Consistency & Shared Memory

Part 2, Chapter 13

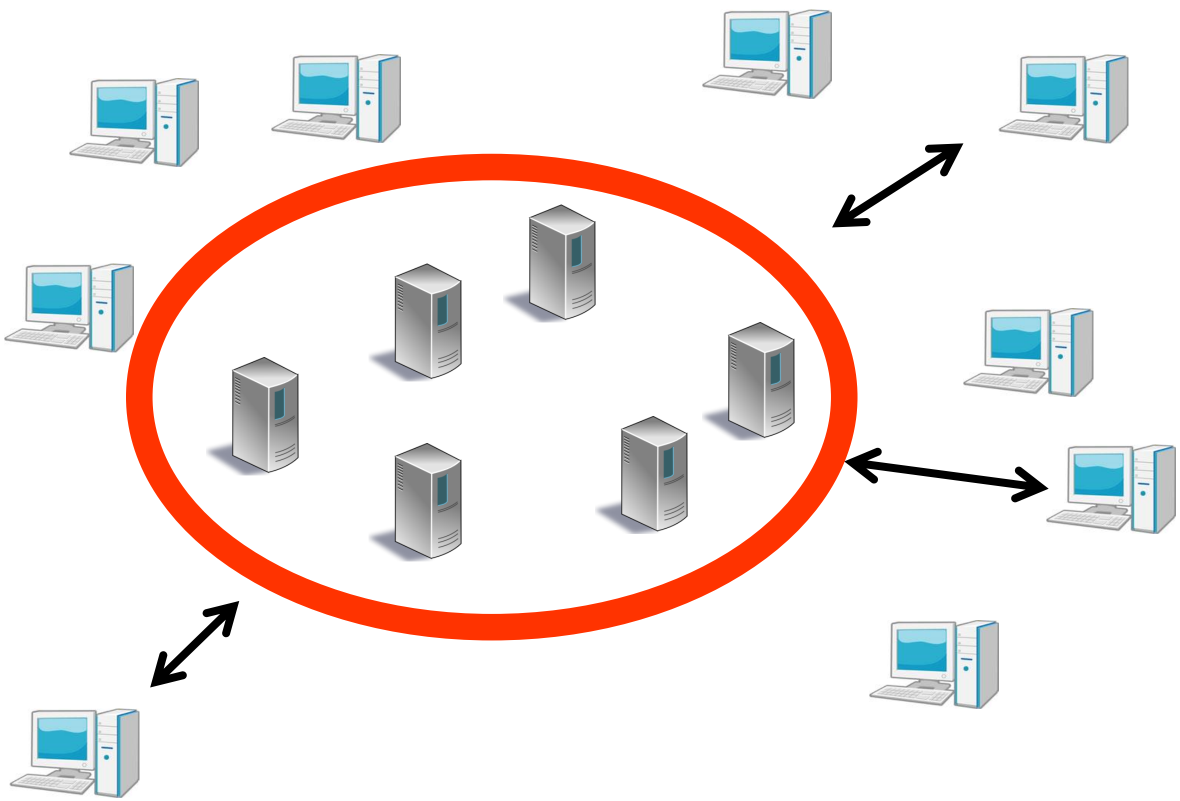
Roger Wattenhofer

Overview

- Consistency
- Shared Memory

Consistency Models (Client View)

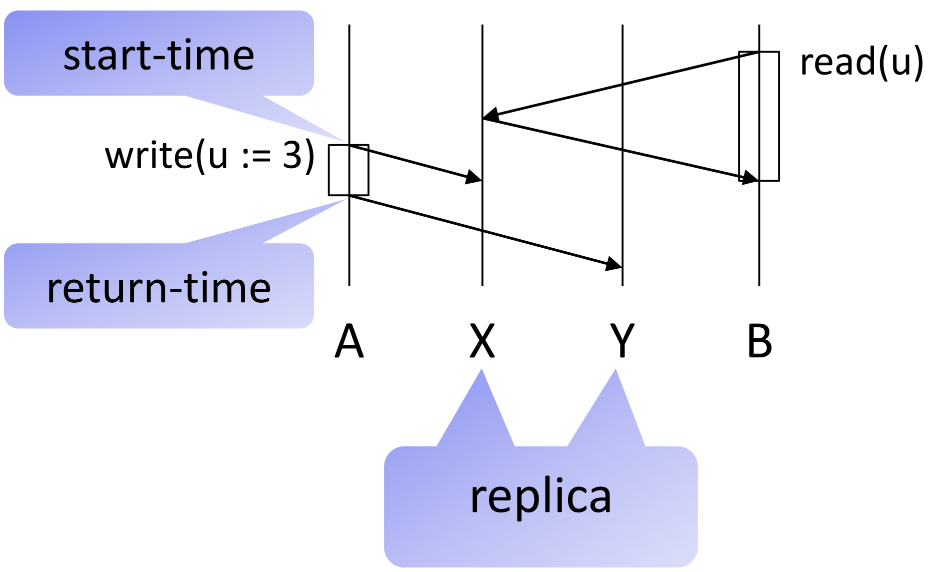
- Interface that describes the system behavior (abstract away implementation details)
- If clients read/write data, they expect the behavior to be the same as for a single storage cell.



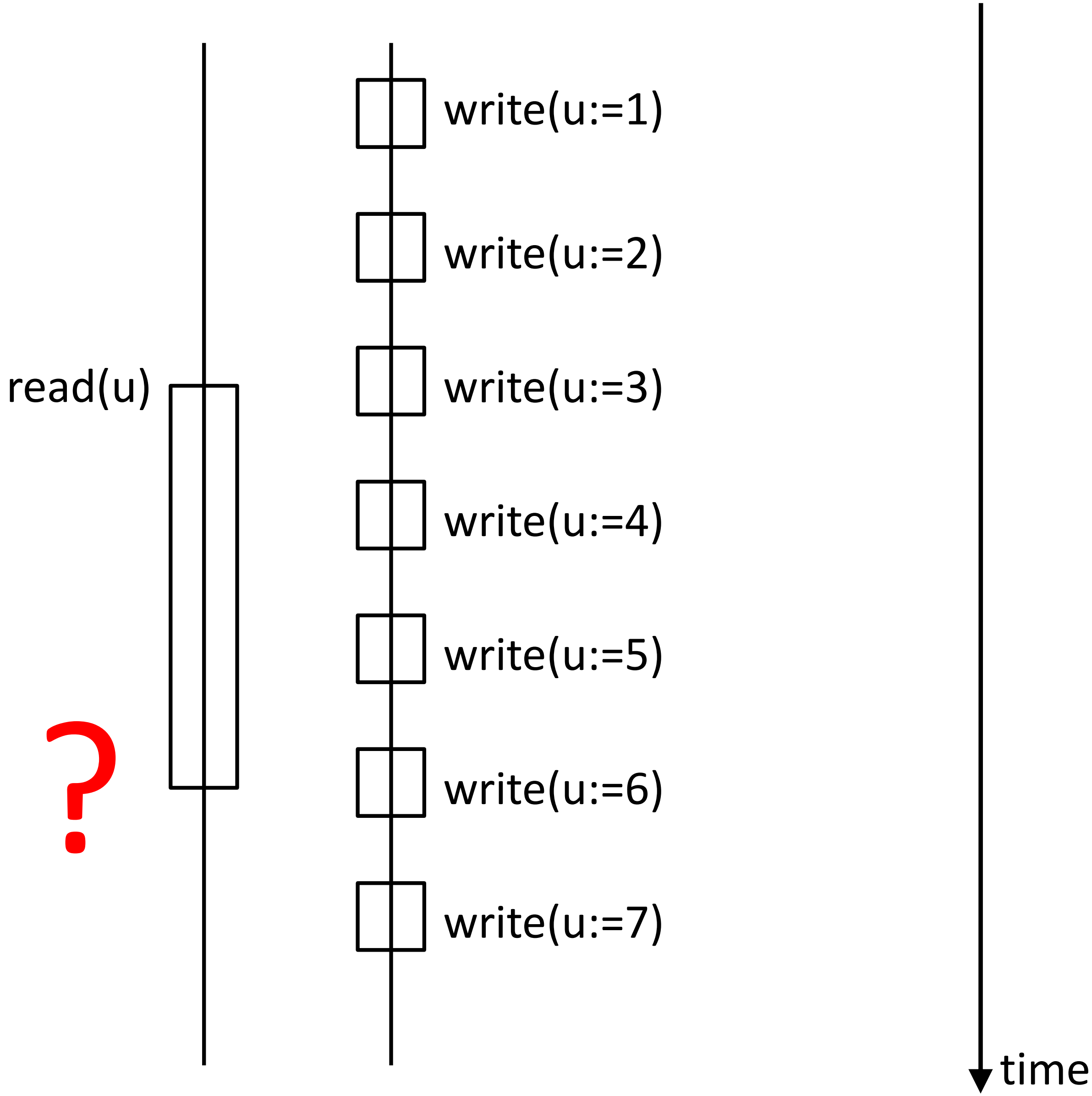
Example

- We have memory that supports 3 types of operations:
  - write(u := x): write value x to the memory location at address u
  - read(u): Read value stored at address u and return it
  - snapshot(): return a map that contains all address-value pairs

- Each operation has a start-time T_S and return-time T_R (time it returns to the invoking client). The duration is given by $T_R - T_S$.



Motivation



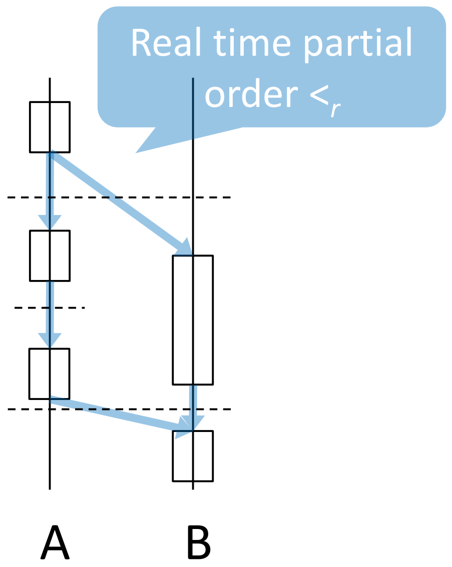
write(u:=1)
write(u:=2)
read(u)
write(u:=3)
write(u:=4)
write(u:=5)
?
write(u:=6)
write(u:=7)
time

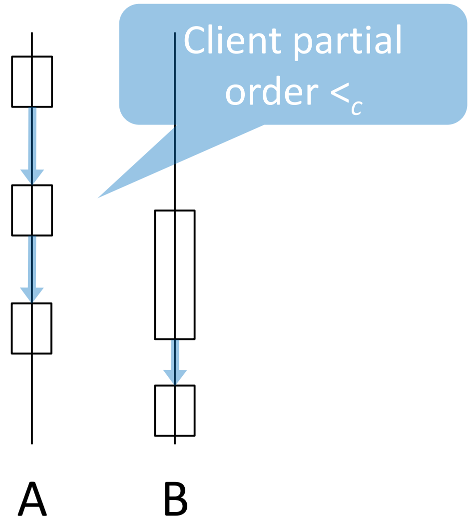
Executions

- We look at executions E that define the (partial) order in which processes invoke operations.

- Real-time partial order of an execution $<_r$:
  - $p <_r q$ means that duration of operation p occurs entirely before duration of q (i.e., p returns before the invocation of q in real time).

- Client partial order $<_c$:
  - $p <_c q$ means p and q occur at the same client, and that p returns before q is invoked.





Strong Consistency: Linearizability

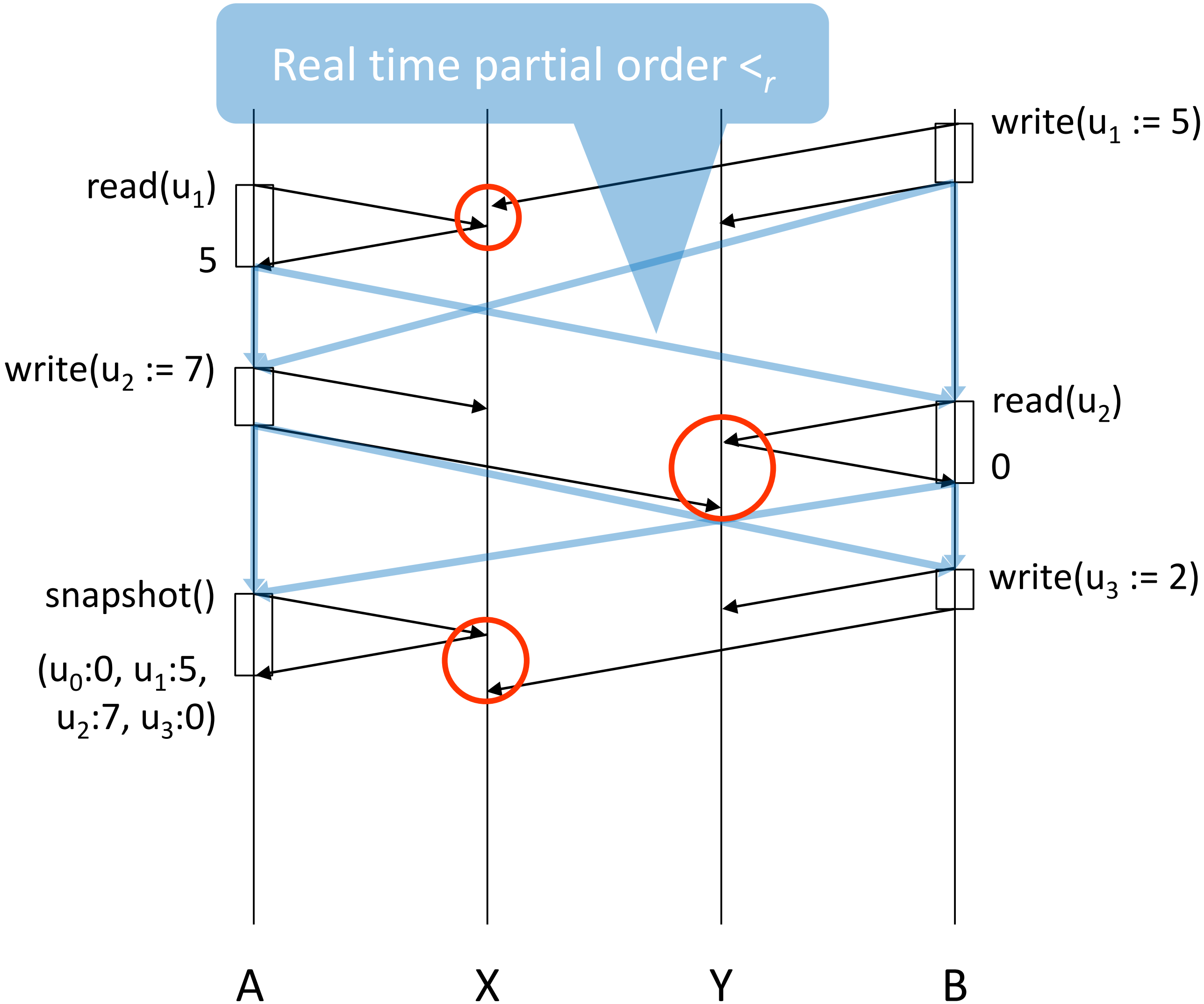
- A replicated system is called linearizable if it behaves exactly as a single-site (unreplicated) system.

Definition

Execution E is linearizable if there exists a sequence H such that:

1) H contains exactly the same operations as E, each paired with the return value received in E
2) The total order of operations in H is compatible with the real-time partial order $<_r$
3) H is a legal history of the data type that is replicated

Example: Linearizable Execution



Valid sequence H:

1.) write(u_1 := 5)
2.) read(u_1) → 5
3.) read(u_2) → 0
4.) write(u_2 := 7)
5.) snapshot() →
(u_0: 0, u_1: 5, u_2:7, u_3:0)
6.) write(u_3 := 2)

For this example, this is the only valid H. In general there might be several sequences H that fullfil all required properties.

Strong Consistency: Sequential Consistency

- Orders at different locations are disregarded if it cannot be determined by any observer within the system.

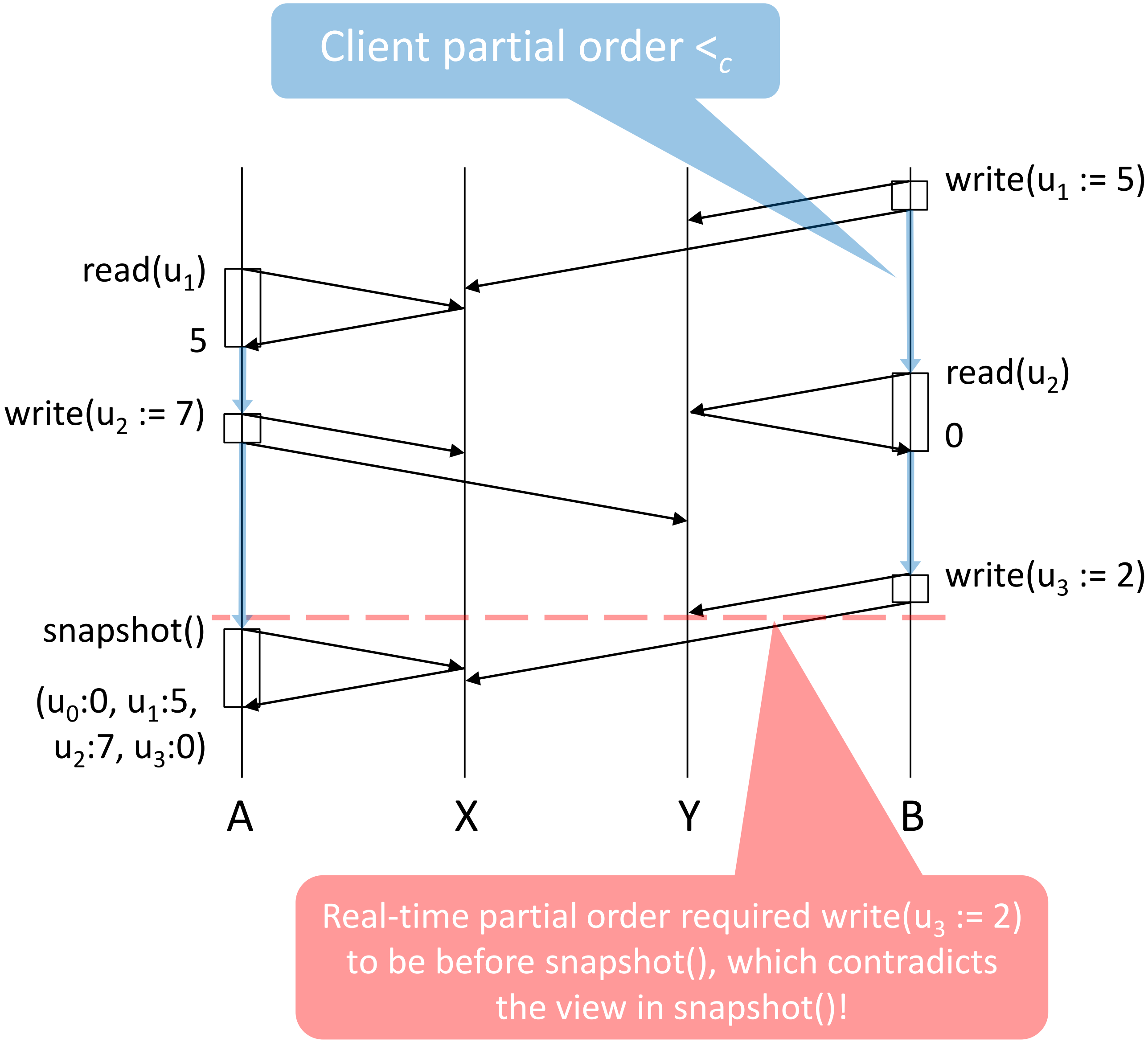
- I.e., a system provides sequential consistency if every node of the system sees the (write) operations on the same memory address in the same order, although the order may be different from the order as defined by real time (as seen by a hypothetical external observer or global clock).

Definition

Execution E is sequentially consistent if there exists a sequence H such that:

1) H contains exactly the same operations as E, each paired with the return value received in E
2) The total order of operations in H is compatible with the client partial order $<_c$
3) H is a legal history of the data type that is replicated

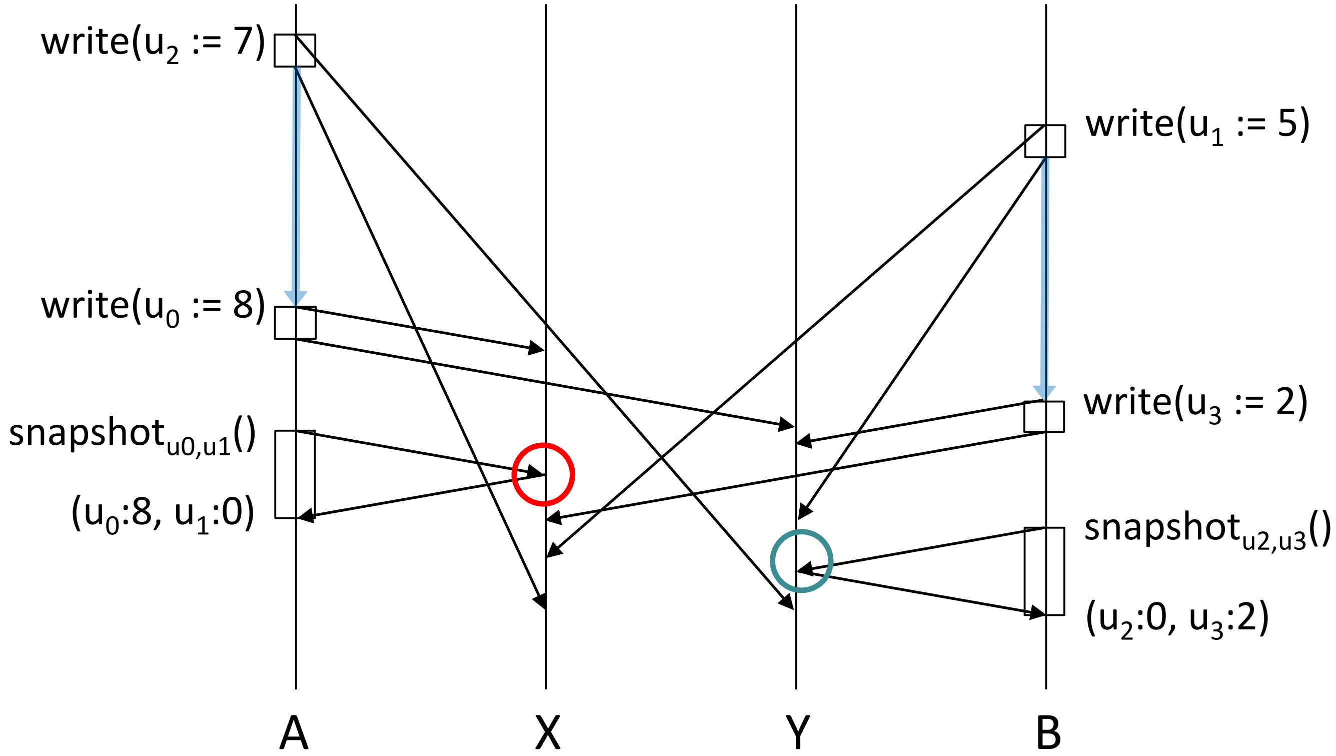
Example: Sequentially Consistent

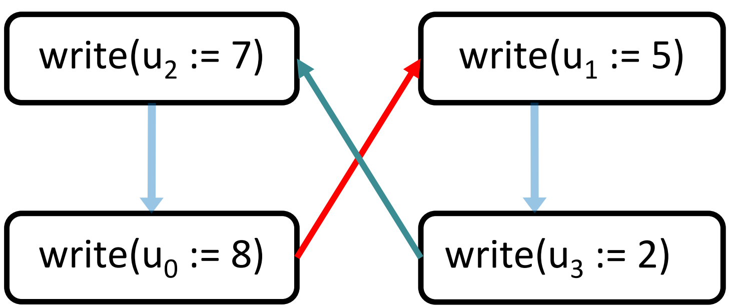


Valid sequence H:

1.) write(u_1 := 5)
2.) read(u_1) → 5
3.) read(u_2) → 0
4.) write(u_2 := 7)
5.) snapshot() →
   (u_0:0, u_1:5, u_2:7, u_3:0)
6.) write(u_3 := 2)

Is Every Execution Sequentially Consistent?

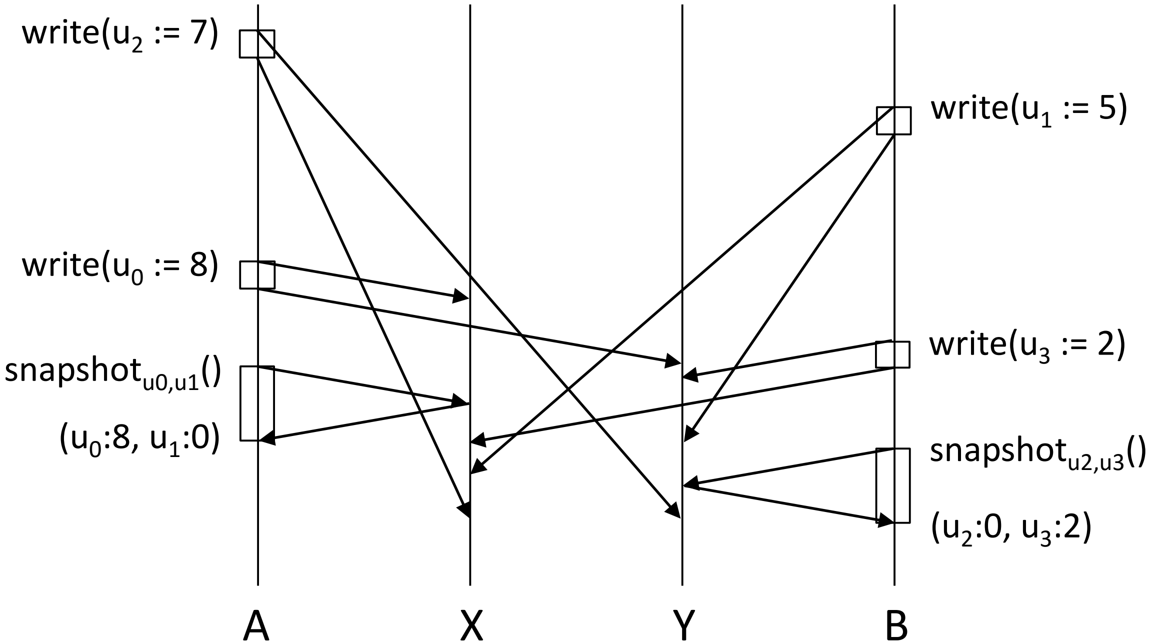




Circular dependencies!

I.e., there is no valid total order and thus above execution is not sequentially consistent

Sequential Consistency does not Compose



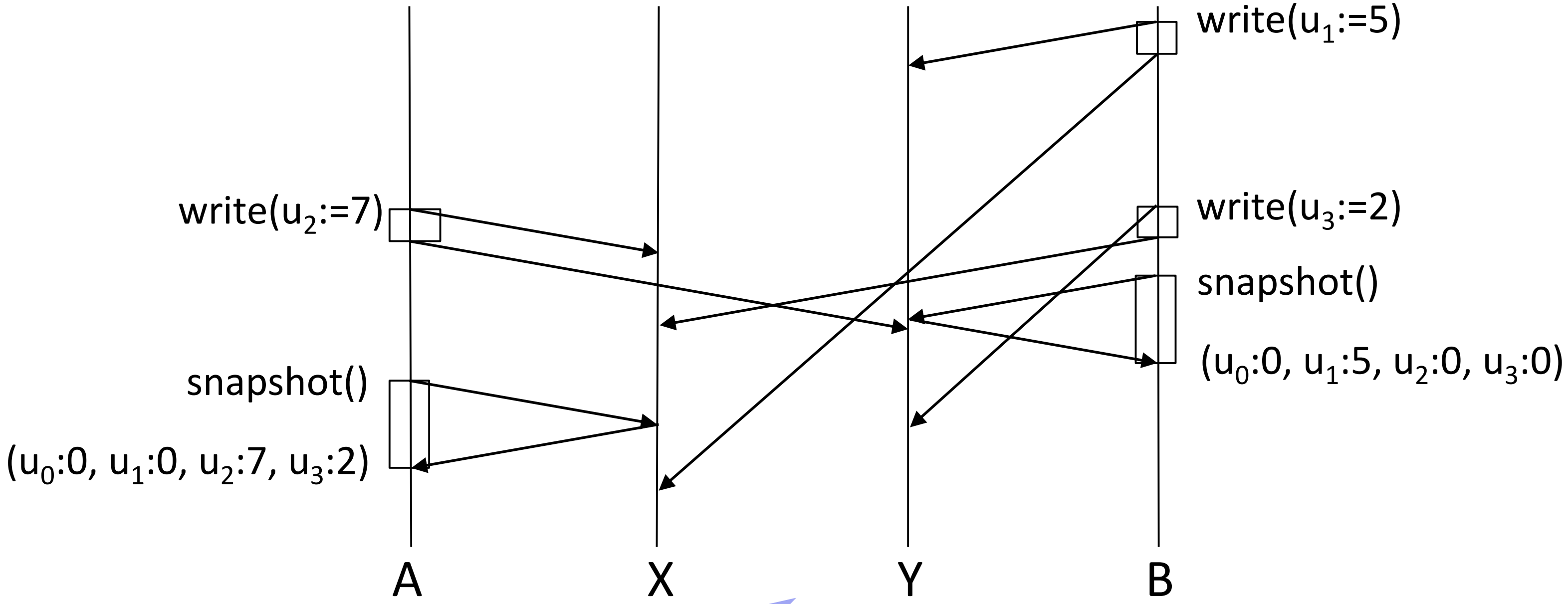
- If we only look at data items 0 and 1, operations are sequentially consistent
- If we only look at data items 2 and 3, operation are also sequentially consistent
- But, as we have seen before, the combination is not sequentially consistent

Sequential consistency does not compose!

(this is in contrast to linearizability)

Weak Consistency

- A considerable performance gain can result if messages are transmitted independently, and applied to each replica whenever they arrive.
  - But: Clients can see inconsistencies that would never happen with unreplicated data.



This execution is NOT sequentially consistent

Causal Consistency

Definition

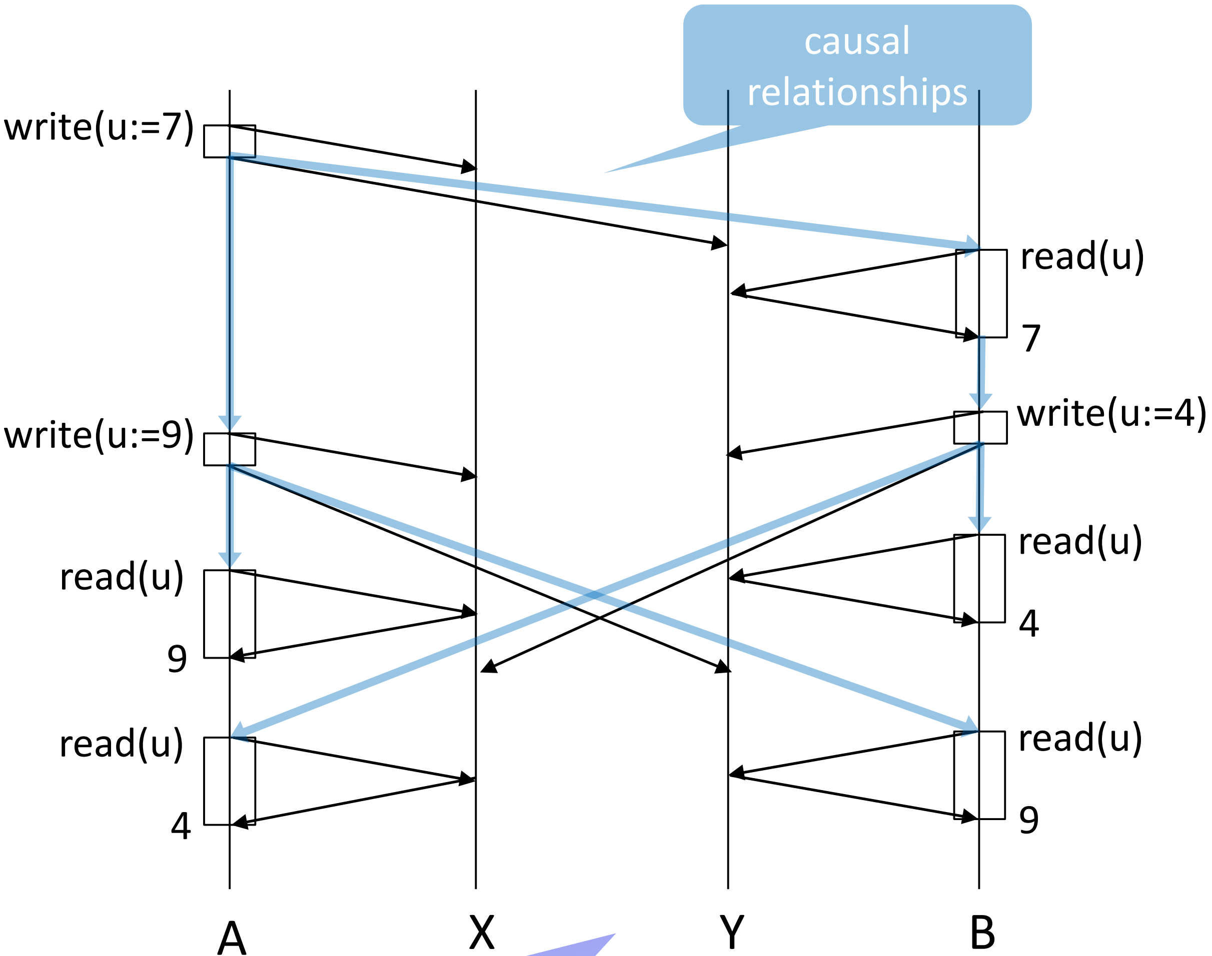
A system provides causal consistency if memory operations that potentially are *causally related* are seen by every node of the system in the same order. Concurrent writes (i.e. ones that are not causally related) may be seen in different order by different nodes.

Definition

The following pairs of operations are causally related:

- Two writes by the same process to any memory location.
- A read followed by a write of the same process (even if the write addresses a different memory location).
- A read that returns the value of a write from any process.
- Two operations that are transitively related according to the above conditions.

Causal Consistency: Example



This execution is causally consistent, but NOT sequentially consistent

Weak Consistency: More Concepts

Definition

Monotonic Read Consistency

If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.

Definition

Monotonic Write Consistency

A write operation by a process on a data item *u* is completed before any successive write operation on *u* by the same process (i.e. system guarantees to serialize writes by the same process).

Definition

Read-your-Writes Consistency

After a process has updated a data item, it will never see an older value on subsequent accesses.

Weak Consistency: Eventual Consistency

Definition

Eventual Consistency

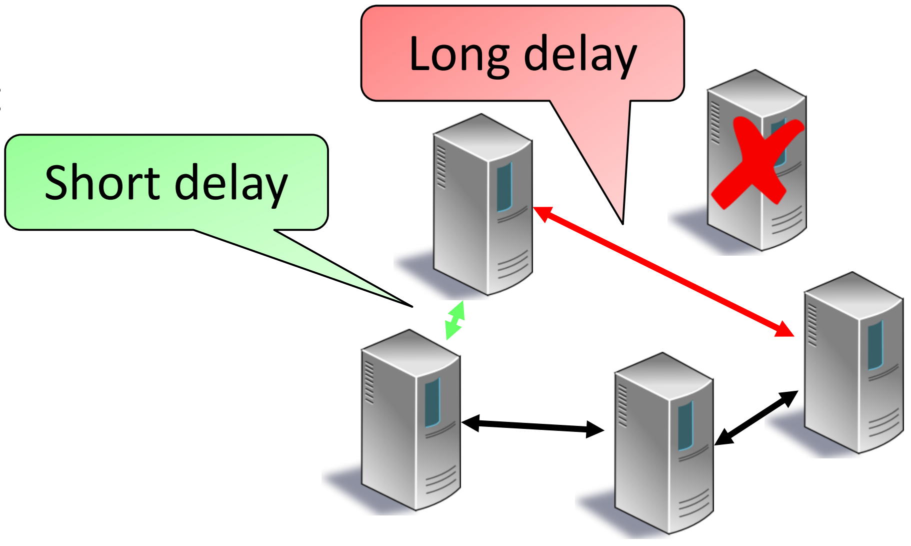
If no new updates are made to the data object, eventually all accesses will return the last updated value.

- Special form of weak consistency
- Allows for „disconnected operation“
- Requires some conflict resolution mechanism
  - After conflict resolution all clients see the same order of operations up to a certain point in time („agreed past“).
  - Conflict resolution can occur on the server-side or on the client-side

Transactions

- In order to achieve consistency, updates have to be atomic
- A write has to be an atomic transaction
  - Updates are synchronized

- Either all nodes (servers) commit a transaction or all abort
- How do we handle transactions in asynchronous systems?
  - Unpredictable messages delays!
- Moreover, any node may fail…
  - Recall that this problem cannot be solved in theory!



Shared Memory Consensus

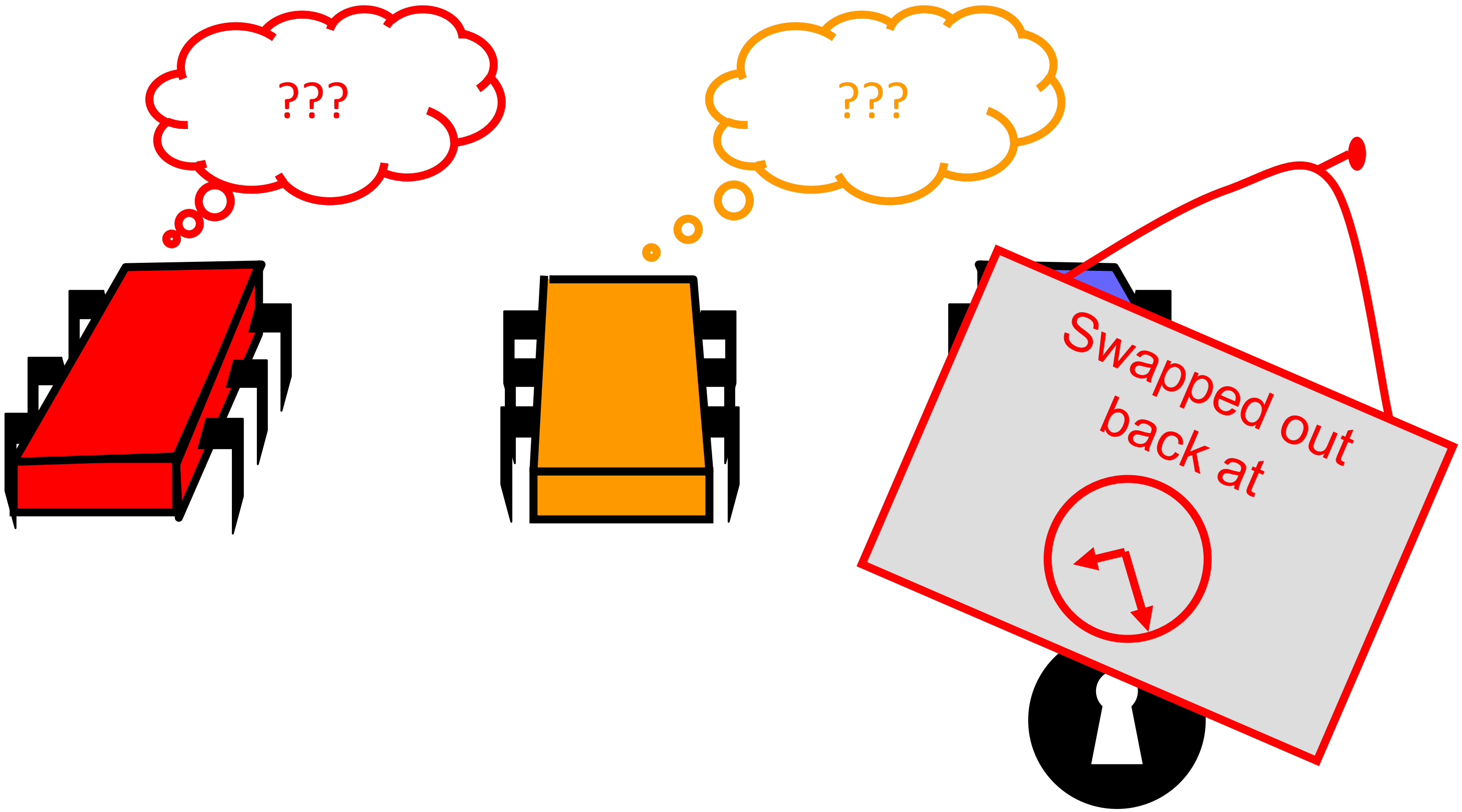
- $n > 1$ processors
- Shared memory is memory that may be accessed simultaneously by multiple threads/processes.
- Processors can atomically *read* or *write* (not both) a shared memory cell

Protocol:

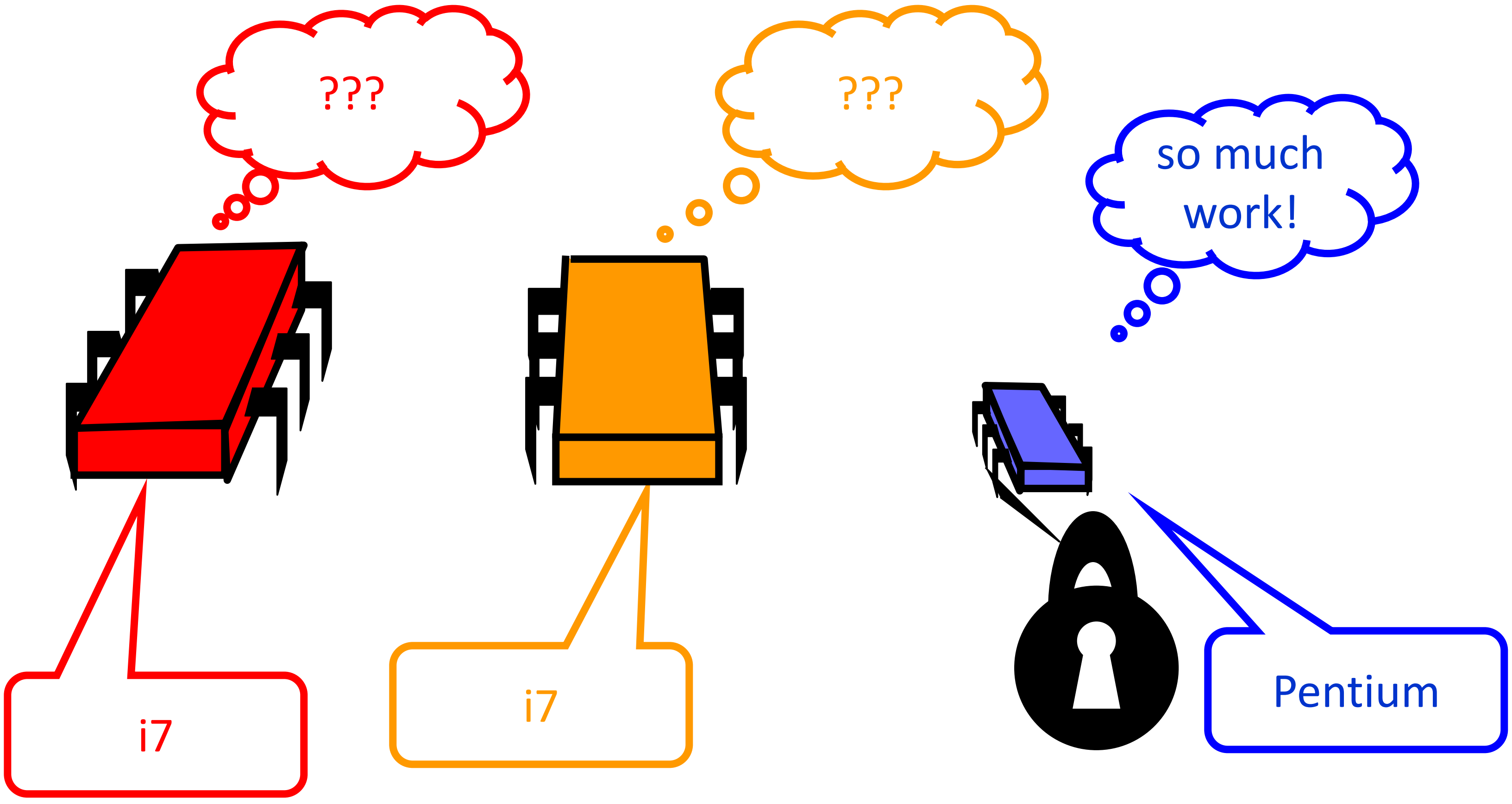
- There is a designated memory cell c.
- Initially c is in a special state "?"
- Processor 1 writes its value v_1 into c, then decides on v_1.
- A processor $j \neq 1$ reads c until j reads something else than "?", and then decides on that.

- Problems with this approach?

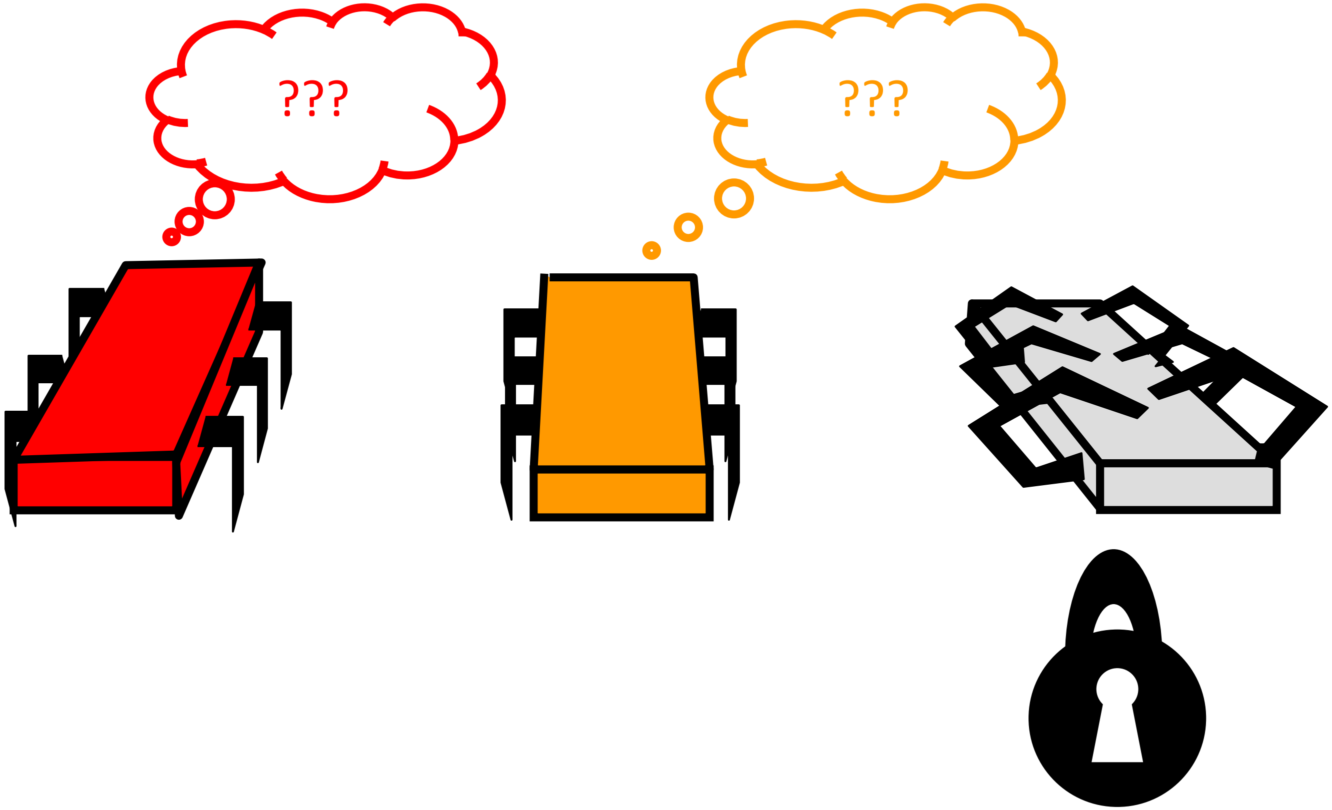
Unexpected Delay



Heterogeneous Architectures



Fault-Tolerance



Wait-free Shared Memory Consensus

- $n > 1$ 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 (method call) 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 (that is, reads and writes to same register do not overlap)

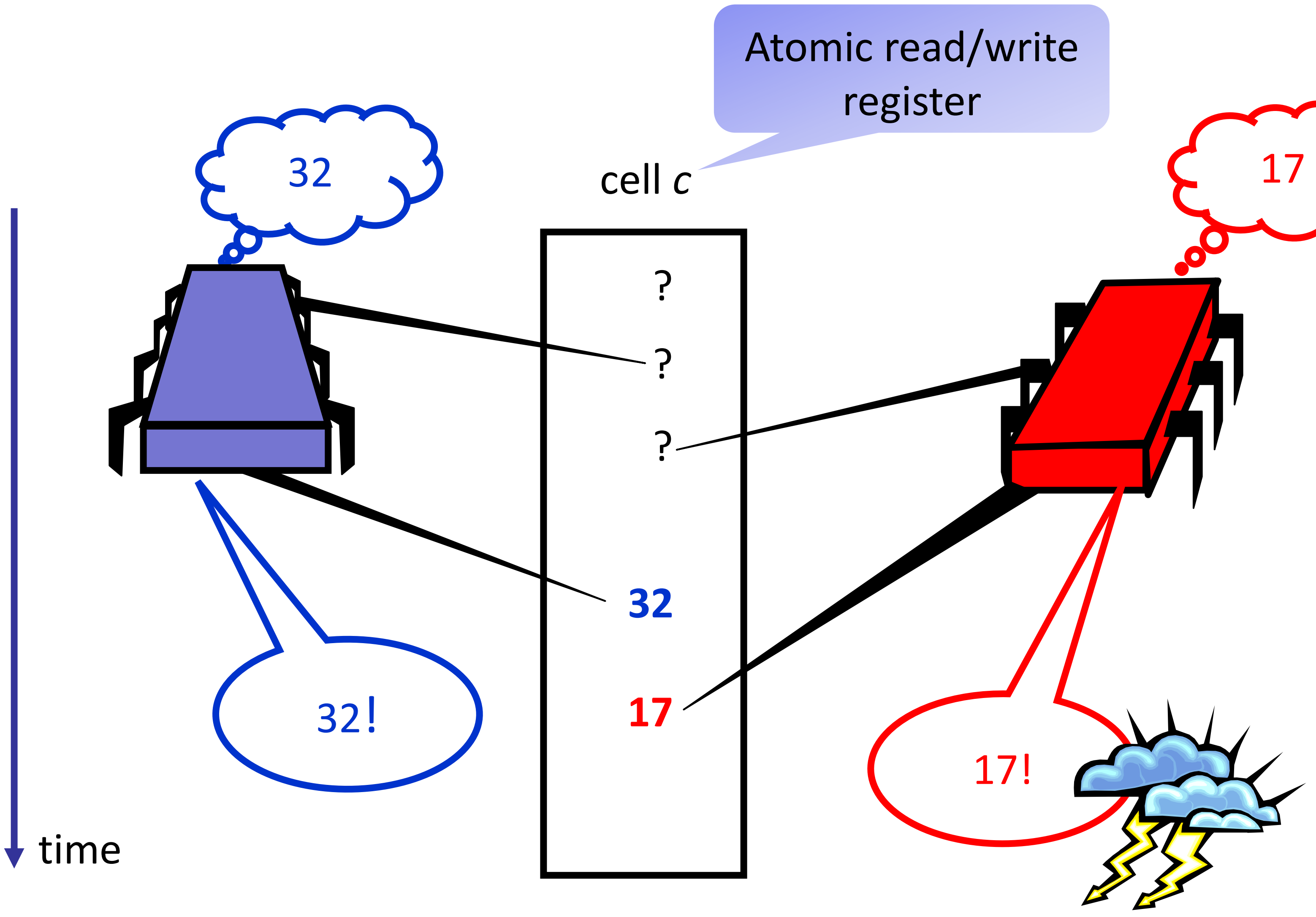
A Wait-free Algorithm

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

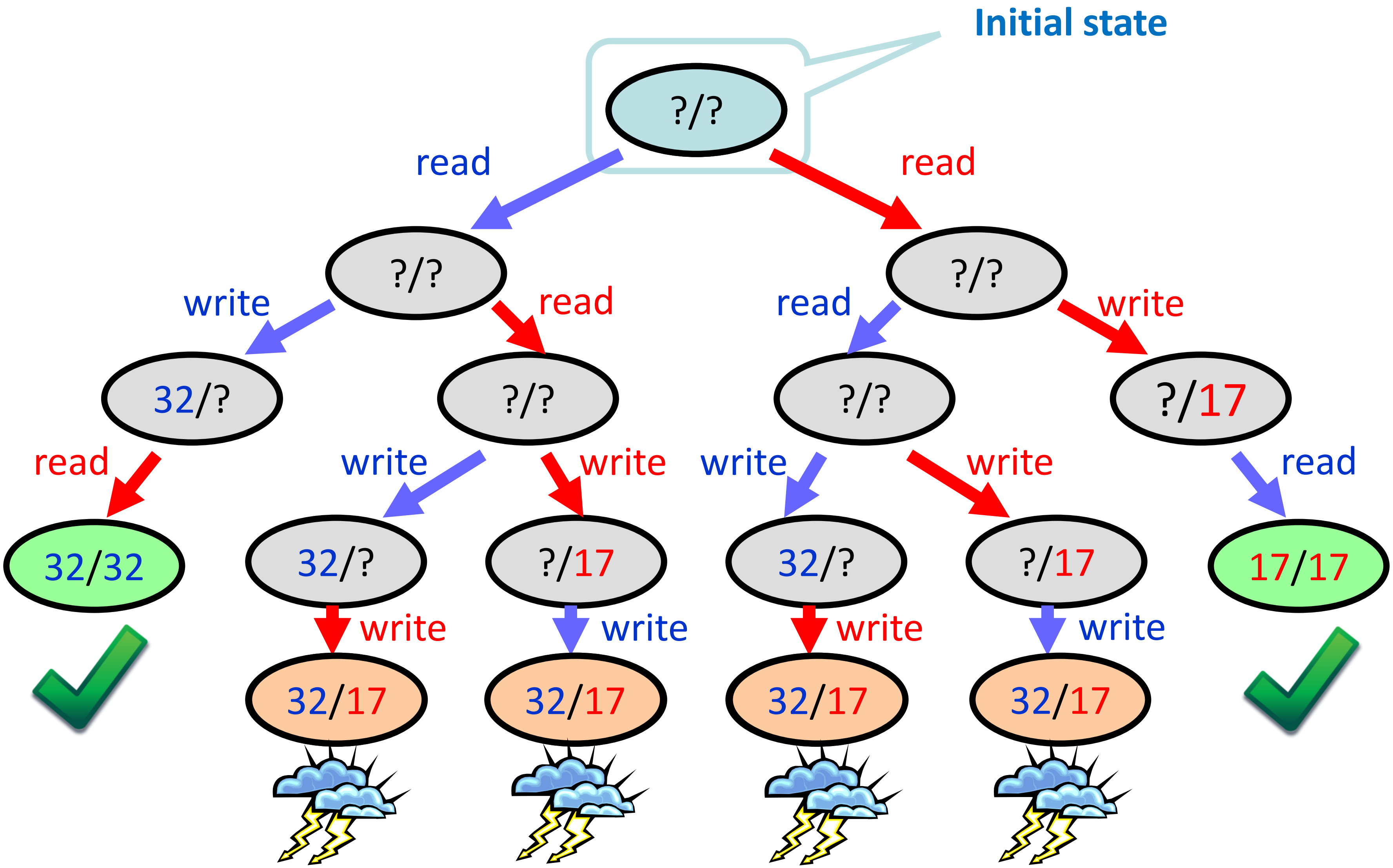
```
r = Read(c);
if (r == "?") then
  Write(c, v_i); decide v_i;
else
  decide r;
```

- Is this algorithm correct…?

An Execution



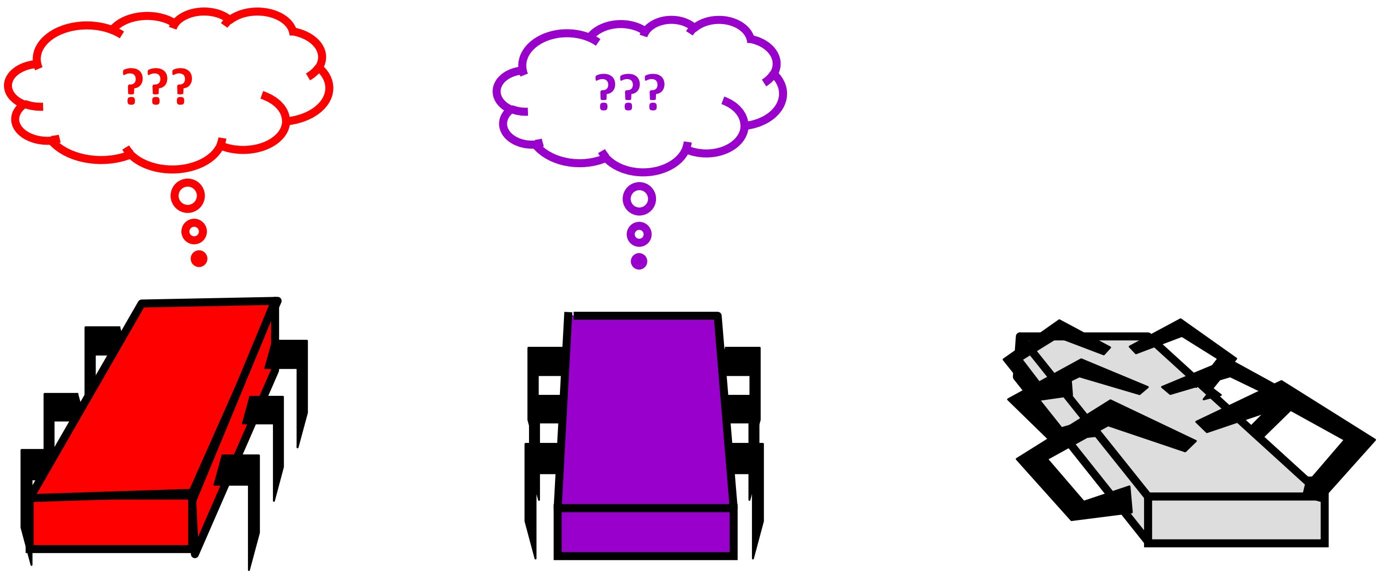
Execution Tree



Theorem

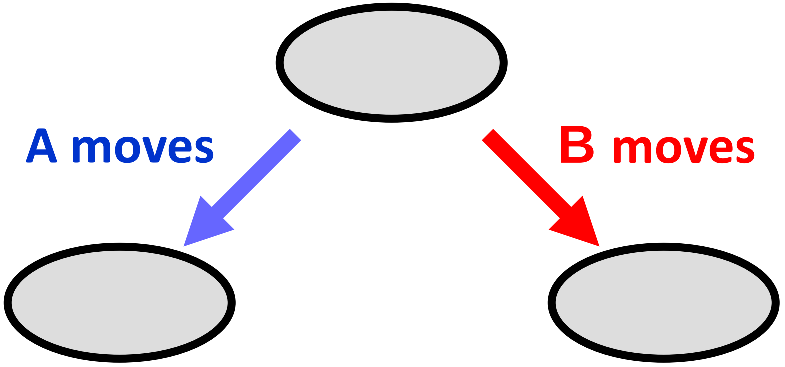
Theorem

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

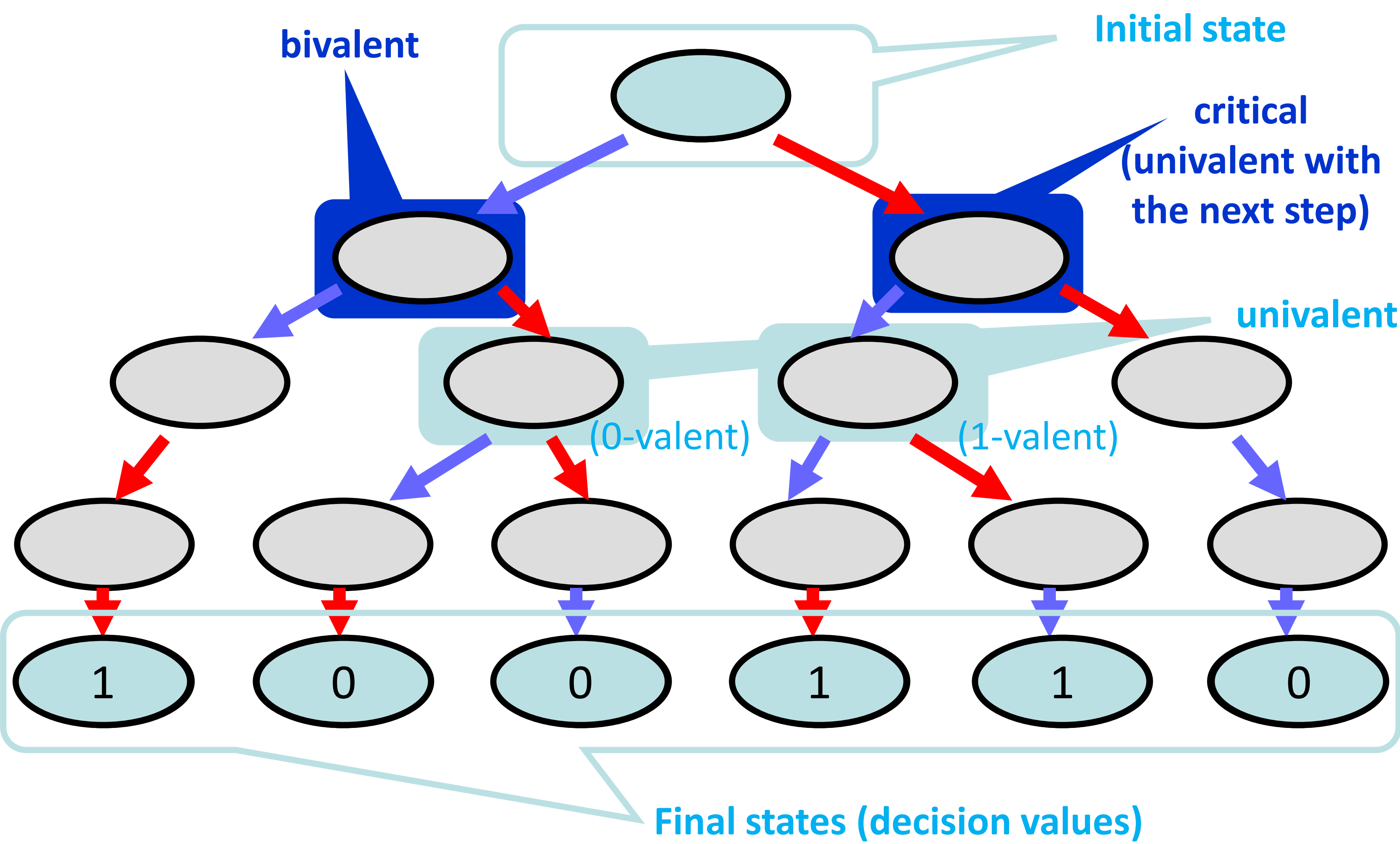


Proof

- Make it simple
  - There are only two threads 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 (of abstract but "correct" algorithm)



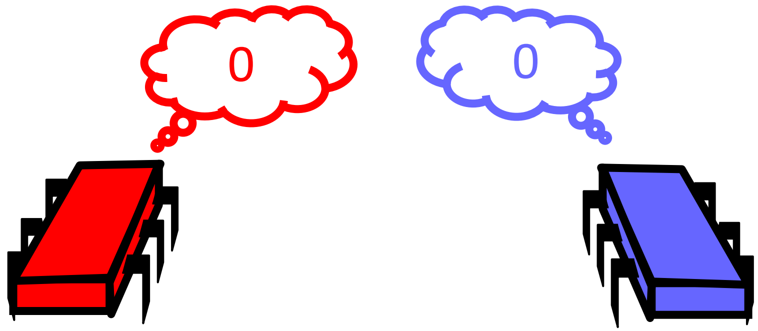
Bivalent vs. Univalent

- 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
  - This means that the outcome is not always fixed from the start

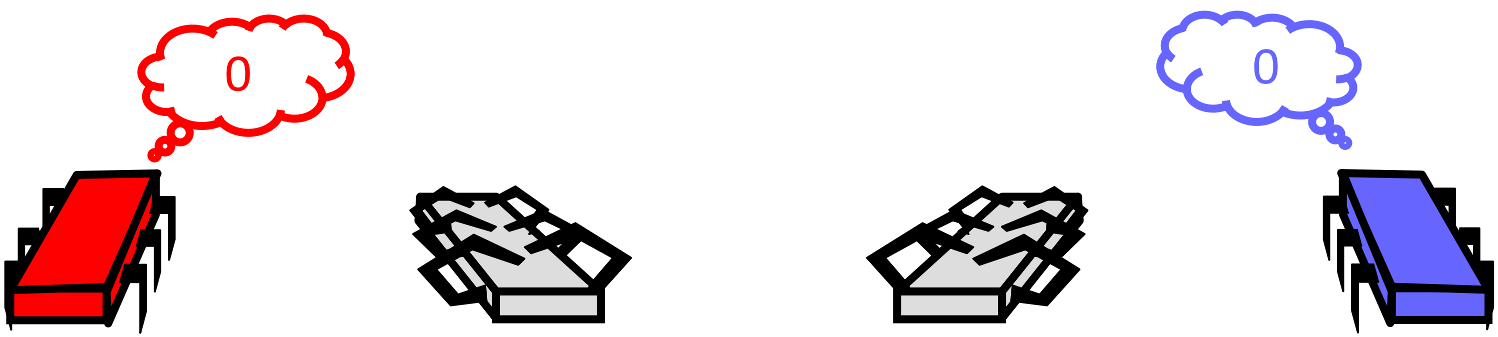
Proof of Claim: A 0-Valent Initial State

- All executions lead to the decision 0



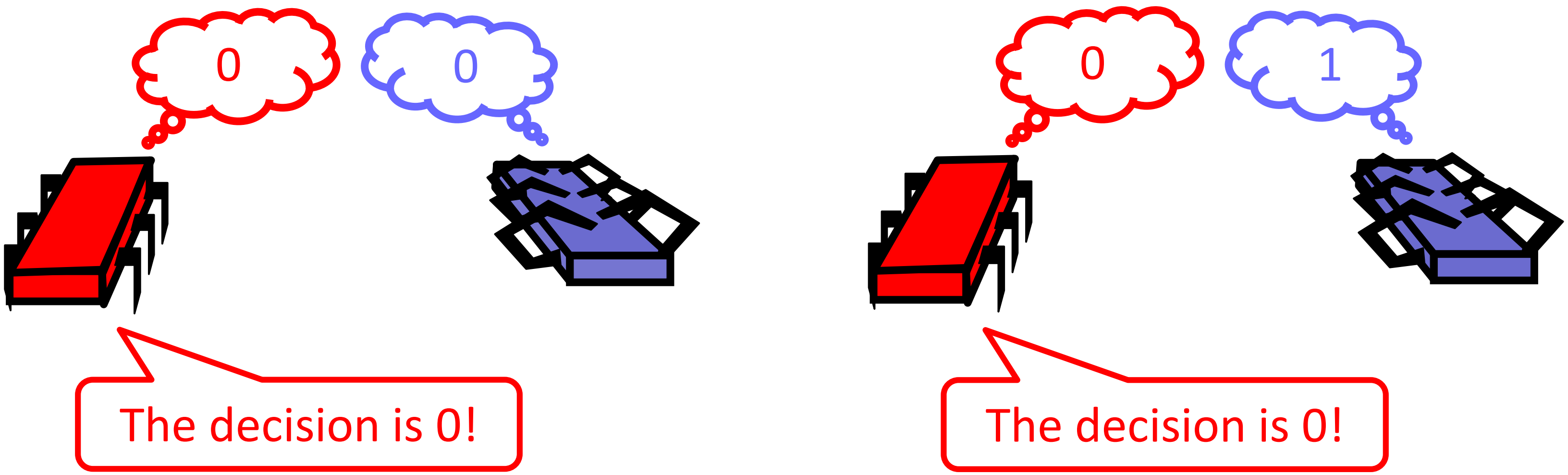
Similarly, the decision is always 1 if both threads start with 1!

- Solo executions also lead to the decision 0



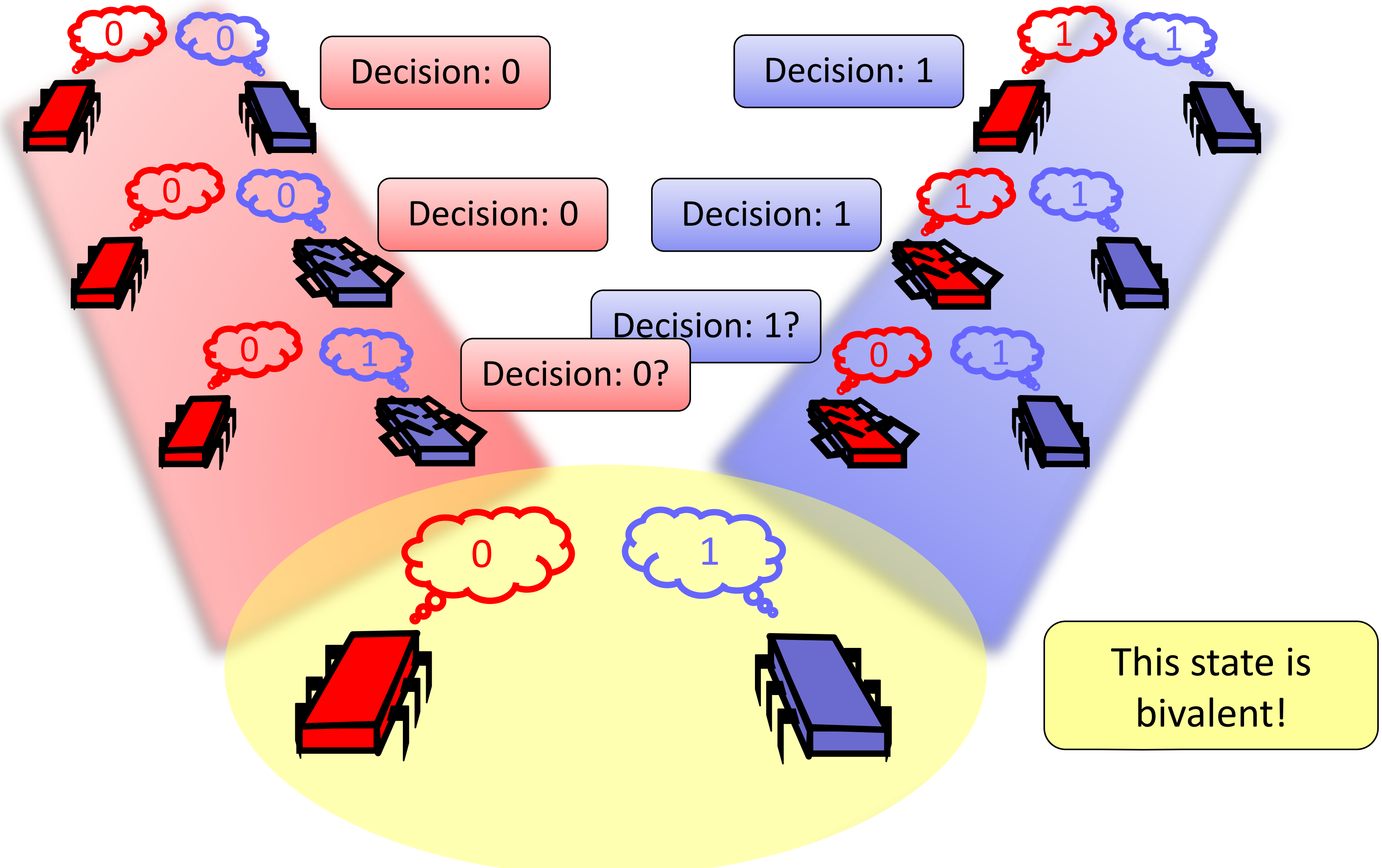
Proof of Claim: Indistinguishable Situations

- These two situations are indistinguishable → The outcome must be the same



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

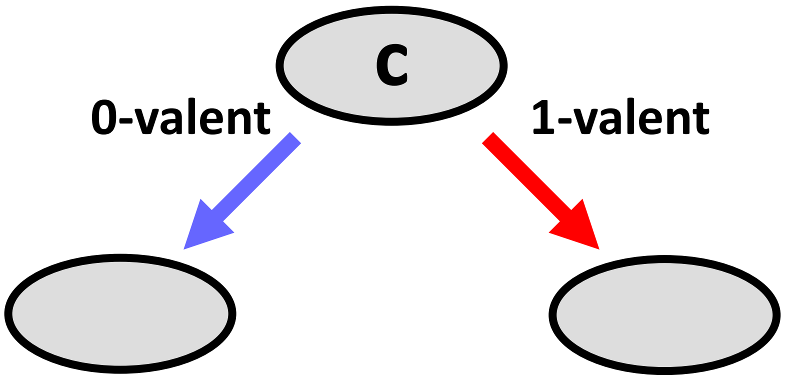
Proof of Claim: A Bivalent Initial State



Critical States

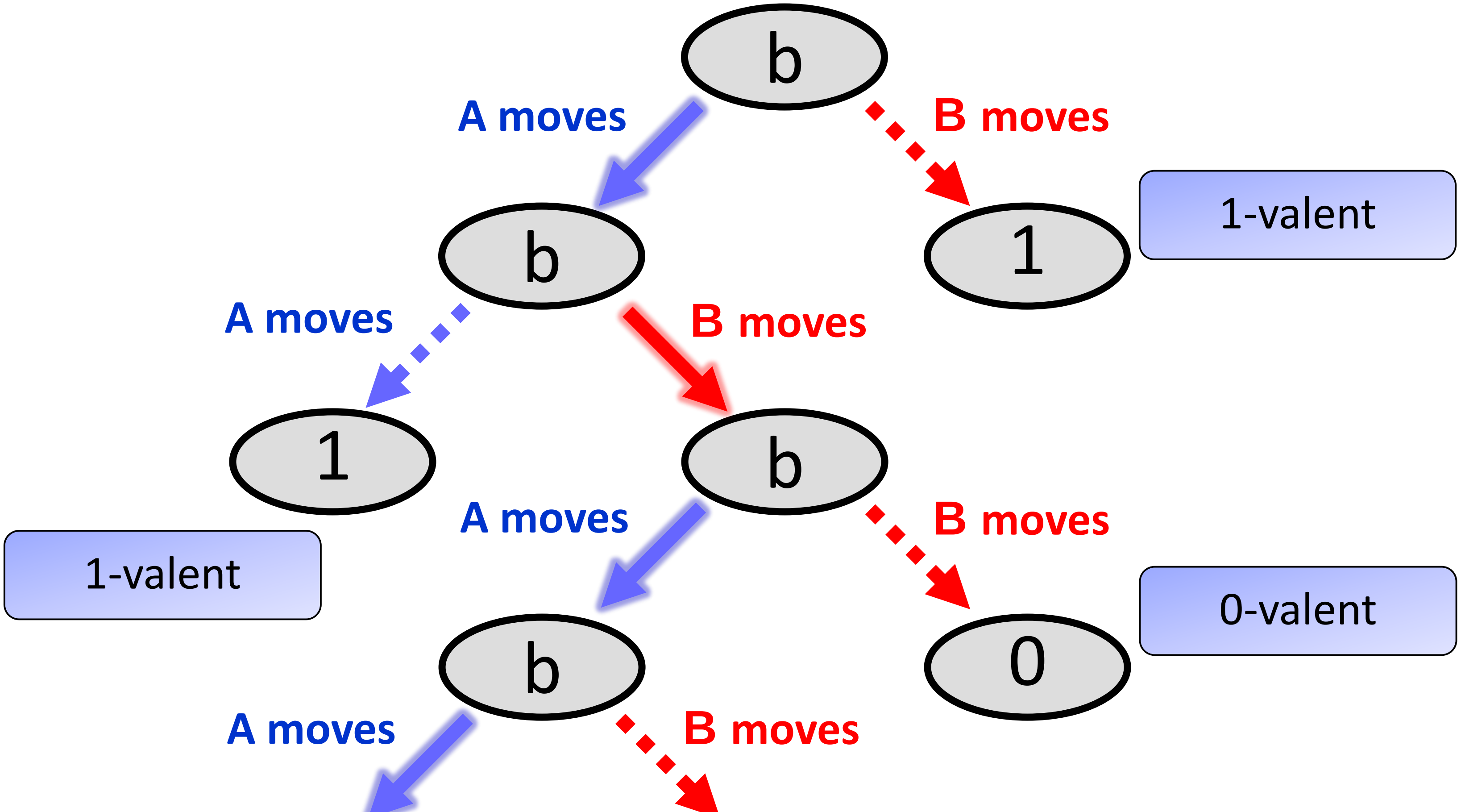
- Starting from a bivalent initial state
- The protocol must reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free
- The goal is now to show that the system can always remain bivalent

A bivalent state is critical if all children states are univalent



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

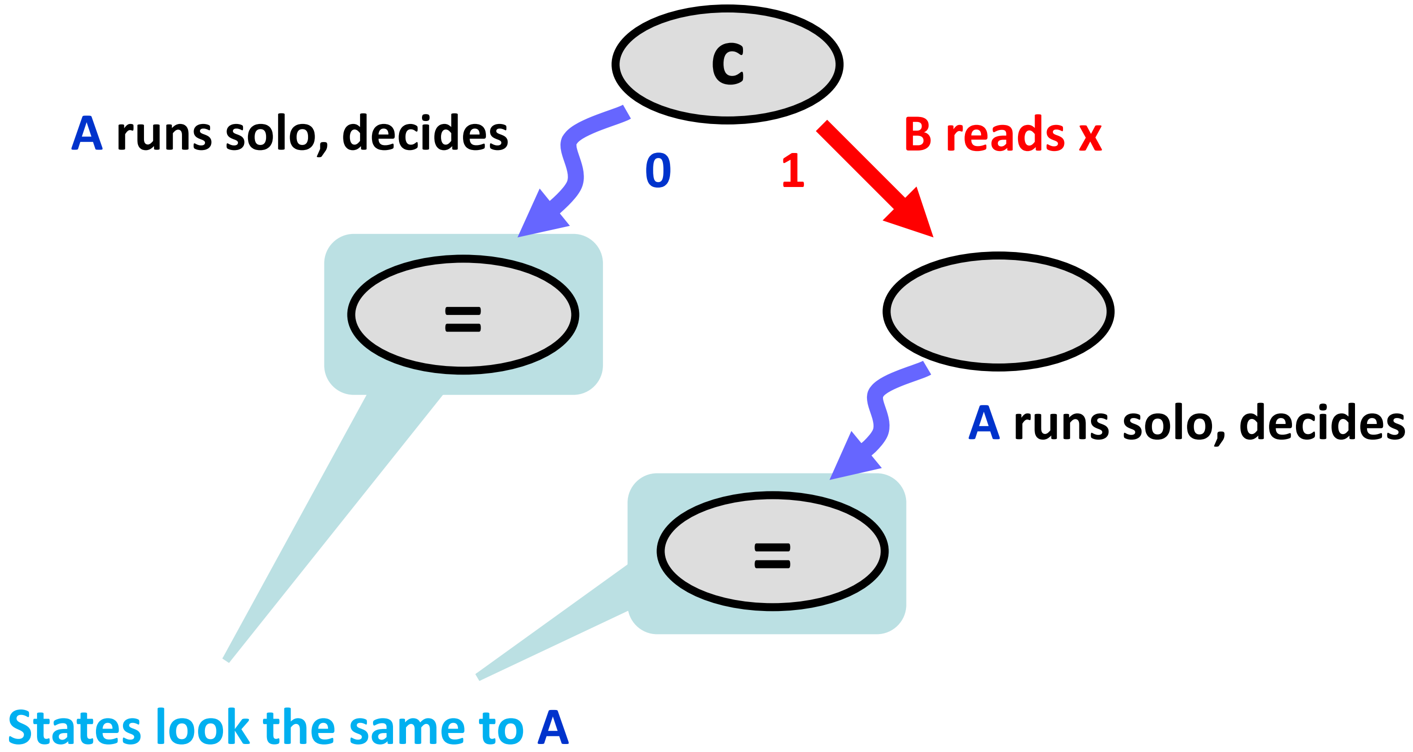
Possible Interactions

A reads x

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

B writes y

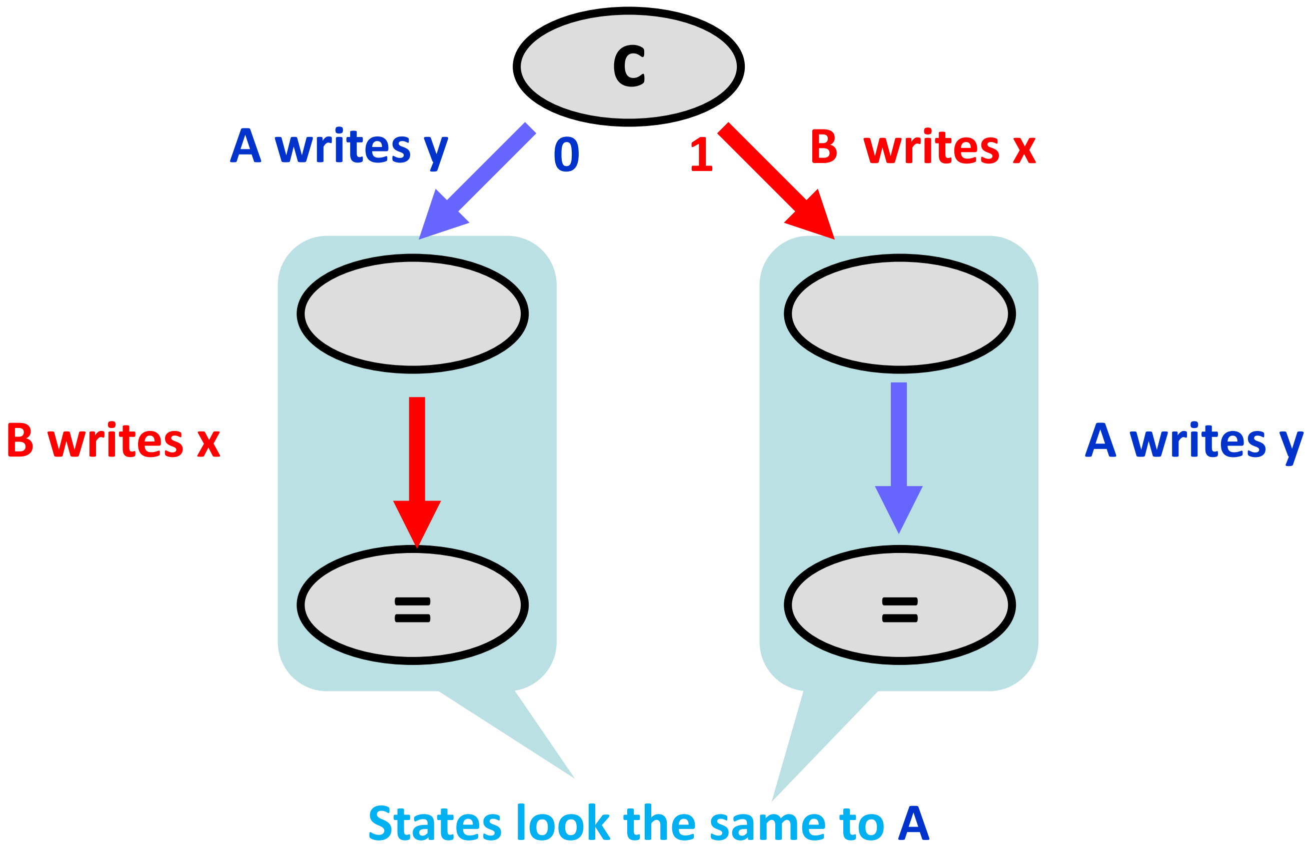
Reading Registers



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	?	?

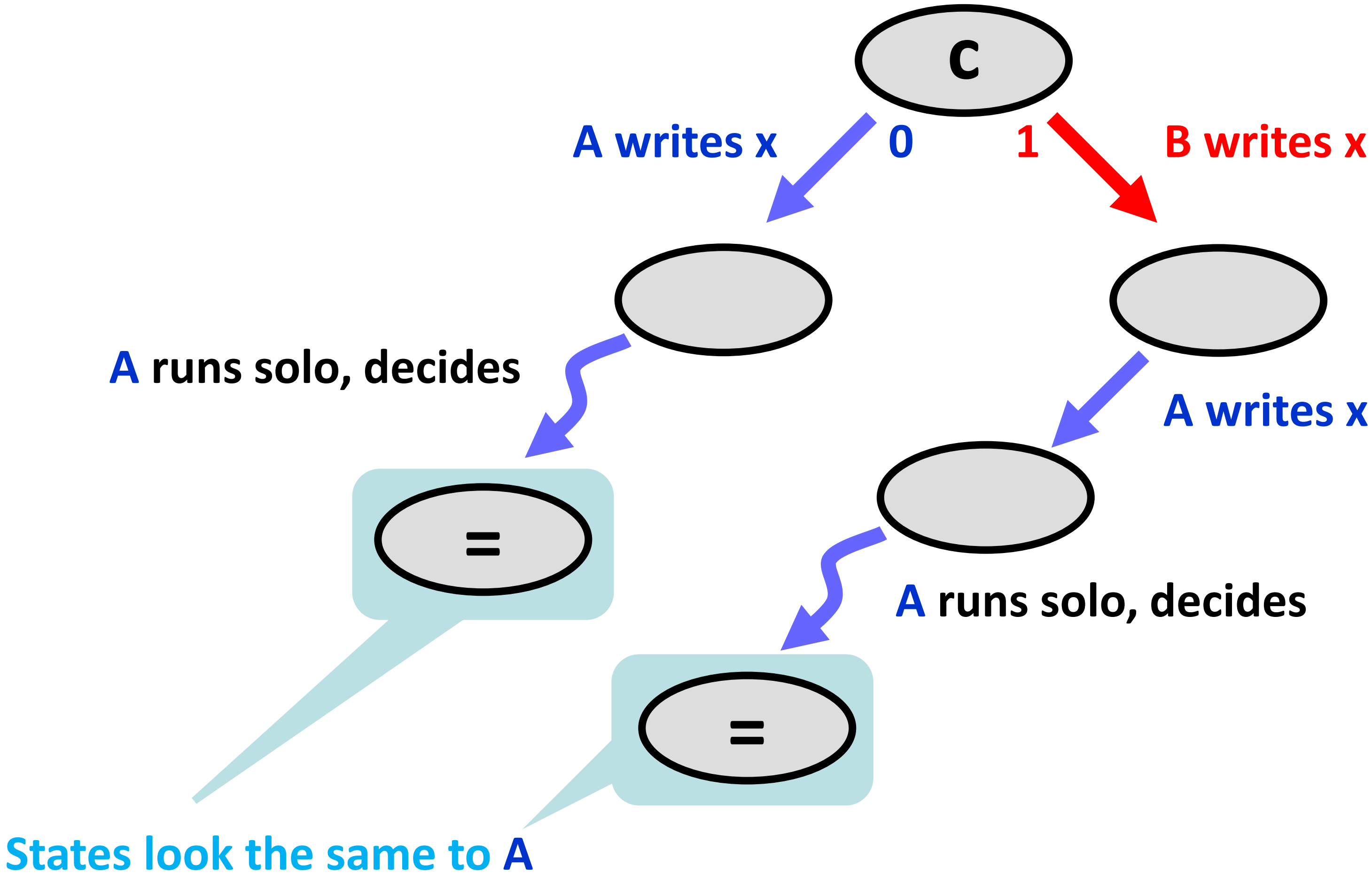
Writing Distinct Registers



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

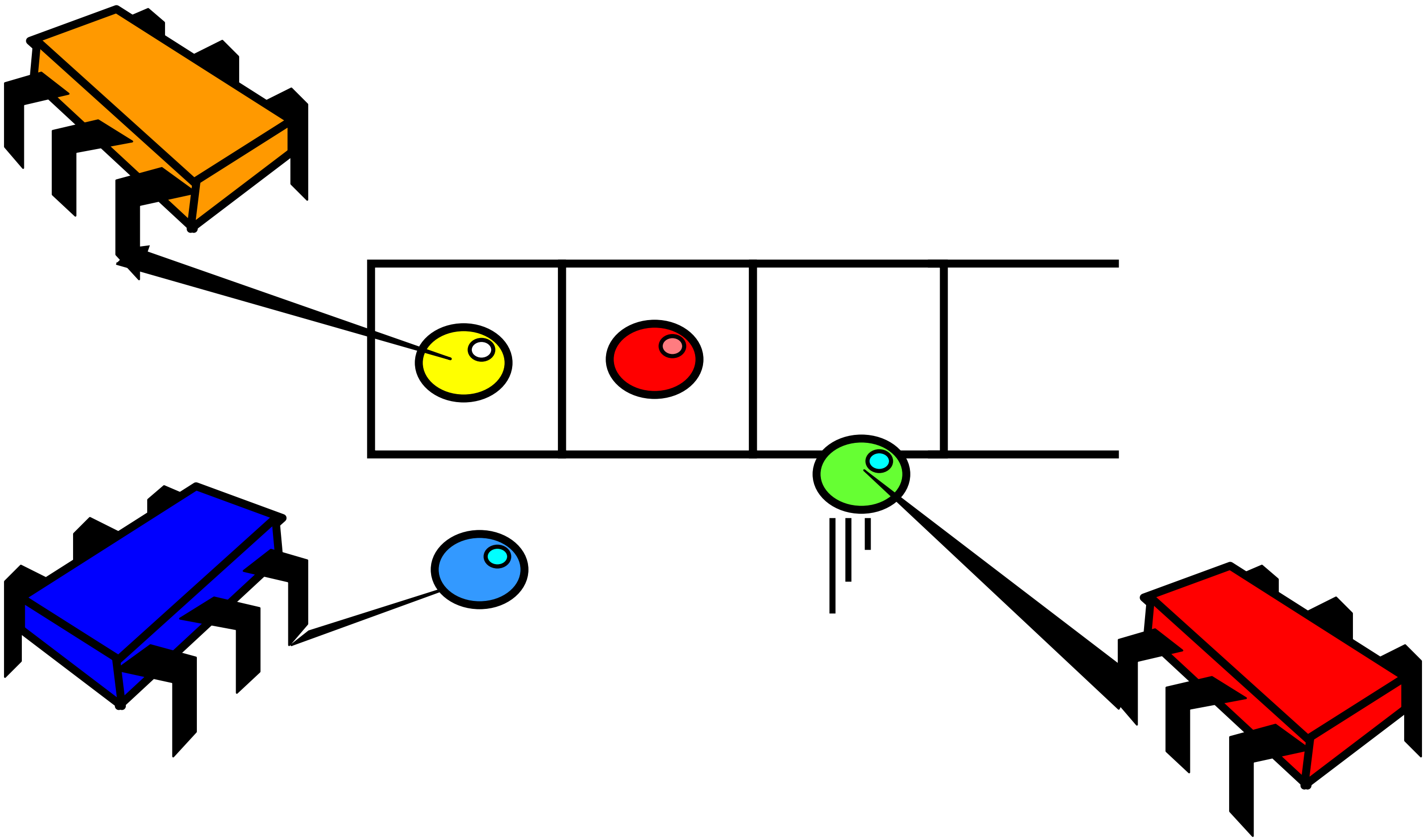


That's All, Folks!

	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

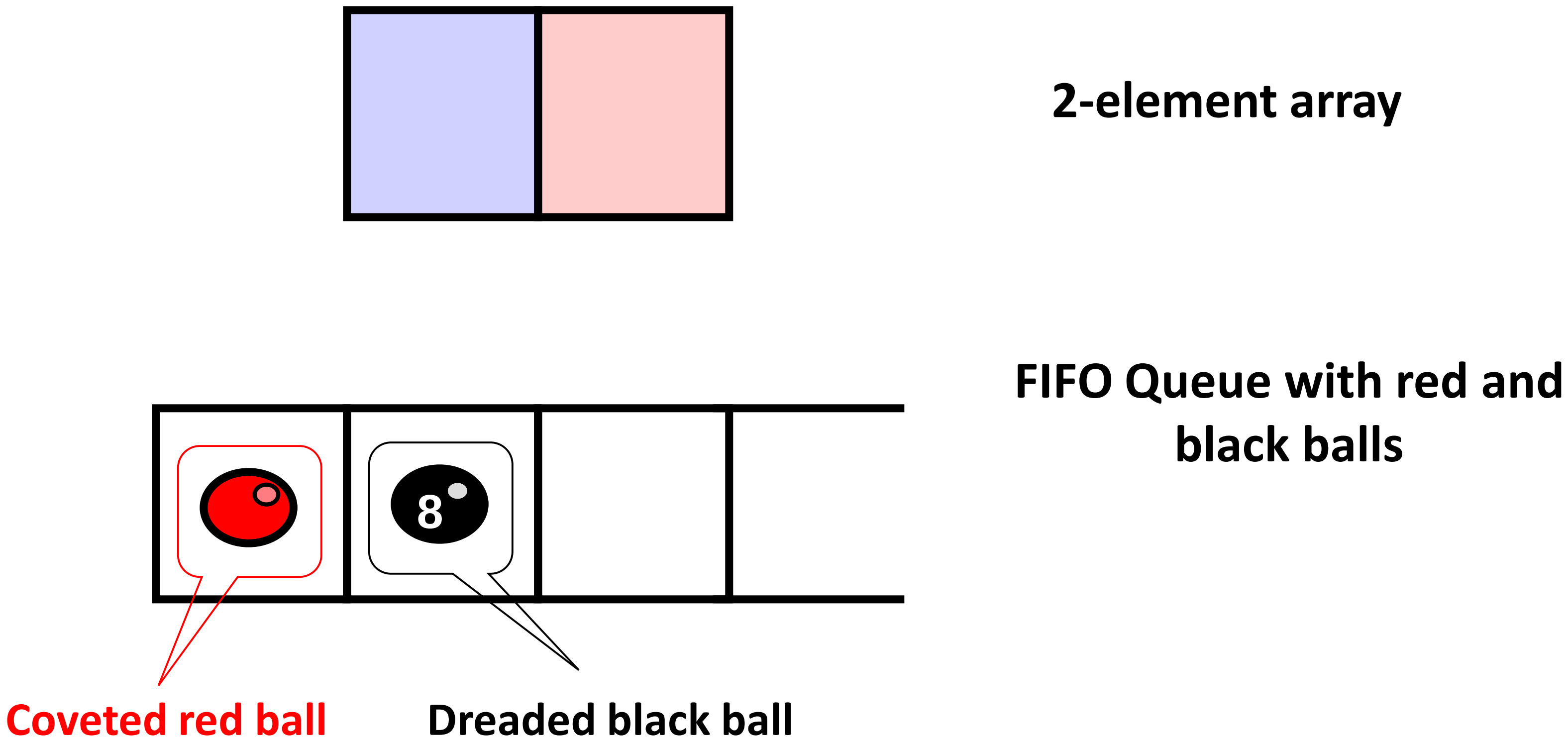
What Does Consensus Have to Do With Distributed Systems?

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



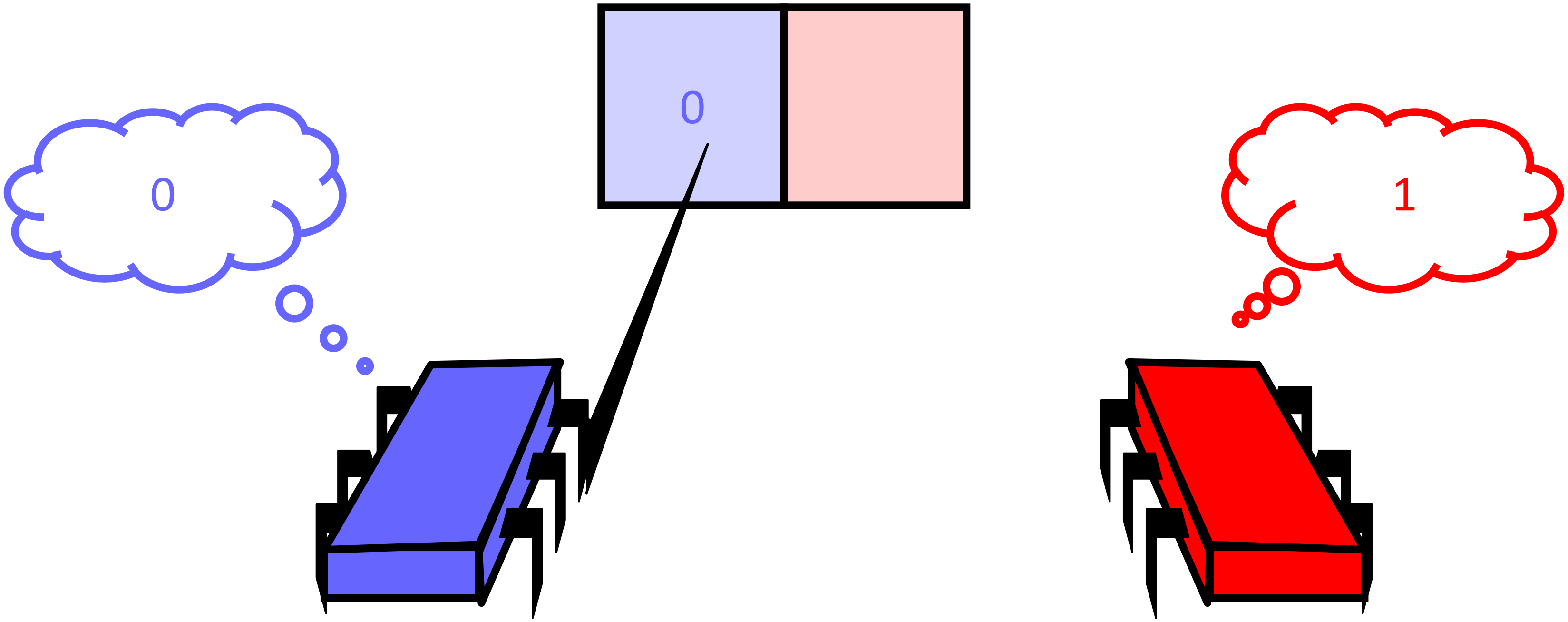
A Consensus Protocol

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



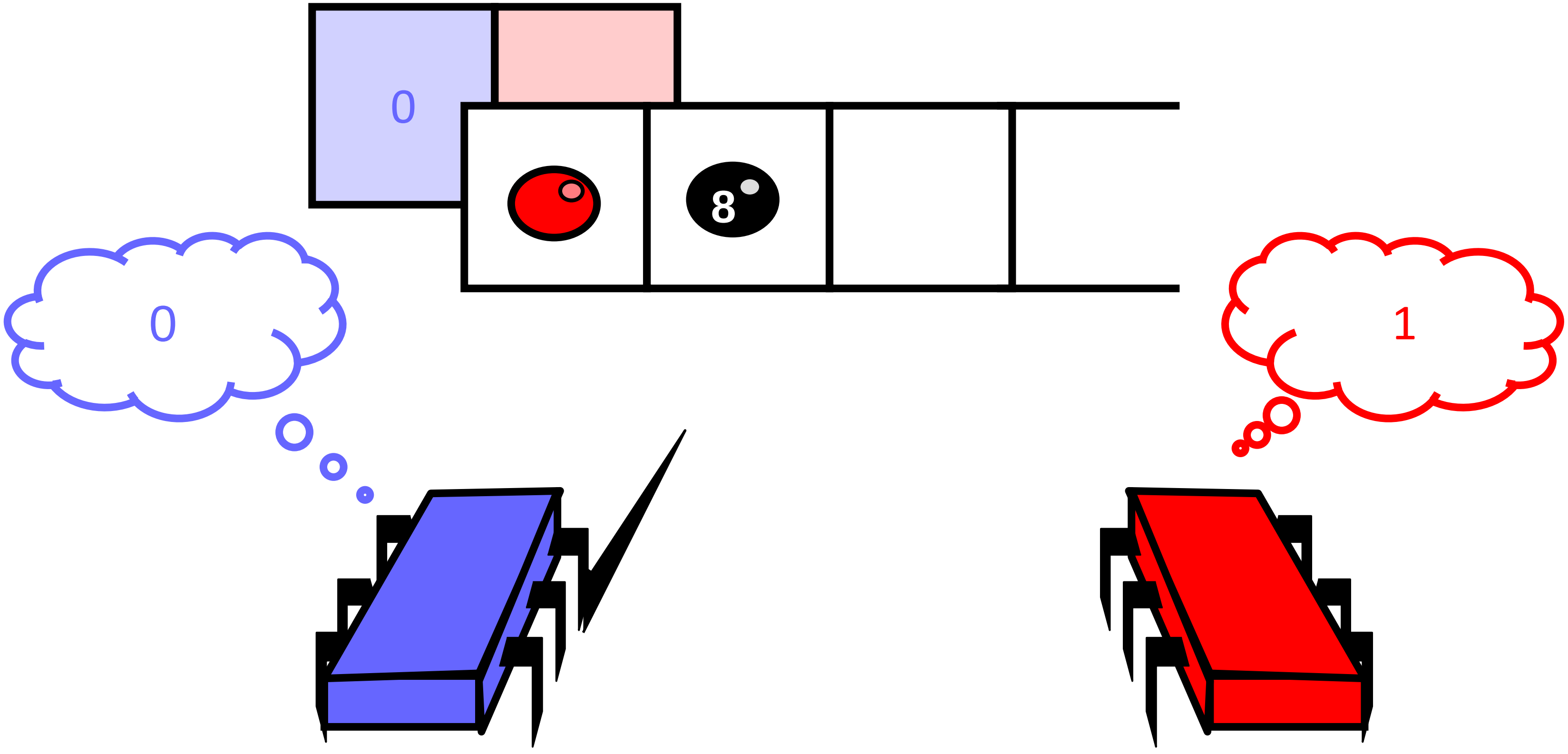
A Consensus Protocol

- Thread *i* writes its value into the array at position *i*

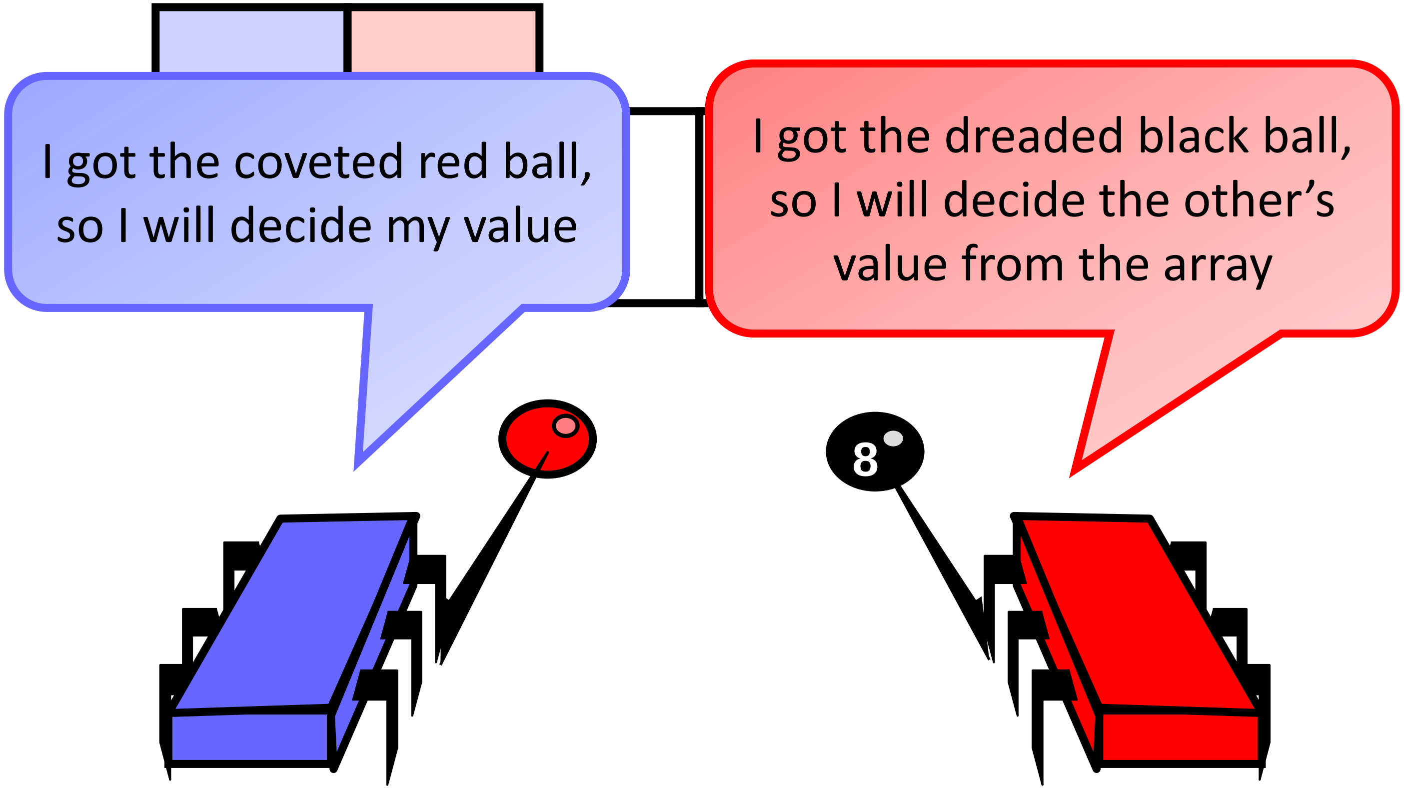


A Consensus Protocol

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



A Consensus Protocol



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 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…

Read-Modify-Write Shared Memory Consensus

- $n > 1$ processors
- 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*="?"
- Every processor *i* does the following

RMW(*c*)

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

atomic step

Discussion

- 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*-thread consensus, but not (*n*+1)-thread consensus
- Example: Atomic read/write registers have consensus number 1
  - Works with 1 process
  - We have shown impossibility with 2

Consensus Number Theorem

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-thread 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: $f_i(f_j(x))=f_i(x)$
- Claim: Any such set of RMW objects has consensus number **exactly 2**

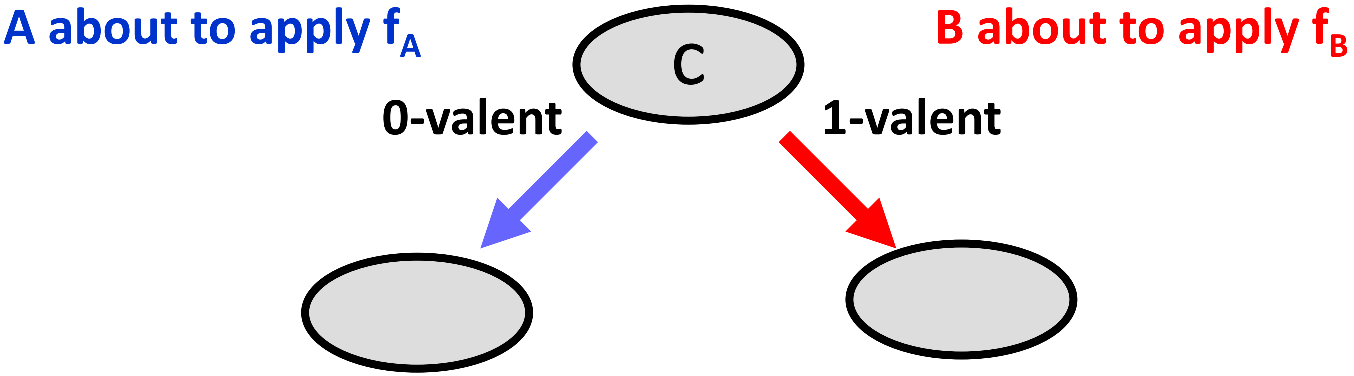
$f_i(x)$ = new value of cell
(not return value of f_i)

Examples:

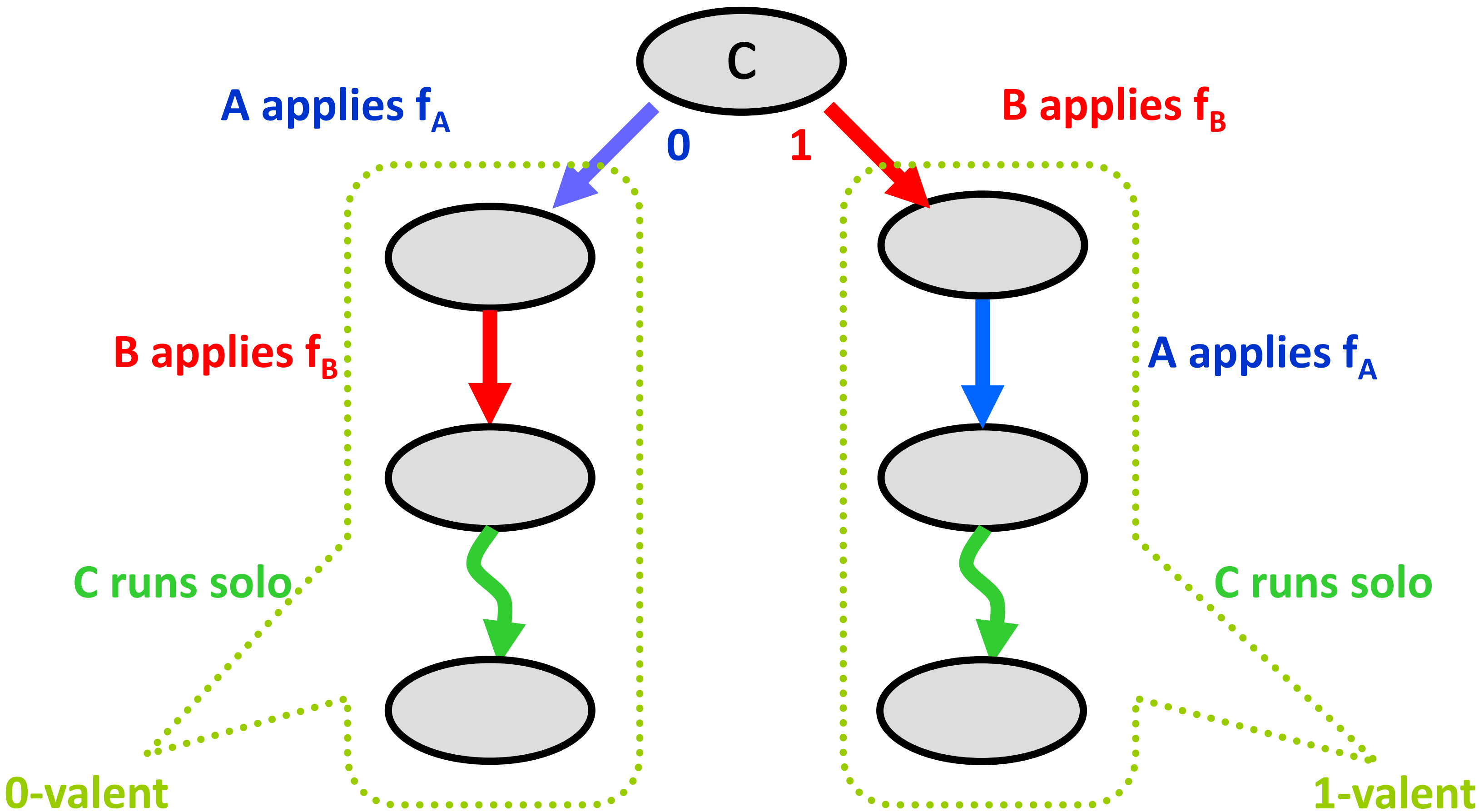
- Overwrite
  - Test&Set , Swap
- Commute
  - Fetch&Inc, Fetch&Add

Proof

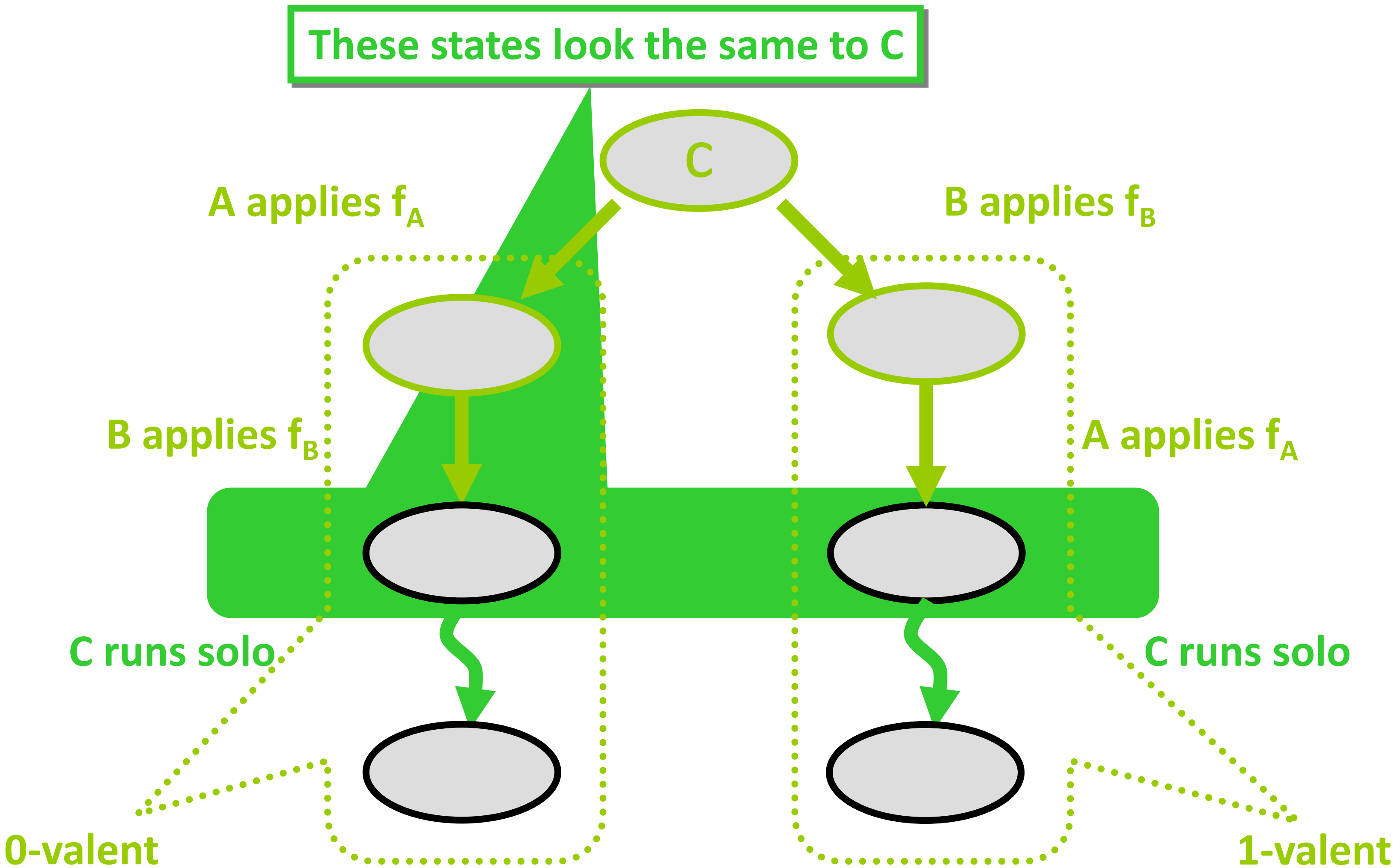
- There are three threads, A, B, and C
- Consider a critical state *c*:



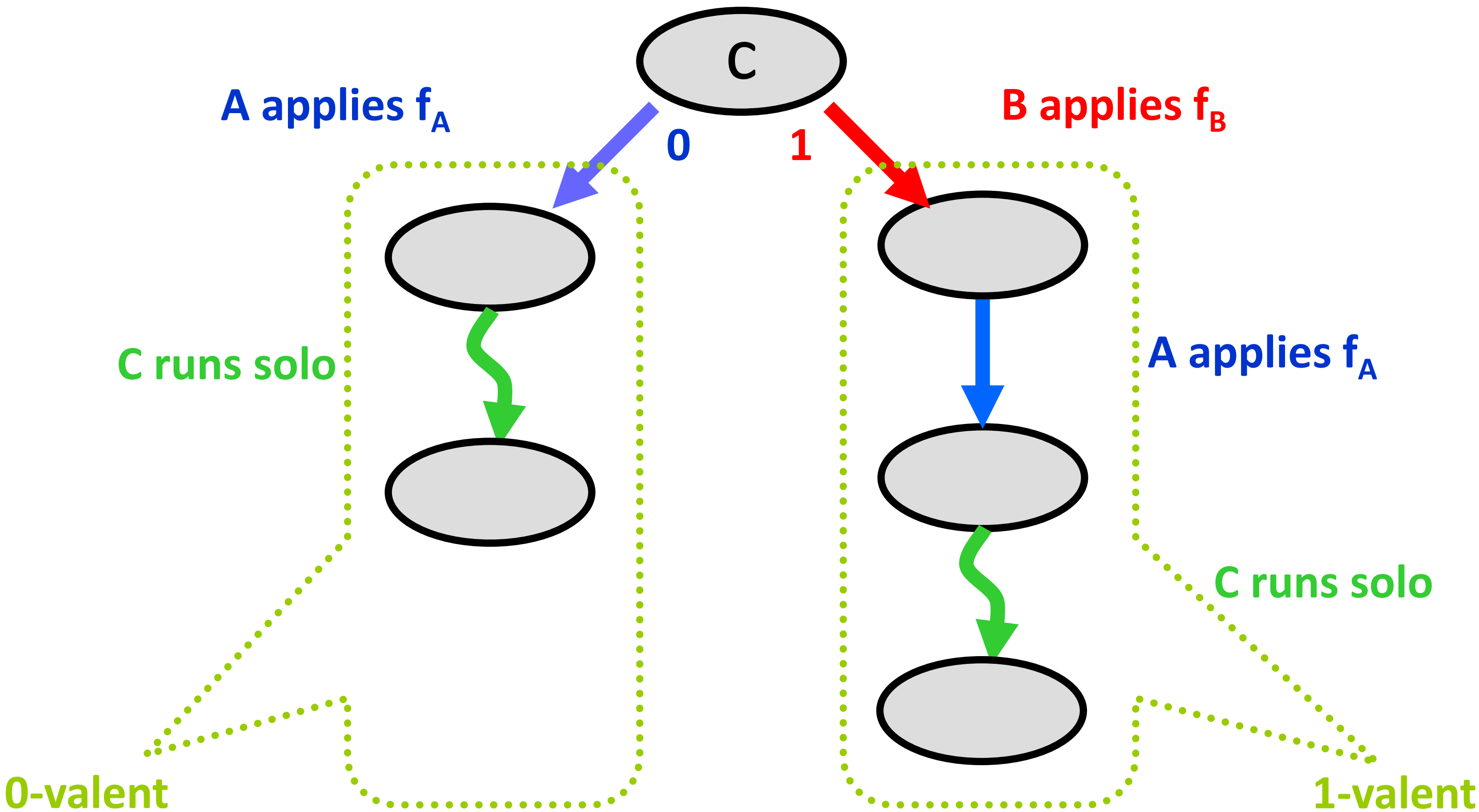
Proof: Maybe the Functions Commute



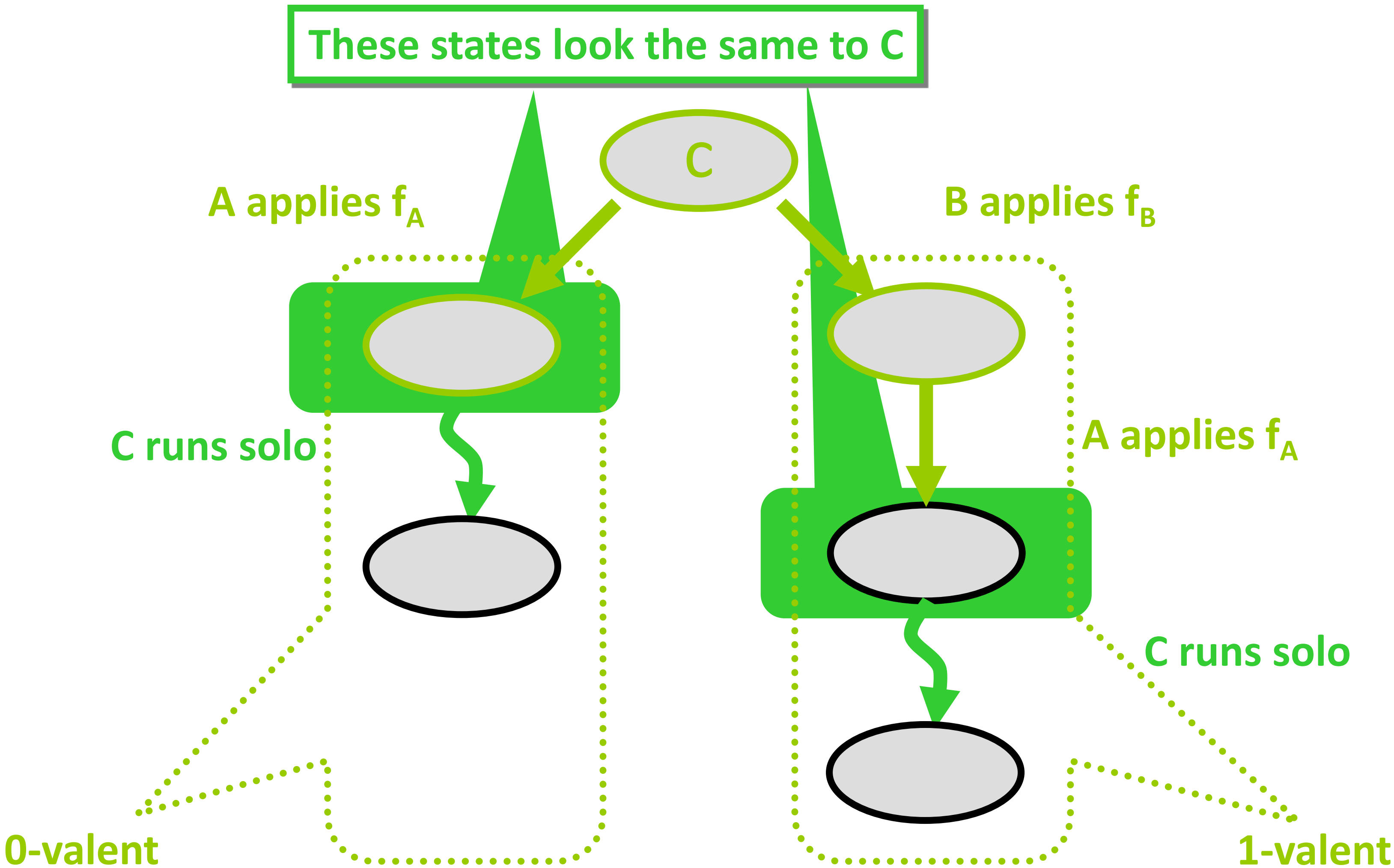
Proof: Maybe the Functions Commute



Proof: Maybe the Functions Overwrite



Proof: Maybe the Functions Overwrite



Impact

- 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

1	2	…	∞
• Read/Write Registers	• Test&Set • Fetch&Inc • Fetch&Add • Swap		• CAS • LL/SC

Credits

- The impossibility result is by Fischer, Lynch, Patterson, 1985
- The consensus hierarchy is by Herlihy, 1991

That’s all, folks!

Questions & Comments?

Roger Wattenhofer