New Technologies in Distributed Computing

Igor Zablotchi

Based on joint work with
Marcos Aguilera, Naama Ben-David, Nachshon Cohen, Tudor David,
Aleksandar Dragojevic, Rachid Guerraoui, Virendra Marathe

[Some slides courtesy of Naama Ben-David and Tudor David]

Introduction

- So far: “traditional” concurrent objects
  - Registers
  - CAS
  - etc.
- Studied for decades & understood well

Introduction

- New technologies are constantly being developed
- They come with opportunities, but also with challenges
- In this lecture, two new technologies
  - RDMA
  - Persistent Memory
- Both topics of ongoing research

Part 1

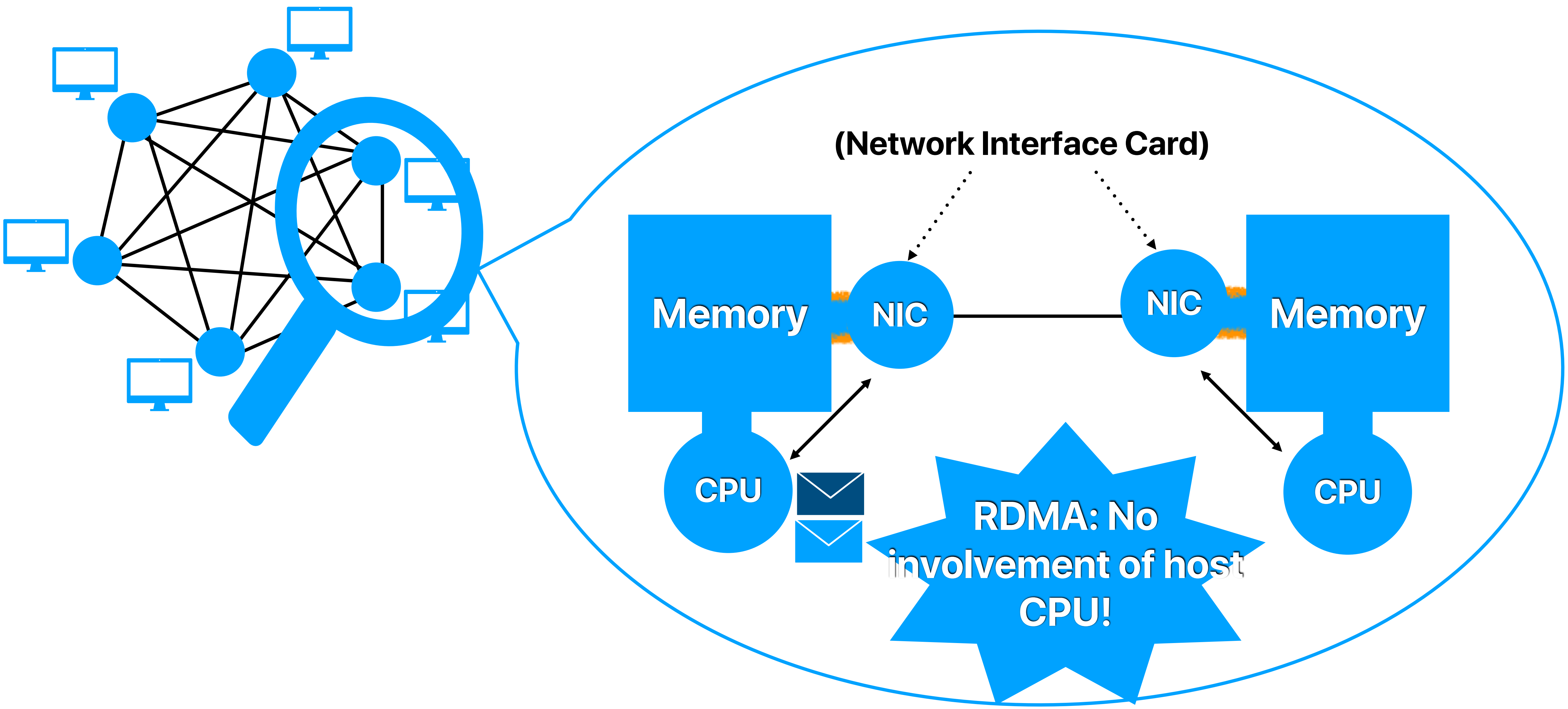
RDMA

Outline

- What is RDMA?
- How we model RDMA
- Notable Results: consensus with RDMA
  - Crash faults
  - Byzantine faults

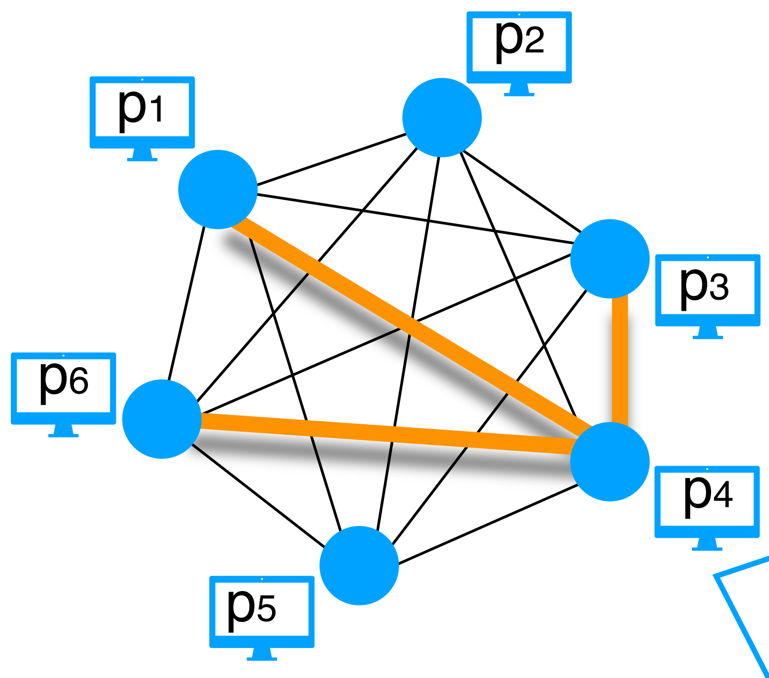
What is RDMA?

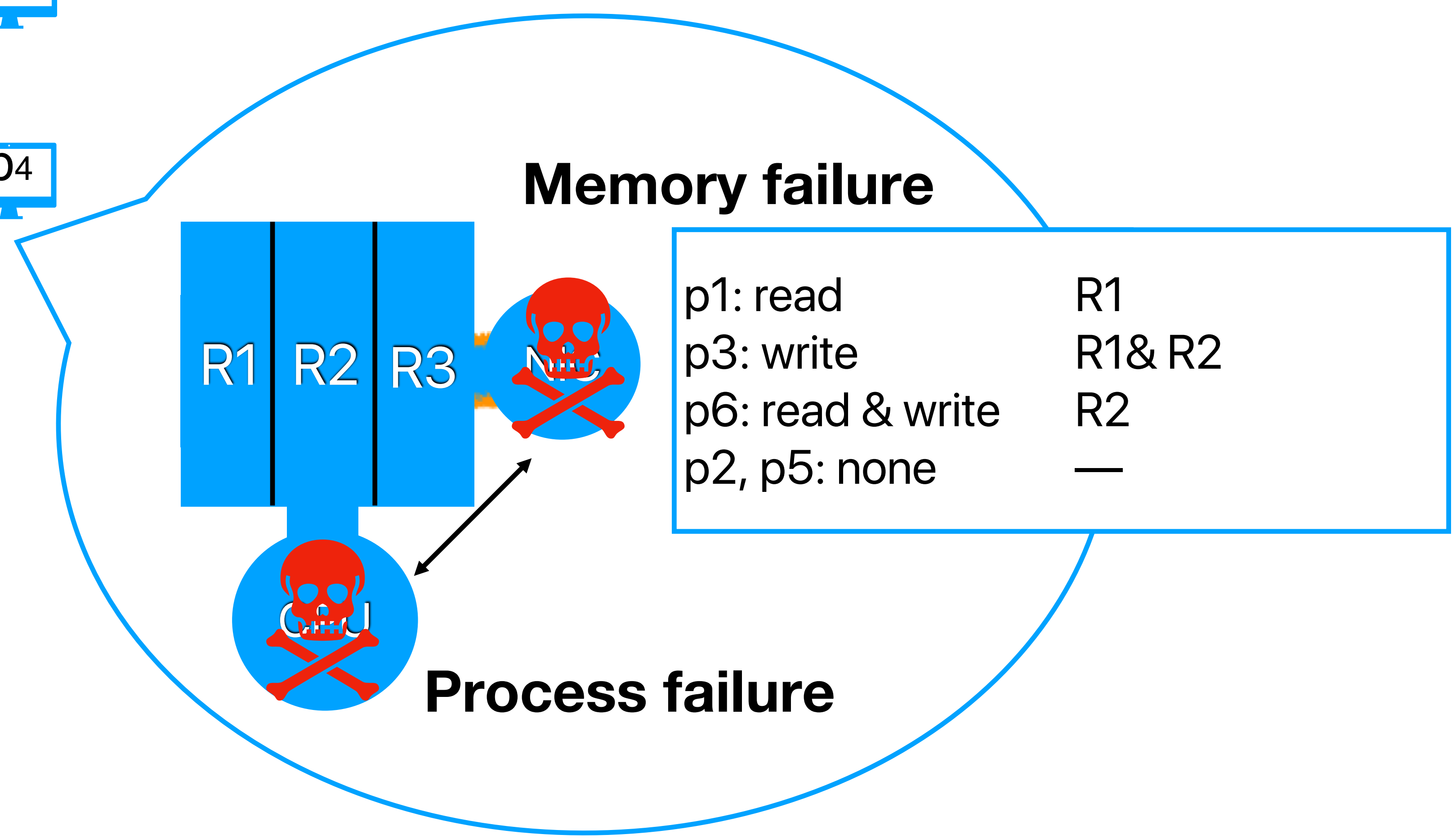
Remote Direct Memory Access (RDMA)



What is RDMA?

- Can choose RDMA connections and permissions
- Can give different permissions for different memory regions

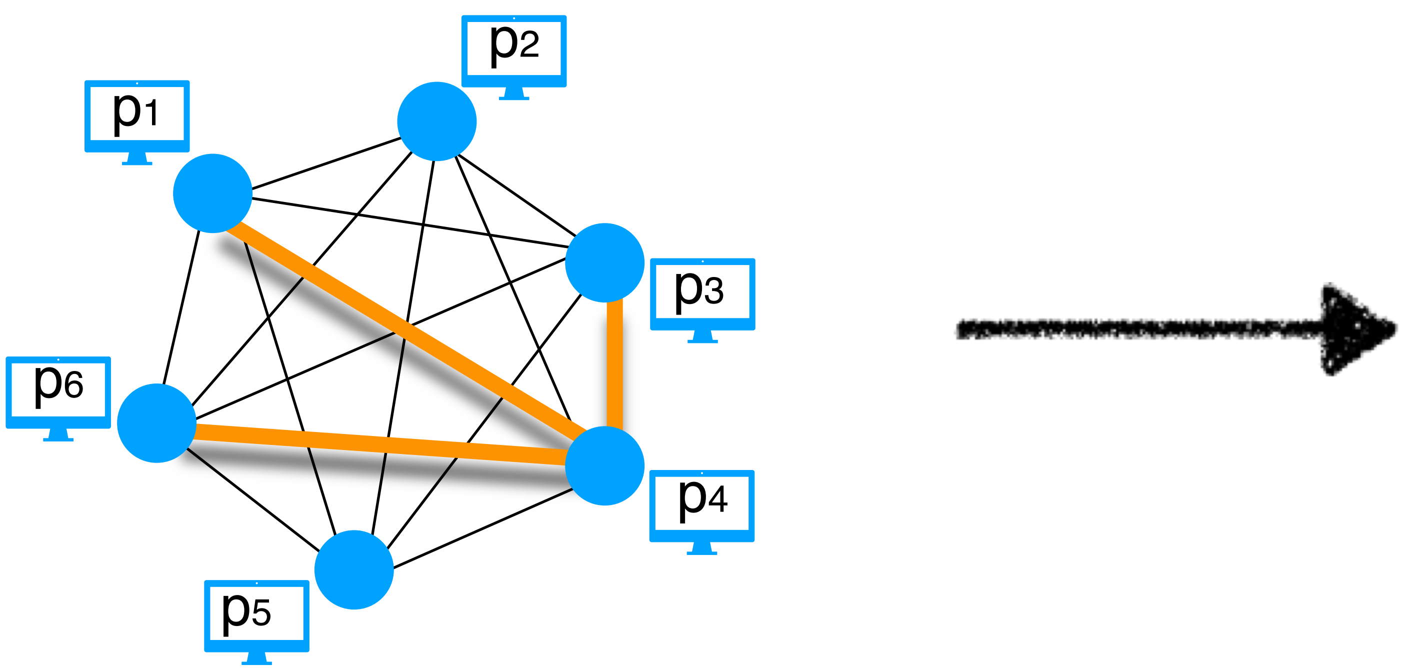


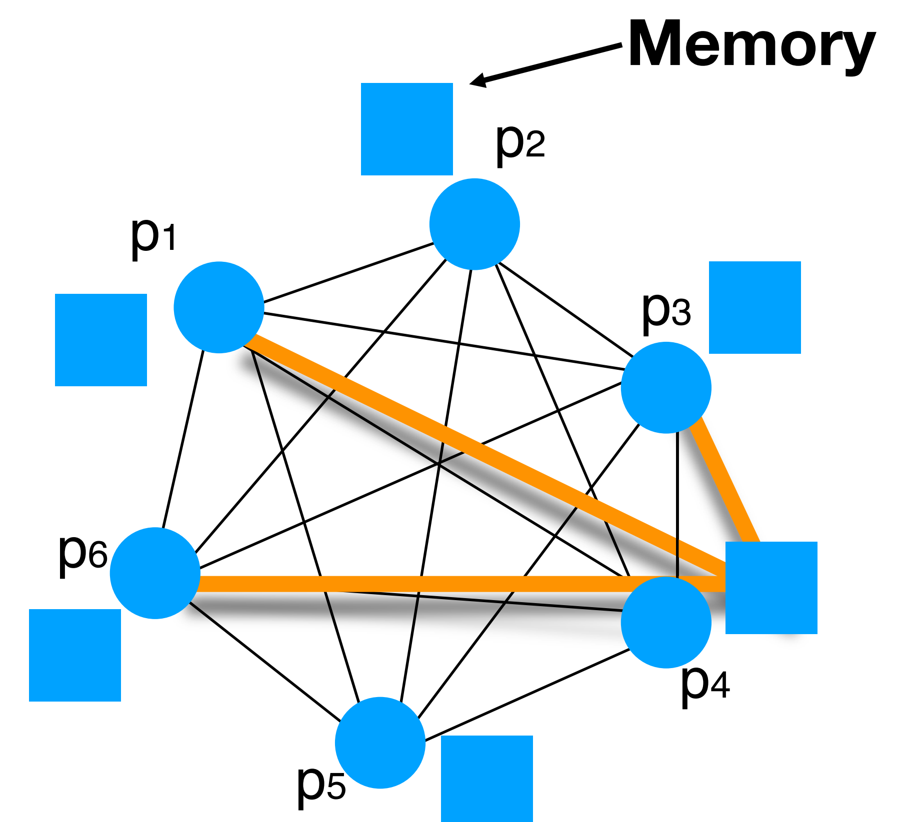


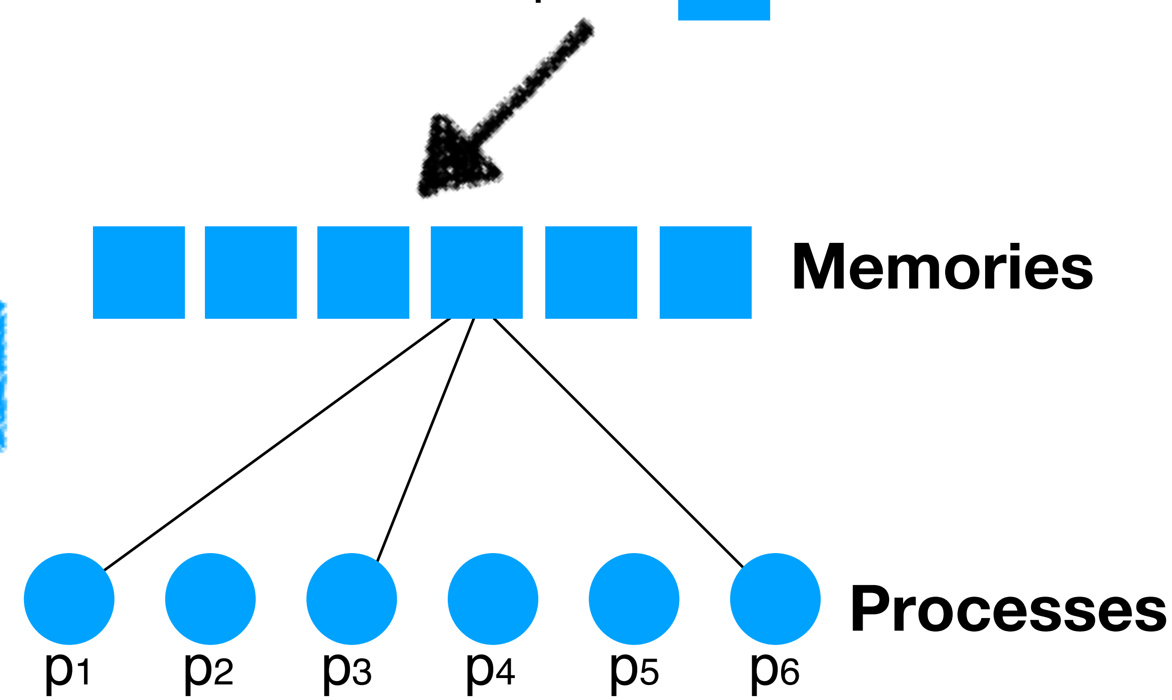
Outline

- What is RDMA?
- How we model RDMA
- Notable Results: consensus with RDMA
    - Crash faults
    - Byzantine faults

Modelling RDMA



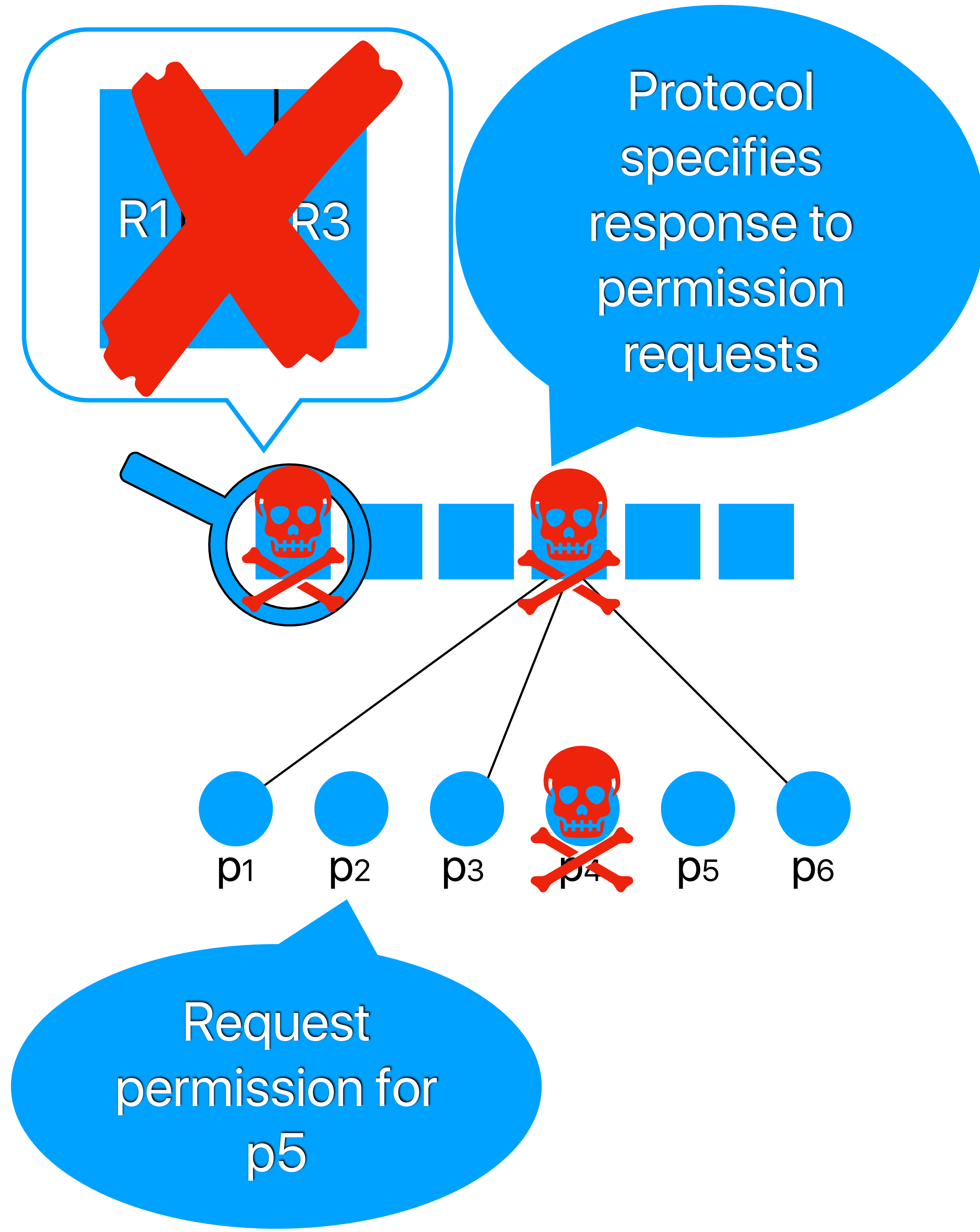




Only represent memory connections

Modelling RDMA

- Decouple processes and memory
- changePermission function can be called on memories
- Failures can occur on processes or memory
- When memory fails, all regions fail together
- We only consider RW memory (registers)



Outline

- What is RDMA?
- How we model RDMA
- Notable Results: consensus with RDMA
  - Crash faults
  - Byzantine faults

Refresher: O-Consensus

Paxos in Shared Memory

```
propose(v):
 while(true)
   Reg[i].T.write(ts);
   val := Reg[1,..,n].highestTspValue();
   if val = ⊥ then val := v;
   Reg[i].V.write(val,ts);
   if ts = Reg[1,..,n].highestTsp() then
      return(val)
   ts := ts + n
```

announce my timestamp

adopt value with highest ts (or mine if none)

announce my value, ts

if my timestamp is the highest, decide

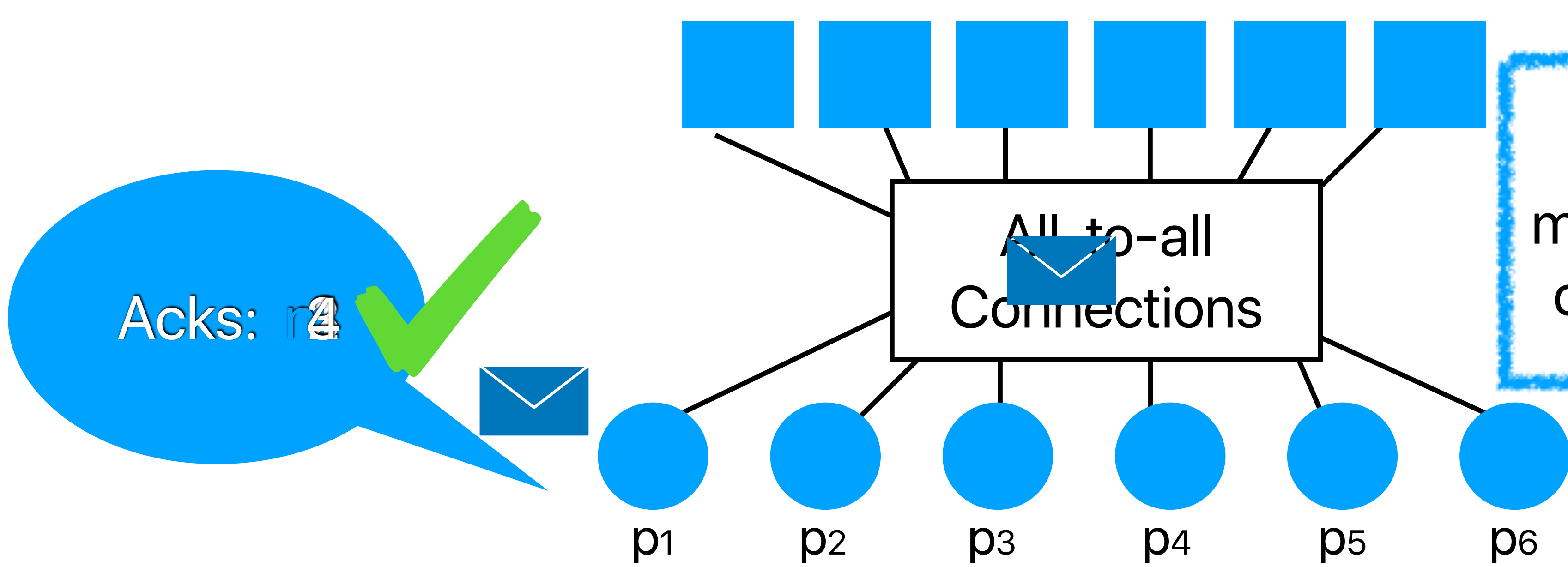
This assumes that shared memory never fails.

🤔 **What if memory can fail?** 🤔

Handling Memory Failures

Replication: Treat all memories the same

Send all write/read requests to all memories, wait to hear acknowledgement from majority



Instead of many faulty memories, we can now think of one non-faulty memory!

O-Consensus w Memory Failures

Disk Paxos [GafniLamport2002]

```
propose(v):
 while(true)
   for every memory m in parallel:
        Reg[m][i].T.write(ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    val := temp[1..m][1..n].highestTspValue();
    if val = ⊥ then val := v;
    for every memory m in parallel:
        Reg[m][i].V.write(val,ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    if ts = temp[1..m][1..n].highestTsp() then
        return(val)
    ts := ts + n
```

announce my timestamp

adopt value with highest ts (or mine if none)

announce my value, ts

if my timestamp is the highest, decide

O-Consensus w Memory Failures

```
propose(v):
 while(true)
   for every memory m in parallel:
        Reg[m][i].T.write(ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    val := temp[1..m][1..n].highestTspValue();
    if val = ⊥ then val := v;
    for every memory m in parallel:
        Reg[m][i].V.write(val,ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    if ts = temp[1..m][1..n].highestTsp() then
        return(val)
    ts := ts + n
```

Why read again here?

☝️ **Need to check if I ran alone!**

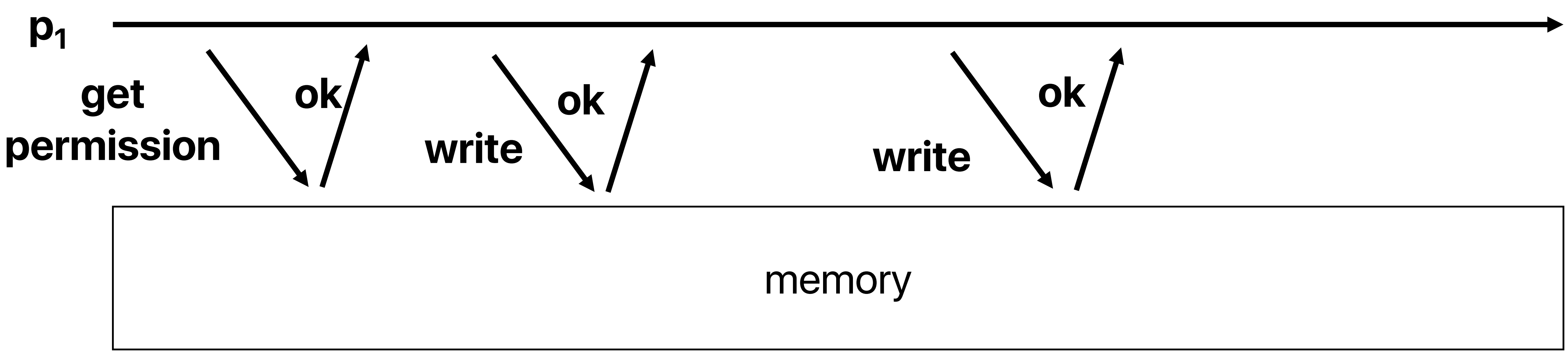
O-Consensus w Memory Failures

- If we don't read again, we might miss a concurrent process's timestamp
- This could lead to violation of agreement

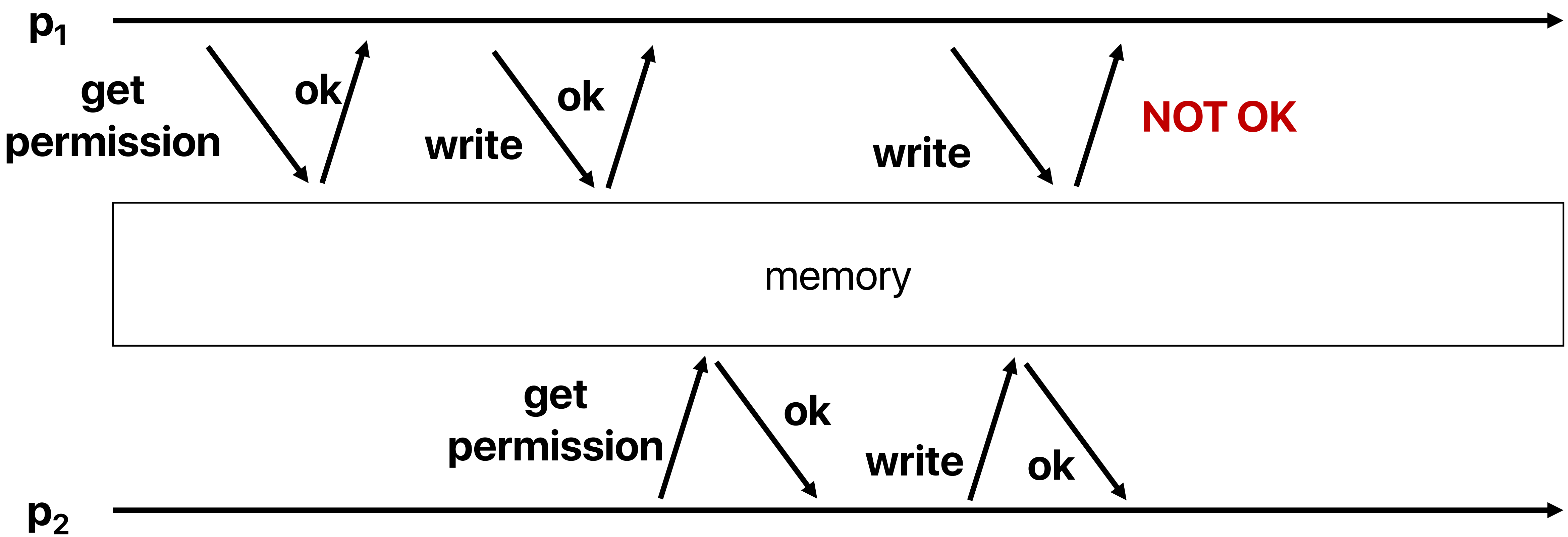
- What if there was another way to determine if there was a concurrent process?
- We wouldn't need the last read!

→ better complexity

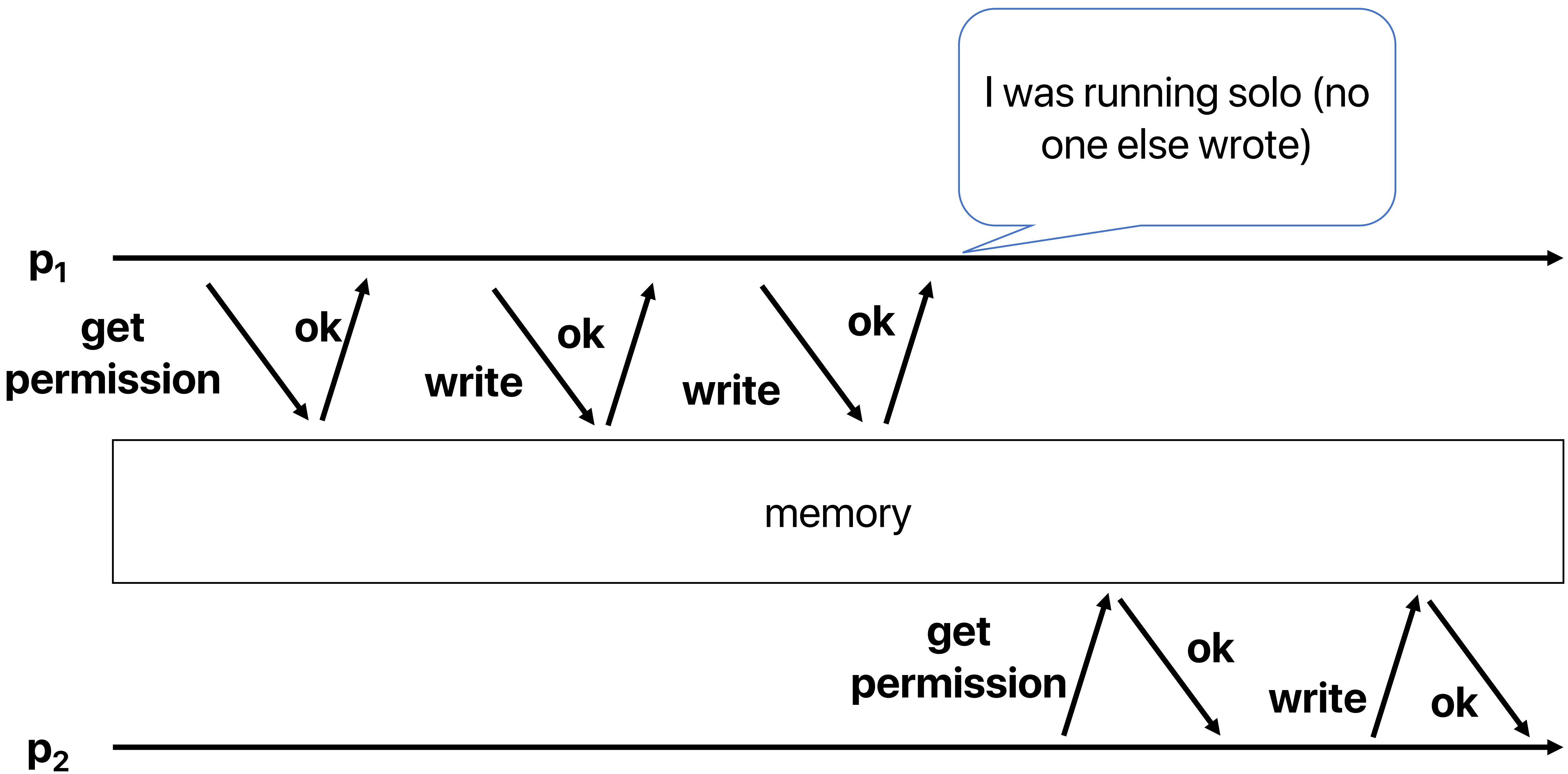
Solo Detection w/ Permissions



Solo Detection w/ Permissions



Solo Detection w/ Permissions



O-Consensus with Memory Failures and Permissions

```
propose(v):
 while(true)
    ts := ts + n
    for every memory m in parallel:
         m.getPermission();
         Reg[m][i].T.write(ts);
         temp[m][1..n] = Reg[m][1..n].read();
     until completed for majority of memories
     if ts < temp[1..m][1..n].highestTsp() then continue;
     val := temp[1..m][1..n].highestTspValue();
     if val = ⊥ then val := v;
     for every memory m in parallel:
         Reg[m][i].V.write(val,ts);
         ~~temp[m][1..n] = Reg[m][1..n].read();~~
     until completed for majority of memories
     if writes succeeded at majority of memories then
         return(val)
```

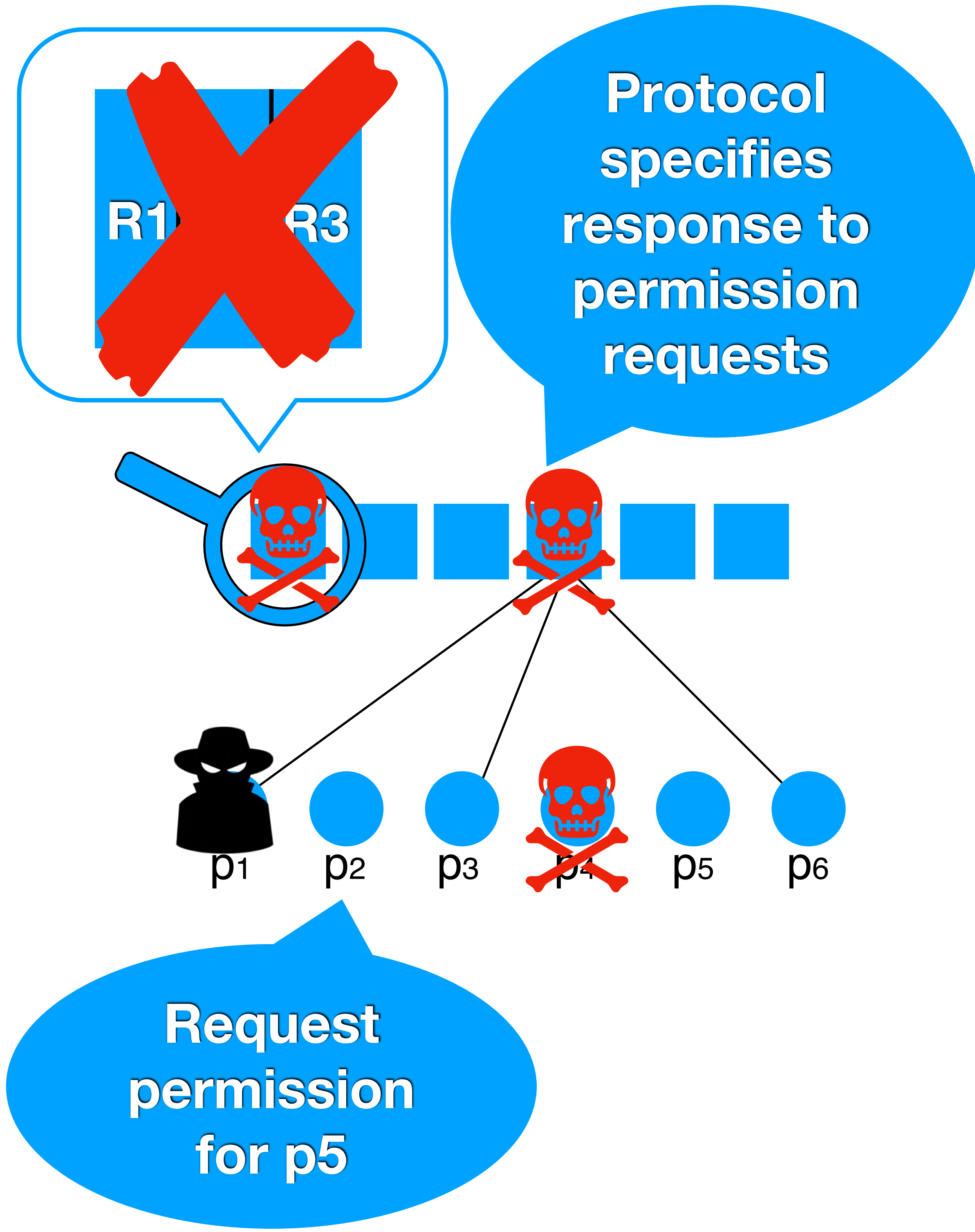
No need to read again!

Outline

- What is RDMA?
- How we model RDMA
- Notable Results: consensus with RDMA
  - Crash faults
  - Byzantine faults

Model: +Byzantine Failures

- Decouple processes and memory
- changePermission function can be called on memories
- Failures can occur on processes or memory
    - **Byzantine failures of processes**
- When memory fails, all regions fail together



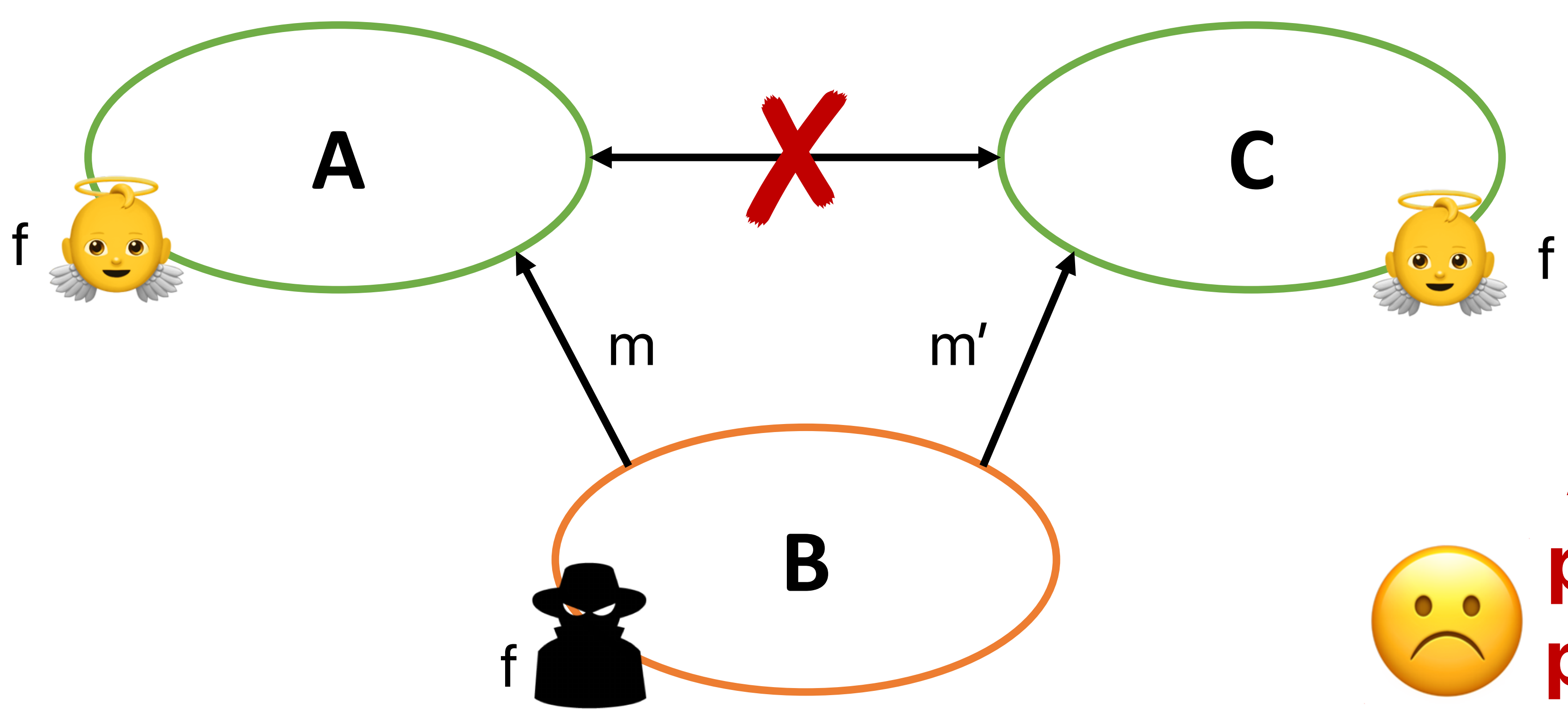
Equivocation

Non-equivocating Broadcast

- **Liveness**: If a correct process *p* broadcasts *m*, then all correct processes eventually deliver *m* from *p*.
- **Agreement**: If *p* and *q* are correct processes, *p* delivers *m* from *r*, and *q* delivers *m′* from *r*, then *m*=*m′* .
- **Validity**: If a correct process delivers *m* from *p*, *p* must have broadcast *m*.

NEB in Message Passing

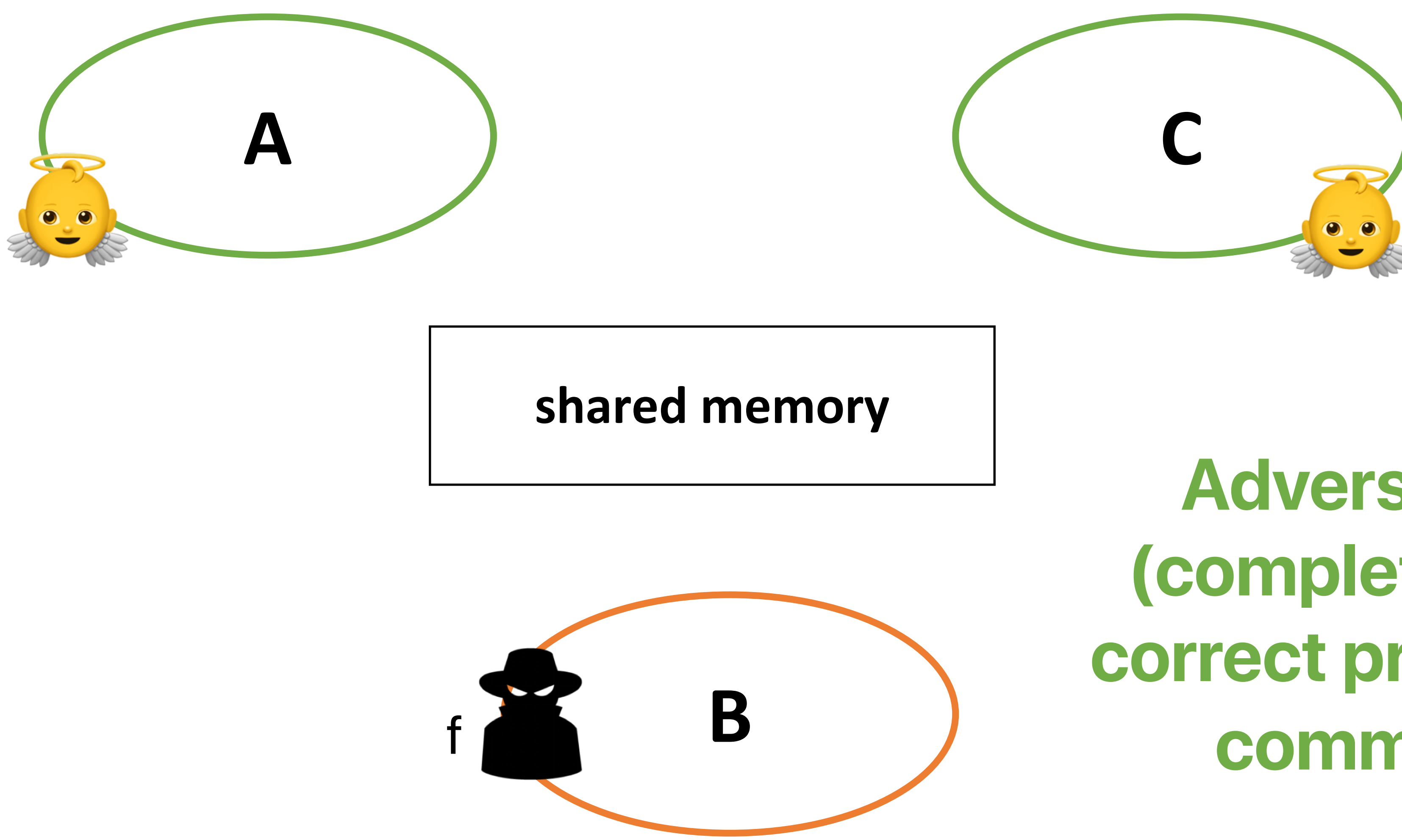
- Requires n=3f+1, where n is the total number of processes and up to f processes can be Byzantine
- Intuition:



Adversary can prevent correct processes from communicating

NEB in Shared Memory

- Only requires n>=f+1
- Intuition:



Adversary cannot (completely) prevent correct processes from communicating

NEB Algorithm—Data

- The processes maintain an array of SWMR registers R[1..n] (process i is the writer of R[i])
- The registers are initialized to $\perp$
- One of the processes (call it s) is the sender, all processes are receivers

NEB Algorithm

- To broadcast m:
  - R[s].write(m)

Side note: the sender cryptographically signs its message so that Byzantine processes cannot lie about what the sender said

- To receive:
  - senderMsg = R[s].read()
  - R[i].write(senderMsg)
  - for i=1..n
    - recvMsg = R[i].read()
    - if recvMsg != ⊥ && recvMsg != senderMsg then
      - return; // found conflicting values (Byzantine sender), don't deliver
  - deliver(senderMsg)

Part 2

Persistent Memory

Outline

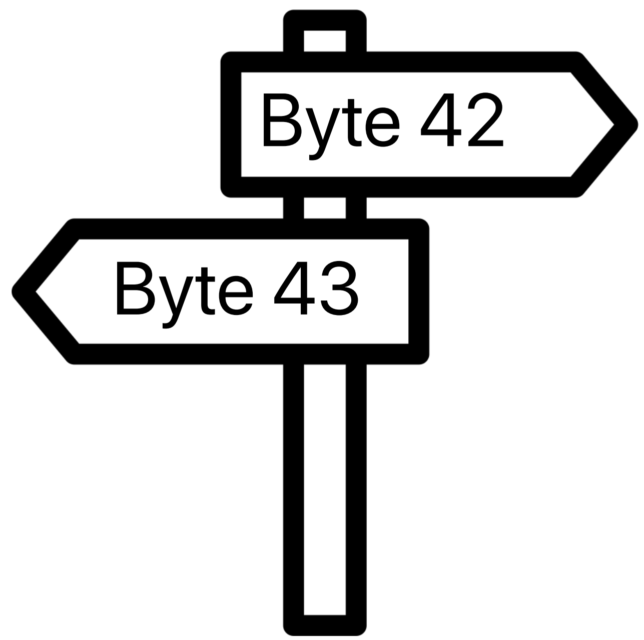
- What is persistent memory?
- How to define correctness for PM?
- Data Structures for PM
- A Lower Bound for PM

Outline

- What is persistent memory?
- How to define correctness for PM?
- Data Structures for PM
- A Lower Bound for PM

What Is Persistent Memory?

Durability in the face of crashes & recoveries



Byte-addressability

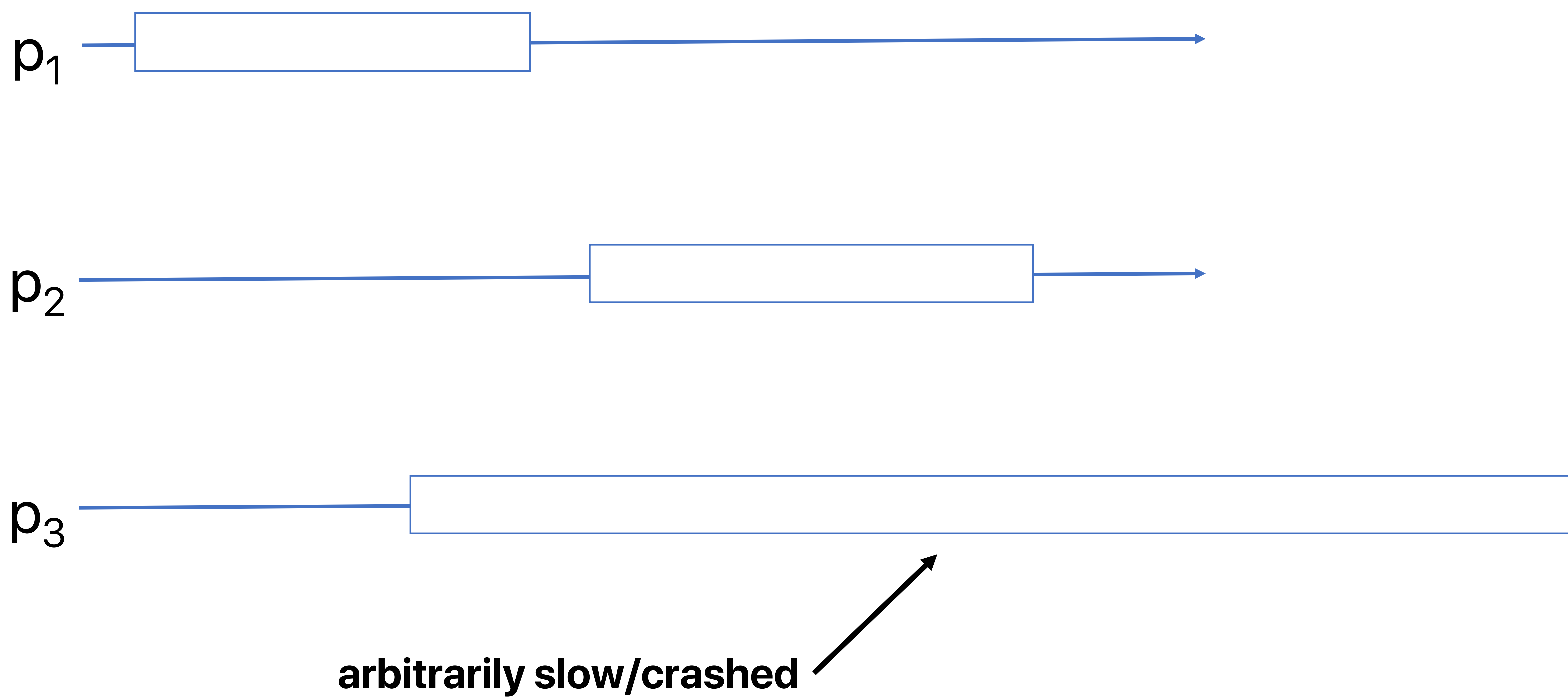
Access times ~ RAM

Outline

- What is persistent memory?
- How to define correctness for PM?
- Data Structures for PM
- A Lower Bound for PM

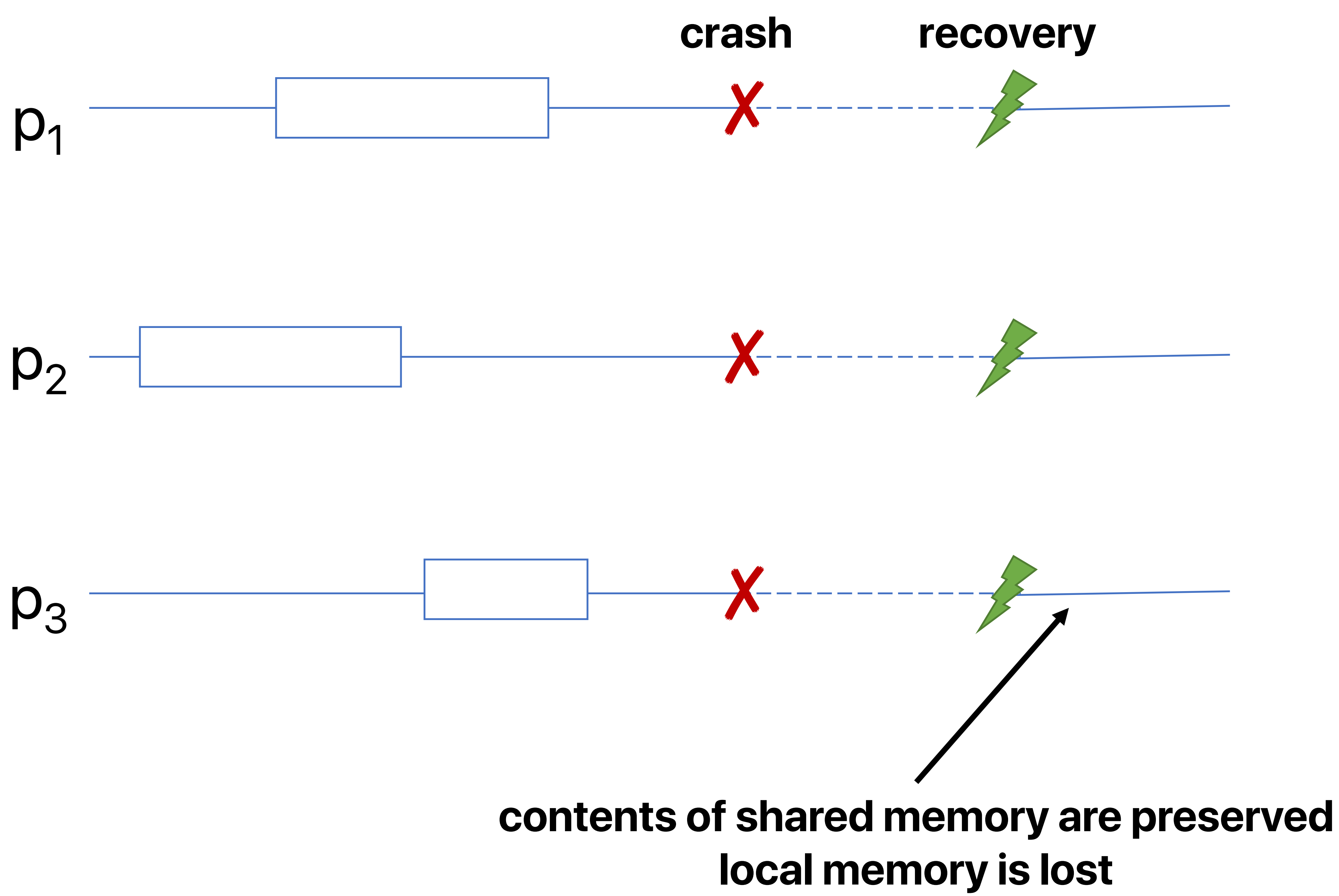
Modelling durability

Process delays & crashes



Modelling durability

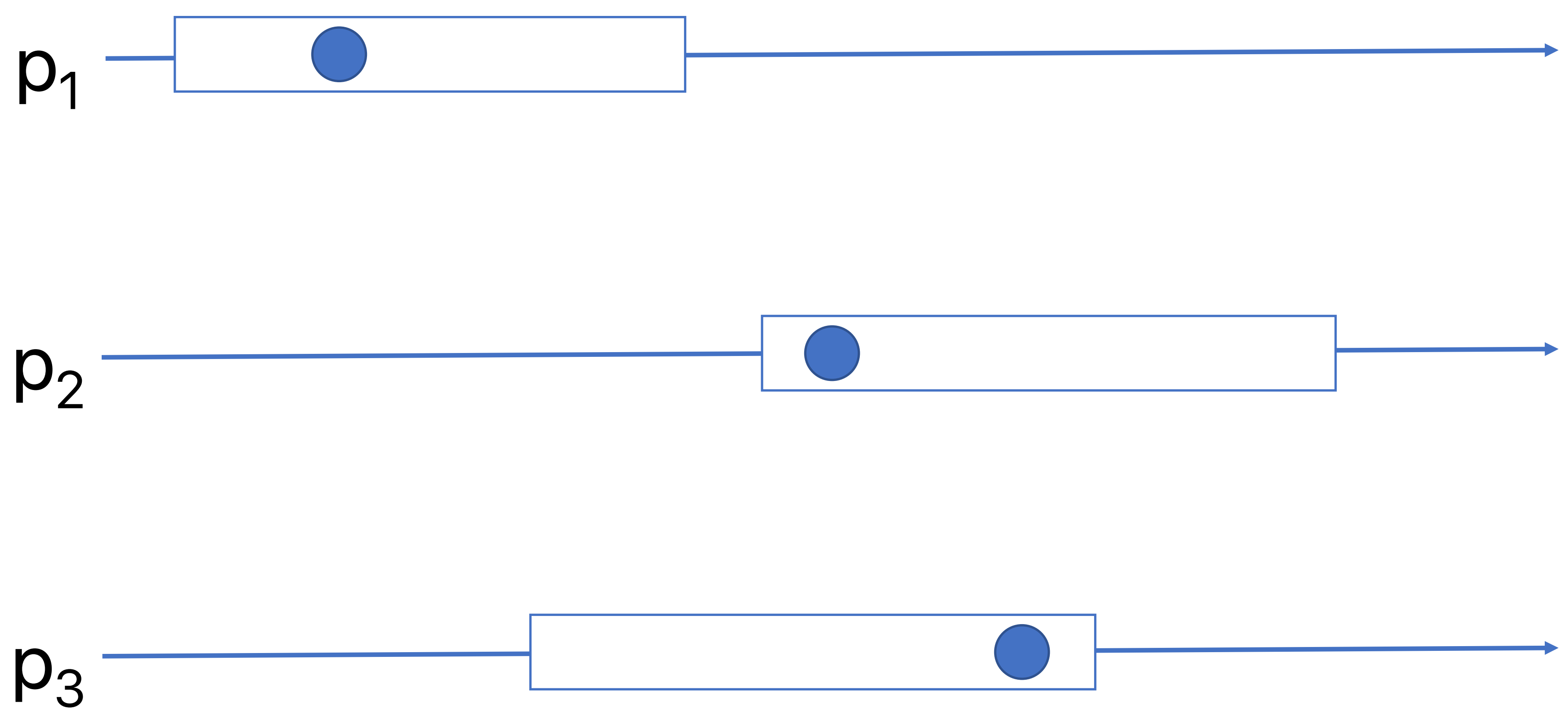
Full-system crash & recover



Recall: Atomicity

- Every operation appears to execute at some indivisible point in time (called linearization point) between the invocation and reply time events

Recall: Atomicity



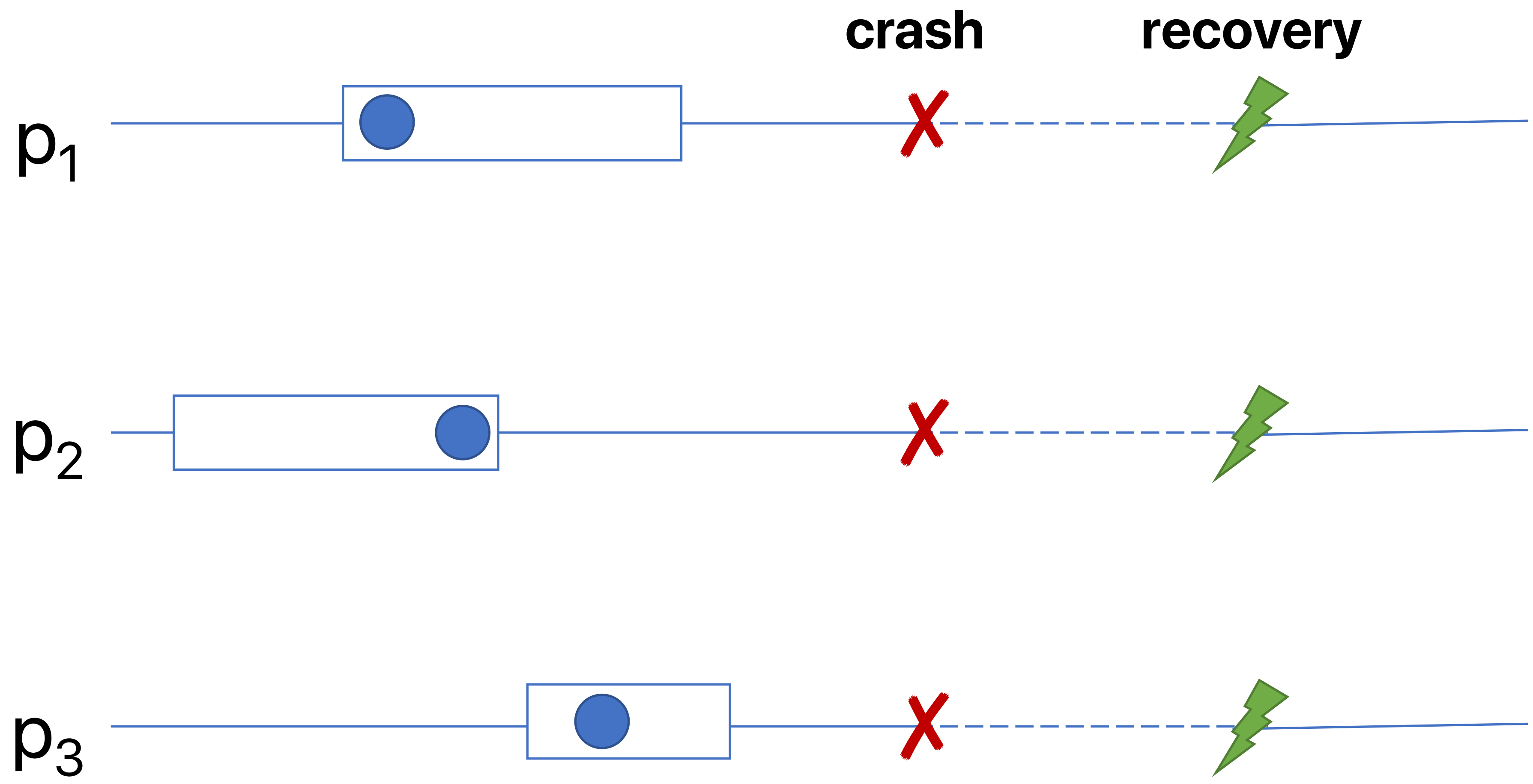
Atomicity & Persistent Memory

- How can we express atomicity in this model?

→ durable linearizability

Modelling durability

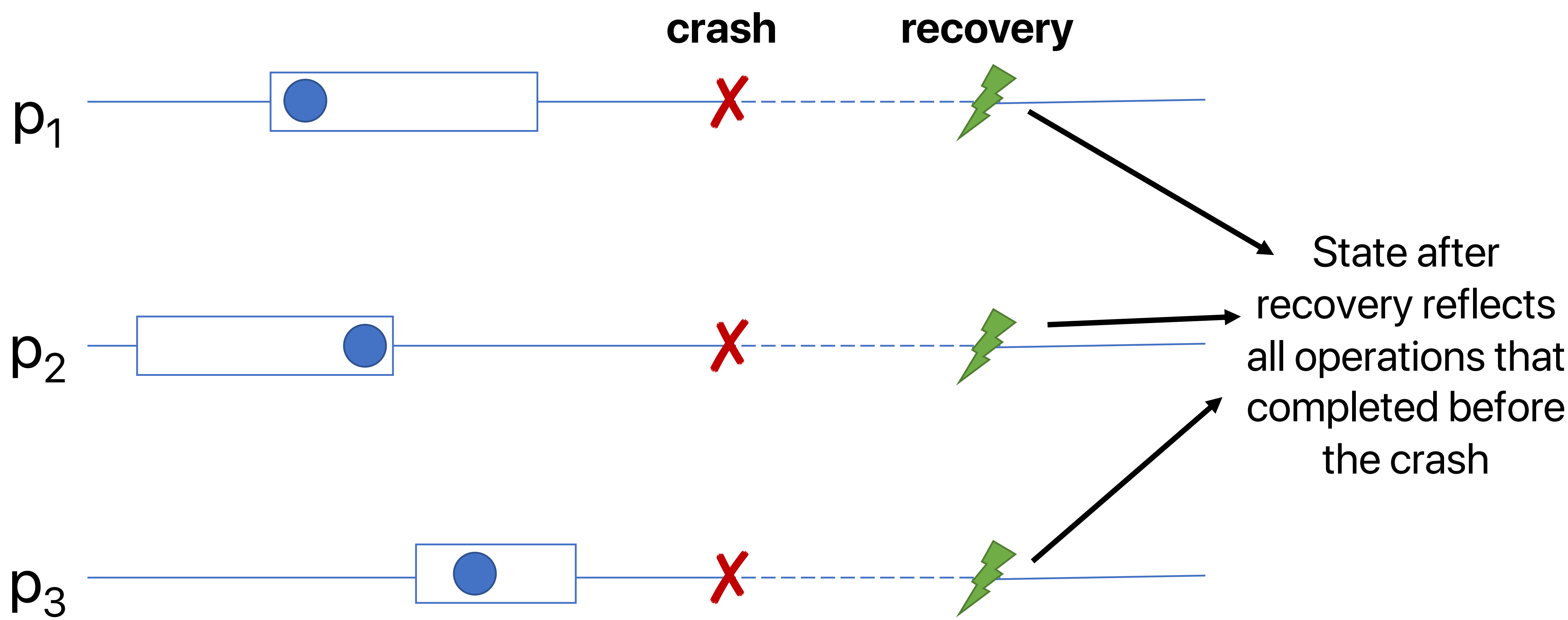
Durable Linearizability



When there is no crash: durable linearizability = atomicity as before

Modelling durability

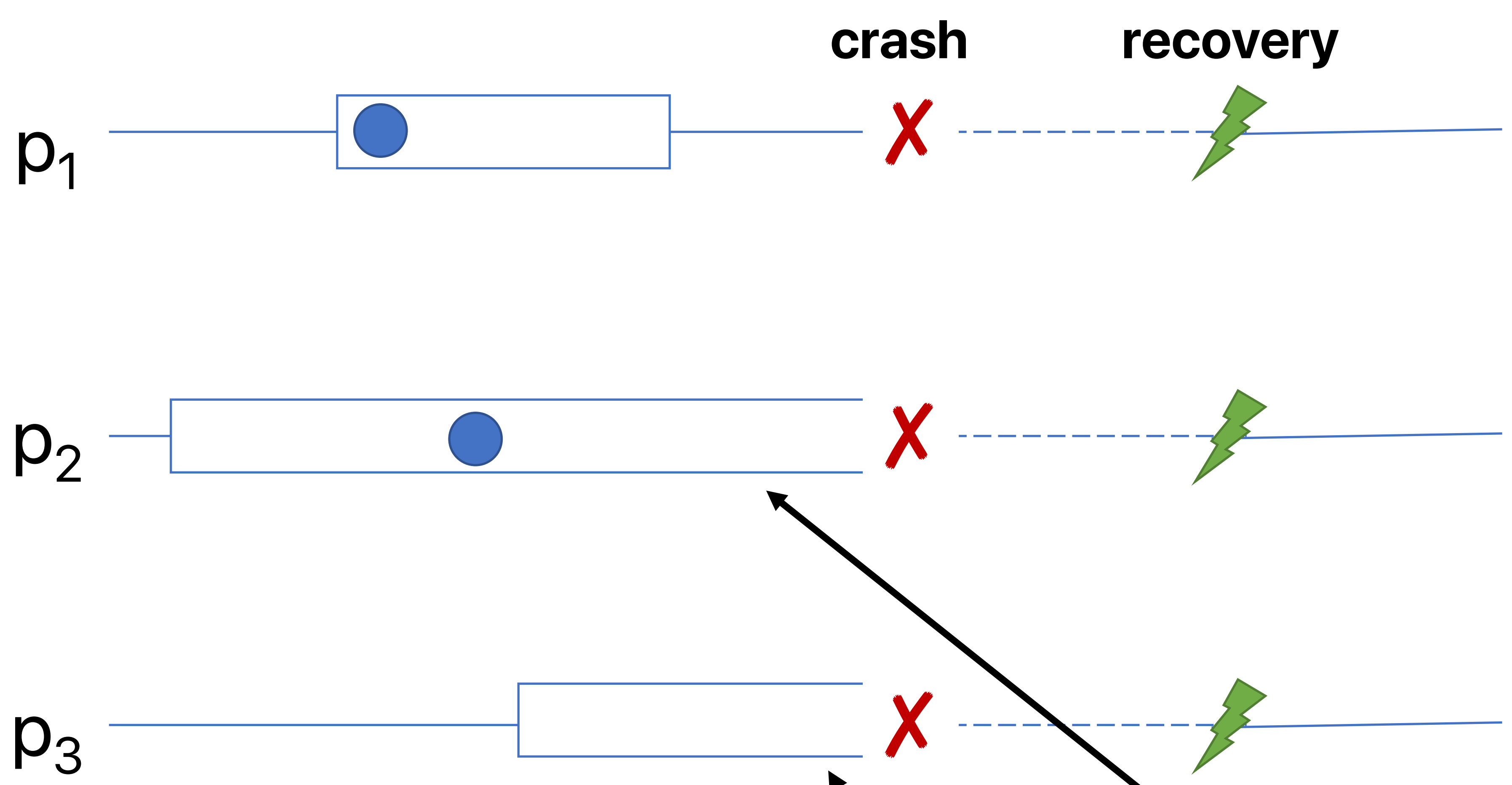
Durable Linearizability



When there is no crash: durable linearizability = atomicity as before

Modelling durability

Durable Linearizability

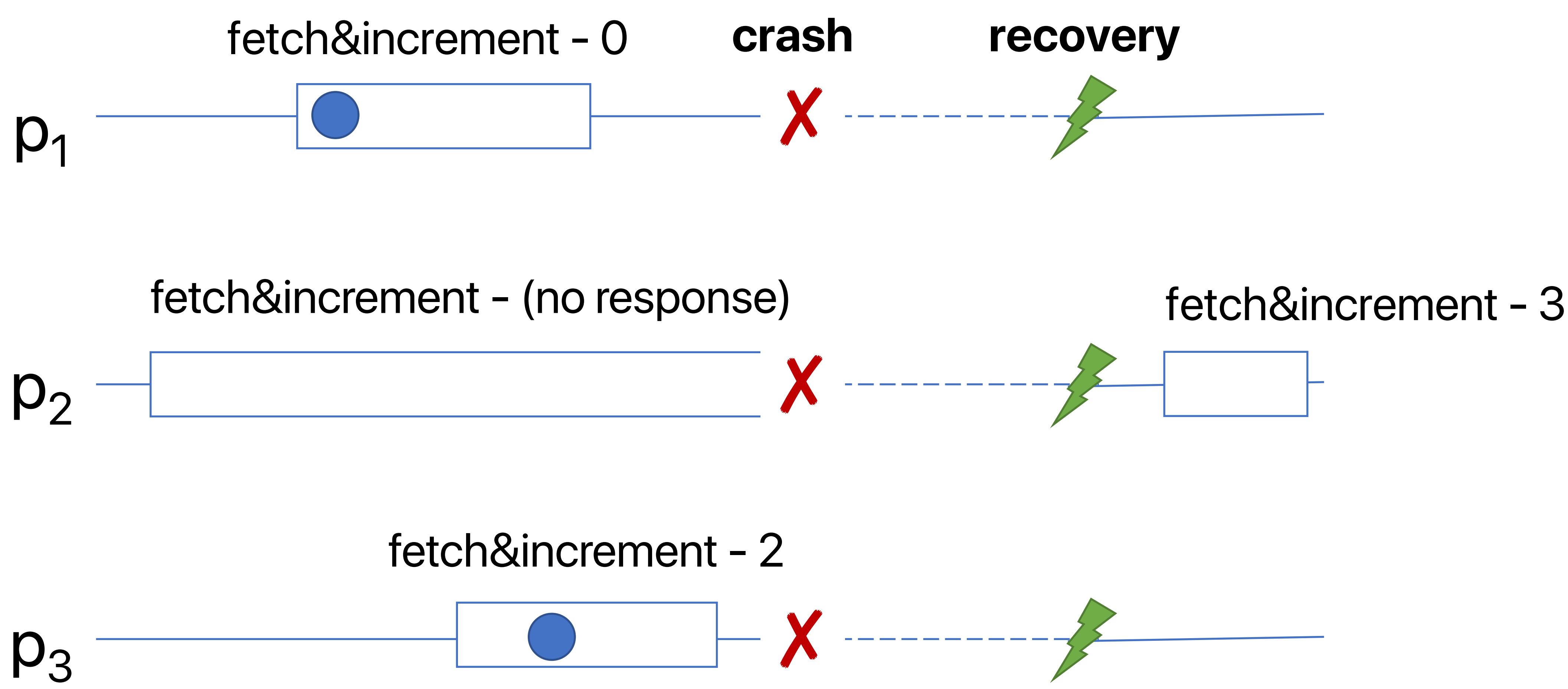


Operations that were ongoing during the crash may be kept (reflected in post-recovery state) or lost (not reflected)

Durable Linearizability

- If:
    1. an operation A depends on an operation B, and
    2. A is reflected in the post-recovery state,
- Then B must also be reflected in the post-recovery state.

Example



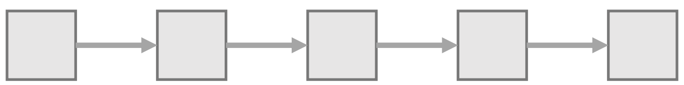
fetch&increment - 0
crash
recovery
p_1
fetch&increment - (no response)
fetch&increment - 3
p_2
fetch&increment - 2
p_3

Outline

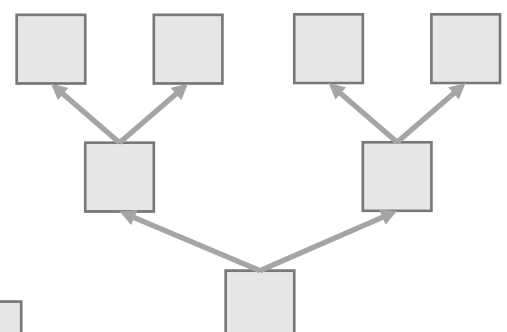
- What is persistent memory?
- How to define correctness for PM?
- Data Structures for PM
- A Lower Bound for PM

Concurrent Data Structures

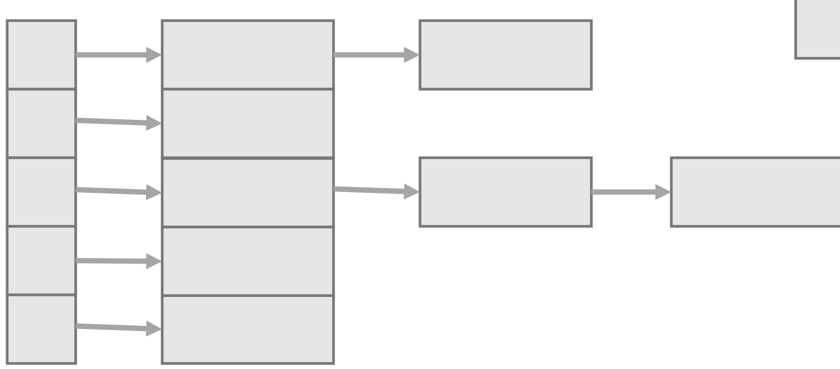
Lists



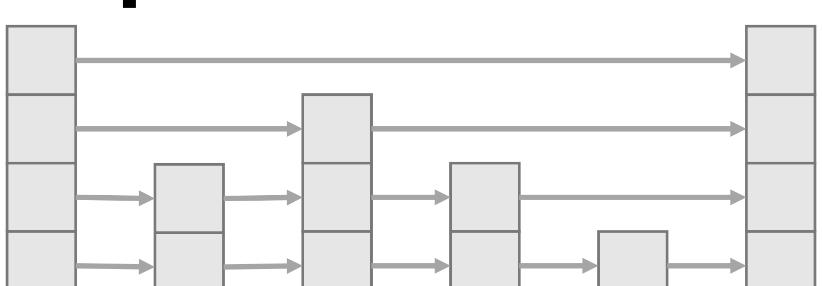
Trees



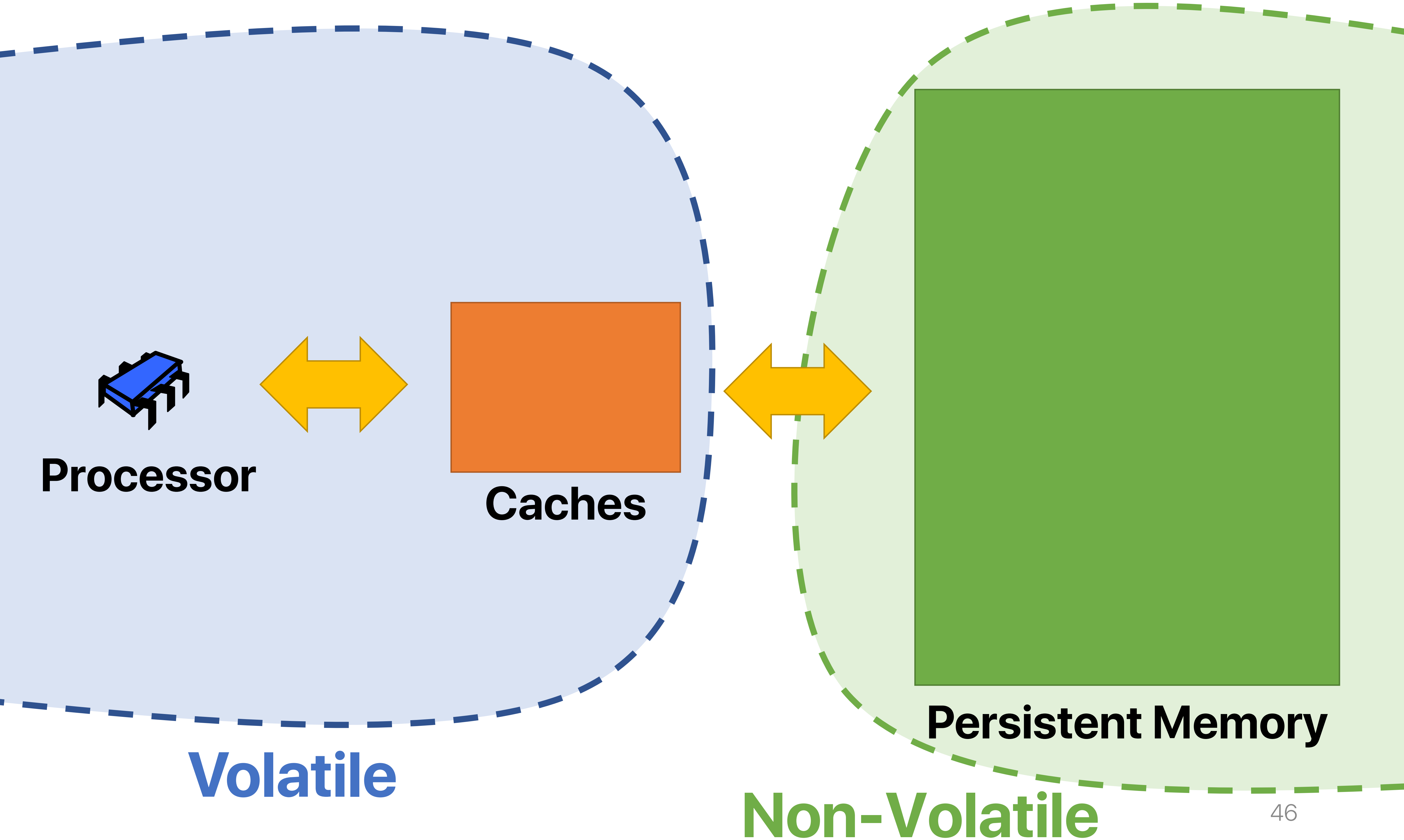
Hash tables



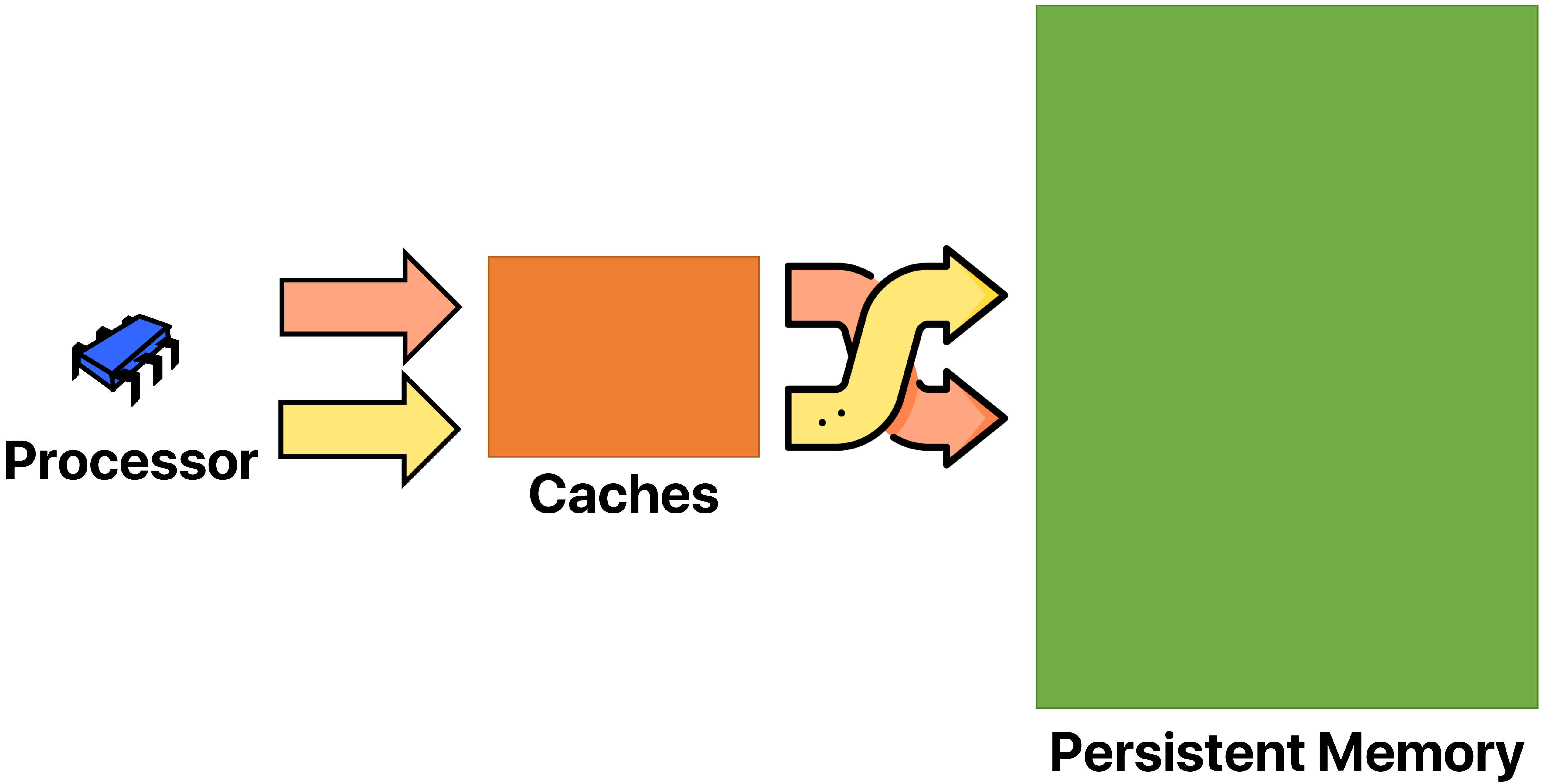
Skip lists



Obstacle #1: Caches are Volatile



Obstacle #2: (Re-)ordering



Obstacles Illustrated

```
1: mark memory as allocated
2: initialize memory
3: change link of node 1
4: change link of node 2
5: done = 1
```

Write-back cache:

```
1: mark allocation
2: initialize mem
3: change link 1
4: change link 2
5: done = 1
```

crash

NV memory:

```
3: change link 1
5: done = 1
```

Upon restart: incorrect state

Obstacles Illustrated

```
1: mark memory as allocated
2: persist allocation
3: initialize memory
4: persist memory content
5: change link of node 1
6: persist new link
7: change link of node 2
8: persist modified link
9: done = 1
```

Write-back cache:

```
1: mark allocation
2: initialize mem
3: change link 1
4: change link 2
5: done = 1
```

NV memory:

```
3: change link 1

5: done = 1
```

crash

Upon restart: incorrect state

Obstacles Illustrated

```
1: mark memory as allocated
2: persist allocation
3: initialize memory
4: persist memory content
5: change link of node 1
6: persist new link
7: change link of node 2
8: persist modified link
9: done = 1
```

Write-back cache:

```
1: mark allocation
2: initialize mem
3: change link 1
4: change link 2
5: done = 1
```

NV memory:

```
1: mark allocation
2: initialize mem
3: change link 1
```

crash

Upon restart: incomplete operat

Common Solution: Logging

```
1: log[0] = starting transaction X
2: persist log[0]
3: log[1] = allocating a node at address A
4: persist log[1]
5: mark memory as allocated
6: persist allocation
7: initialize memory
8: persist memory content
9: log[2] = previous value of link
10: persist log[2]
11: change link 1
12: persist modified link
13: log[3] = previous value of link
14: persist log[3]
15: change link 2
16: persist modified link
17: done = 1
18: persist done
19: mark transaction X as finished
```

Frequent waiting for data to be persisted

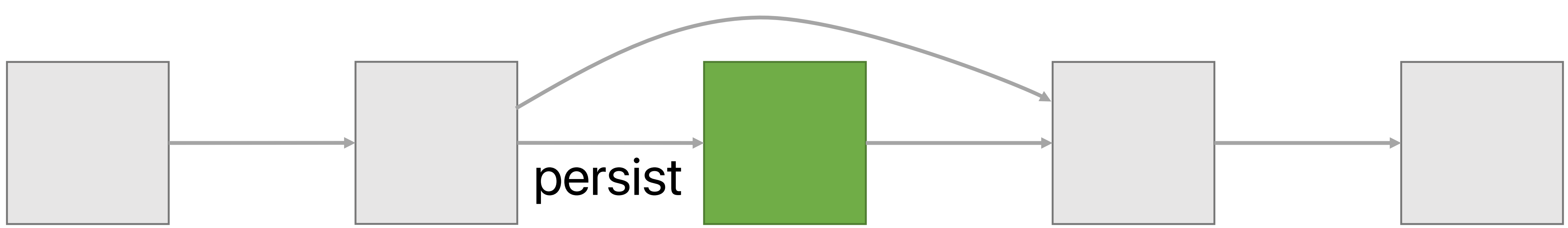
The Problem with Logging

- Logging -> frequent waiting
  - slows down data structure performance
- Data structure performance is essential to overall system performance

The solution: reduce (or eliminate) logging

Recall: Durable Linearizability

- After a restart, the structure reflects:
  - all operations completed (linearized) before the crash;
  - (potentially) some operations that were ongoing when the crash occurred;



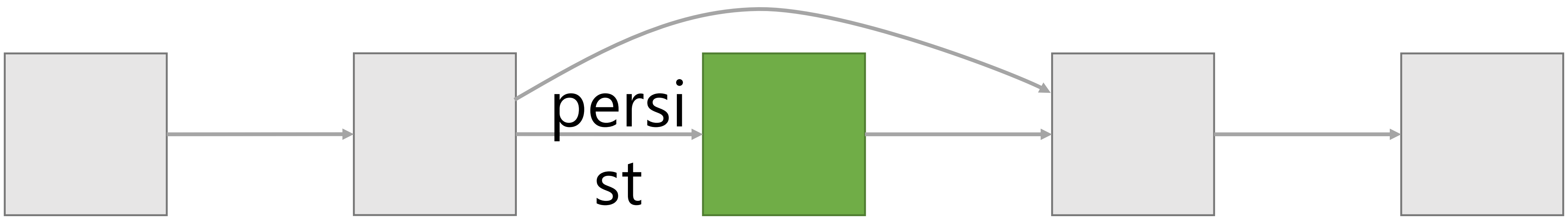
If crash between steps 2 and 3, violation of durable linearizability

1. Persistently allocate and initialize
2. Add link to new node
3. Persist link to new node

Log-free Data Structures

- The main idea: use lock-free algorithms
  - They never leave the structure in an inconsistent state
  - No need for logging in the data structure algorithm

Log-free Data Structures

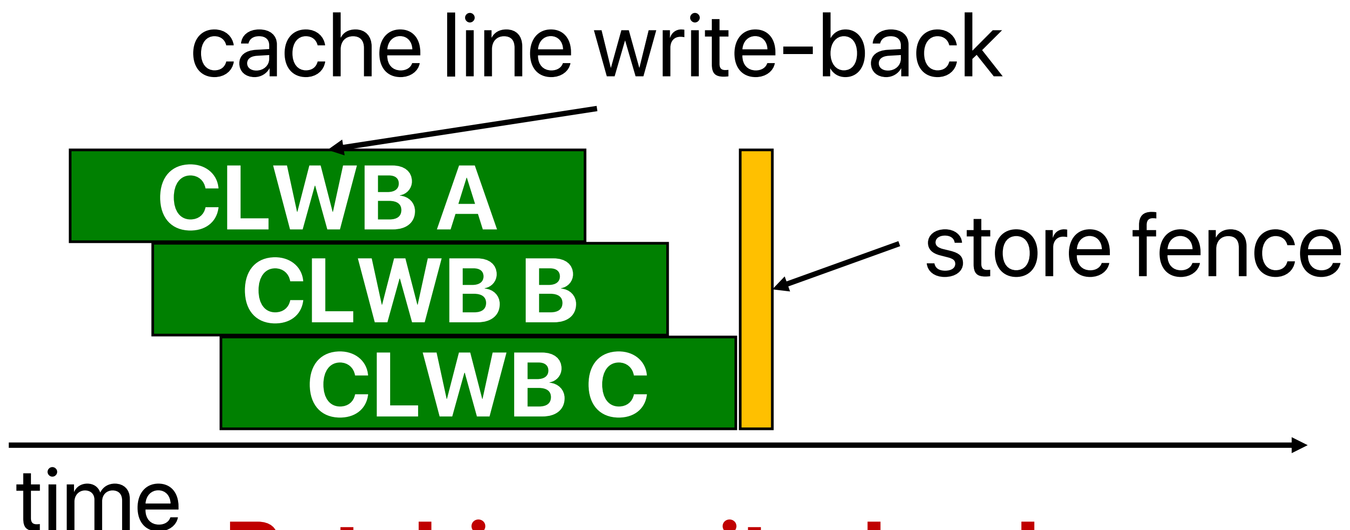


1. Persistently allocate and initialize node
2. Add marked link to new node
3. Persist link to new node
4. Remove mark

Other threads - persist marked link if needed

Link-and-persist: atomic “modify” and “persist” link

Going Further: Batching



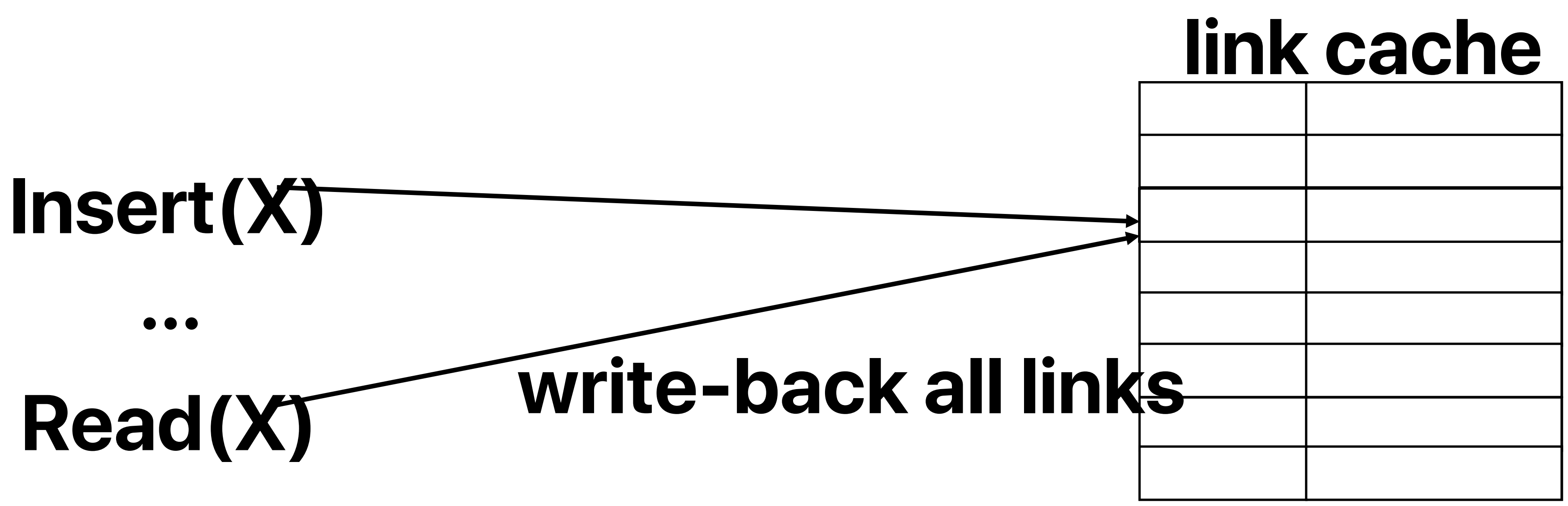
Batching write-backs: beneficial for performance

Going Further: Batching

- A link only needs to be persisted when an operation depends on it
- Store all un-persisted links in a fast concurrent cache
- When an operation directly depends on a link in the cache:

batch write-backs of all links in the cache

(and empty the cache)



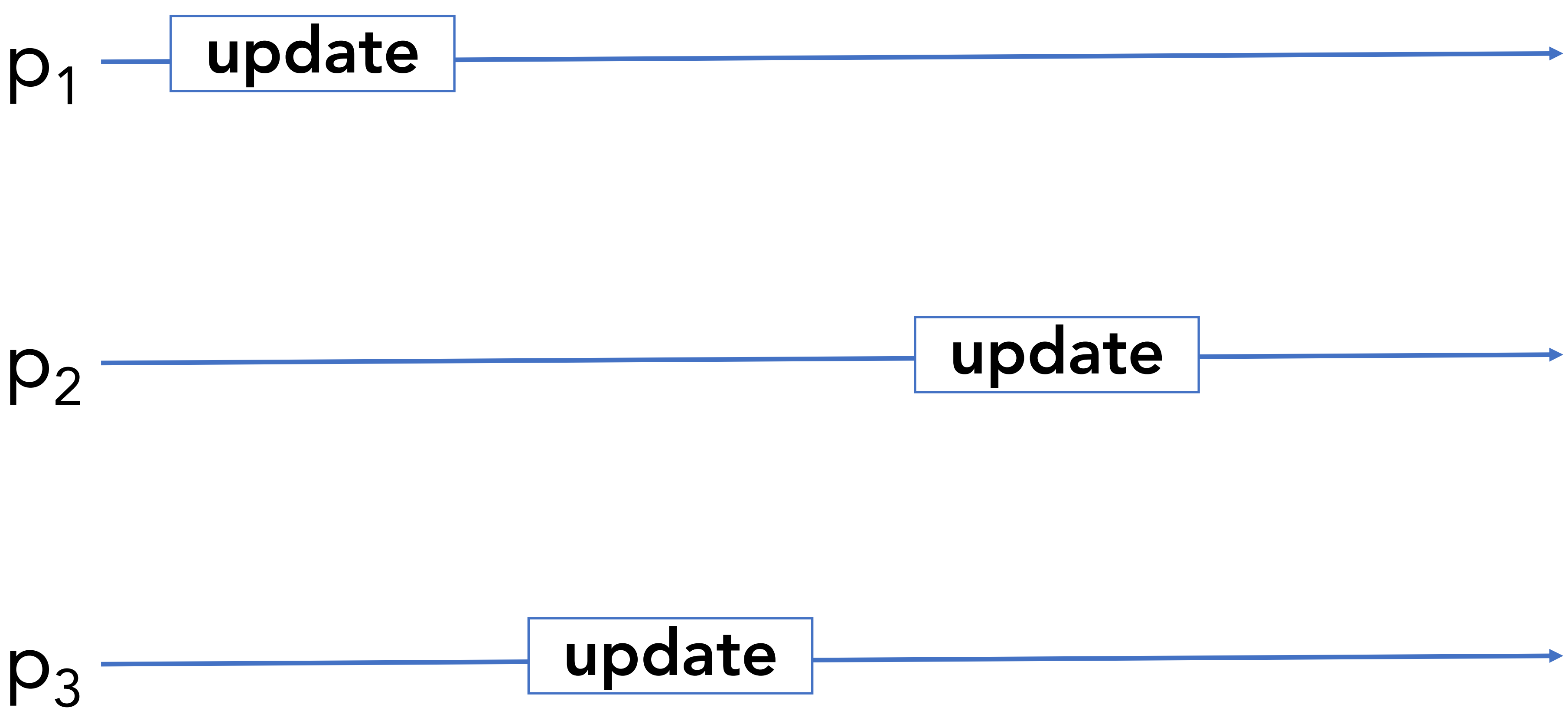
Outline

- What is persistent memory?
- How to define correctness for PM?
- Data Structures for PM
- A Lower Bound for PM

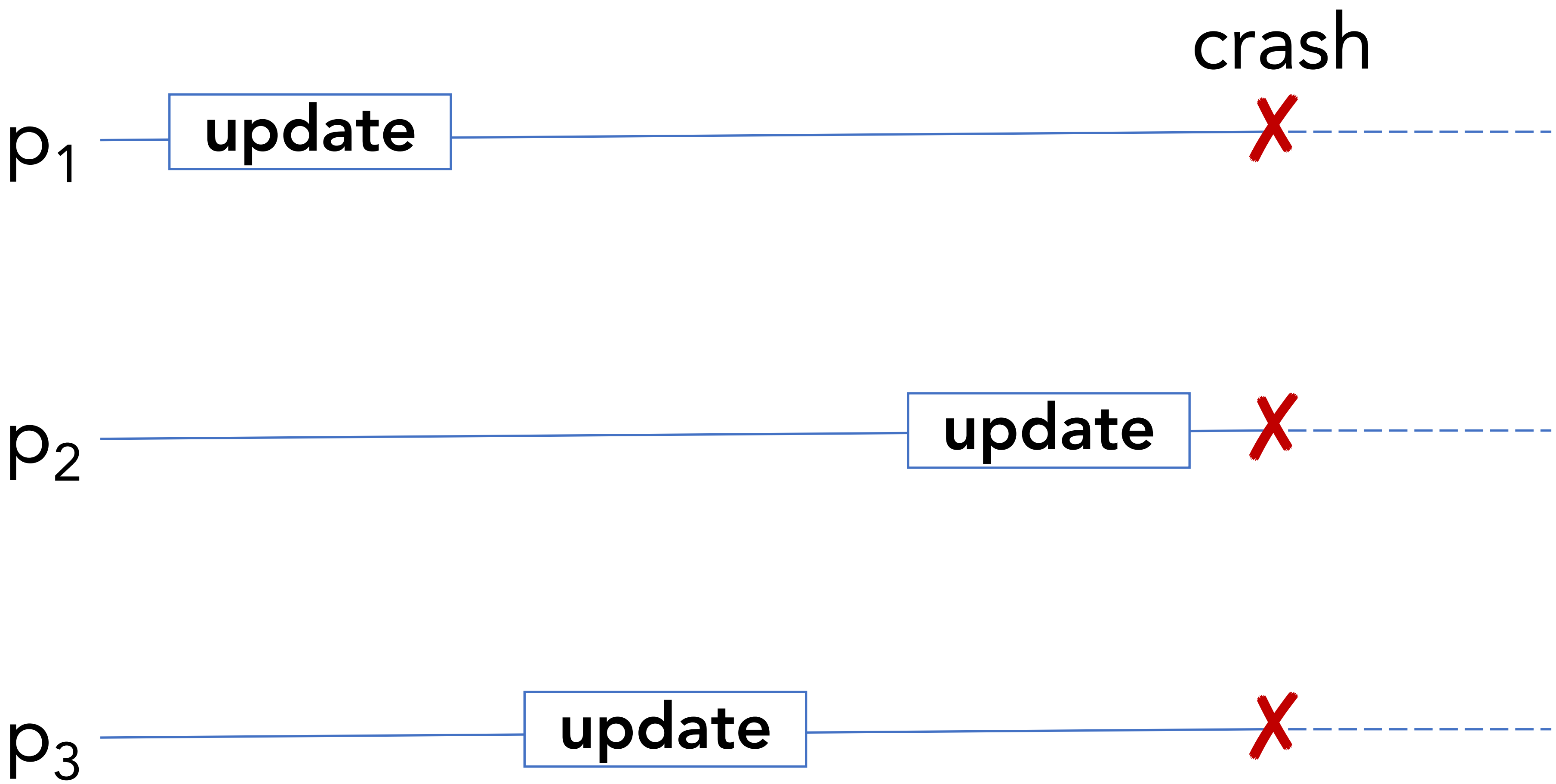
You Can't Eliminate Fences

- For any lock-free concurrent implementation of a persistent object
- there exists an execution E such that
- in E, every update operation performs at least 1 persistent fence

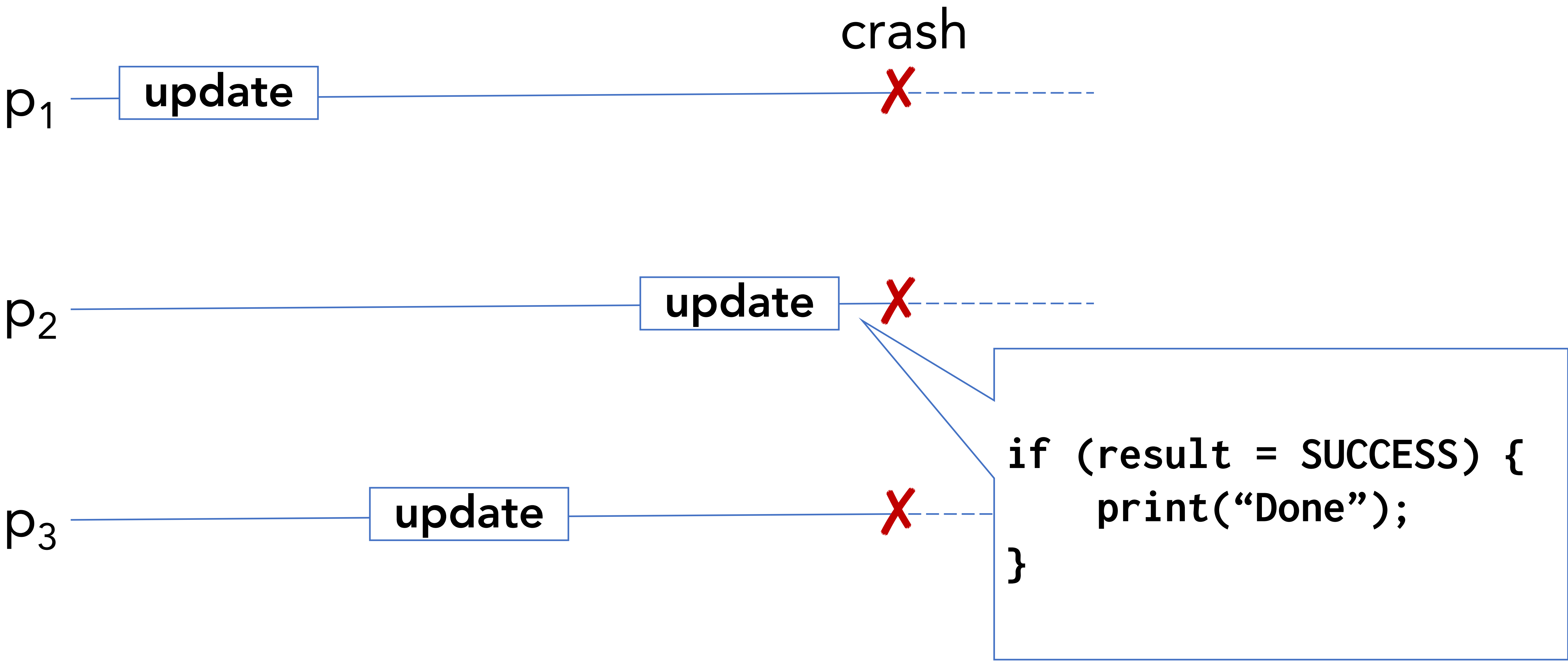
Lower Bound: Sequential Case



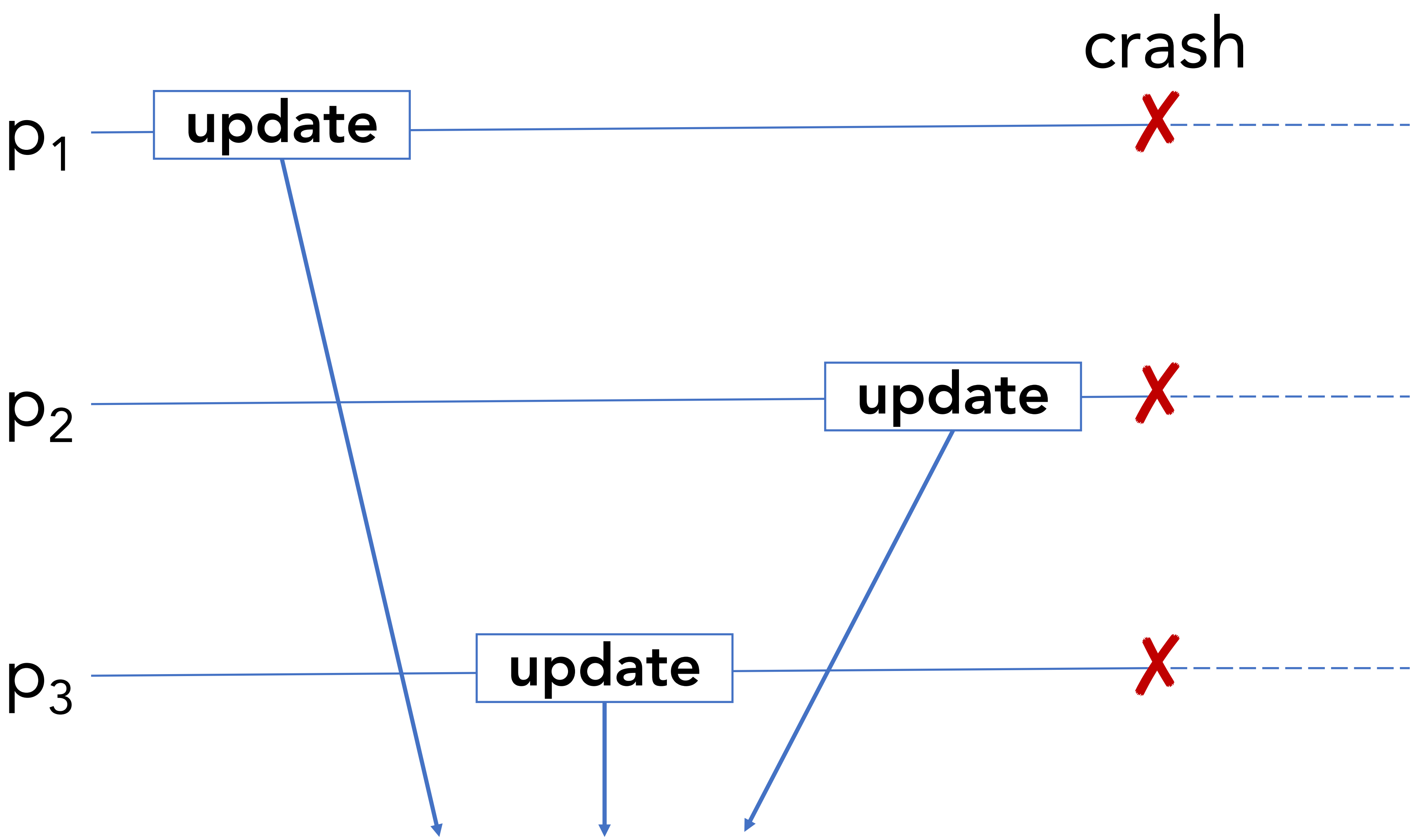
Lower Bound: Sequential Case



Lower Bound: Sequential Case

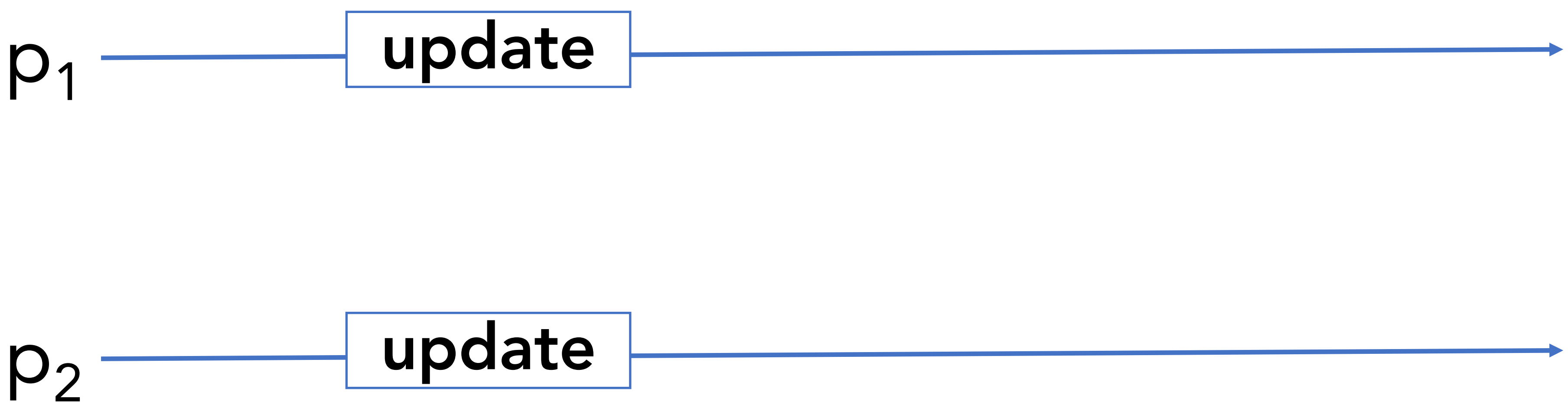


Lower Bound: Sequential Case

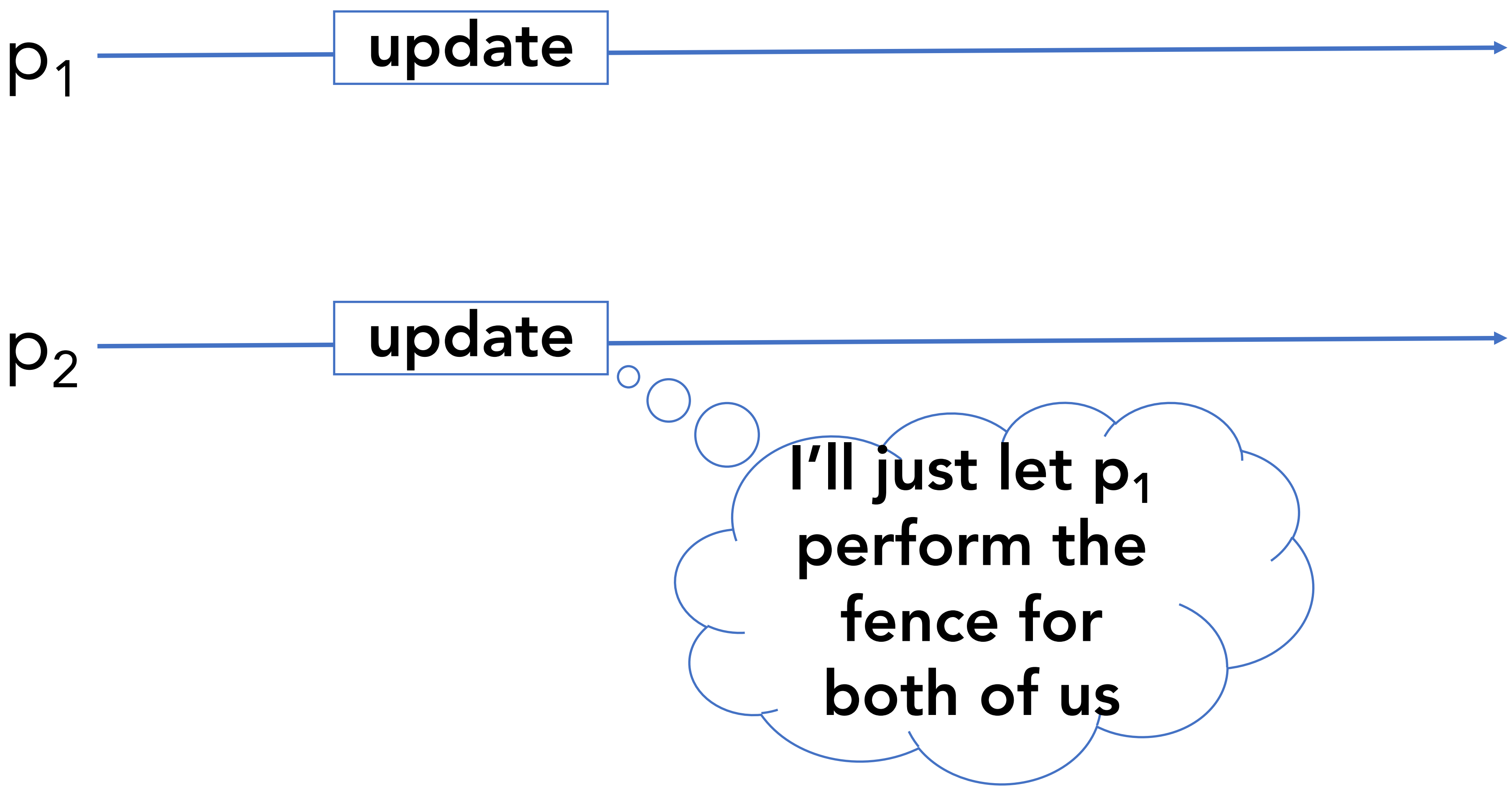


Need at least 1 persistent fence for every update.

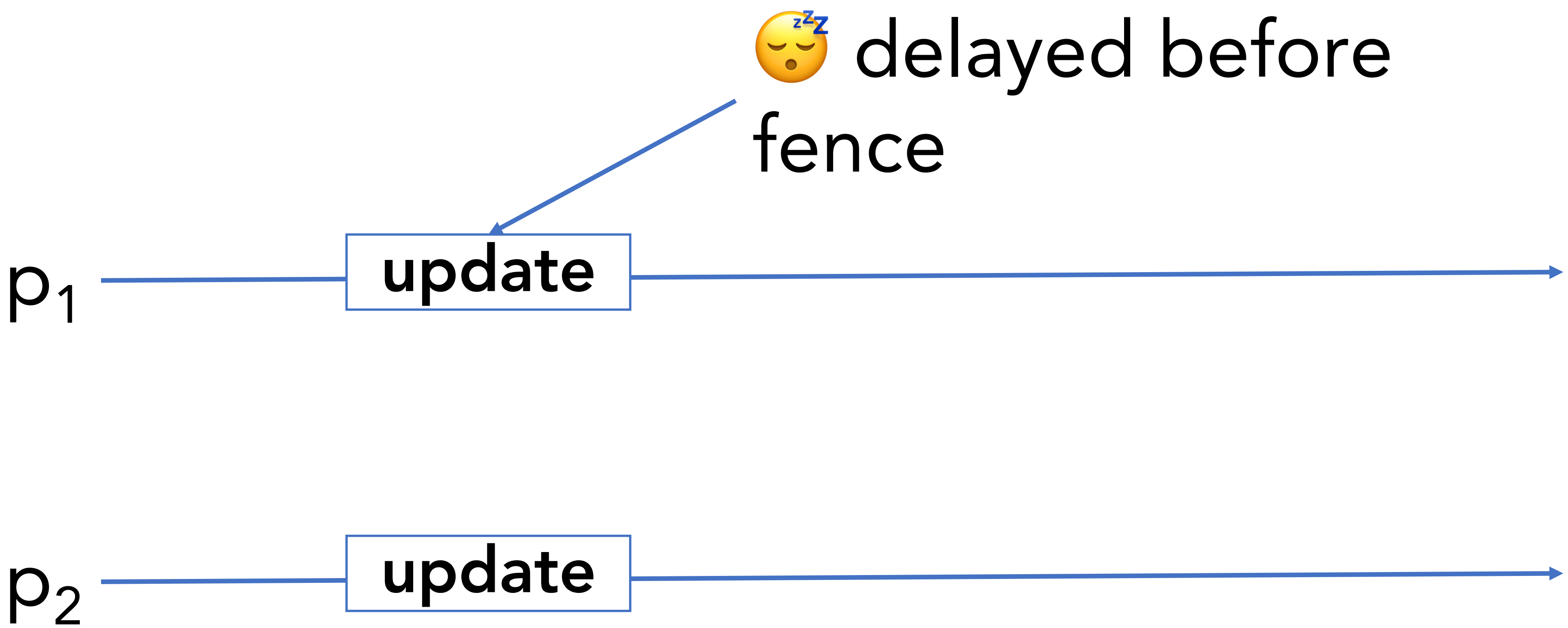
Lower Bound: Concurrent Case



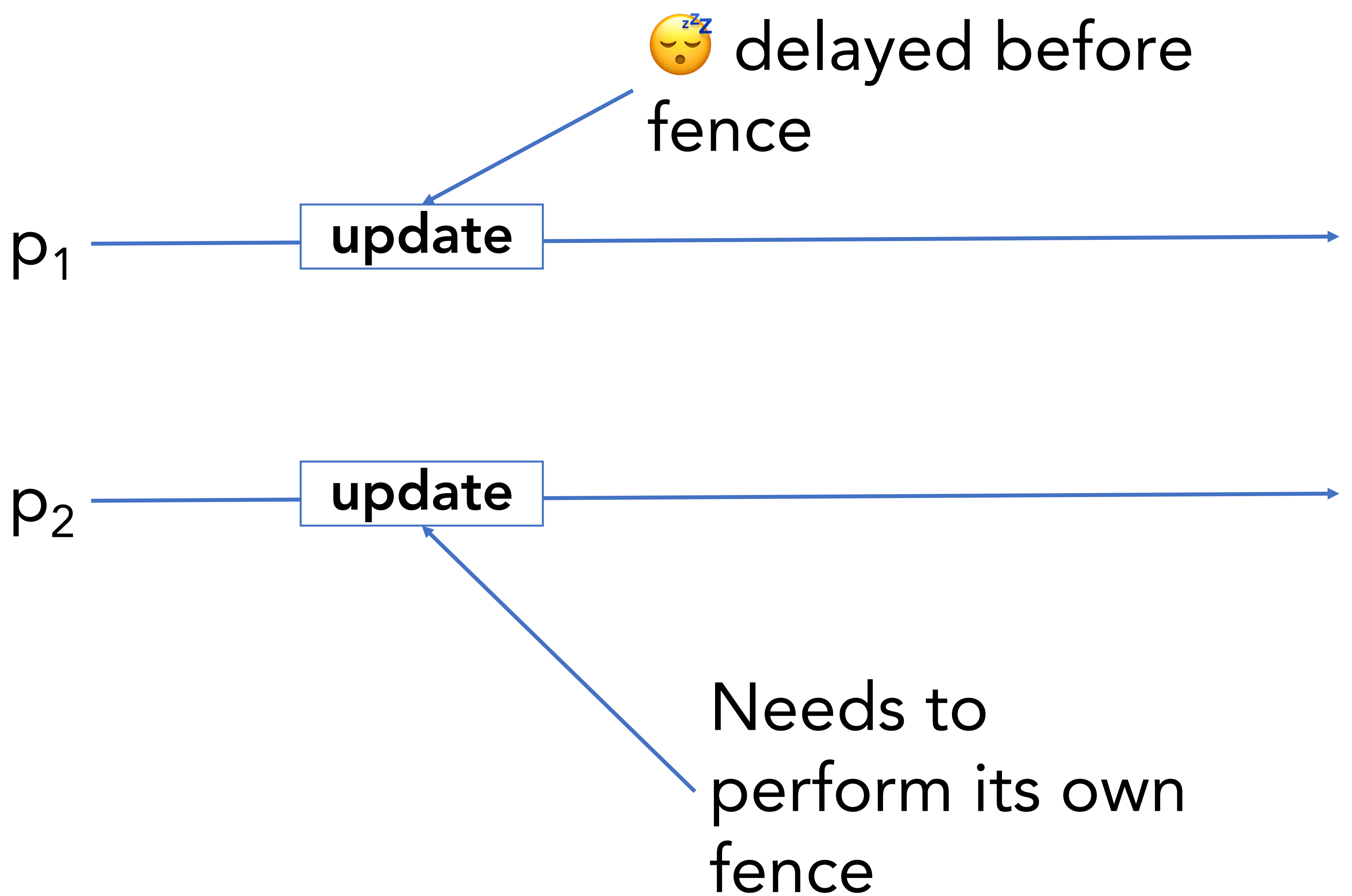
Lower Bound: Concurrent Case



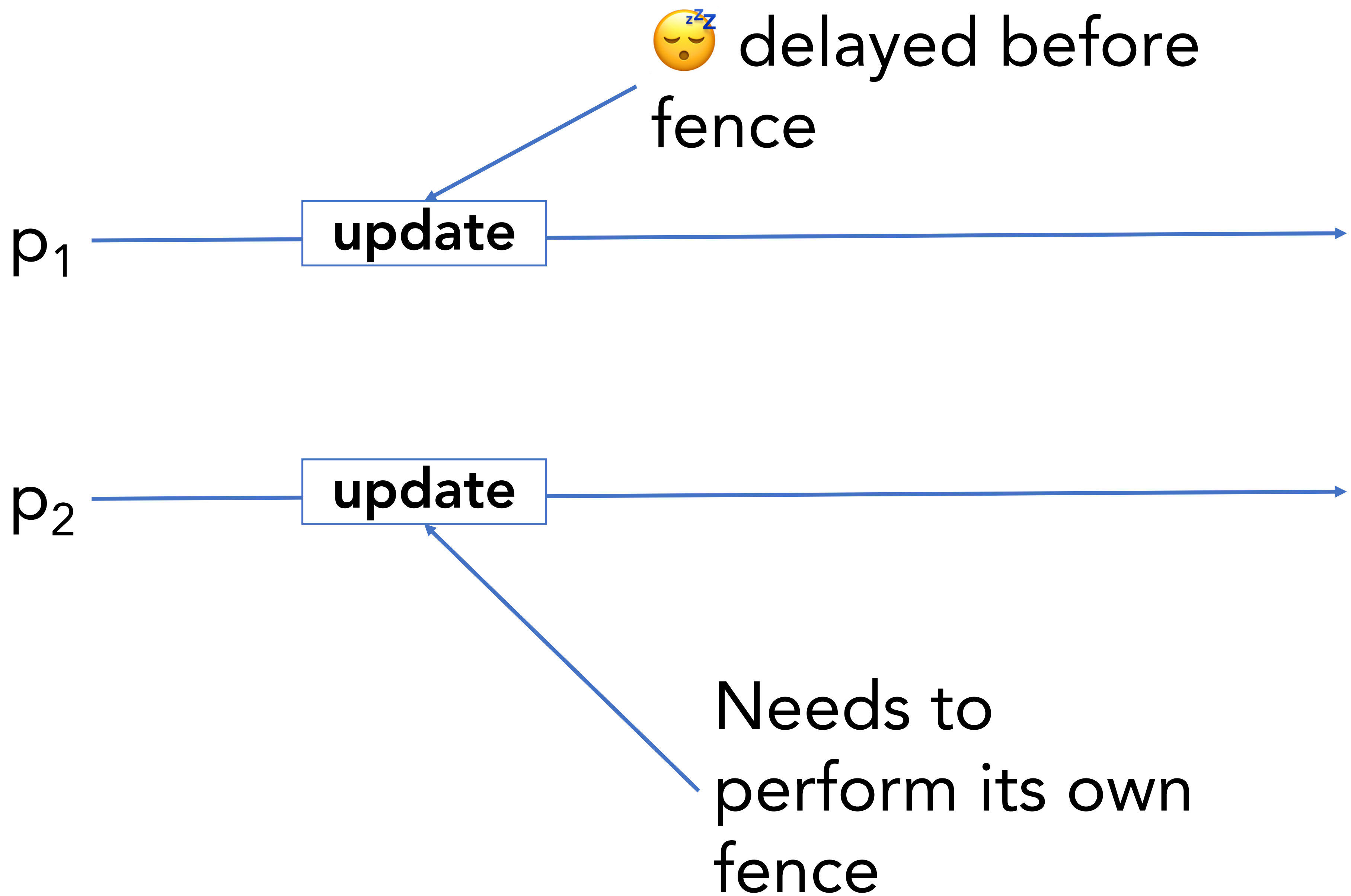
Lower Bound: Concurrent Case



Lower Bound: Concurrent Case



Lower Bound: Concurrent Case



Both processes perform one fence per update operation.