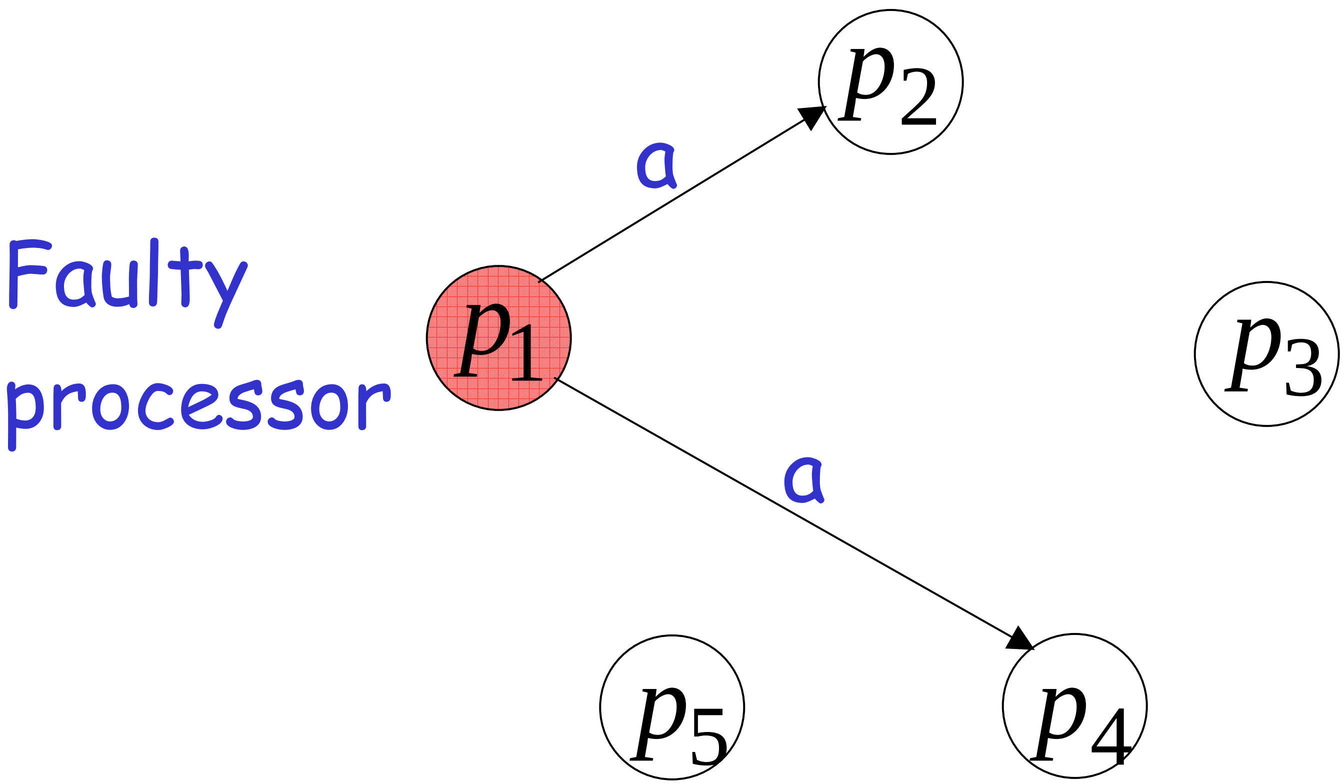


Crash Failures

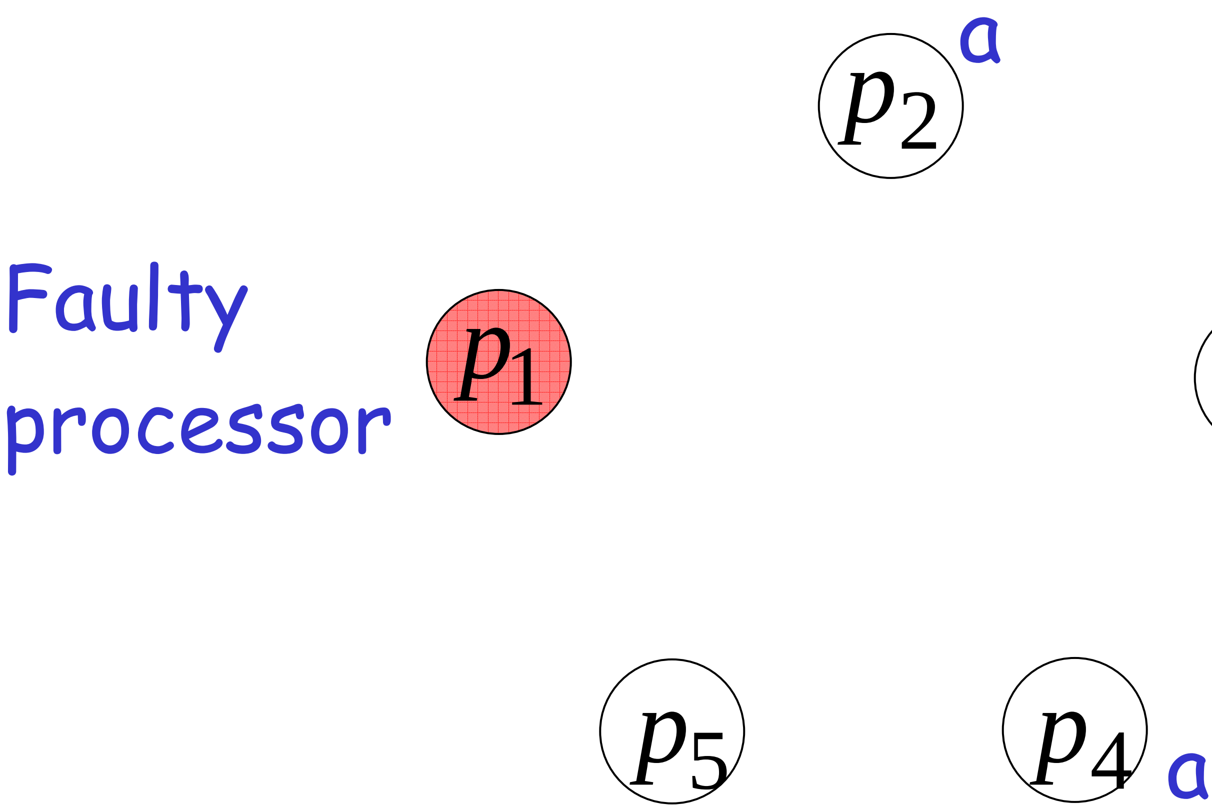
Faulty processor



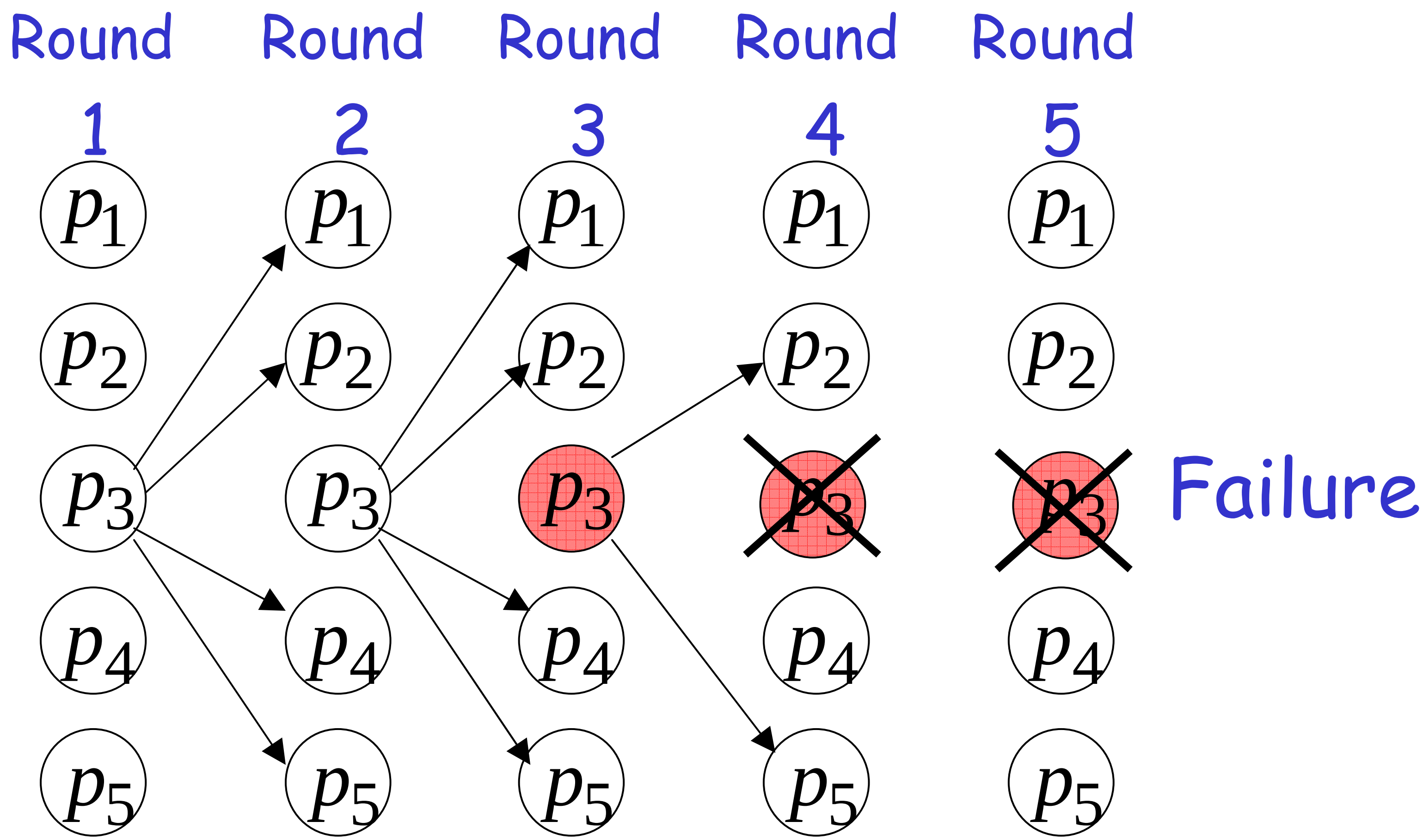
Some of the messages are lost, they are never received



Effect

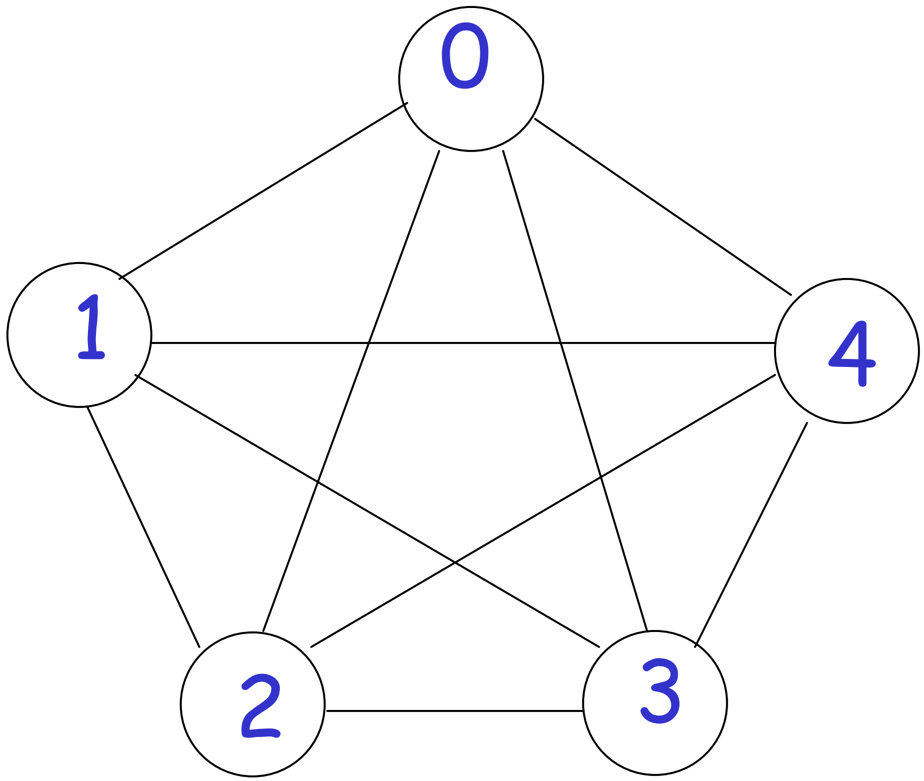


After a failure, the process disappears from the network



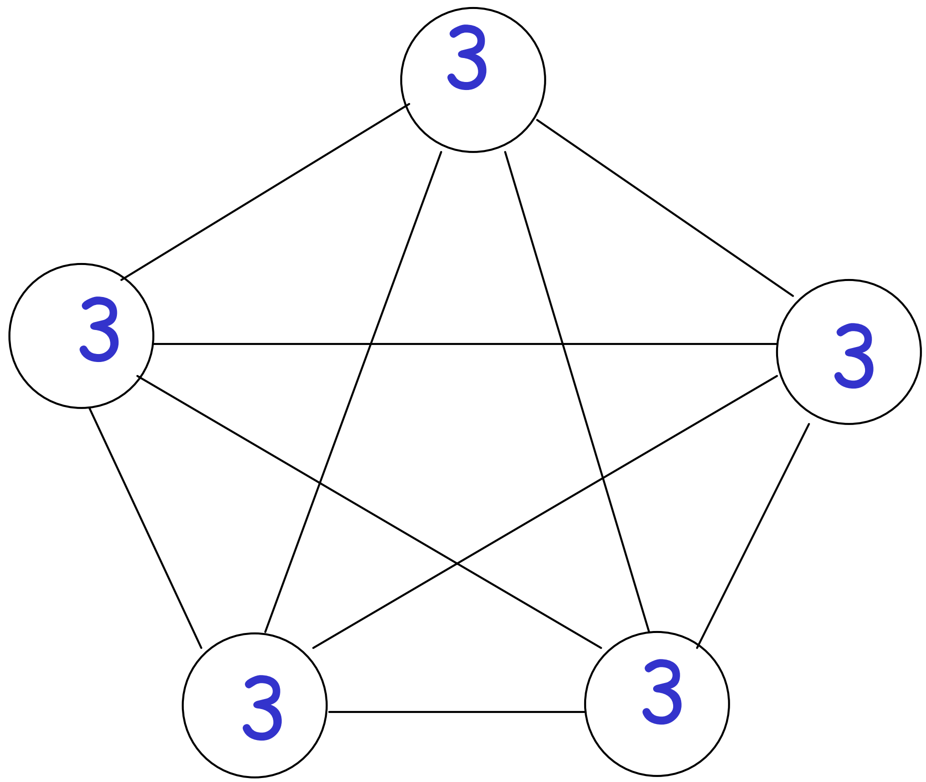
Consensus: Everybody has an initial value

Start



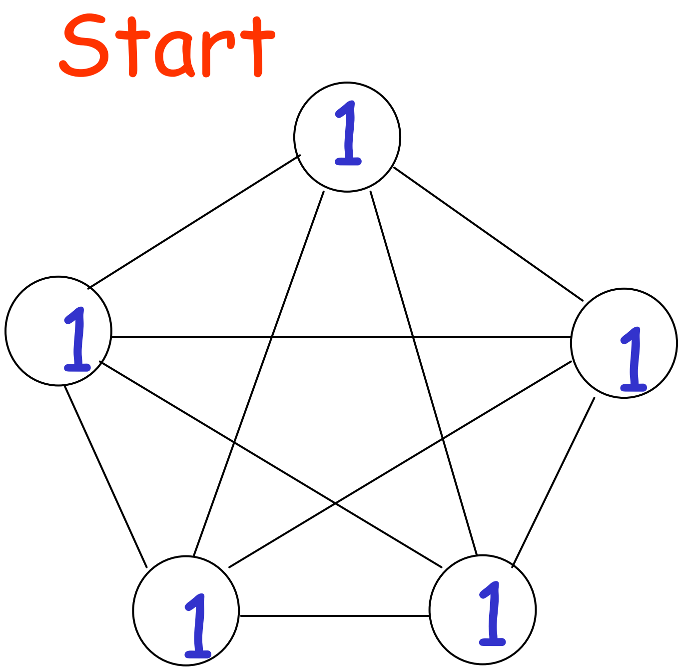
Everybody must decide on the same value

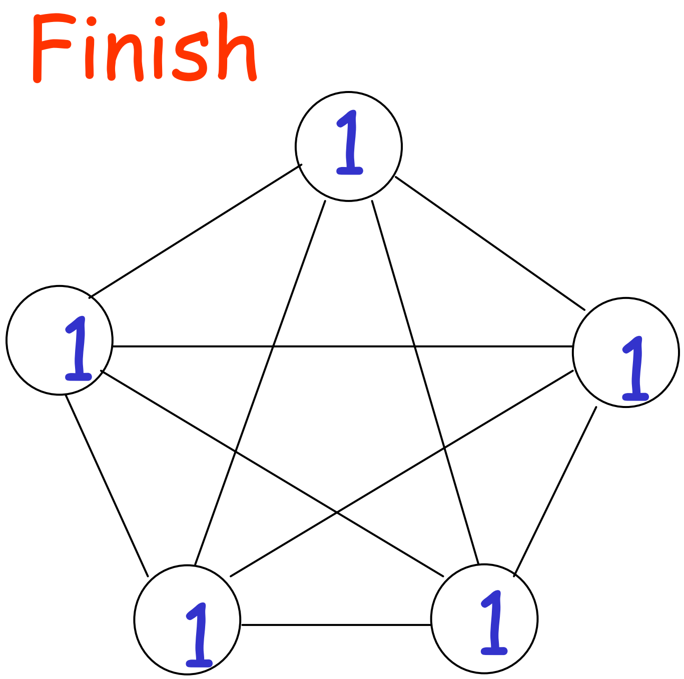
Finish



Validity condition:

If everybody starts with the same value they must decide on that value





A simple algorithm

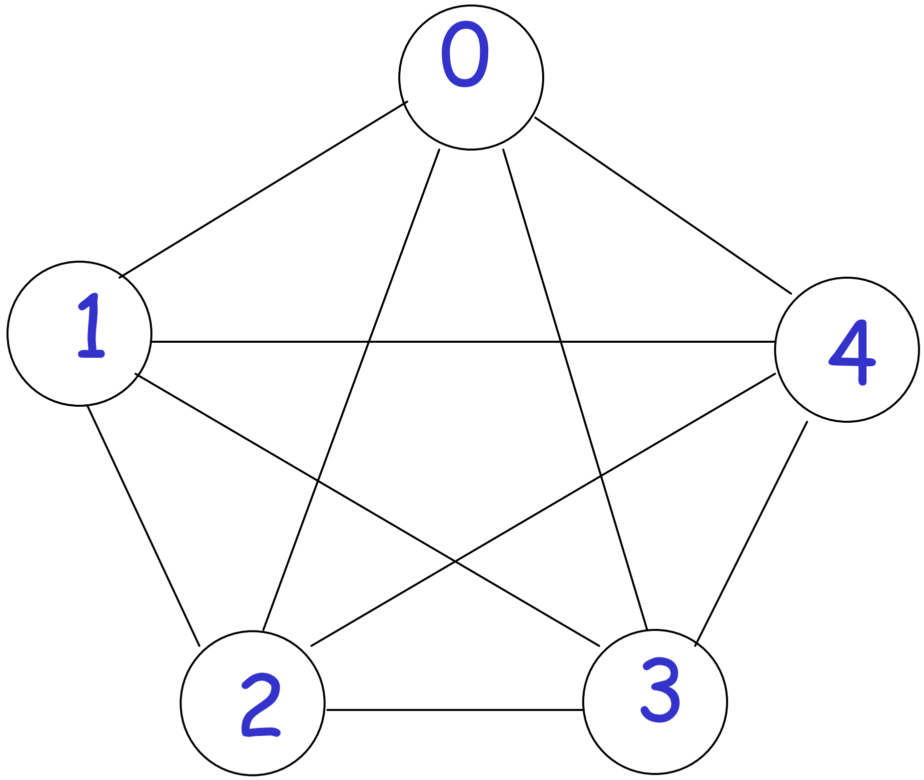
Each processor:

1. Broadcasts value to all processors

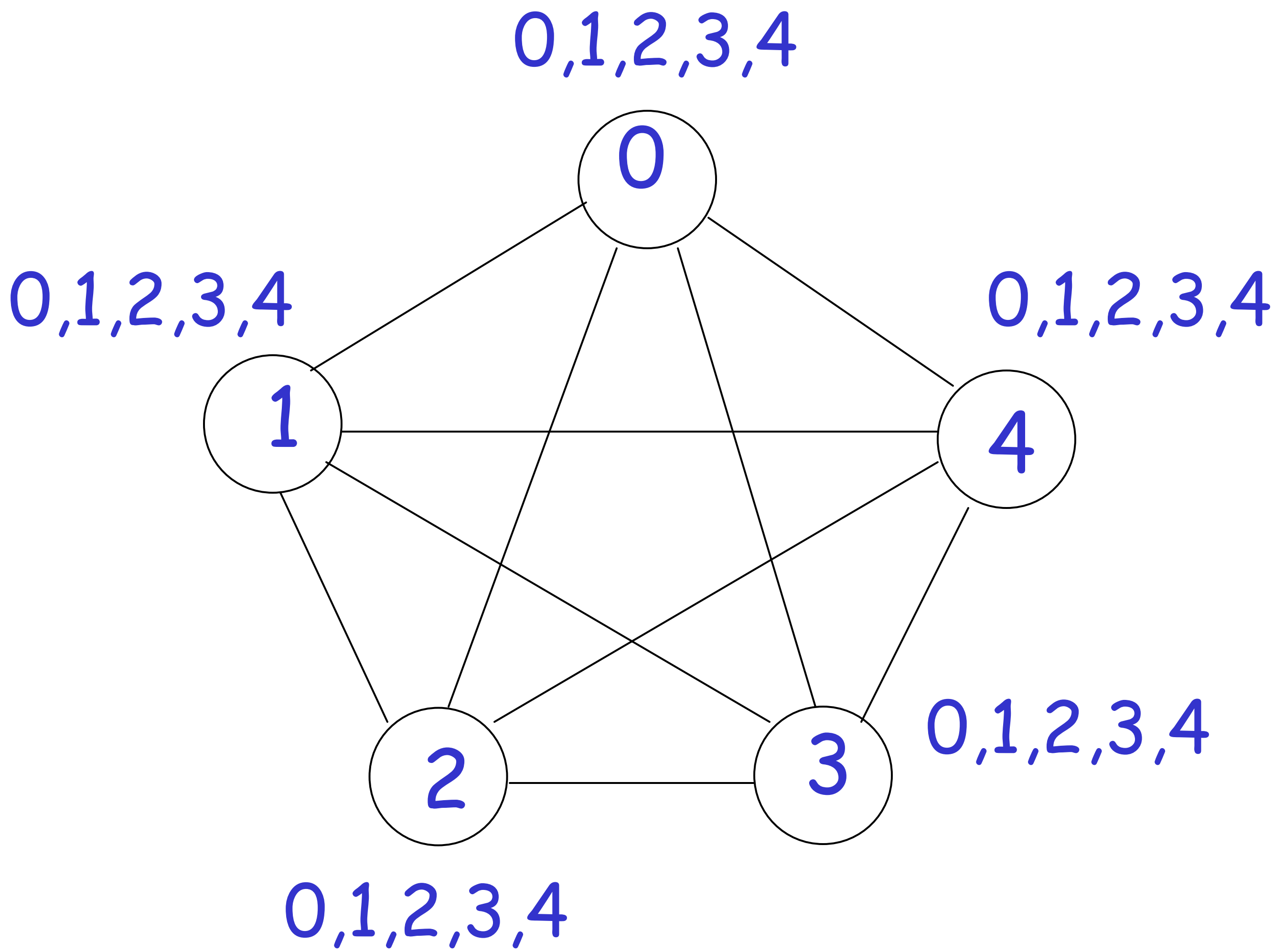
2. Decides on the minimum

(only one round is needed)

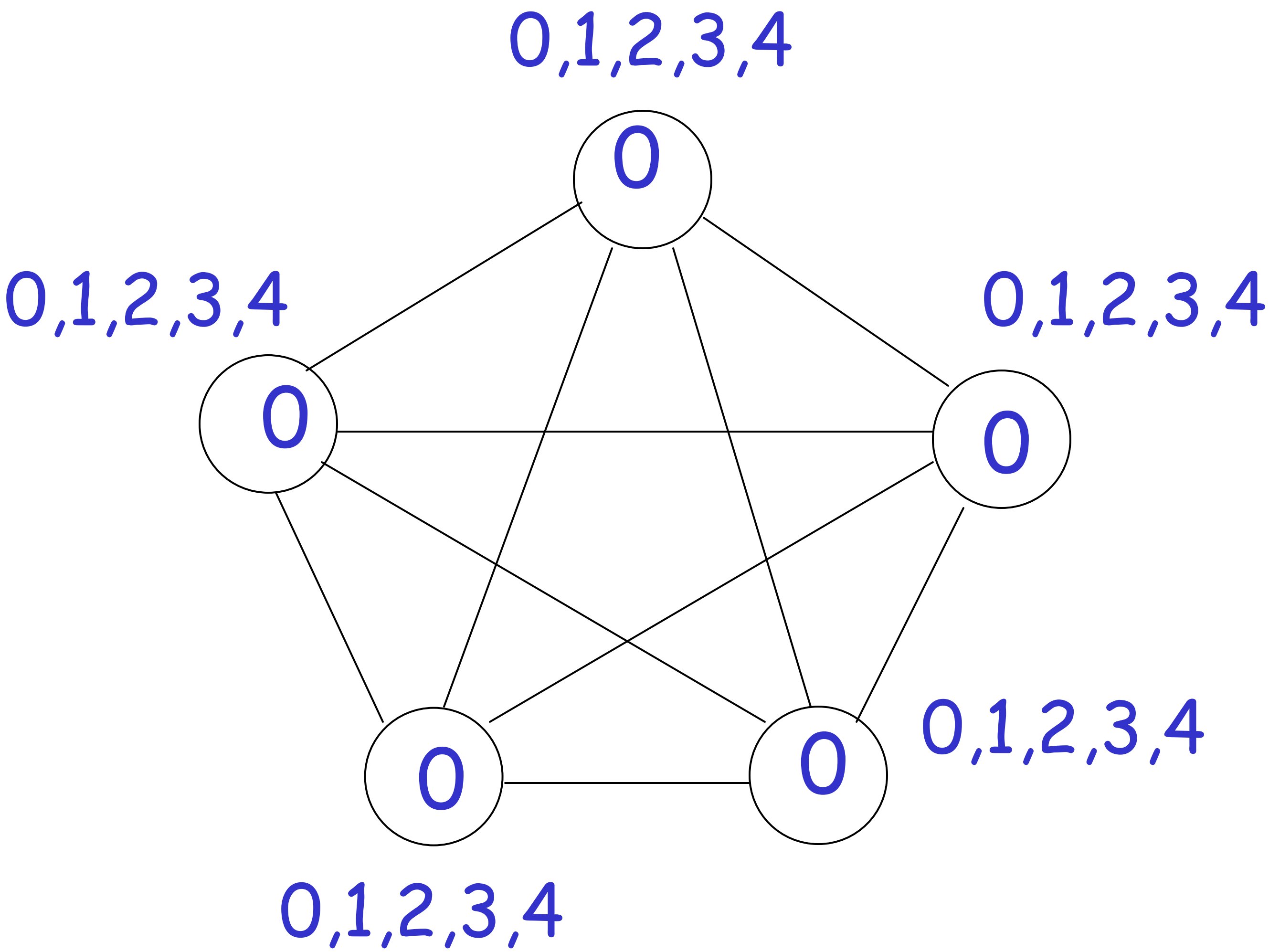
Start



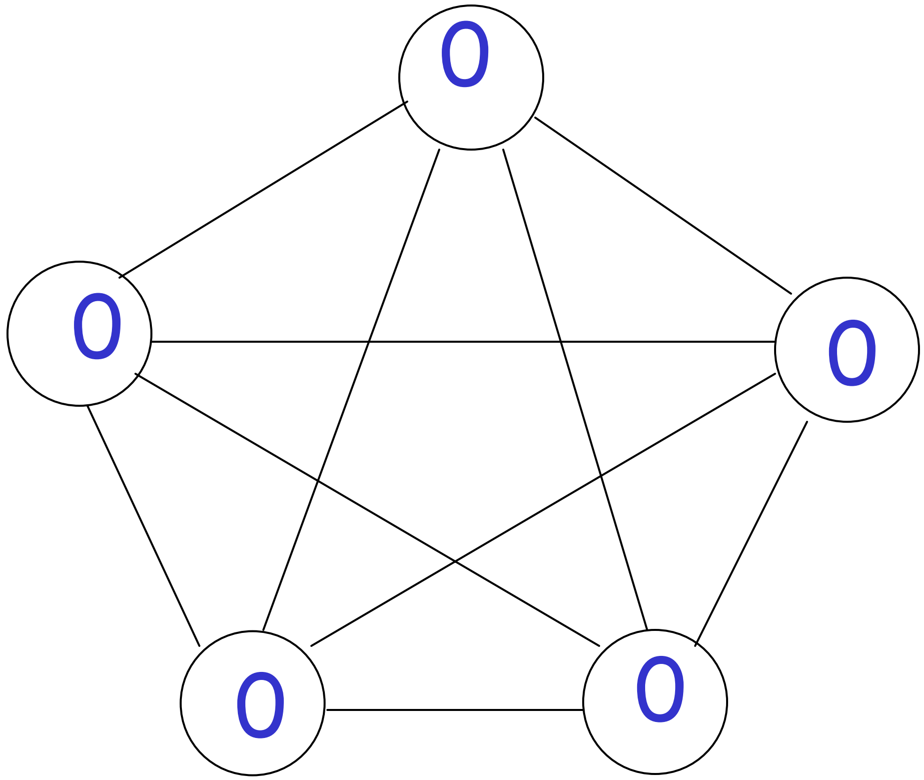
Broadcast values



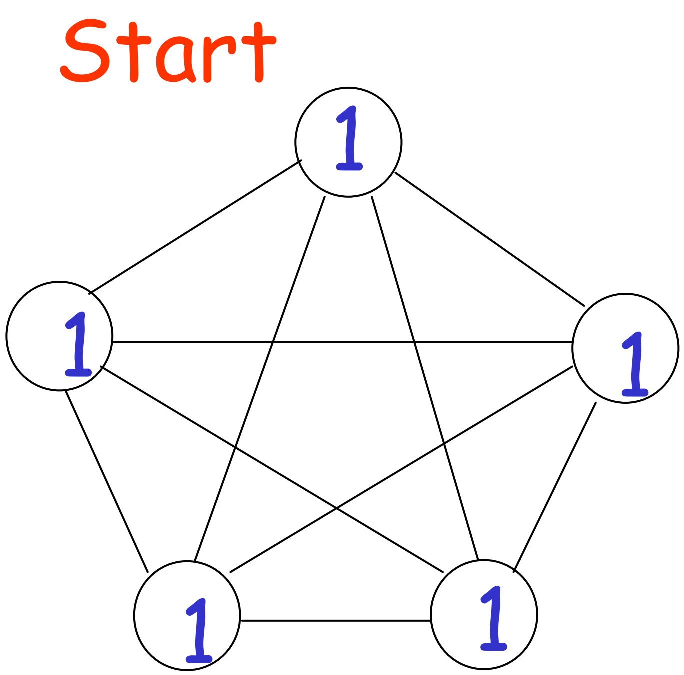
Decide on minimum

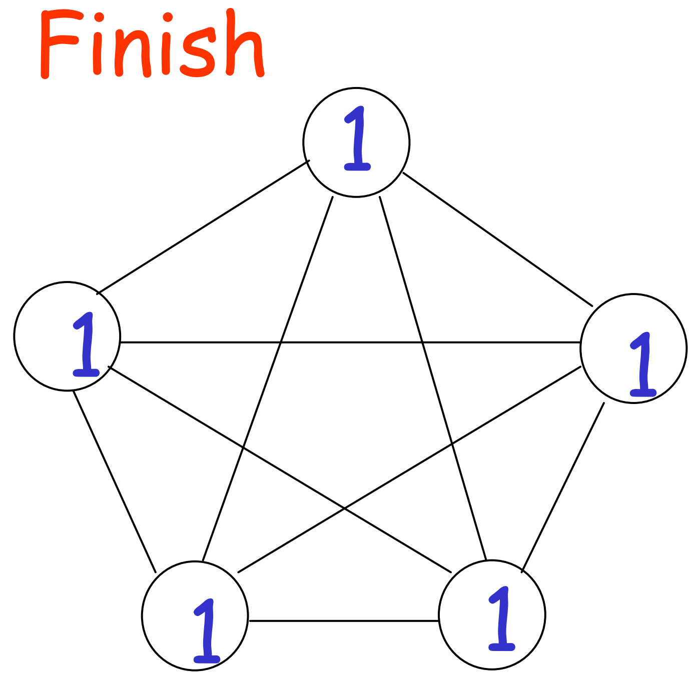


Finish



This algorithm satisfies the validity condition





If everybody starts with the same initial value, everybody sticks to that value (minimum)

Consensus with Crash Failures

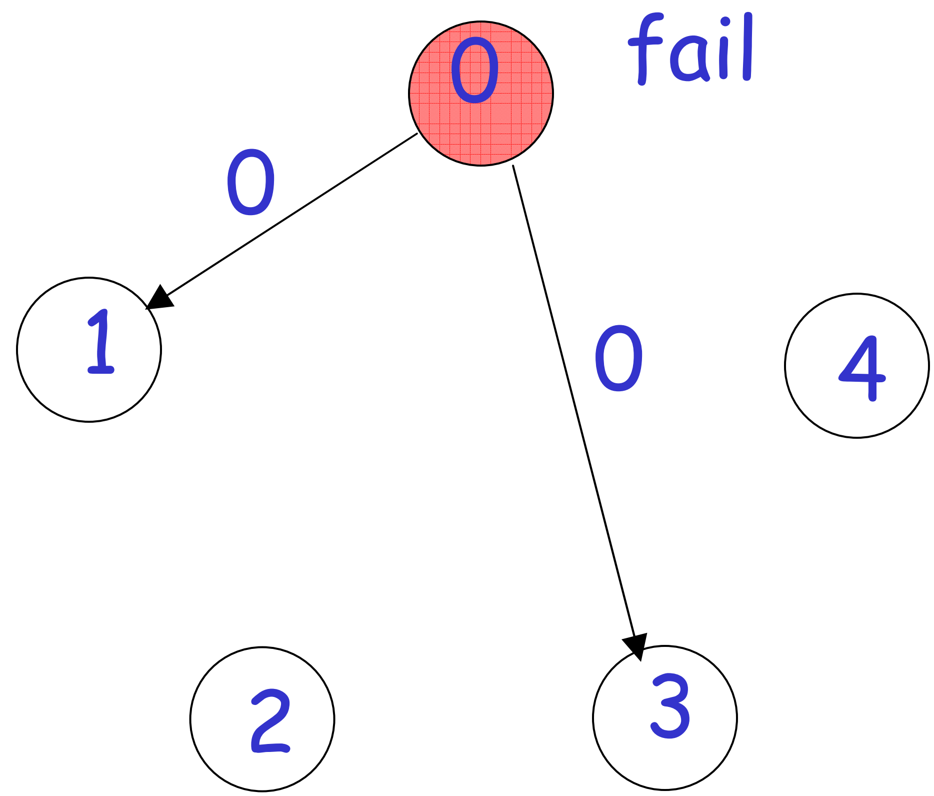
The simple algorithm doesn't work

Each processor:

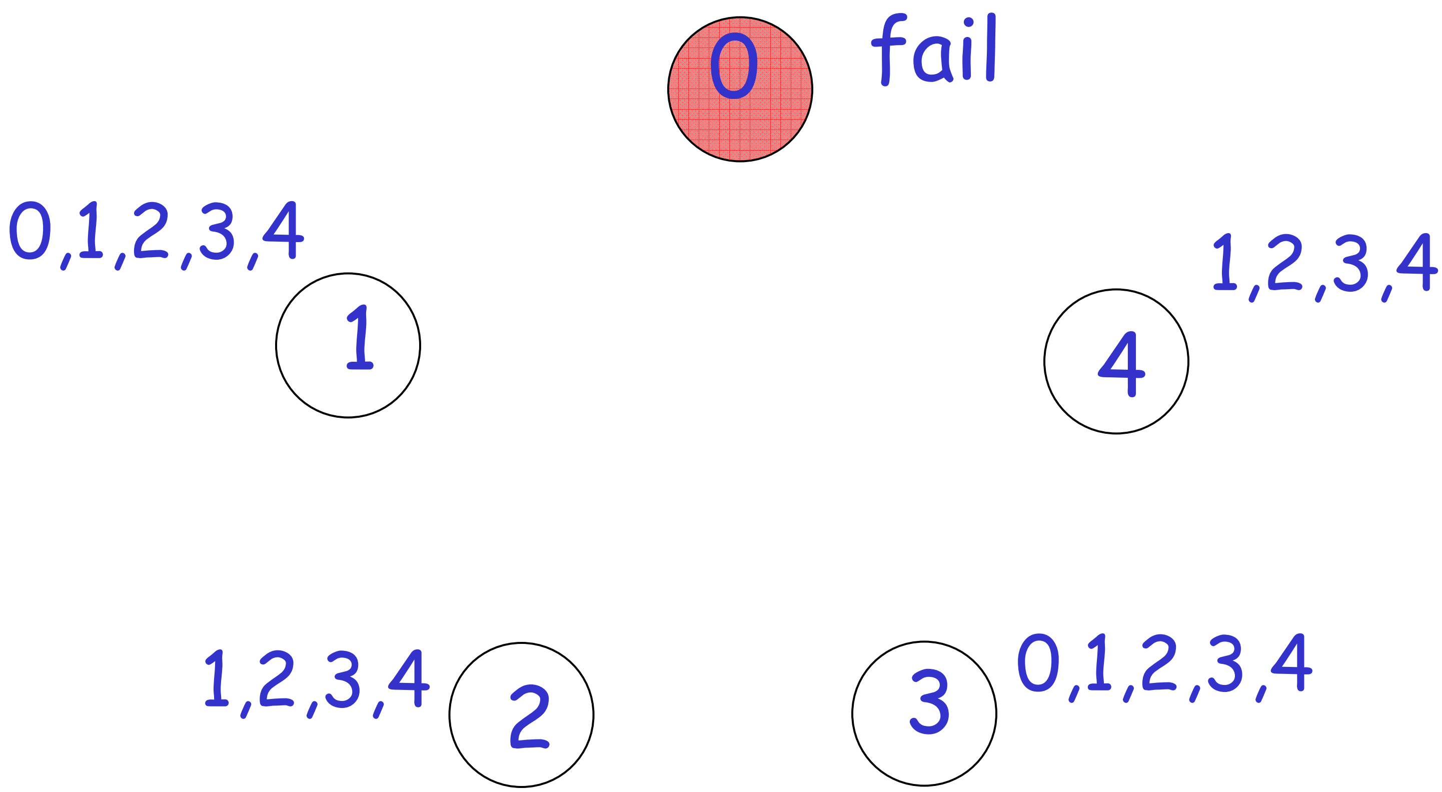
1. Broadcasts value to all processors

2. Decides on the minimum

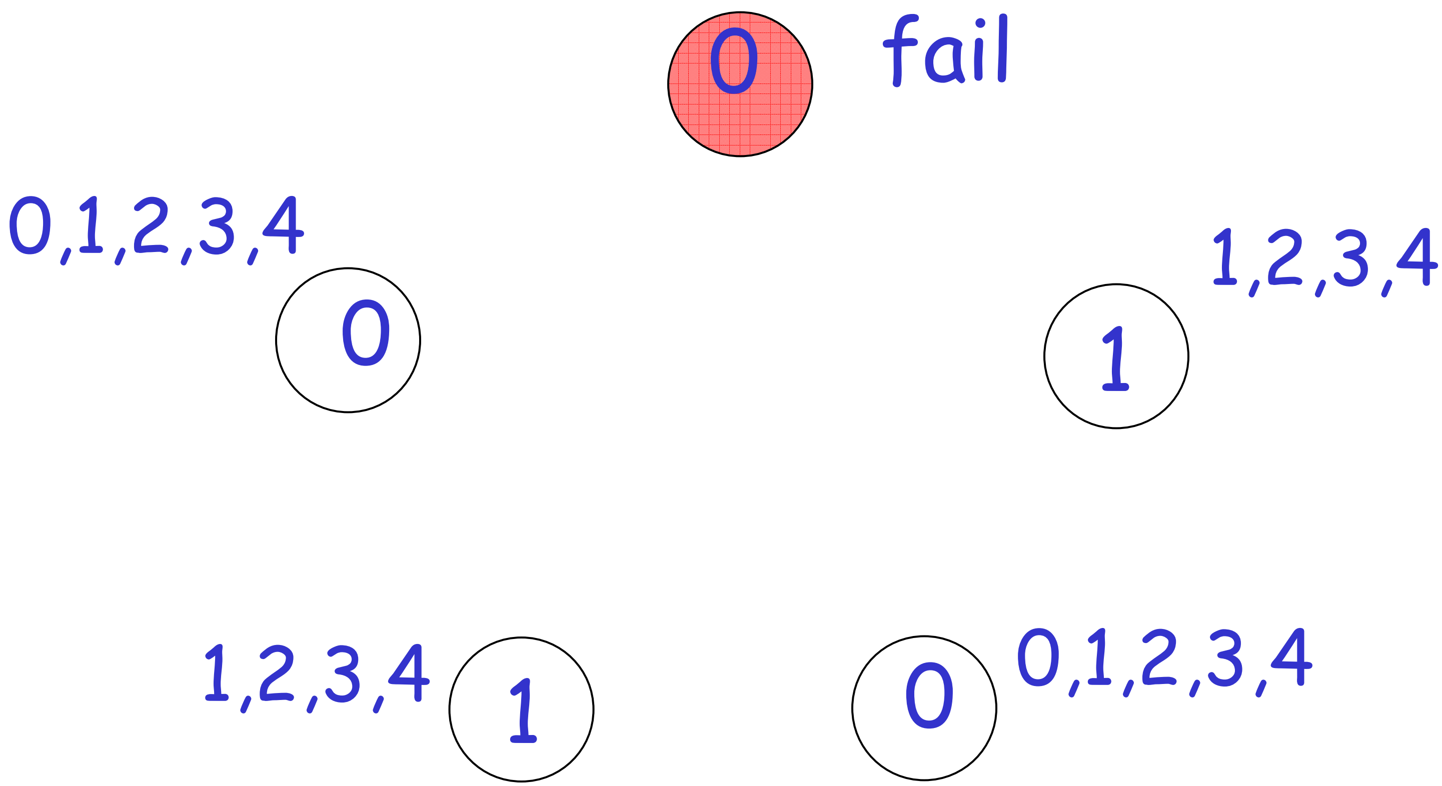
Start The failed processor doesn't broadcast its value to all processors



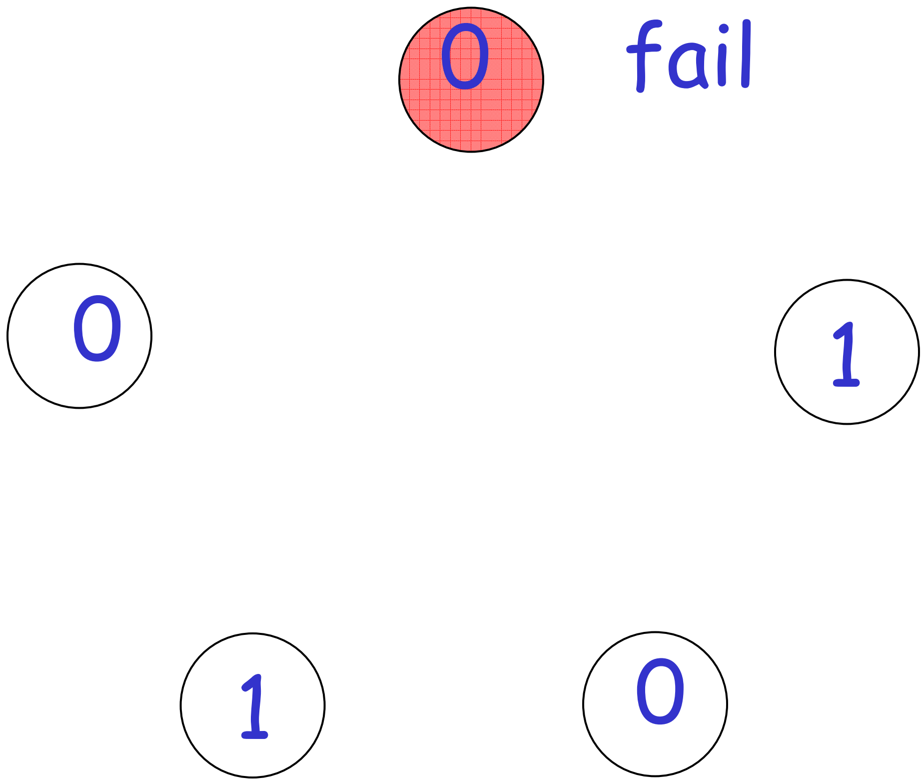
Broadcasted values



Decide on minimum



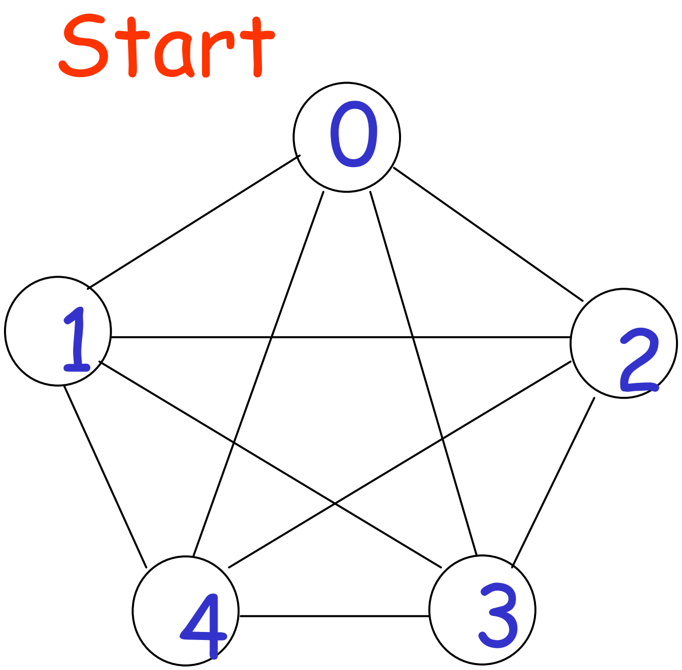
Finish - No Consensus!

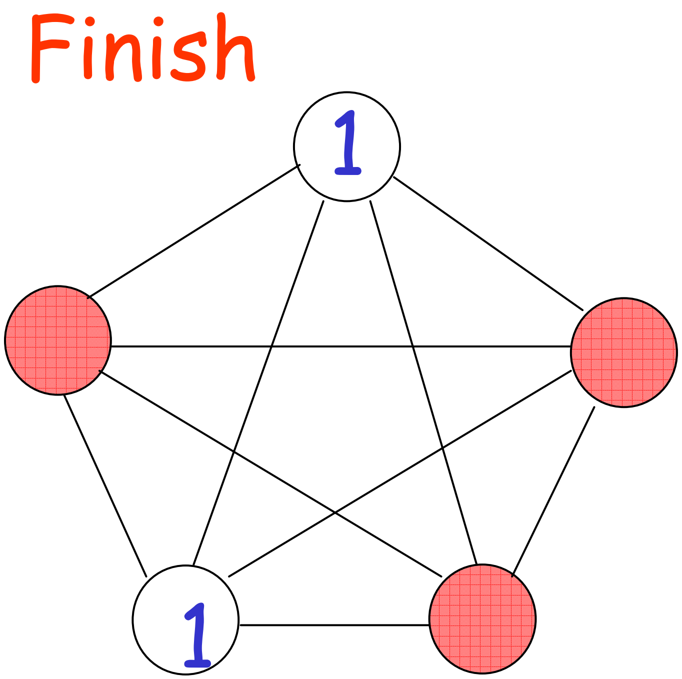


If an algorithm solves consensus for f failed processes we say it is

an f-resilient consensus algorithm

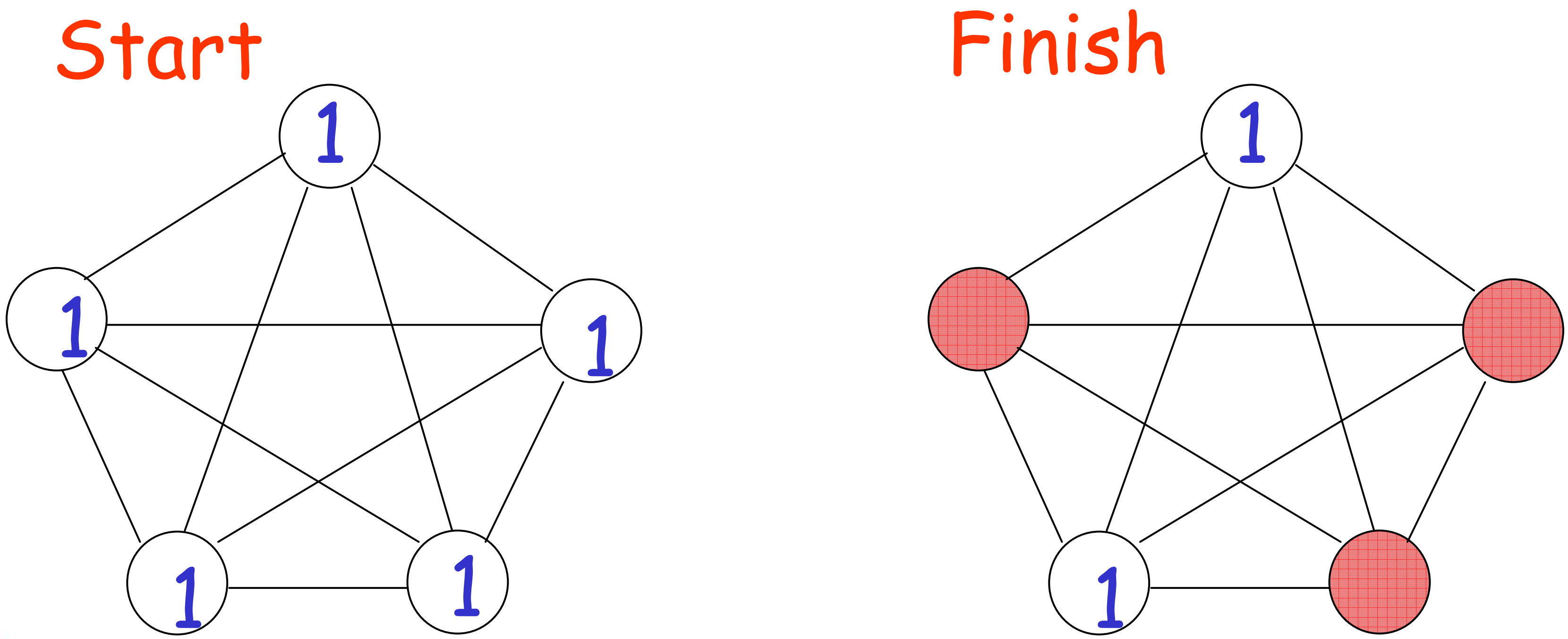
Example: The input and output of a 3-resilient consensus algorithm





New validity condition:

all non-faulty processes decide on a value that is available initially.



An f-resilient algorithm

Round 1:

Broadcast my value

Round 2 to round f+1:

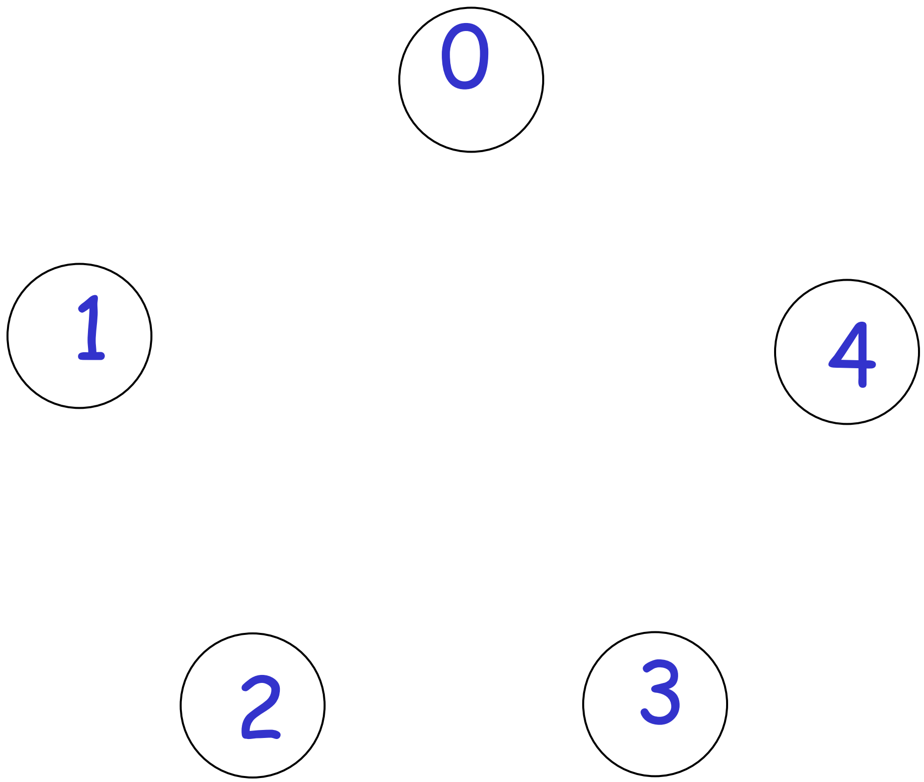
Broadcast any new received values

End of round f+1:

Decide on the minimum value received

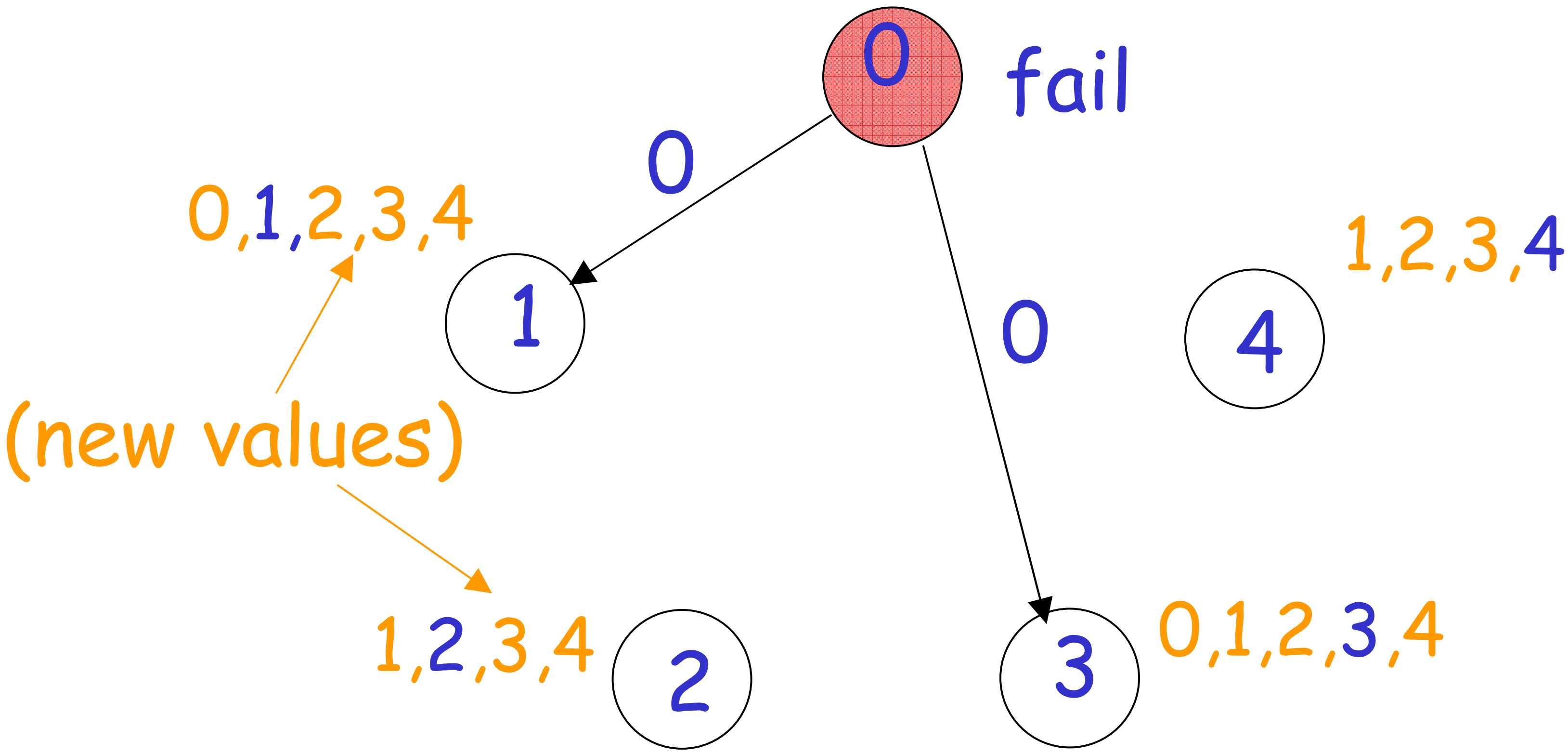
Example: f=1 failures, f+1=2 rounds needed

Start



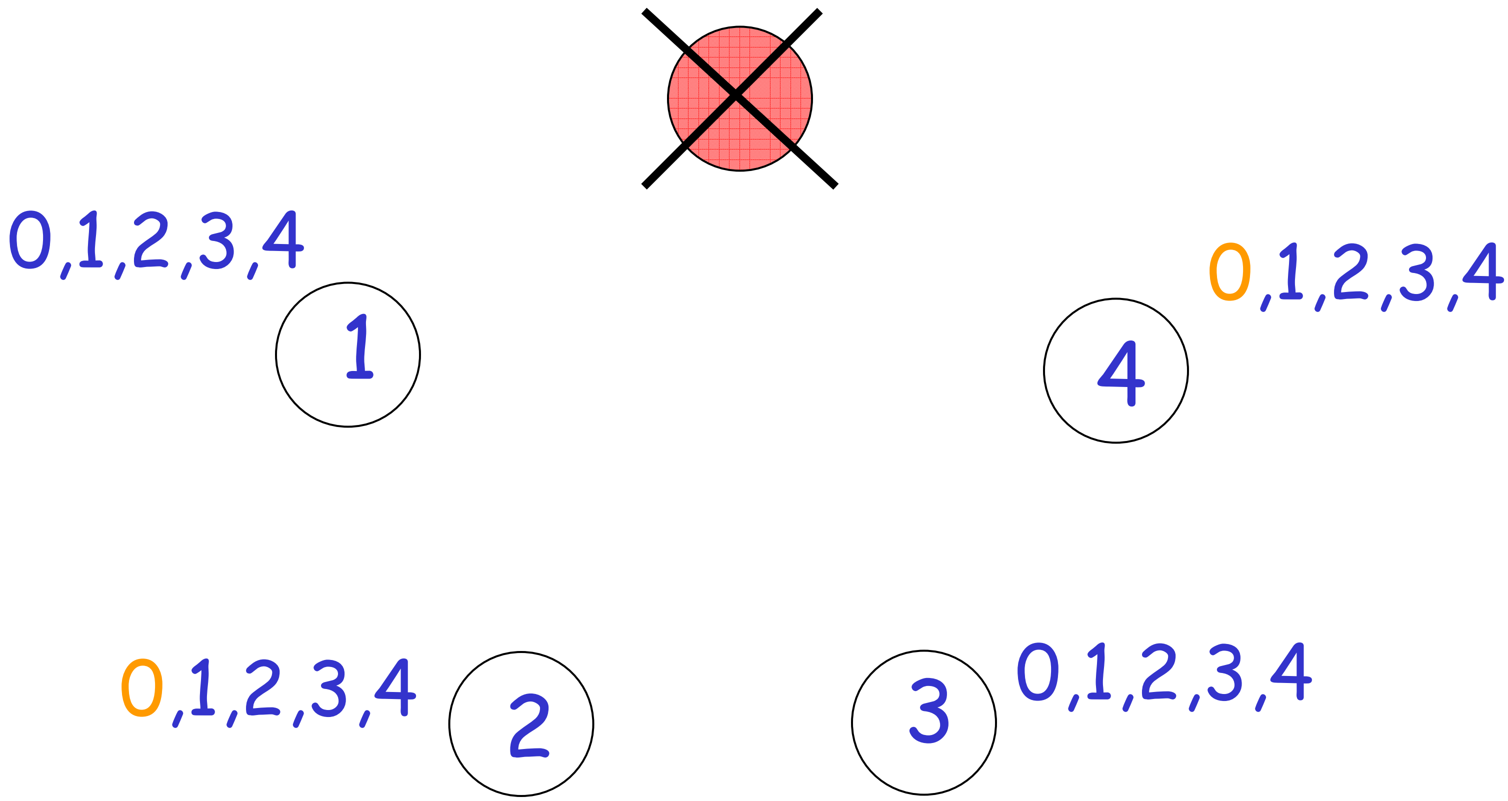
Example: f=1 failures, f+1 = 2 rounds needed

Round 1 Broadcast all values to everybody



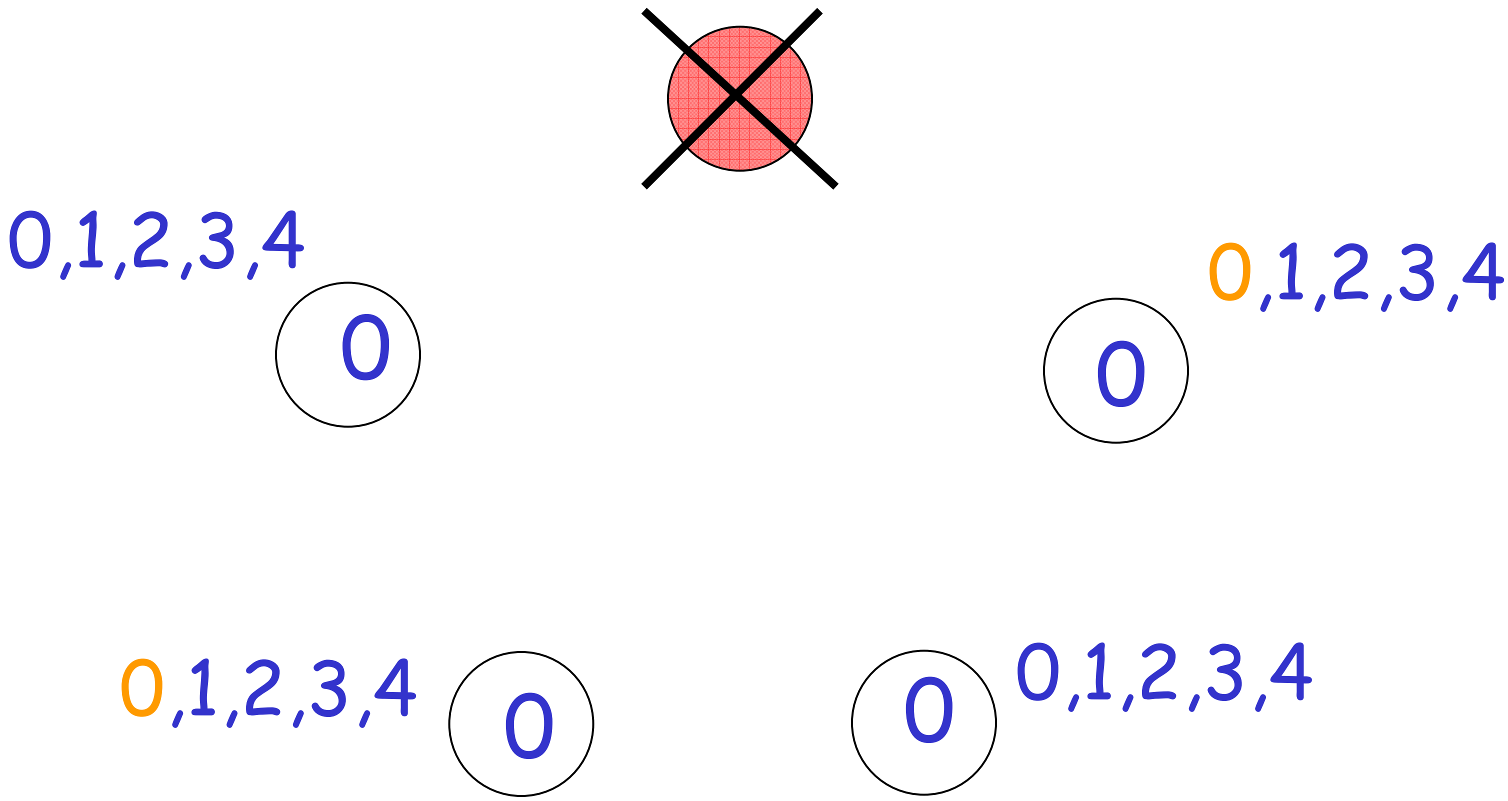
Example: f=1 failures, f+1 = 2 rounds needed

Round 2 Broadcast all new values to everybody



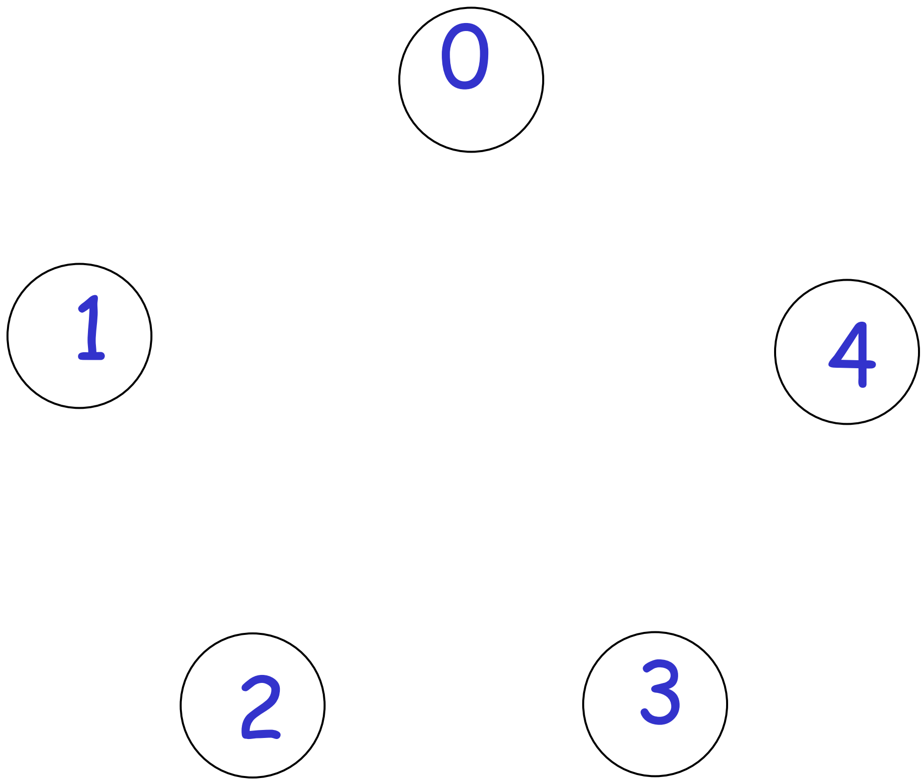
Example: f=1 failures, f+1 = 2 rounds needed

Finish Decide on minimum value



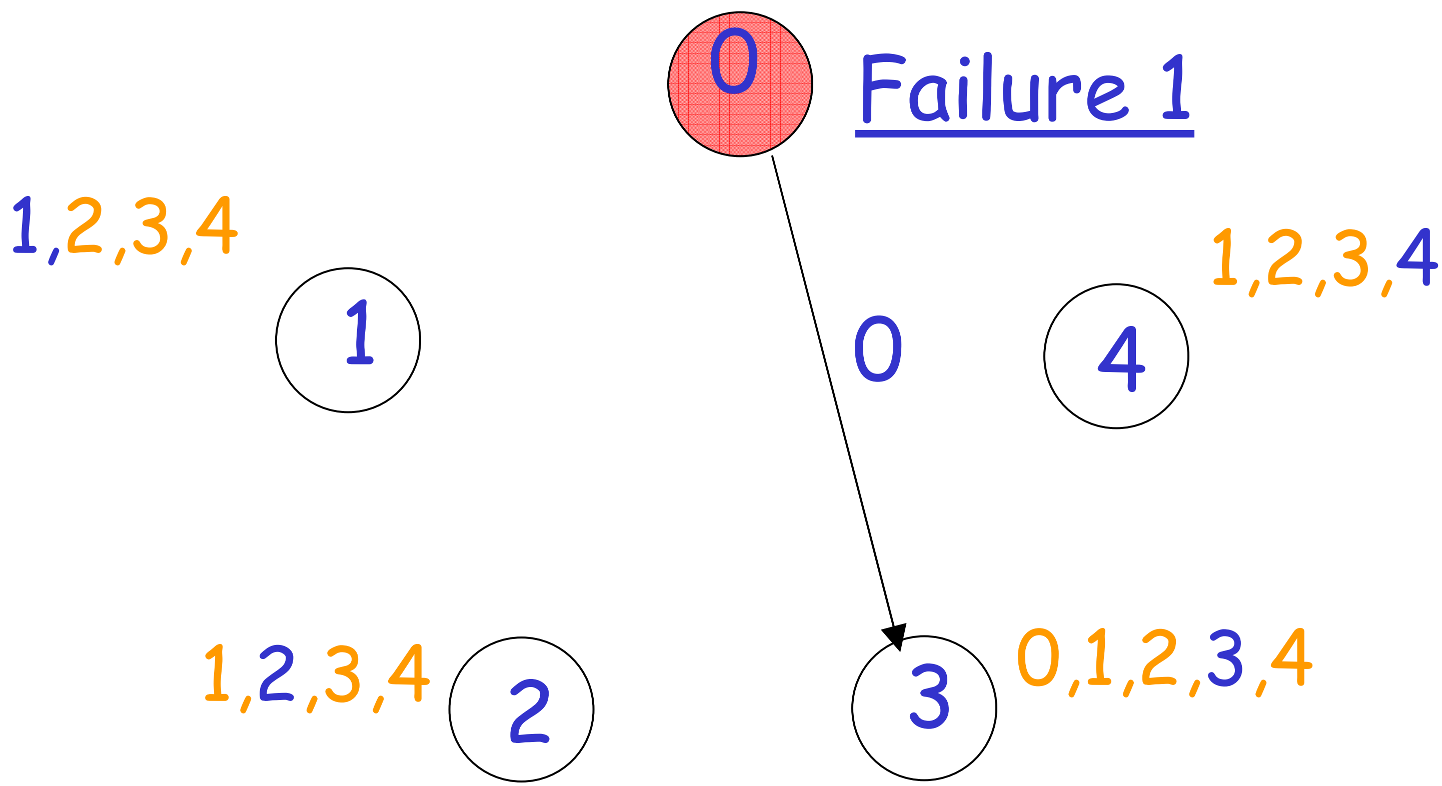
Example: f=2 failures, f+1 = 3 rounds needed

Start   Example of execution with 2 failures



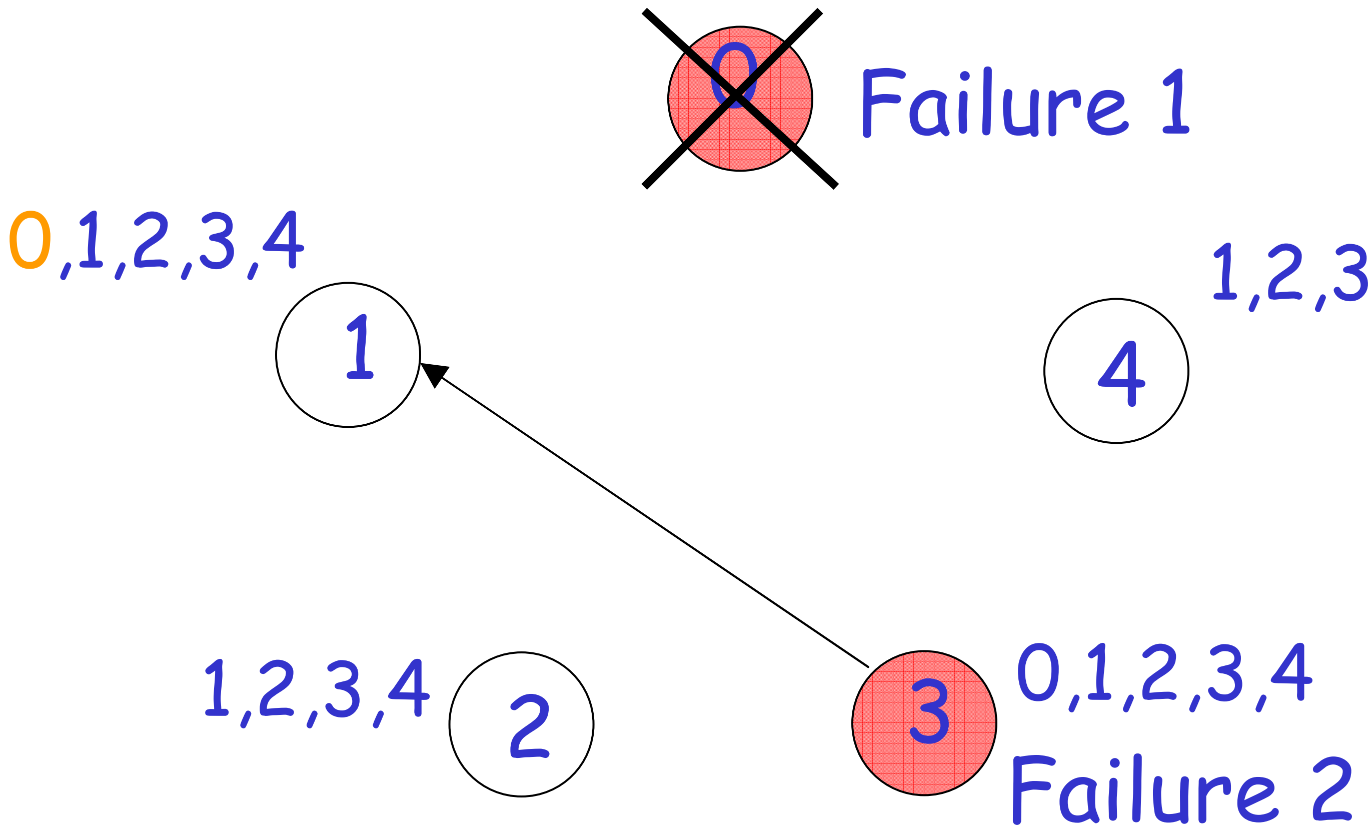
Example: f=2 failures, f+1 = 3 rounds needed

Round 1   Broadcast all values to everybody



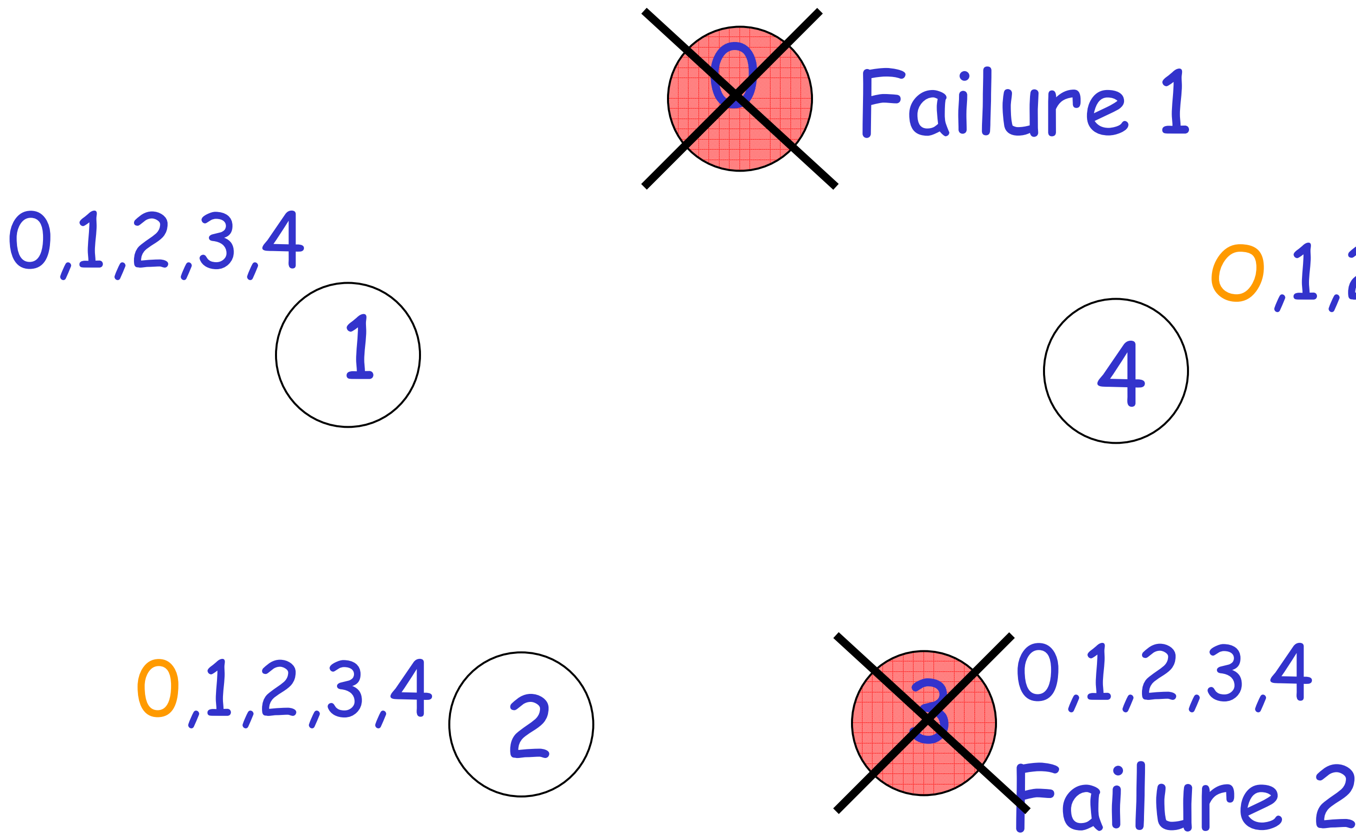
Example: f=2 failures, f+1 = 3 rounds needed

Round 2 Broadcast new values to everybody



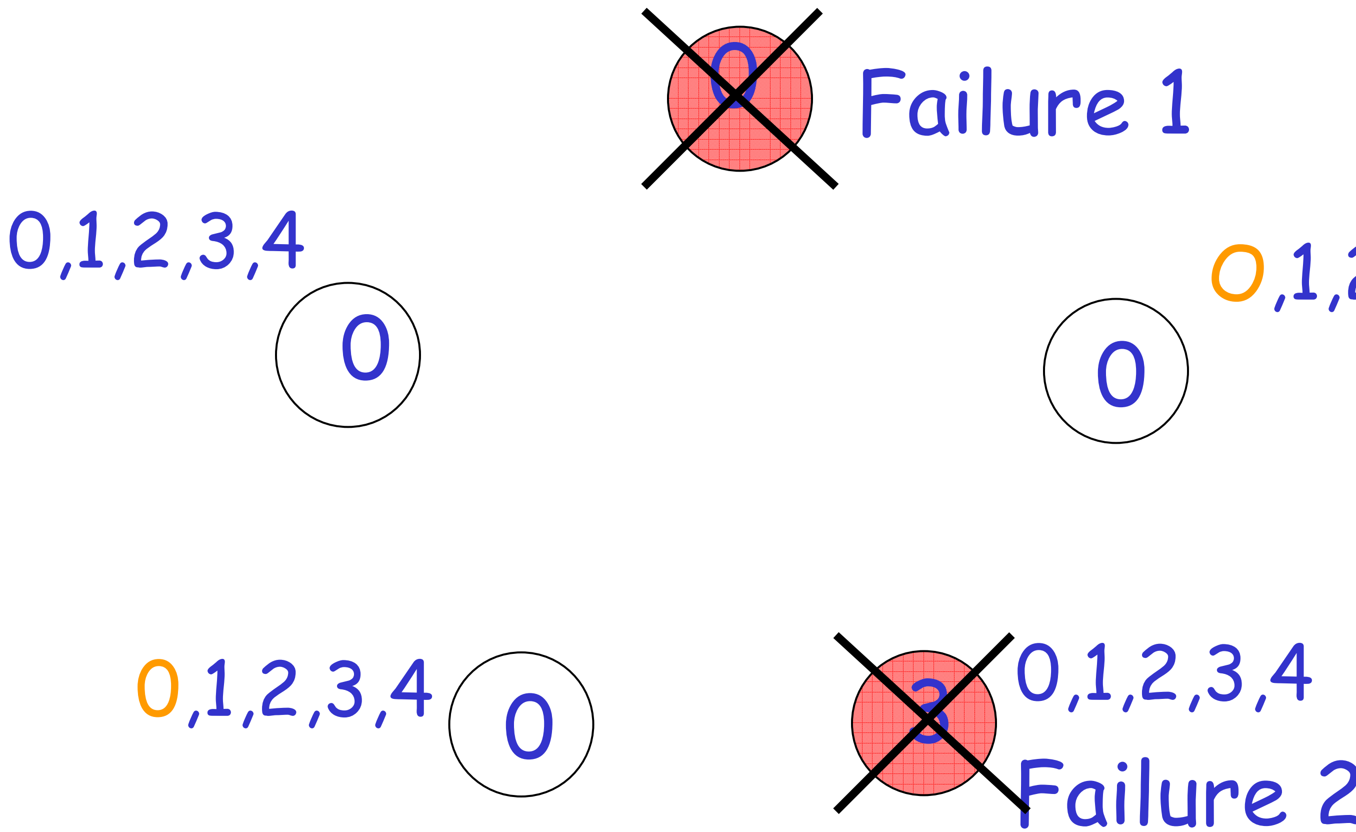
Example: f=2 failures, f+1 = 3 rounds needed

Round 3   Broadcast new values to everybody



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

Finish Decide on the minimum value

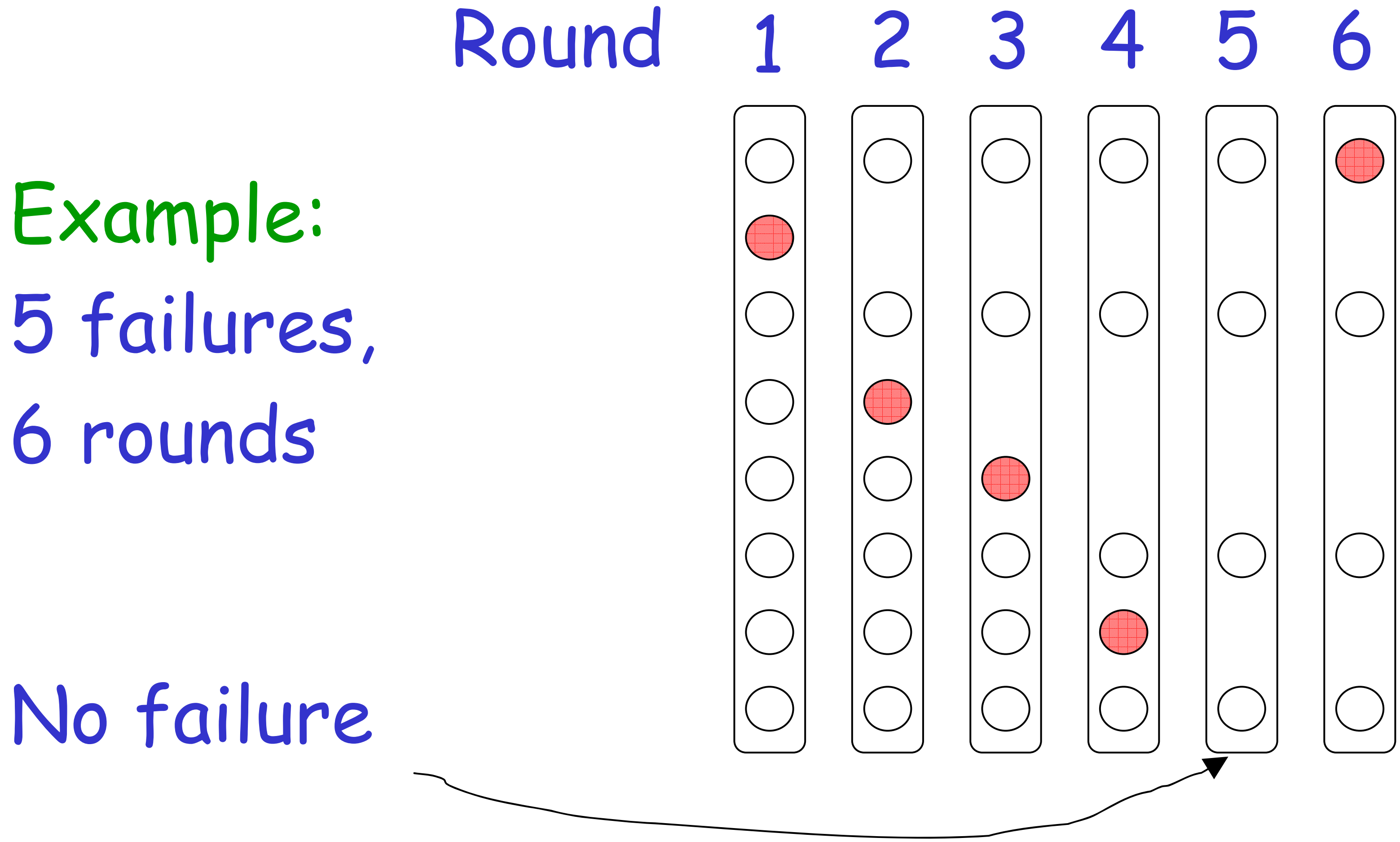


If there are f failures and f+1 rounds then there is a round with no failed process

Example:

5 failures,

6 rounds



At the end of the round with no failure:

- Every (non faulty) process knows about all the values of all the other participating processes

- This knowledge doesn't change until the end of the algorithm

Therefore, at the end of the round with no failure:

Everybody would decide on the same value

However, as we don't know the exact position of this round,
we have to let the algorithm execute
for $f+1$ rounds

Validity of algorithm:

when all processes start with the same input value then the consensus is that value

This holds, since the value decided from each process is some input value

A Lower Bound

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

Proof sketch:

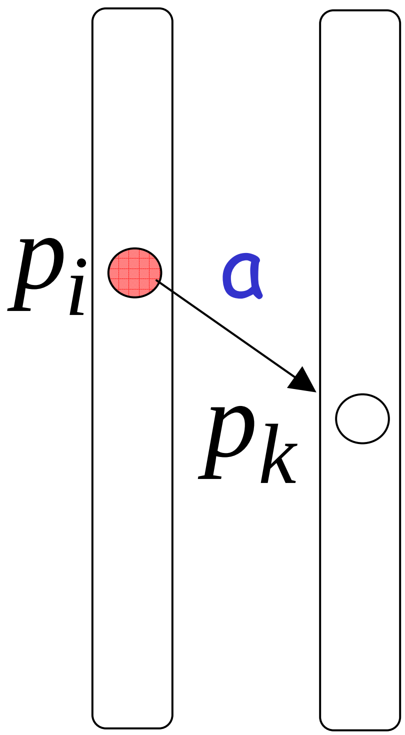
Assume for contradiction that f or less rounds are enough

Worst case scenario:

There is a process that fails in each round

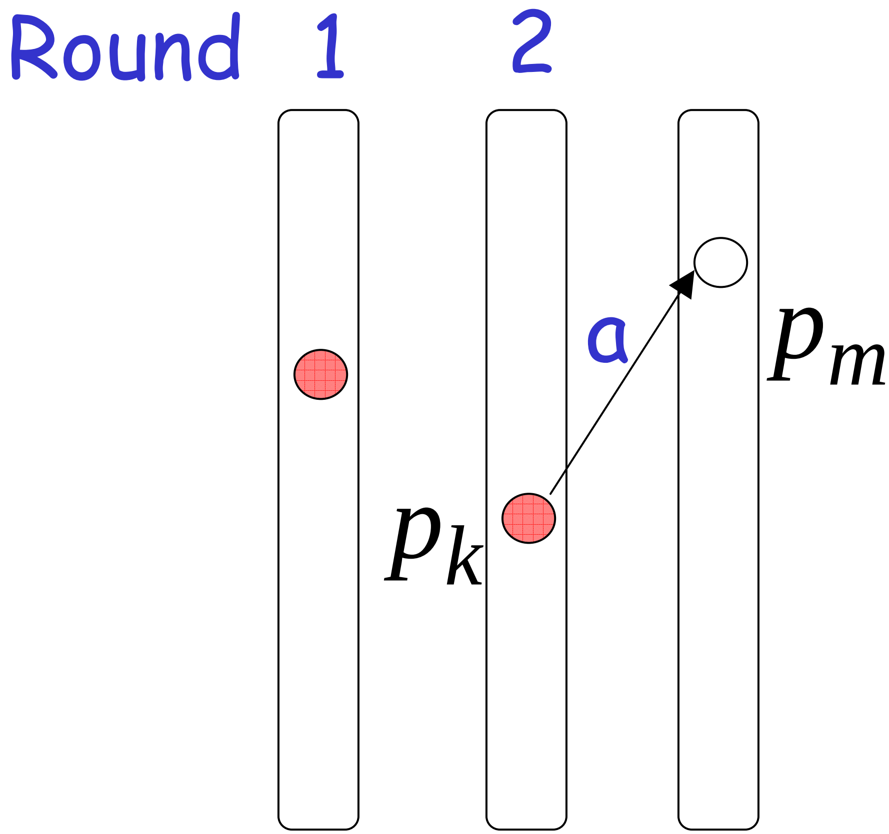
Worst case scenario

Round 1



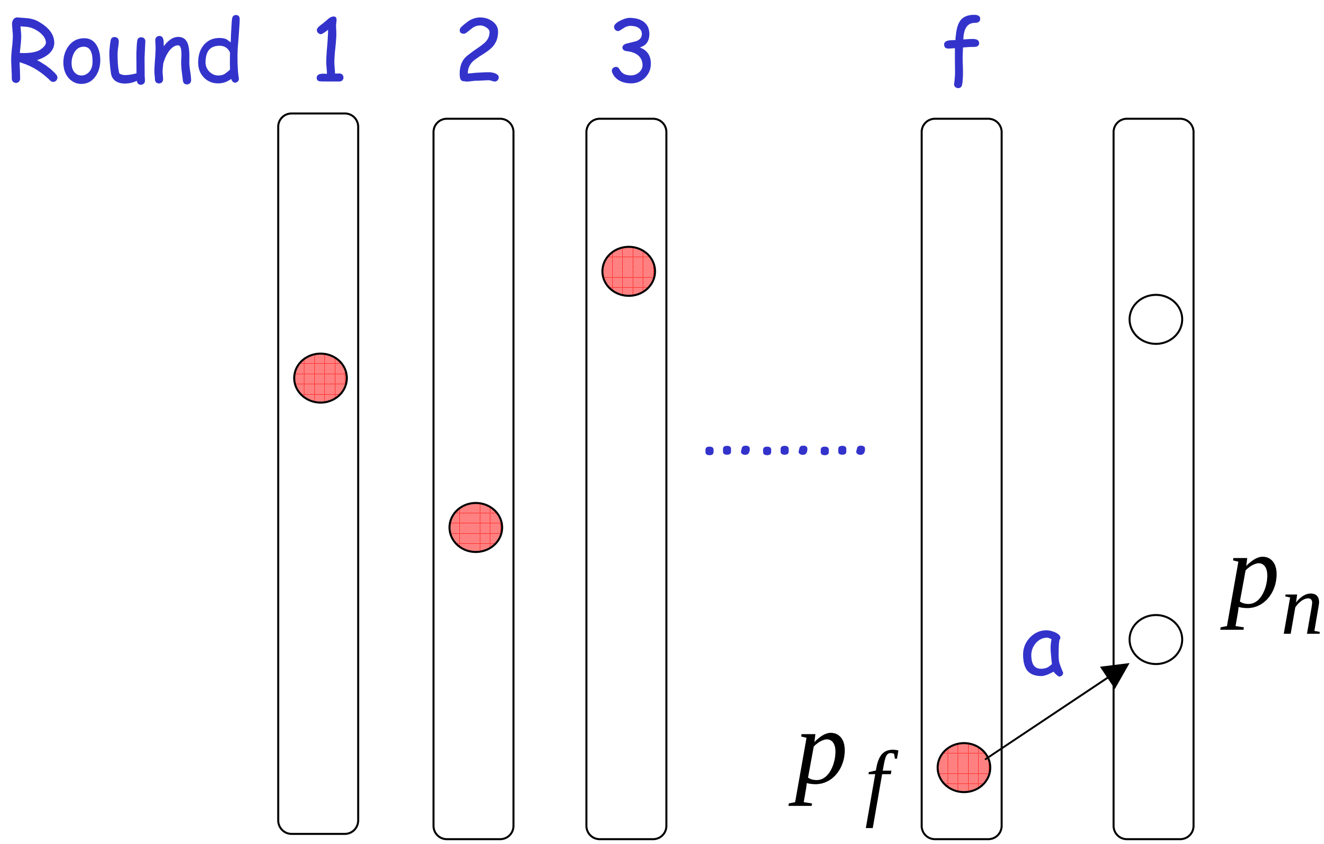
before process p_i fails, it sends its value **a** to only one process p_k

Worst case scenario



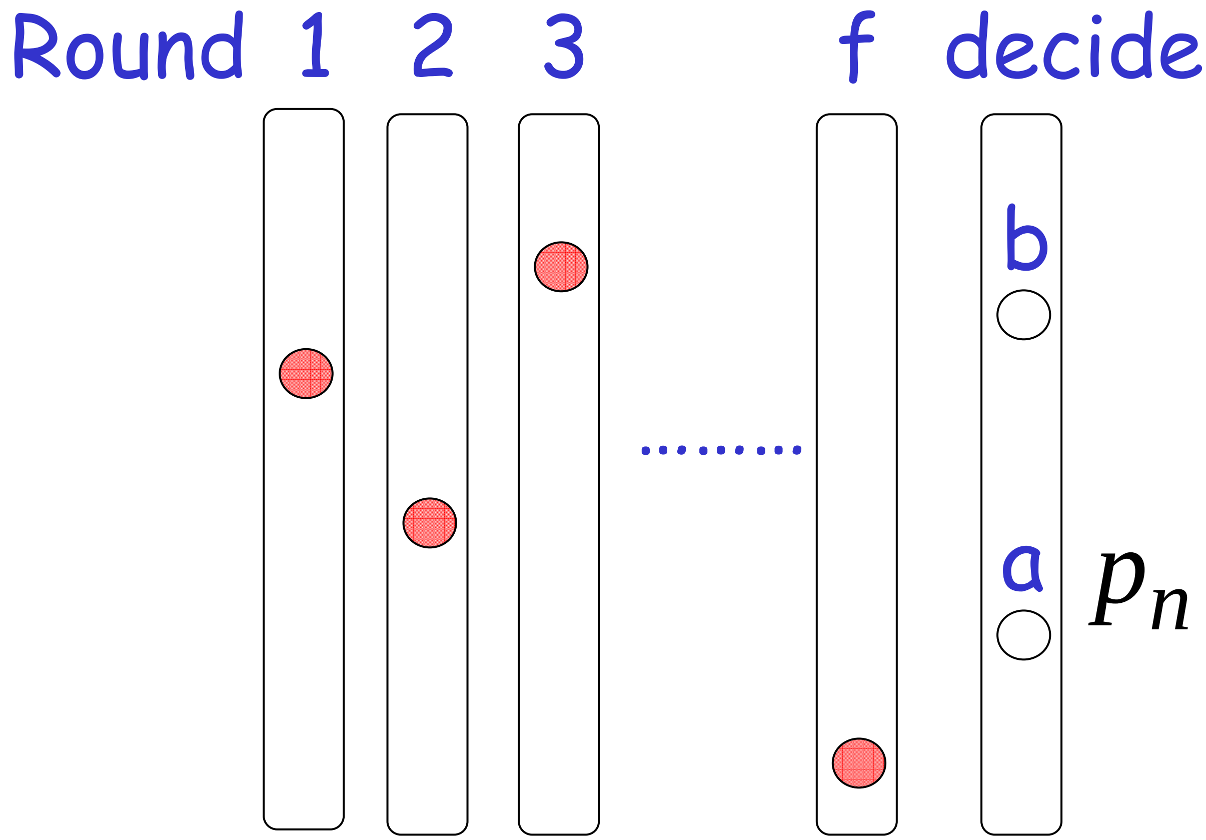
before process p_k fails, it sends value **a** to only one process p_m

Worst case scenario



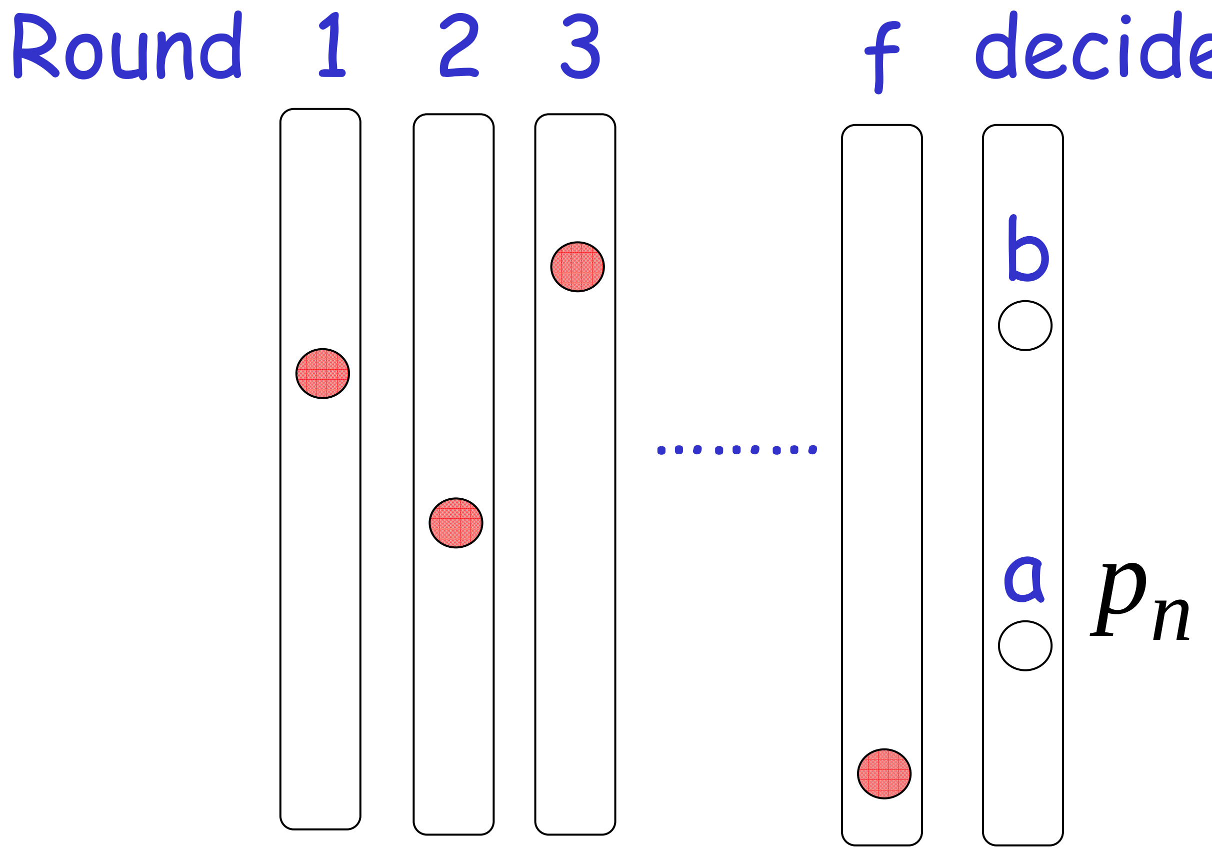
At the end of round **f** only one process p_n knows about value **a**

Worst case scenario



Process p_n may decide on **a**, and all other processes may decide on another value **(b)**

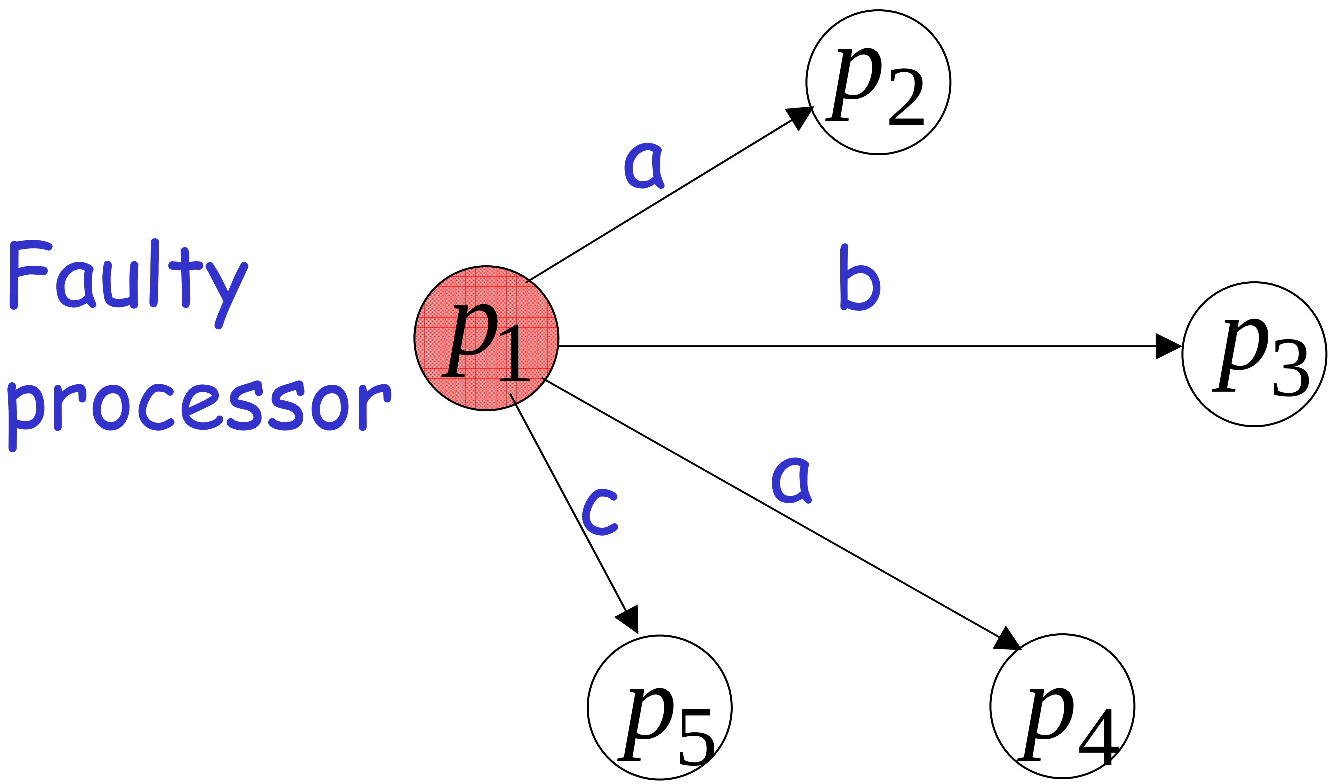
Worst case scenario



Therefore f rounds are not enough

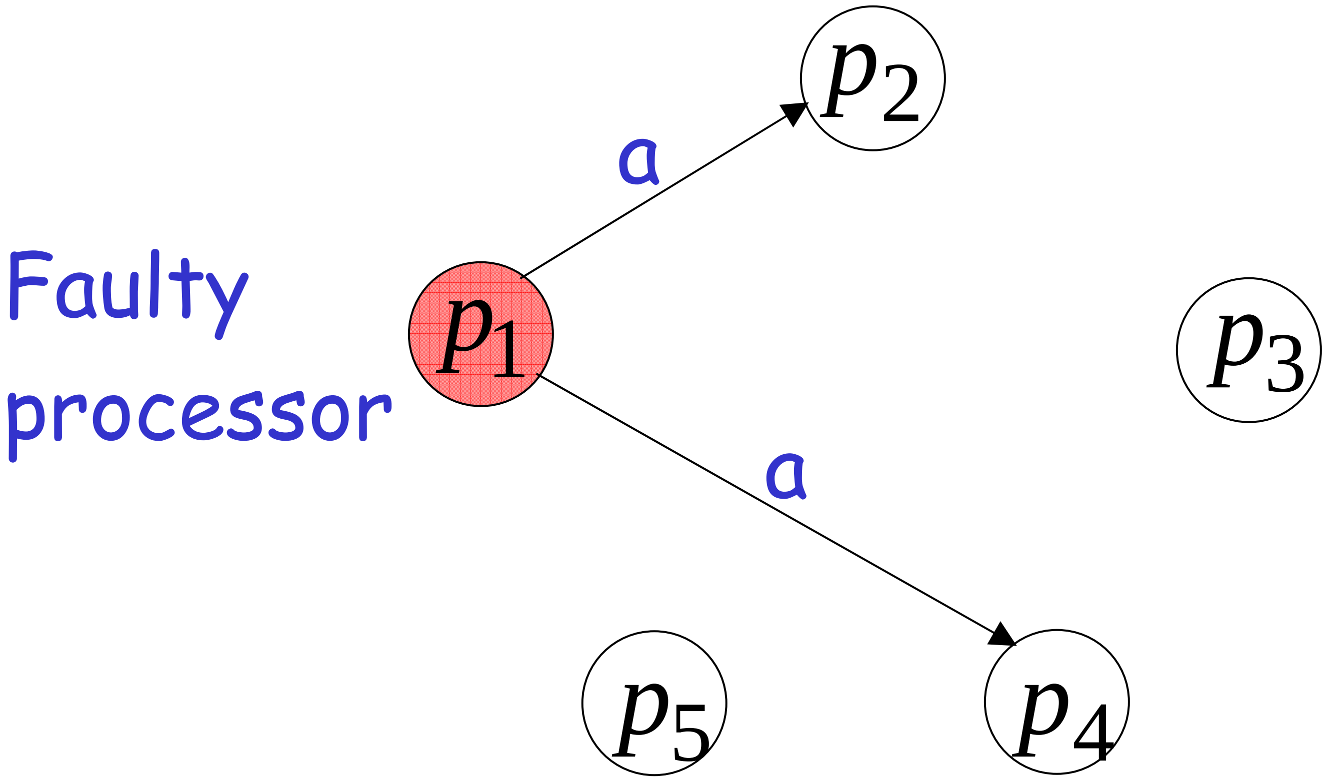
At least f+1 rounds are needed

Consensus #5
Byzantine Failures

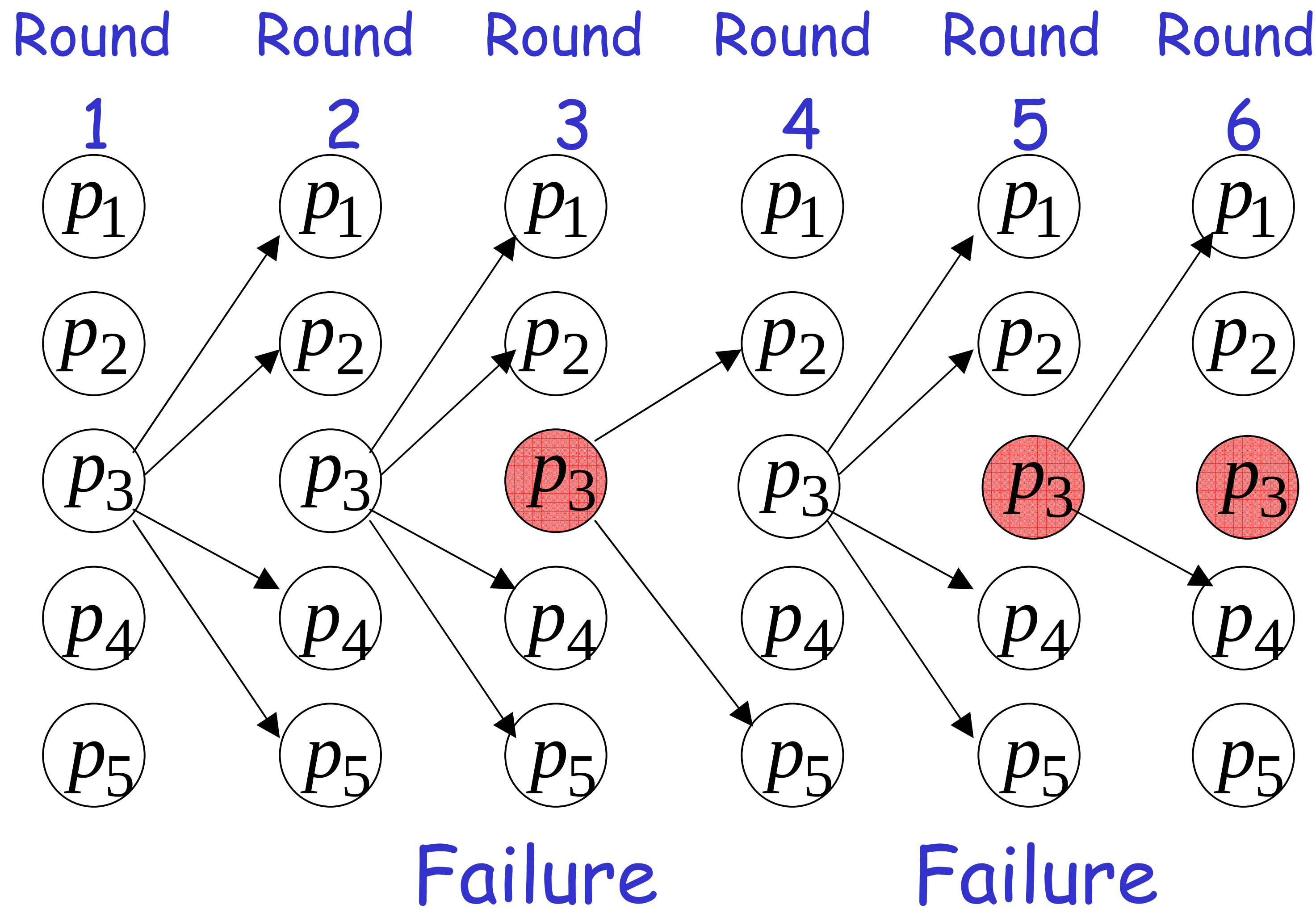


Different processes receive different values

Some messages may be lost



A Byzantine process can behave like a Crashed-failed process



After failure the process continues functioning in the network

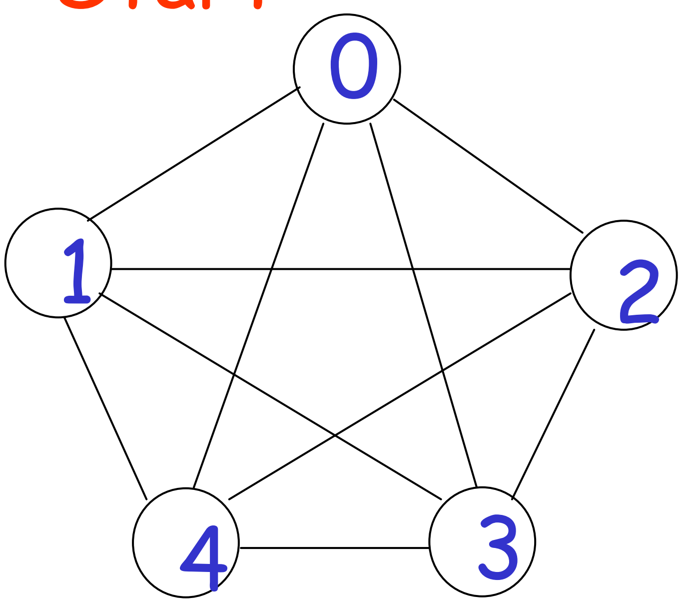
Consensus with Byzantine Failures

f-resilient consensus algorithm:

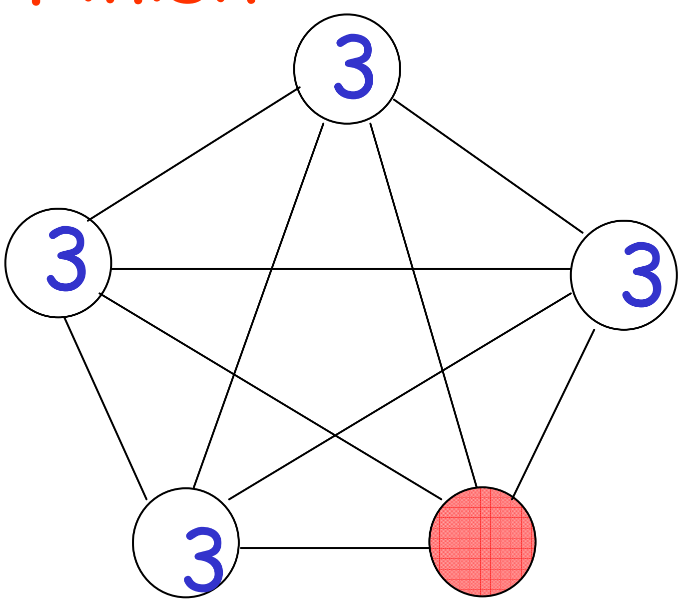
solves consensus for f failed processes

Example: The input and output of a 1-resilient consensus algorithm

Start



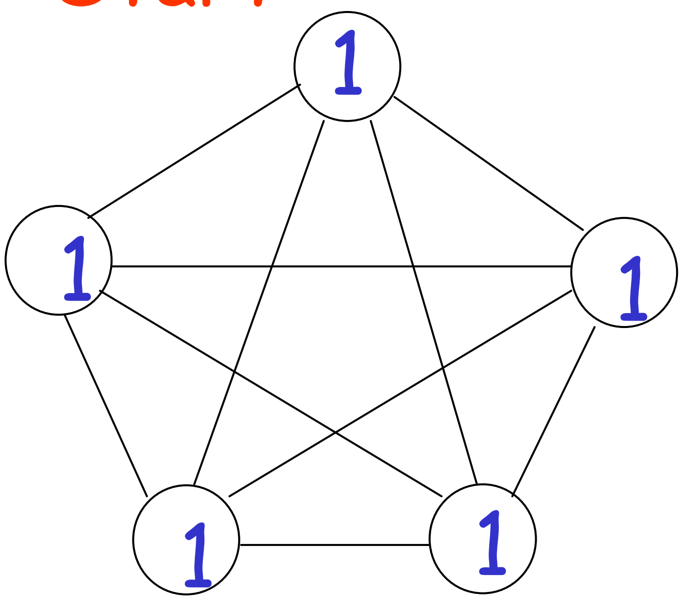
Finish



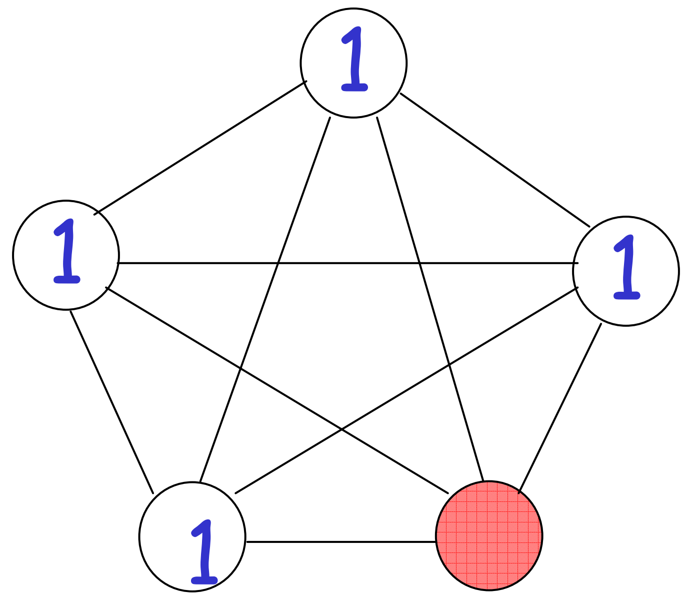
Validity condition:

if all non-faulty processes start with the same value then all non-faulty processes decide on that value

Start



Finish



Lower bound on number of rounds

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

Proof: follows from the crash failure lower bound

Upper bound on failed processes

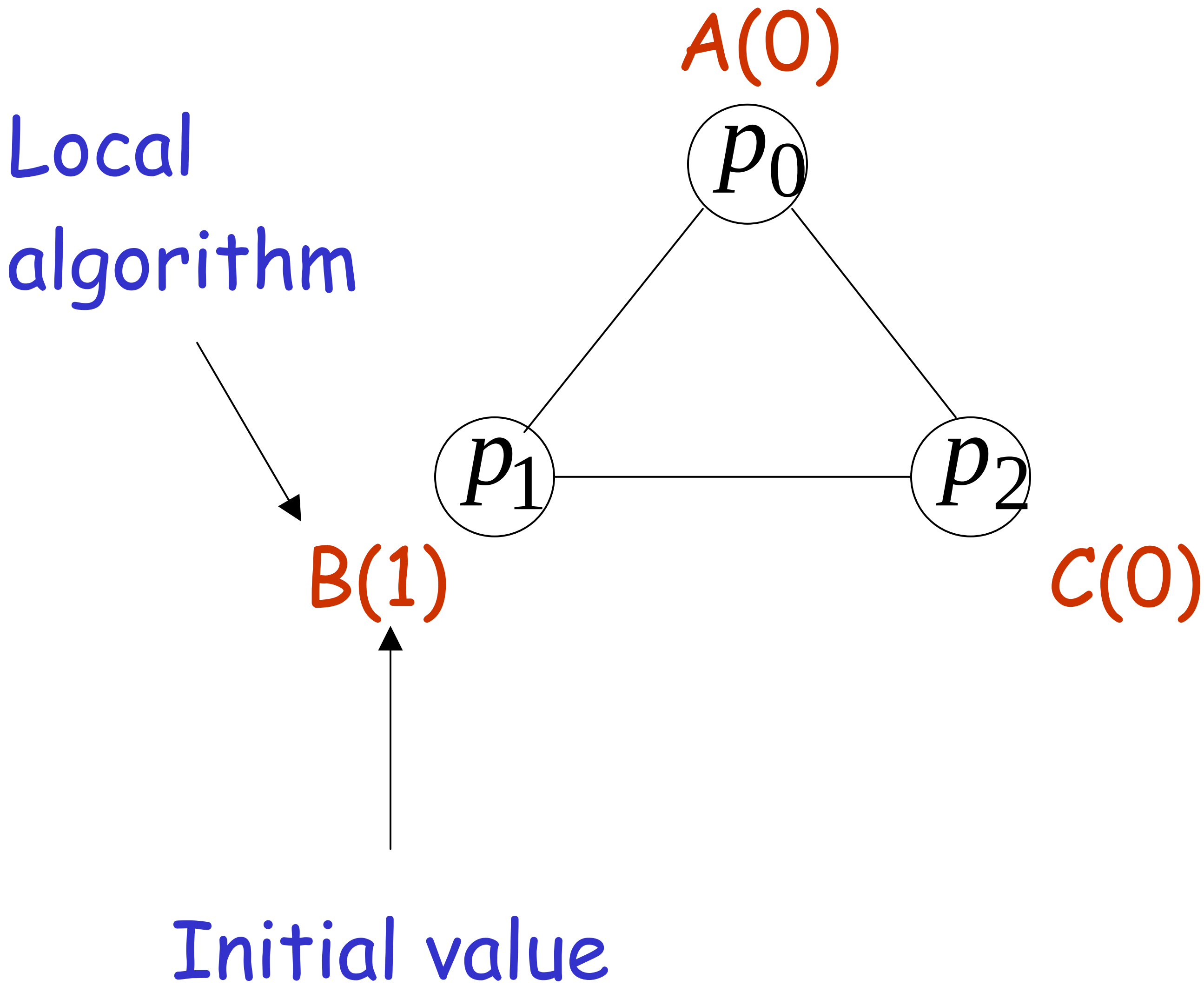
Theorem: There is no f-resilient algorithm for n processes, where $f \geq n/3$

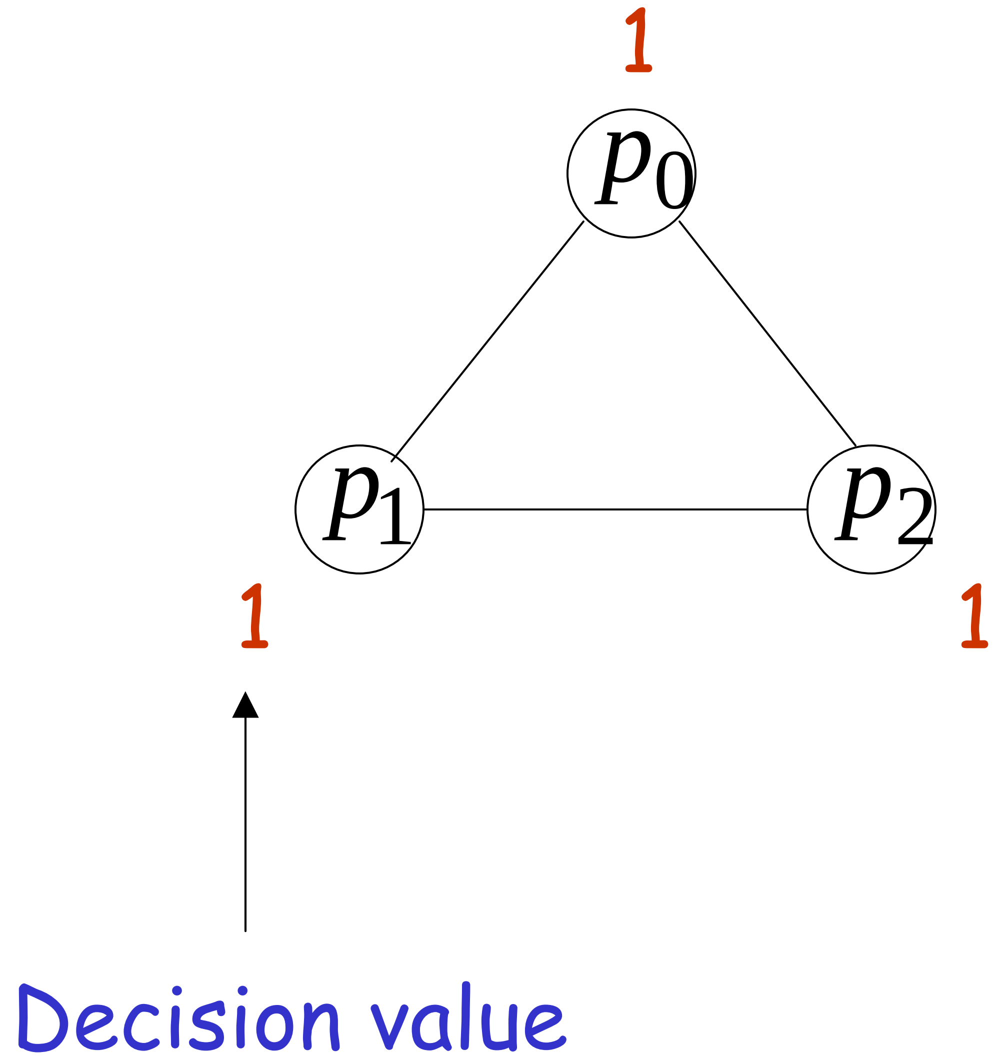
Plan: First we prove the 3 process case, and then the general case

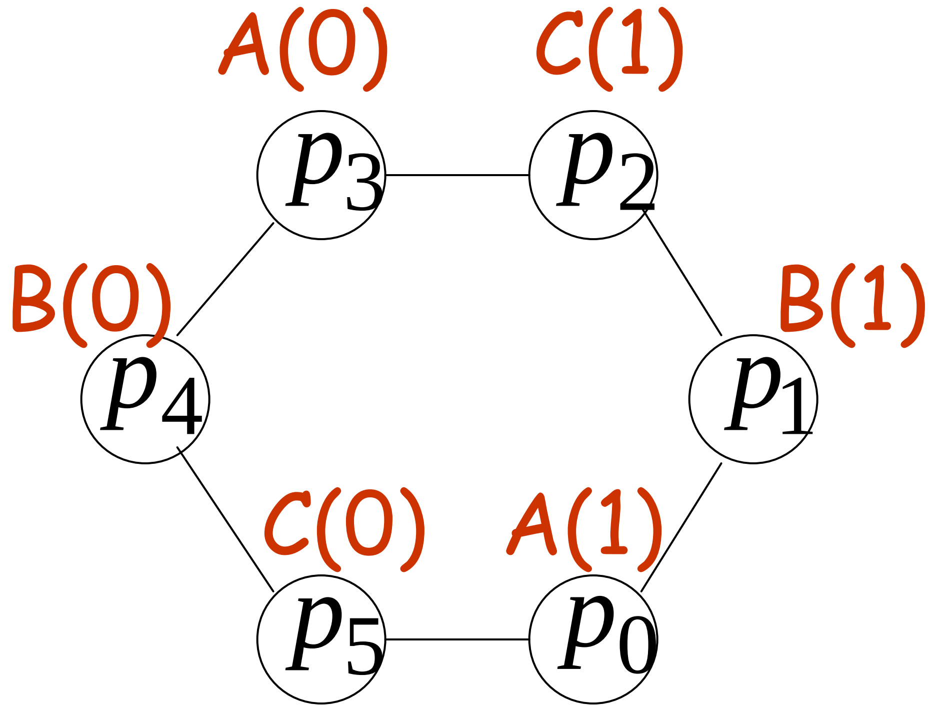
The 3 processes case

Lemma: There is no 1-resilient algorithm for 3 processes

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

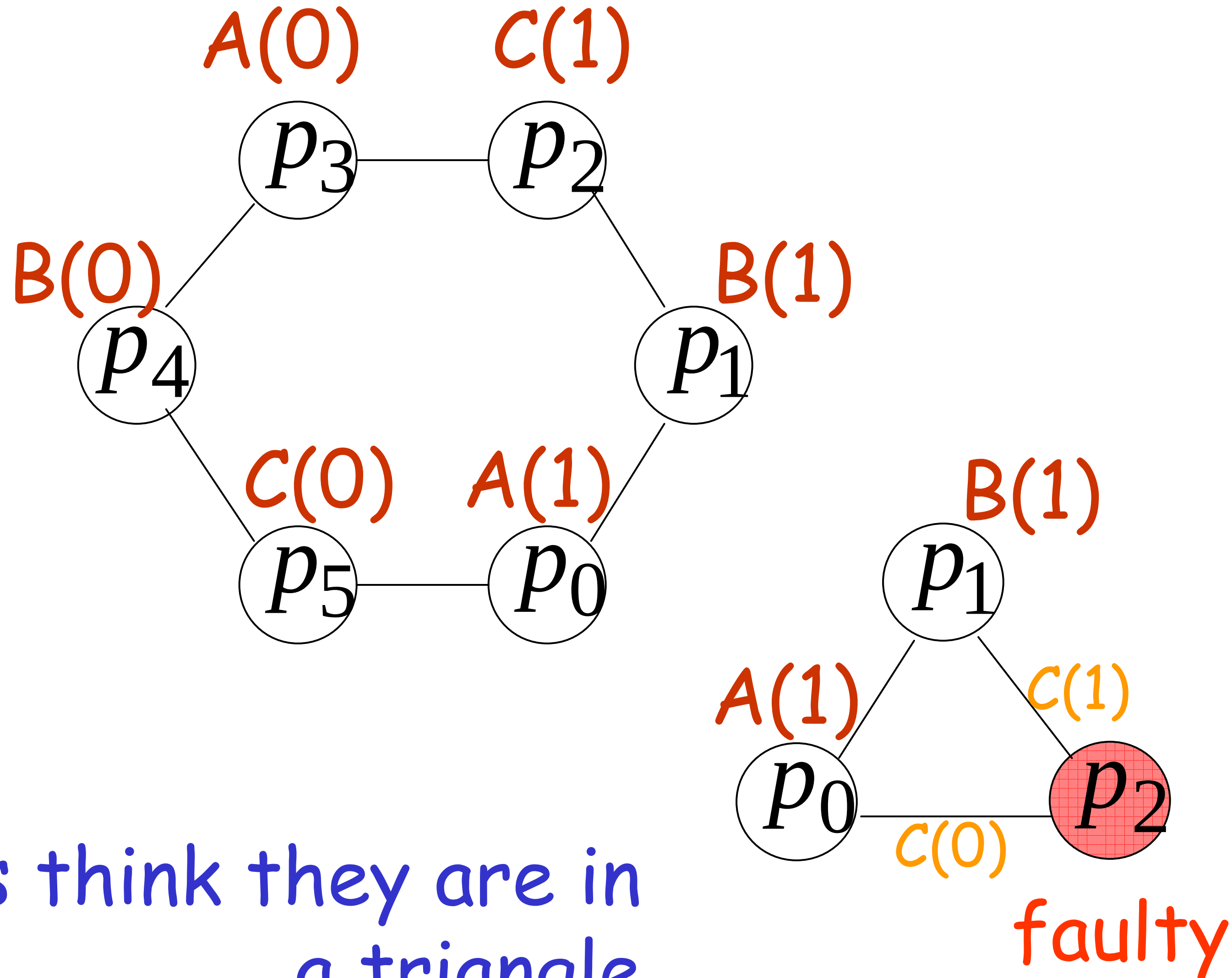

A(0)
p_0
Local
algorithm
p_1
p_2
B(1)
C(0)
Initial value


1
p_0
p_1
p_2
1
1
Decision value

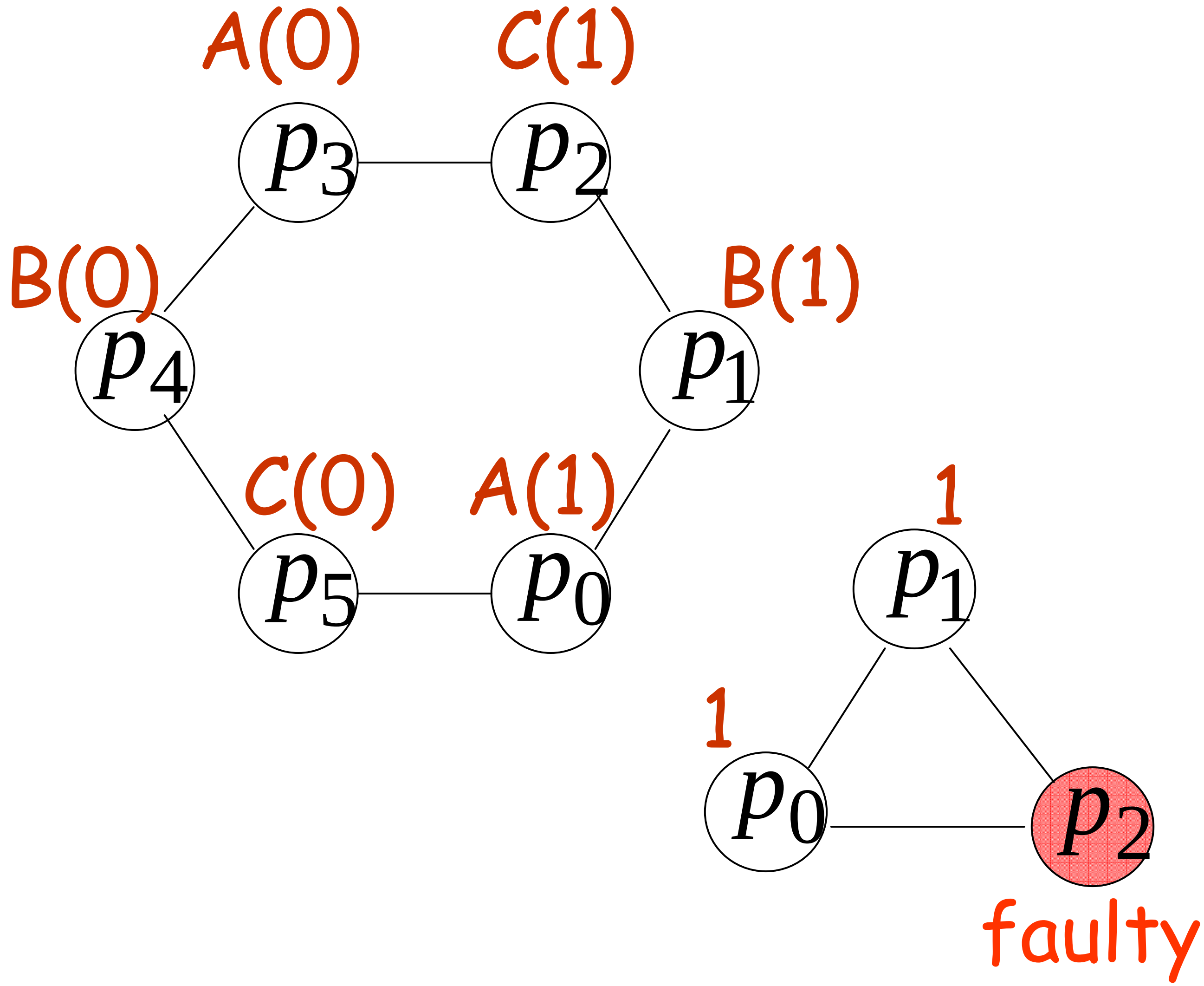


Assume 6 processes are in a ring

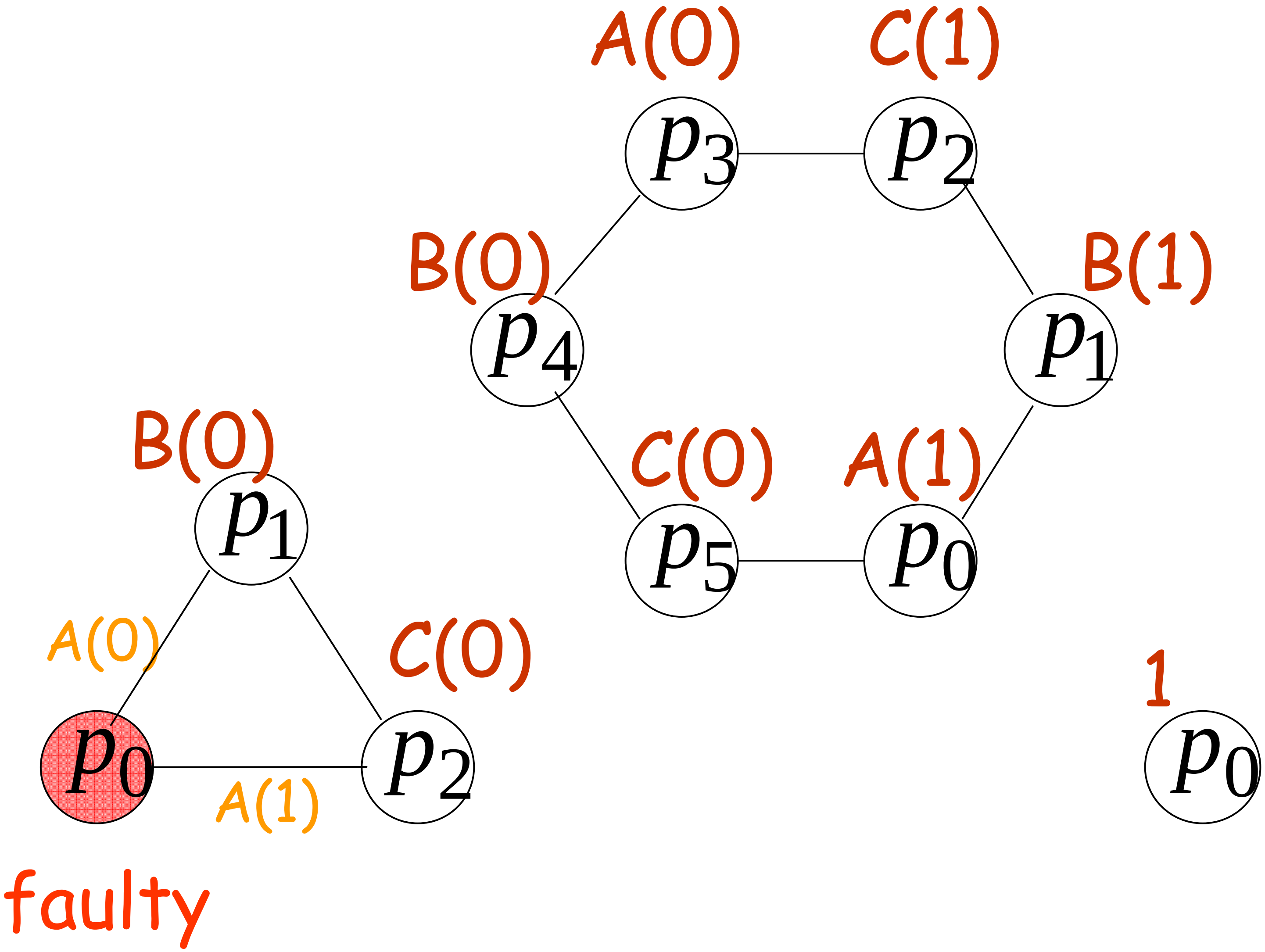
(just for fun)

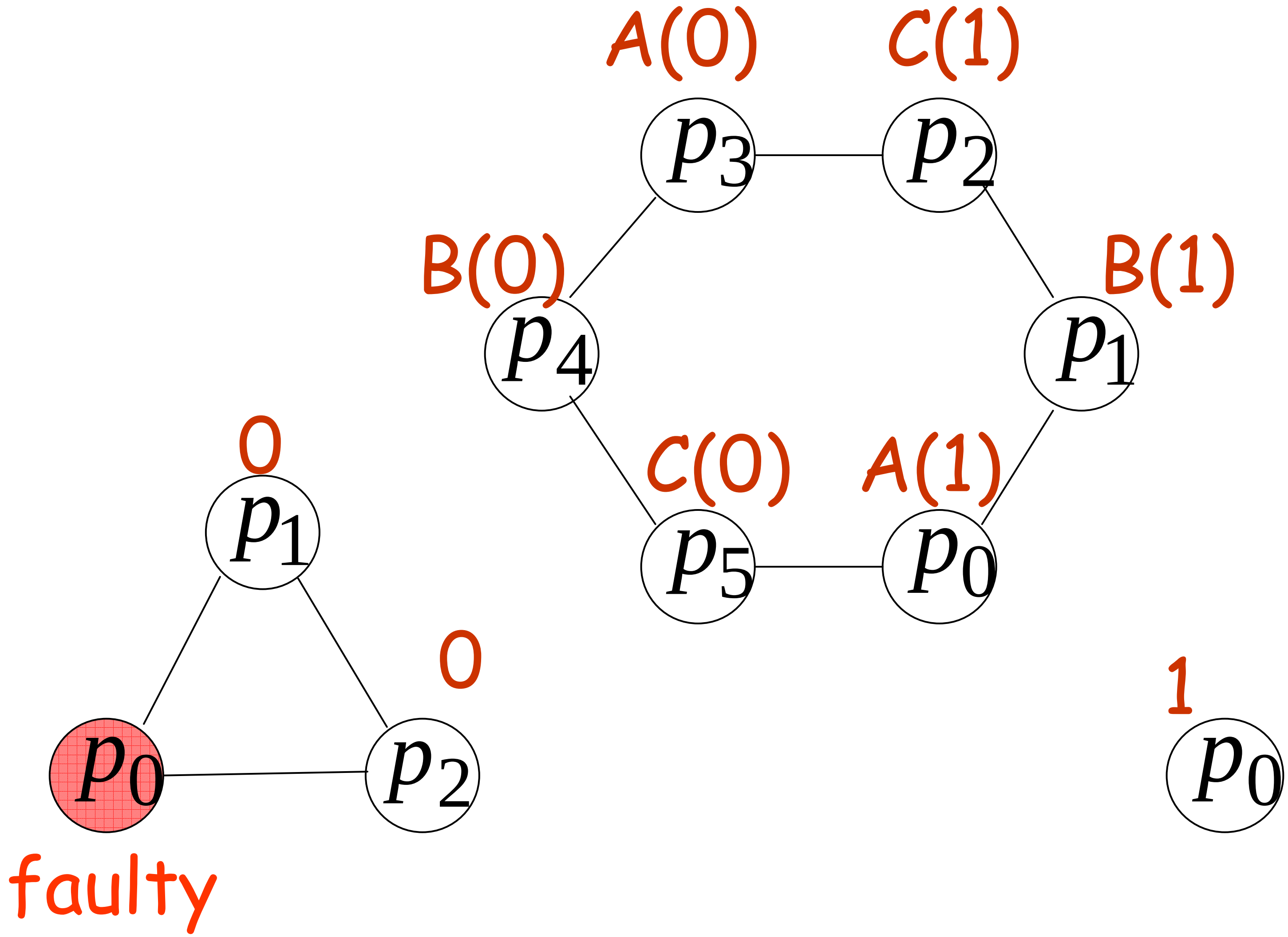


Processes think they are in a triangle

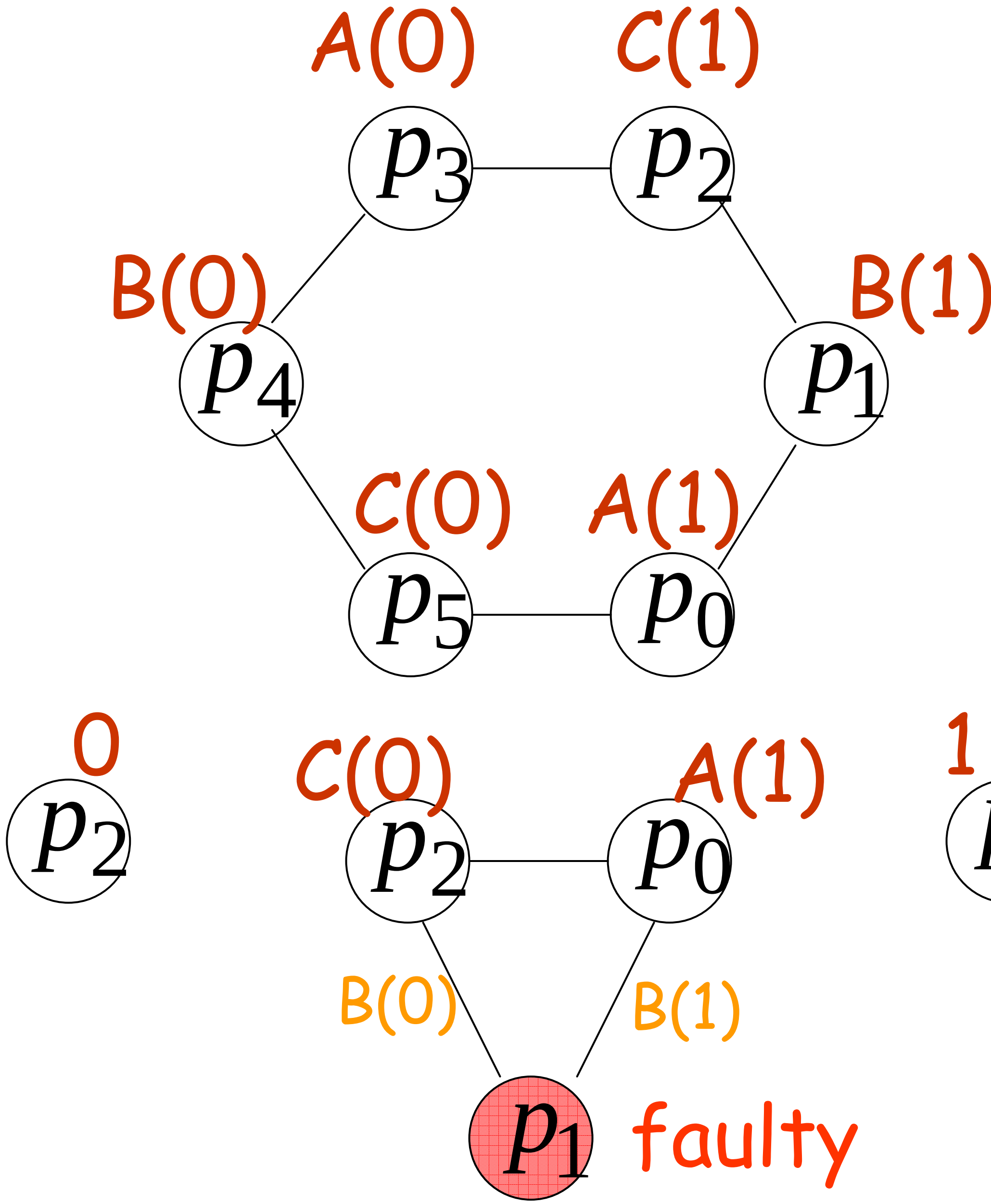


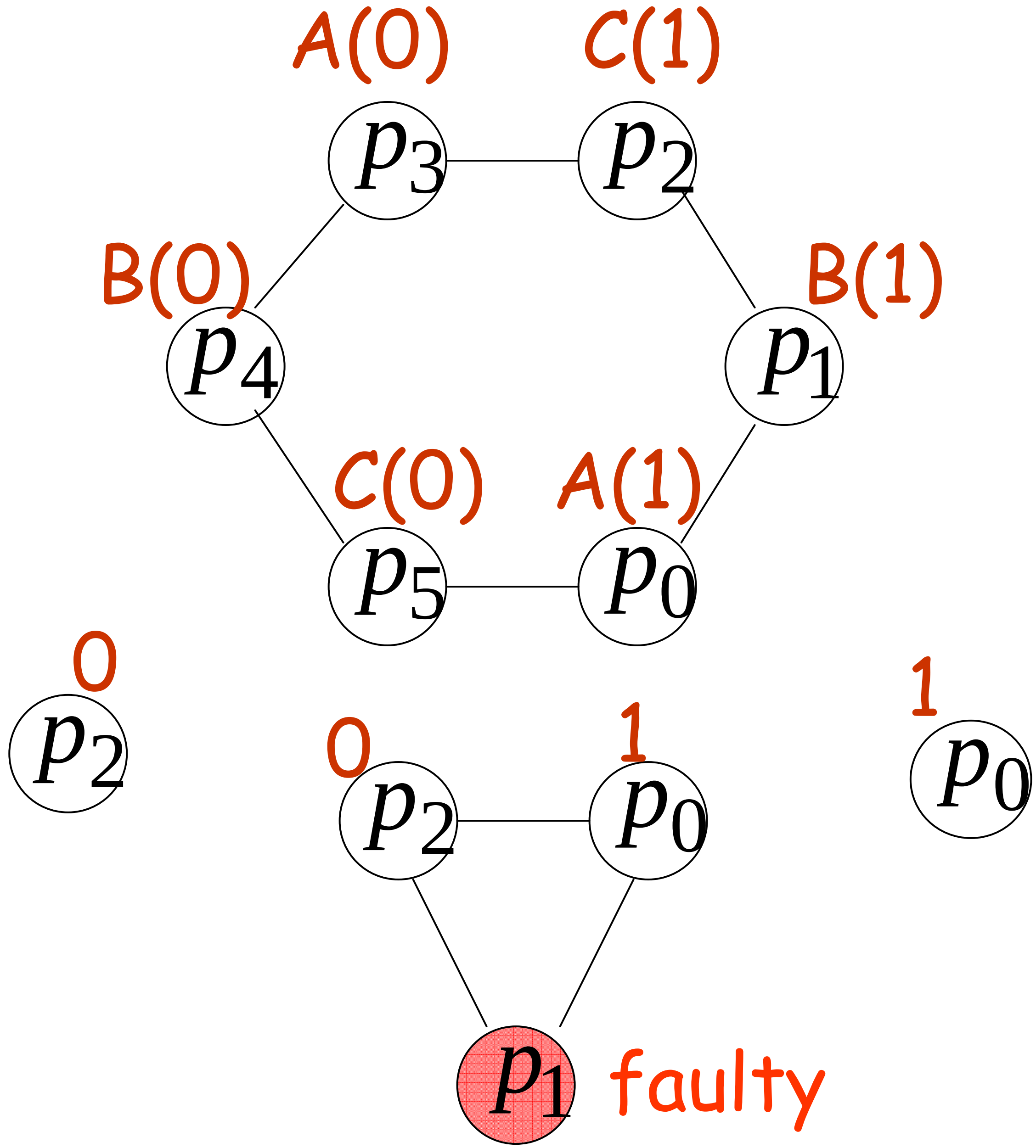
(validity condition)


A(0)
C(1)
B(0)
B(1)
B(0)
C(0)
A(1)
A(0)
C(0)
1
A(1)
faulty

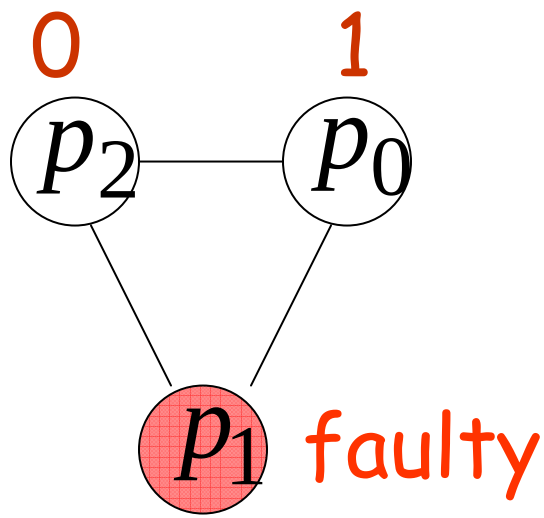


(validity condition)


A(0)
C(1)
p_3
p_2
B(0)
B(1)
p_4
p_1
C(0)
A(1)
p_5
p_0
0
1
C(0)
A(1)
p_2
p_2
p_0
p_0
B(0)
B(1)
p_1
faulty


A(0)
C(1)
B(0)
B(1)
C(0)
A(1)
p_3
p_2
p_4
p_1
p_5
p_0
0
p_2
0
1
p_2
p_0
1
p_0
p_1
faulty

Impossibility



Conclusion

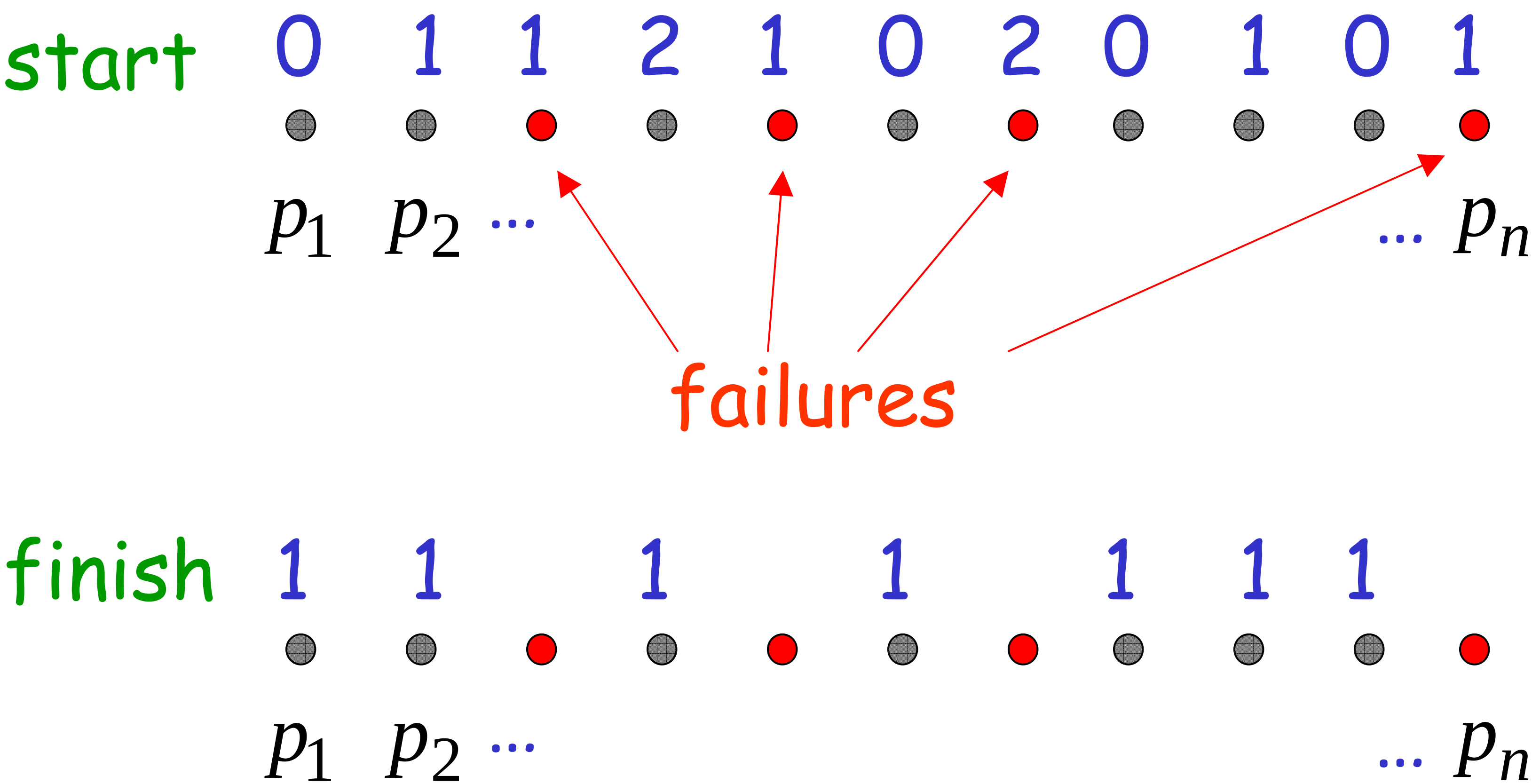
There is no algorithm that solves consensus for 3 processes in which 1 is a byzantine process

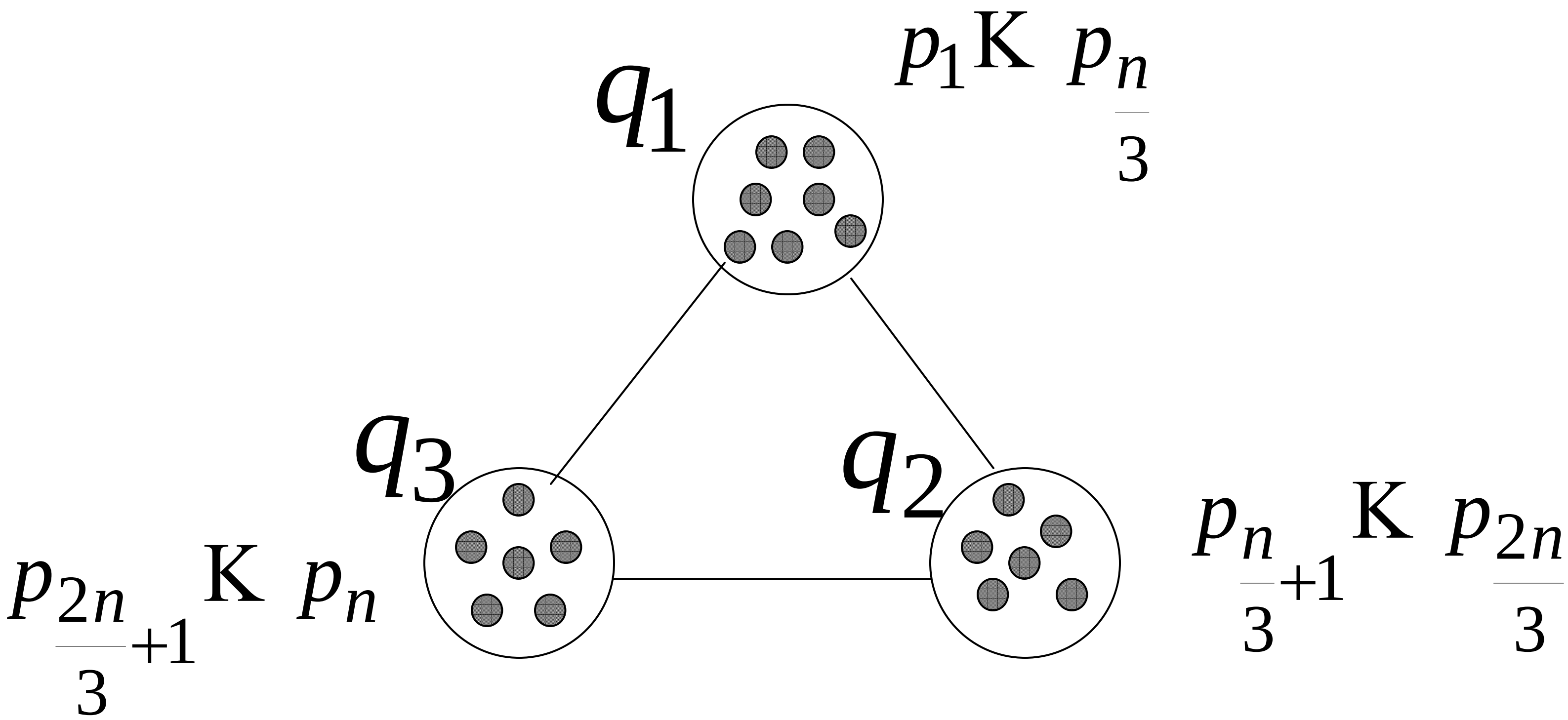
The n processes case

Assume for contradiction that there is an f-resilient algorithm A for n processes, where $f \geq n/3$

We will use algorithm A to solve consensus for 3 processes and 1 failure (which is impossible, thus we have a contradiction)

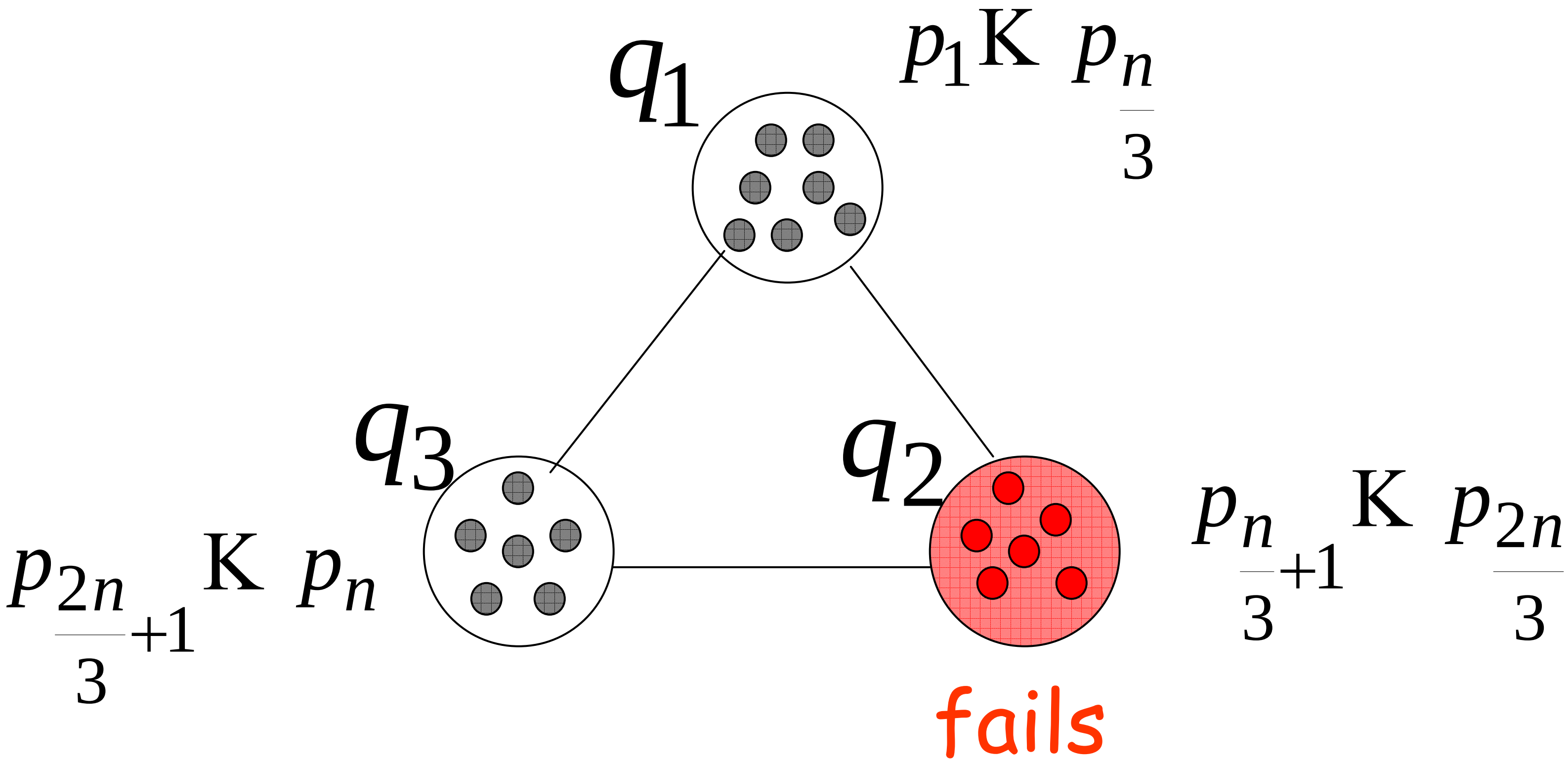
Algorithm A





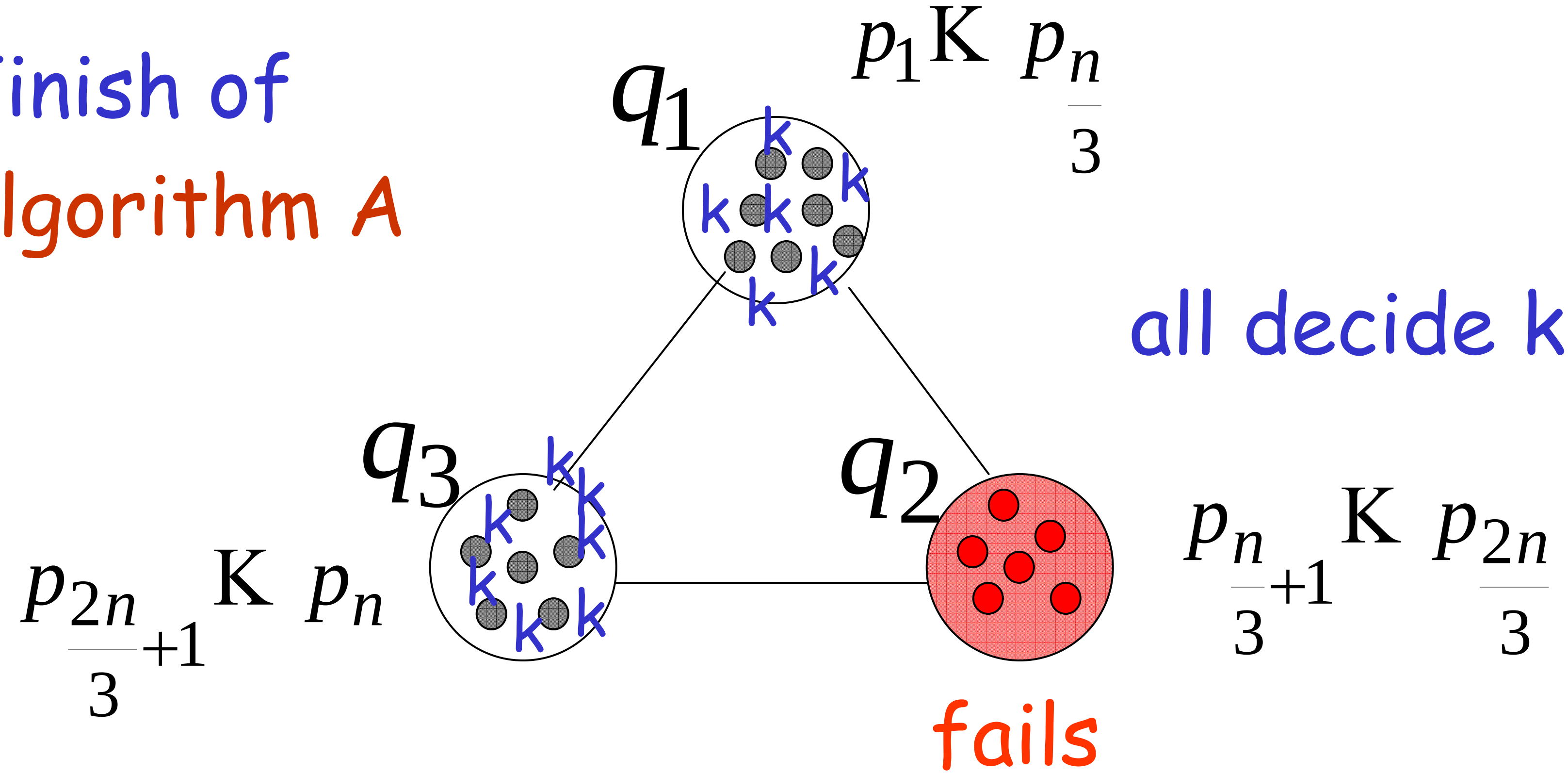
Each process *q* simulates algorithm *A*

on *n*/3 of "*p*" processes



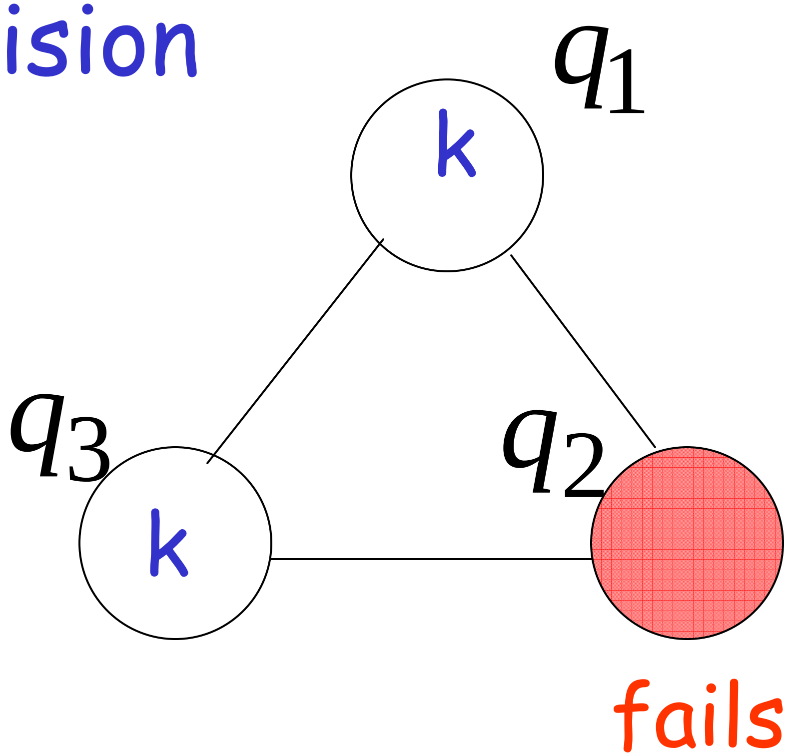
When a single *q* is byzantine, then *n*/3 of the "*p*" processes are byzantine too.

Finish of algorithm A



algorithm A tolerates $n/3$ failures

Final decision



We reached consensus with 1 failure

Impossible!!!

Conclusion

There is no f-resilient algorithm for n processes with $f \geq n/3$

The King Algorithm

solves consensus with n processes and f failures where $f < n/4$ in $f+1$ "phases"

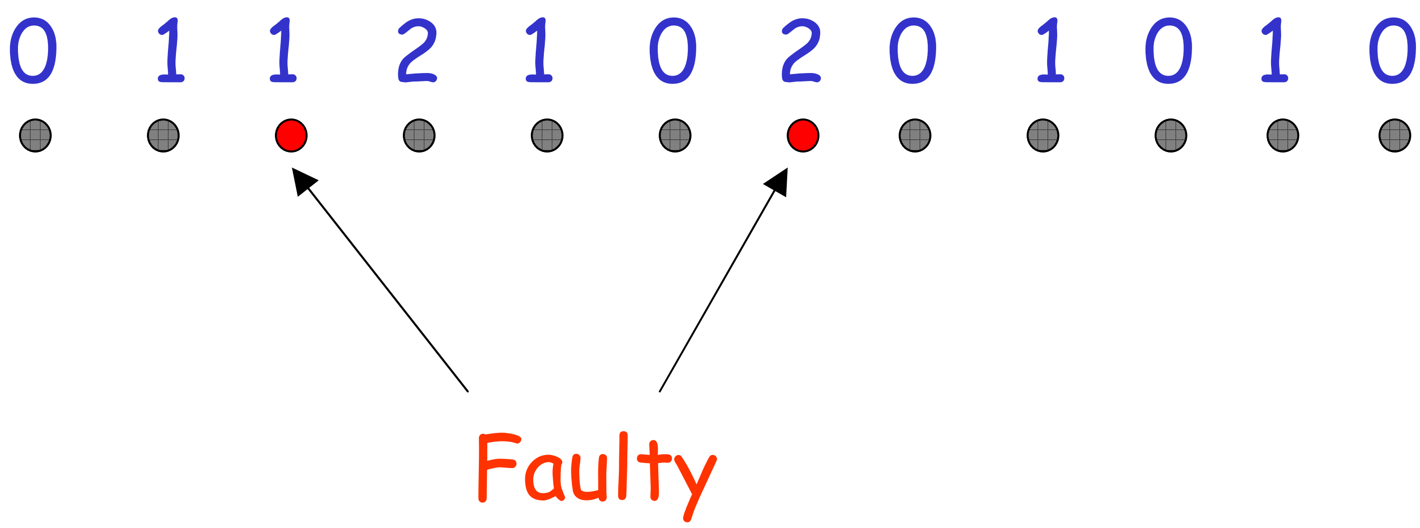
There are f+1 phases

Each phase has two rounds

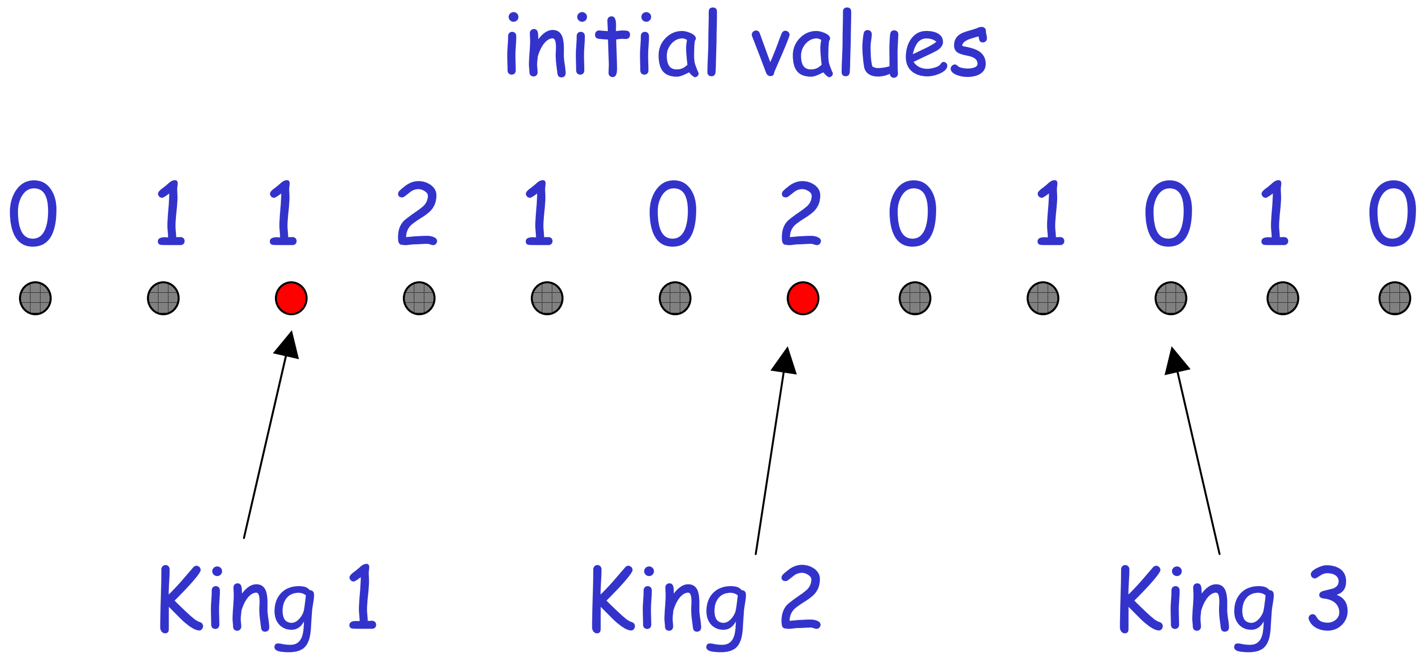
In each phase there is a different king

Example: 12 processes, 2 faults, 3 kings

initial values



Example: 12 processes, 2 faults, 3 kings



Remark: There is a king that is not faulty

The **King** algorithm

Each processor p_i has a preferred value v_i

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

The **King** algorithm: Phase k

Round 1, processor p_i :

- Broadcast preferred value v_i
- Set v_i to the majority of values received

The **King** algorithm: Phase k

Round 2, king p_k :

- Broadcast new preferred value v_k

Round 2, process p_i :

- If v_i had majority of less than $\frac{n}{2}+f$

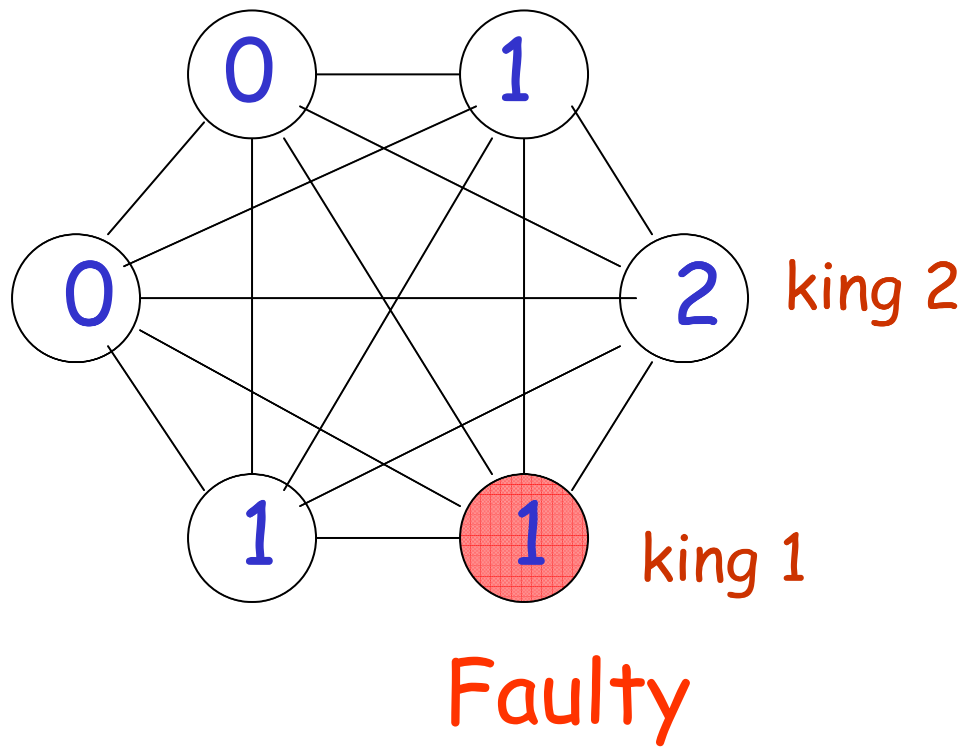
then set v_i to v_k

The **King** algorithm

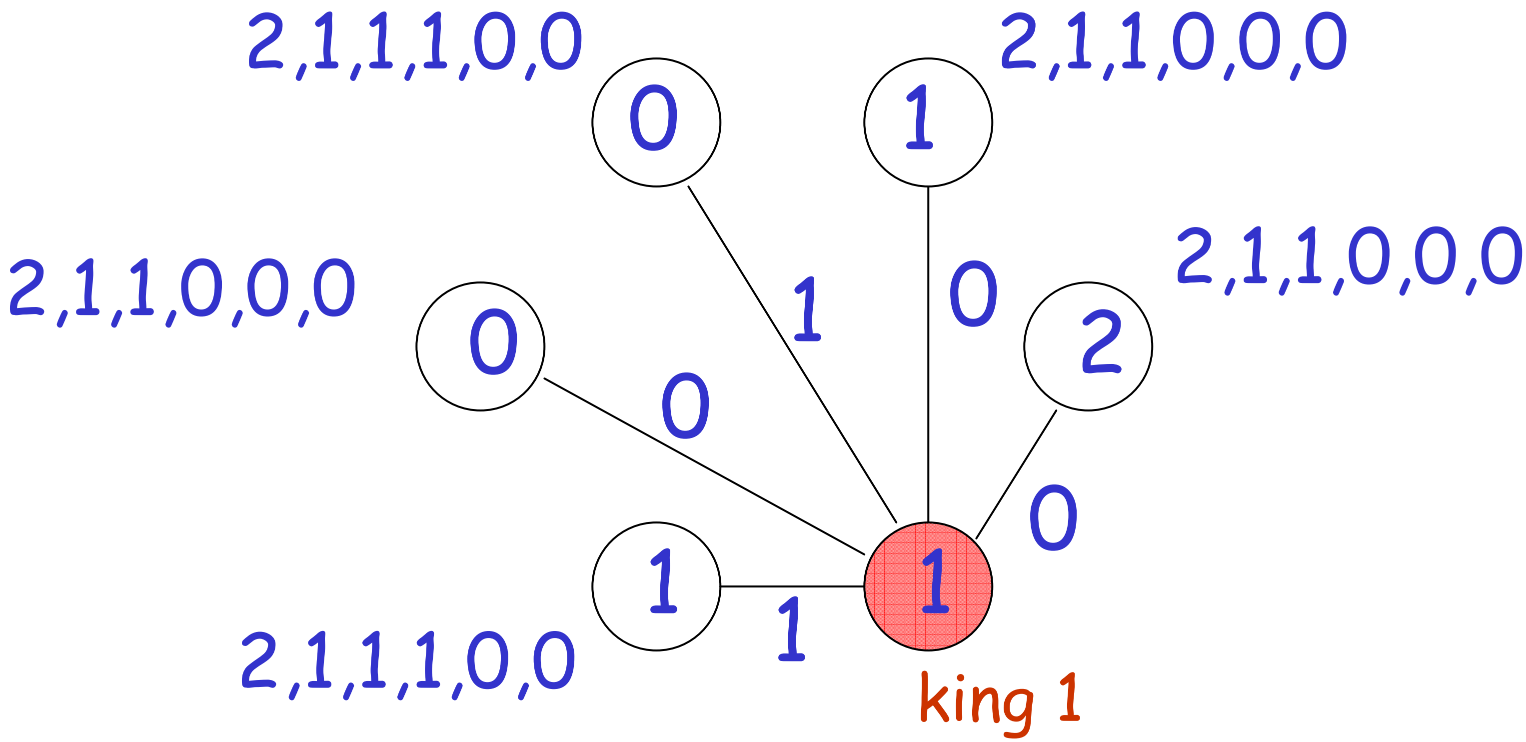
End of Phase f+1:

Each process decides on preferred value

Example: 6 processes, 1 fault

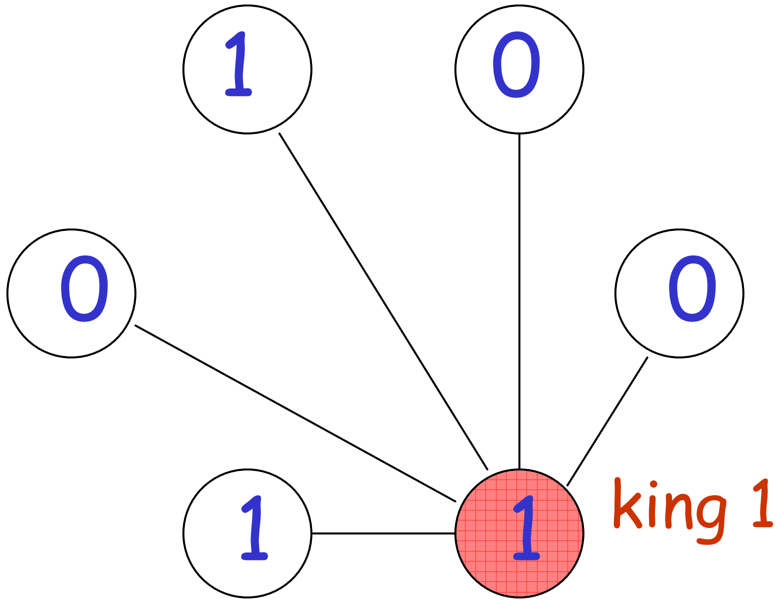


Phase 1, Round 1



Everybody broadcasts

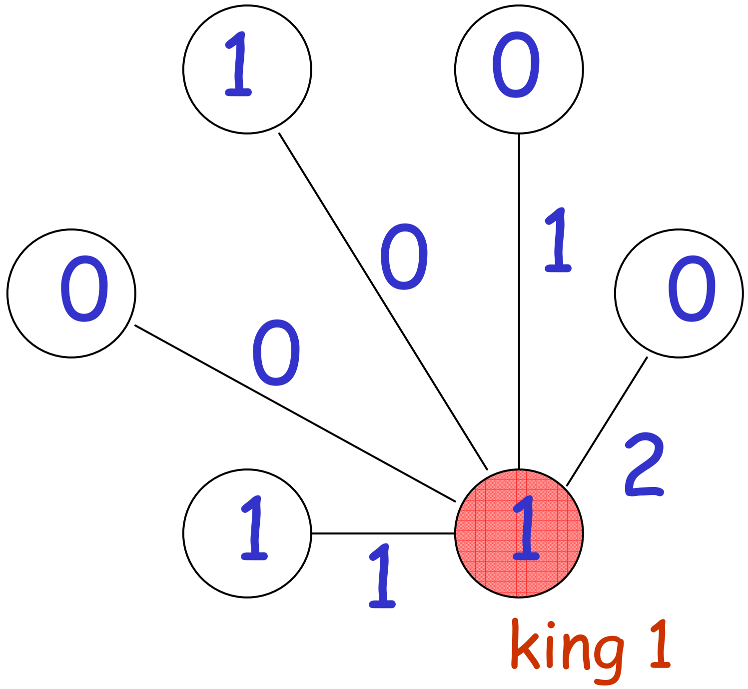
Phase 1, Round 1 — Choose the majority



Each majority population was $3 \leq \frac{n}{2} + f = 4$

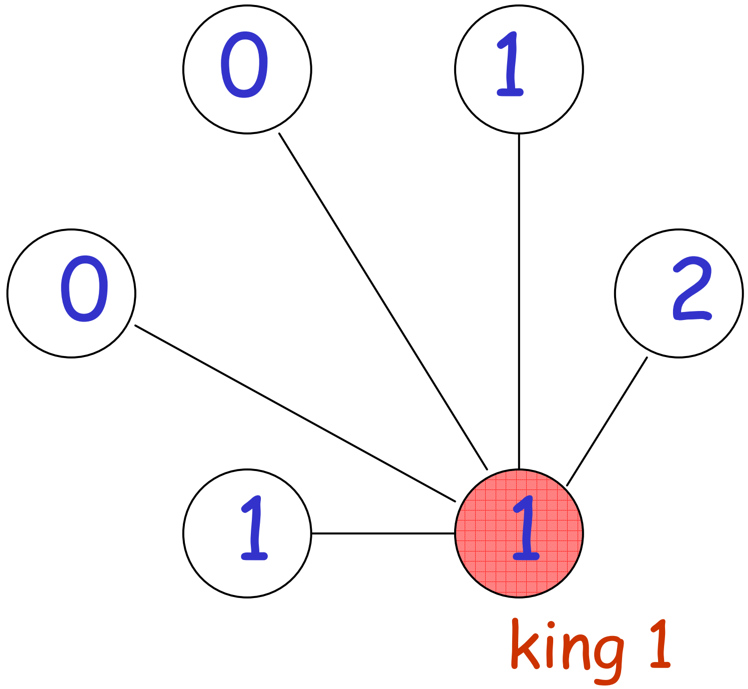
On round 2, everybody will choose the king's value

Phase 1, Round 2



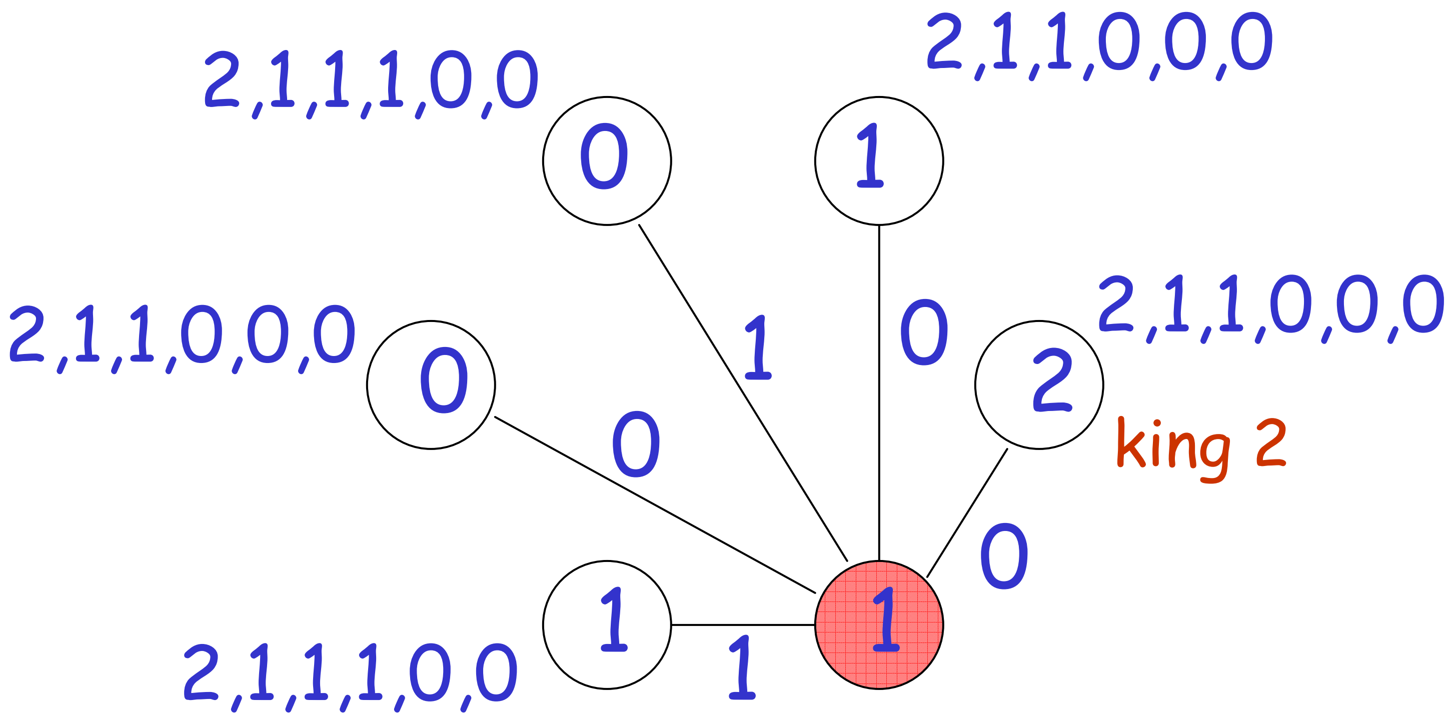
The king broadcasts

Phase 1, Round 2



Everybody chooses the king's value

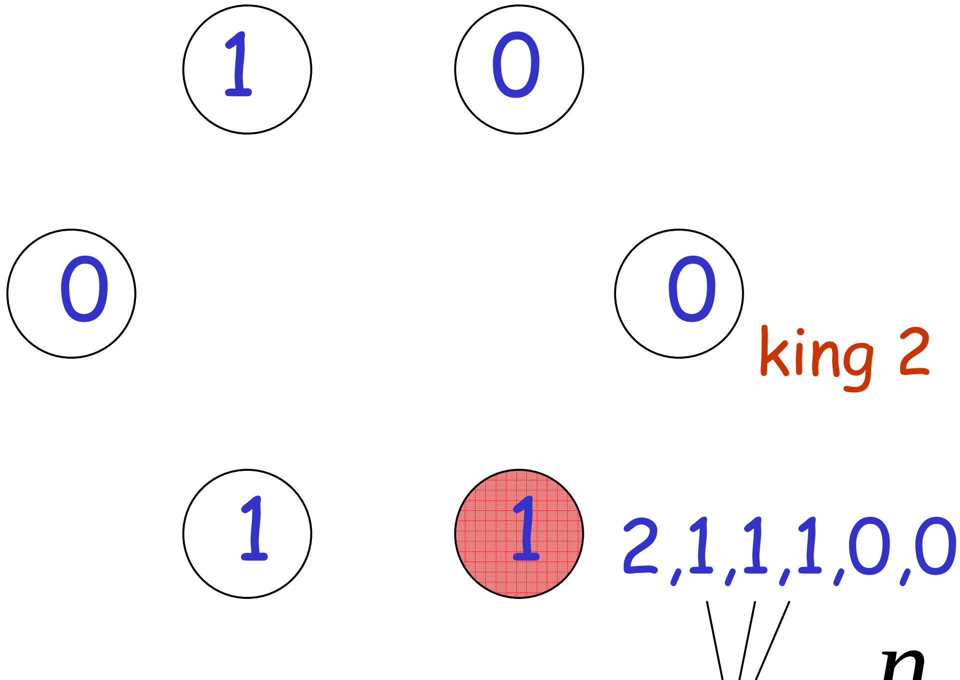
Phase 2, Round 1



Everybody broadcasts

Phase 2, Round 1

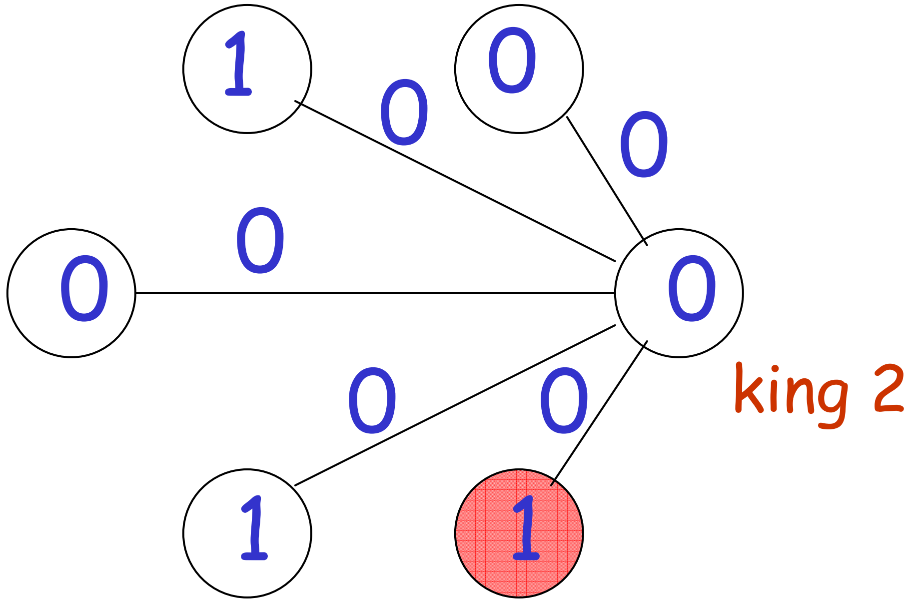
Choose the majority



Each majority population is $3 \le \frac{n}{2} + f = 4$

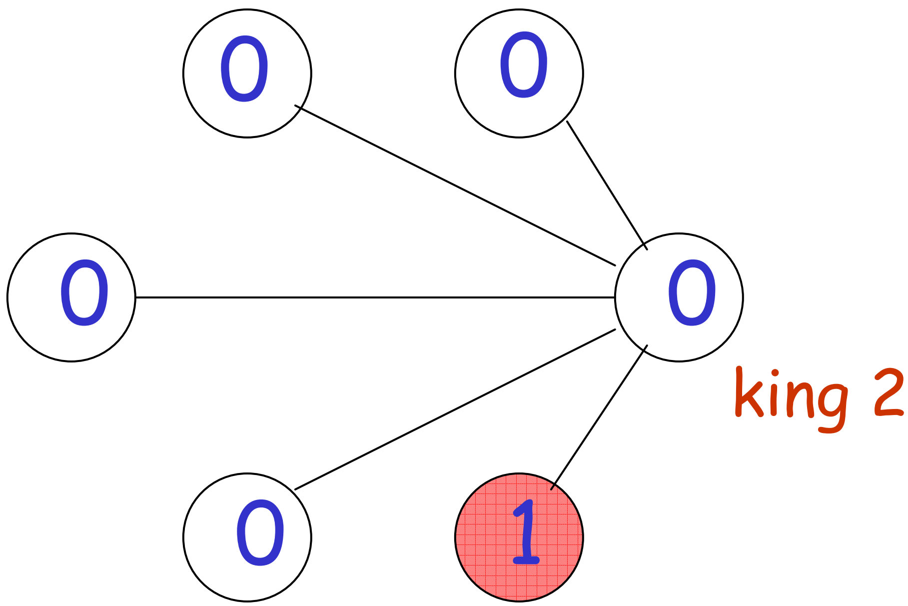
On round 2, everybody will choose the king's value

Phase 2, Round 2



The king broadcasts

Phase 2, Round 2



Everybody chooses the king's value

Final decision

Invariant / Conclusion

In the round where the king is non-faulty, everybody will choose the king's value **v**

After that round, the majority will remain value **v** with a majority population which is at least $n - f > \frac{n}{2} + f$

Exponential Algorithm

solves consensus with n processes and f failures where $f < n/3$ in $f+1$ "phases"

But: uses messages with exponential size

Consensus #6
Randomization

- So far we looked at deterministic algorithms only. We have seen that there is no asynchronous algorithm.

- Can one solve consensus if we allow our algorithms to use randomization?

Yes, we can!

- We tolerate some processes to be faulty (at most f stop failures)
- General idea: Try to push your initial value; if other processes do not follow, try to push one of the suggested values randomly.

Randomized Algorithm

- At most f stop-failures (assume $n > 9f$)
- For process p_i with initial input $x \in \{0,1\}$:

1. Broadcast Proposal(x, round)
2. Wait for n-f Proposal messages.
3. If at least n-2f messages have value v, then x := v, else x := undecided.

Randomized Algorithm

4. Broadcast Bid(x, round).
5. Wait for n-f Bid messages.
6. If at least n-2f messages have value v,
   then decide on v.
   If at least n-4f messages have value v,
   then x := v.
   Else choose x randomly ($p(0) = p(1) = \frac{1}{2}$)
7. Go back to step 1 (next round).

What do we want?

- Agreement: Non-faulty processes decide non-conflicting values.
- Validity: If all have the same input, that input should be decided.
- Termination: All non-faulty processes *eventually* decide.

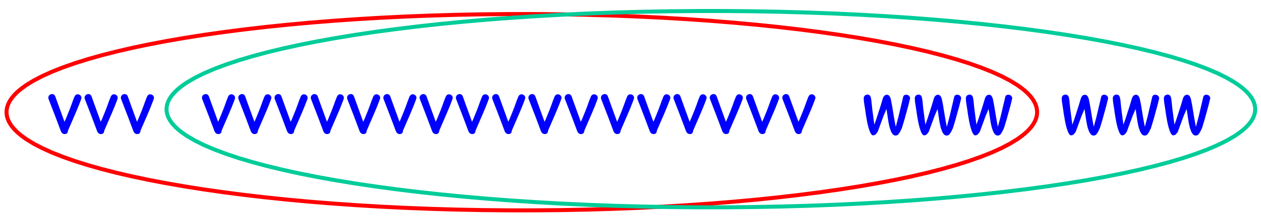
All processes have same input

- Then everybody will agree on that input in the very first round already.
- Validity follows immediately

- If not, then any decision is fine!
- Validity follows too (in any case).

What if process i decides in step 6a (Agreement)...?

- Then process i has received at least n-2f Bid messages with value v.



- Then everybody else has received at least n-3f messages will value v, and thus everybody will propose v next round, and thus decide v.

What about termination?

- We have seen that if a process decides in step 6a, all others will follow in the next round at latest.
- If in step 6b/c, all processes choose the same value (with probability 2^{-n}), all give the same bid, and terminate in the next round.

Byzantine & Asynchronous?

- The presented protocol is in fact already working in the Byzantine case!
- (That's why we have "n-4f" in the protocol and "n-3f" in the proof.)

But termination is awfully slow…

- In expectation, about the same number of processes will choose 1 or 0 in step 6c.
- The probability that a strong majority of processes will propose the same value in the next round is exponentially small.

Naïve Approach

- In step 6c, all processes should choose the same value! (Reason: validity is not a problem anymore since for sure there exist 0's and 1's and therefore we can savely always propose the same…)
- Replace 6c by: "choose x := 1"!

Problem of Naïve Approach

- What if a majority of processes bid 0 in round 4? Then some of the processes might go into 6b (setting x=0), others into 6c (setting x=1). Then the picture is again not clear in the next round
- Anyway: Approach 1 is deterministic! We know (#2) that this doesn't work!

Shared/Common Coin

- The idea is to replace 6c with a subroutine where all the processes compute a so-called shared (a.k.a. common, "global") coin.
- A shared coin is a random binary variable that is 0 with constant probability, and 1 with constant probability.

Shared Coin Algorithm

Code for process i:

1. Set local coin c_i := 0 with probability 1/n, else (w.h.p.) c_i := 1.
2. Use reliable broadcast* to tell all processes about your local coin c_i.
3. If you receive a local coin c_j of another process j, add j to the set $coins_i$, and memorize c_j.

Shared Coin Algorithm

4. If you have seen exactly n-f local coins then copy the set $coins_i$ into the set $seen_i$ (but do not stop extending $coins_i$ if you see new coins)
5. Use reliable broadcast to tell all processes about your set $seen_i$.

Shared Coin Algorithm

6. If you have seen at least n-f $seen_j$ which satisfy $seen_j \subseteq coins_i$, then terminate with:
7. If you have seen at least a single local coin with $c_j = 0$ then return 0, else (if you have seen 1-coins only) return 1.

Why does the shared coin algorithm terminate?

- For simplicity we look at f crash failures only, assuming that $3f < n$.
- Since at most f processes crash you will see at least n-f local coins in step 4.
- For the same reason you will see at least n-f seen sets in step 6.
- Since we used reliable broadcast, you will eventually see all the coins that are in the other's sets.

Why does the algorithm work?

- Looks like magic at first…
- General idea: a third of the local coins will be seen by all the processes! If there is a "0" among them we're done. If not, chances are high that there is no "0" at all.
- Proof details: next few slides…

Proof: Matrix

- Let i be the first process to terminate (reach step 7)
- For process i we draw a matrix of all the sets $seen_j$ (columns) and local coins c_k (rows) process i has seen.
- We draw an "X" in the matrix if and only if set $seen_j$ includes coin c_k.

Proof: Matrix (f=2, n=7, n-f=5)

	$seen_1$	$seen_3$	$seen_5$	$seen_6$	$seen_7$
$coin_1$	X	X	X	X	X
$coin_2$			X	X	X
$coin_3$	X	X	X	X	X
$coin_5$	X	X	X		X
$coin_6$	X	X	X	X	
$coin_7$	X	X		X	X

- Note that there are at least $(n-f)^2$ X's in this matrix ($\geq$n-f rows, n-f X's in each row).

Proof: Matrix

- Lemma 1: There are at least f+1 rows where at least f+1 cells have an "X".
- Proof: Suppose by contradiction that this is not the case. Then the number of X is bounded from above by $f \cdot (n-f) + (n-f) \cdot f$, ...

Few rows have many X

All other rows have at most f X

Proof: Matrix

$|X| \leq 2f(n-f)$

we use $3f < n \rightarrow 2f < n-f$

$< (n-f)^2$

but we know that $|X| \geq (n-f)^2$

$\leq |X|$.

A contradiction!

Proof: The set W

- Let W be the set of local coins where the rows in the matrix have more than f X's.
- Lemma 2: All local coins in the set W are seen by all processes (that terminate).
- Proof: Let $w \in W$ be such a local coin. With Lemma 1 we know that w is at least in f+1 seen sets. Since each process must see at least n-f seen sets (before terminating), these sets overlap, and w will be seen.

Proof: End game

- Theorem: With constant probability all processes decide 0, with constant probability all processes decide 1.
- Proof: With probability $(1-1/n)^n \approx 1/e$ all processes choose $c_i = 1$, and therefore all will decide 1.
- With probability $1-((1-1/n)^{|W|})$ there is at least one 0 in the set W. Since $|W| \approx n/3$ this probability is constant. Using Lemma 2 we know that in this case all processes will decide 0.

Back to Randomized Consensus

- Plugging the shared coin back into the randomized consensus algorithm is all we needed.
- If some of the processes go into 6b and, the others still have a constant chance that they will agree on the same shared coin.
- The randomized consensus protocol finishes in a constant number of rounds!

Improvements

- For crash-failures, there is a constant expected time algorithm which tolerates f failures with $2f < n$.
- For Byzantine failures, there is a constant expected time algorithm which tolerates f failures with $3f < n$.
- Similar algorithms have been proposed for the shared memory model.

Databases et al.

- Consensus plays a vital role in many distributed systems, most notably in distributed databases:
  - Two-Phase-Commit (2PC)
  - Three-Phase-Commit (3PC)

Summary

- We have solved consensus in a variety of models; particularly we have seen
  - algorithms
  - wrong algorithms
  - lower bounds
  - impossibility results
  - reductions
  - etc.

Credits

- The impossibility result (#2) is from Fischer, Lynch, Patterson, 1985.
- The hierarchy (#3) is from Herlihy, 1991.
- The synchronous studies (#4) are from Dolev and Strong, 1983, and others.
- The Byzantine studies (#5) are from Lamport, Shostak, Pease, 1980ff., and others.
- The first randomized algorithm (#6) is from Ben-Or, 1983.

Questions?

Distributed
Computing
Group
Roger Wattenhofer