



Chapter 6

Consensus

Distributed Systems

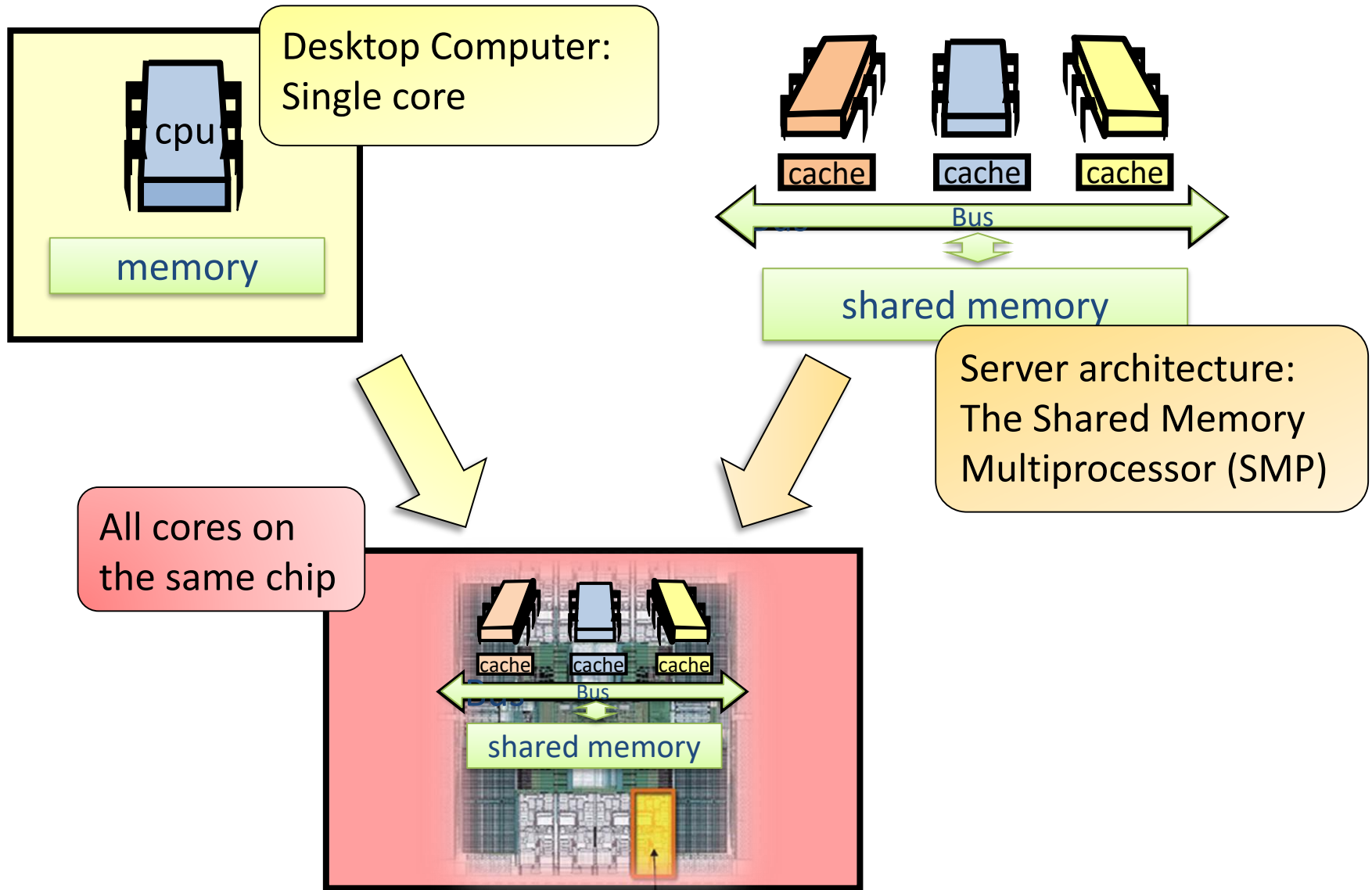
SS 2015

Fabian Kuhn

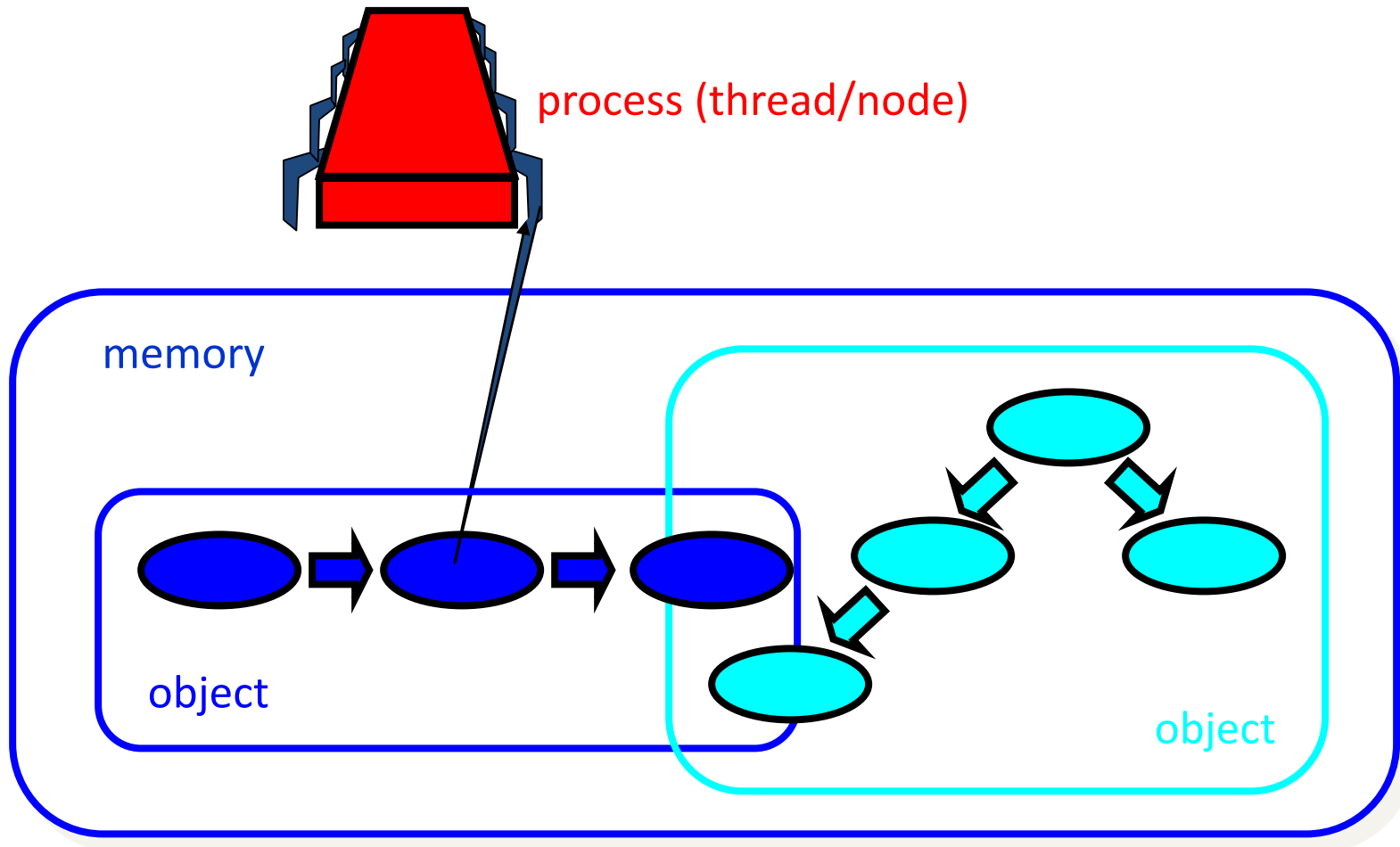
- Introduction
- Consensus #1: Shared Memory
- Consensus #2: Wait-free Shared Memory
- Consensus #3: Read-Modify-Write Shared Memory
- Consensus #4: Synchronous Systems
- Consensus #5: Byzantine Failures
- Consensus #6: A Simple Algorithm for Byzantine Agreement
- Consensus #7: The Queen Algorithm
- Consensus #8: The King Algorithm
- Consensus #9: Byzantine Agreement Using Authentication
- Consensus #10: A Randomized Algorithm
- Shared Coin

- Slides by R. Wattenhofer (ETHZ)

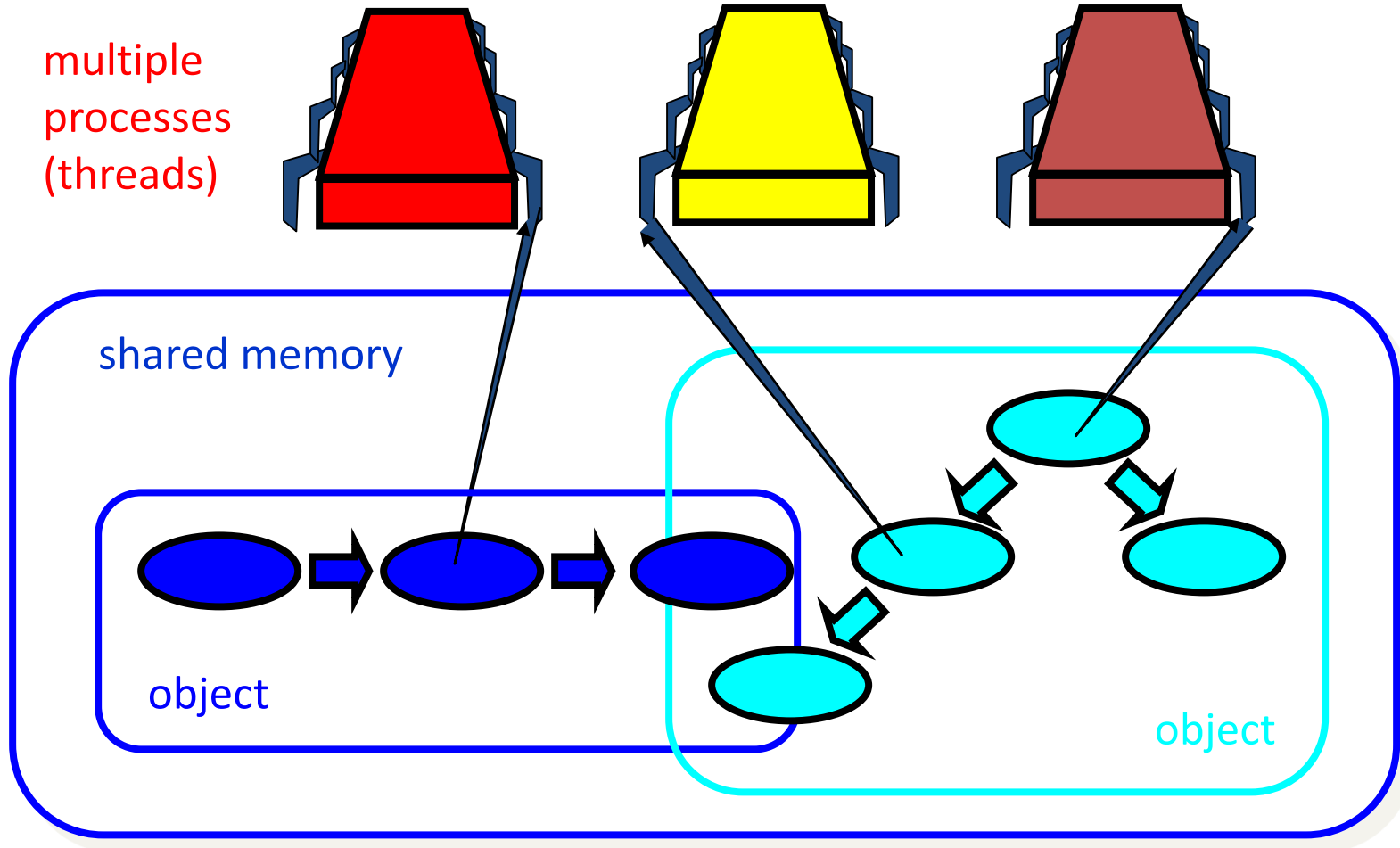
From Single-Core to Multicore Computers



Sequential Computation

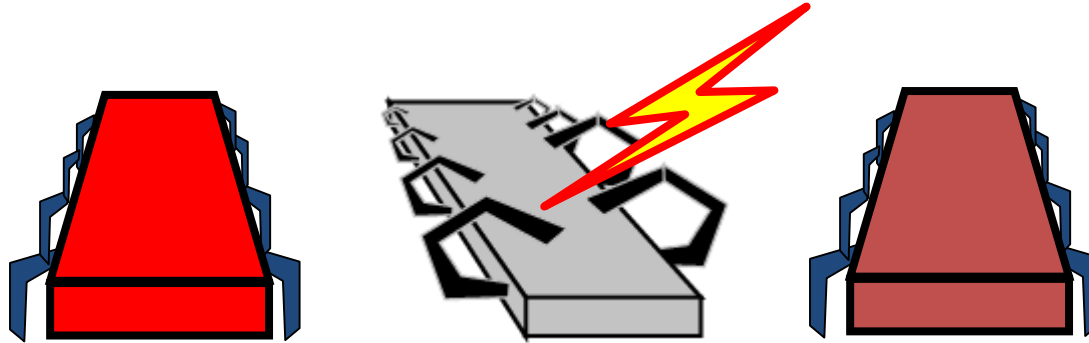


Concurrent Computation



Fault Tolerance & Asynchrony

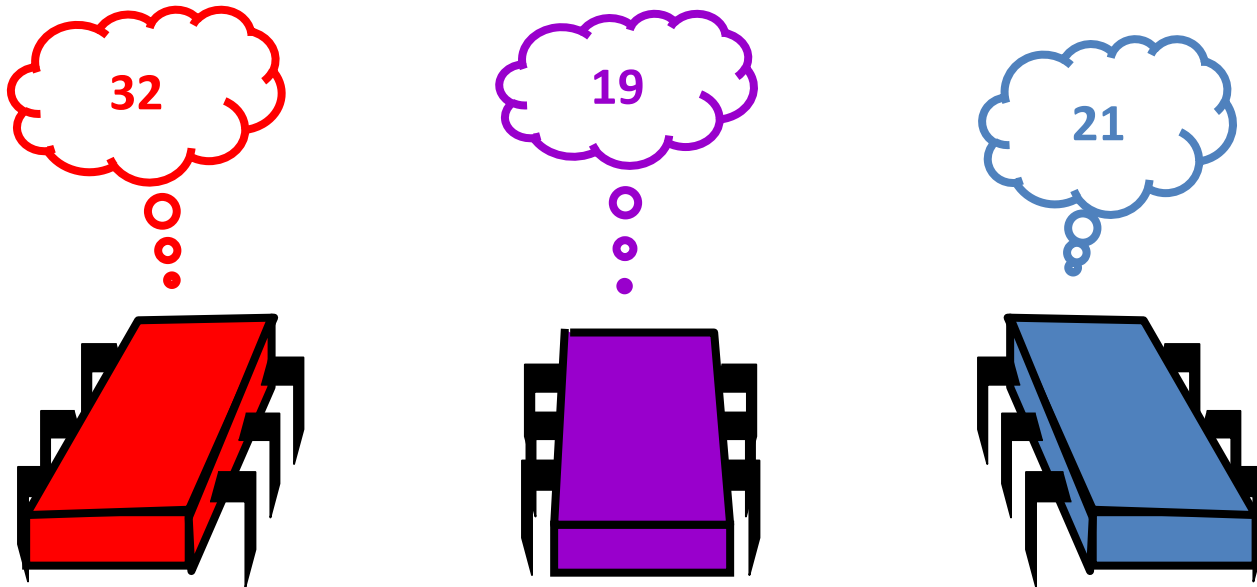
processes



- Why fault-tolerance?
 - Even if processes do not die, there are “near-death experiences”
- Sudden unpredictable delays:
 - Cache misses (short)
 - Page faults (long)
 - Scheduling quantum used up (really long)

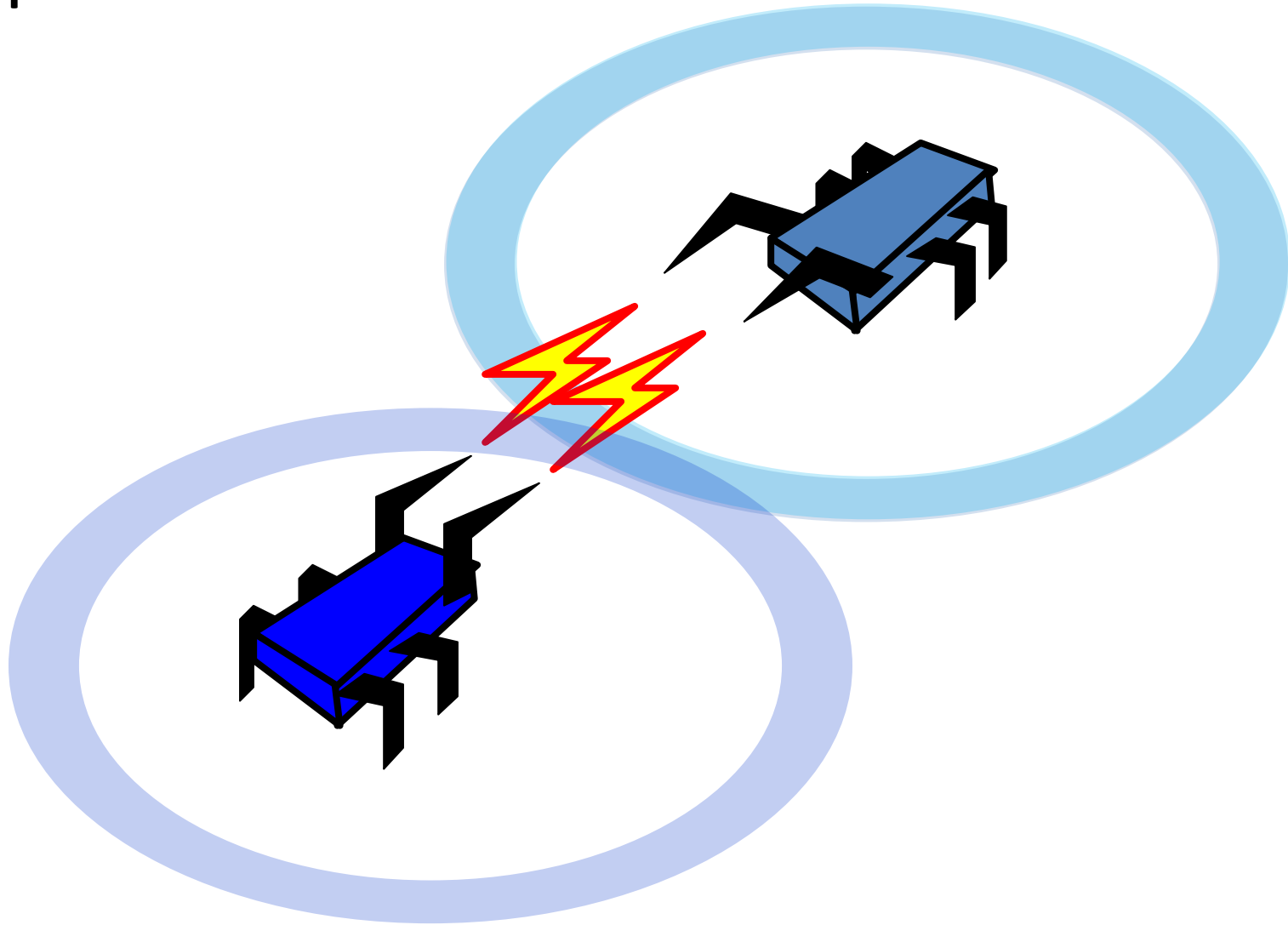
Consensus

Each thread/process has a private input



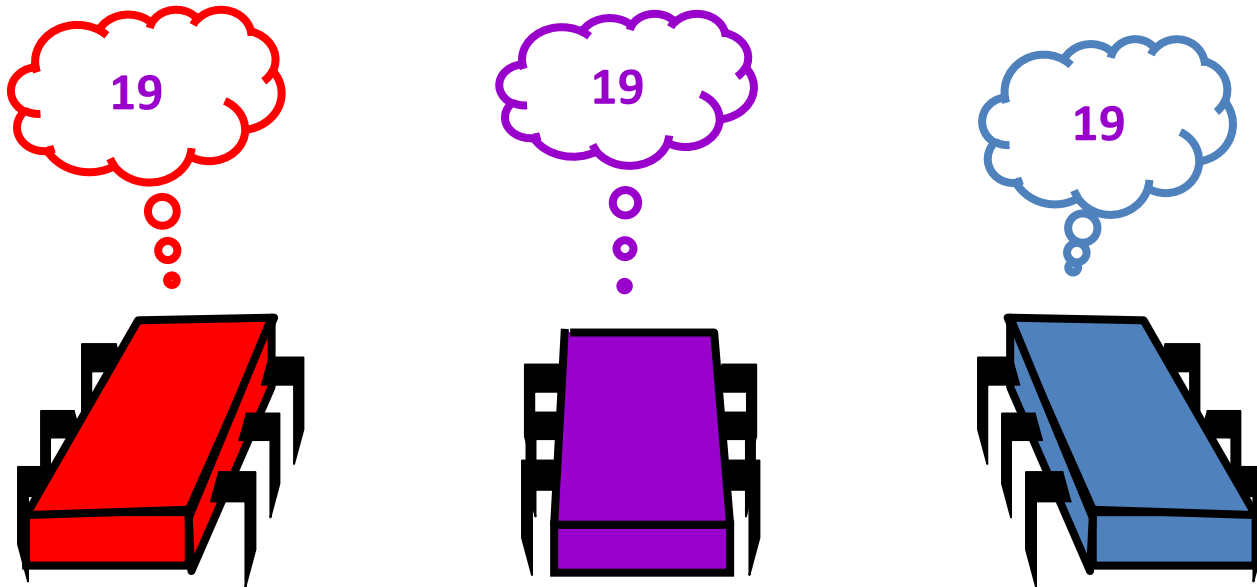
Consensus

The processes communicate



Consensus

They agree on some process's input



Consensus More Formally

Setting:

- n processes/threads/nodes v_1, v_2, \dots, v_n
- Each process has an input $x_1, x_2, \dots, x_n \in \mathcal{D}$
- Each (non-failing) process computes an output $y_1, y_2, \dots, y_n \in \mathcal{D}$

Agreement:

The outputs of all non-failing processes are equal.

Validity:

If all inputs are equal to x , all outputs are equal to x .

Termination:

All non-failing processes terminate after a finite number of steps.

Remarks

- Validity might sometimes depend on the (failure) model

Two Generals:

- The two generals (coordinated attack) problem is a variant of 2-node, binary consensus.
- Model: Communication is synchronous, messages can be lost
- Validity: If no messages are lost, and both nodes have the same input x , x needs to be the output
- We have seen that the problem cannot be solved in this setting.

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
 - and much more...
- We will see that in some models, consensus is possible, in some other models, it is not
- The goal is to learn whether for a given model consensus is possible or not ... and prove it!

Consensus #1: Shared Memory

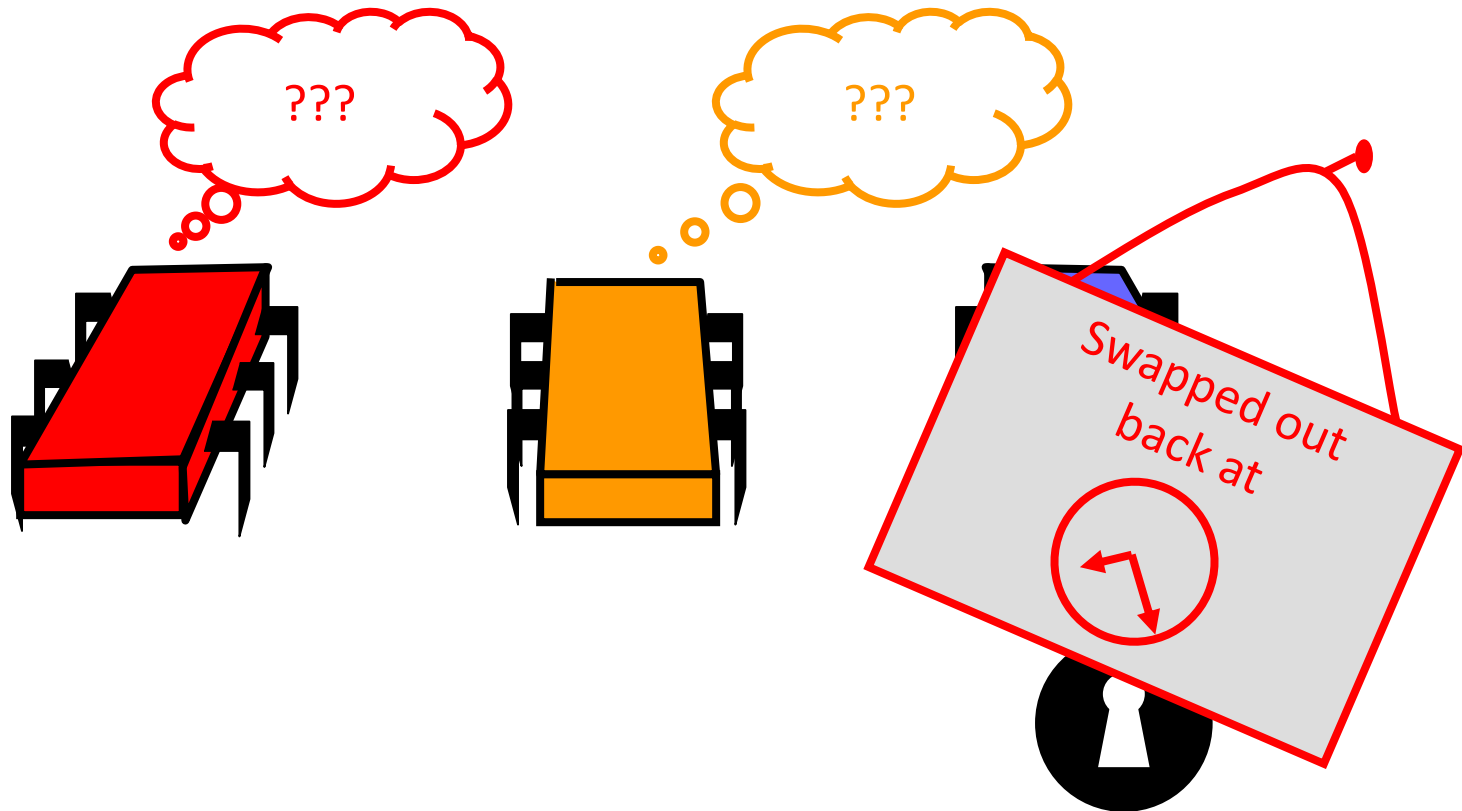
- $n > 1$ processors
- Shared memory is memory that may be accessed simultaneously by multiple threads/processes.
- Processors can atomically *read* or *write* (not both) a shared memory cell

Protocol:

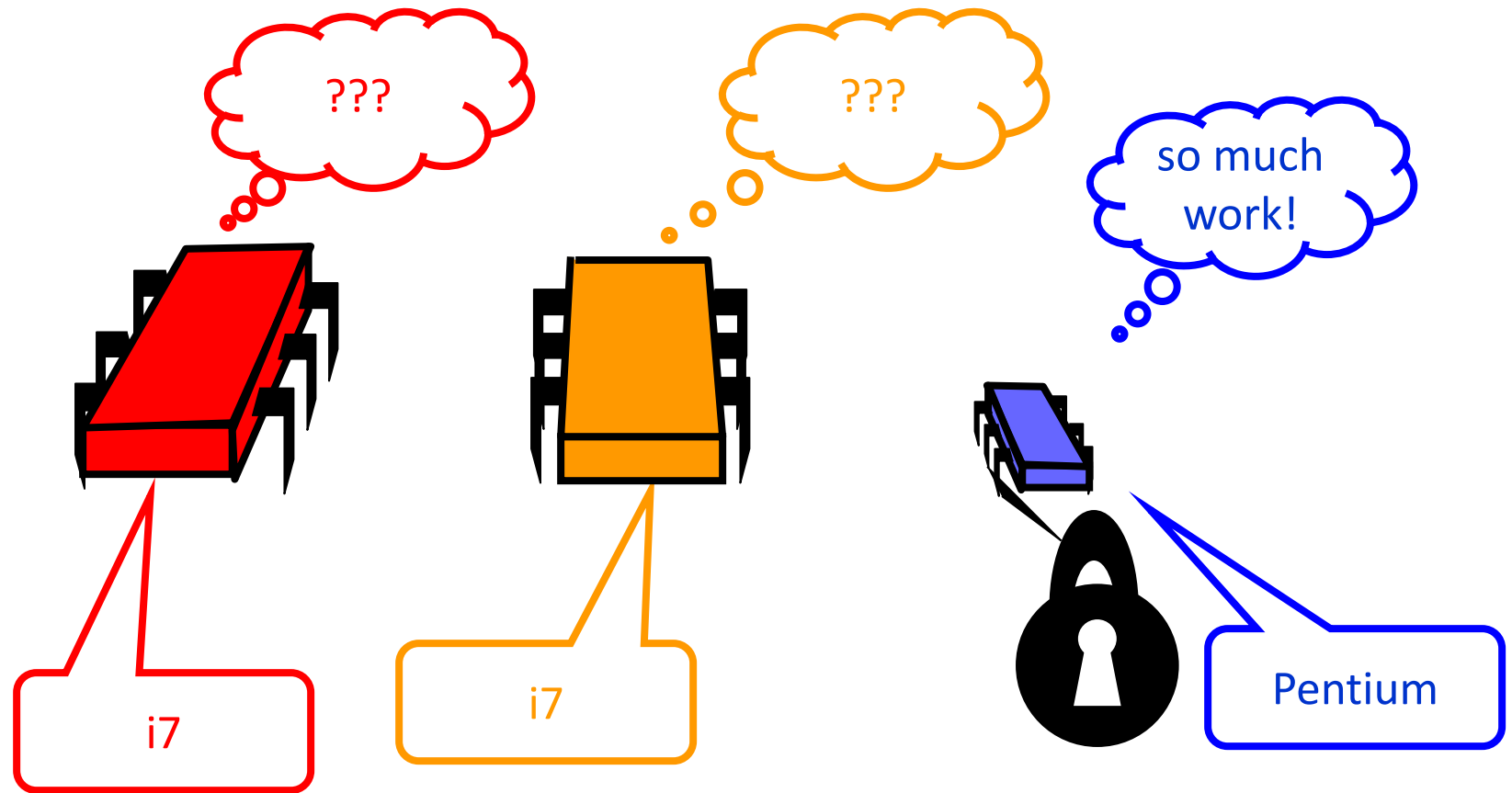
- There is a designated memory cell c .
- Initially c is in a special state “?”
- Processor 1 writes its value v_1 into c , then decides on v_1 .
- A processor $j \neq 1$ reads c until j reads something else than “?”, and then decides on that.

- Problems with this approach?

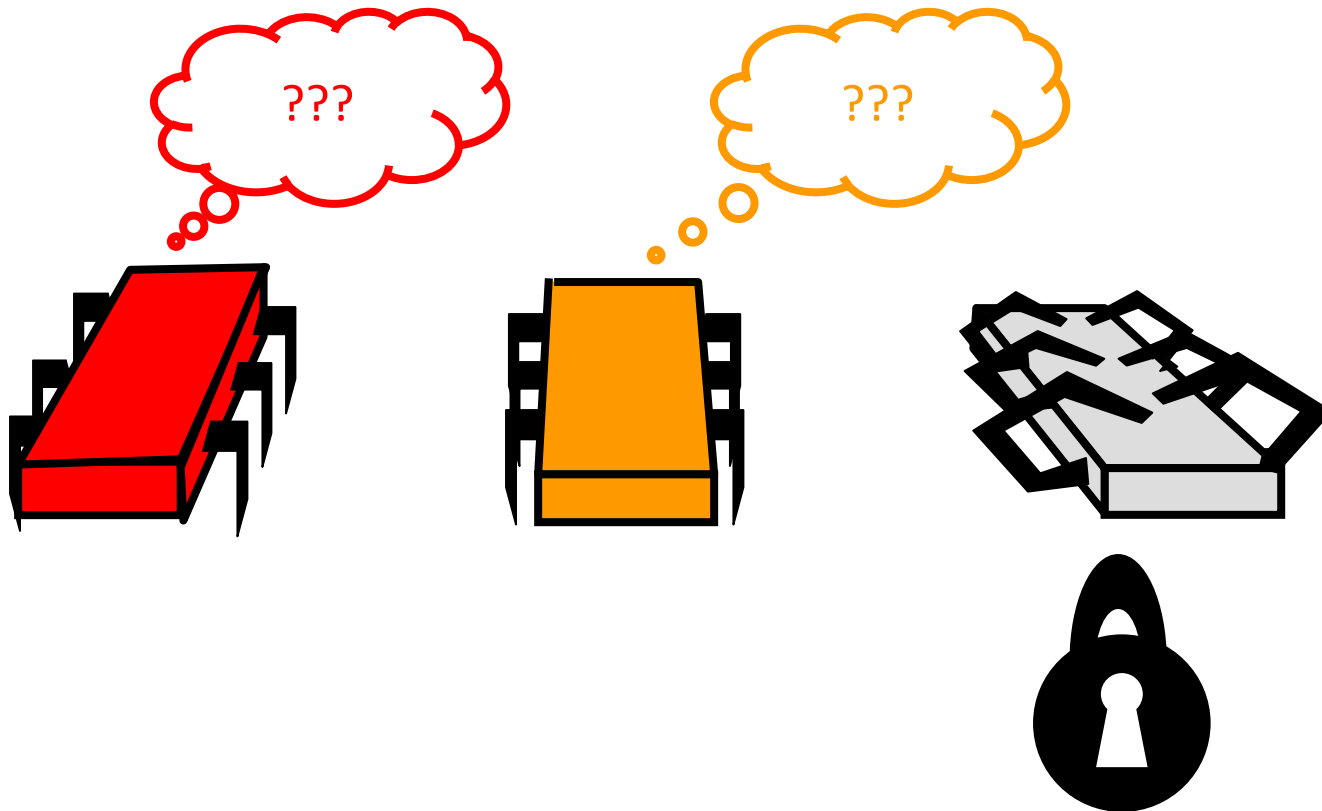
Unexpected Delay



Heterogeneous Architectures

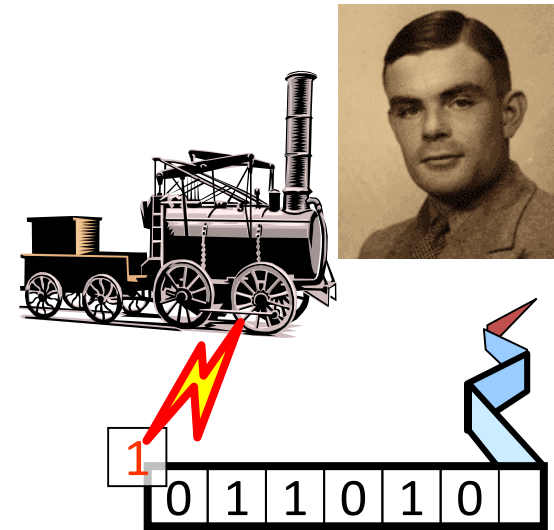


Fault-Tolerance

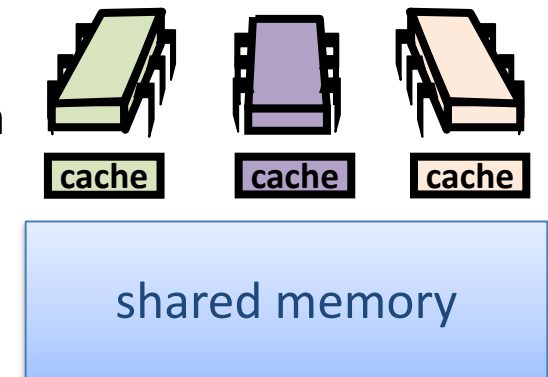


Computability

- Definition of computability
 - Computable usually means Turing-computable, i.e., the given problem can be solved using a Turing machine
 - Strong mathematical model!



- Shared-memory computability
 - Model of asynchronous concurrent computation
 - Computable means it is wait-free computable on a multiprocessor
 - Wait-free...?



Consensus #2: Wait-free Shared Memory



- $n > 1$ processors
- Processors can atomically *read* or *write* (not both) a shared memory cell
- Processors might crash (stop... or become very slow...)

Wait-free implementation:

- Every process completes in a finite number of steps
- Implies that locks cannot be used → The thread holding the lock may crash and no other thread can make progress
- We assume that we have wait-free atomic registers (i.e., reads and/or writes to same register do not overlap)

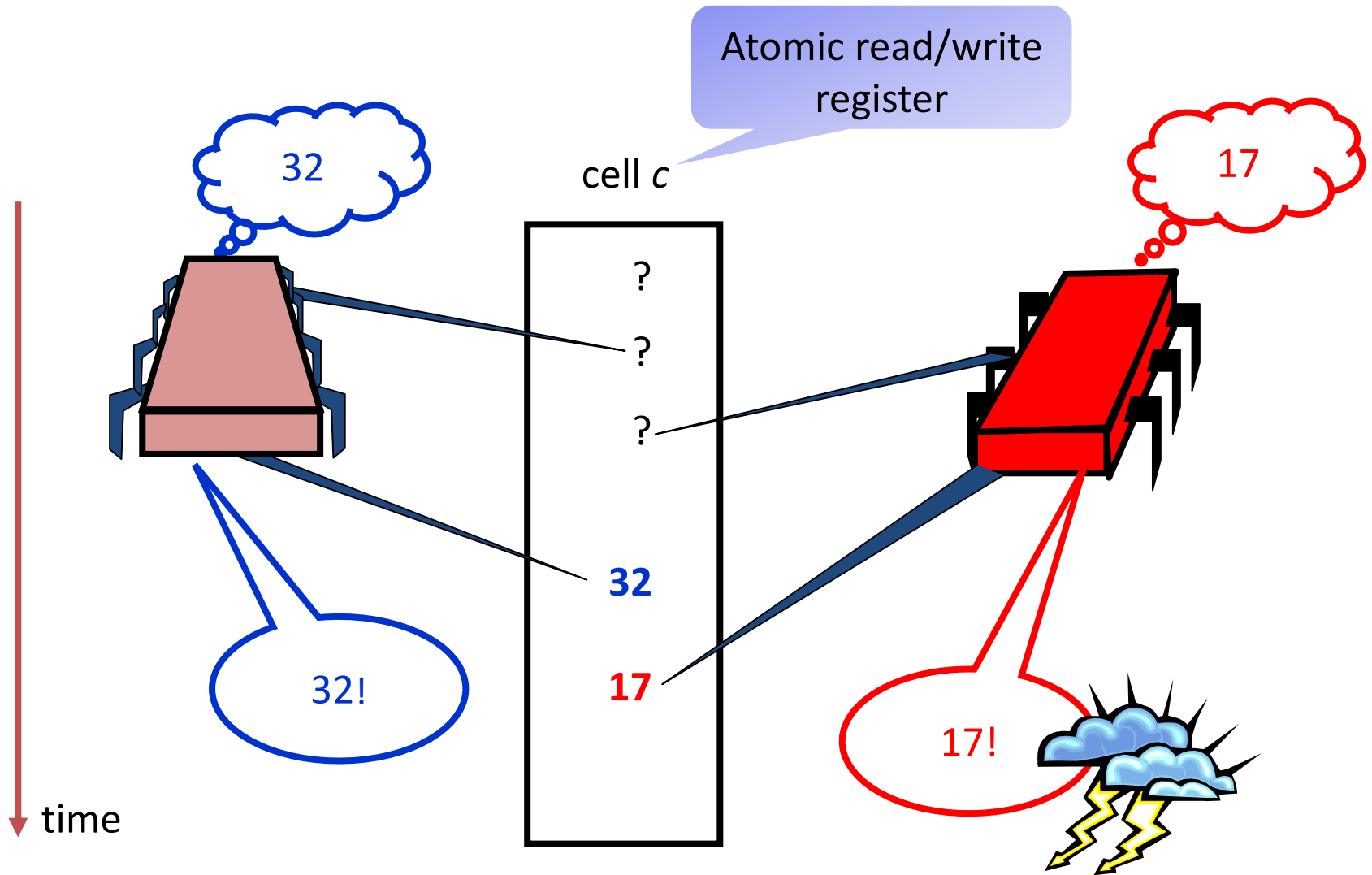
A Wait-Free Algorithm

- There is a cell c , initially $c = \text{"?"}$
- Every processor i does the following:

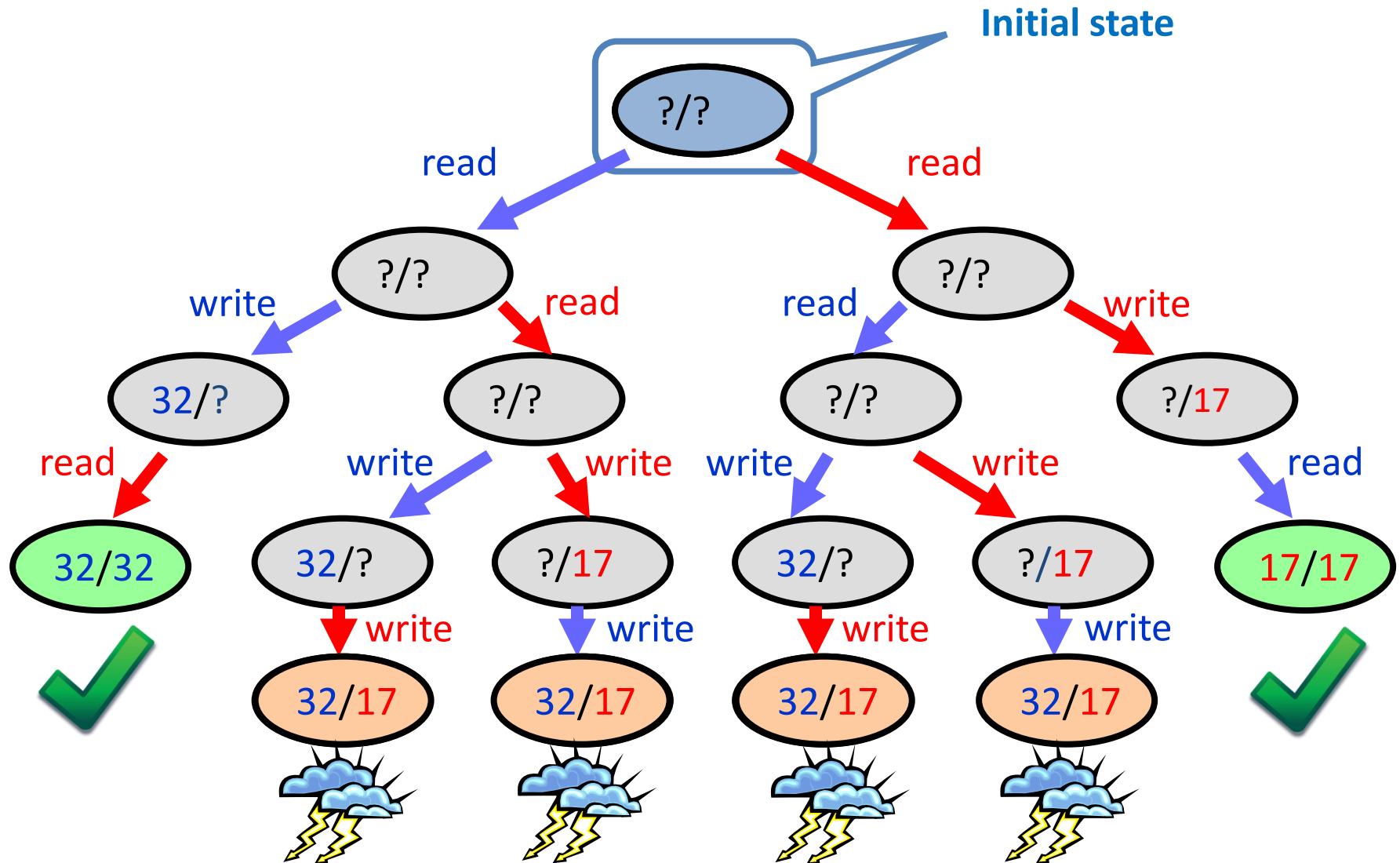
```
r = read(c);  
if (r == "?") then  
    write(c, vi); decide vi;  
else  
    decide r;
```

- Is this algorithm correct...?

An Execution



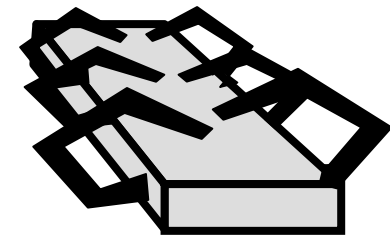
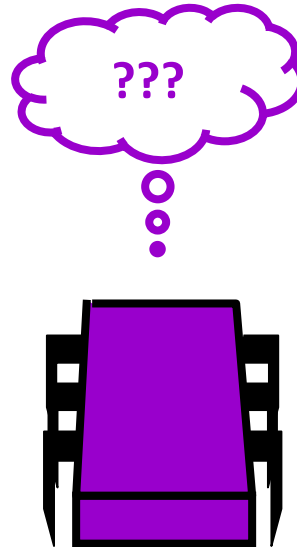
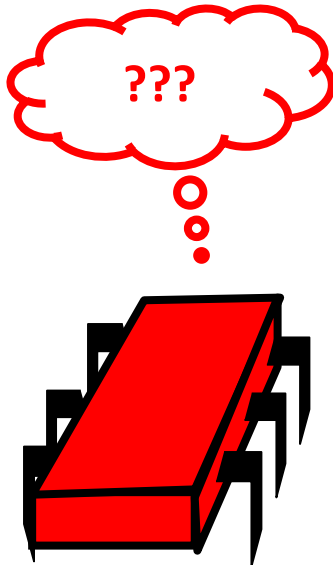
Execution Tree



Impossibility

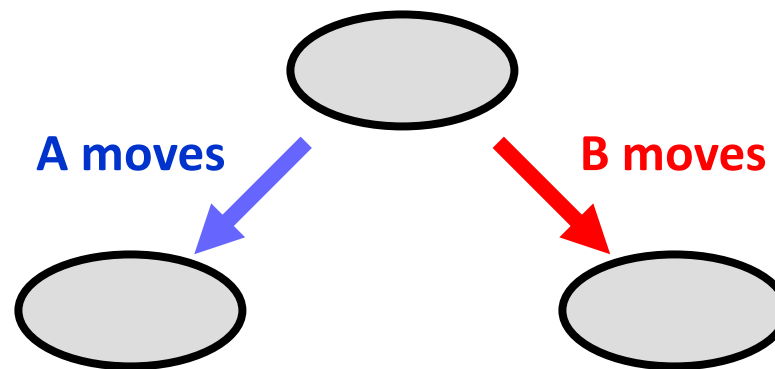
Theorem

There is no wait-free consensus algorithm using read/write atomic registers.

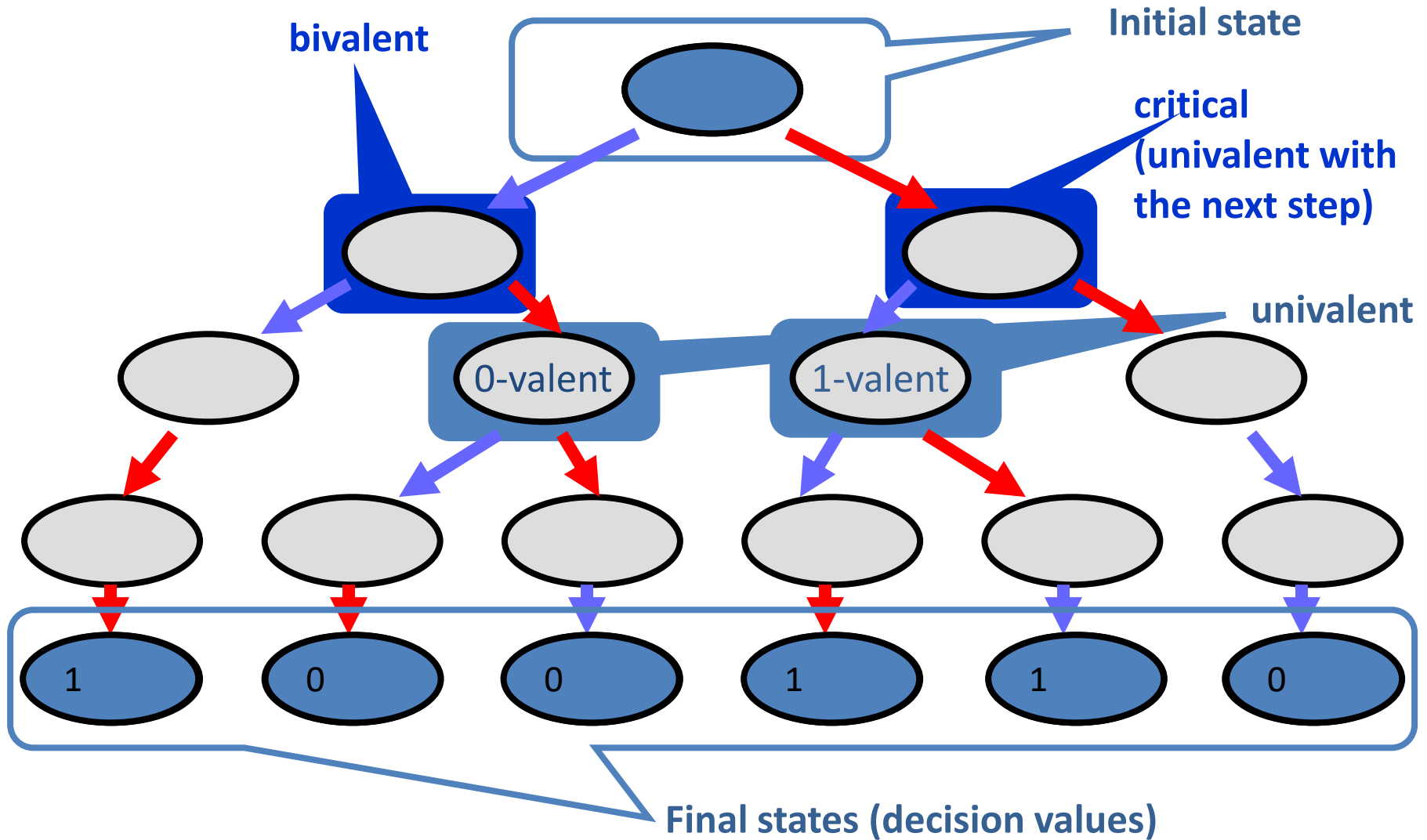


Proof

- Make it simple
 - There are only **two processes A** and **B** and the **input is binary**
- Assume that there is a protocol
- In this protocol, either **A** or **B** “moves” in each step
- Moving means
 - Register read
 - Register write



Execution Tree



Bivalent vs. Univalent

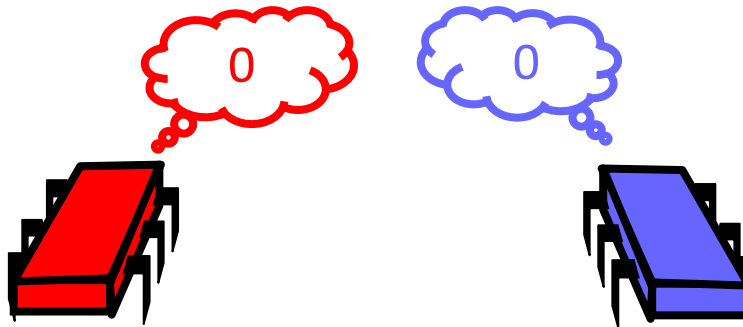
- Wait-free computation is a tree
- Bivalent system states
 - Outcome is not fixed
- Univalent states
 - Outcome is fixed
 - Maybe not “known” yet
 - 1-valent and 0-valent states

Claim:

- **Some initial system state is bivalent**
- Hence, the outcome is not always fixed from the start

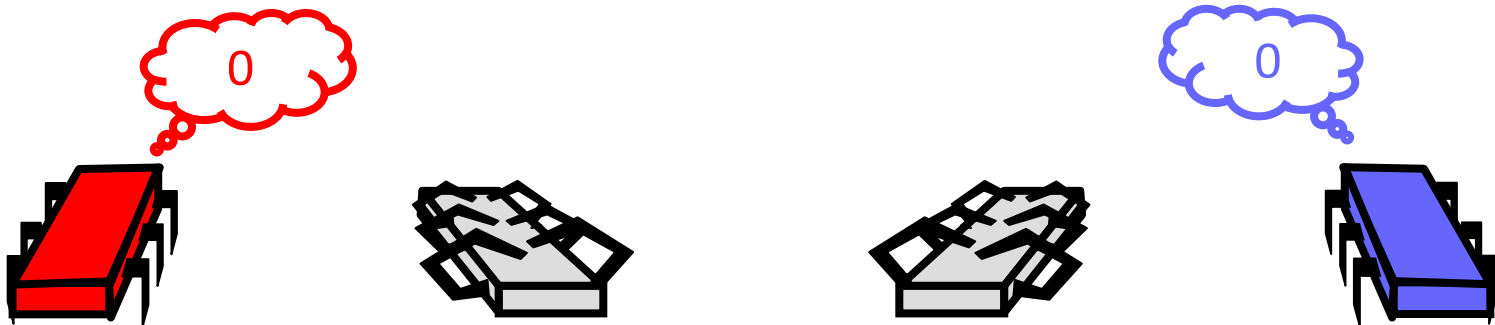
Proof of Claim: A 0-Valent Initial State

- All executions lead to the decision 0



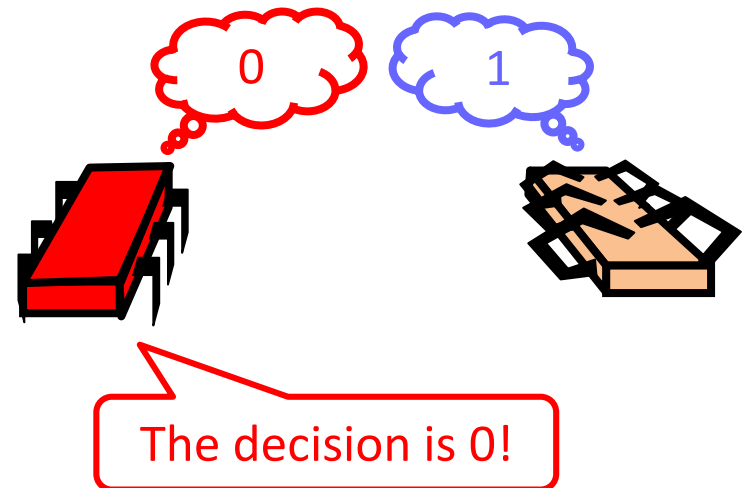
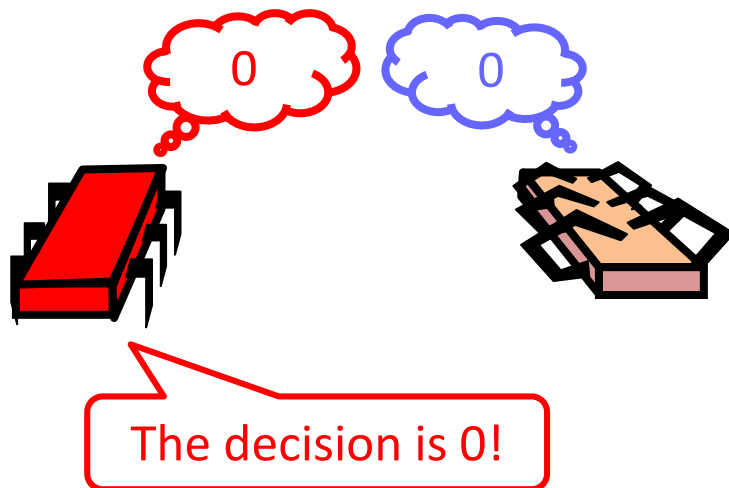
Similarly, the decision is always 1 if both threads start with 1!

- Solo executions also lead to the decision 0



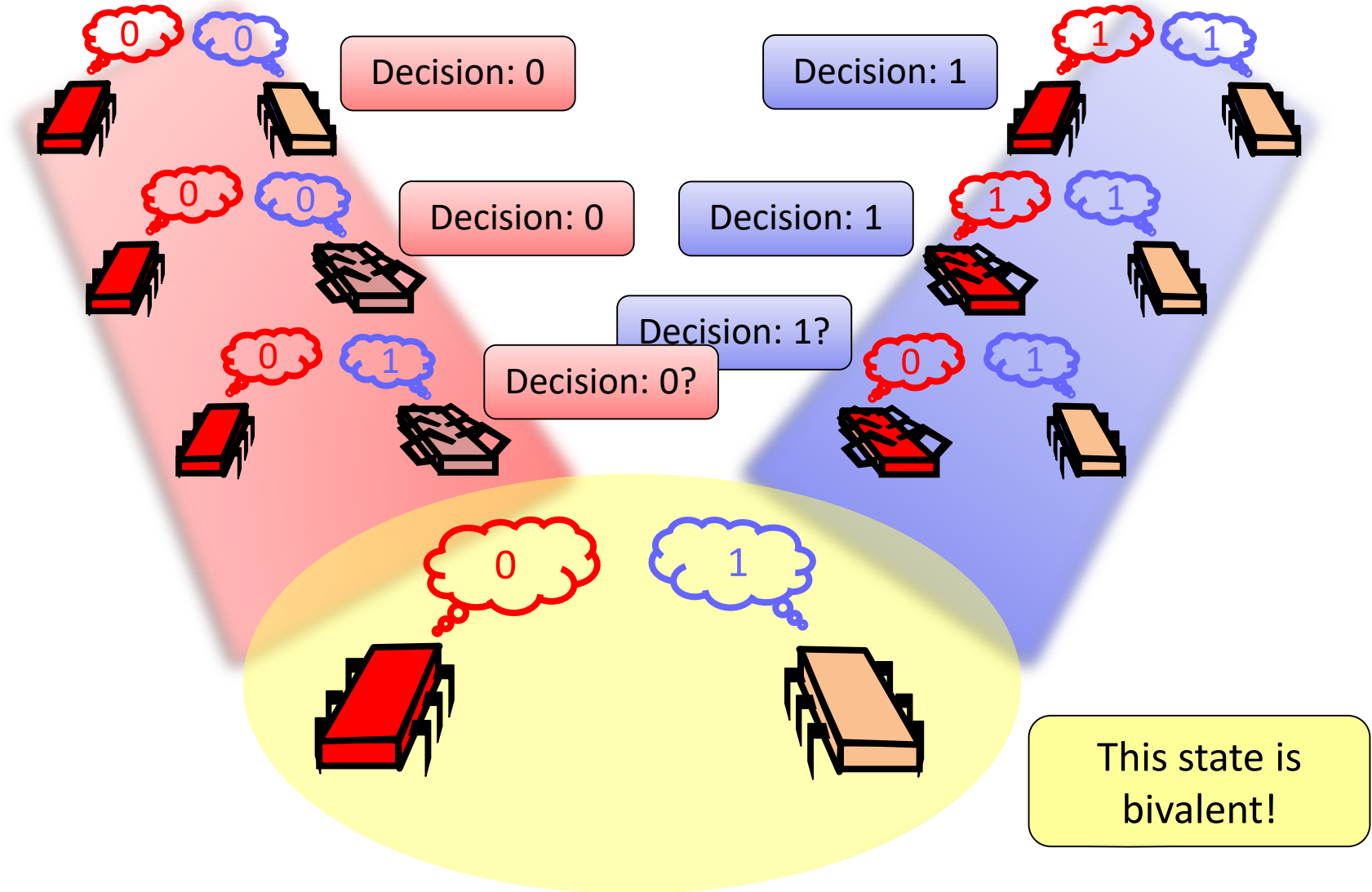
Proof of Claim: Indistinguishable Situations

- Situations are indistinguishable to red process
⇒ The outcome must be the same



Similarly, the decision is 1 if the red thread crashed!

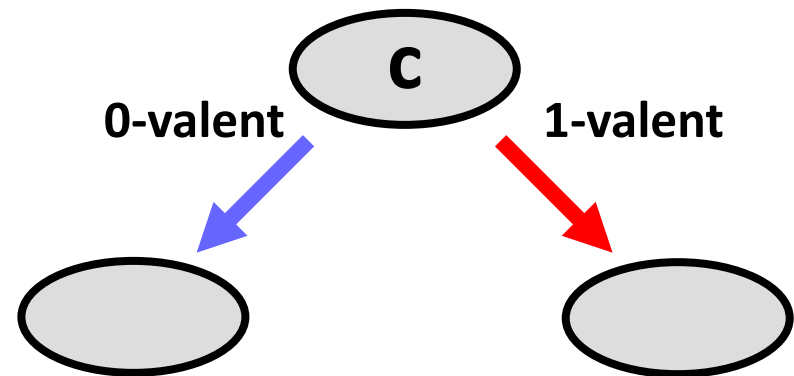
Proof of Claim: A Bivalent Initial State



Critical States

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

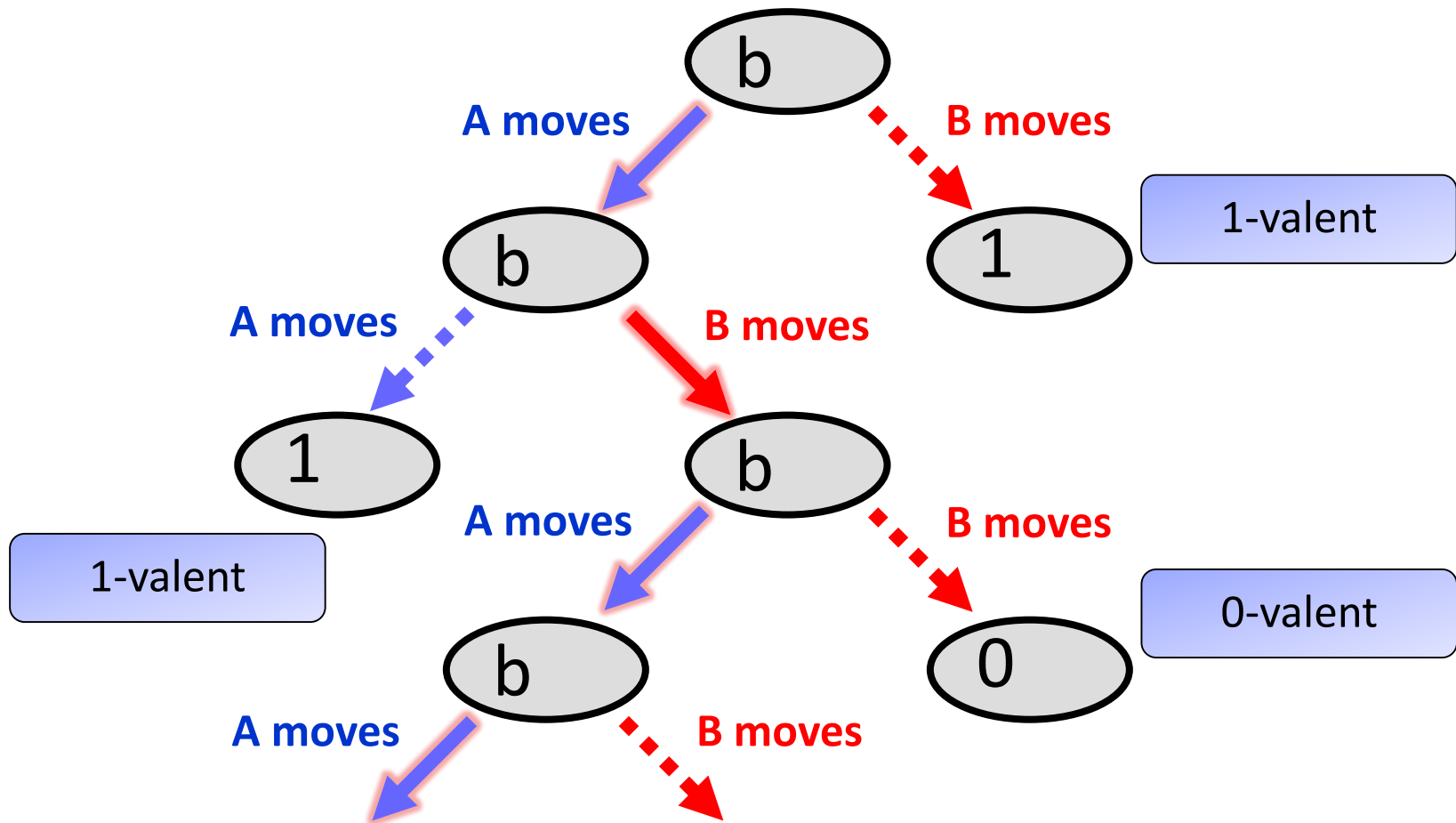
A bivalent state is critical if all children states are univalent



- The goal is now to show that the system can always remain bivalent

Reaching a Critical State

- The system can remain bivalent forever if there is always an action that prevents the system from reaching a critical state:



Model Dependency

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

Steps with Shared Read/Write Registers

- Processes/Threads
 - Perform reads and/or writes
 - To the same or different registers
 - Possible interactions?

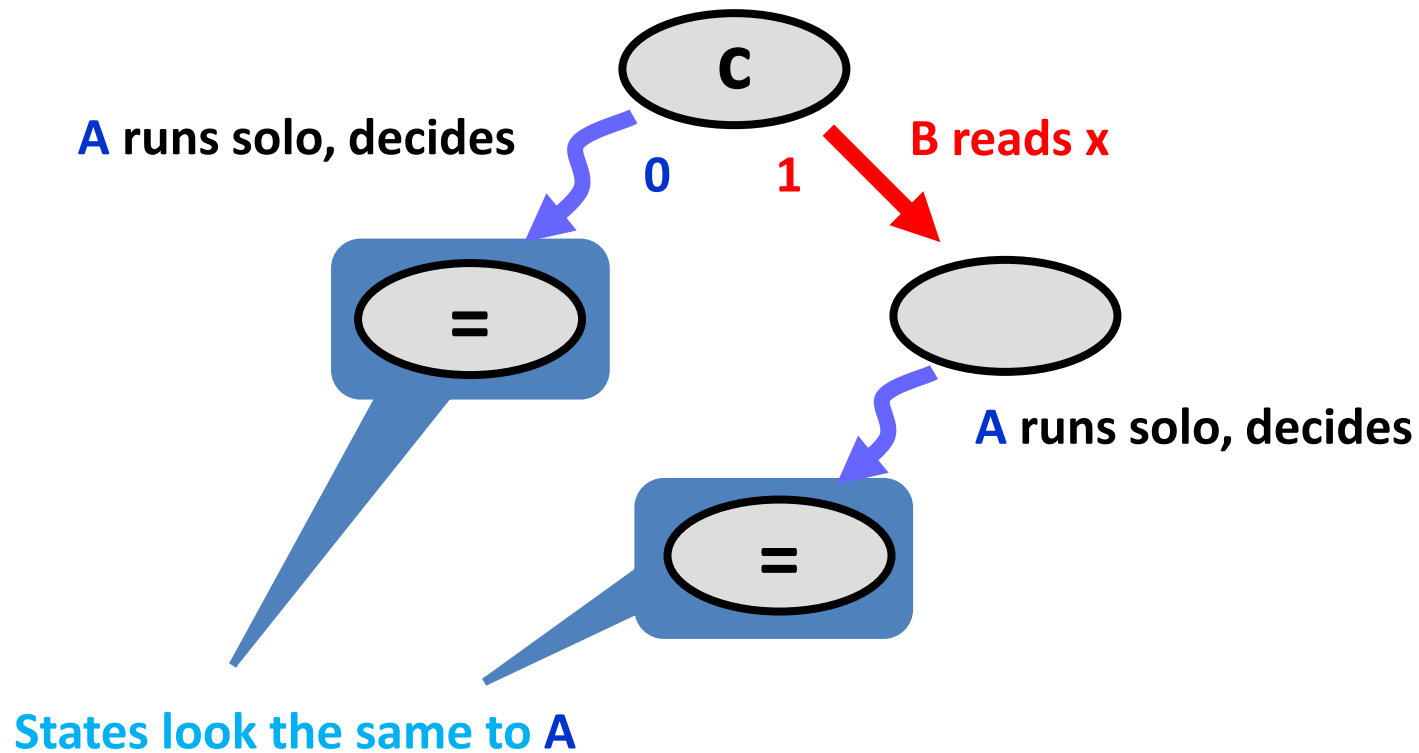
Possible Interactions

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

A reads x

B writes y

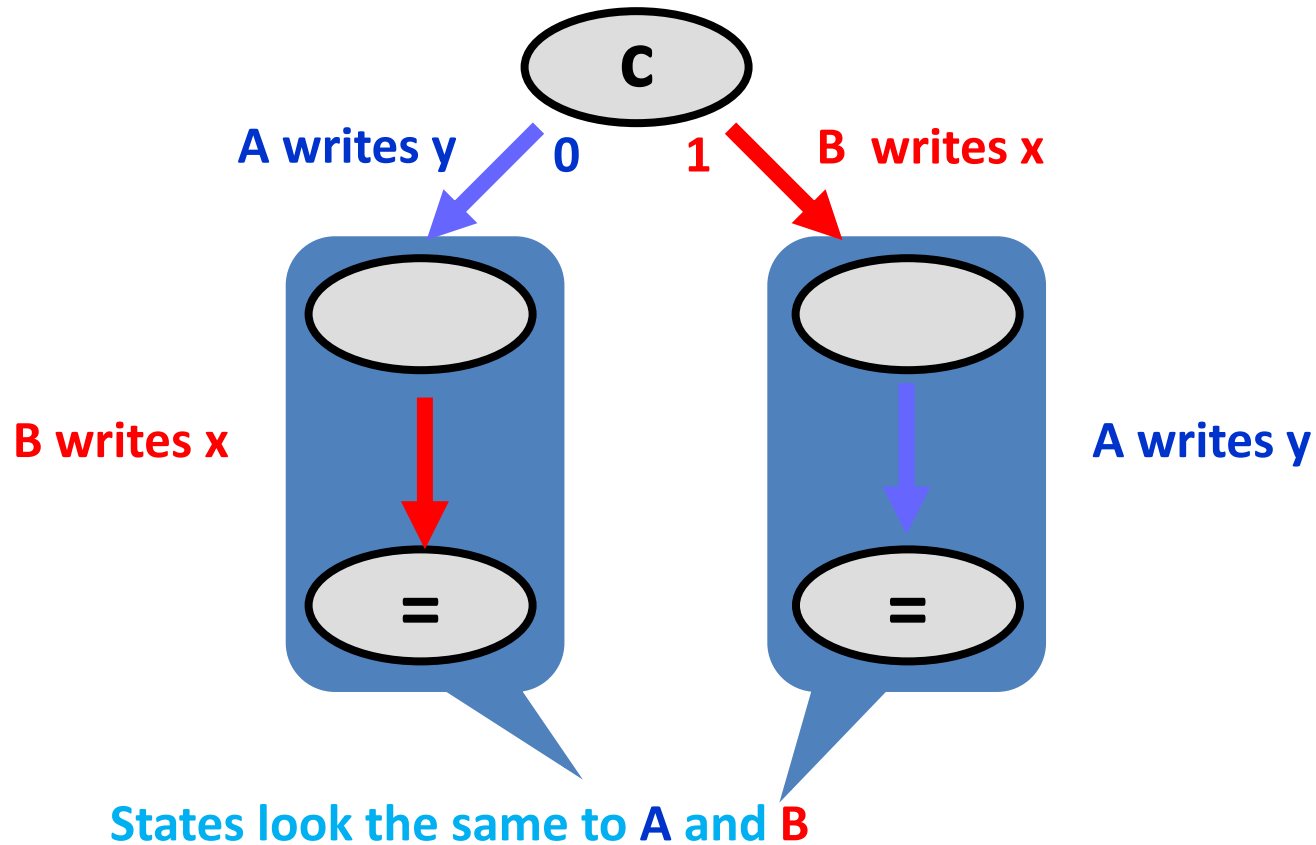
Reading Registers



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	?	?

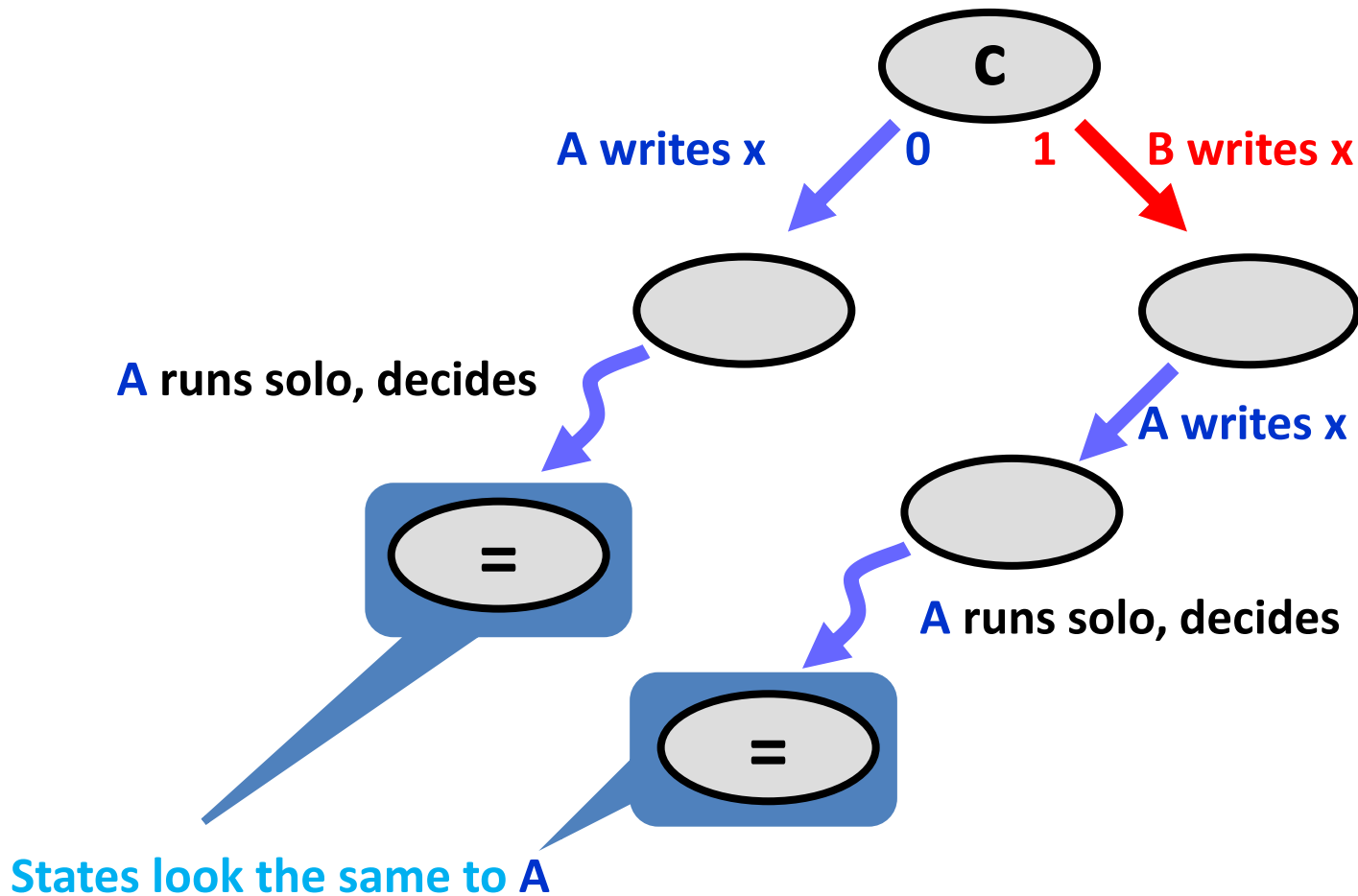
Writing Distinct Registers



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

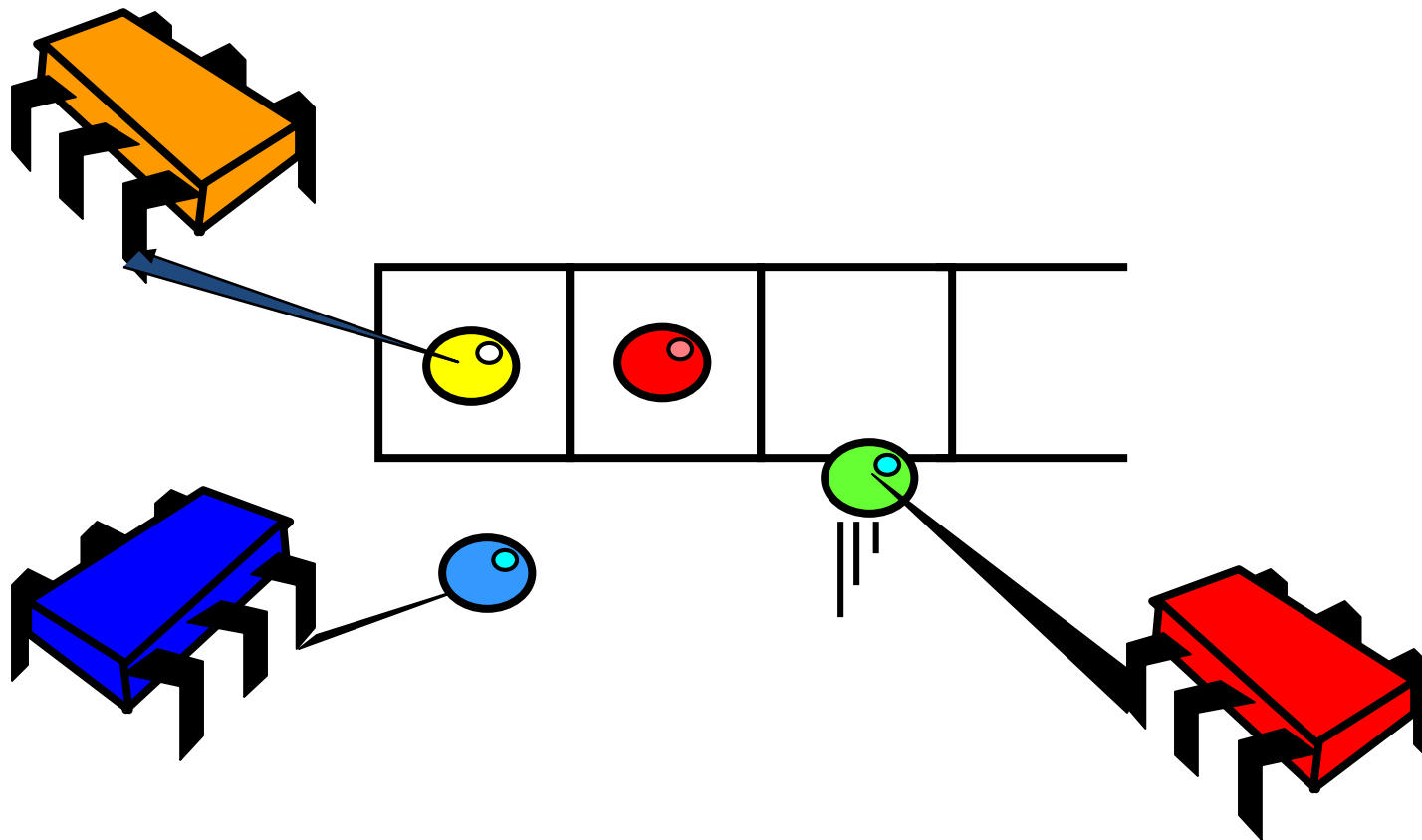


This Concludes the Proof 😊

	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

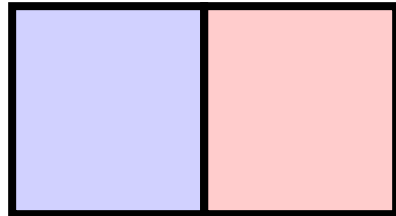
Consensus in Distributed Systems?

- We want to build a concurrent FIFO Queue with multiple dequeuers

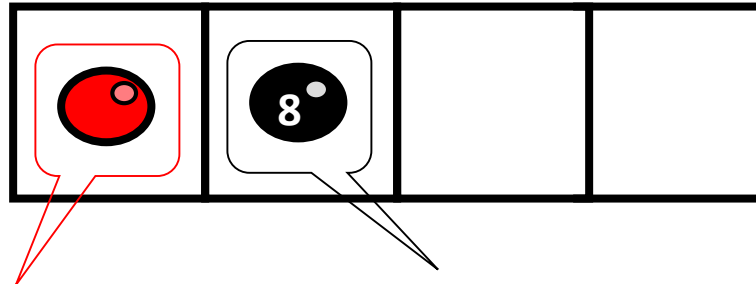


A Consensus Protocol

- Assume we have such a FIFO queue and a 2-element array



2-element array



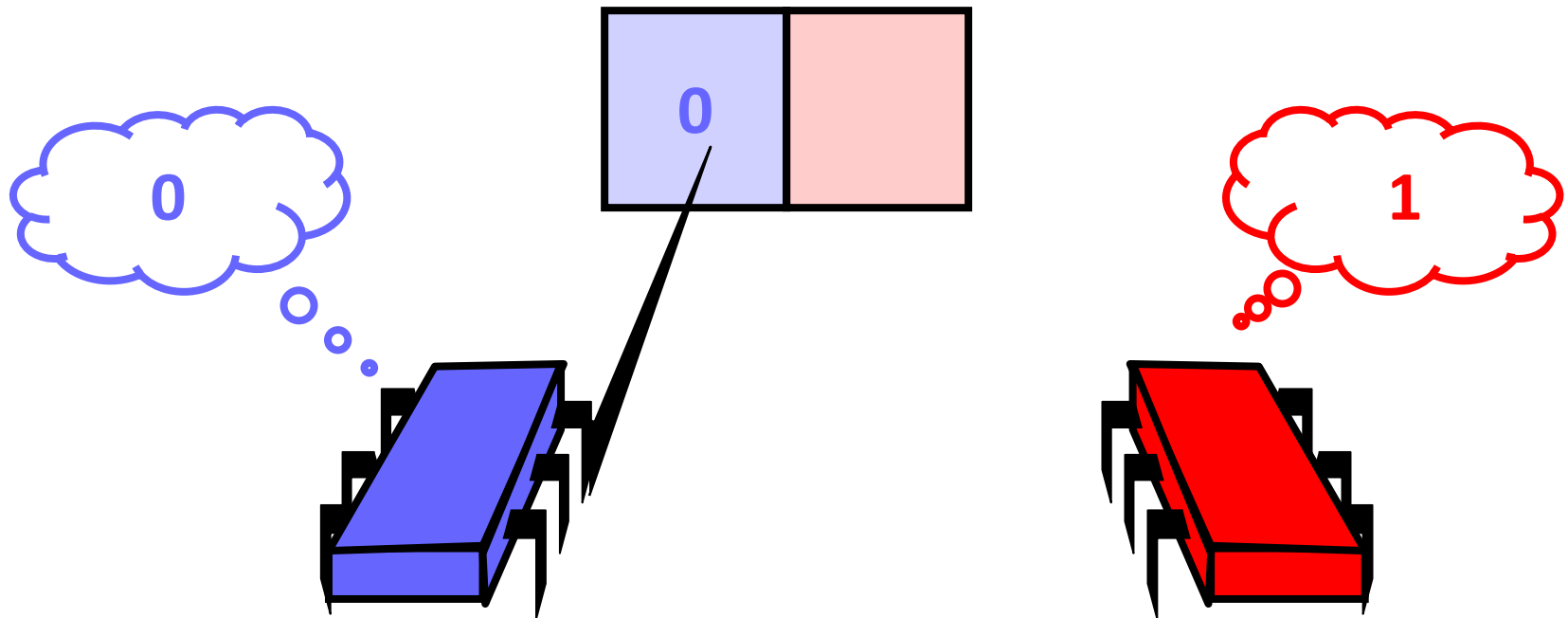
Coveted red ball

Dreaded black ball

FIFO Queue with red and black balls

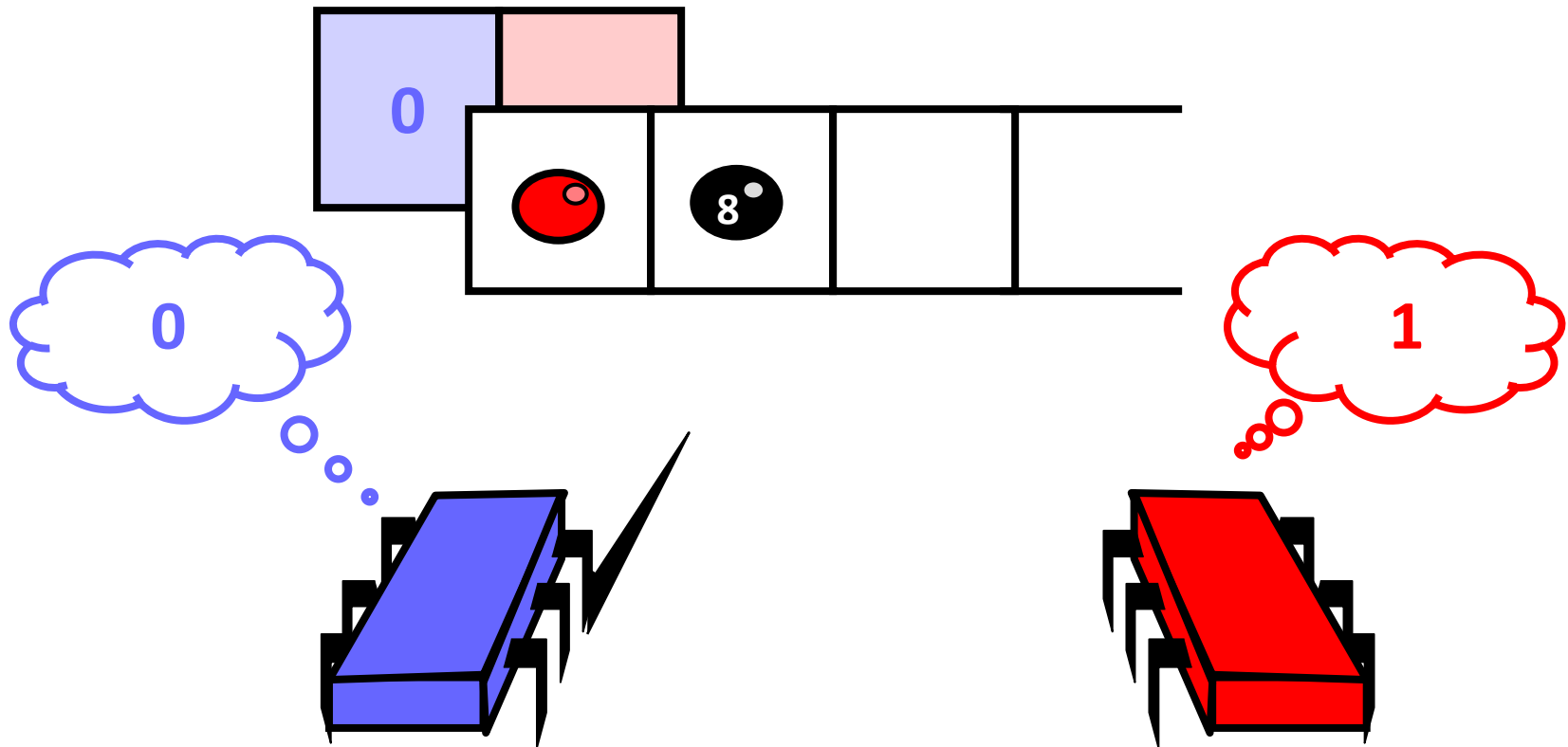
A Consensus Protocol

- Process i writes its value into the array at position i

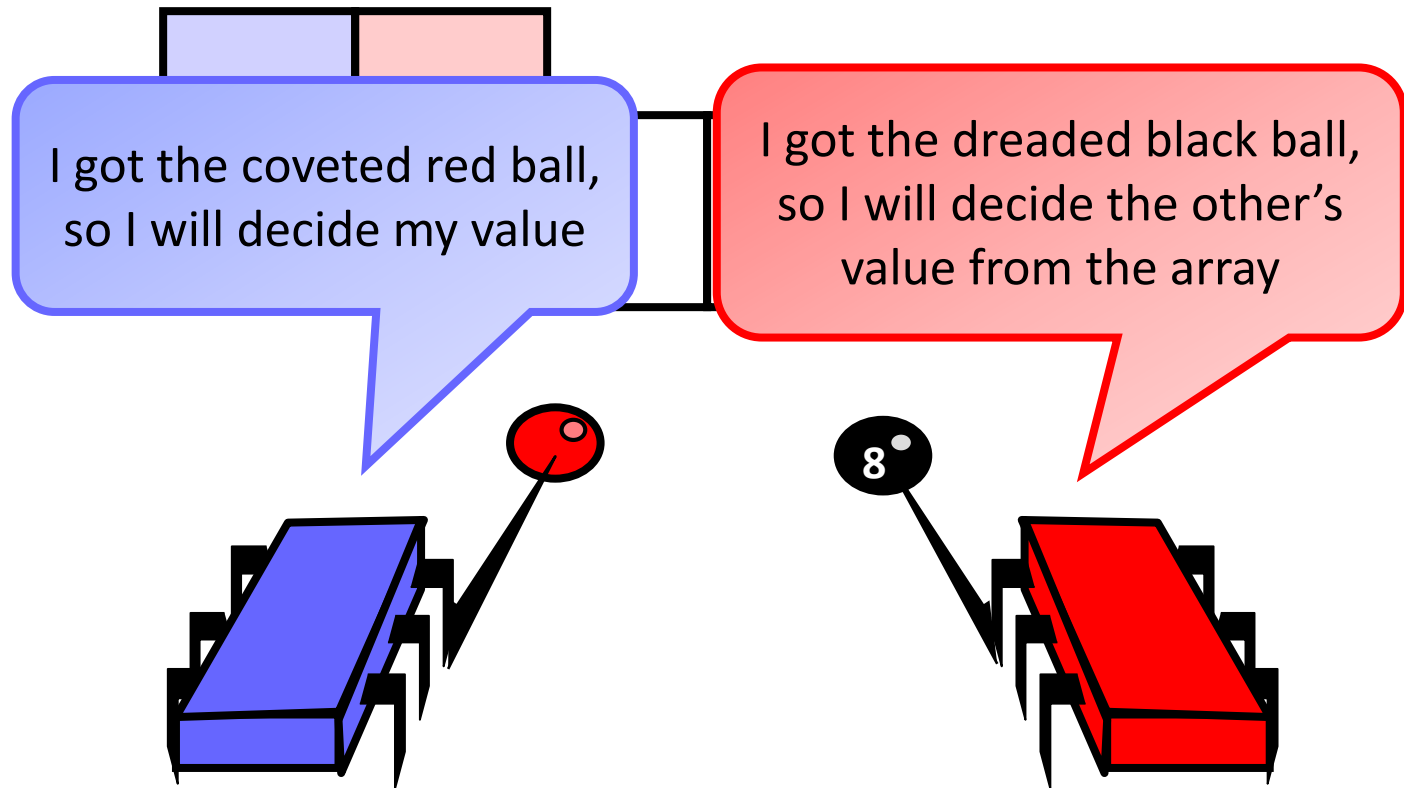


A Consensus Protocol

- Then, the thread takes the next element from the queue



A Consensus Protocol



Why does this work?

- If one thread gets the red ball, then the other gets the black ball
- Winner can take its own value
- Loser can find winner's value in array
 - Because processes 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 Memory



- $n > 1$ processes (processors/nodes/threads)
- Wait-free implementation
- Processors can **read and write** a shared memory cell **in one atomic step**: the value written can depend on the value read
- We call this a read-modify-write (RMW) register
- Can we solve consensus using a RMW register...?

Consensus Protocol Using a RMW Register

- There is a cell c , initially $c = \text{"?"}$
- Every processor i does the following

RMW(c)

```
if (c == "?") then
  write(c, vi); decide vi
else
  decide c;
```

atomic step

- Protocol works correctly
 - One processor accesses c 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:
 - Cell c
 - Function f
- Method call:
 - Replaces value x of cell c with $f(x)$
 - Returns value x of cell c

Read-Modify-Write

```
public class RMW {  
    private int value;  
  
    public synchronized int rmw(function f) {  
        int prior = this.value;  
        this.value = f(this.value);  
        return prior;  
    }  
}
```

Return prior value

Apply function

Read-Modify-Write: Read

```
public class RMW {  
    private int value;  
  
    public synchronized int read() {  
        int prior = this.value;  
        this.value = this.value;  
        return prior;  
    }  
}
```

Identify function

Read-Modify-Write: Test&Set

```
public class RMW {  
    private int value;  
  
    public synchronized int TAS() {  
        int prior = this.value;  
        this.value = 1;  
        return prior;  
    }  
}
```

Constant function

Read-Modify-Write: Fetch&Inc

```
public class RMW {  
    private int value;  
  
    public synchronized int FAI() {  
        int prior = this.value;  
        this.value = this.value+1;  
        return prior;  
    }  
}
```

Increment function

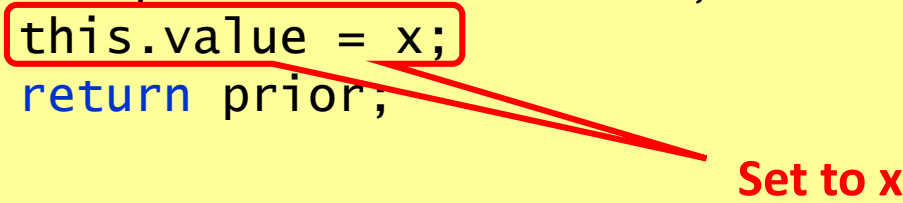
Read-Modify-Write: Fetch&Add

```
public class RMW {  
    private int value;  
  
    public synchronized int FAA(int x) {  
        int prior = this.value;  
        this.value = this.value+x;  
        return prior;  
    }  
}
```

Addition function

Read-Modify-Write: Swap

```
public class RMW {  
    private int value;  
  
    public synchronized int swap(int x) {  
        int prior = this.value;  
        this.value = x;  
        return prior;  
    }  
}
```



Set to x

Read-Modify-Write: Compare&Swap

```
public class RMW {  
    private int value;  
  
    public synchronized int CAS(int old, int new) {  
        int prior = this.value;  
        if(this.value == old)  
            this.value = new;  
        return prior;  
    }  
}
```

“Complex” function

Definition of Consensus Number

- An object has **consensus number n**
 - If it can be used
 - Together with atomic read/write registers
 - To implement n -process consensus, but not $(n + 1)$ -process consensus
- Example: Atomic read/write registers have consensus number 1
 - Works with 1 process
 - We have shown impossibility with 2

Theorem

If you can implement X from Y and X has consensus number c , then Y has consensus number **at least c** .

- Consensus numbers are a useful way of measuring synchronization power
- An alternative formulation:
 - 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

- A RMW is *non-trivial* if there exists a value v such that $v \neq f(v)$
 - Test&Set, Fetch&Inc, Fetch&Add, Swap, Compare&Swap, general RMW...
 - But not read

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

- A two-process consensus protocol using any non-trivial RMW object:

```

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

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: $f_i(f_j(x))=f_i(x)$

$f_i(x)$ = new value of cell
(not return value of f_i)

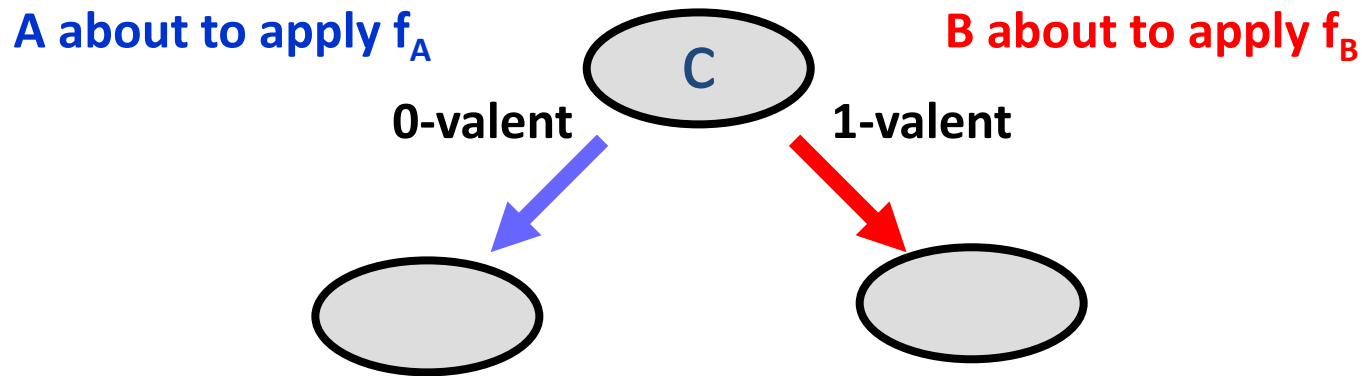
Claim: Any such set of RMW objects has consensus number **exactly 2**

Examples:

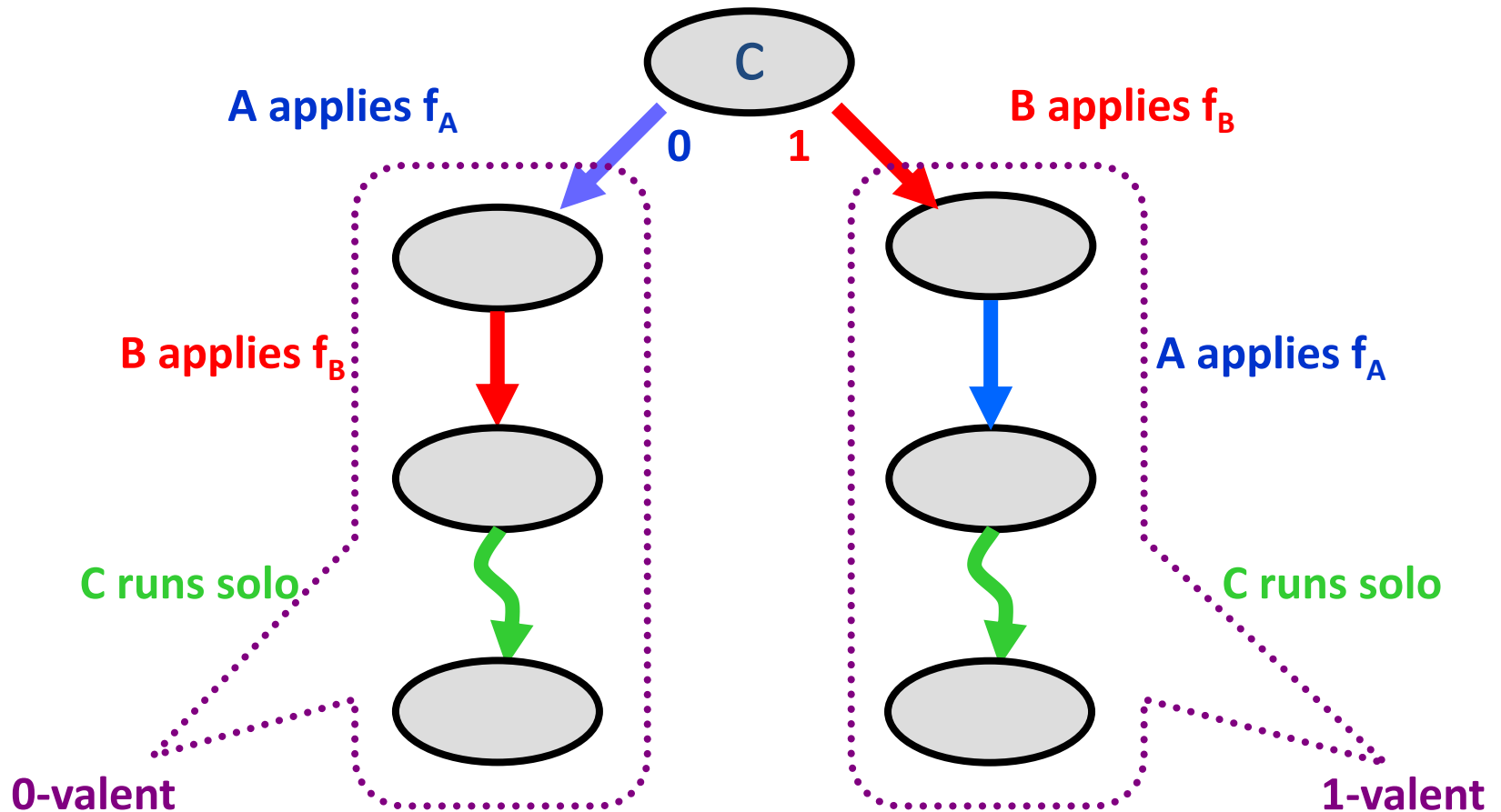
- Overwrite
 - Test&Set , Swap
- Commute
 - Fetch&Inc, Fetch&Add

Proof

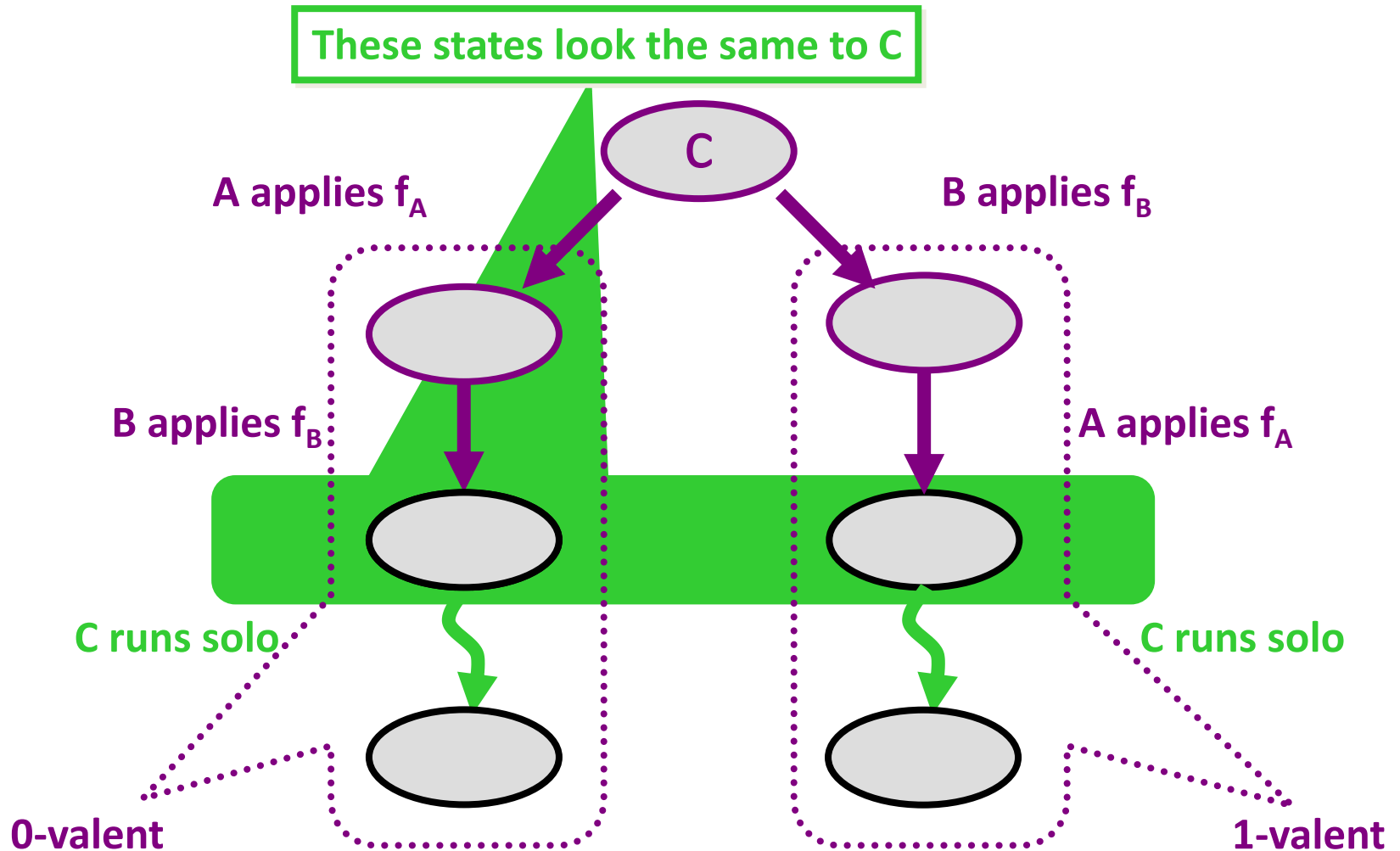
- There are three threads, **A**, **B**, and **C**
- Consider a critical state **c**:



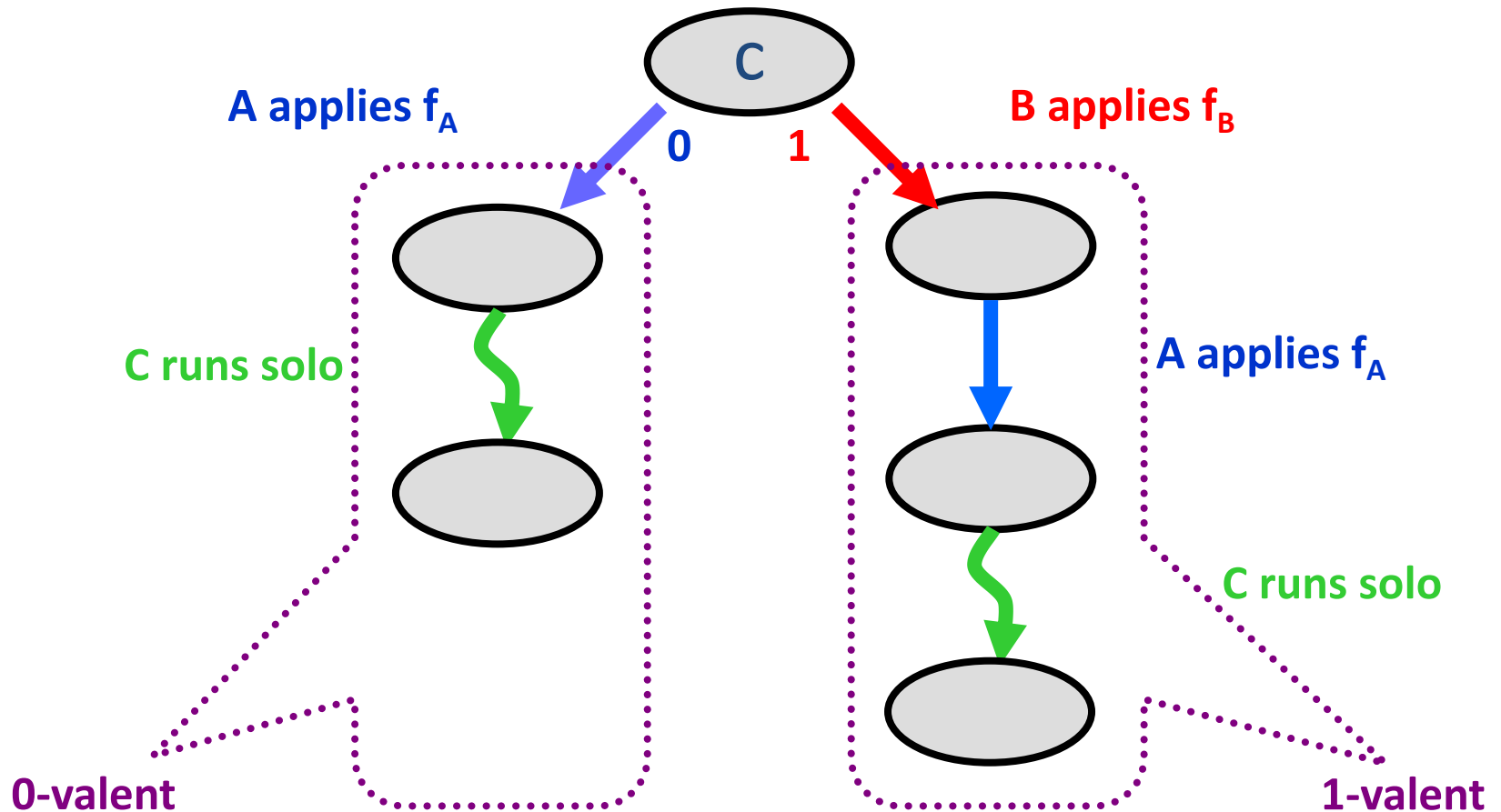
Proof: Maybe the Functions Commute



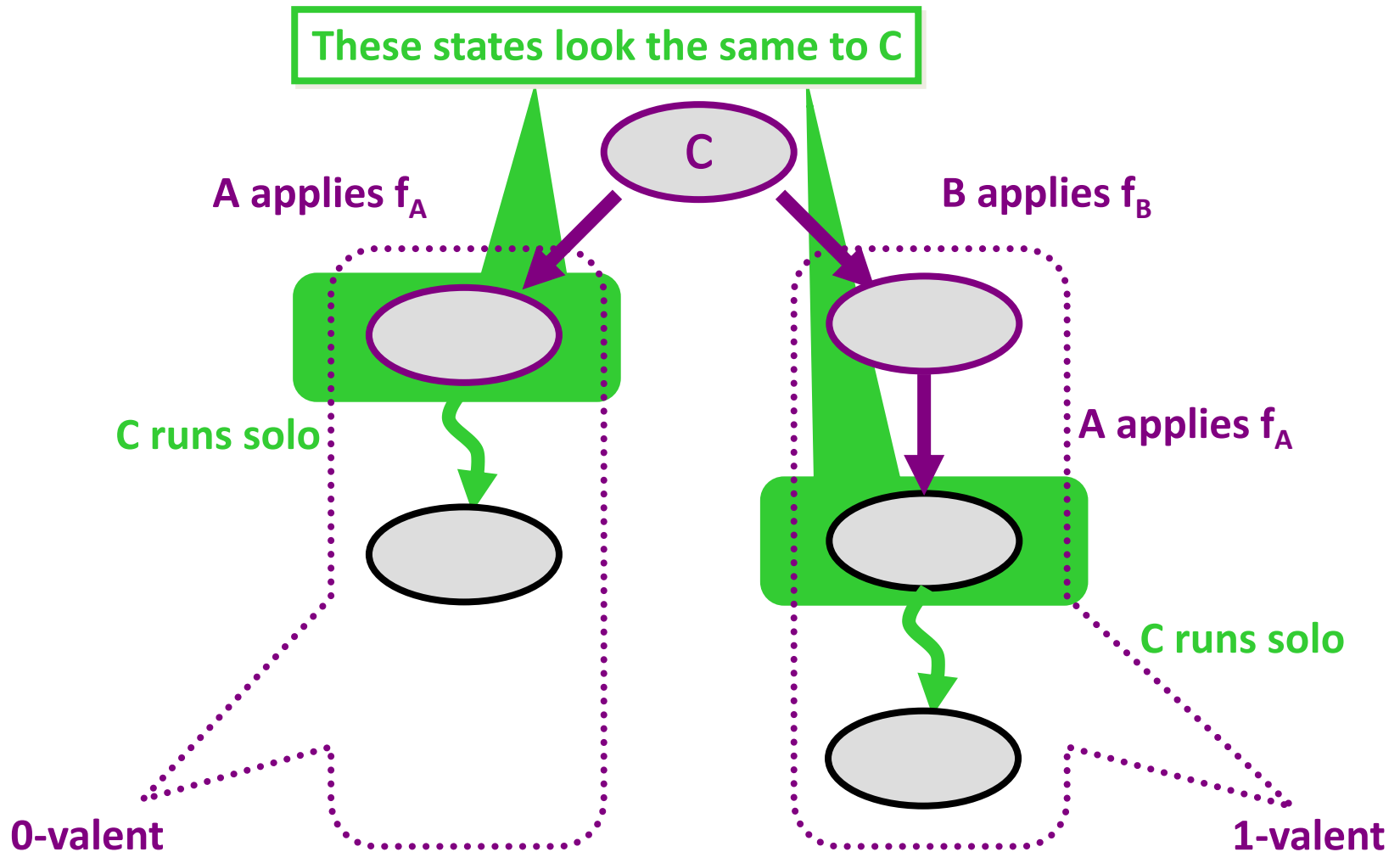
Proof: Maybe the Functions Commute



Proof: Maybe the Functions Overwrite



Proof: Maybe the Functions Overwrite



- Many early machines used these “weak” RMW instructions
 - Test&Set (IBM 360)
 - Fetch&Add (NYU Ultracomputer)
 - Swap
- We now understand their limitations

Consensus with Compare & Swap

```
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];  
    }  
}
```

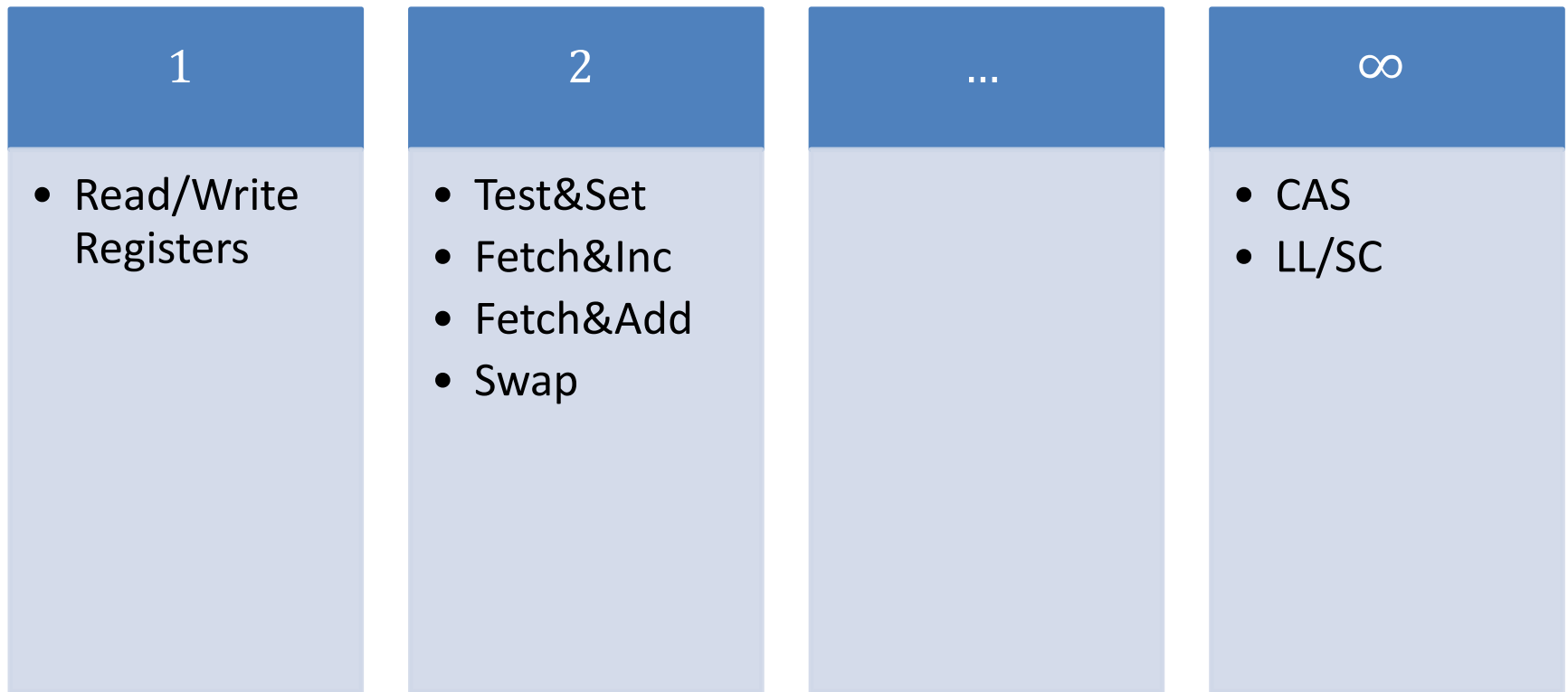
Initialized to -1

Am I first?

Yes, return my input

No, return other's input

The Consensus Hierarchy





Chapter 6

Consensus

Distributed Systems

SS 2015

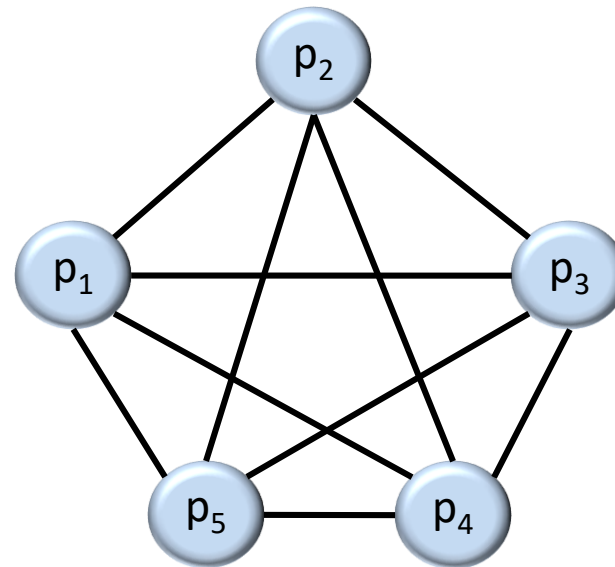
Fabian Kuhn

- Introduction
- Consensus #1: Shared Memory
- Consensus #2: Wait-free Shared Memory
- Consensus #3: Read-Modify-Write Shared Memory
- Consensus #4: Synchronous Systems
- Consensus #5: Byzantine Failures
- Consensus #6: A Simple Algorithm for Byzantine Agreement
- Consensus #7: The Queen Algorithm
- Consensus #8: The King Algorithm
- Consensus #9: Byzantine Agreement Using Authentication
- Consensus #10: A Randomized Algorithm
- Shared Coin

- Slides by R. Wattenhofer (ETHZ)

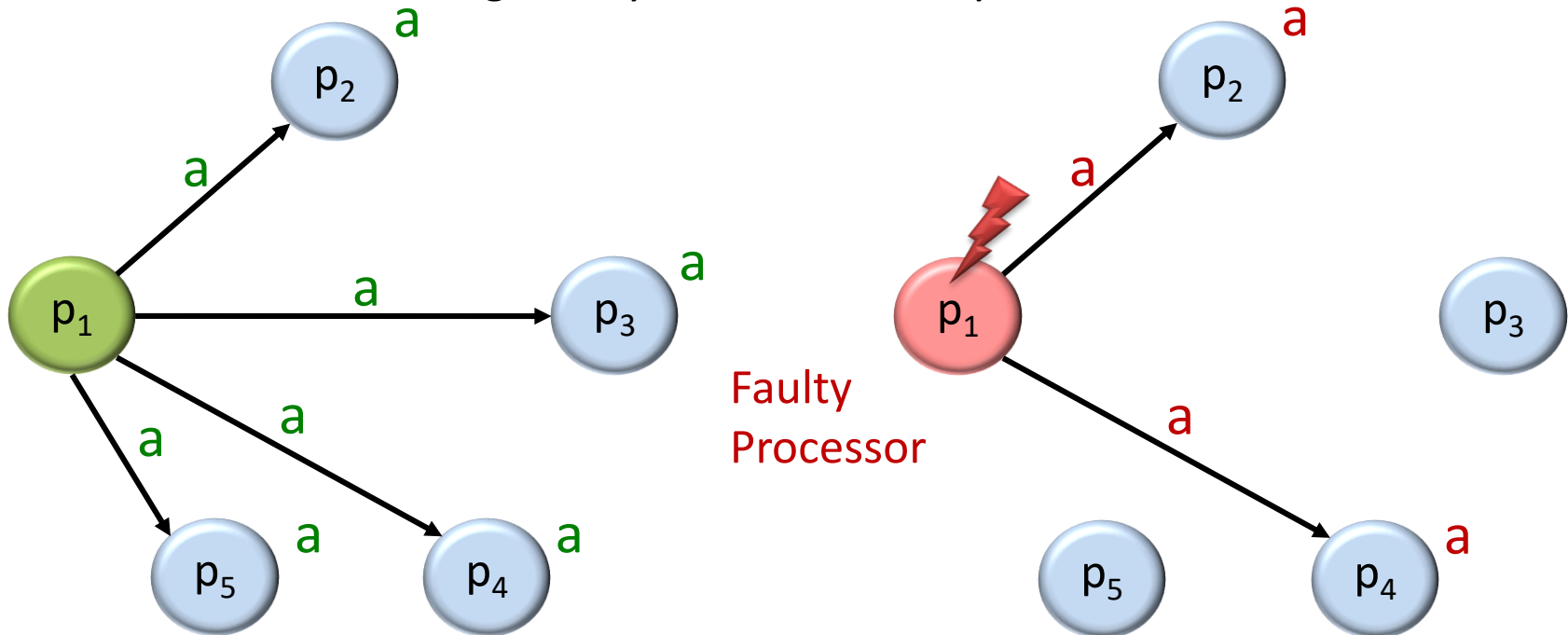
Consensus #4: Synchronous Systems

- One can sometimes tell if a processor had crashed
 - Timeouts
 - Broken TCP connections
- Can one solve consensus at least in synchronous systems?
- Model
 - All communication occurs in synchronous rounds
 - Complete communication graph

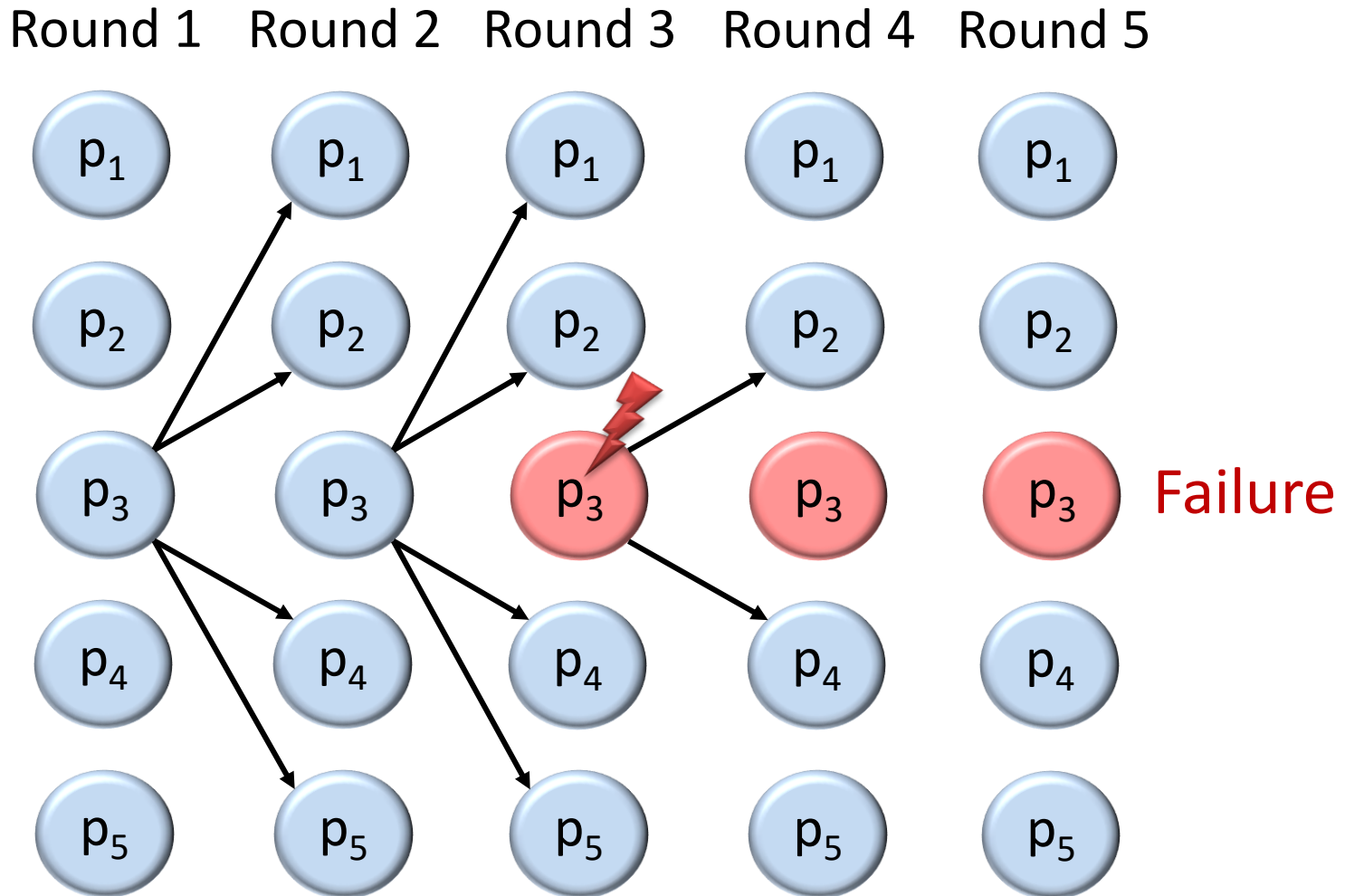


Crash Failures

- Broadcast: Send a message to all nodes in one round
 - At the end of the round everybody receives the message **a**
 - Every process can broadcast a value in each round
- Crash Failures: A broadcast can fail if a process crashes
 - Some of the messages may be lost, i.e., they are never received



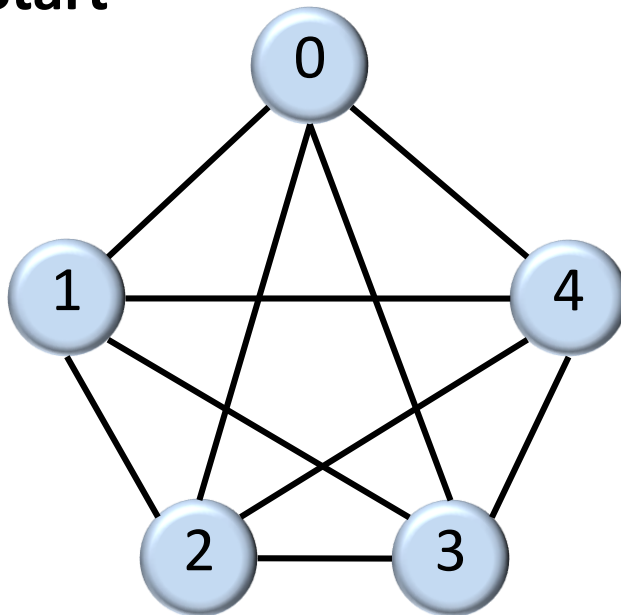
Process disappears after failure



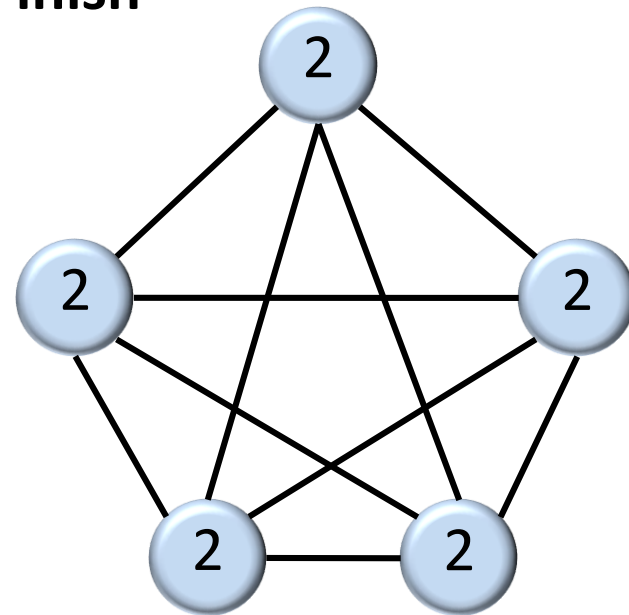
Consensus Repetition

- **Input:** everybody has an initial value
- **Agreement:** everybody must decide on the same value

Start



Finish

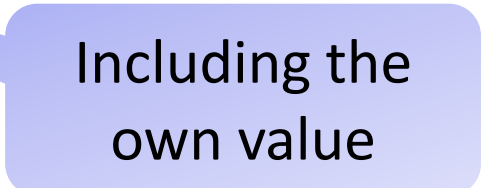


- **Validity condition:** If everybody starts with the same value, everybody must decide on that value

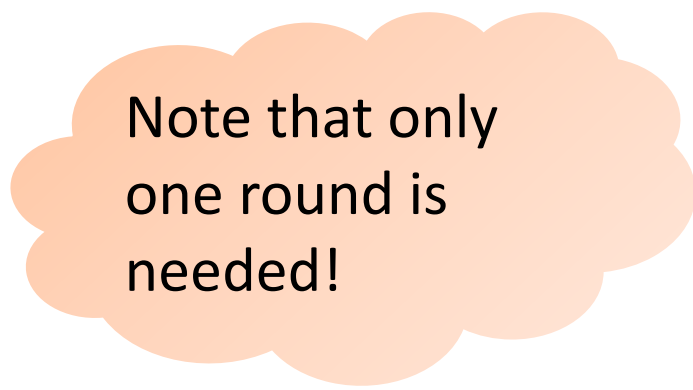
A Simple Consensus Algorithm

Each process:

1. Broadcast own value
2. Decide on the minimum of all received values



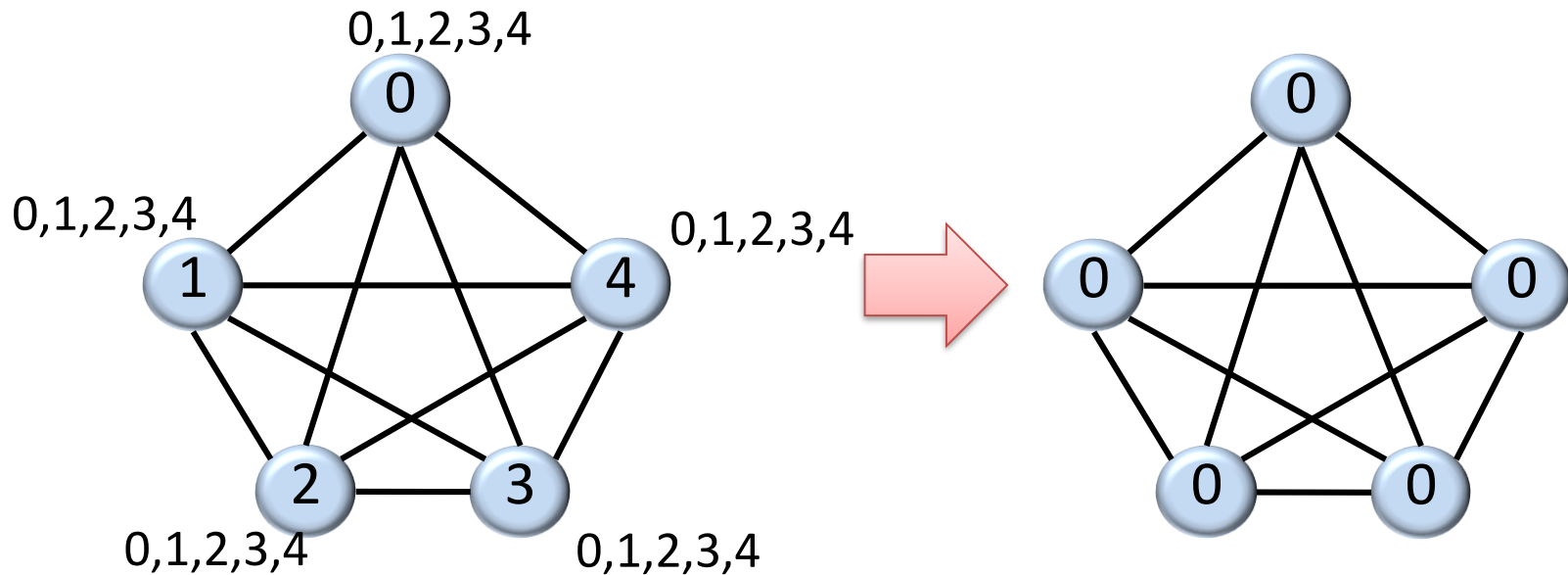
Including the
own value



Note that only
one round is
needed!

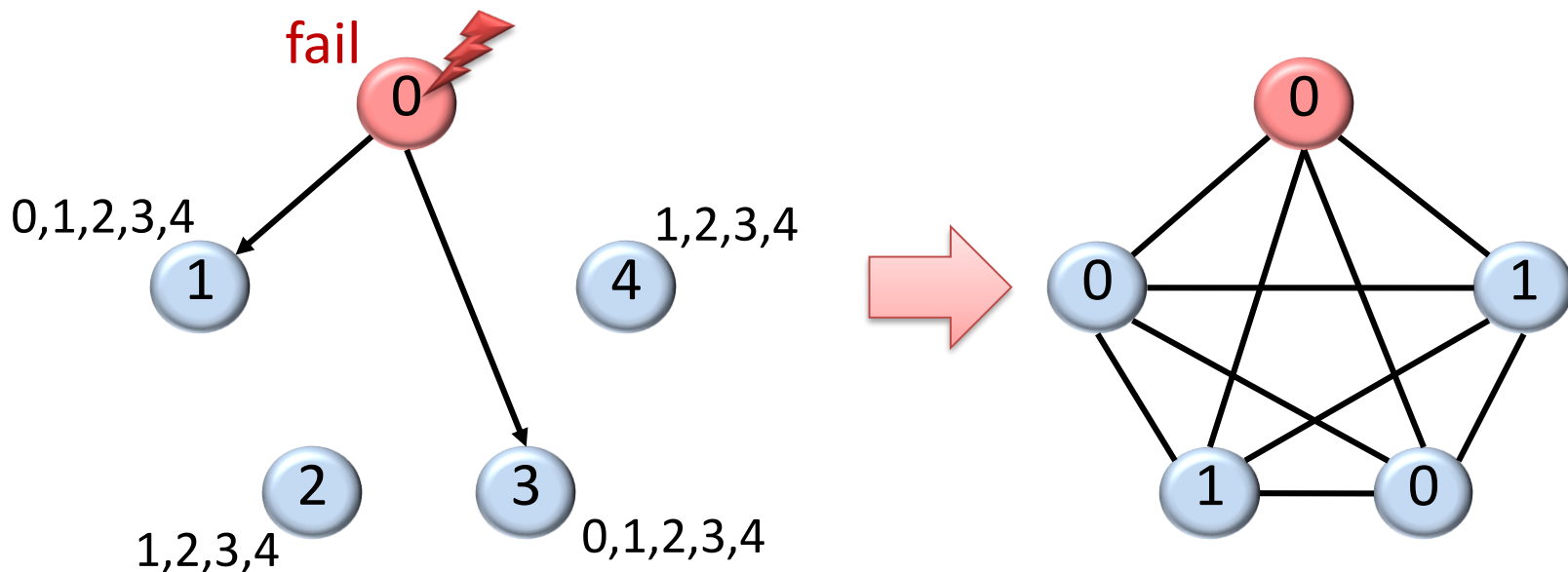
Execution Without Failures

- Broadcast values and decide on minimum → Consensus!
- Validity condition is satisfied: If everybody starts with the same initial value, everybody sticks to that value (minimum)



Execution With Failures

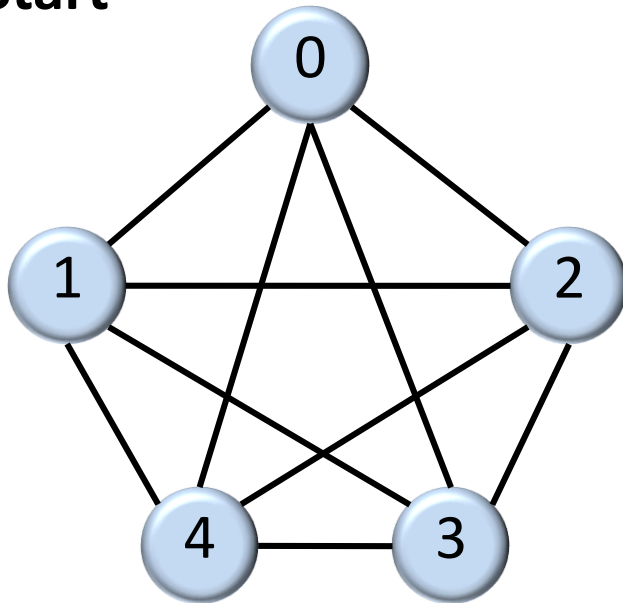
- The failed processor doesn't broadcast its value to all processors
- Decide on minimum \rightarrow No consensus!



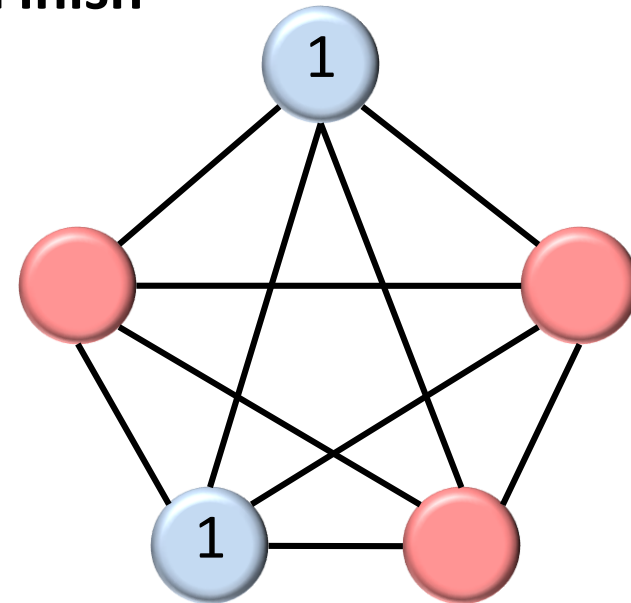
f -Resilient Consensus Algorithm

- 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 alg.

Start



Finish



- **Refined validity condition:**
All processes decide on a value that is available initially

An f -Resilient Consensus Algorithm



Each process:

Round 1:

Broadcast own value

Round 2 to round $f + 1$:

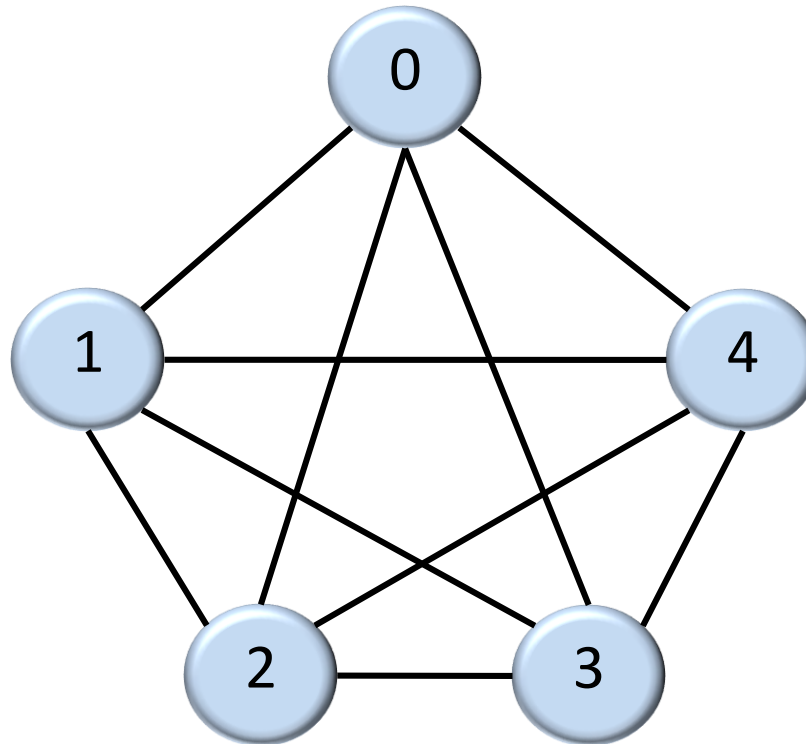
Broadcast the minimum of the received values
unless it has been sent before

End of round $f + 1$:

Decide on the minimum value received

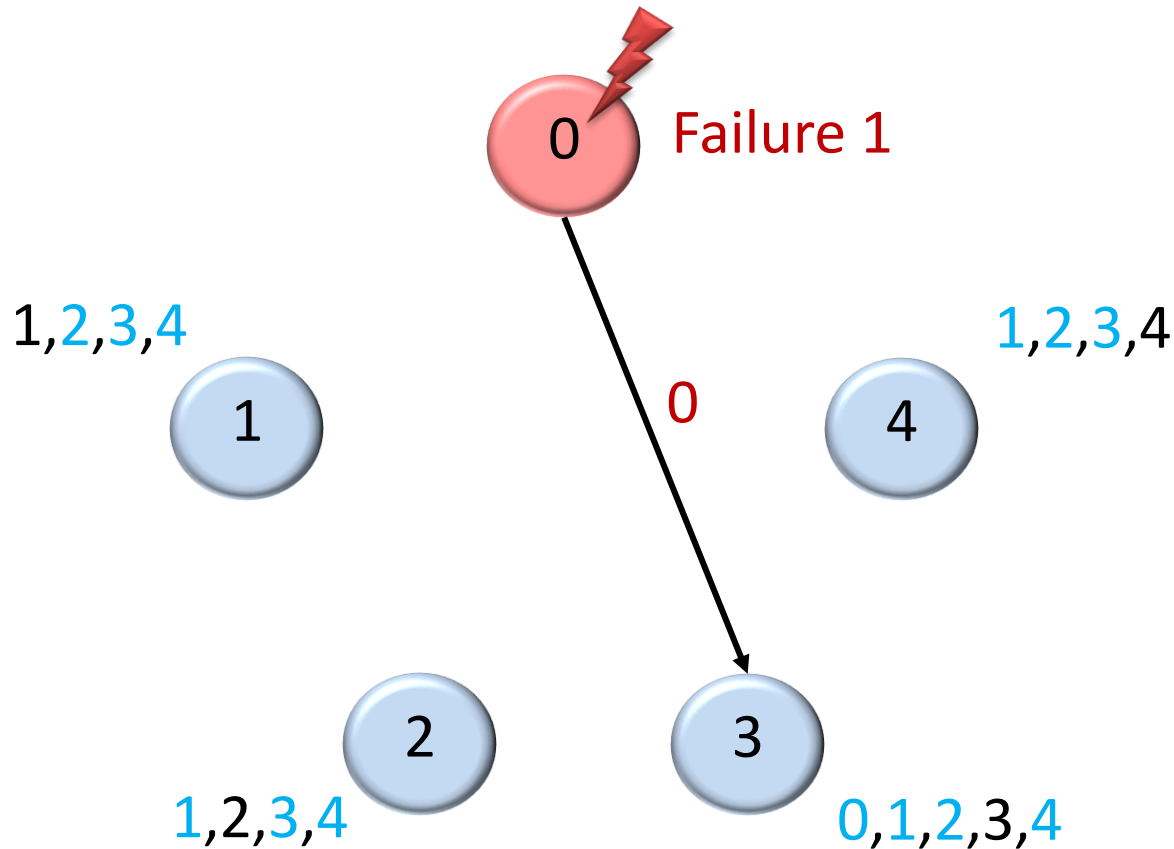
An f -Resilient Consensus Algorithm

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



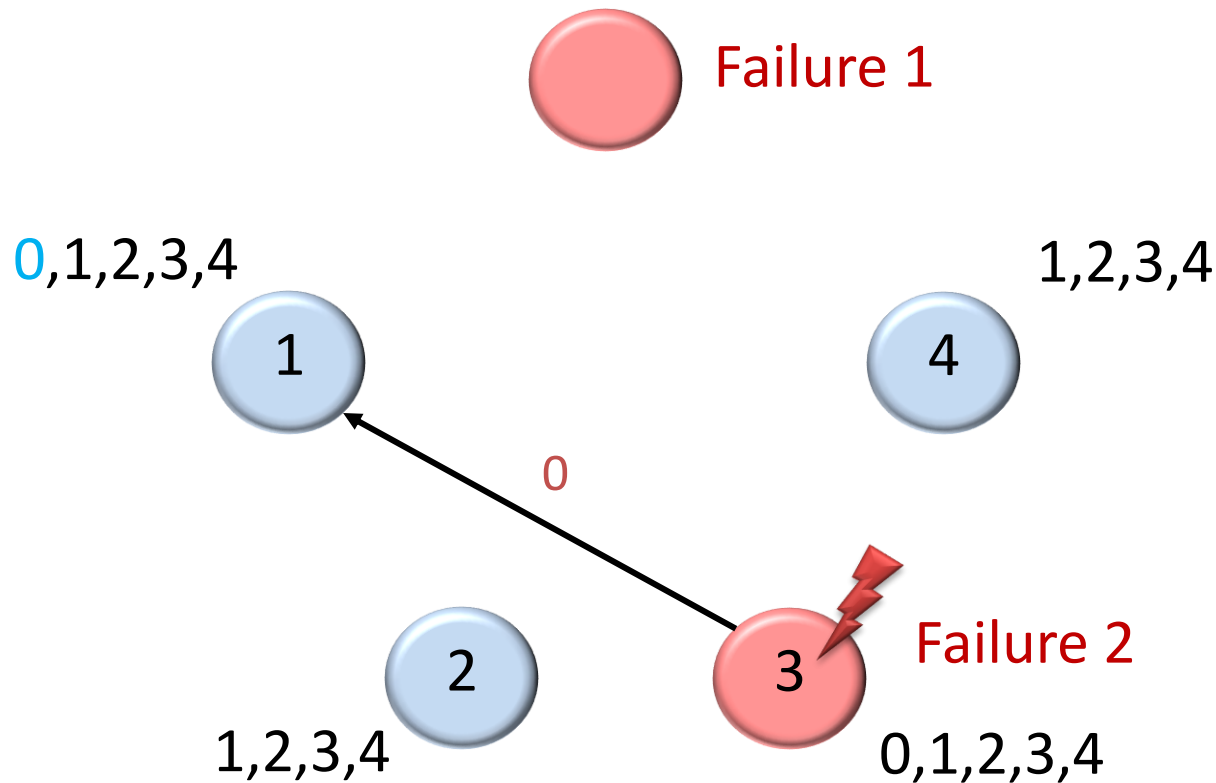
An f -Resilient Consensus Algorithm

- Round 1: Broadcast all values to everybody



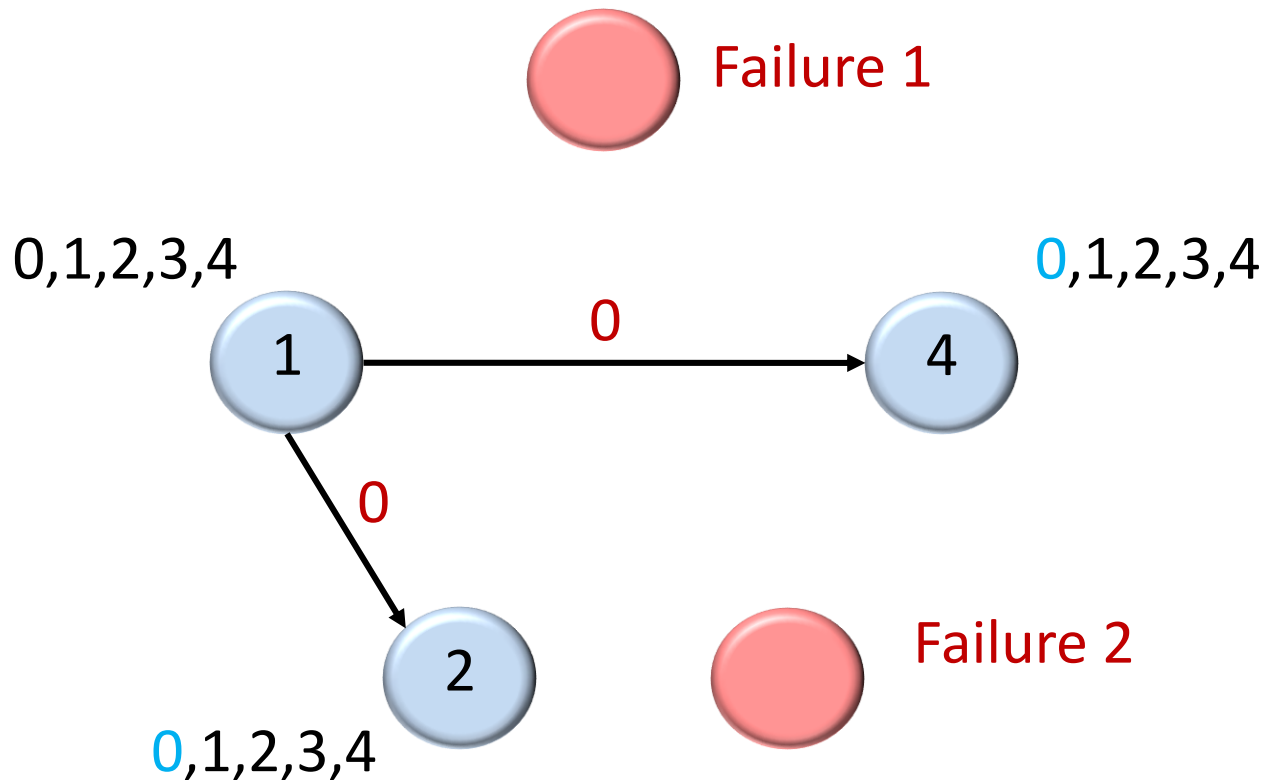
An f -Resilient Consensus Algorithm

- Round 2: Broadcast all new values to everybody



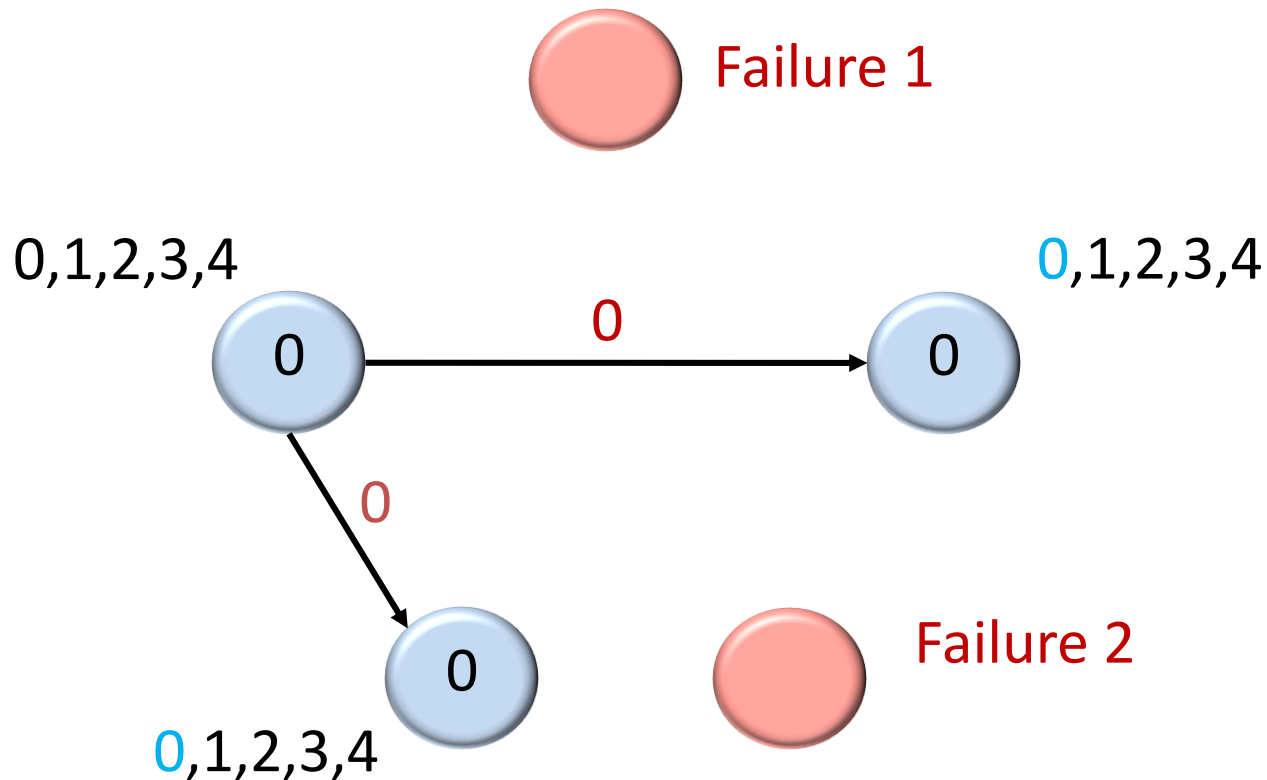
An f -Resilient Consensus Algorithm

- Round 3: Broadcast all new values to everybody



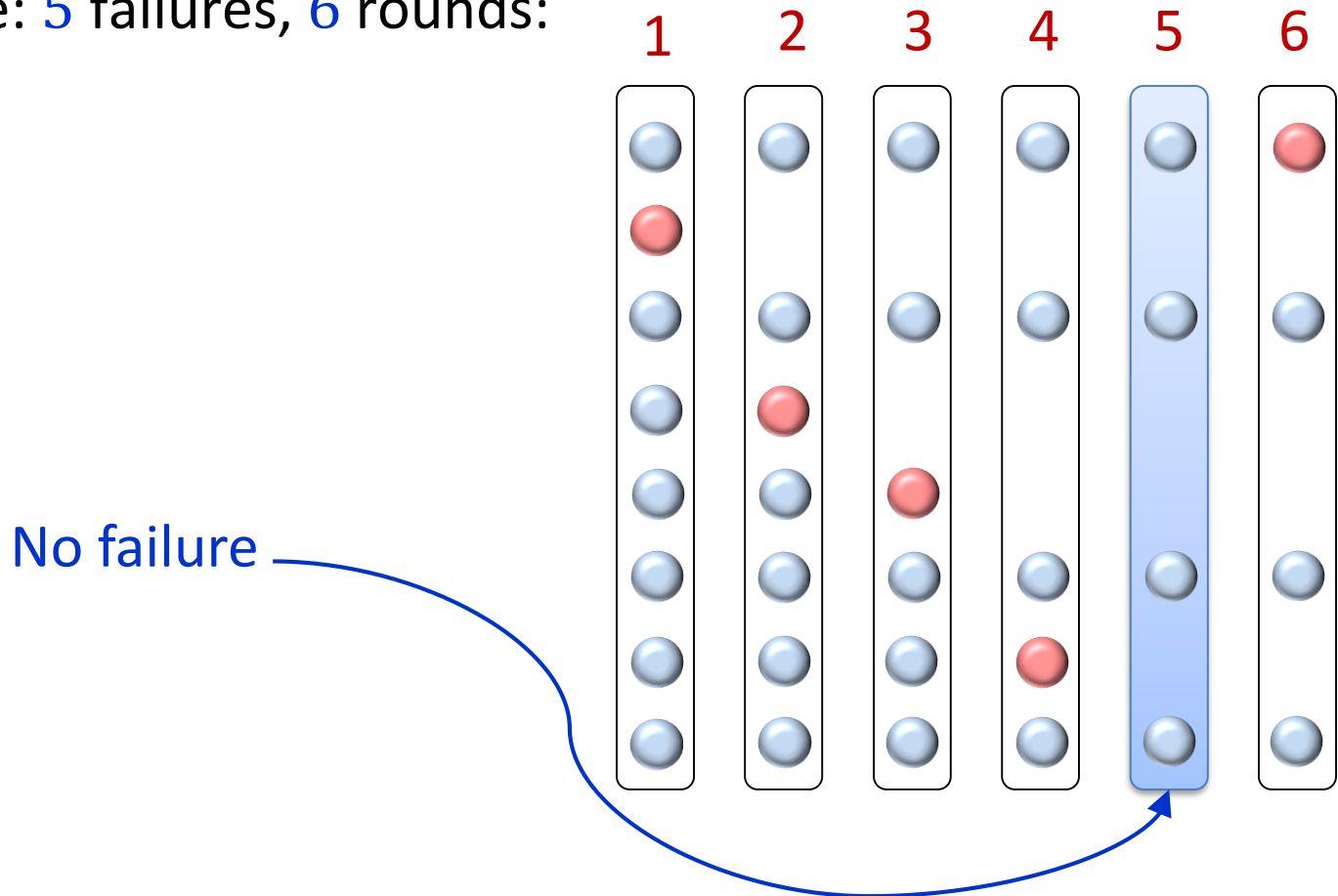
An f -Resilient Consensus Algorithm

- Decide on minimum \rightarrow Consensus!



Analysis

- 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, everybody will 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:** When all processes start with the same input value, then consensus is that value

Theorem

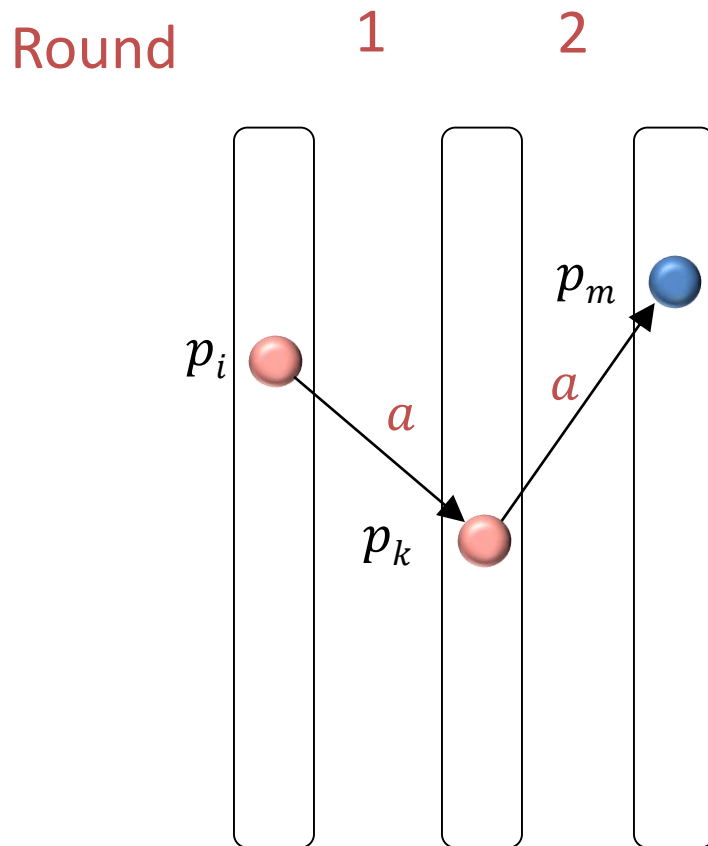
If at most $f \leq n - 2$ of n nodes of a synchronous message passing system can crash, at least $f + 1$ rounds are needed to solve consensus.

Proof idea:

- Show that f rounds are not enough if $n \geq f + 2$
- Before proving the theorem, we consider a

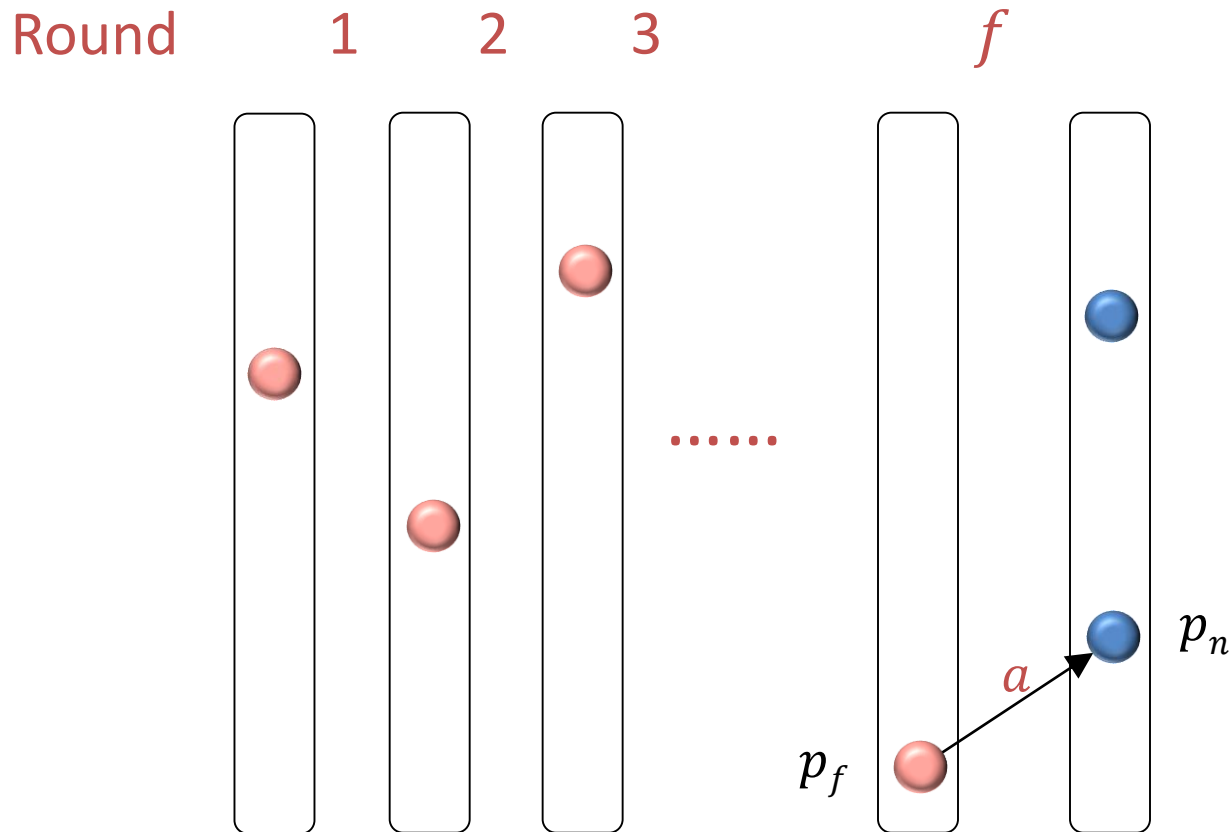
“worst-case scenario”: In each round one of the processes fails

Lower Bound on Rounds: Intuition



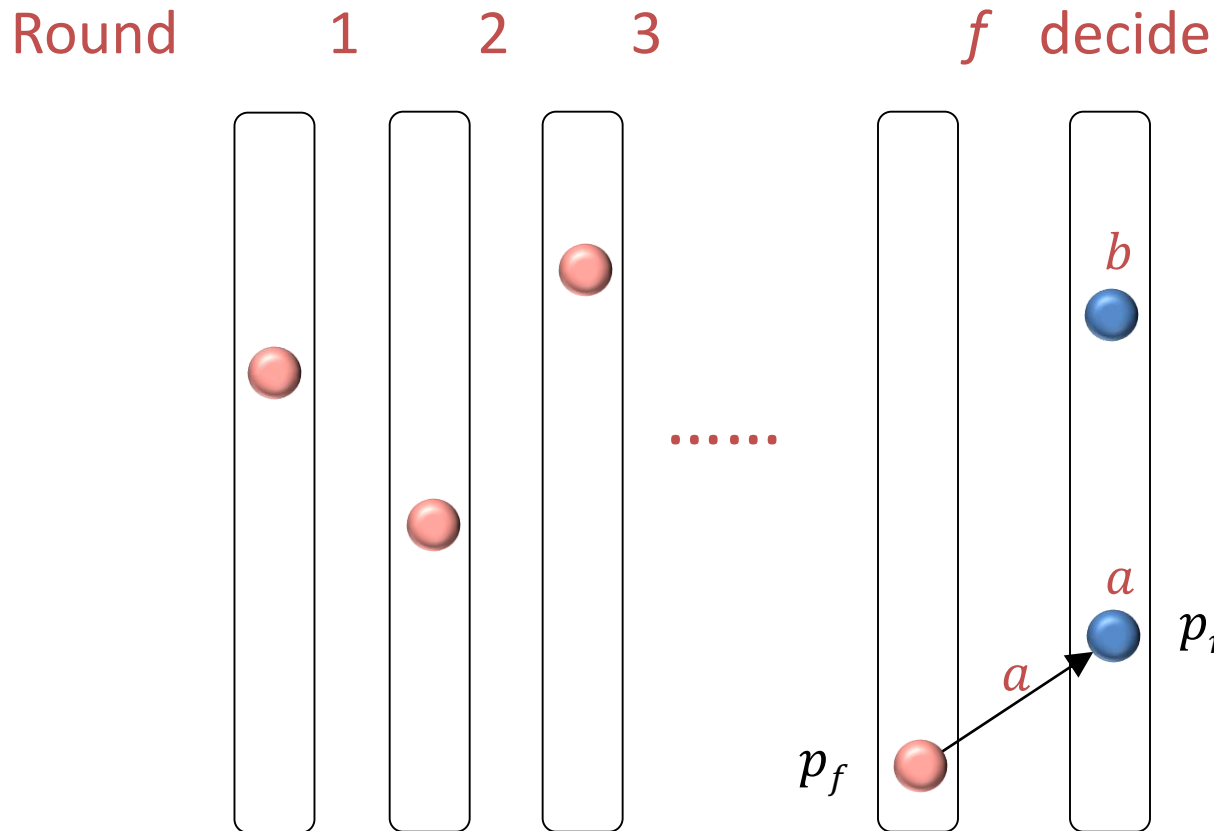
- Before process p_i fails, it sends its value a only to one process p_k
- Before process p_k fails, it sends its value a to only one process p_m

Lower Bound on Rounds: Intuition



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

Lower Bound on Rounds: Intuition



- Process p_n may decide on a and all other processes may decide on another value b
- f rounds are not enough
 \Rightarrow at least $f + 1$ rounds are needed