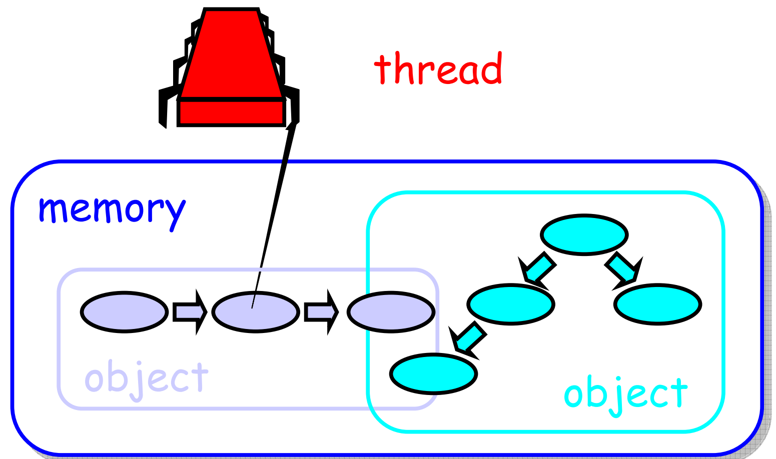
Chapter 6 CONSENSUS

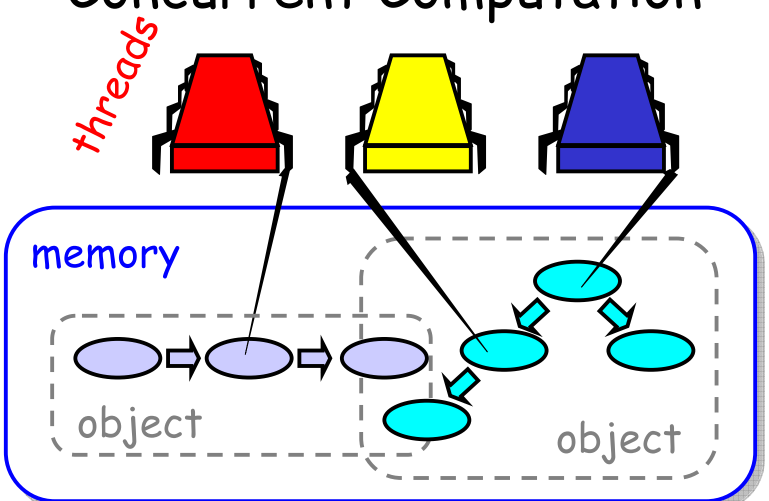
Distributed Computing Group

Computer Networks
Summer 2007

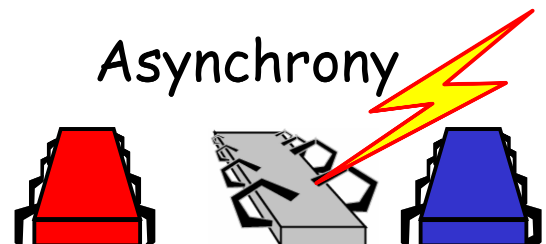
Sequential Computation



Concurrent Computation



Asynchrony

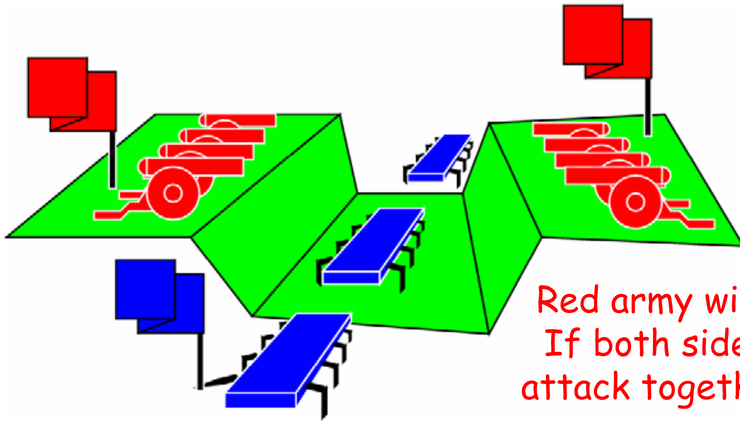


Sudden unpredictable delays
- Cache misses (*short*)
- Page faults (*long*)
- Scheduling quantum used up (*really long*)

Model Summary

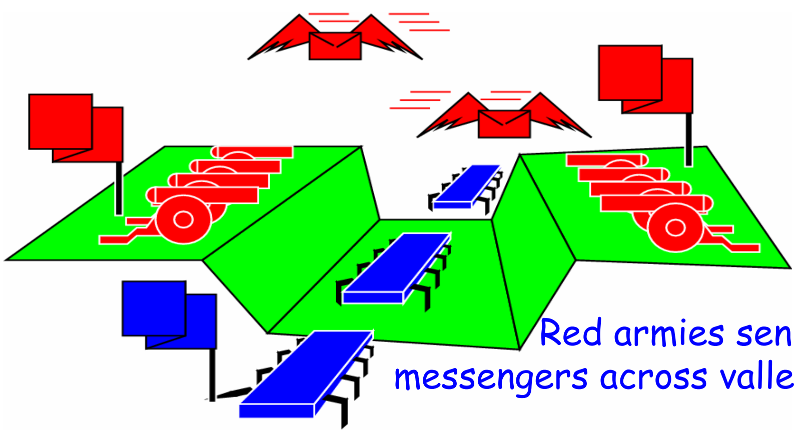
- *Multiple* *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays
- (Many similarities to message passing)

The Two Generals



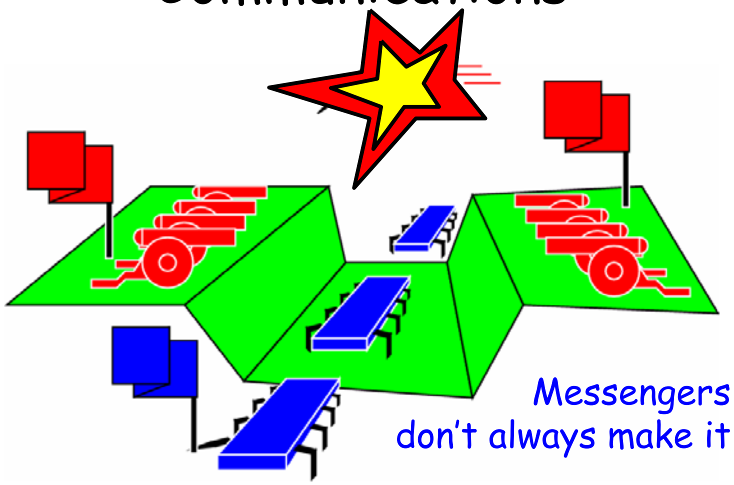
Red army wins
If both sides
attack together

Communications



Red armies send
messengers across valley

Communications



Messengers
don't always make it

Your Mission

Design a protocol to ensure that red armies attack simultaneously

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

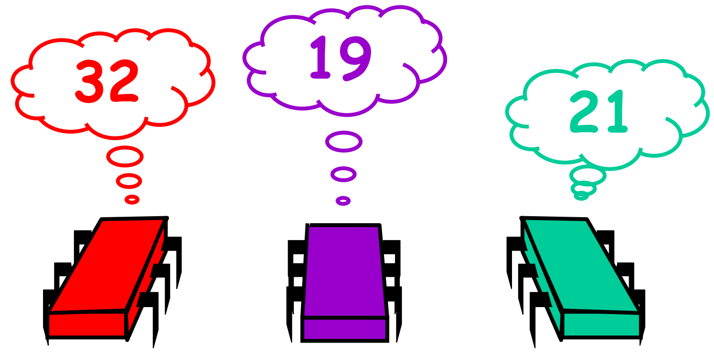
Proof

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

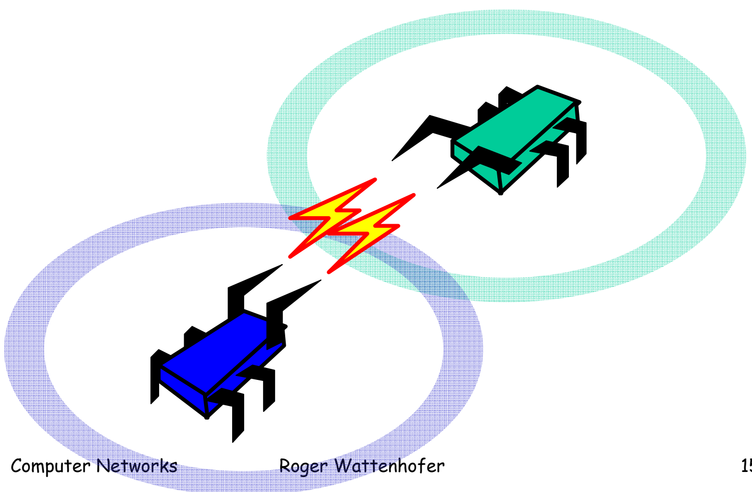
Fundamental Limitation

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

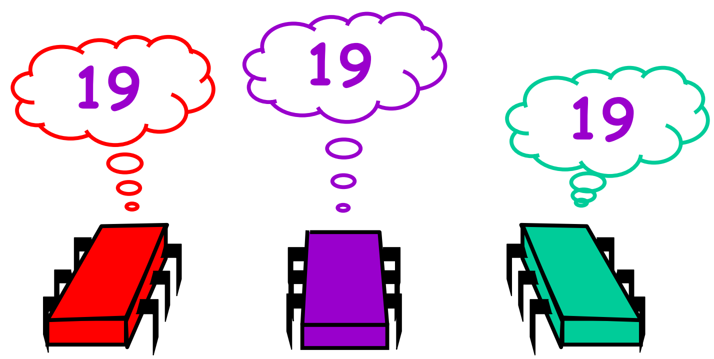
Consensus: Each Thread has a Private Input



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!*

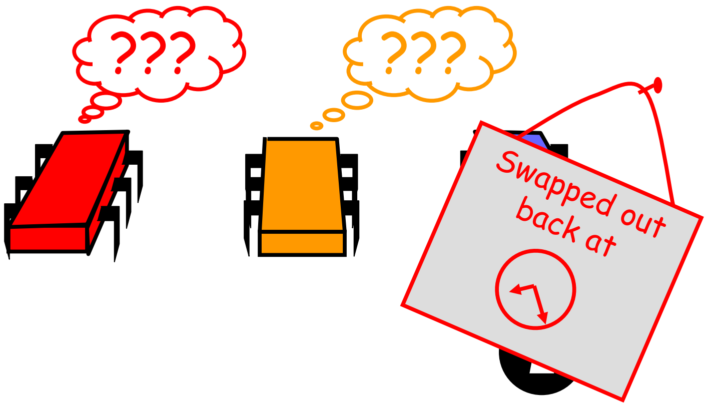
Consensus #1 shared memory

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

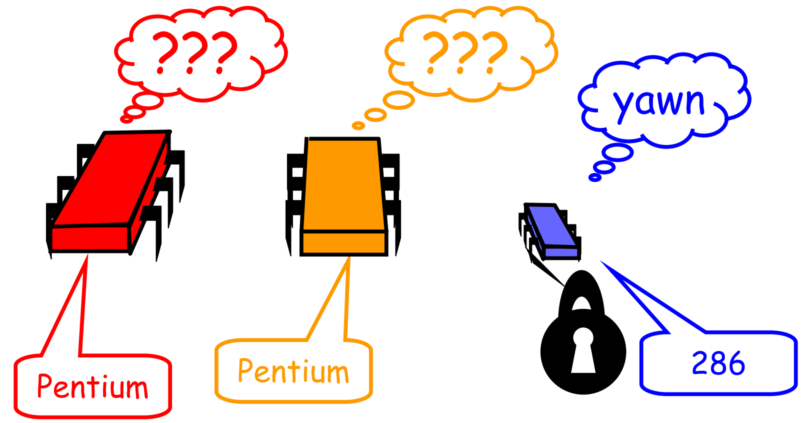
Protocol (Algorithm?)

- 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.

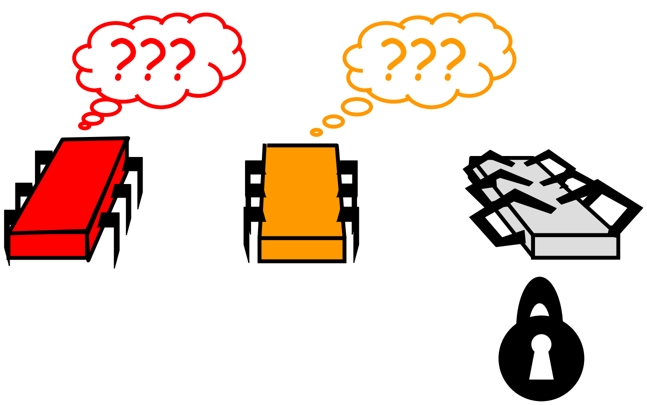
Unexpected Delay



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

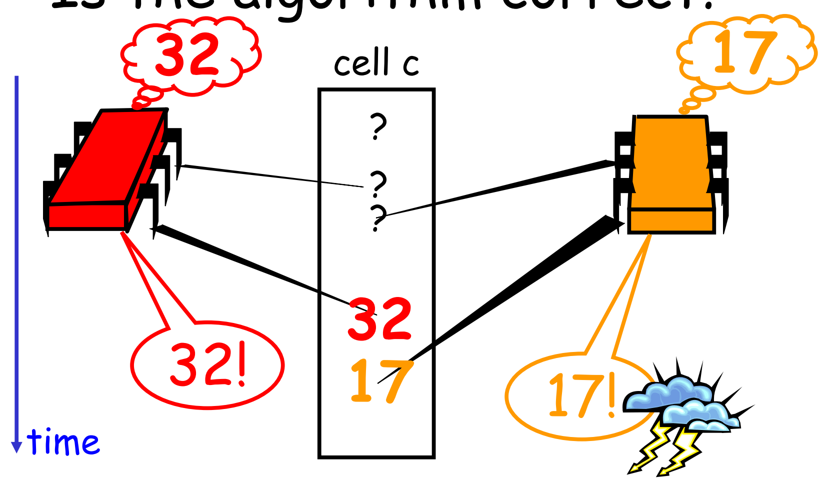
- Every process (method call) completes in a finite number of steps
- Implies no mutual exclusion
- We assume that we have wait-free atomic registers (that is, reads and writes to same register do not overlap)

A wait-free algorithm…

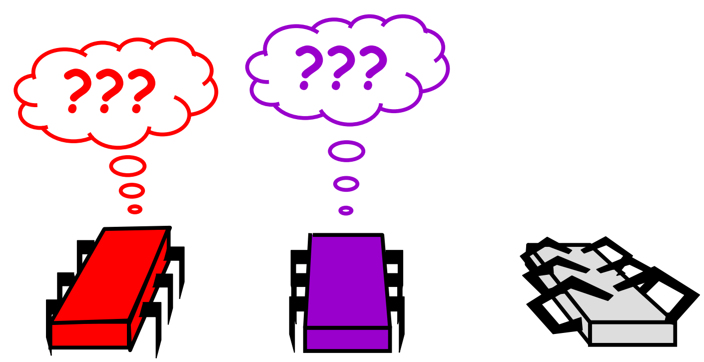
- There is a cell c, initially c="?"
- Every processor i does the following

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


Is the algorithm correct?



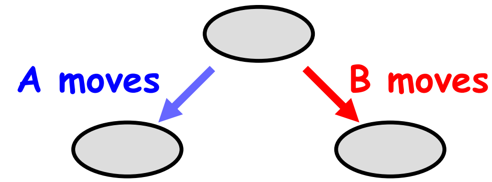
Theorem:
No wait-free consensus



Proof Strategy

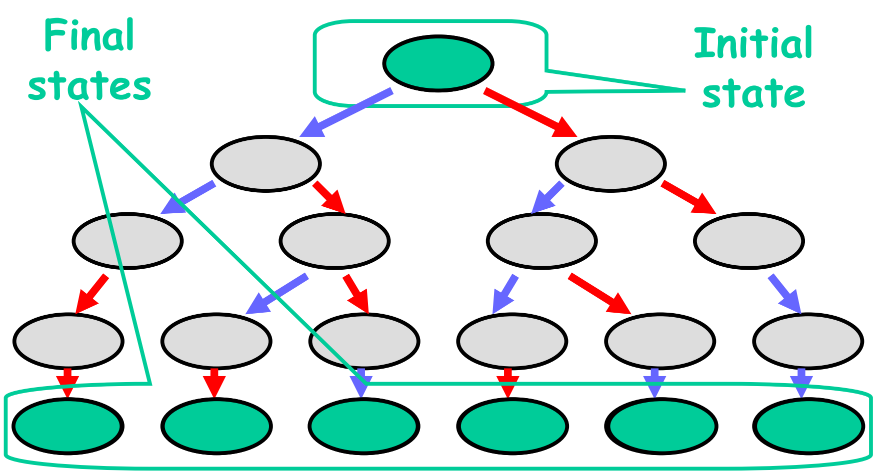
- Make it simple
  - n = 2, binary input
- Assume that there is a protocol
- Reason about the properties of any such protocol
- Derive a contradiction

Wait-Free Computation

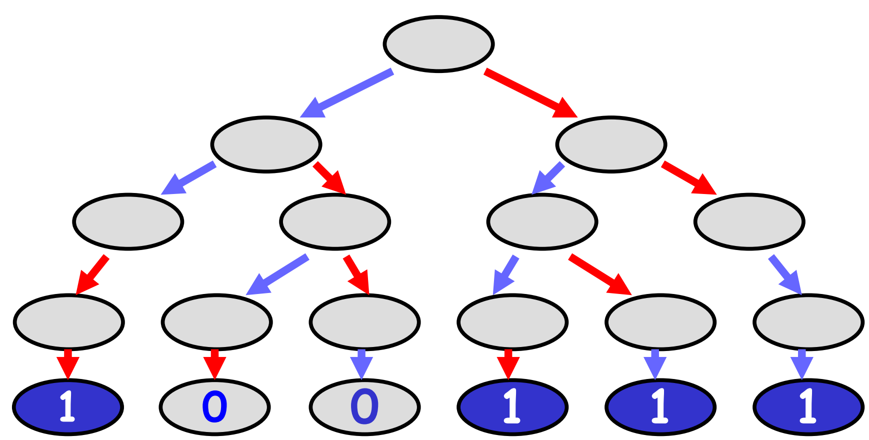


- Either A or B "moves"
- Moving means
  - Register read
  - Register write

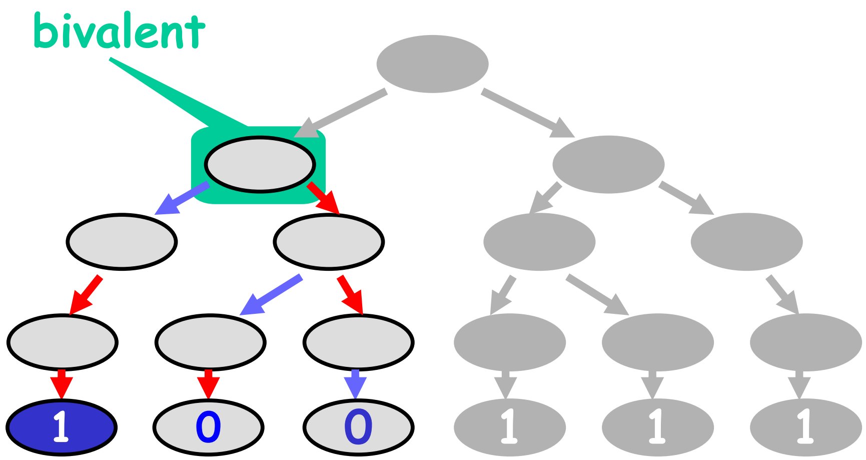
The Two-Move Tree



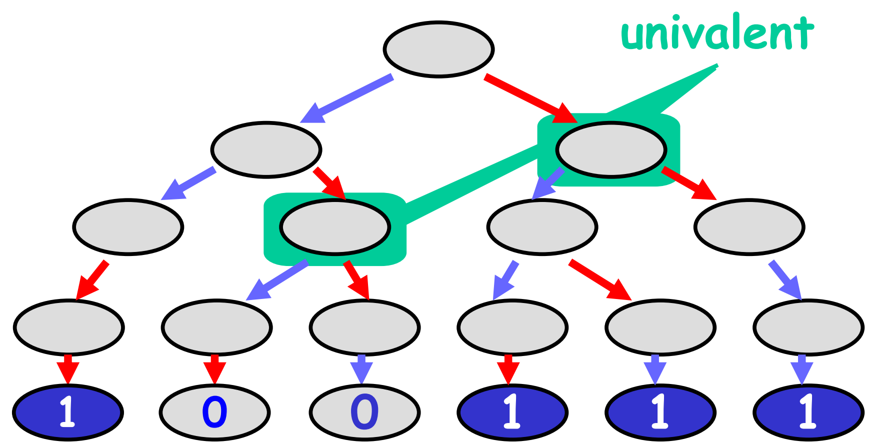
Decision Values



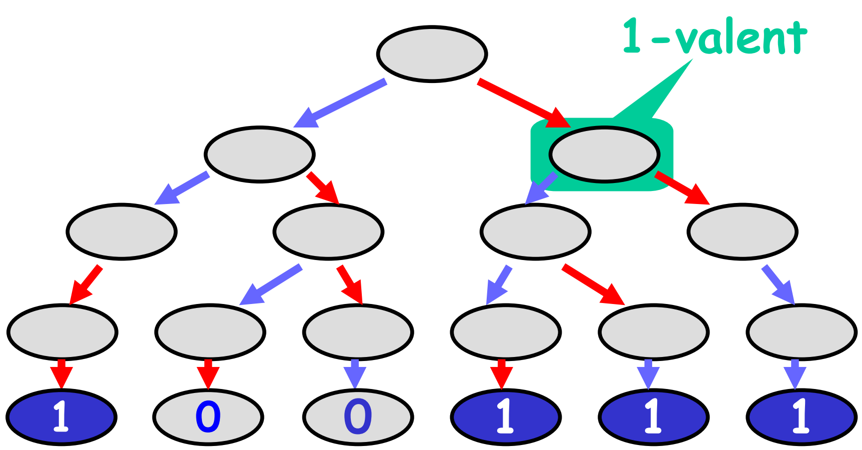
Bivalent: Both Possible



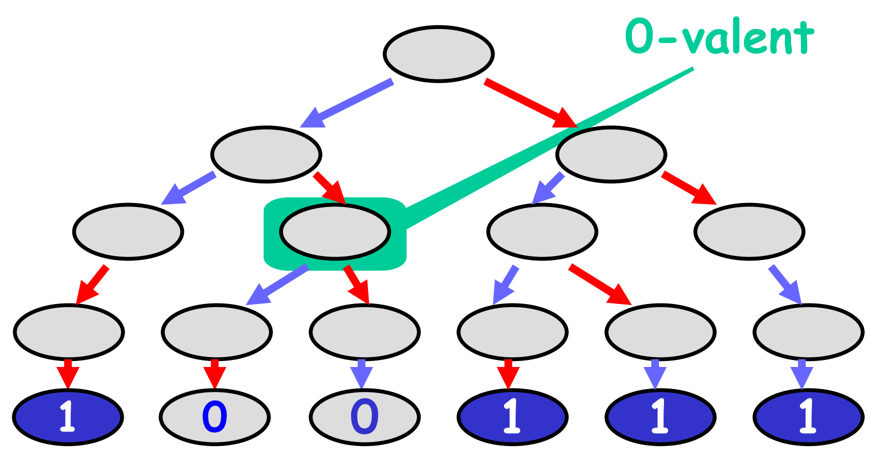
Univalent: Single Value Possible



1-valent: Only 1 Possible



0-valent: Only 0 possible



Summary

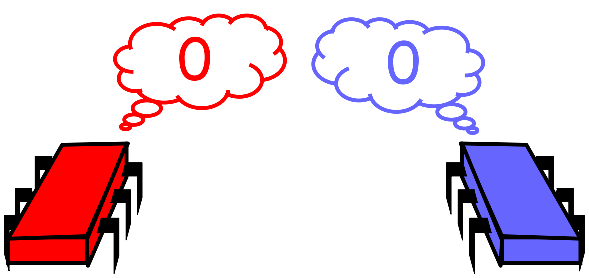
- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed
- Univalent states
  - Outcome is fixed
  - May not be "known" yet
  - 1-Valent and 0-Valent states

Claim

Some initial system state is bivalent

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

A 0-Valent Initial State

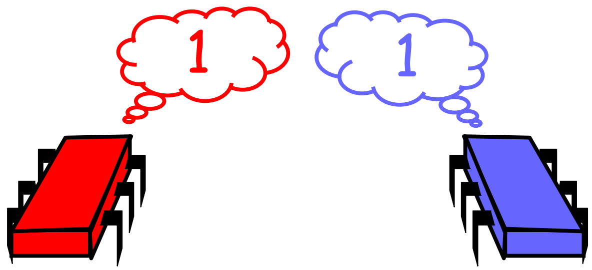


- All executions lead to decision of 0

A 0-Valent Initial State

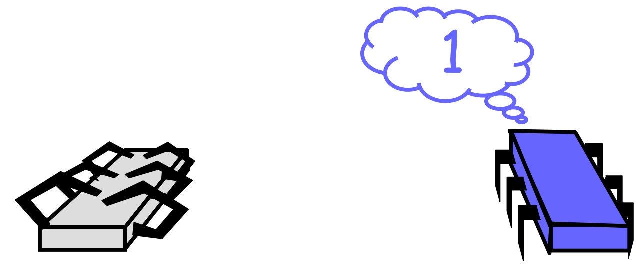
- Solo execution by A also decides 0

A 1-Valent Initial State



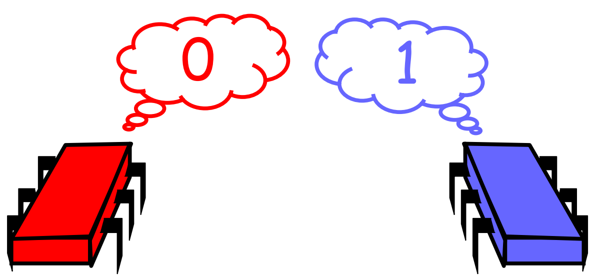
- All executions lead to decision of 1

A 1-Valent Initial State



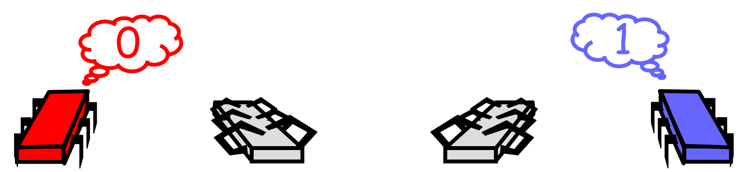
- Solo execution by B also decides 1

A Univalent Initial State?



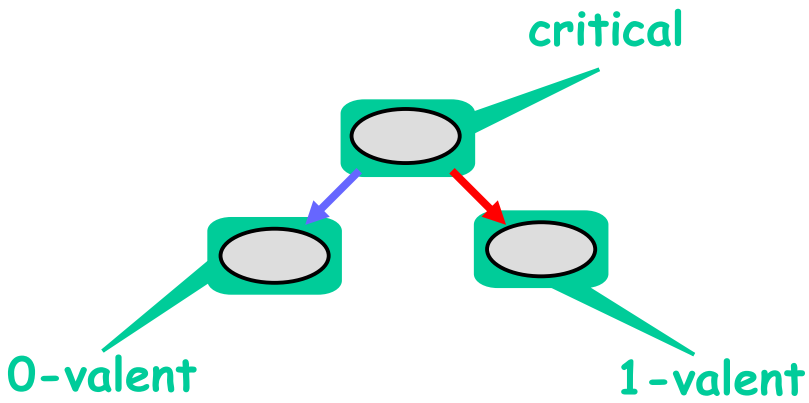
- Can all executions lead to the same decision?

State is Bivalent



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

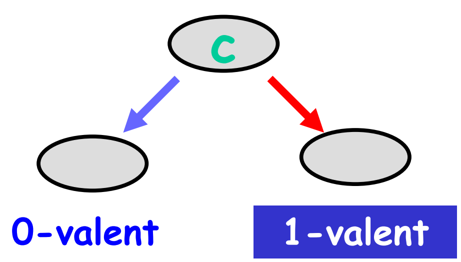
Critical States



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

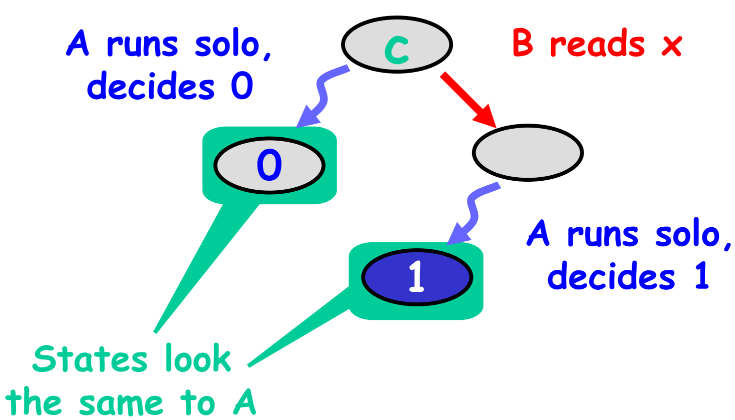
What are the Threads Doing?

- 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()	?	?	?	?

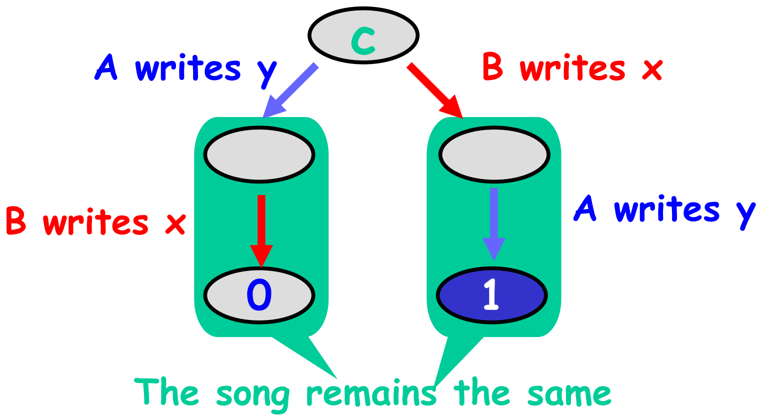
Reading Registers



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	?	?

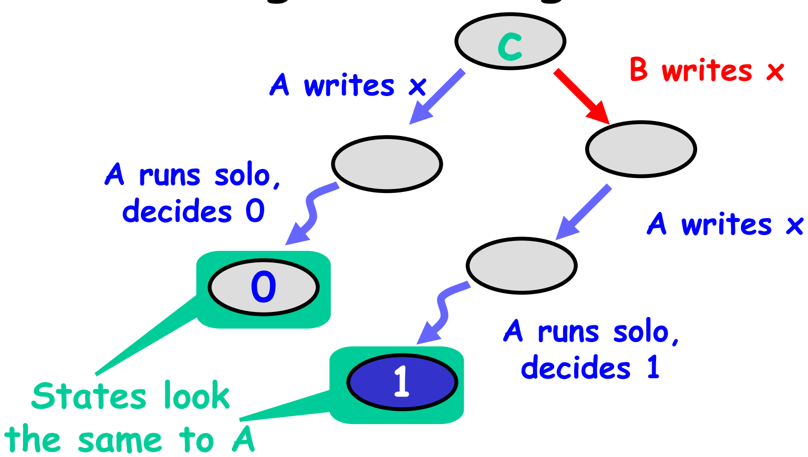
Writing Distinct Registers



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



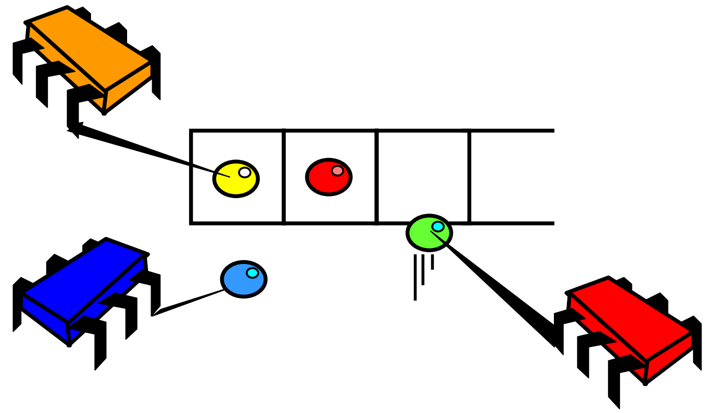
That's All, Folks!

	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

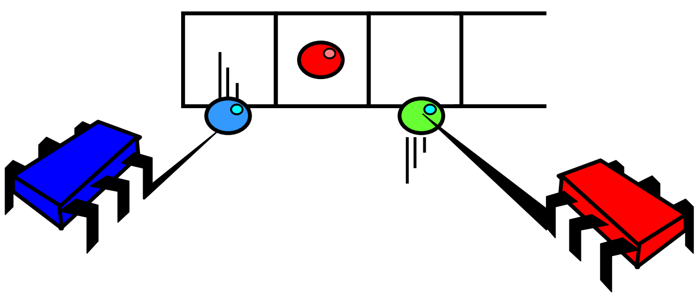
Theorem

- 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

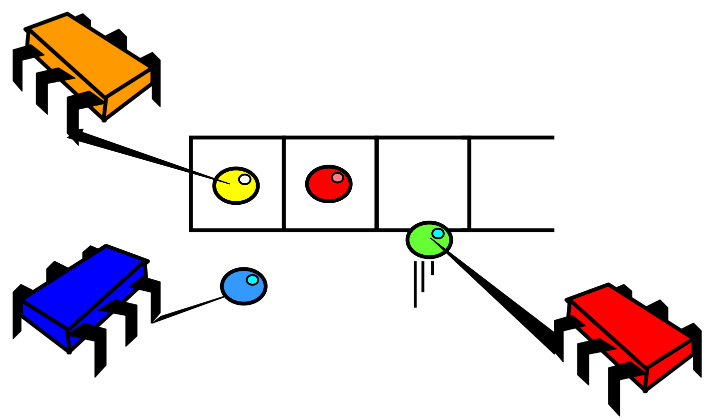
What Does Consensus have to do with Distributed Systems?



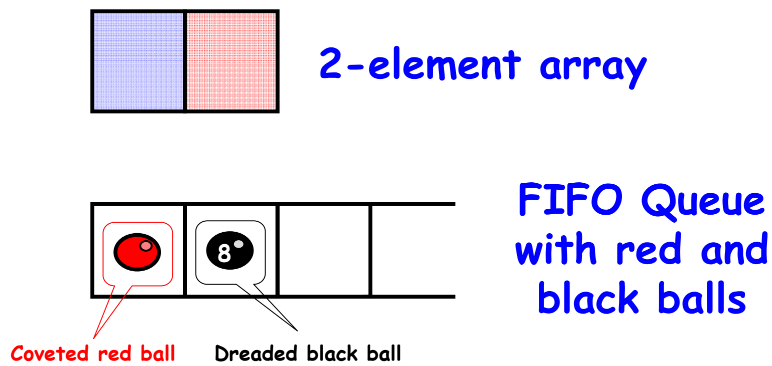
We want to build a Concurrent FIFO Queue



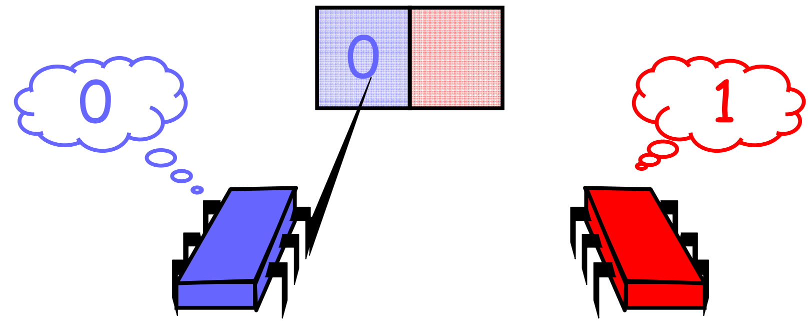
With Multiple Dequeuers!



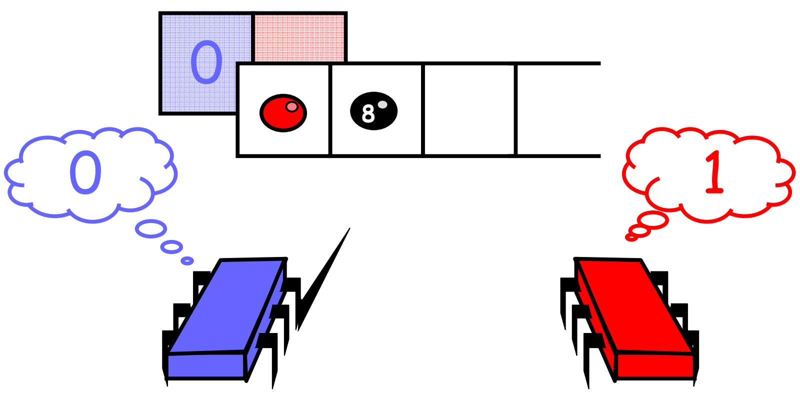
A Consensus Protocol



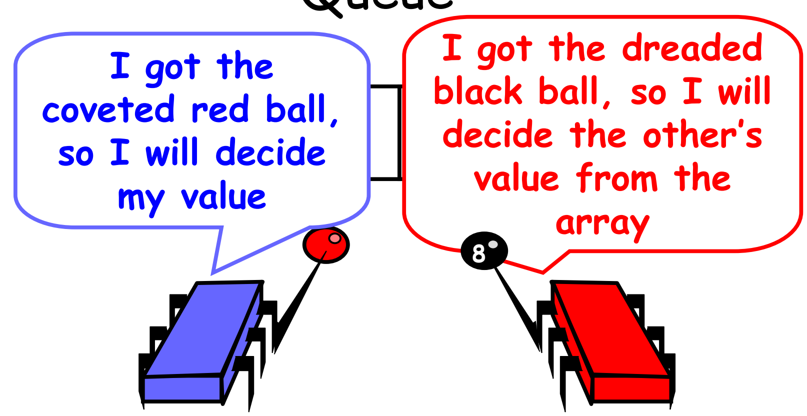
Protocol: Write Value to Array



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, vi); decide vi;
    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)**

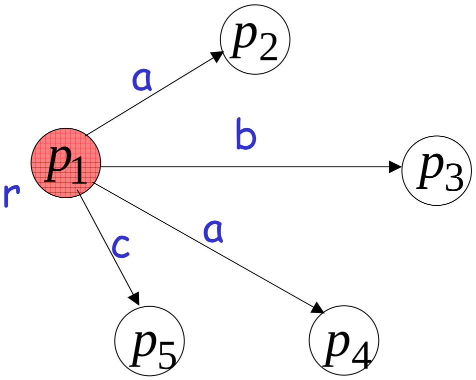
Consensus #4
Synchronous Systems

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

- Q: Can one solve consensus at least in synchronous systems with f failures?
- A: Yes, but f+1 rounds needed

Consensus #5
Byzantine Failures

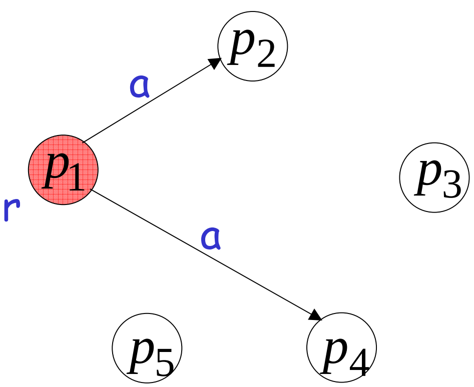
Faulty processor



Different processes receive different values

Some messages may be lost

Faulty processor



A Byzantine process can behave like a Crashed-failed process

Consensus with Byzantine Failures

f-resilient consensus algorithm:

solves consensus for f failed processes

Q: Is this possible?

A: Yes, but 3f+1 processes needed!

Atomic Broadcast

- One process wants to broadcast message to all other processes
- Either everybody should receive the (same) message, or nobody should receive the message
- Closely related to Consensus: First send the message to all, then agree!

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.

Summary

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