New Technologies in Concurrent Algorithms

Igor Zablotchi

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

[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

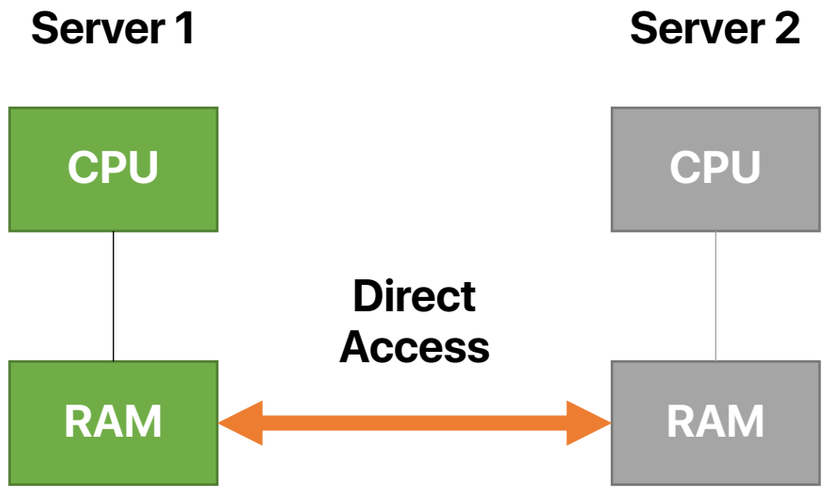
RDMA

Outline

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

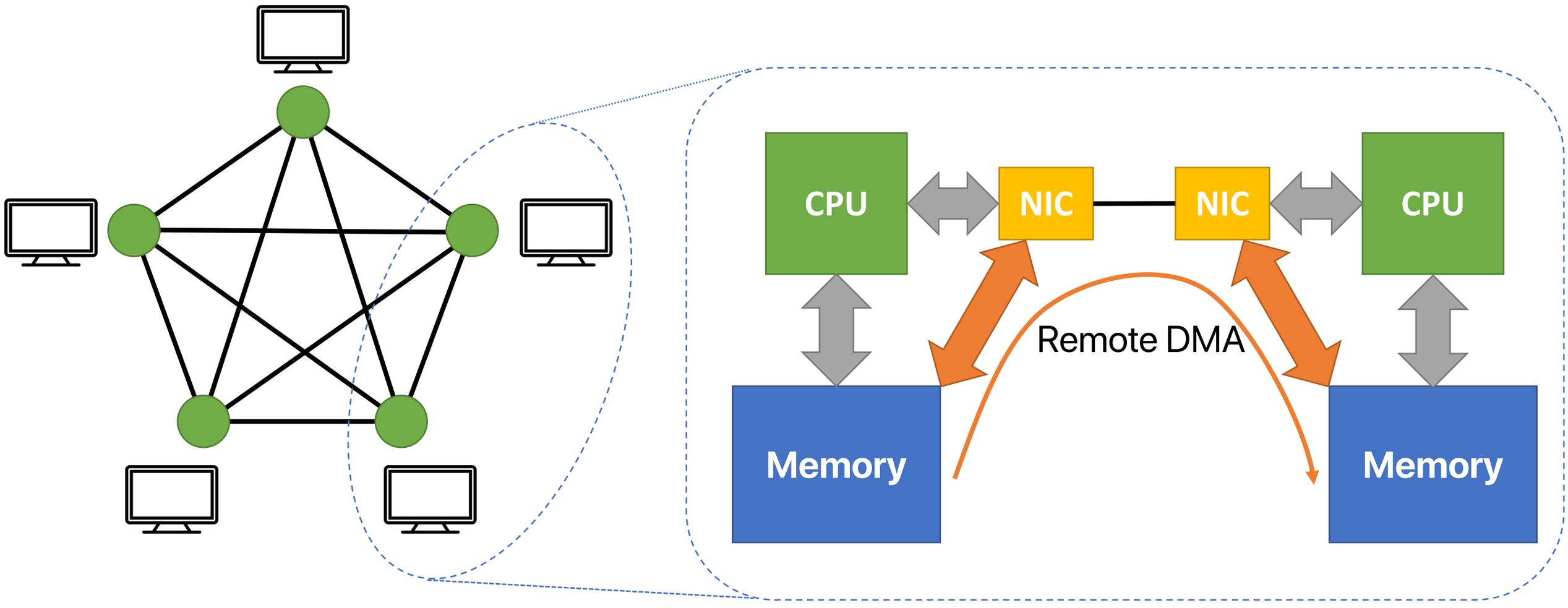
RDMA: Overview

- Networking hardware feature
- Direct access to remote memory
  - No CPU at remote side
  - No OS at either side
- Good performance
  - ~1us latency
  - ~100Gbps bandwidth
- Configurable access permissions

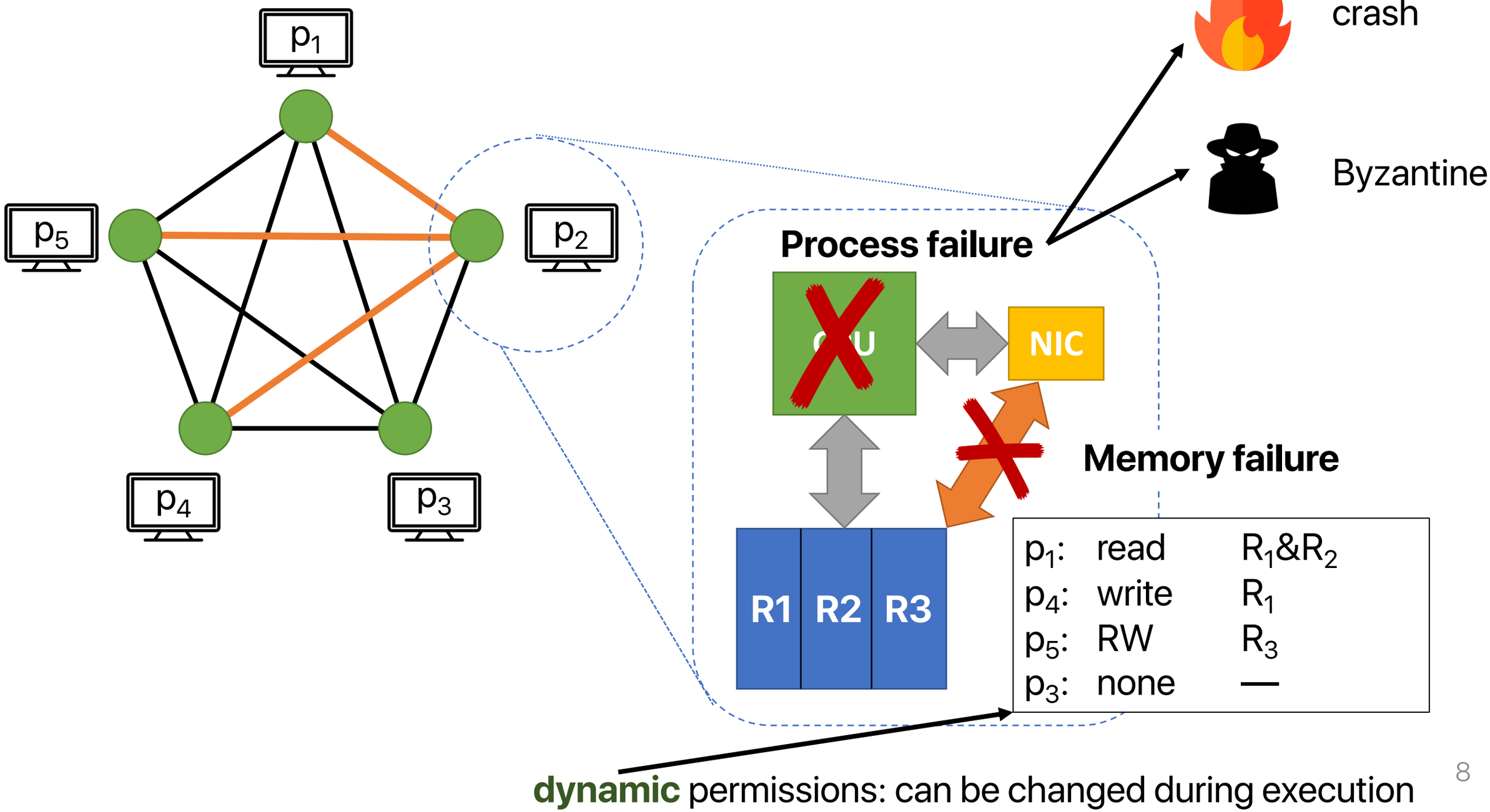


RDMA

Remote Direct Memory Access (RDMA)



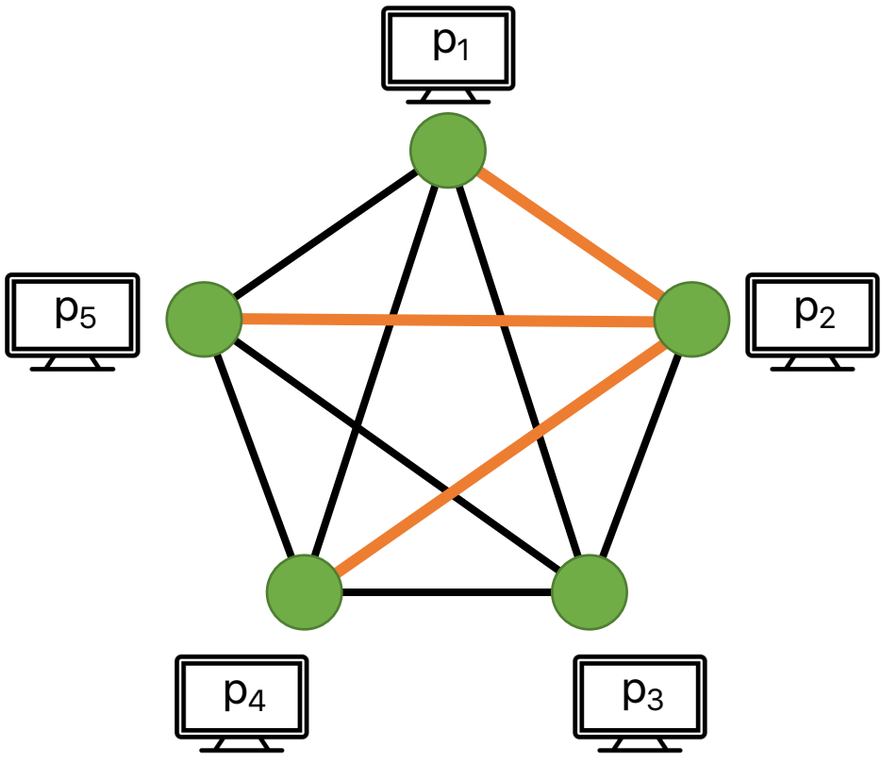
RDMA: Permissions and Failures

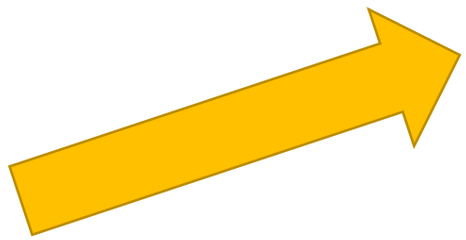


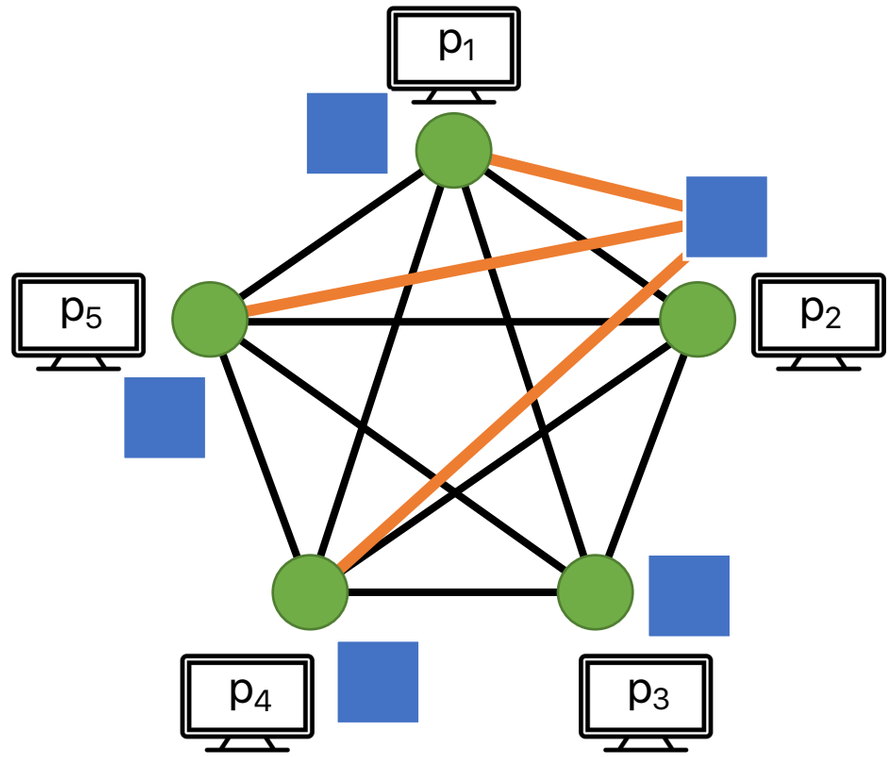
Outline

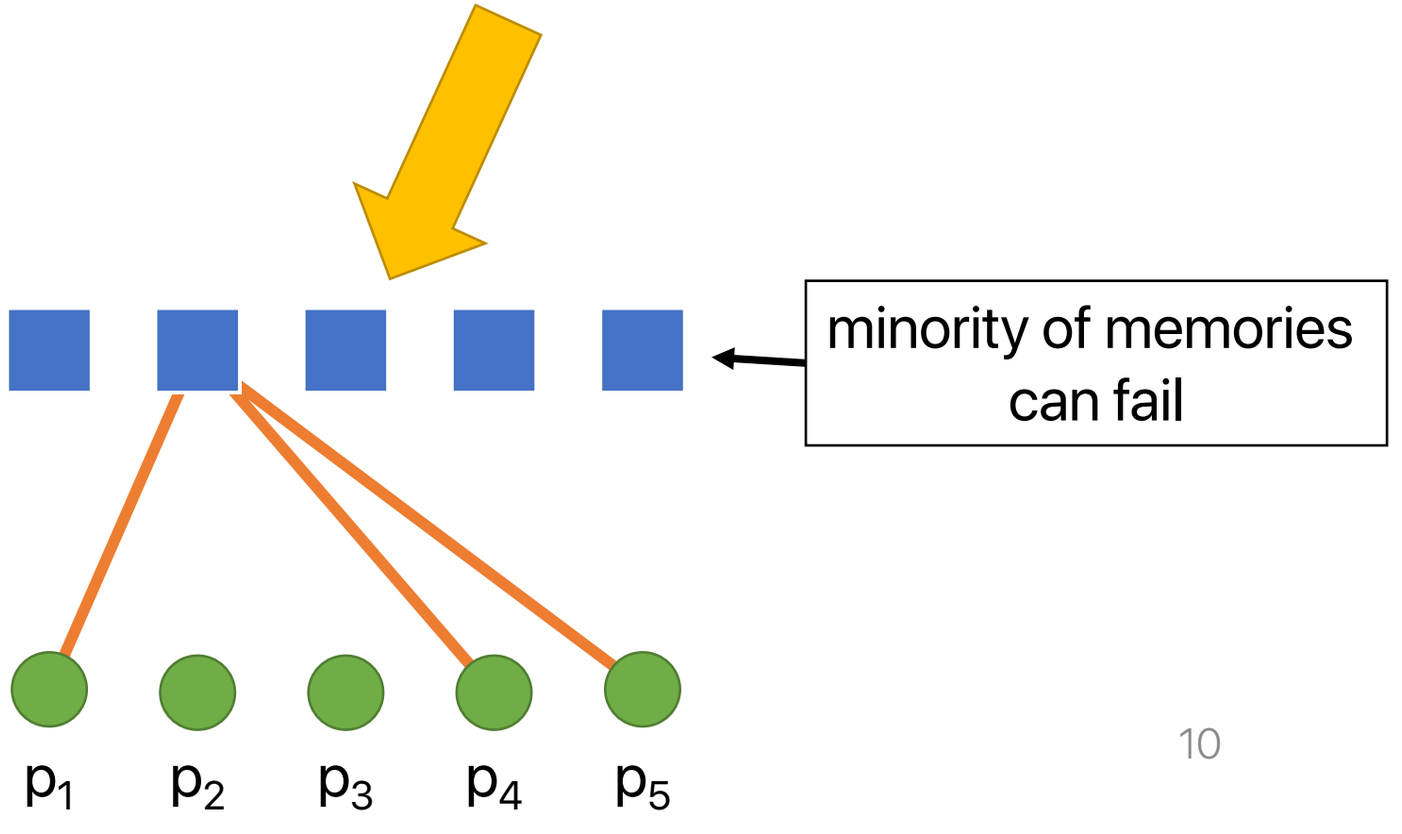
- What is RDMA?
- How we model RDMA
- Notable Results: consensus with RDMA
  - Crash faults
  - Byzantine faults

Modelling RDMA









Outline

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

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

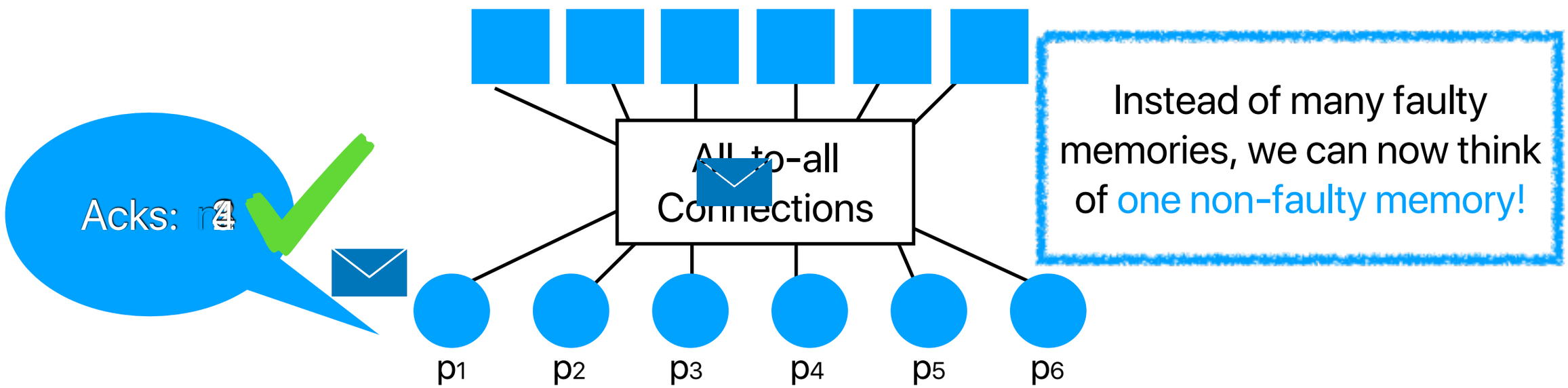
This assumes that shared memory never fails.

🤔 **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



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

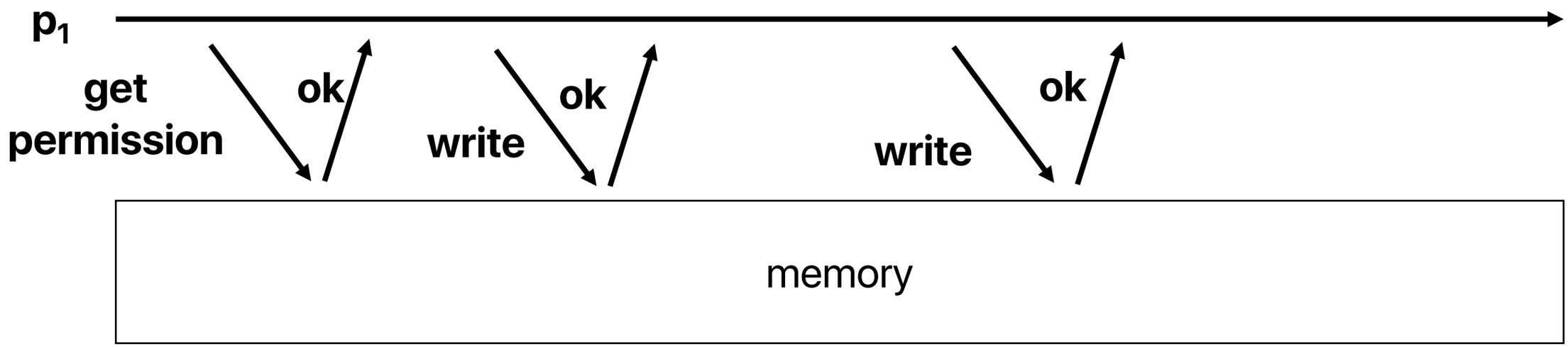
- If we don’t read again, we might miss a concurrent process’s timestamp
- This could lead to violation of agreement

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

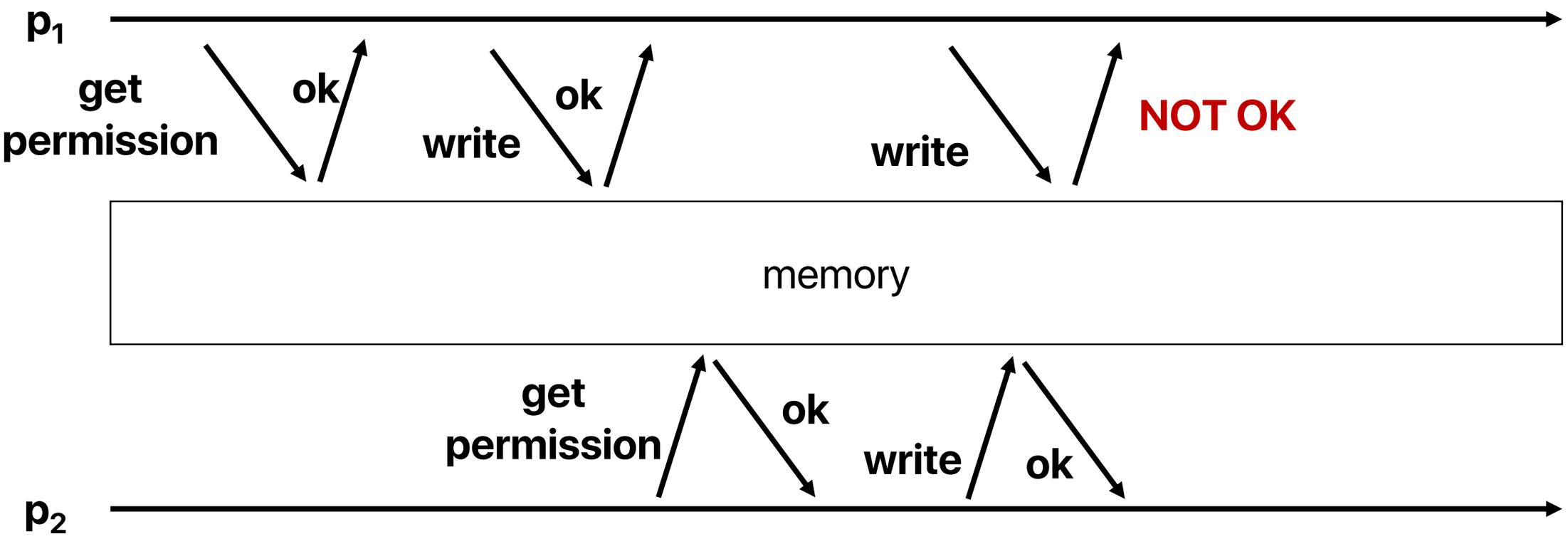
→ better complexity

Solo Detection w/ Permissions

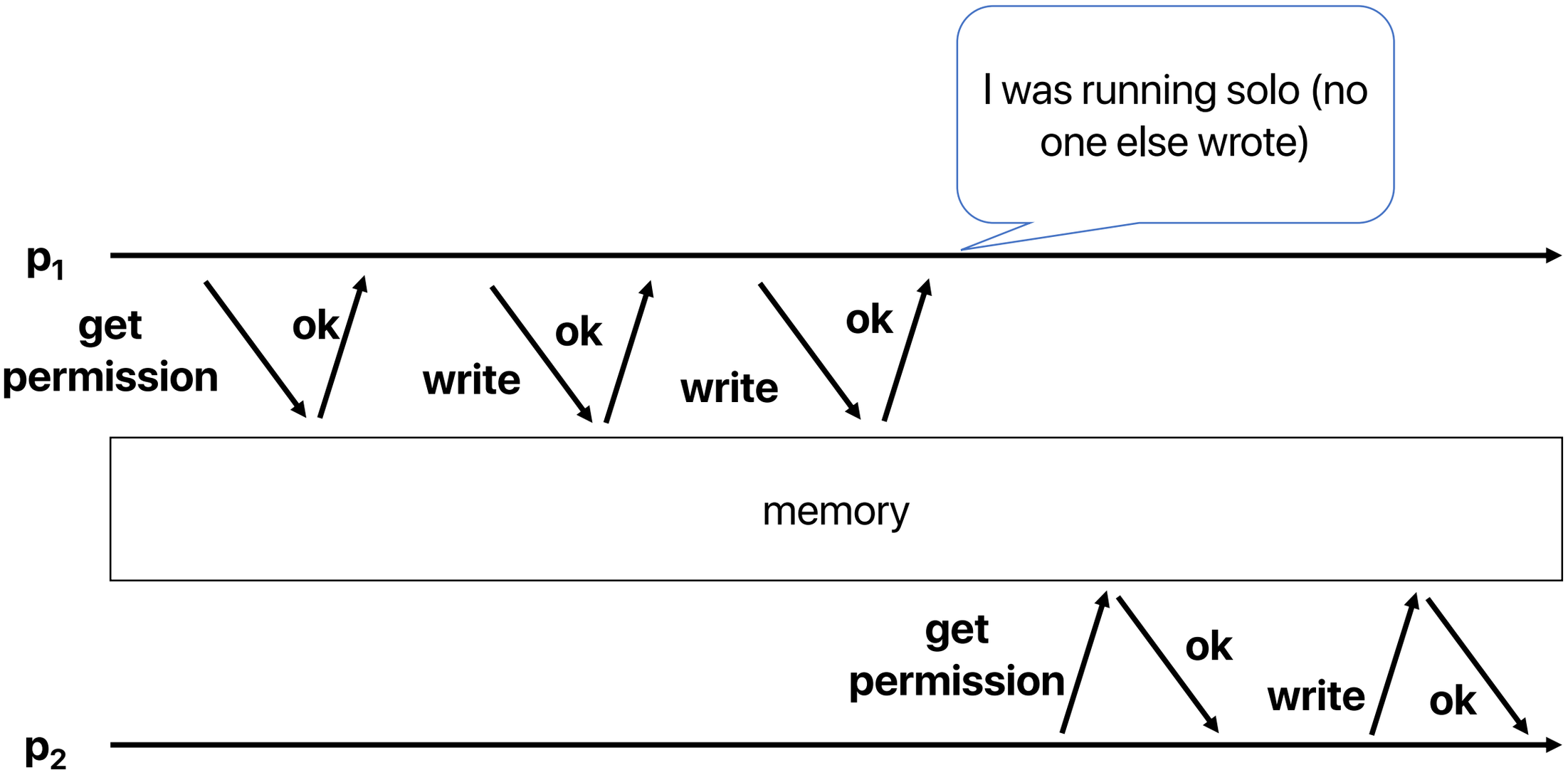
Idea: Memory gives write permission to the last process that requested it.
→ Only one process has write permission on a memory at any time.



Solo Detection w/ Permissions



Solo Detection w/ Permissions



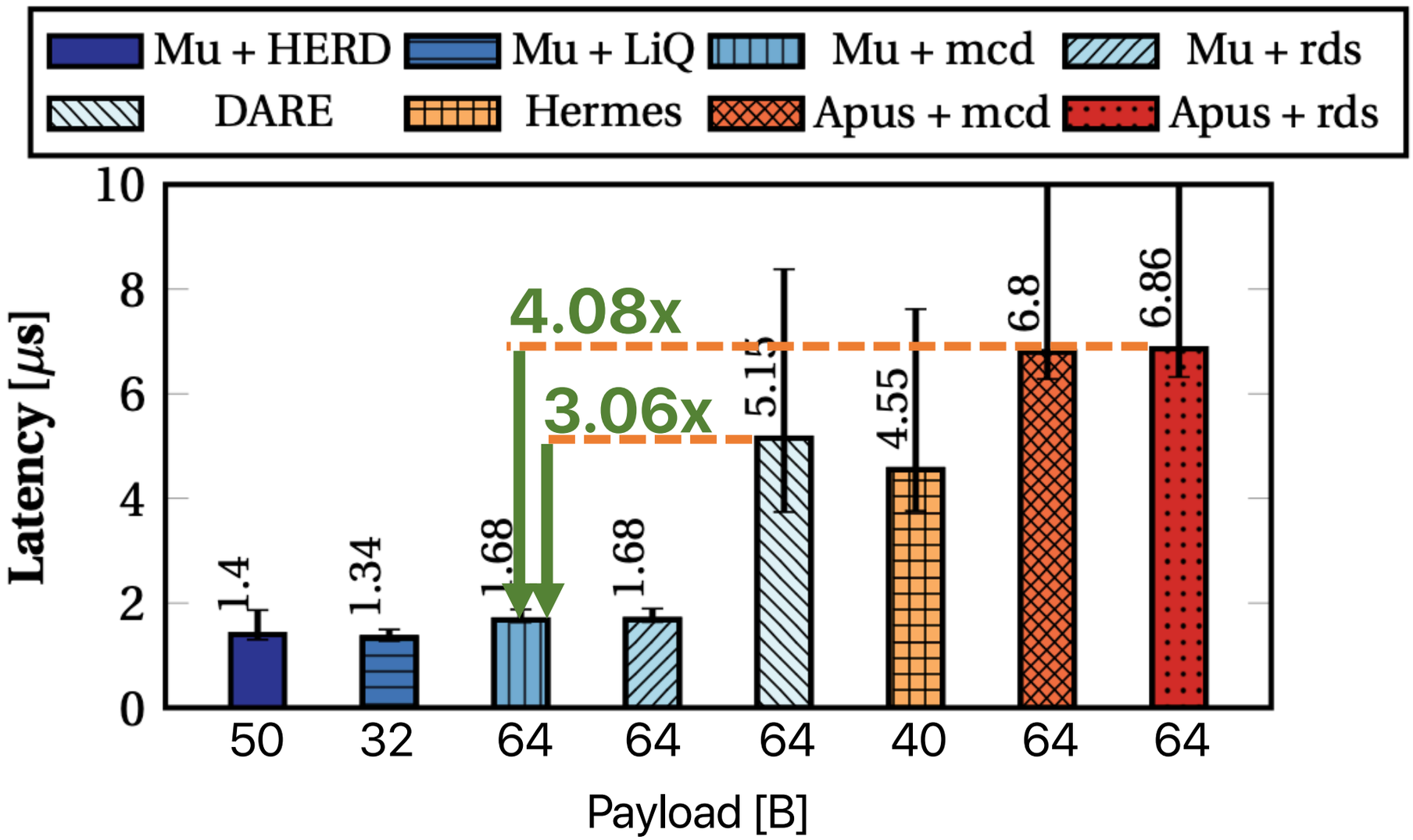
O-Consensus with Memory Failures and Permissions

```
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!

Quick Look: Replication Latency

[3x replication, 100Gbps Infiniband]

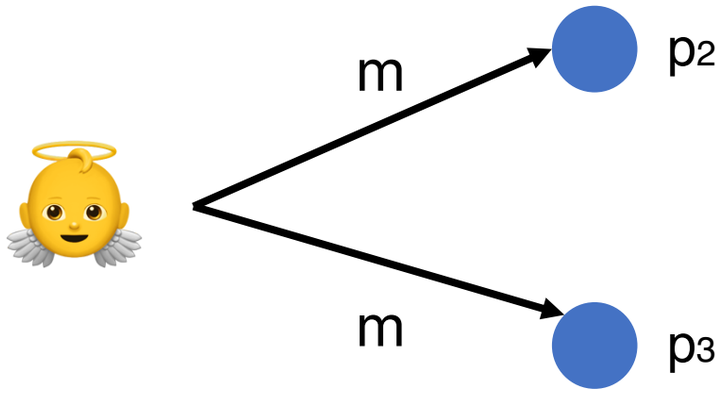


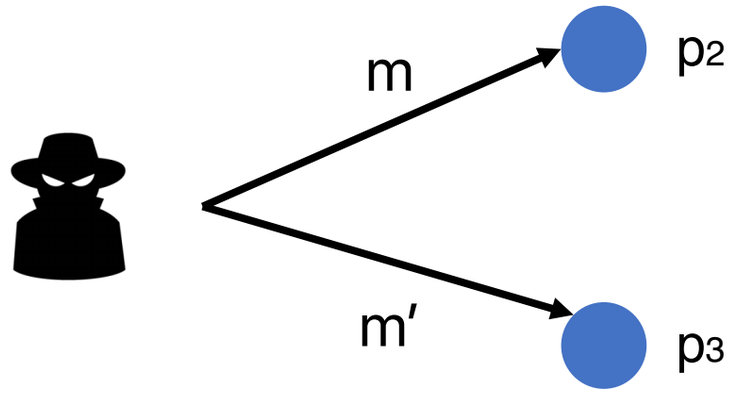
3-4x faster than state-of the art

Outline

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

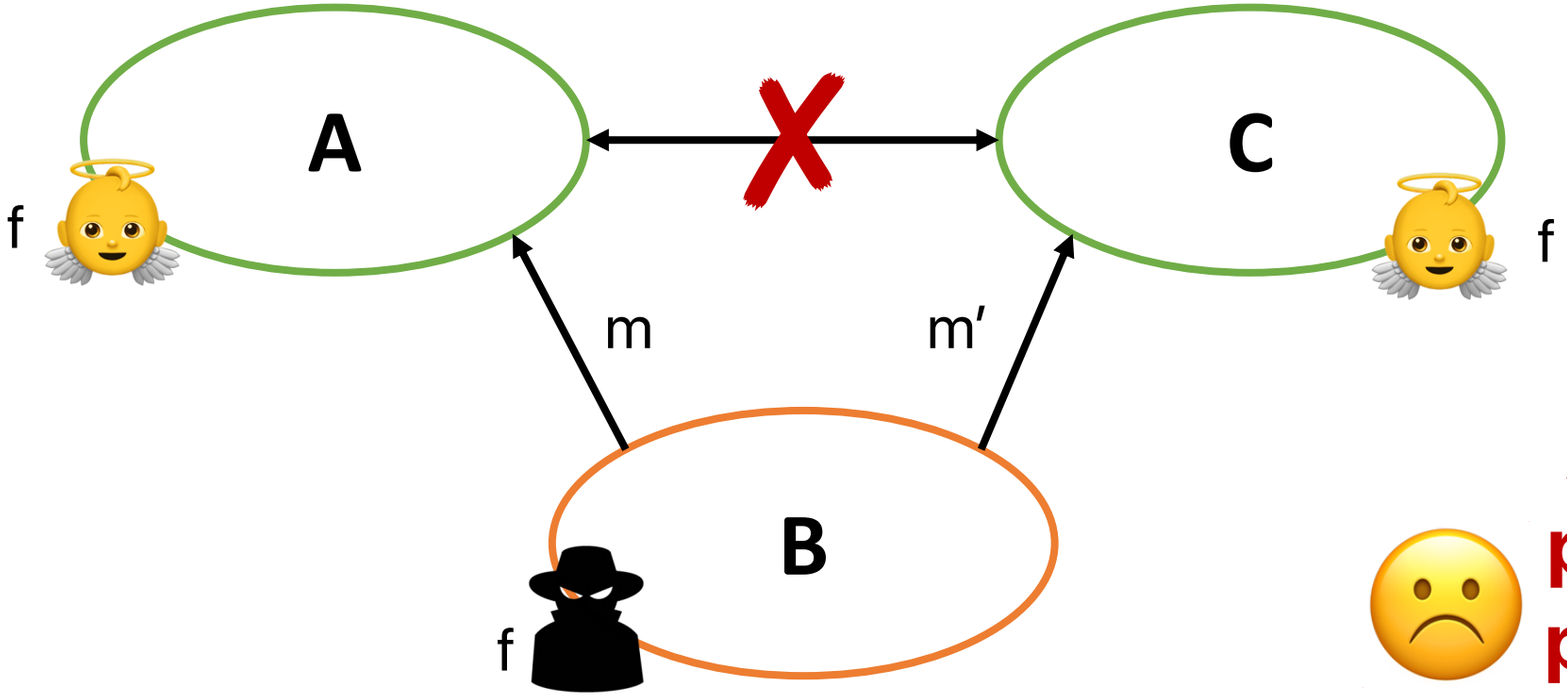
Equivocation





Preventing Equivocations in Message Passing

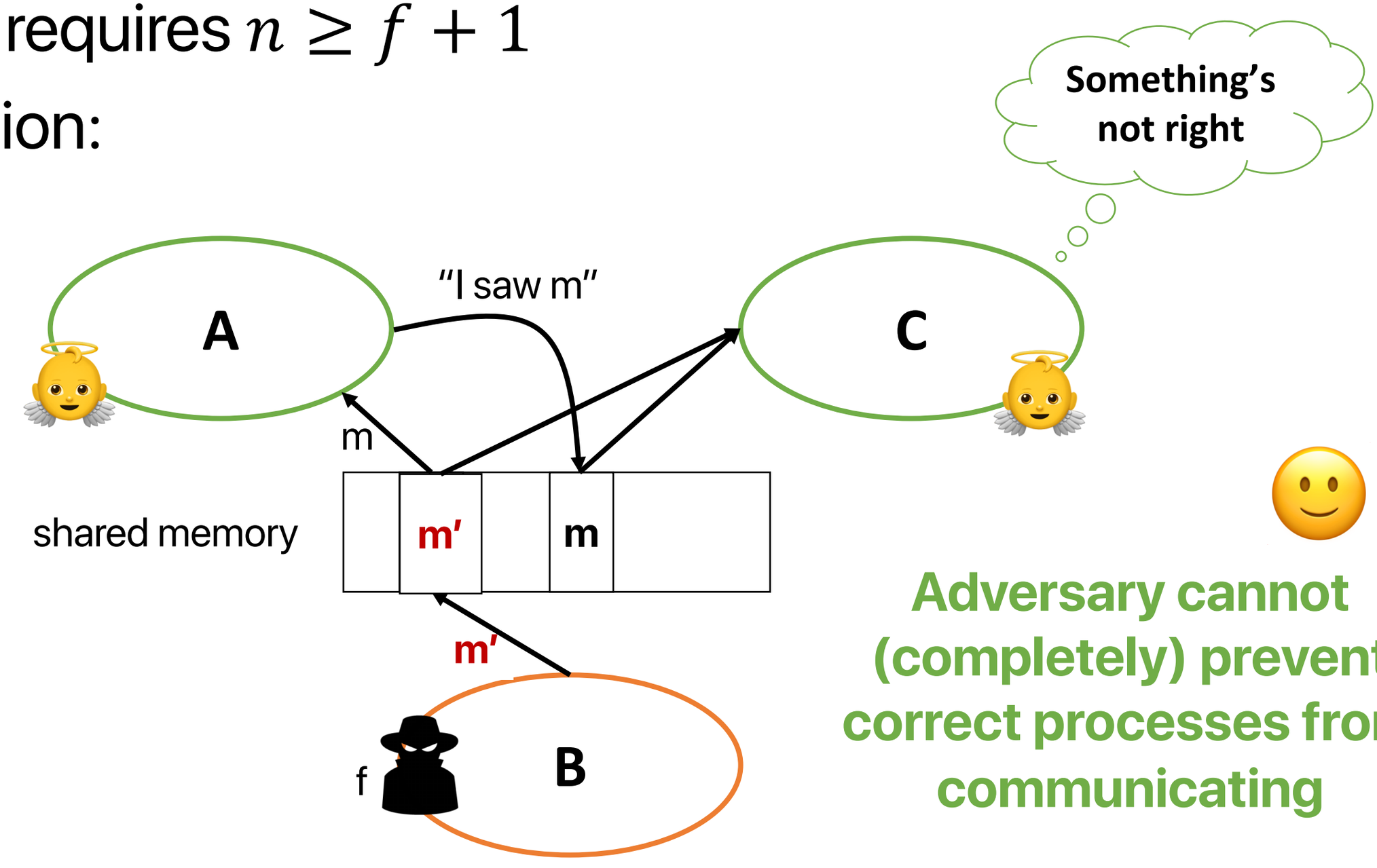
- Requires n=3f+1, where n is the total number of processes and up to f processes can be Byzantine
- Intuition:



Adversary can prevent correct processes from communicating

Preventing Equivocation in Shared Memory

- Only requires $n \geq f + 1$
- Intuition:



Adversary cannot (completely) prevent correct processes from communicating

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 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:
  - while (true)
    - senderMsg = R[s].read()
    - if (senderMsg == ⊥) then continue
    - 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)

Recap — RDMA

- What is RDMA?
- How we model RDMA
- Notable Results: consensus with RDMA
  - Crash faults: reducing communication complexity
  - Byzantine faults: “easy” non-equivocation

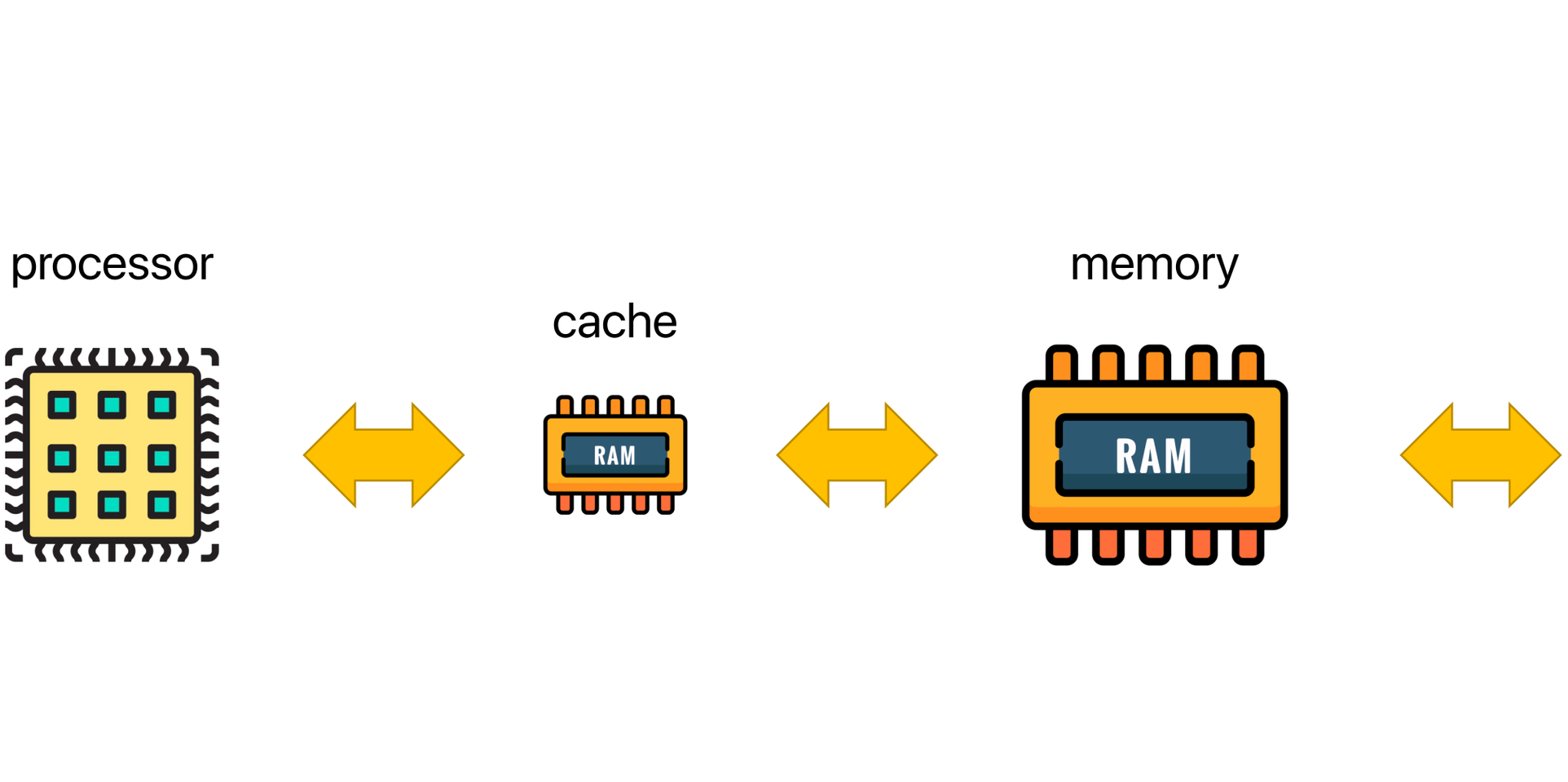
Part 2

Persistent Memory

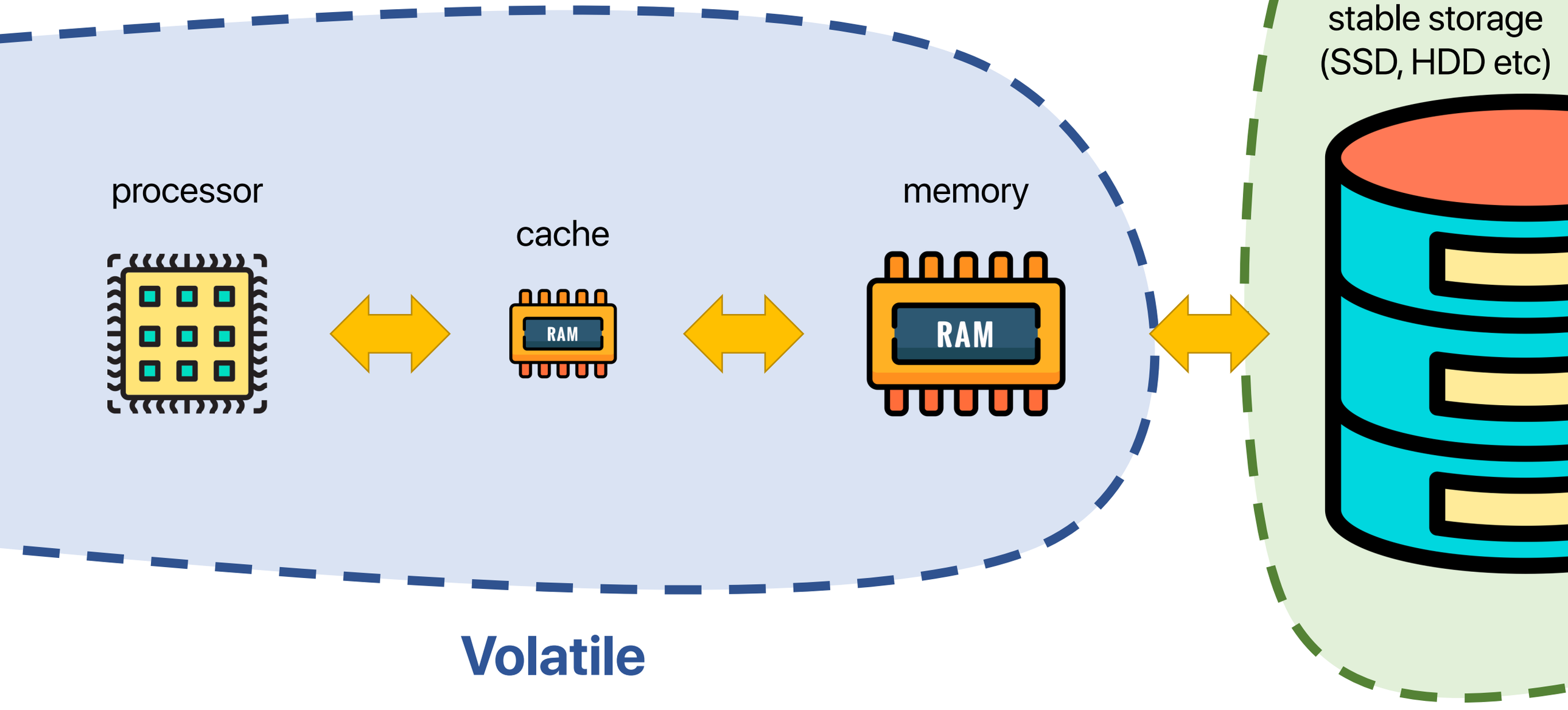
Outline

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

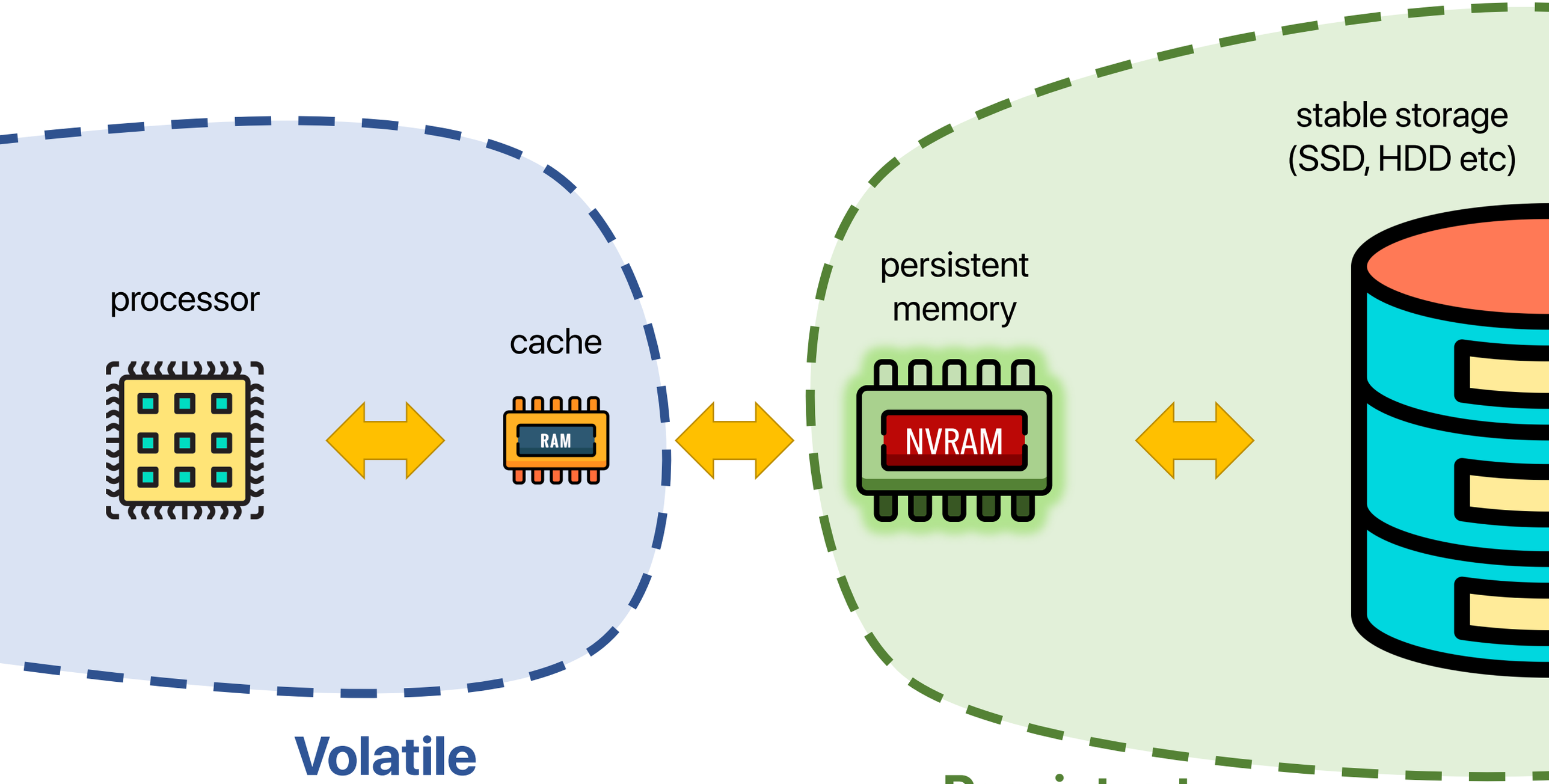
What is Persistent Memory?



What is Persistent Memory?

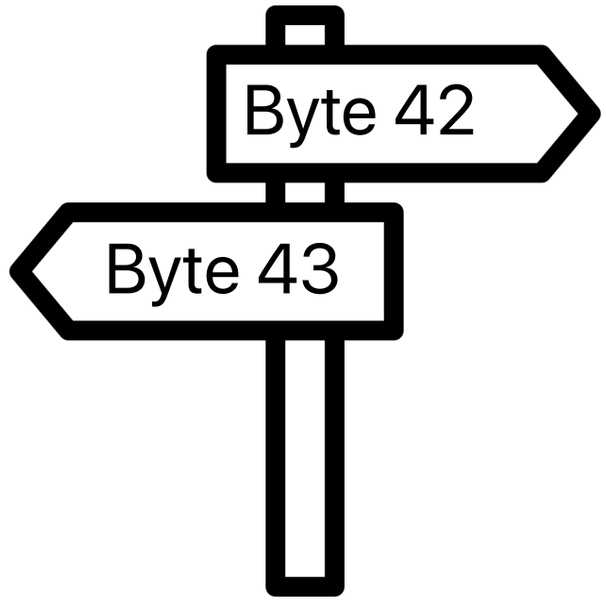


What is Persistent Memory?



What Is Persistent Memory?

Durability in the face of crashes & recoveries



Byte-addressability

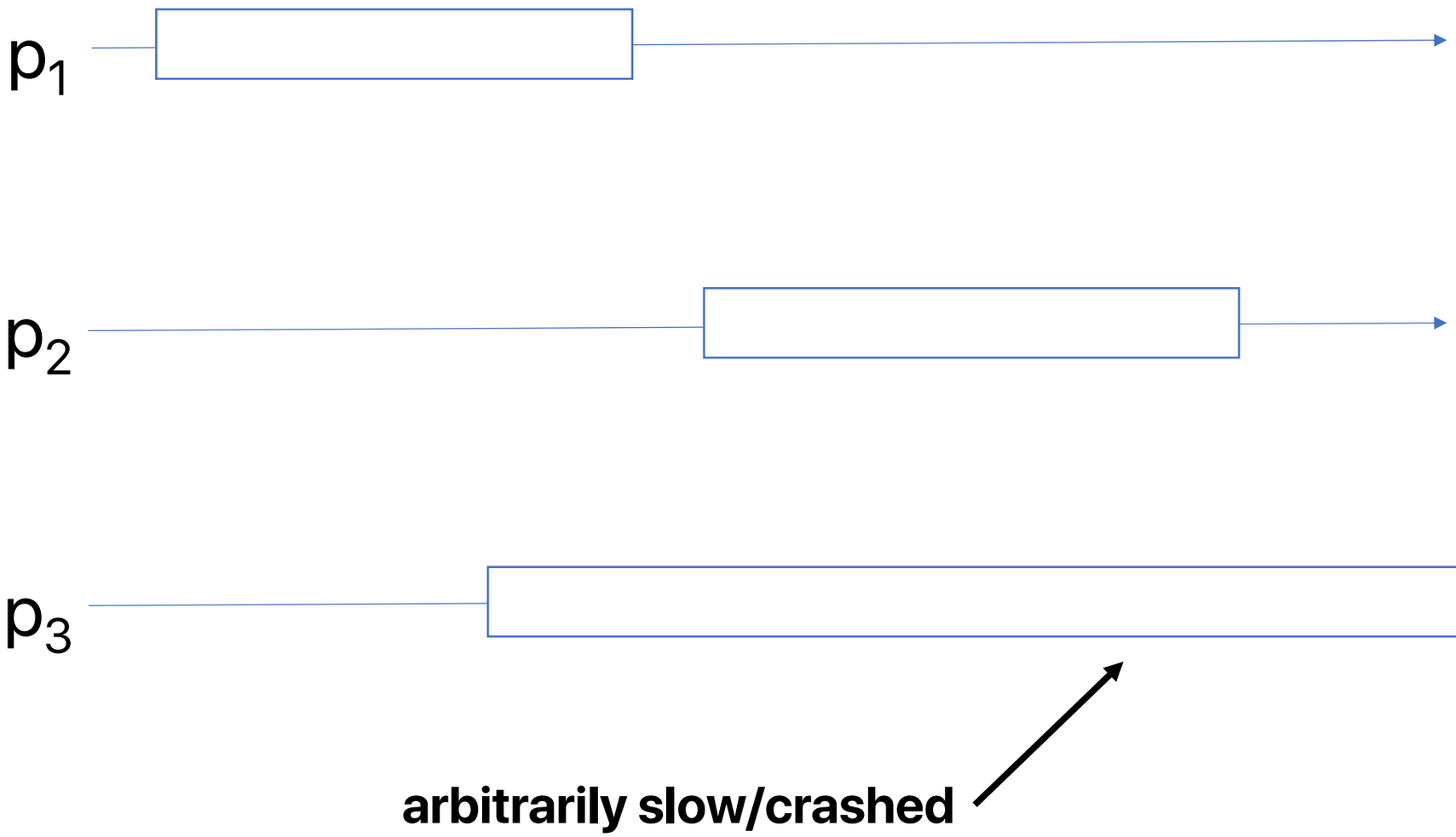
Access times ~ RAM

Outline

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

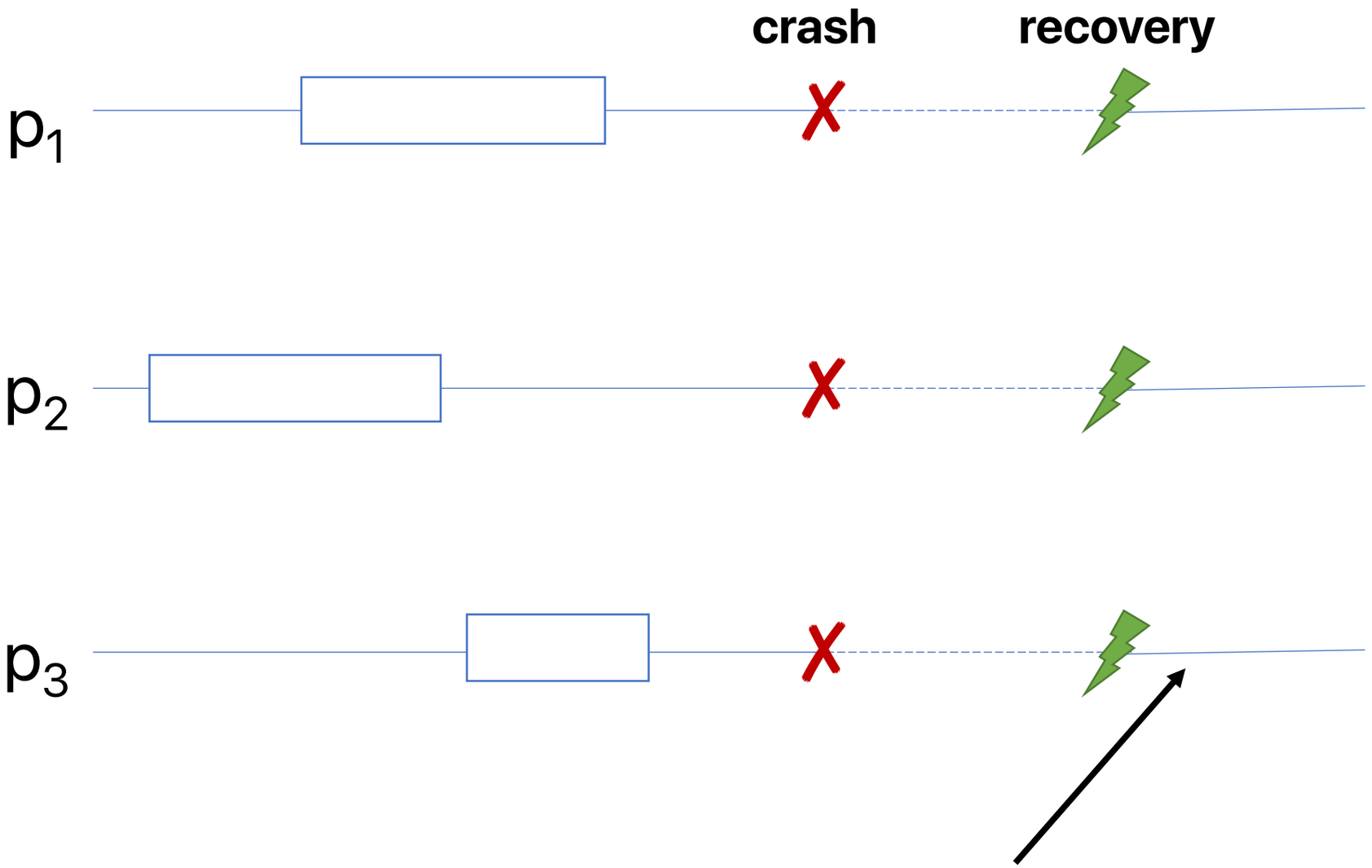
Modelling durability

Process delays & crashes



Modelling durability

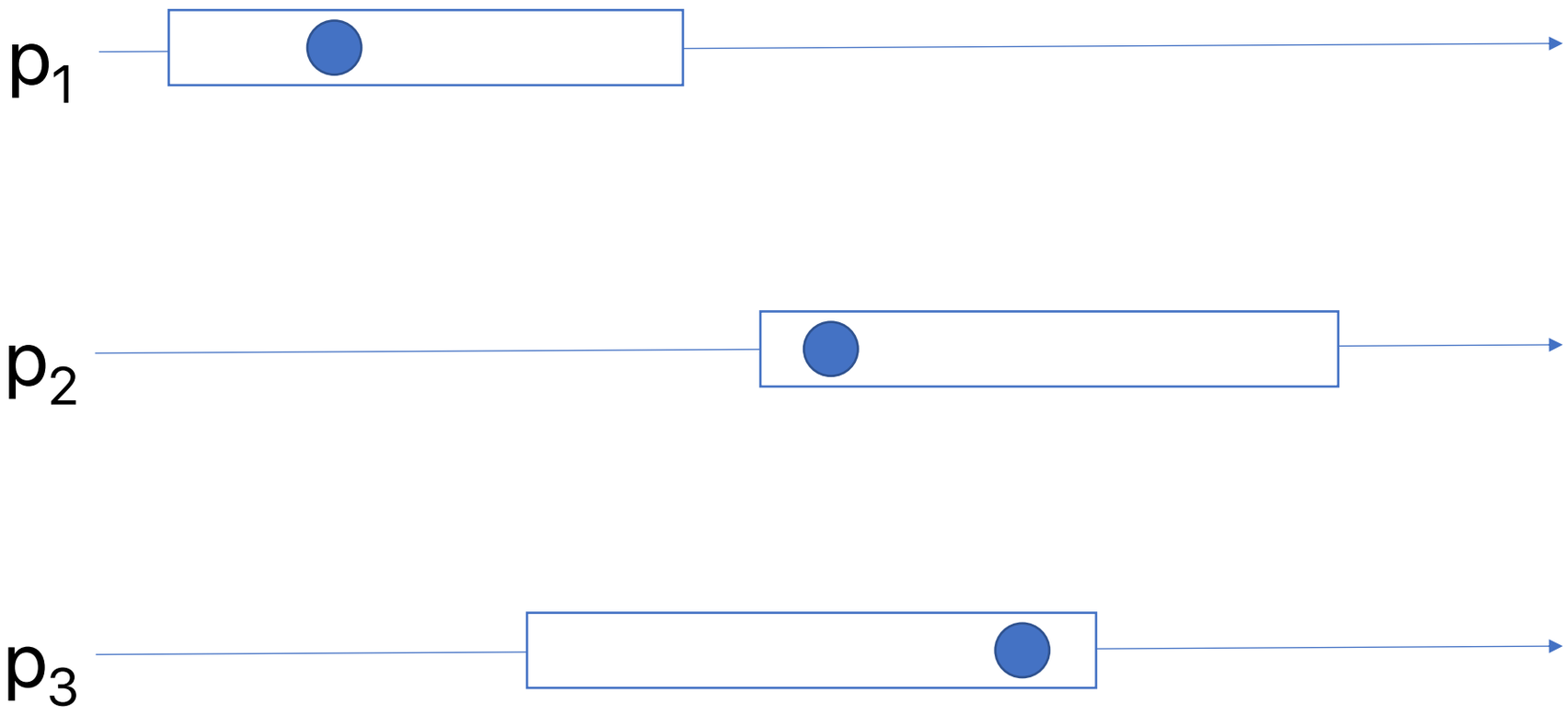
Full-system crash & recover



Recall: Atomicity

- Every operation appears to execute at some indivisible point in time (called linearization point) between its invocation and response

Recall: Atomicity



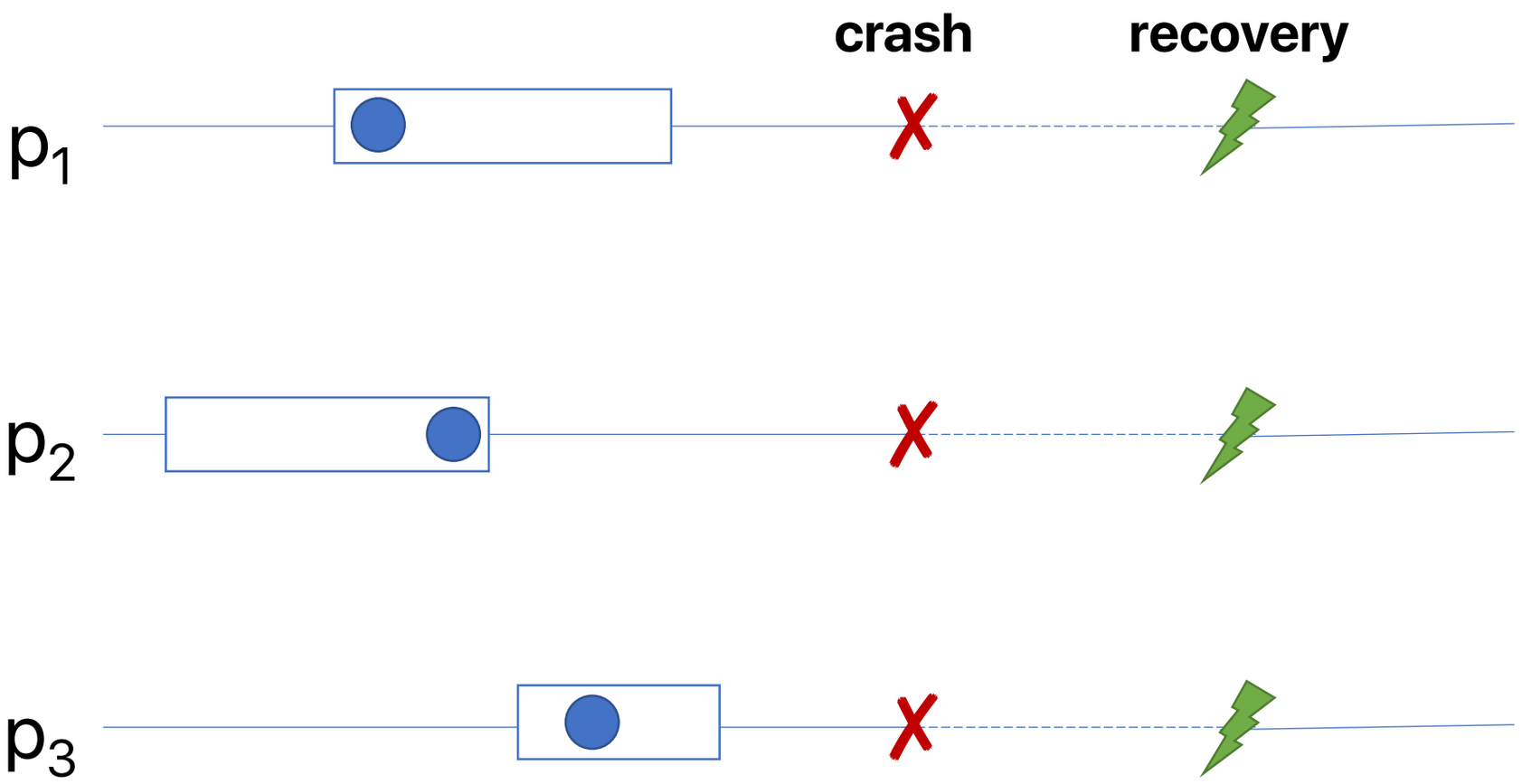
Atomicity & Persistent Memory

- How can we express atomicity in this model?

→ durable linearizability

Modelling durability

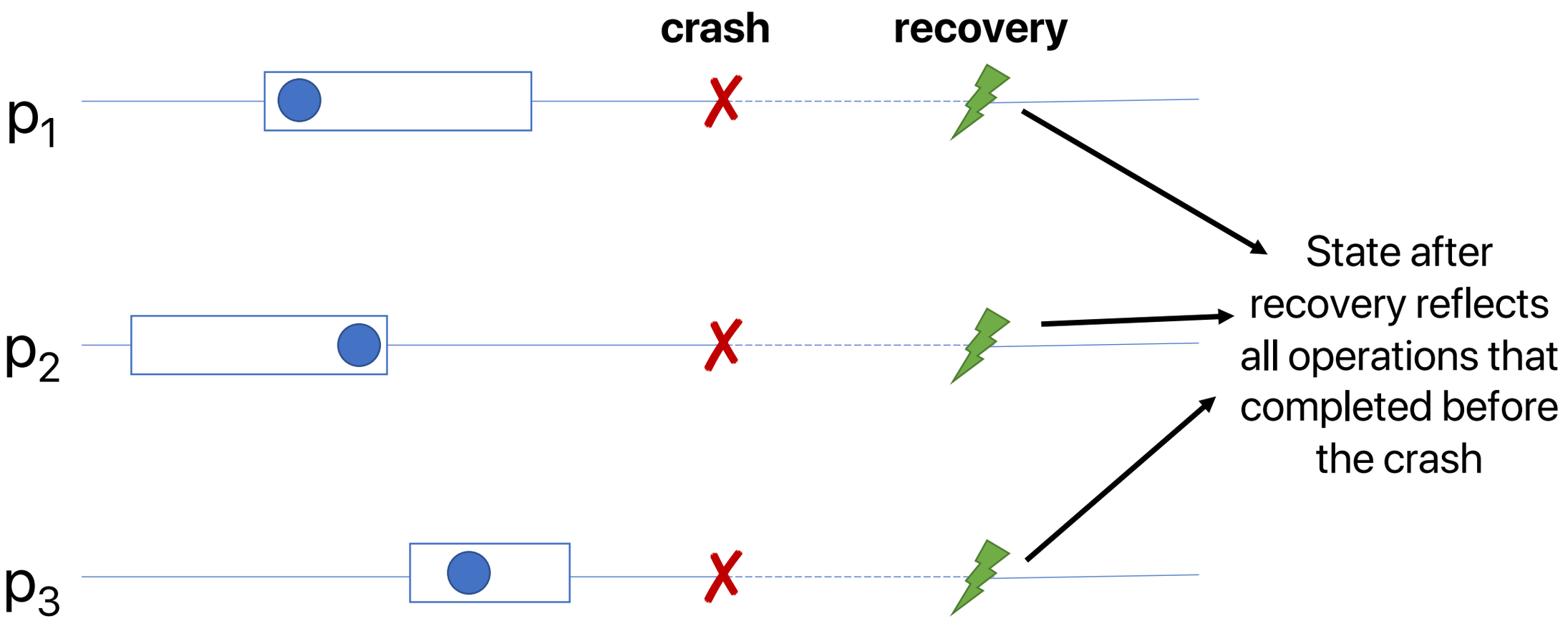
Durable Linearizability



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

Modelling durability

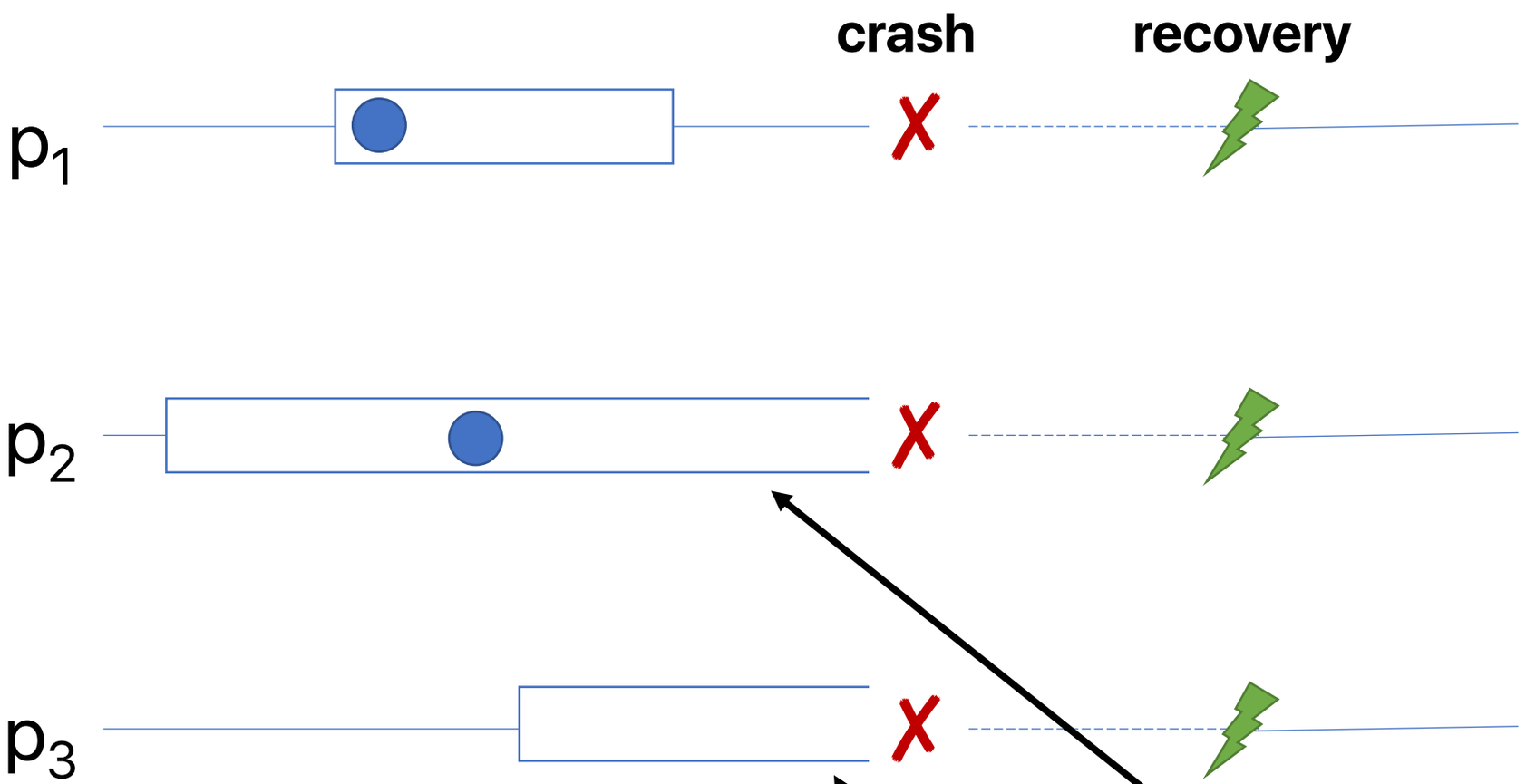
Durable Linearizability



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

Modelling durability

Durable Linearizability

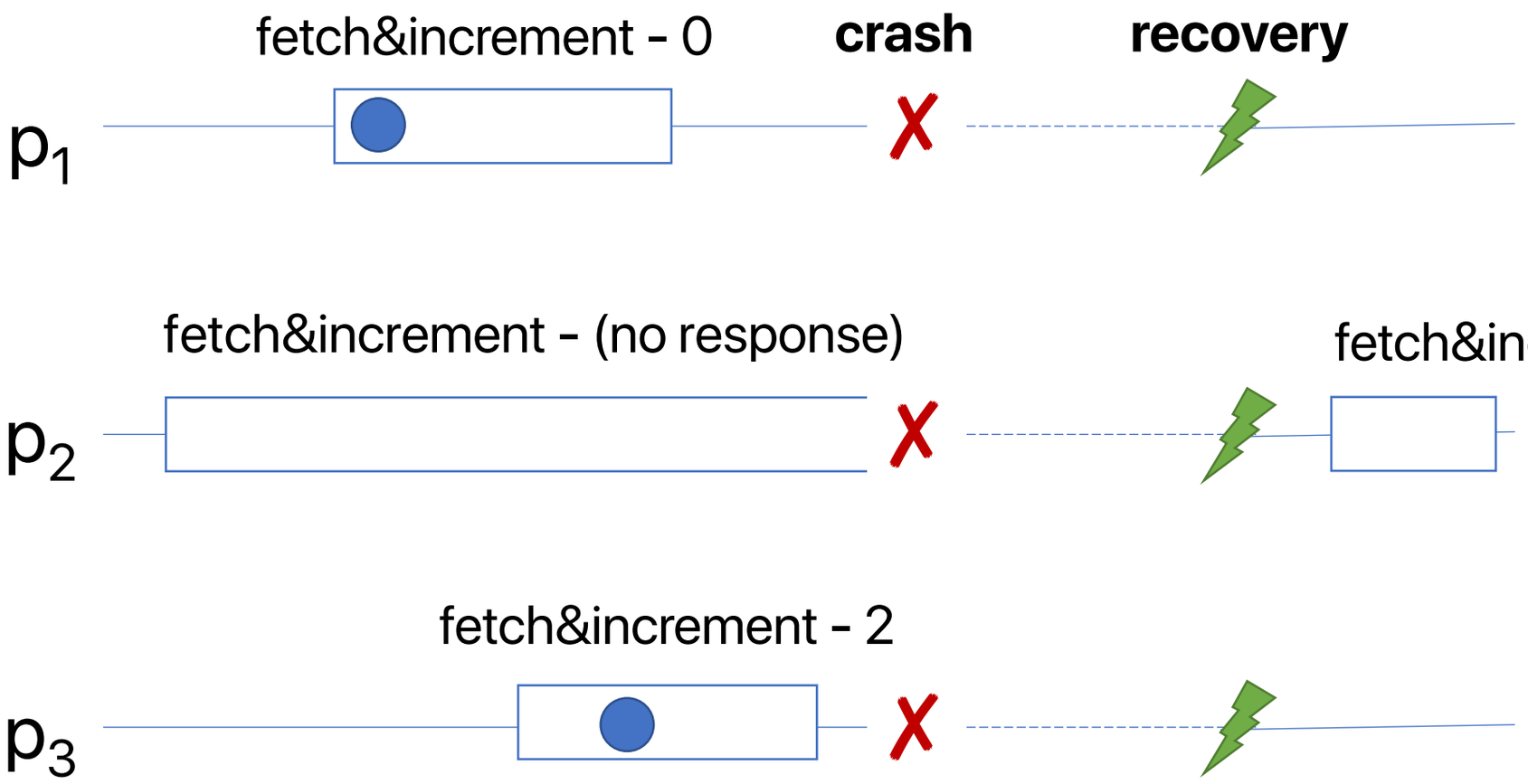


Operations that were ongoing during the crash may be kept (reflected in post-recovery state) or lost (not reflected)

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

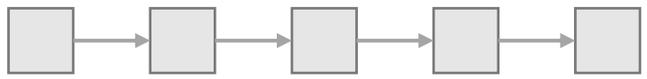

fetch&increment - 0
crash
recovery
p_1
fetch&increment - (no response)
fetch&increment - 3
p_2
fetch&increment - 2
p_3

Outline

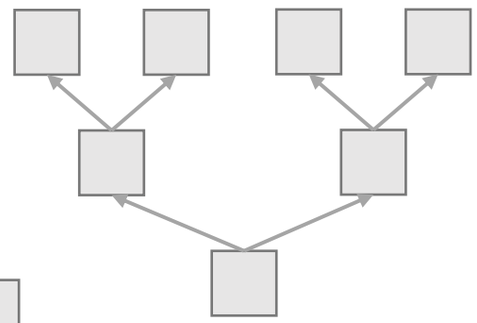
- What is persistent memory?
- How to define correctness for PM?
- Data Structures for PM
- A Lower Bound for PM

Concurrent Data Structures

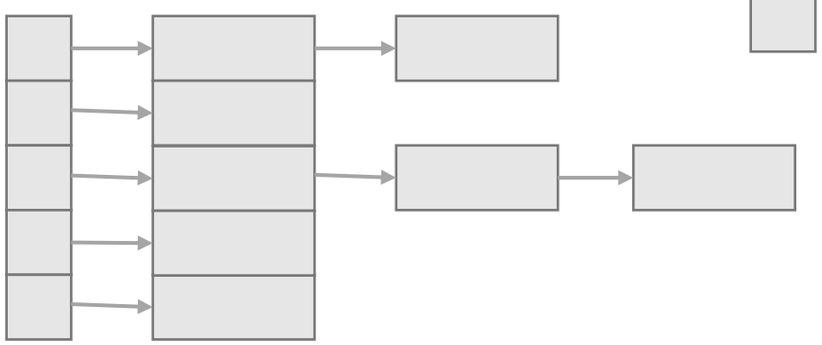
Lists



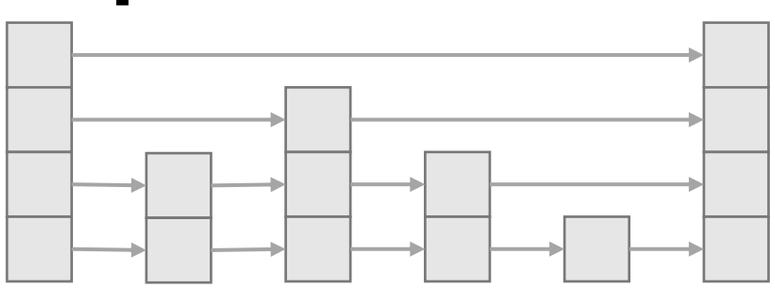
Trees

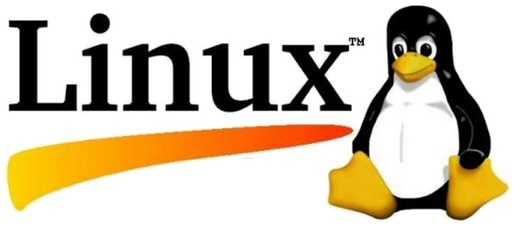


Hash tables

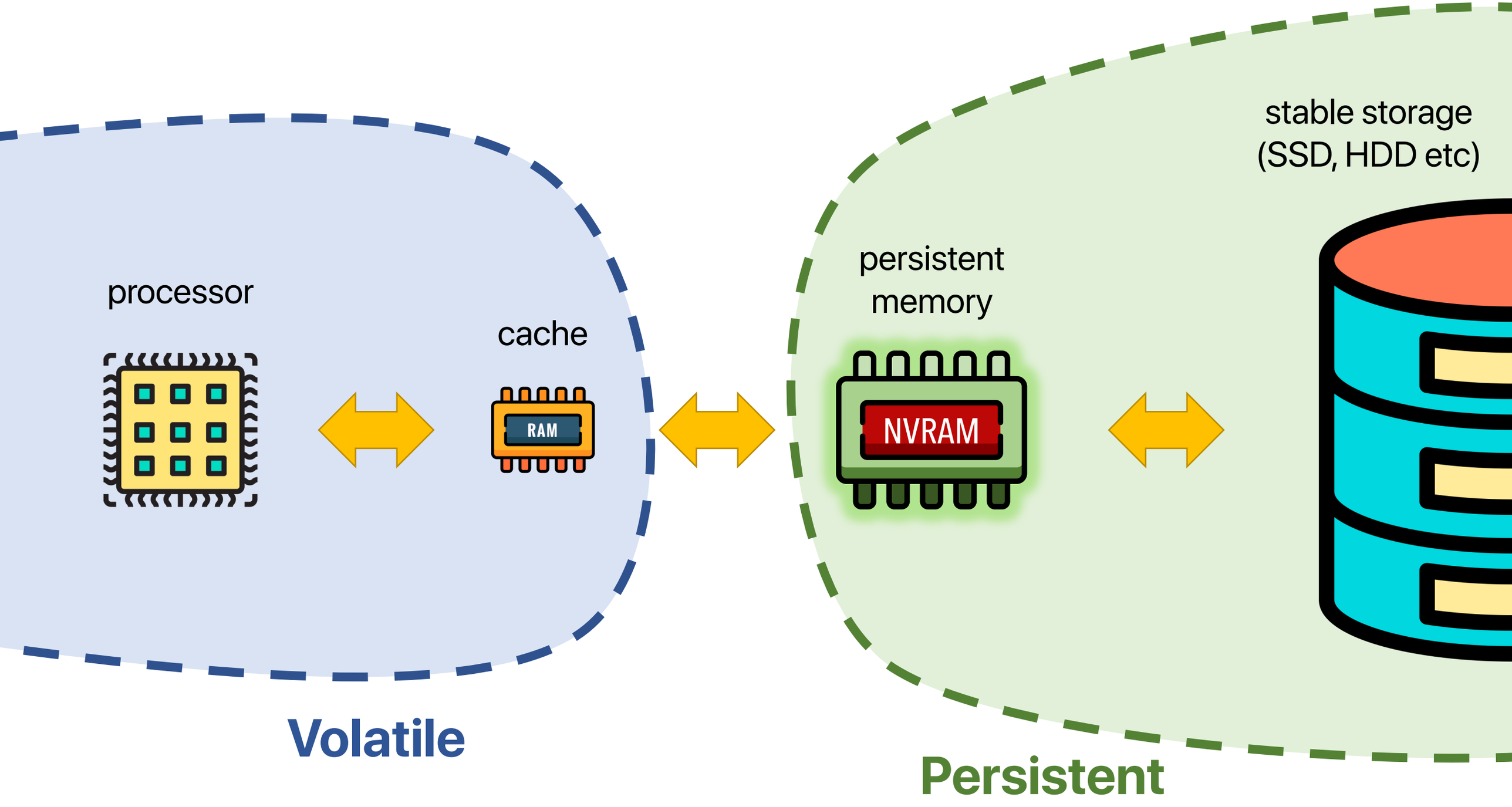


Skip lists

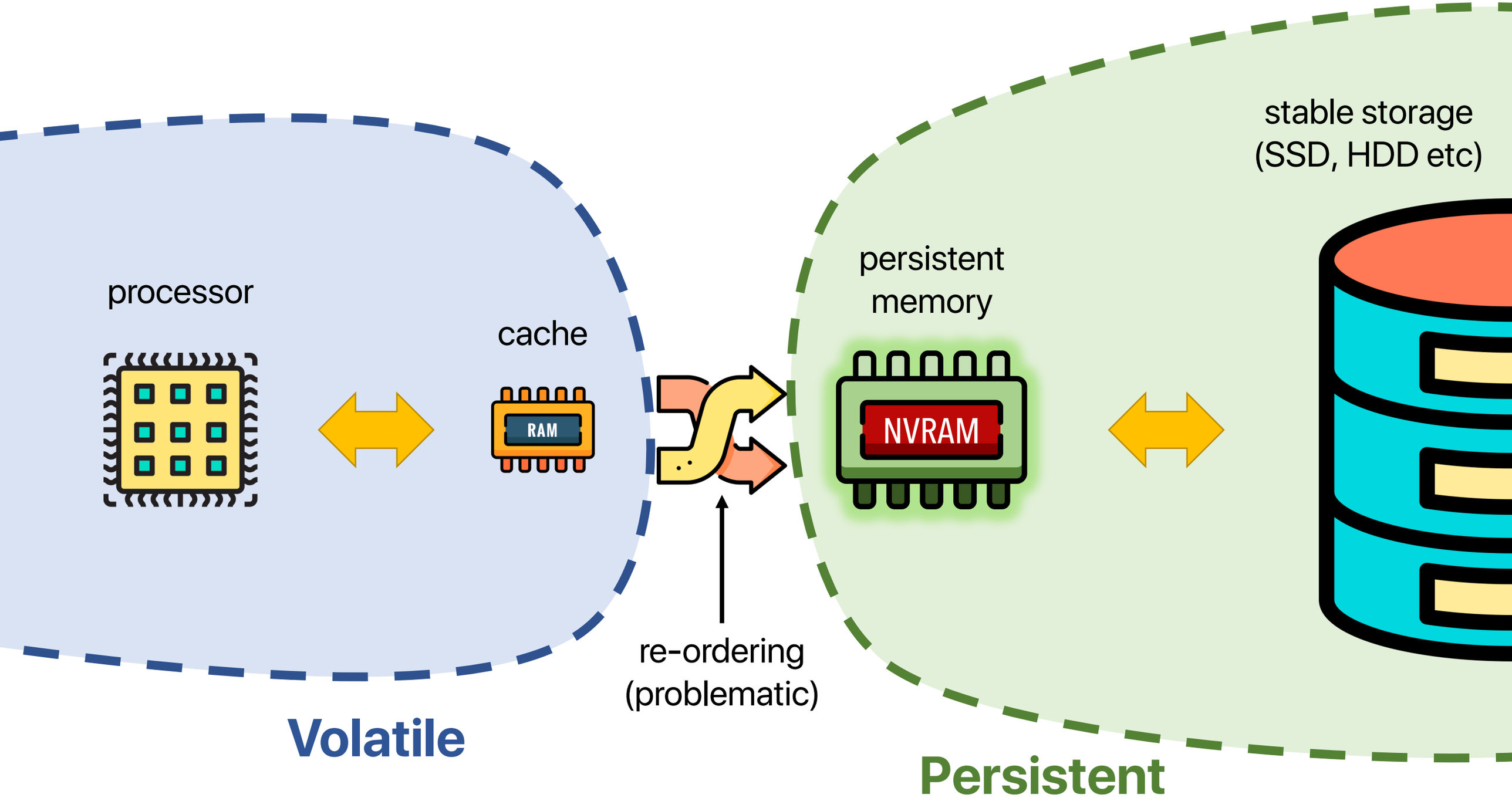




Challenge #1: Caches are Volatile



Challenge #2: Re-ordering



Challenges 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

Flushes & Fences

- Instructions that address PM challenges
- Cache-line Flushes
  - Asynchronously flush cache line contents to PM
  - E.g., clflushopt, clwb on Intel x86
- Persistent Fences
  - Stall until any pending flushes complete
  - E.g., sfence, LOCK-prefixed instructions on x86
- Fences dominate cost of durability

Challenges 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
```


crash

NV memory:

```
3: change link 1

5: done = 1
```

Upon restart: incorrect state

Challenges 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
```


crash

NV memory:

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

Upon restart: incomplete operation

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

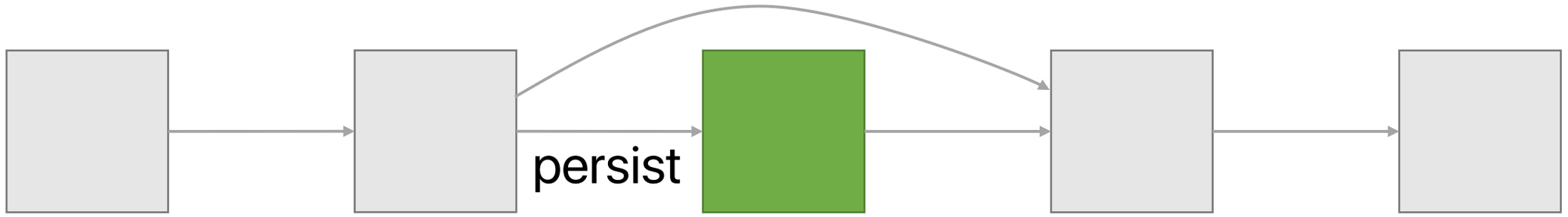
The Problem with Logging

- 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

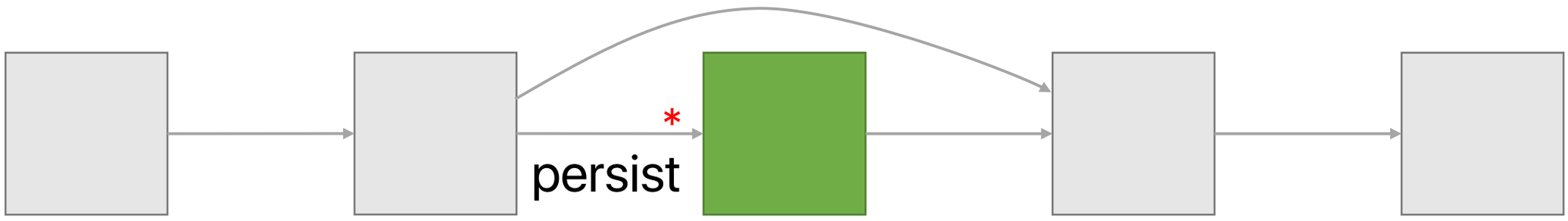
- After a restart, the structure reflects:
  - all operations completed (linearized) before the crash;
  - (potentially) some operations that were ongoing when the crash occurred;



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

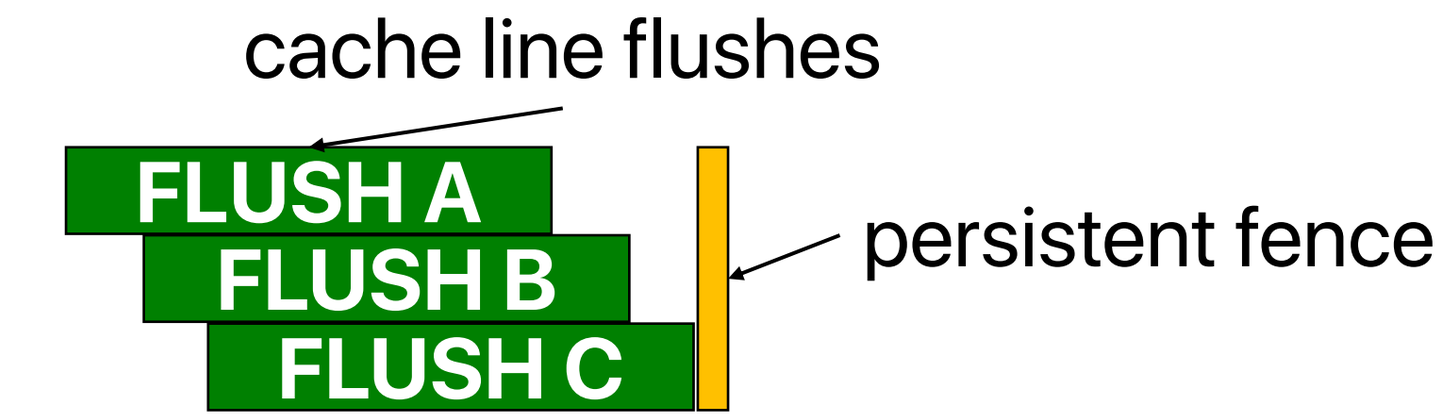


1. Persistently allocate and initialize node
2. Add marked link to new node
3. Persist link to new node
4. Remove mark

Other threads - persist marked link if needed

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

Going Further: Batching



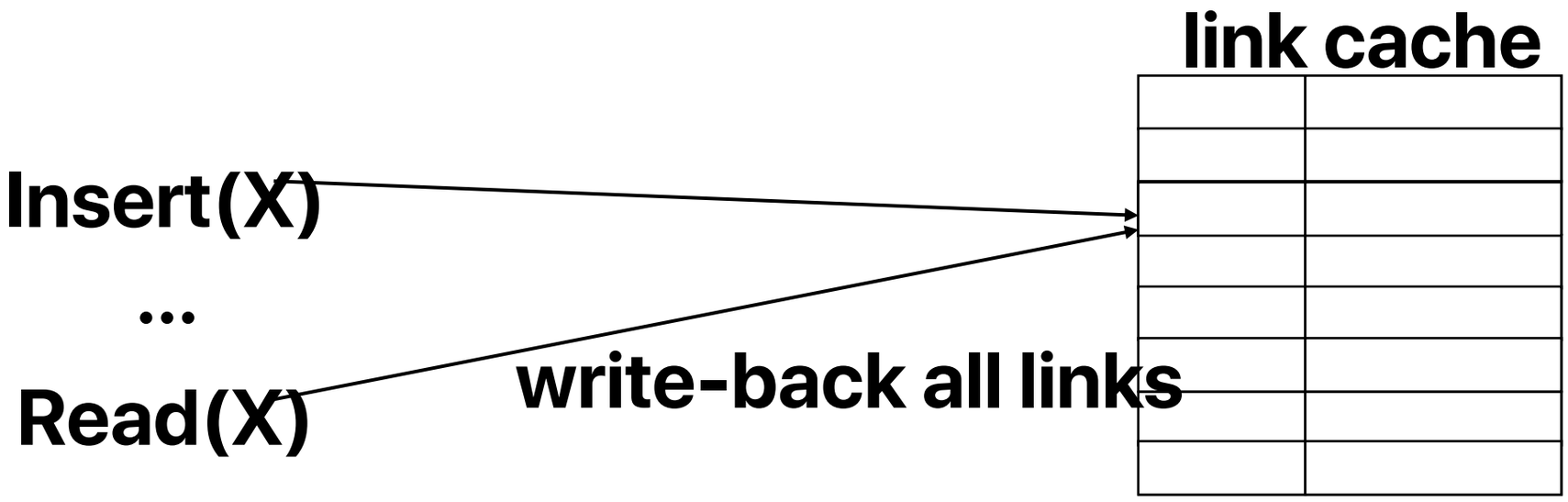
Batching flushes:
beneficial for performance

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 flushes of all links in the cache

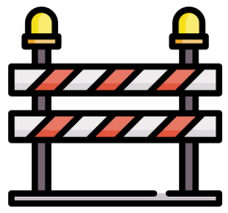
(and empty the cache)



Outline

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

Persistent Fences



- Persistent fences prevent re-ordering
- But they are expensive (slow)

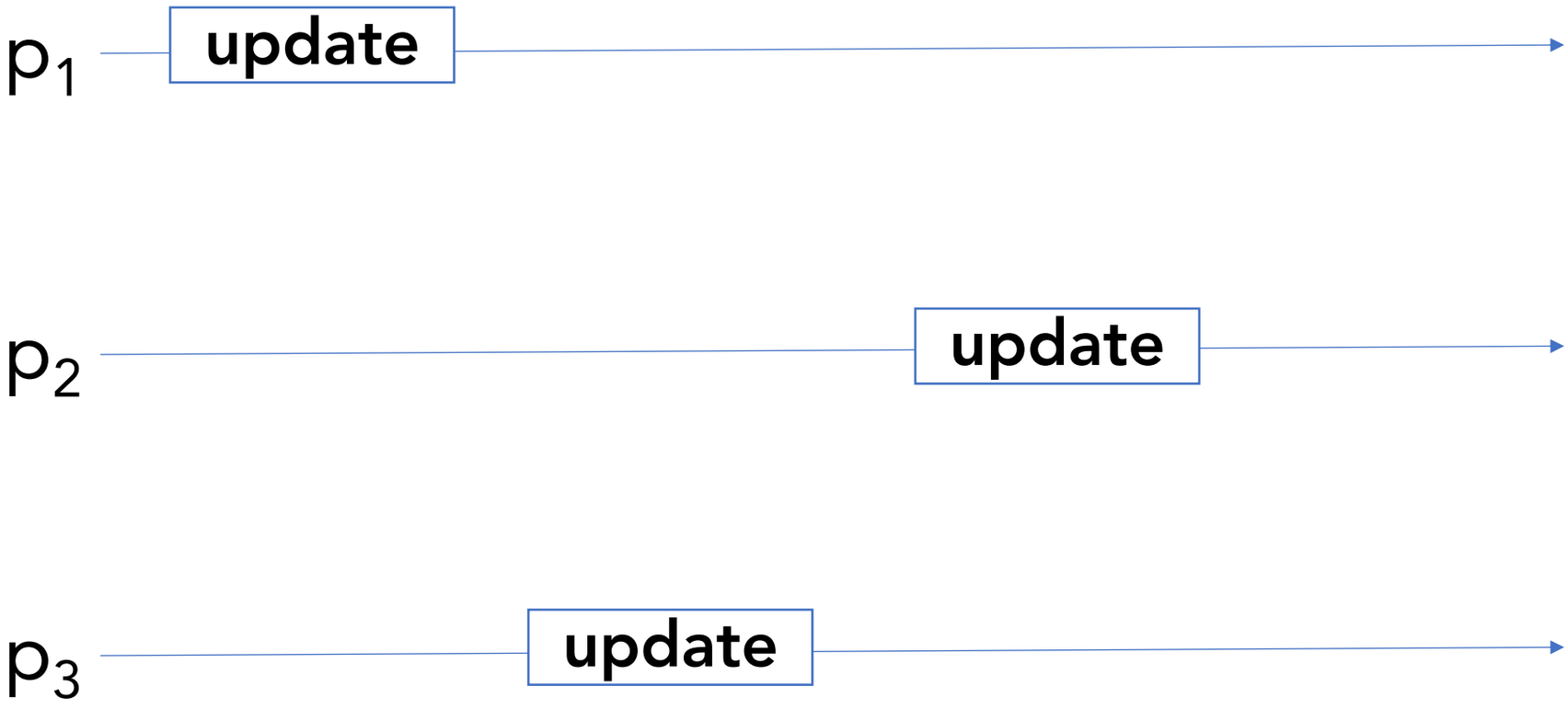
Q: Can we avoid persistent fences?

A: No, they are unavoidable.
But 1 fence per operation is sufficient.

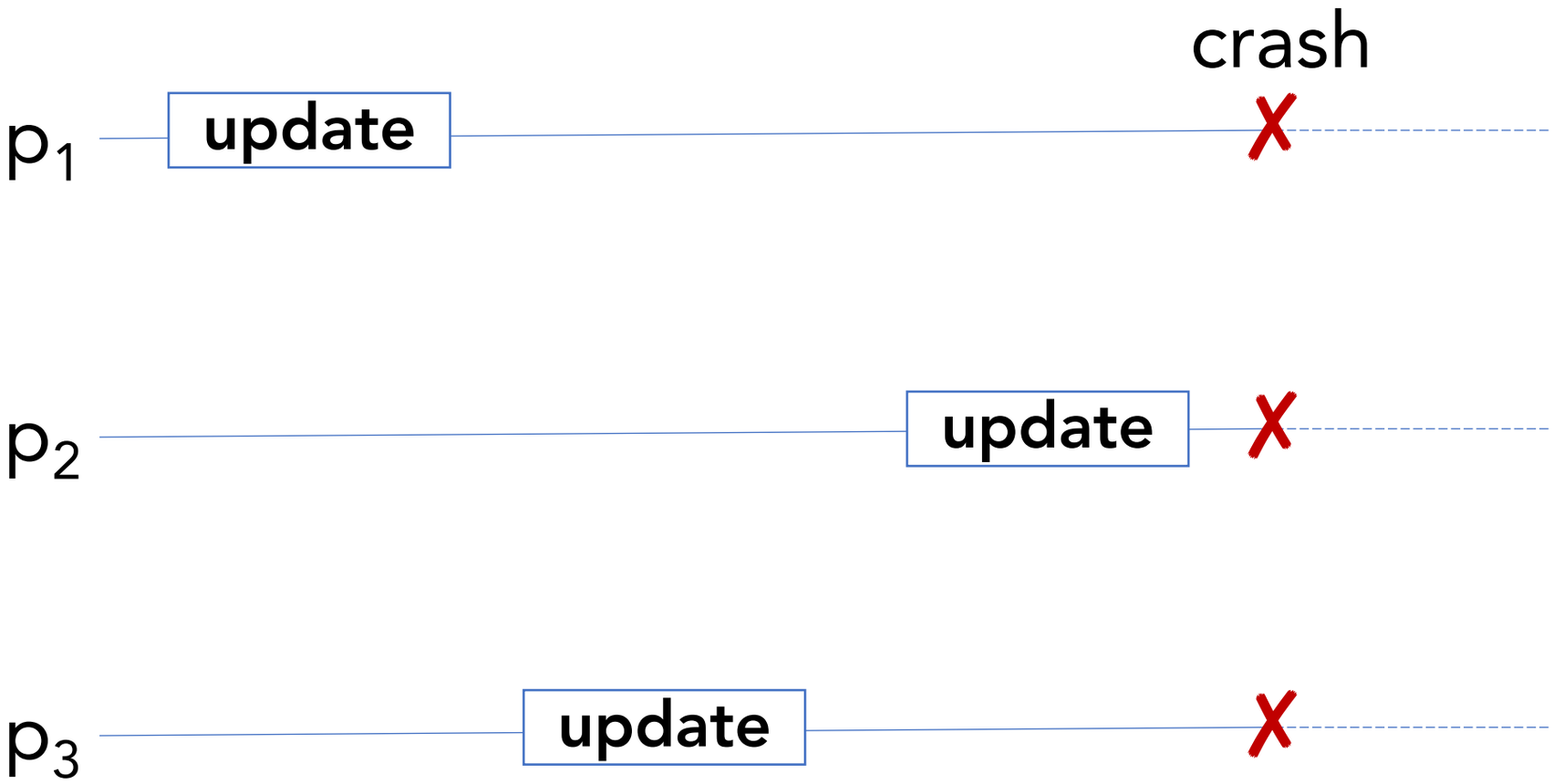
You Can’t Eliminate Fences

- 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

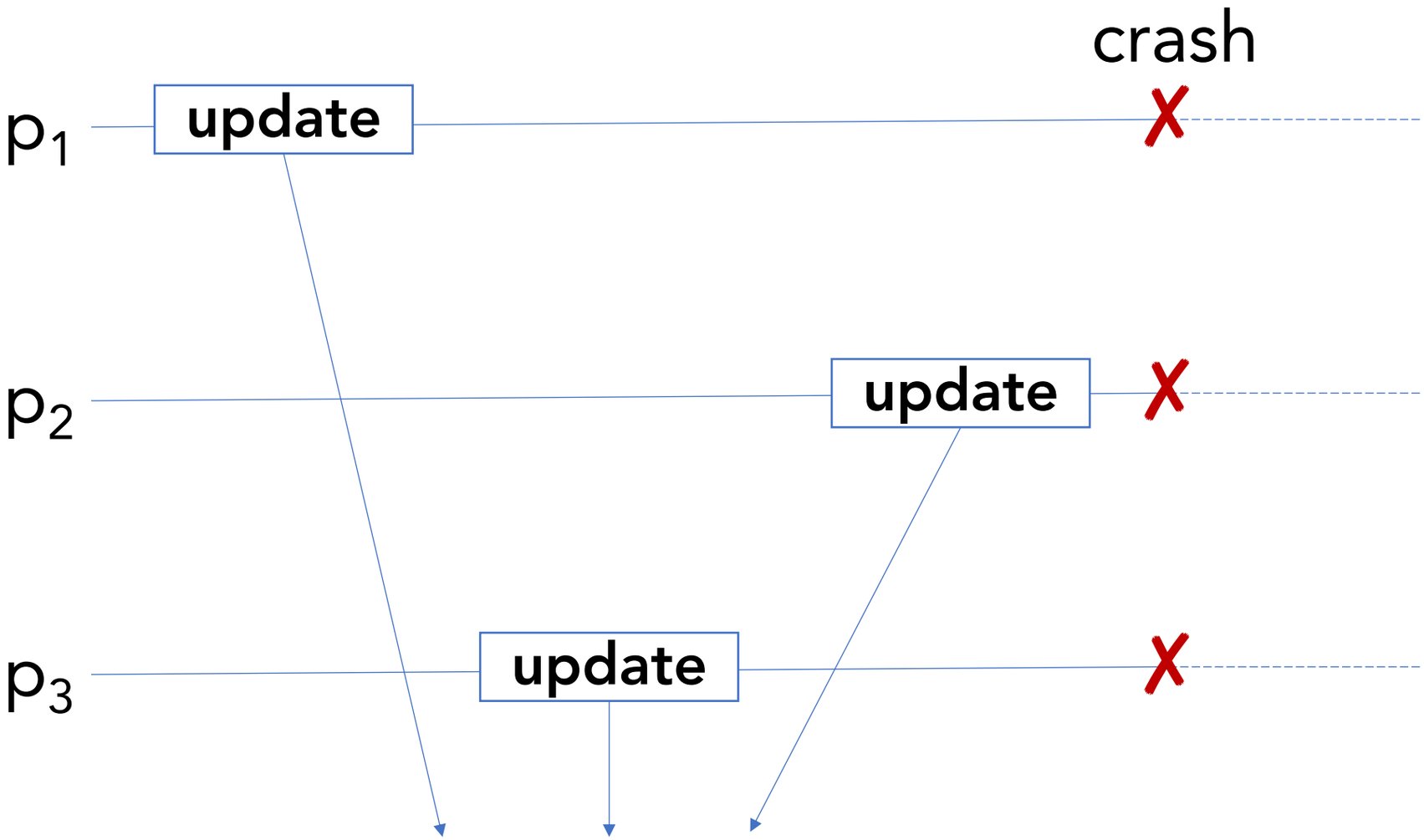
Lower Bound: Sequential Case



Lower Bound: Sequential Case



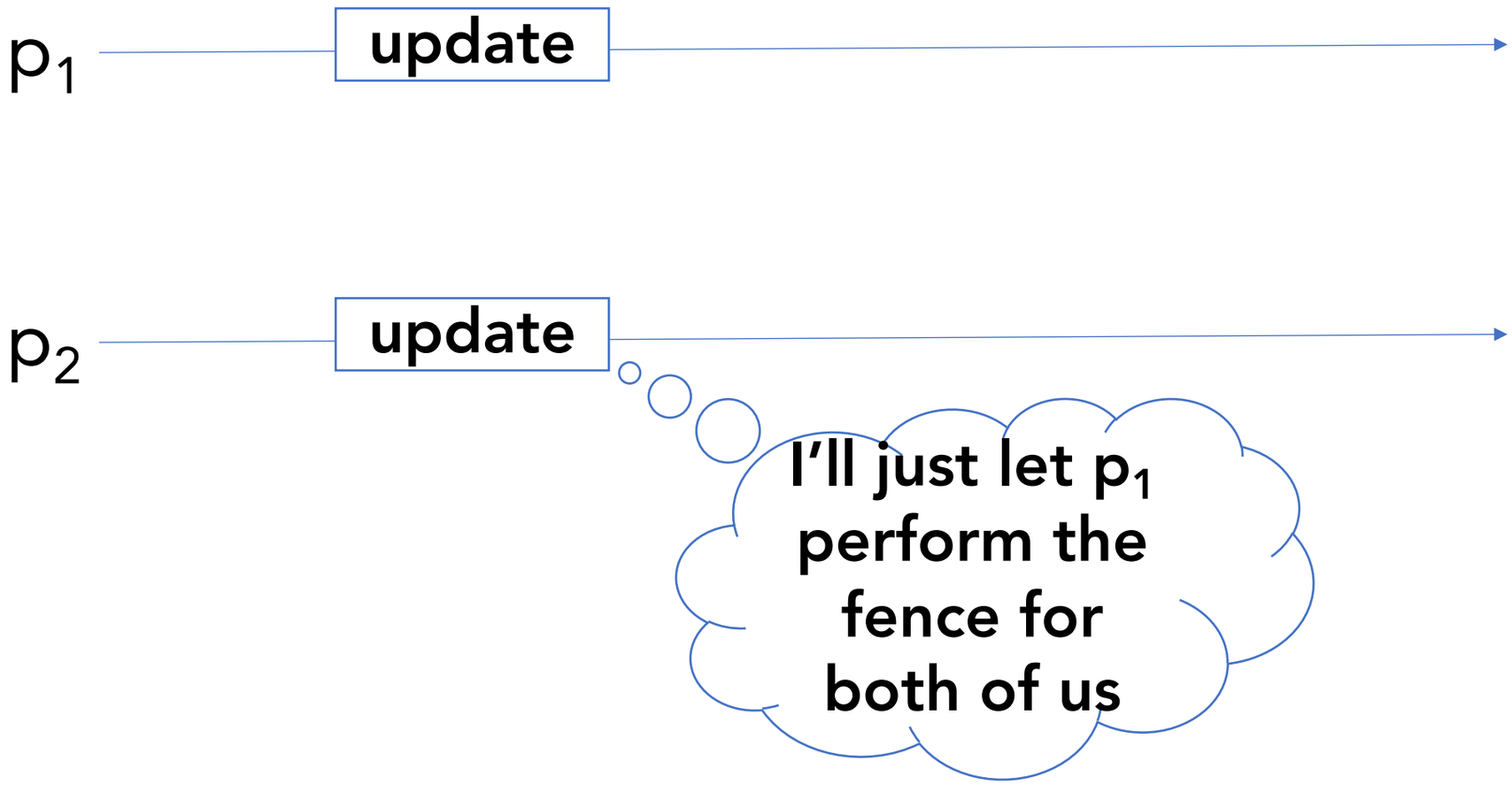
Lower Bound: Sequential Case



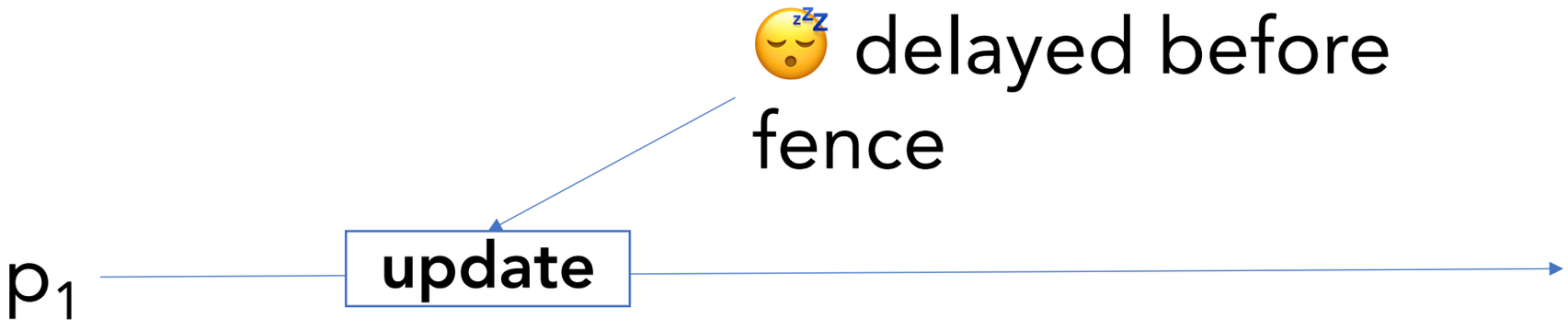
Need at least 1 persistent fence for every update.

Lower Bound: Concurrent Case

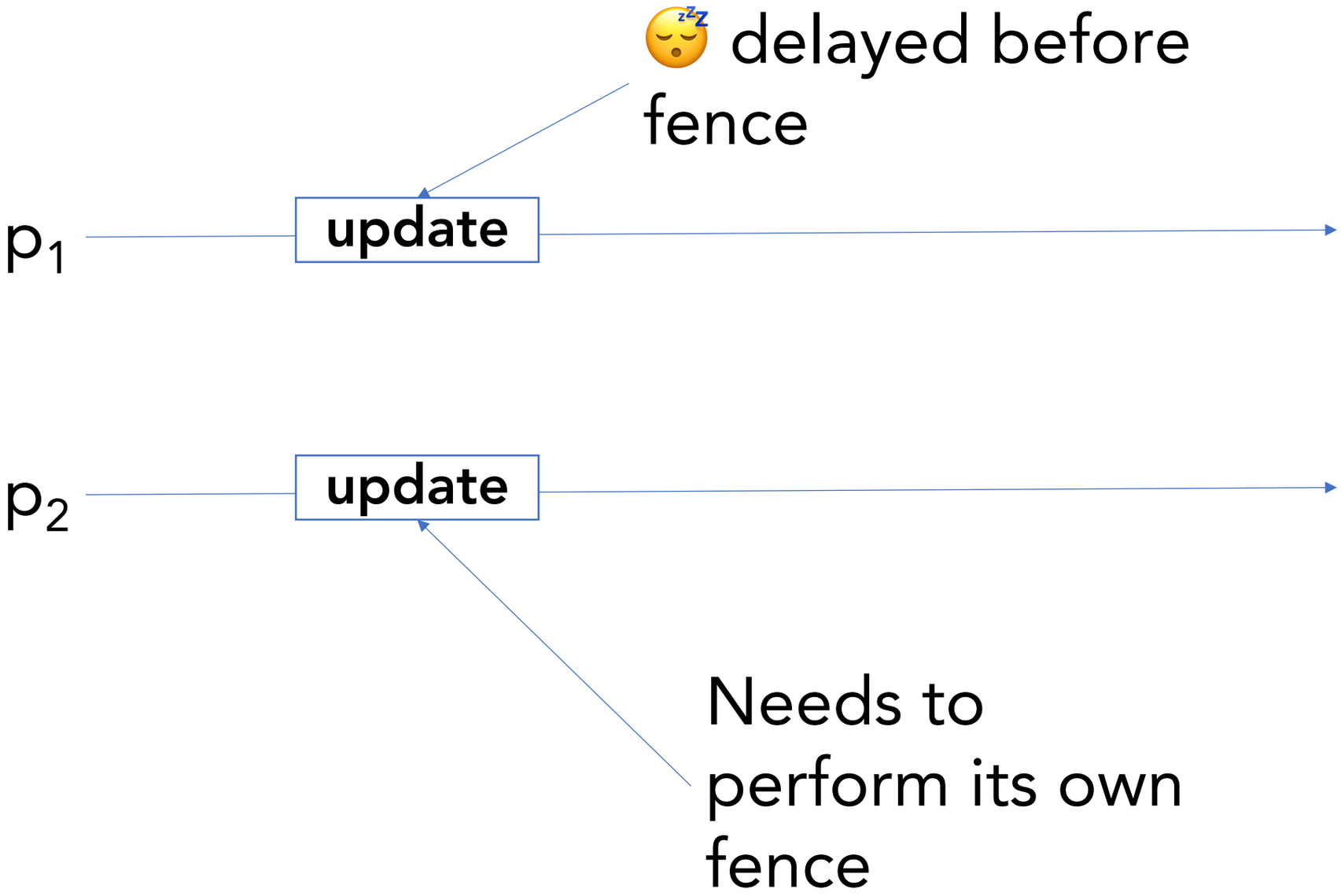
Lower Bound: Concurrent Case



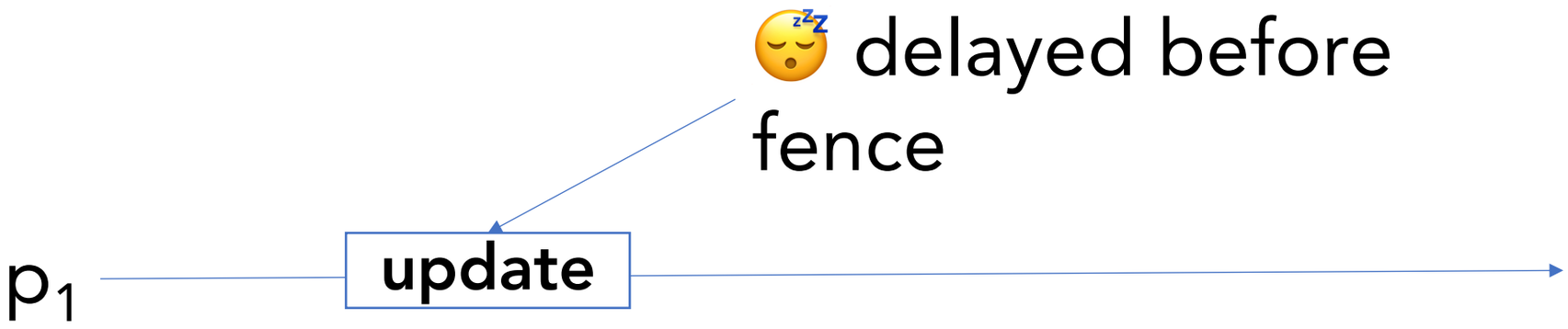
Lower Bound: Concurrent Case

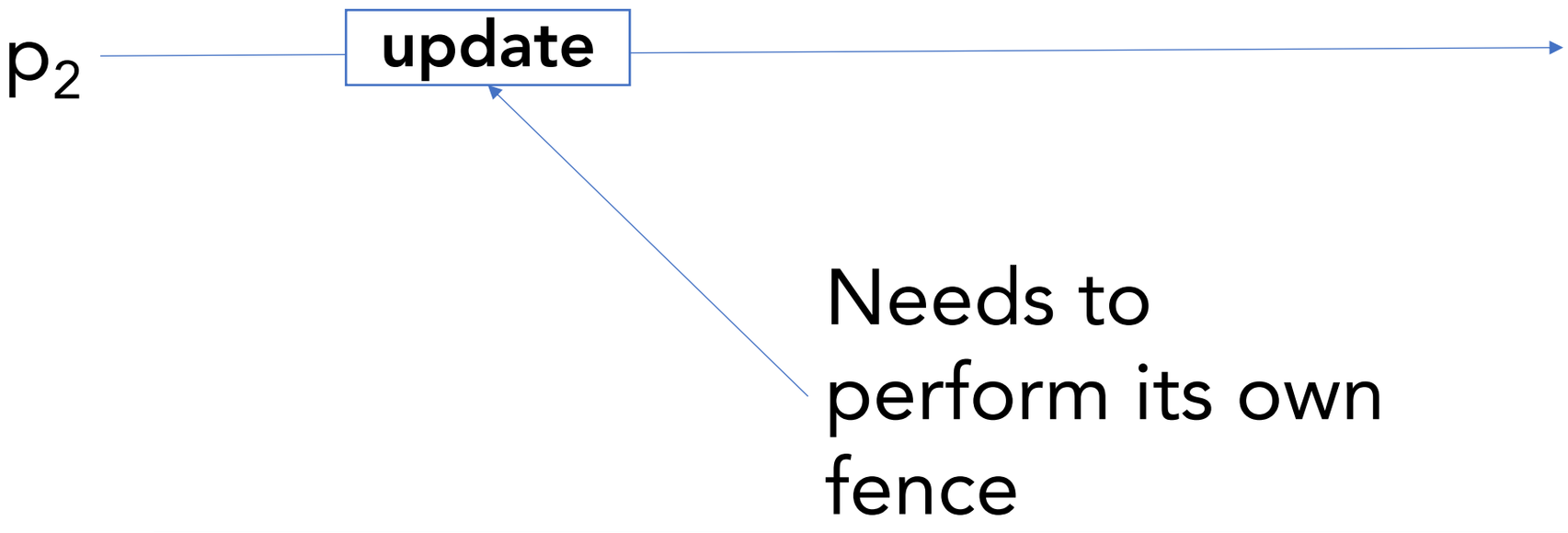


Lower Bound: Concurrent Case



Lower Bound: Concurrent Case





Both processes perform one fence per update operation.

Recap — Persistent Memory

- What is persistent memory?
- How to define correctness for PM?
- Efficient Data Structures for PM
- A Lower Bound for PM – can’t always avoid fences

Papers Referenced

1. Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, Igor Zablotchi. *The Impact of RDMA on Agreement*. PODC 2019.
2. Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, Athanasios Xygkis, Igor Zablotchi. *Microsecond Consensus for Microsecond Applications*. OSDI 2020.
3. Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, Igor Zablotchi. *Log-Free Concurrent Data Structures*. In USENIX ATC 2018.
4. Nachshon Cohen, Rachid Guerraoui, Igor Zablotchi. The Inherent Cost of Remembering Consistently. SPAA 2019.