Distributed Updates

CSE 4/562: Database Systems | Lecture 23

DB. Sys.: T.C.B.: Ch. 17.1-17.5

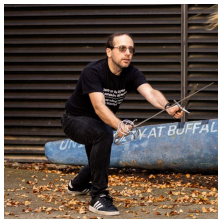
Updates

- Checkpoint 4 due Mon
- Course Reviews 0/31
  - At 28/31 I will publish a question worth no less than 10/100 pt from the final
- Final Exam
  - Will replace midterm if better

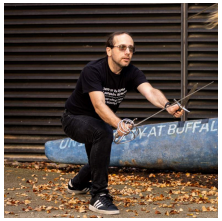
How do we preserve ACID in distributed databases.

Setting

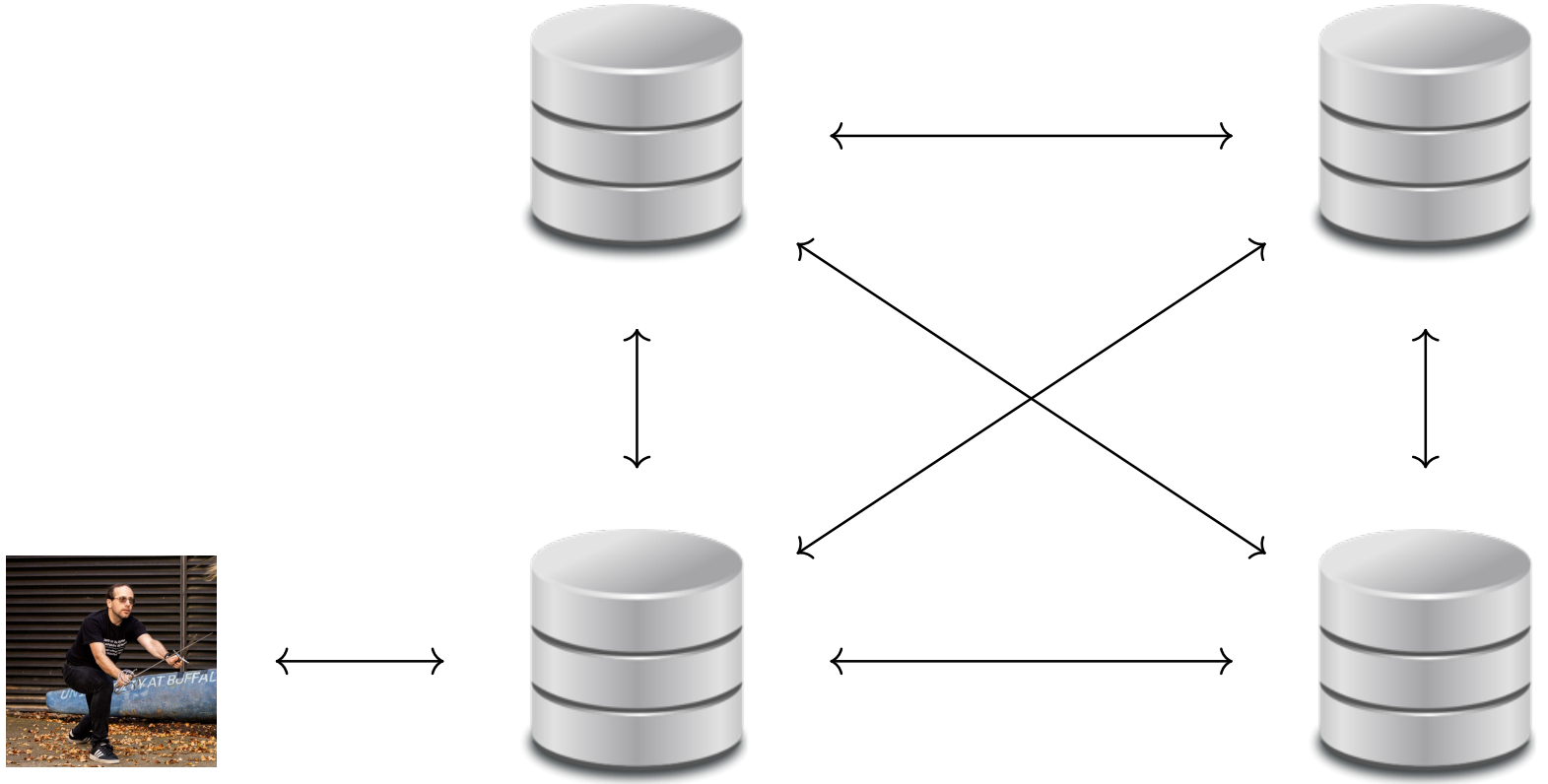
Setting



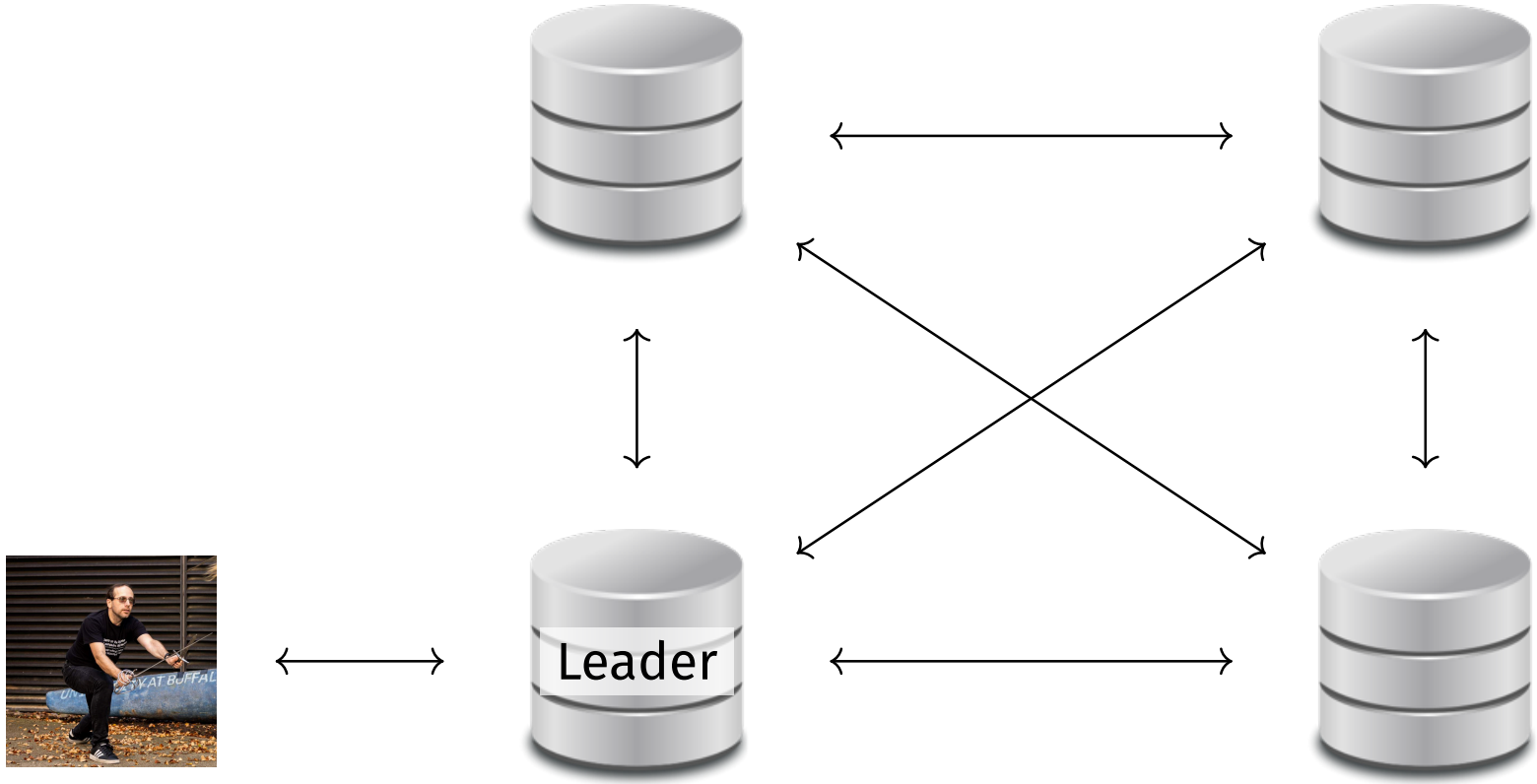
Setting



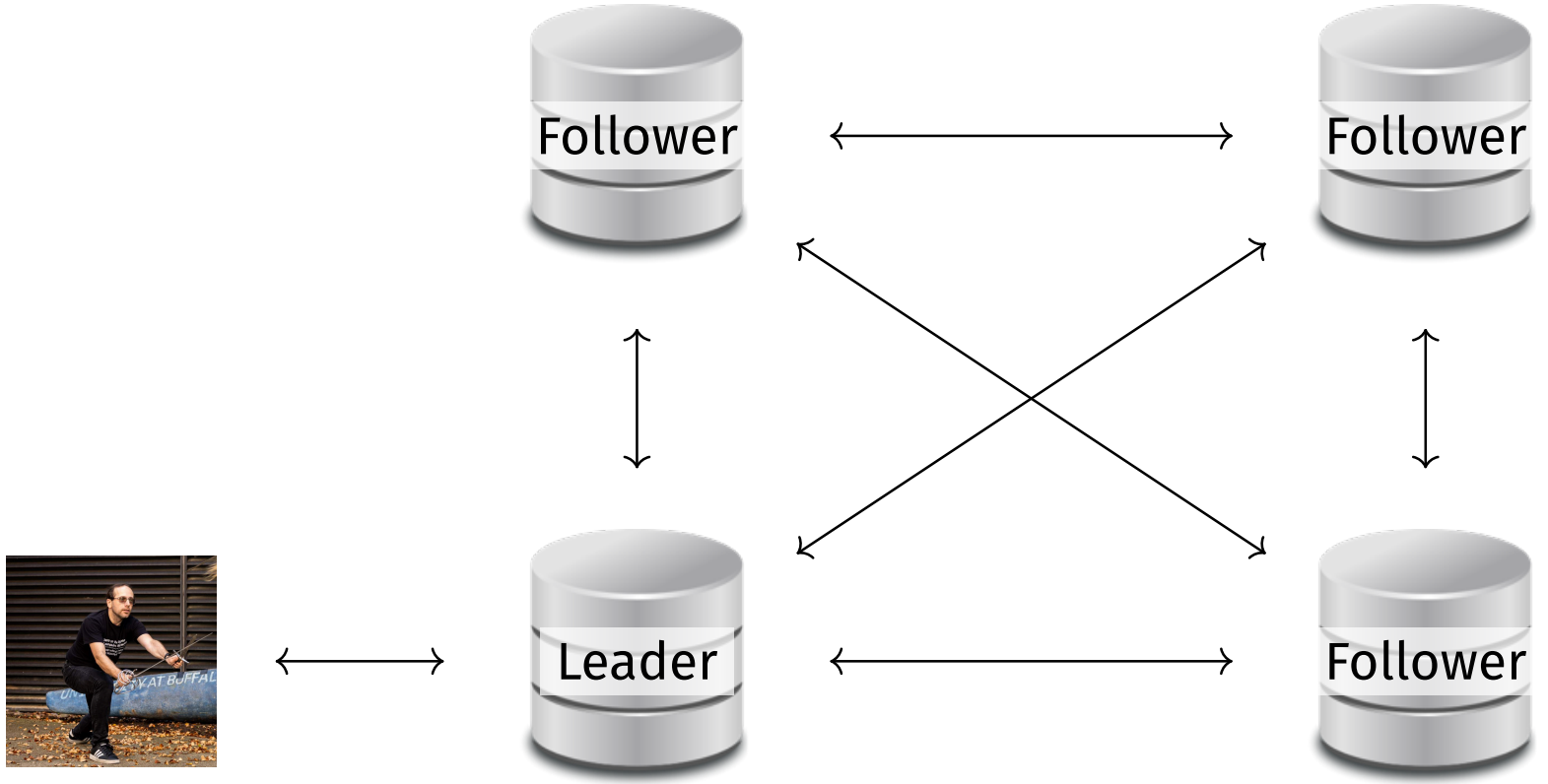
Setting



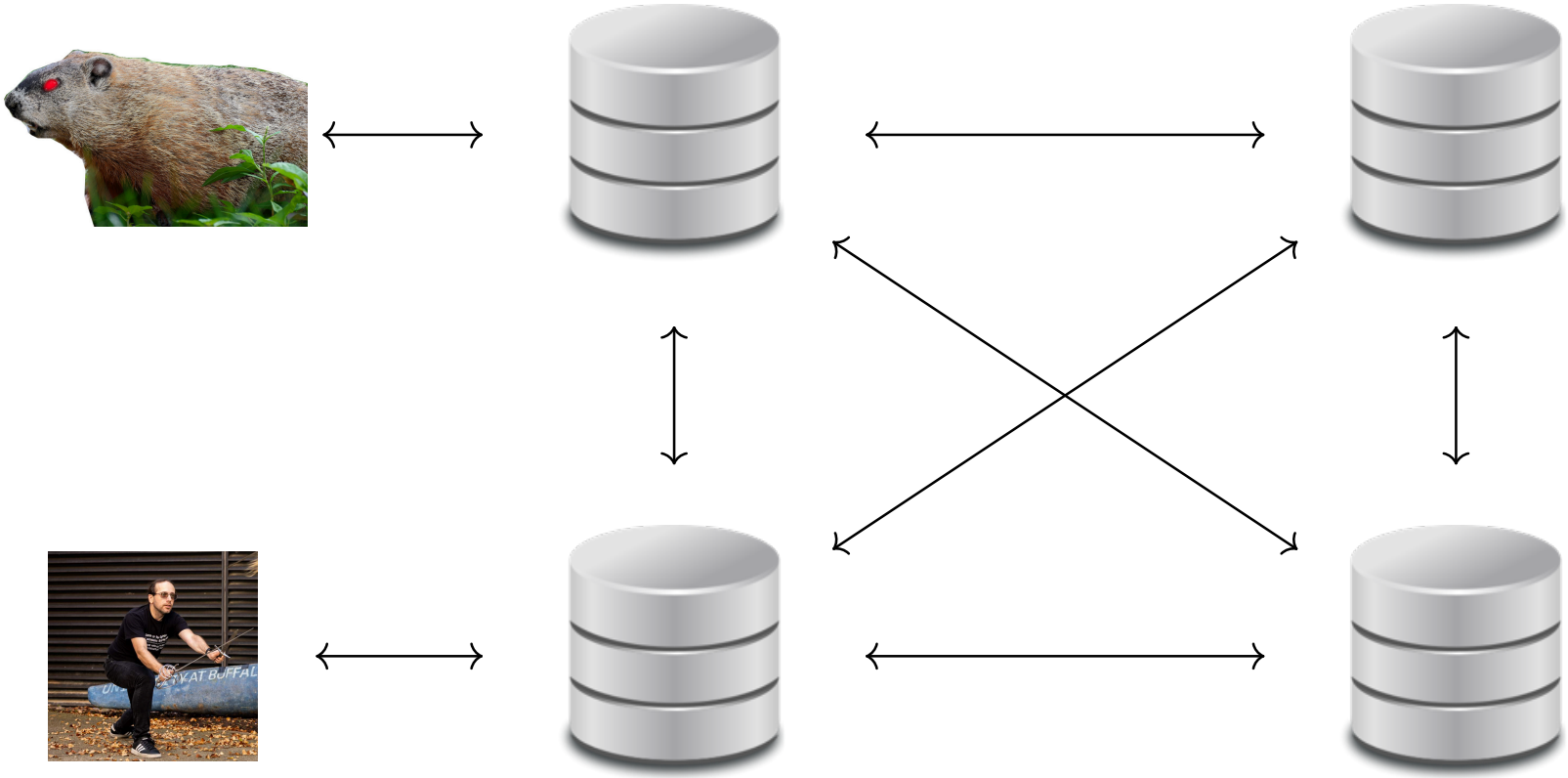
Setting



Setting



Setting



Assumptions

1. Each node can enforce ACID **locally**.
2. No malicious nodes.
3. Messages arrive in full or not at all.
4. No global state.

Each node has to make local decisions without global state.

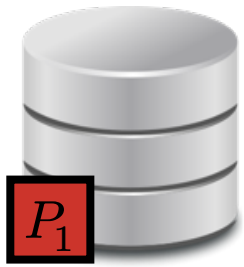
Message Ordering is Hard

Image credit: Openclipart.org

Message Ordering is Hard

Image credit: Openclipart.org

Message Ordering is Hard



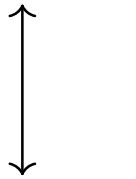


Image credit: Openclipart.org

Message Ordering is Hard

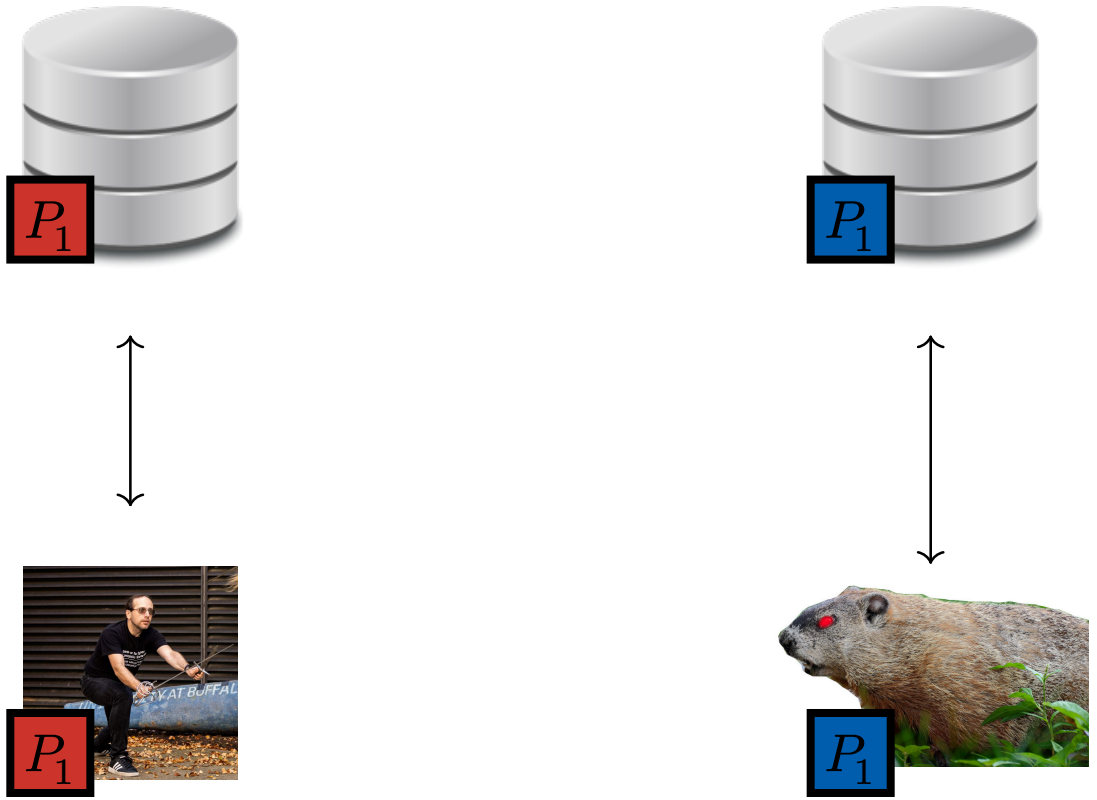


Image credit: Openclipart.org

Message Ordering is Hard

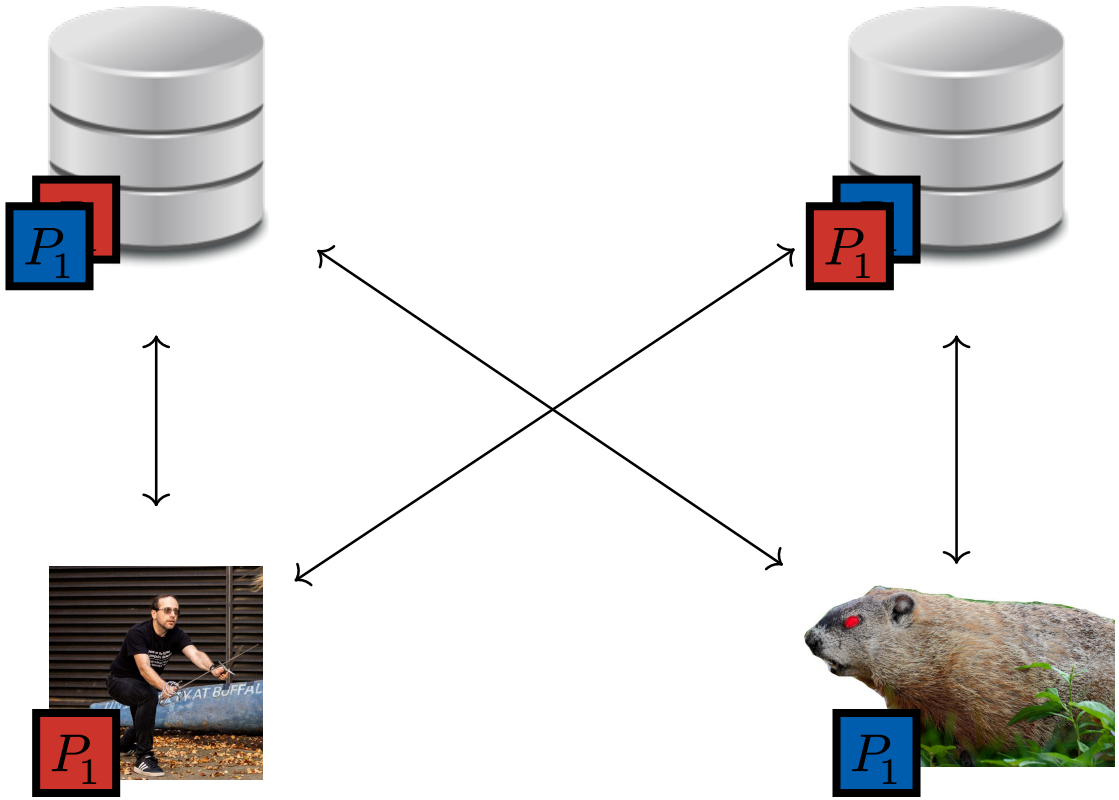


Image credit: Openclipart.org

Message Ordering is Hard

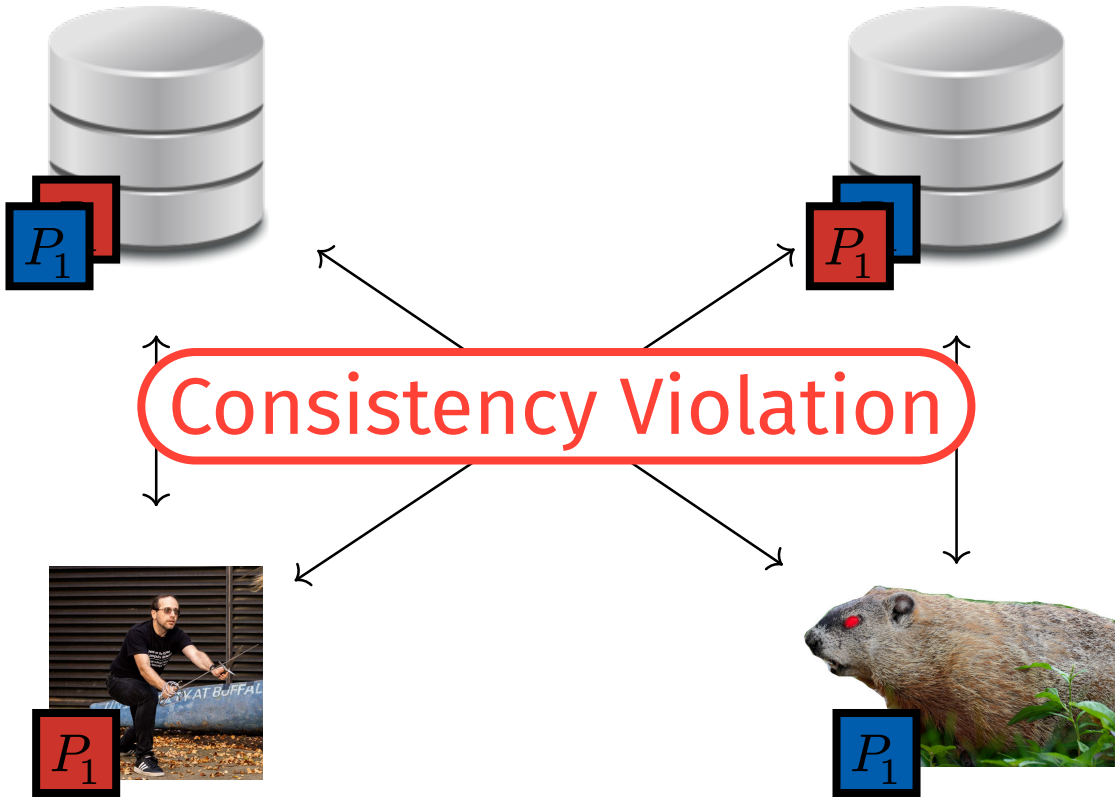


Image credit: Openclipart.org

Message Ordering is Hard

Isolation Violation

Nodes might see transaction requests in a different order.

Idea 1: Route all requests through one node.

Single Leader

Image credit: Openclipart.org

Single Leader

Image credit: Openclipart.org

Single Leader

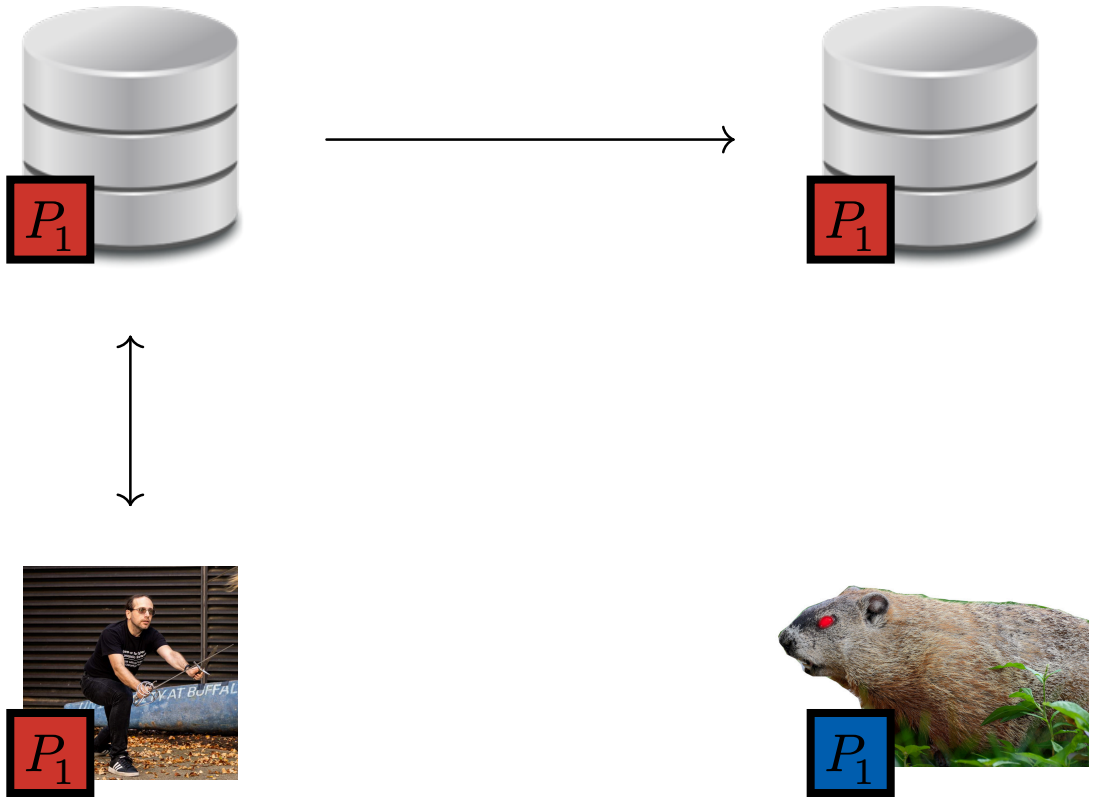


Image credit: Openclipart.org

Single Leader

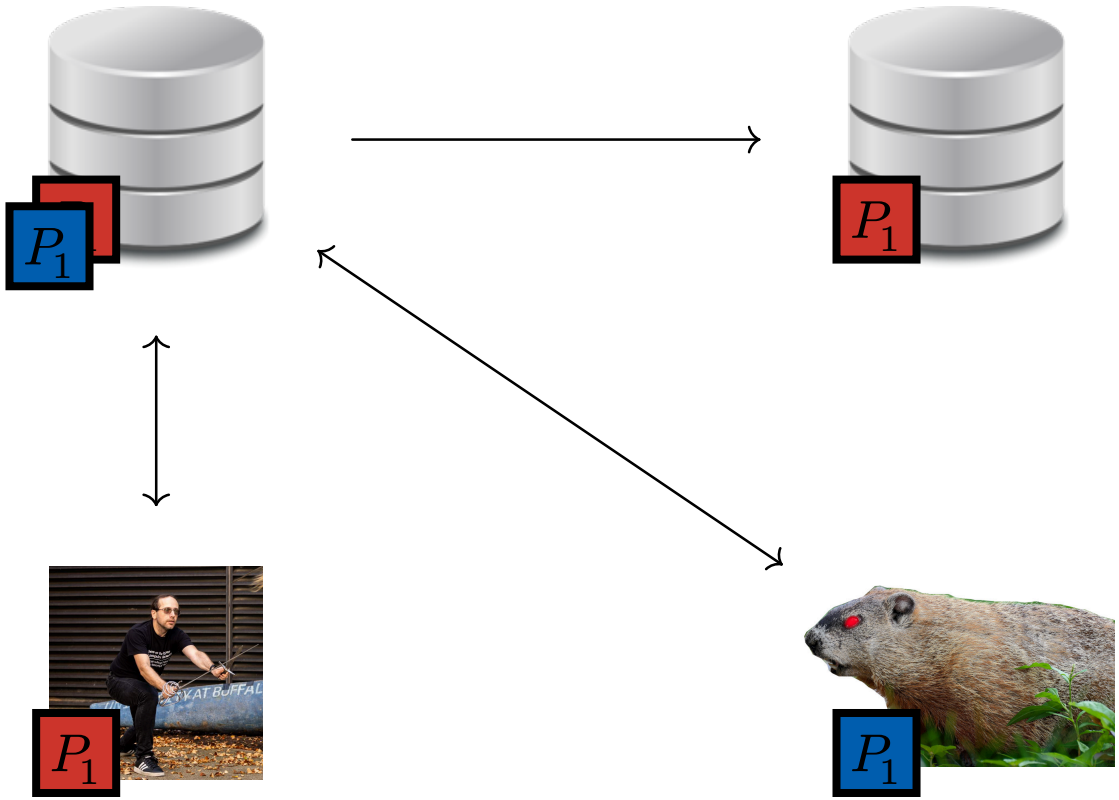


Image credit: Openclipart.org

Single Leader

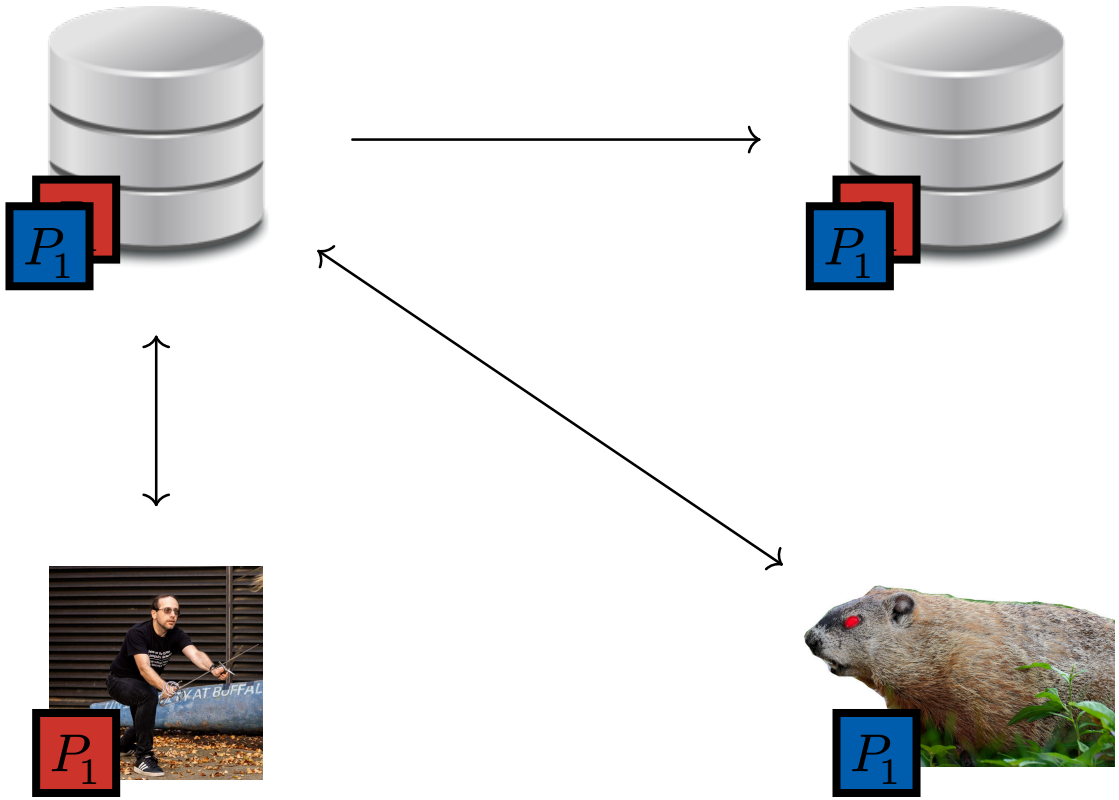


Image credit: Openclipart.org

Pros

Cons

Pros

- Simple

Cons

Pros

- Simple
- Guaranteed to be correct (DB 2 sees exactly what DB 1 sees)

Cons

Pros

- Simple
- Guaranteed to be correct (DB 2 sees exactly what DB 1 sees)

Cons

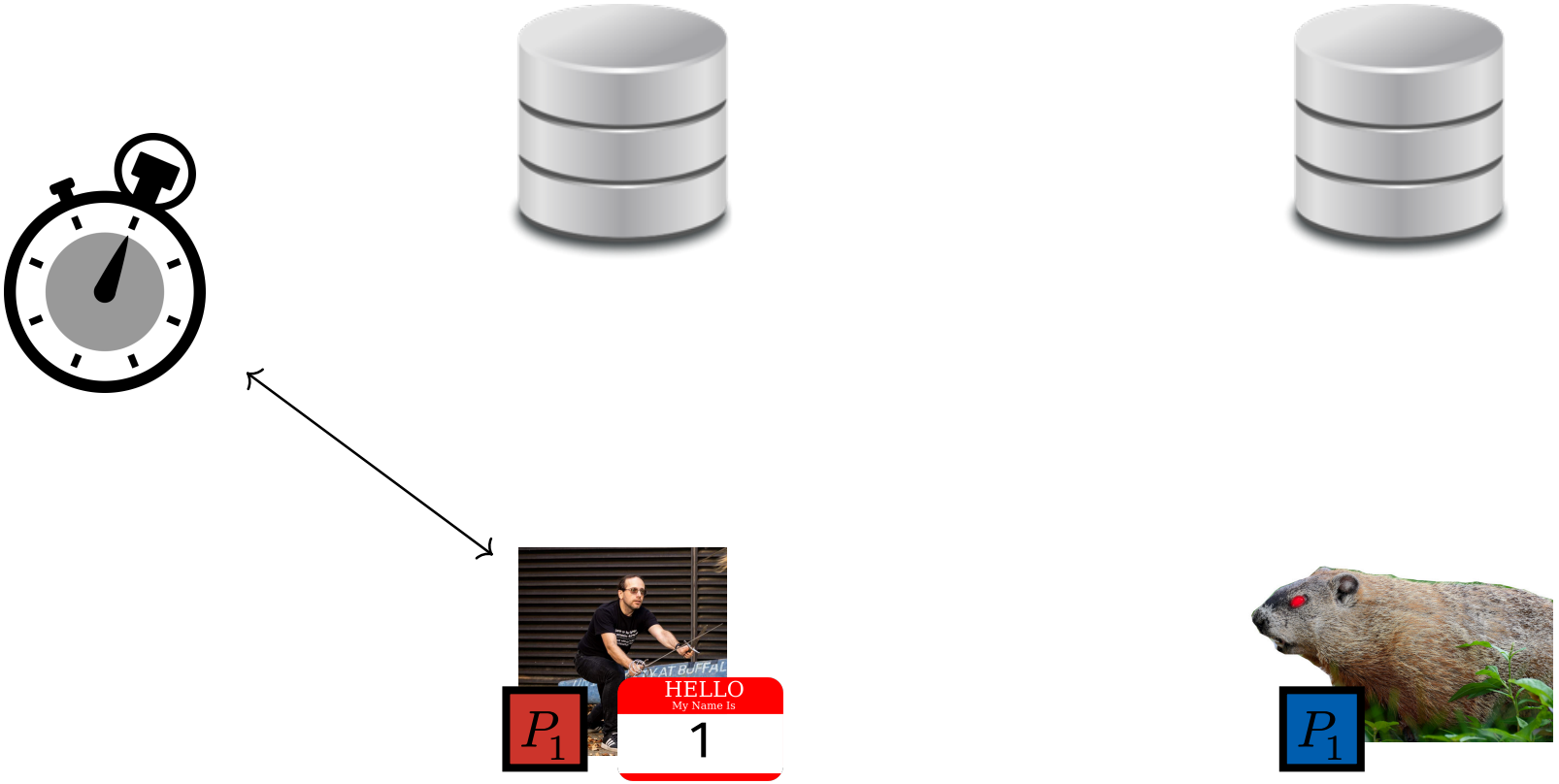
- Single Node is a Bottleneck

Timestamp Server

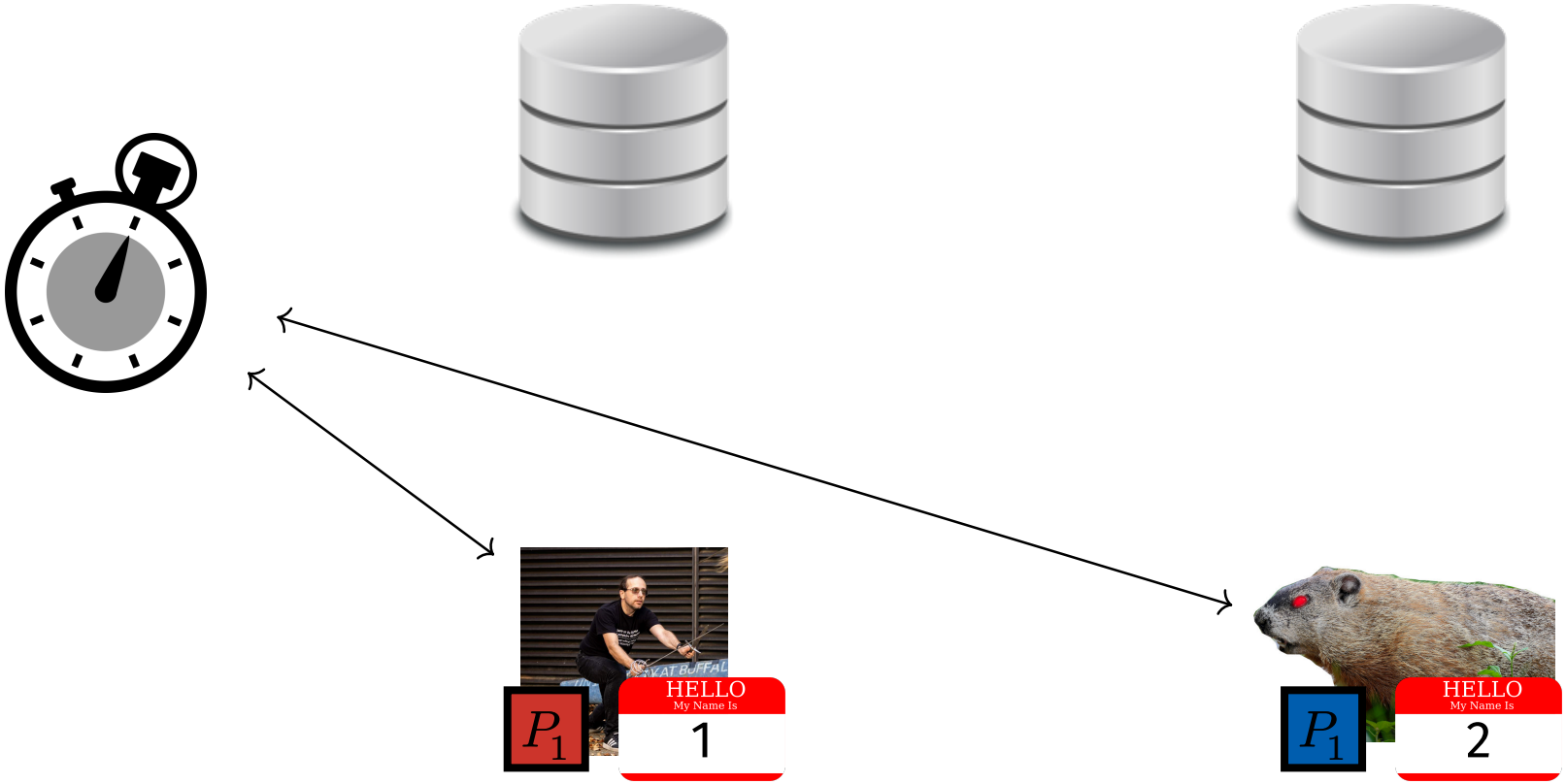
Idea 1: One node defines a serial order.

Timestamp Server

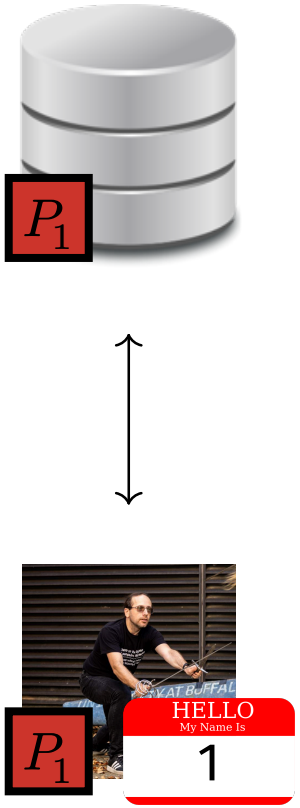
Timestamp Server

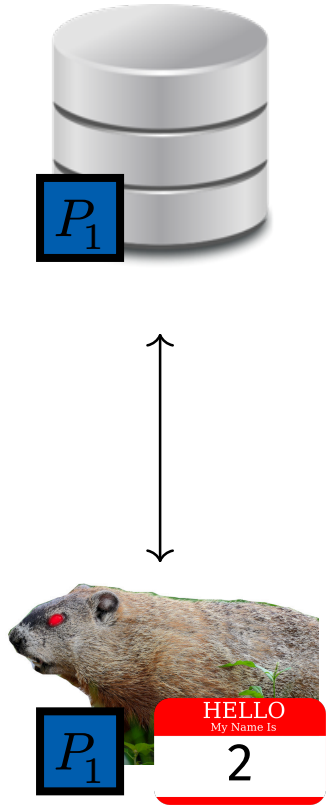


Timestamp Server

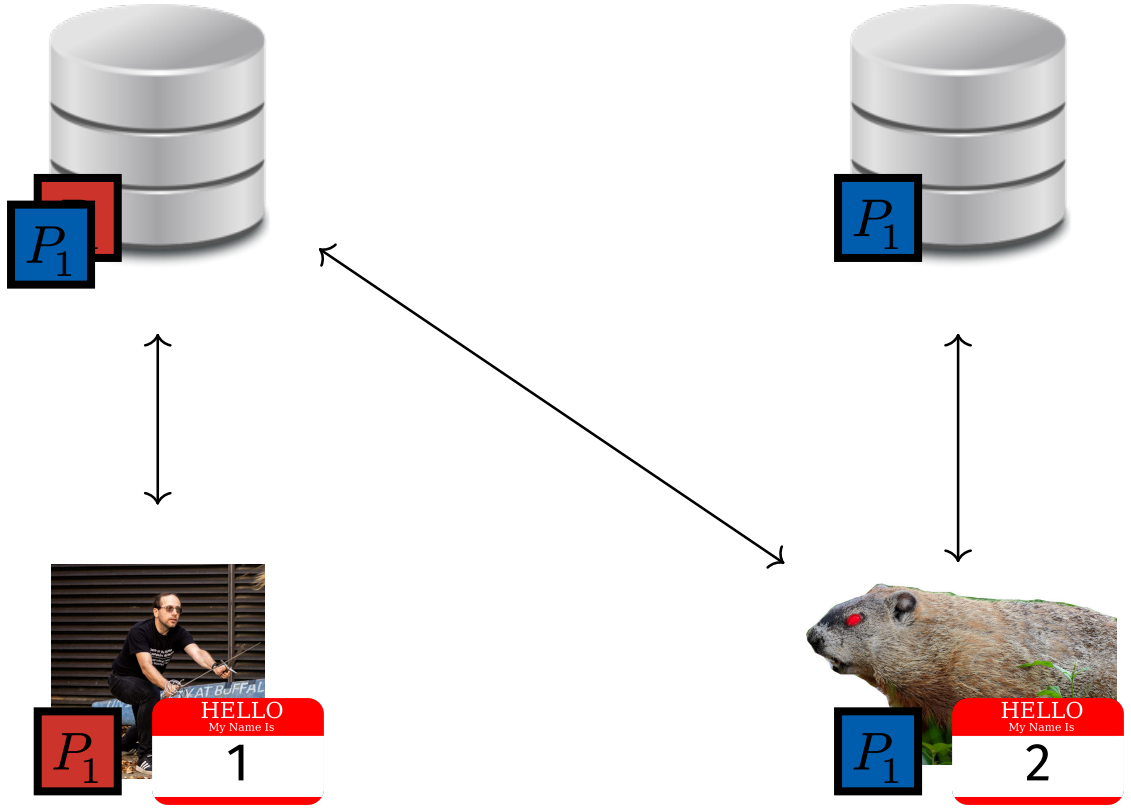


Timestamp Server

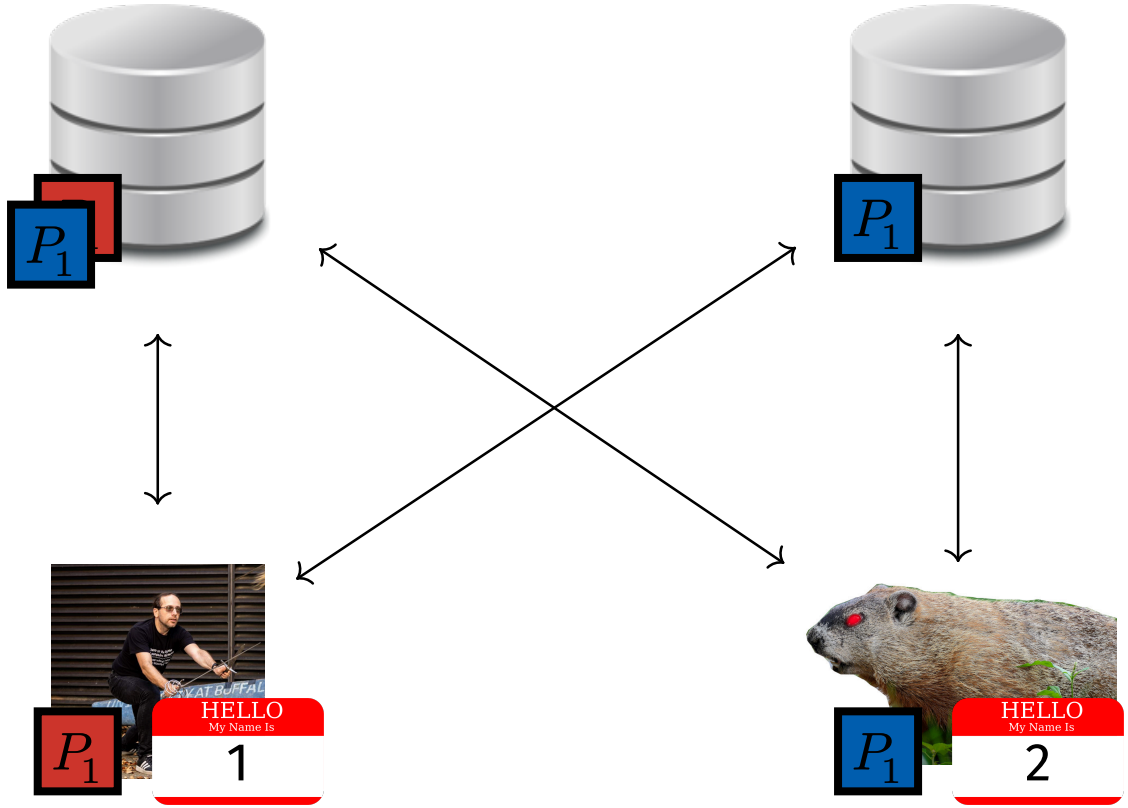




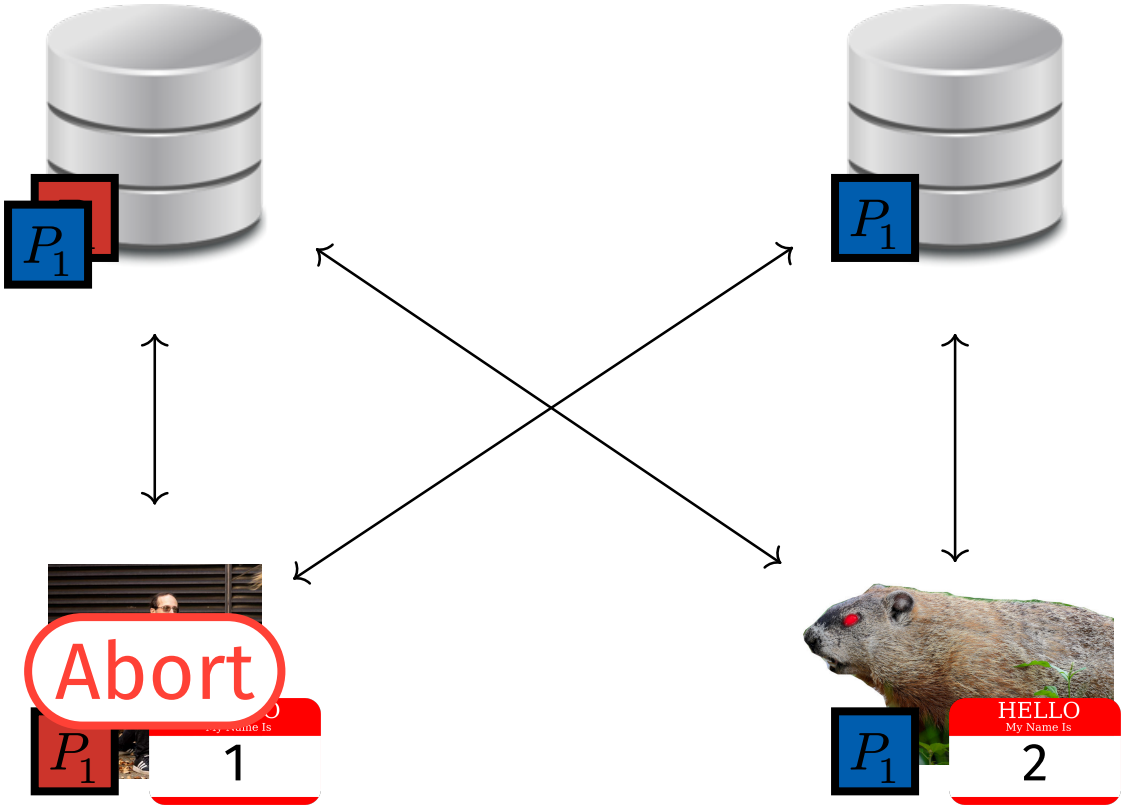
Timestamp Server



Timestamp Server



Timestamp Server



Pros

Cons

Pros

- Serving timesetamps is fast

Cons

Pros

- Serving timesetamps is fast
- No other bottlenecks

Cons

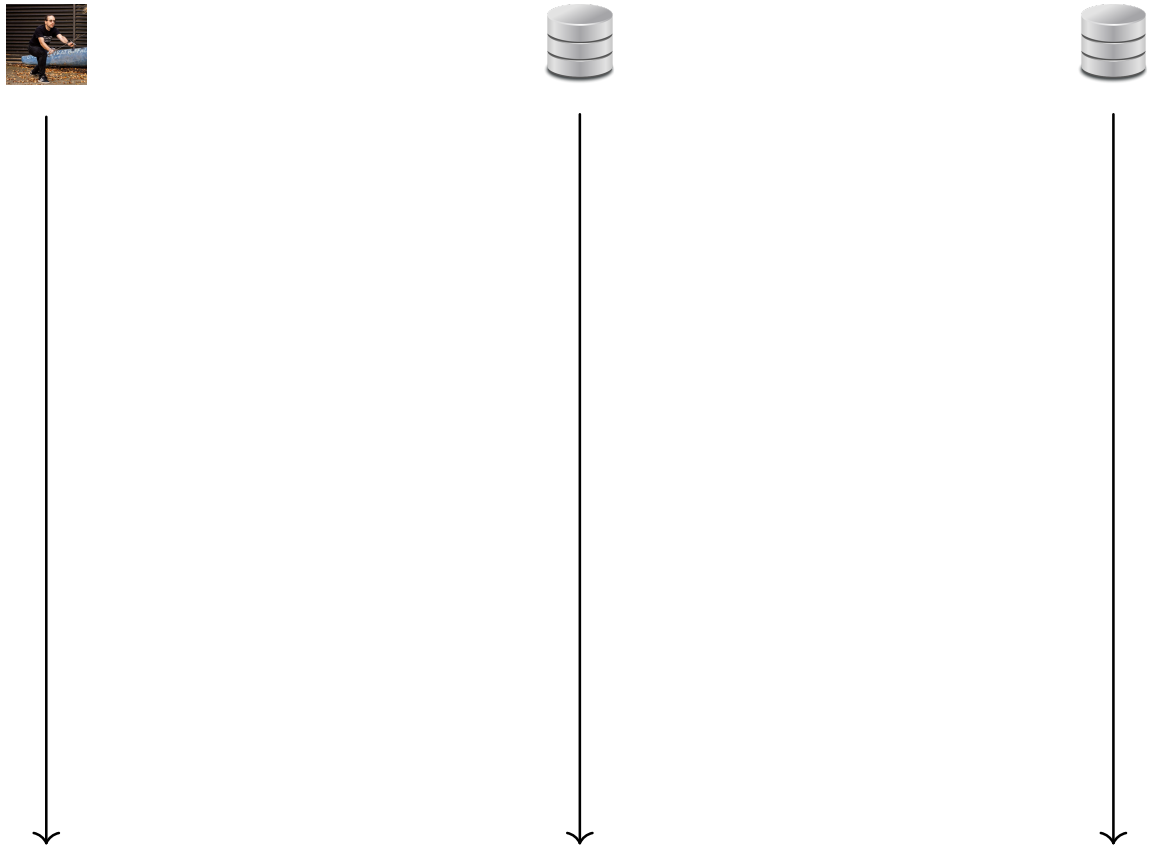
Pros

- Serving timesetamps is fast
- No other bottlenecks

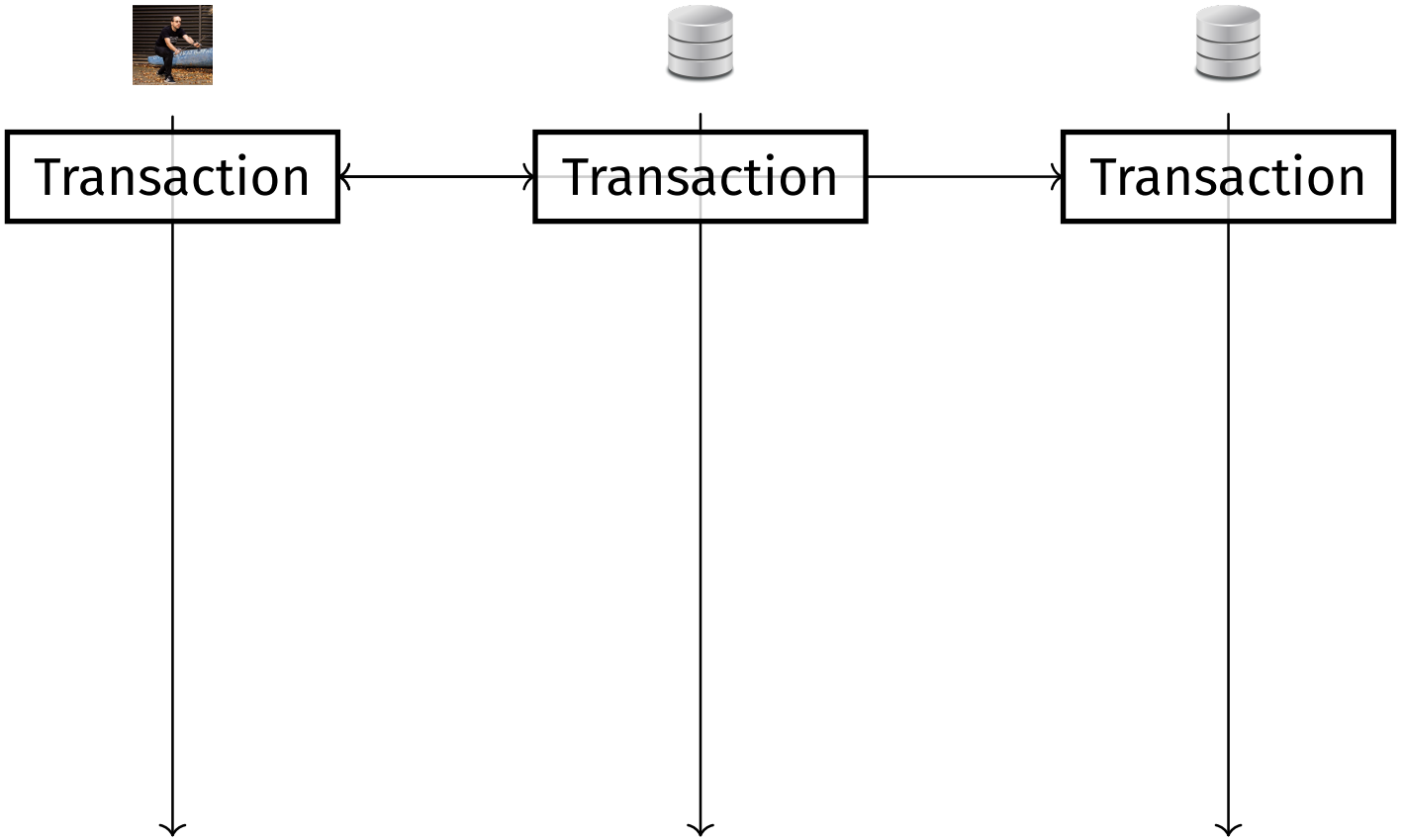
Cons

- When is a transaction safe to commit?

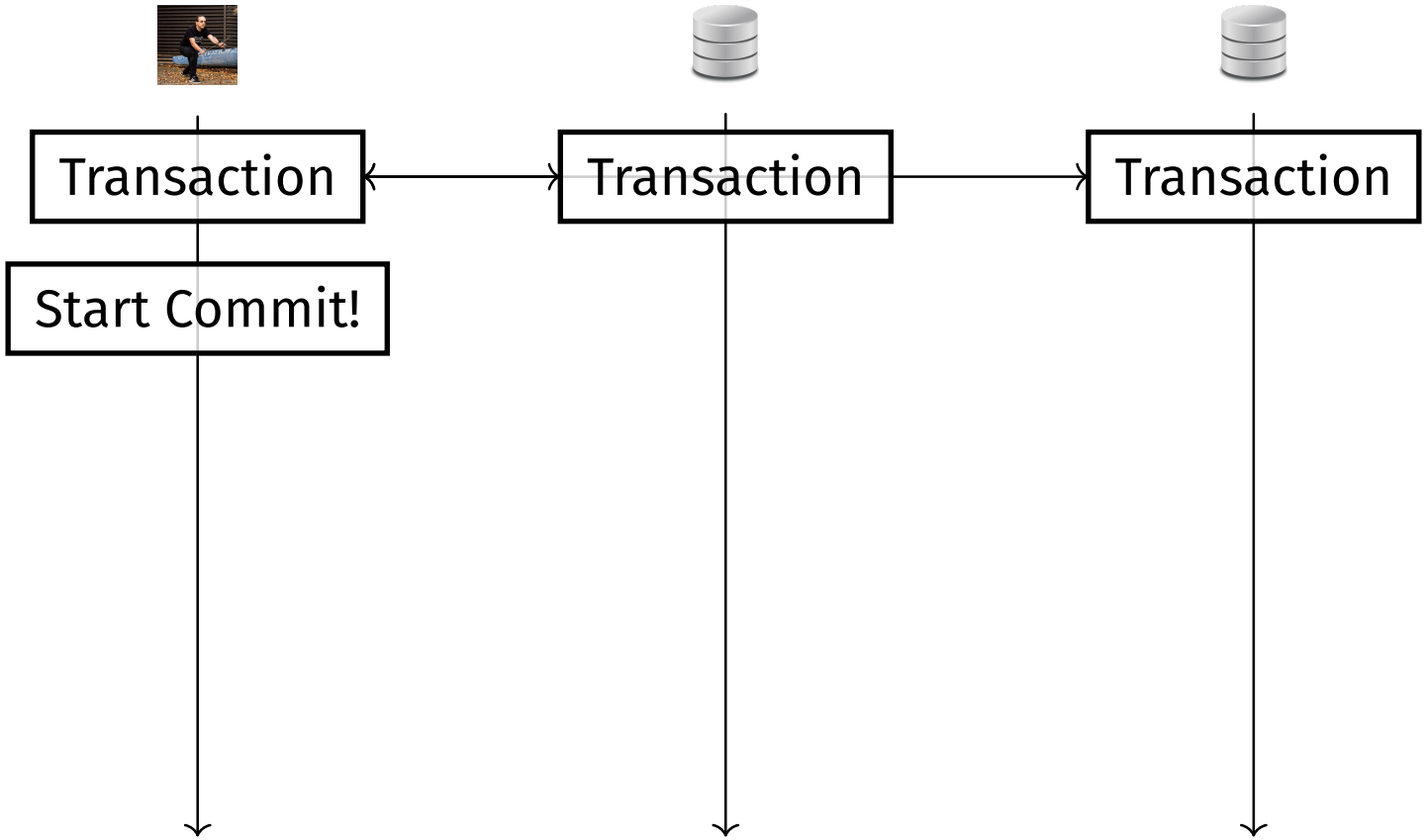
Committing



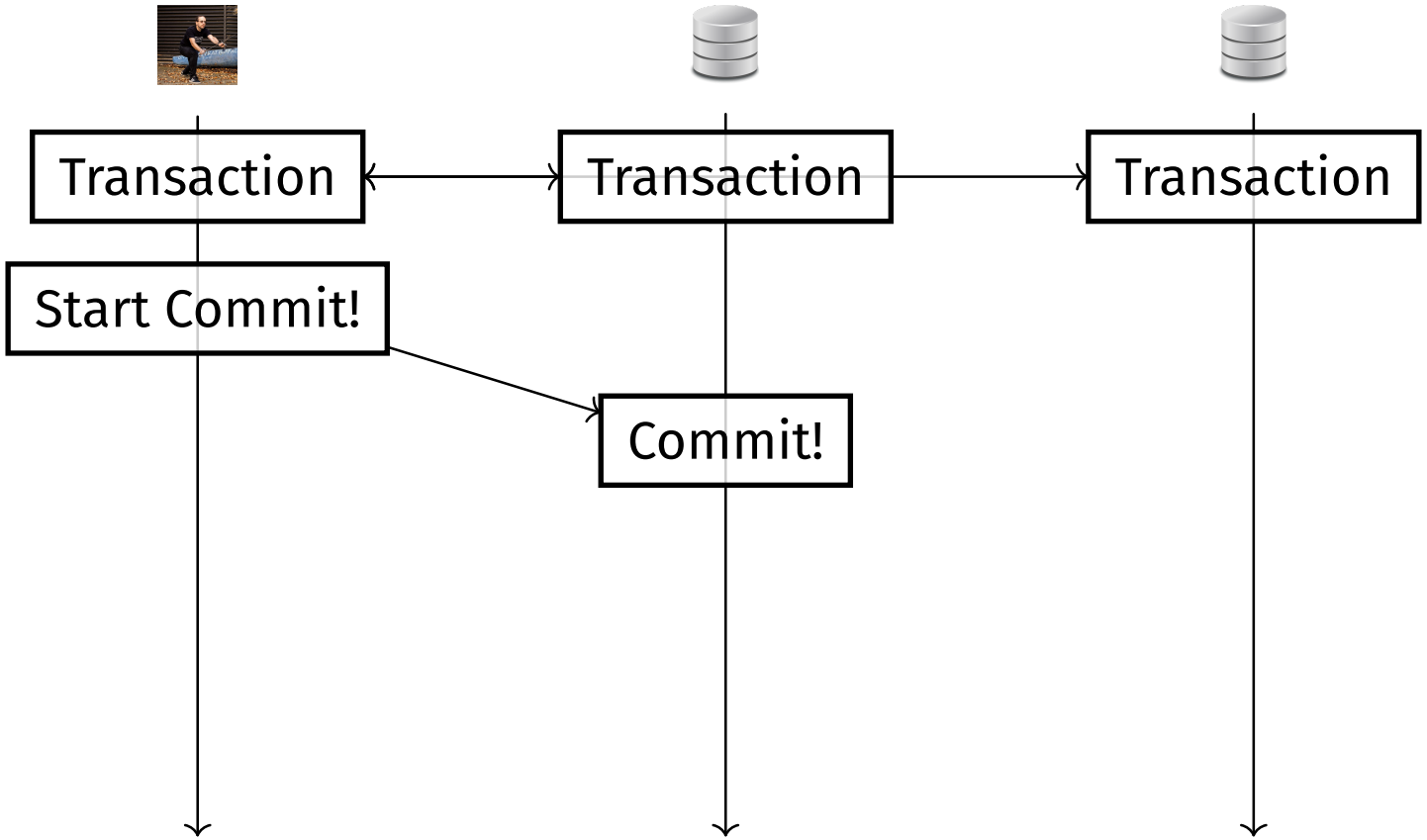
Committing



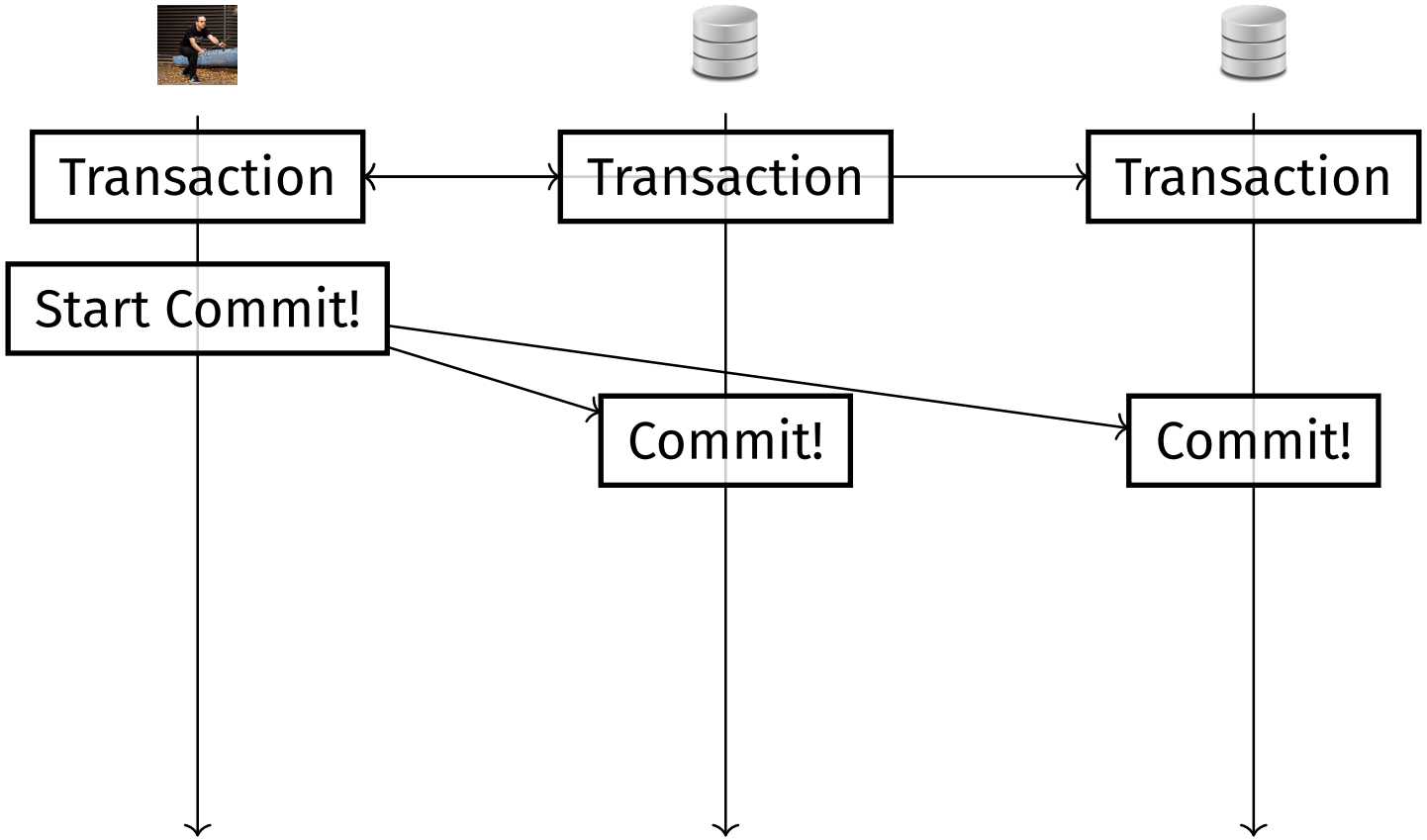
Committing



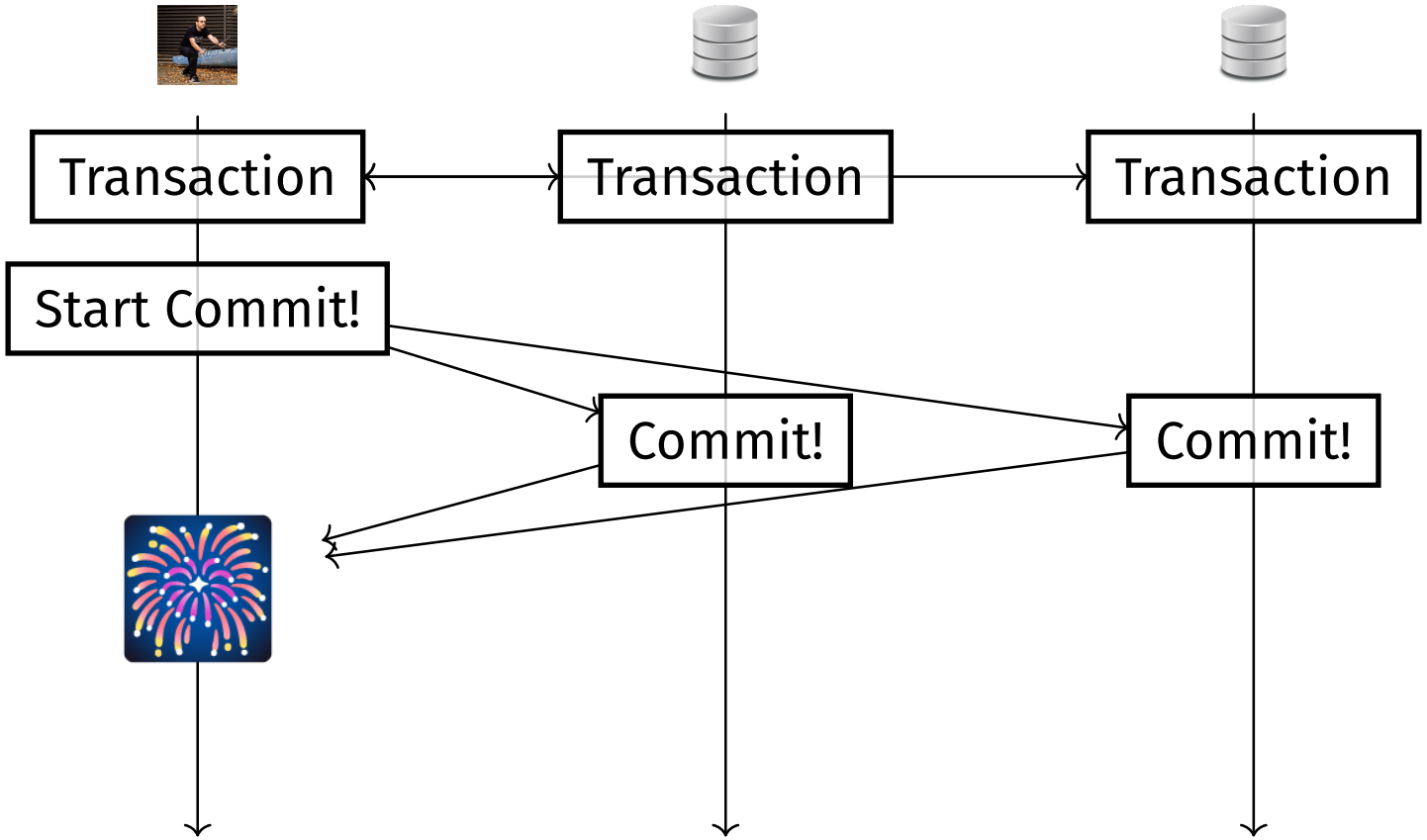
Committing


Transaction
Transaction
Transaction
Start Commit!
Commit!

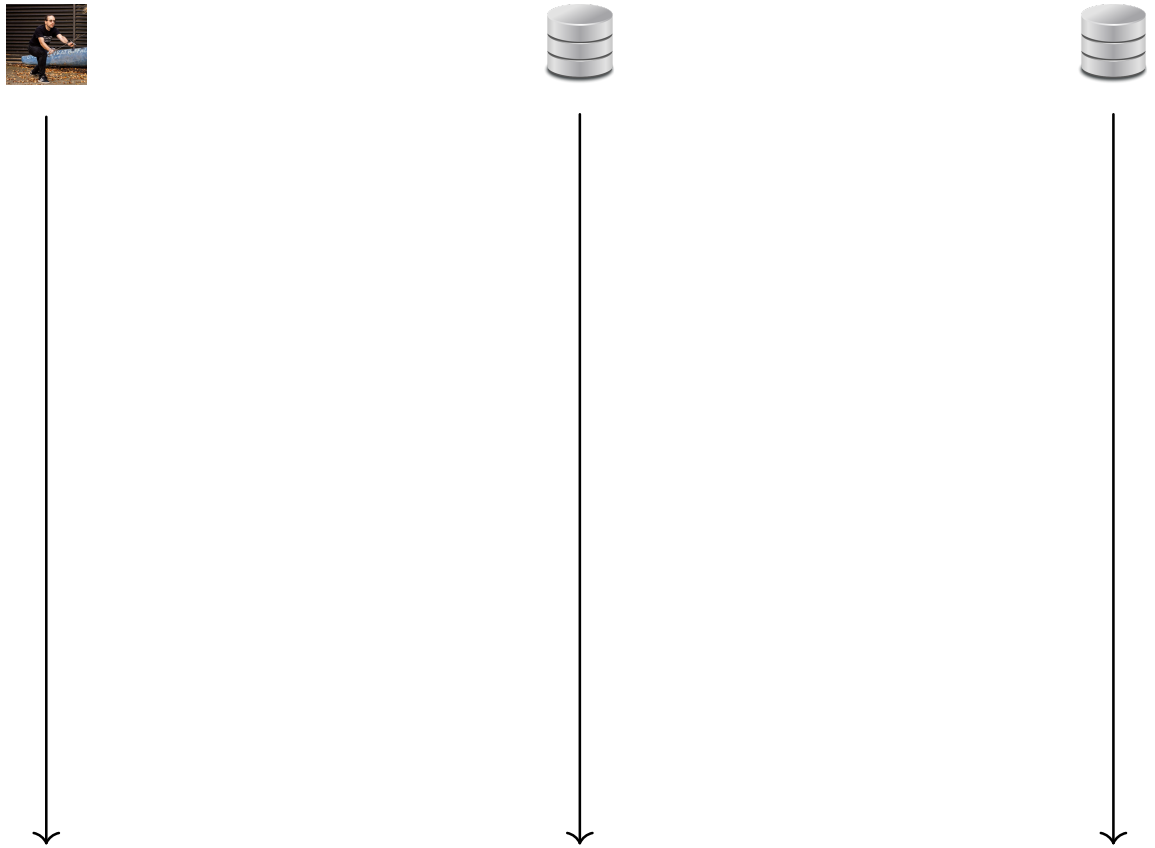
Committing


Transaction
Transaction
Transaction
Start Commit!
Commit!
Commit!

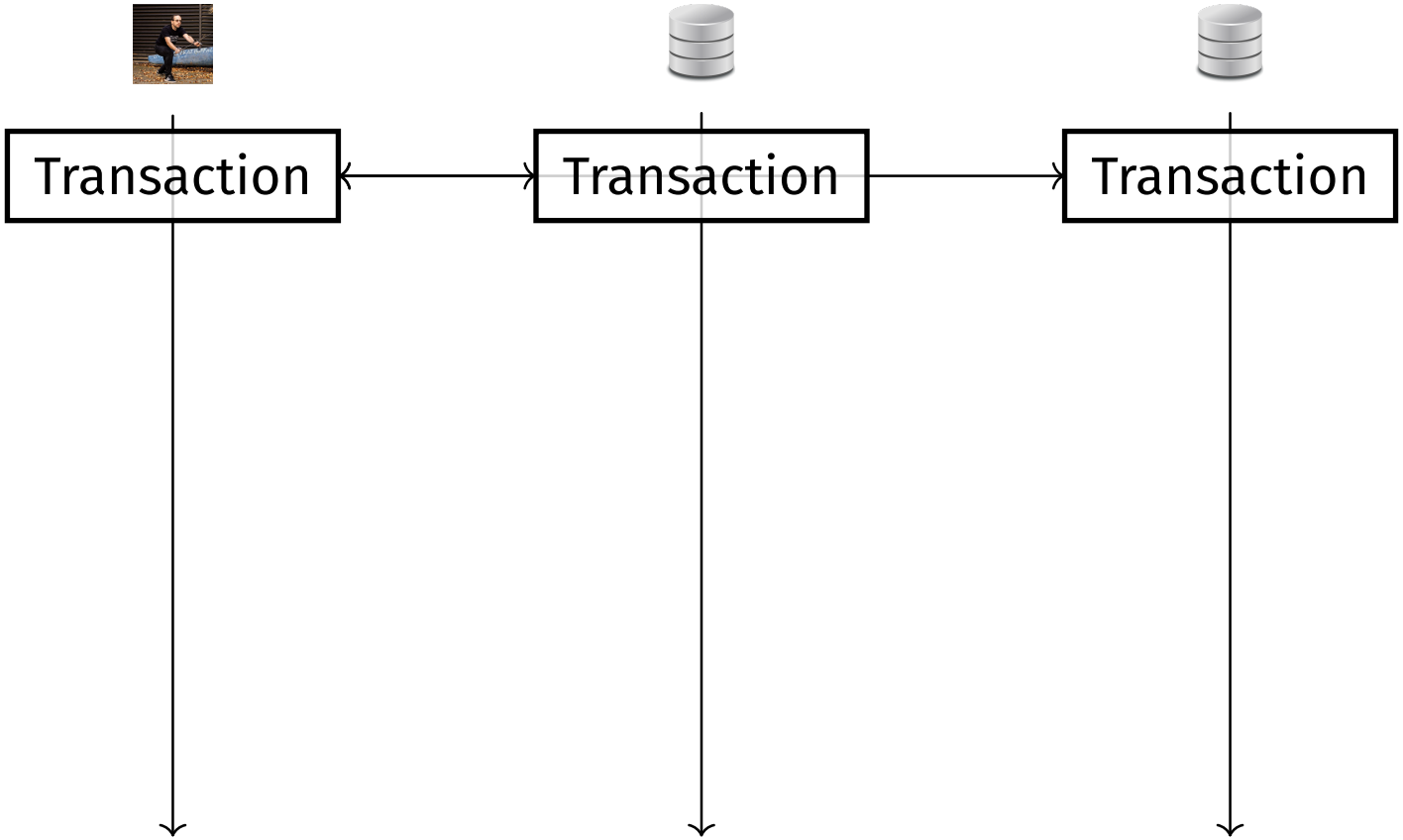
Committing


Transaction
Transaction
Transaction
Start Commit!
Commit!
Commit!

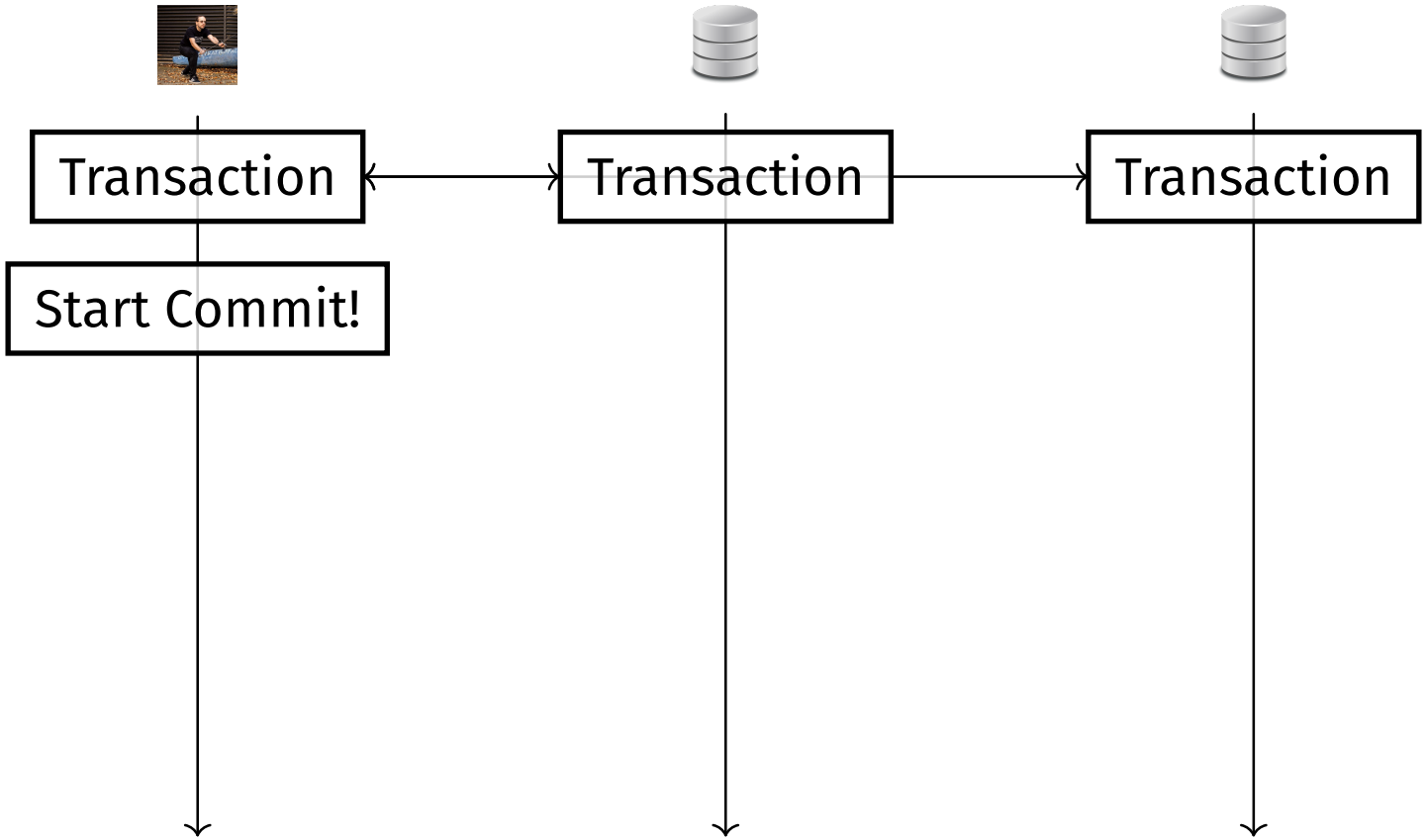
Committing



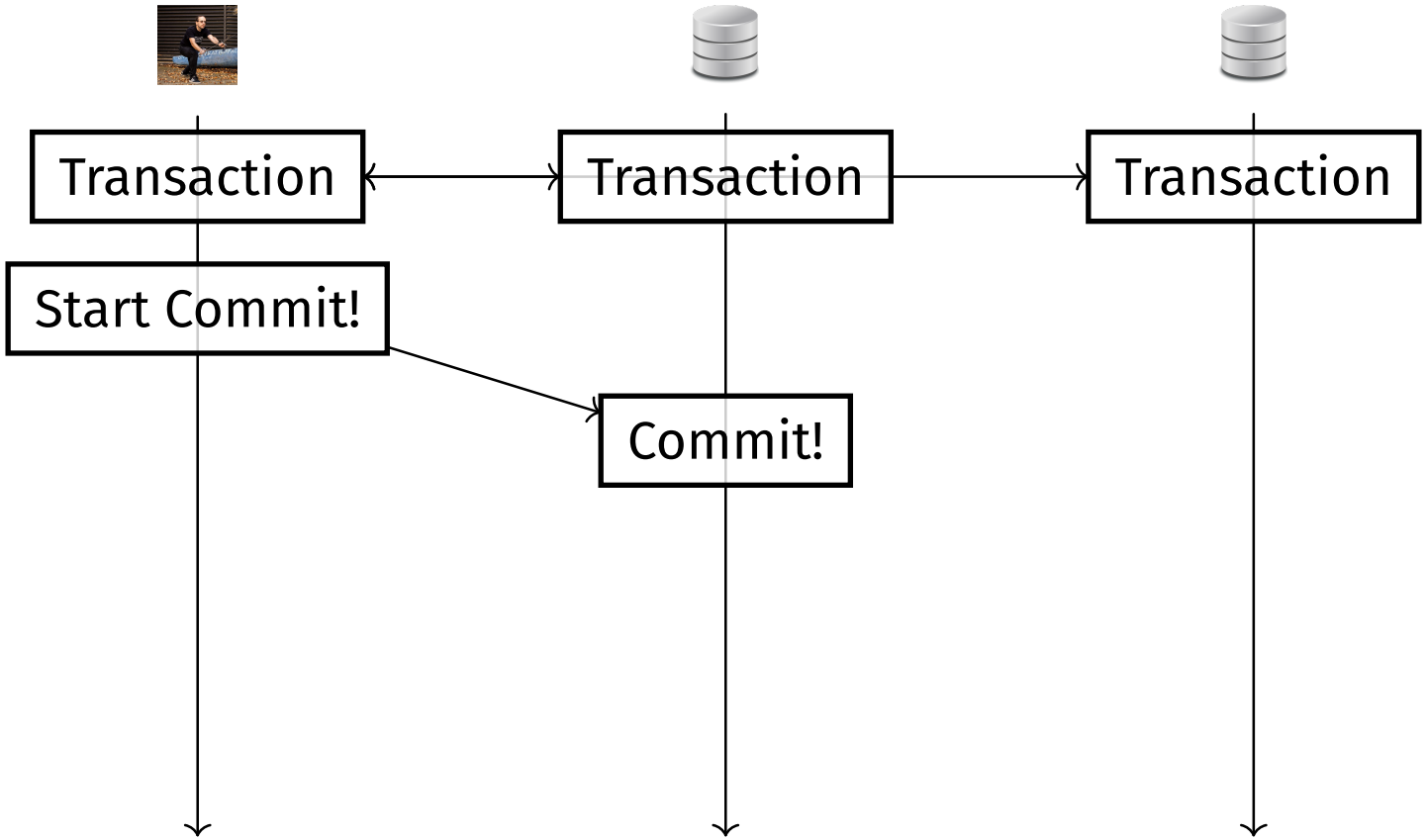
Committing



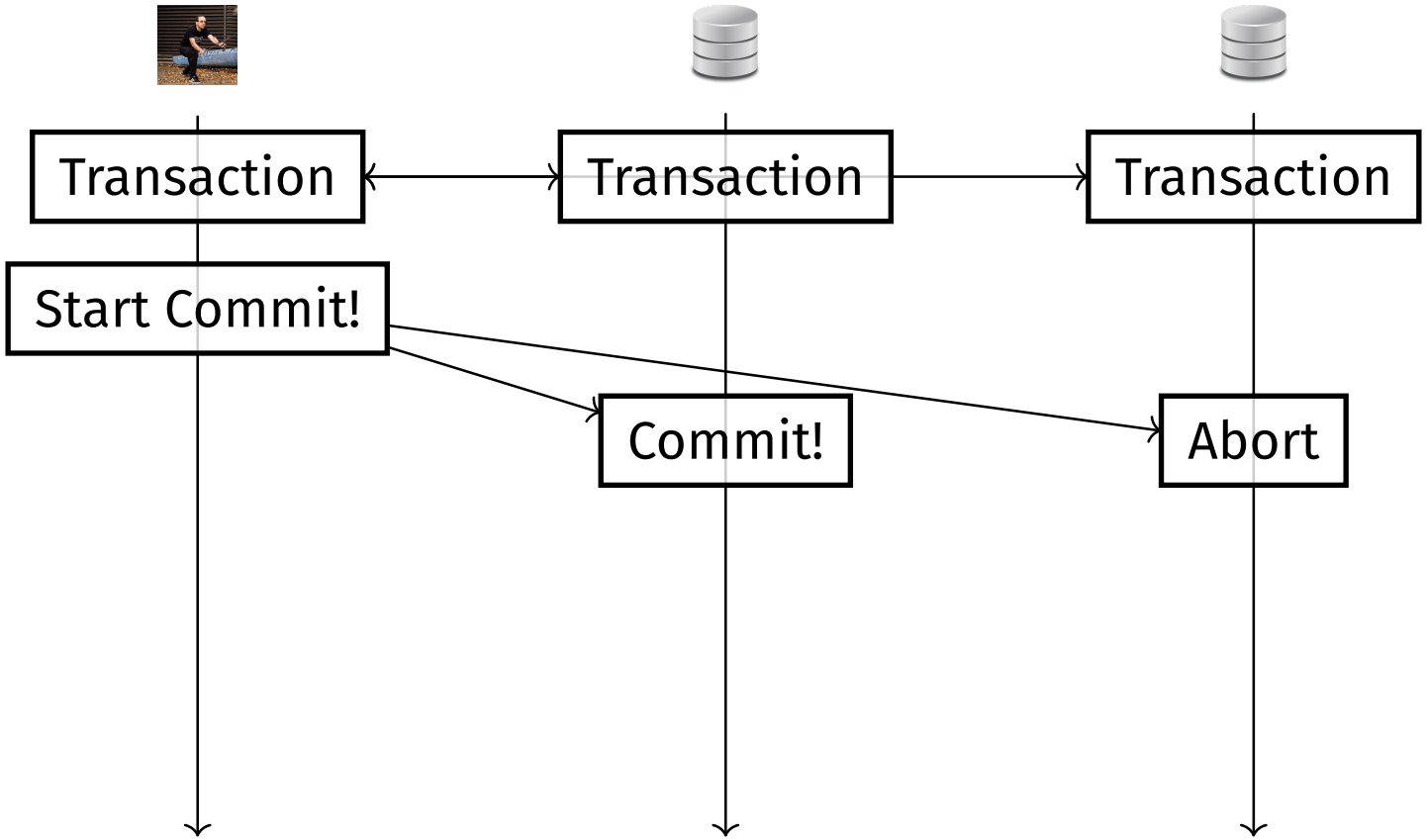
Committing



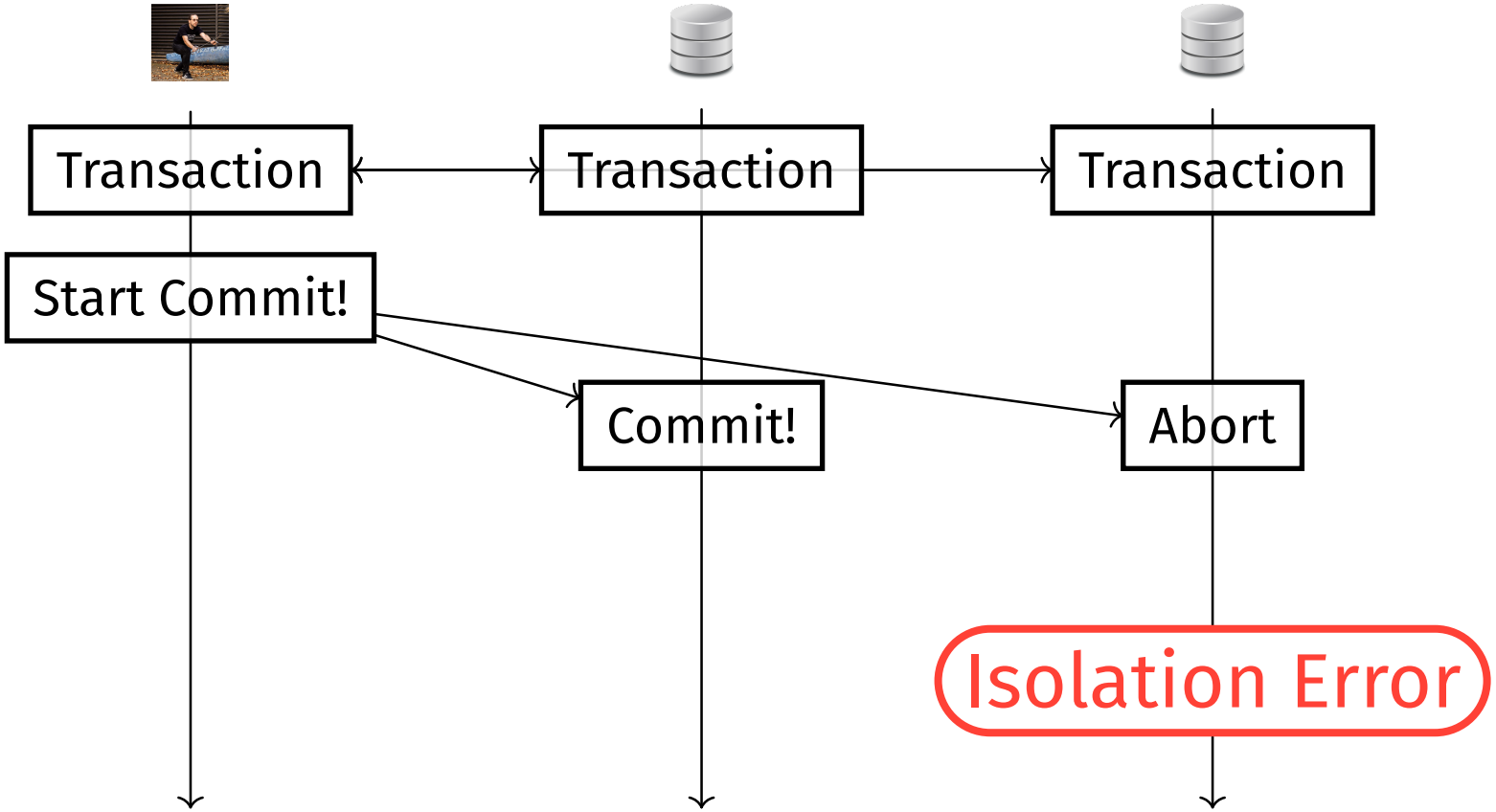
Committing


Transaction
Transaction
Transaction
Start Commit!
Commit!

Committing


Transaction
Transaction
Transaction
Start Commit!
Commit!
Abort

Committing


Transaction
Transaction
Transaction
Start Commit!
Commit!
Abort
Isolation Error

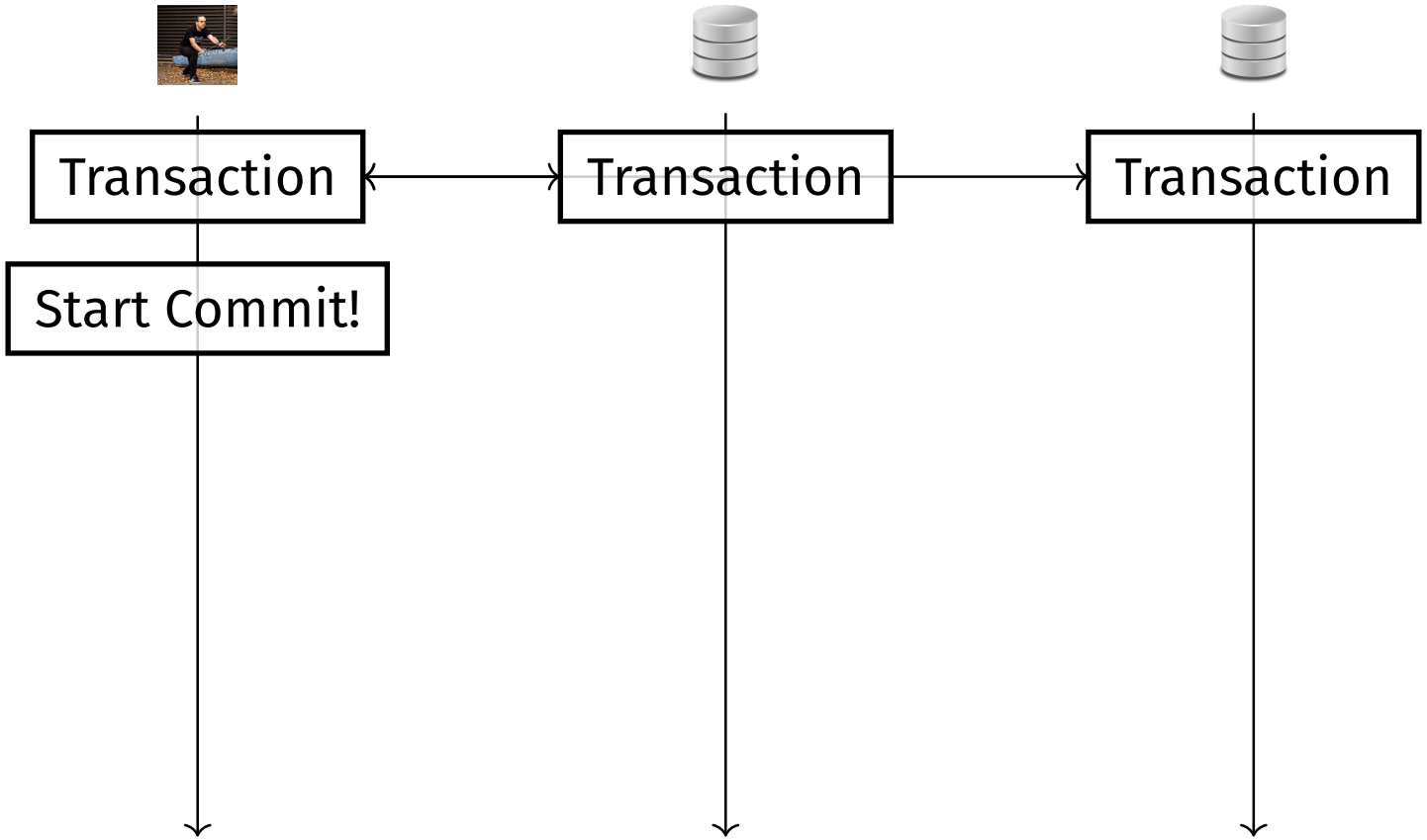
Multiple things could break during commit

1. Optimistic concurrency control detects a concurrency violation
2. Hardware failure violates durability
3. Network failure prevents commits.

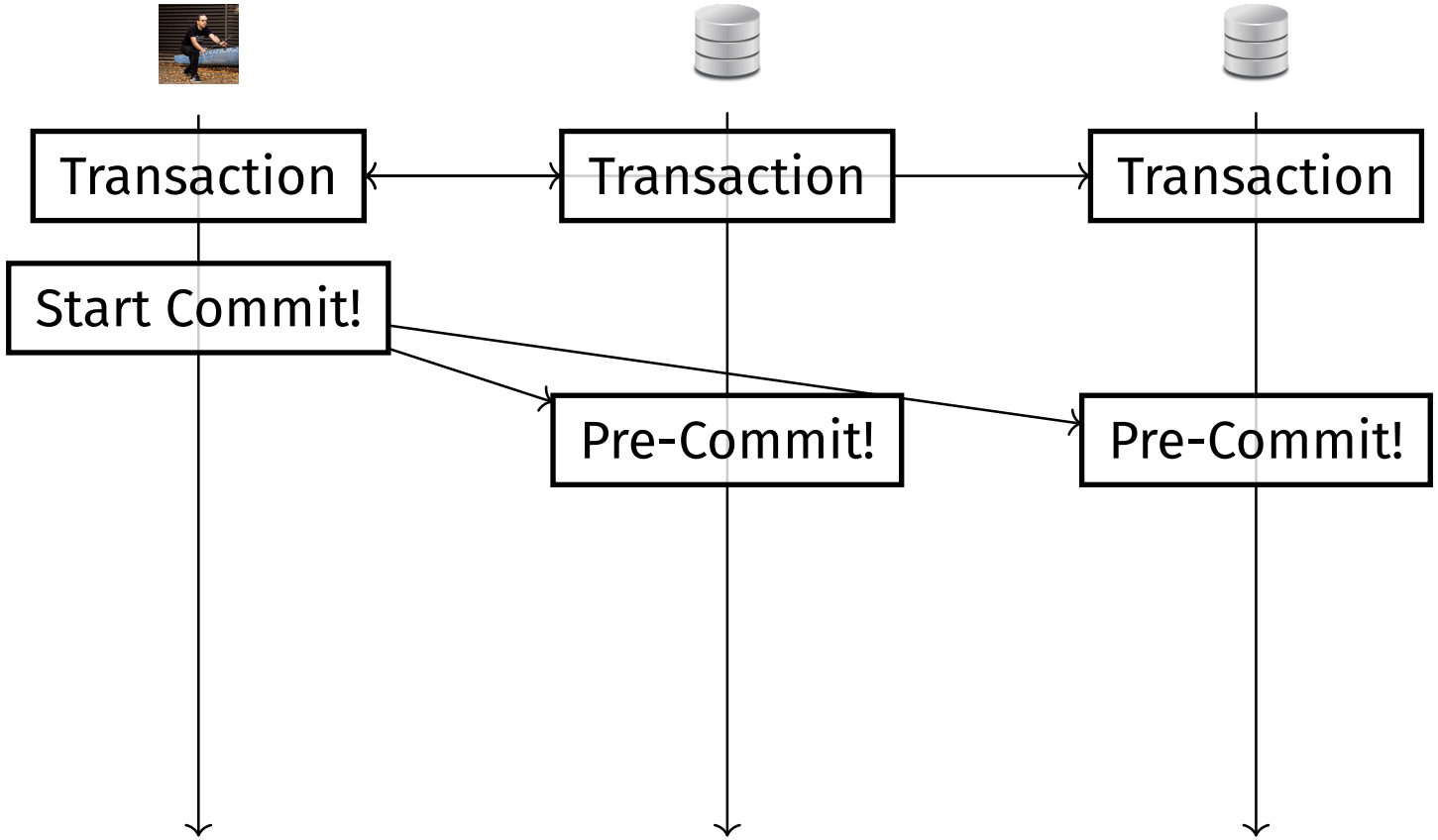
Committing With Validation

Idea: Add a validation step.

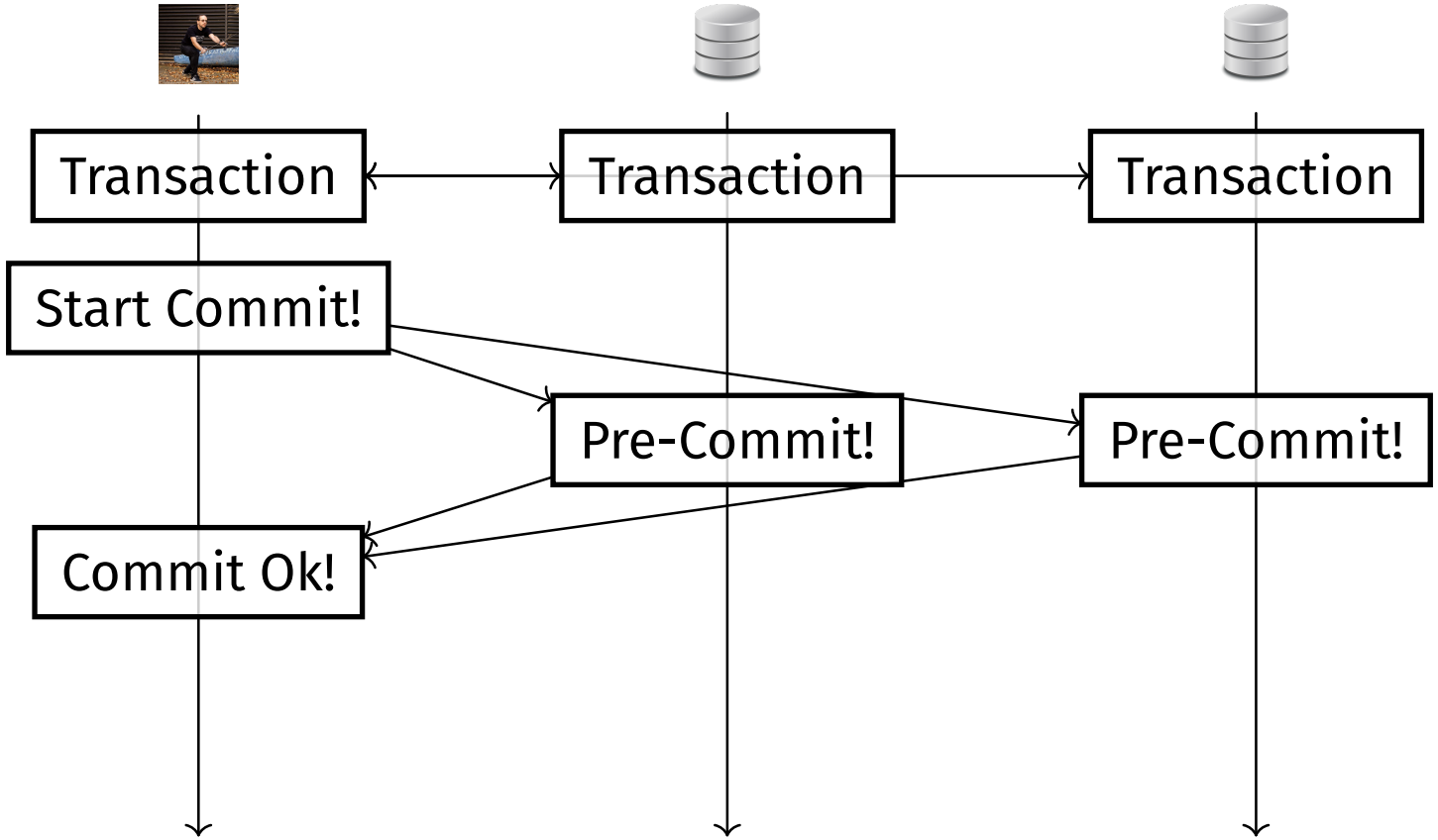
Committing With Validation



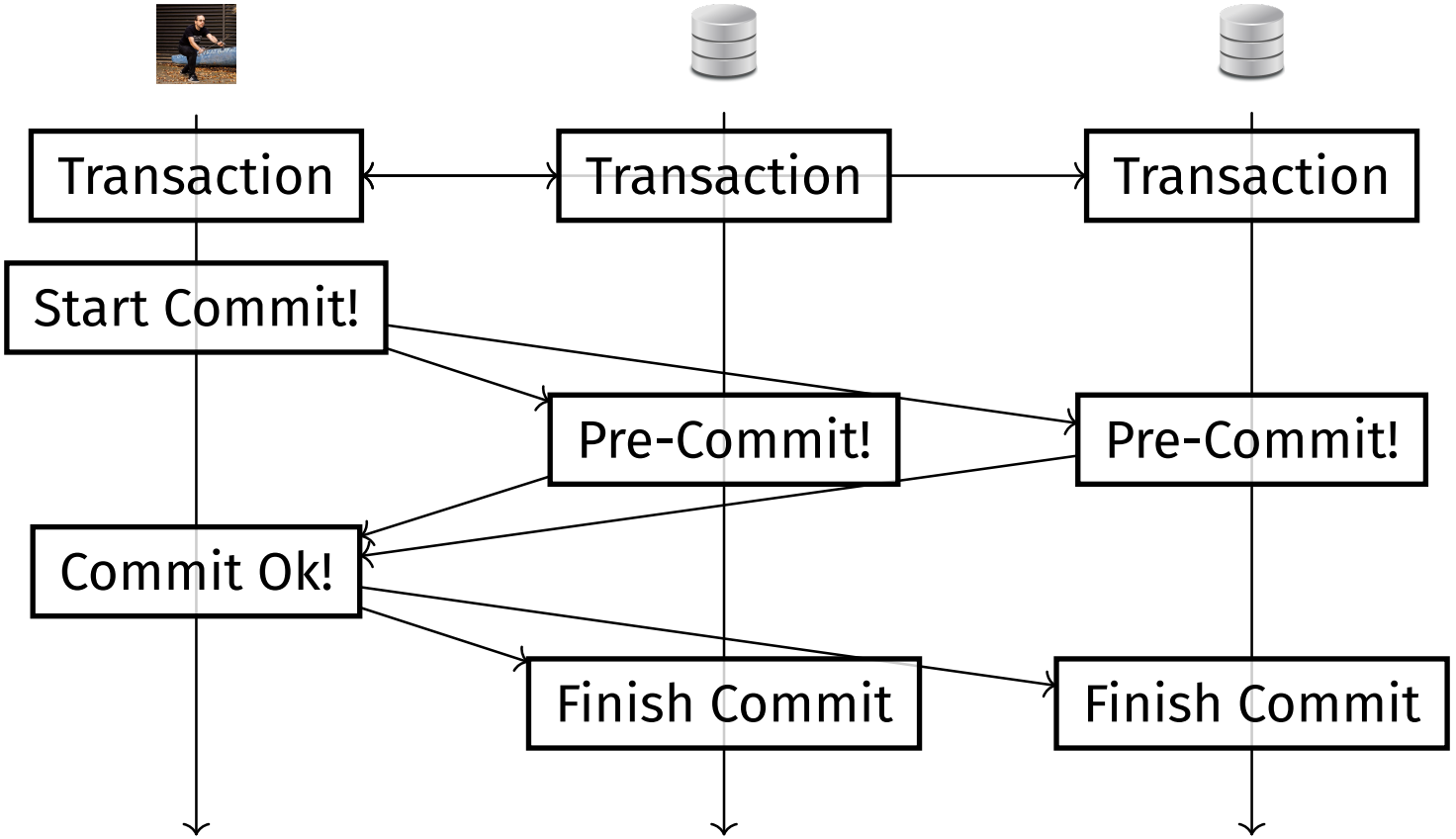
Committing With Validation



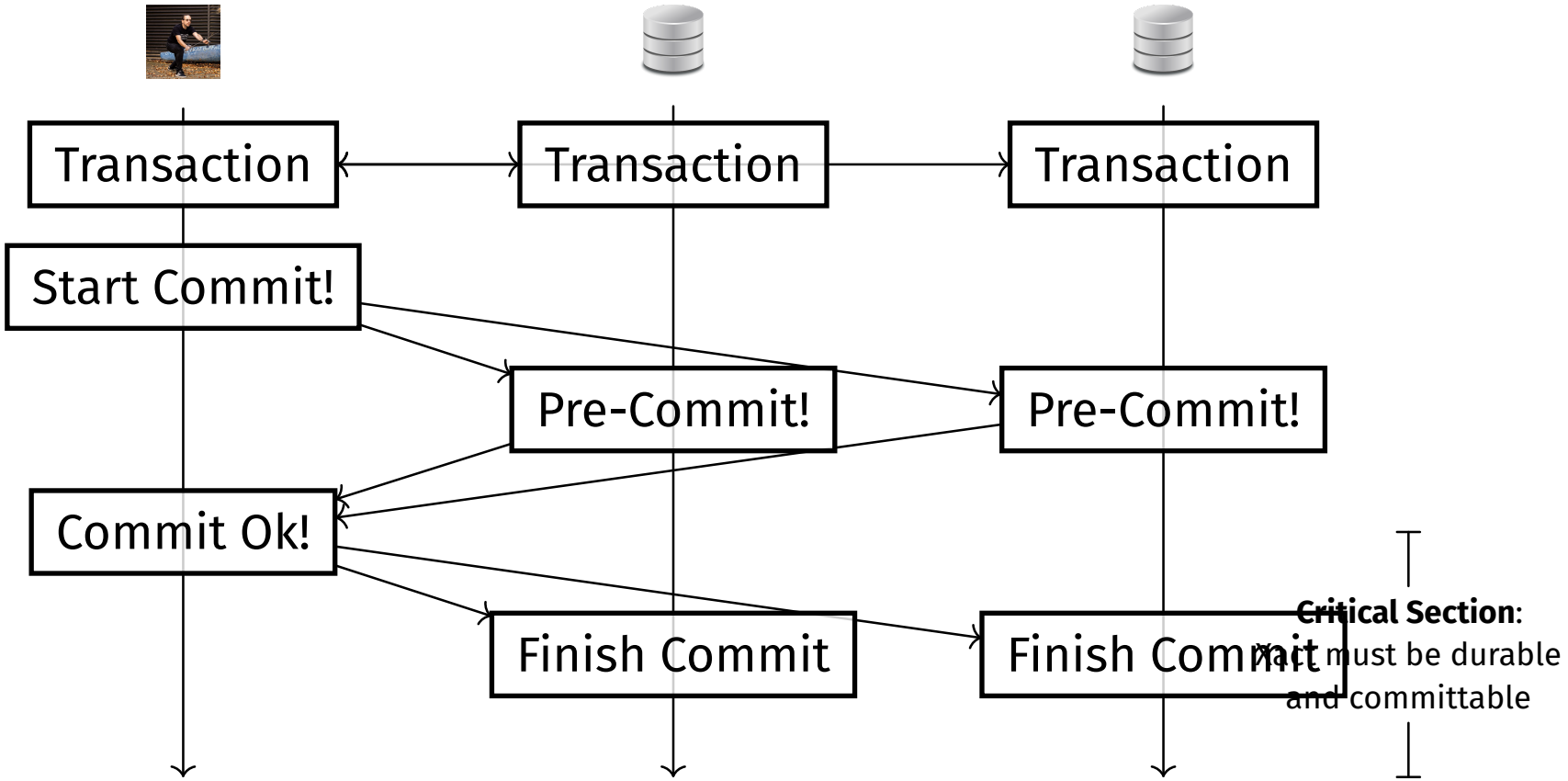
Committing With Validation



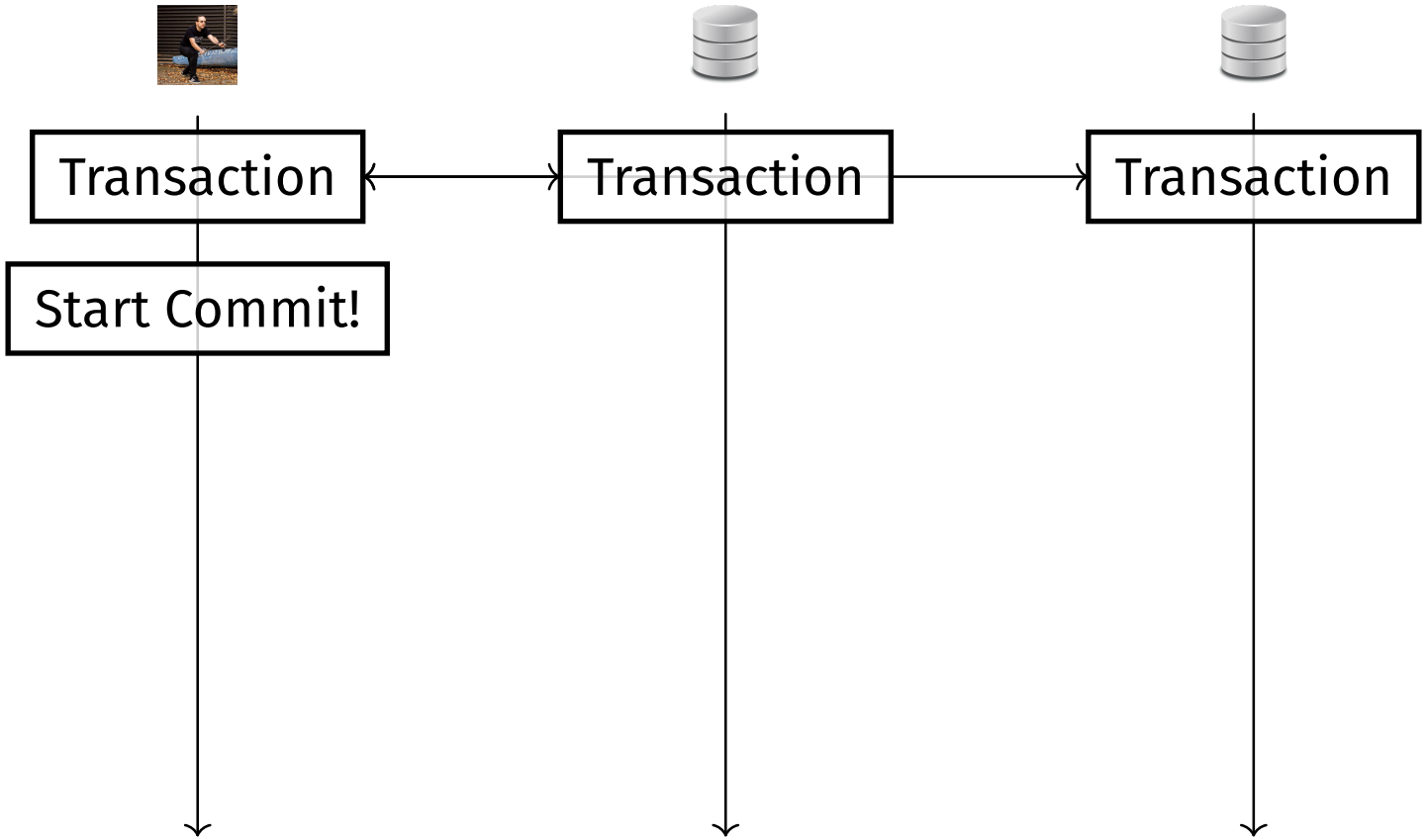
Committing With Validation



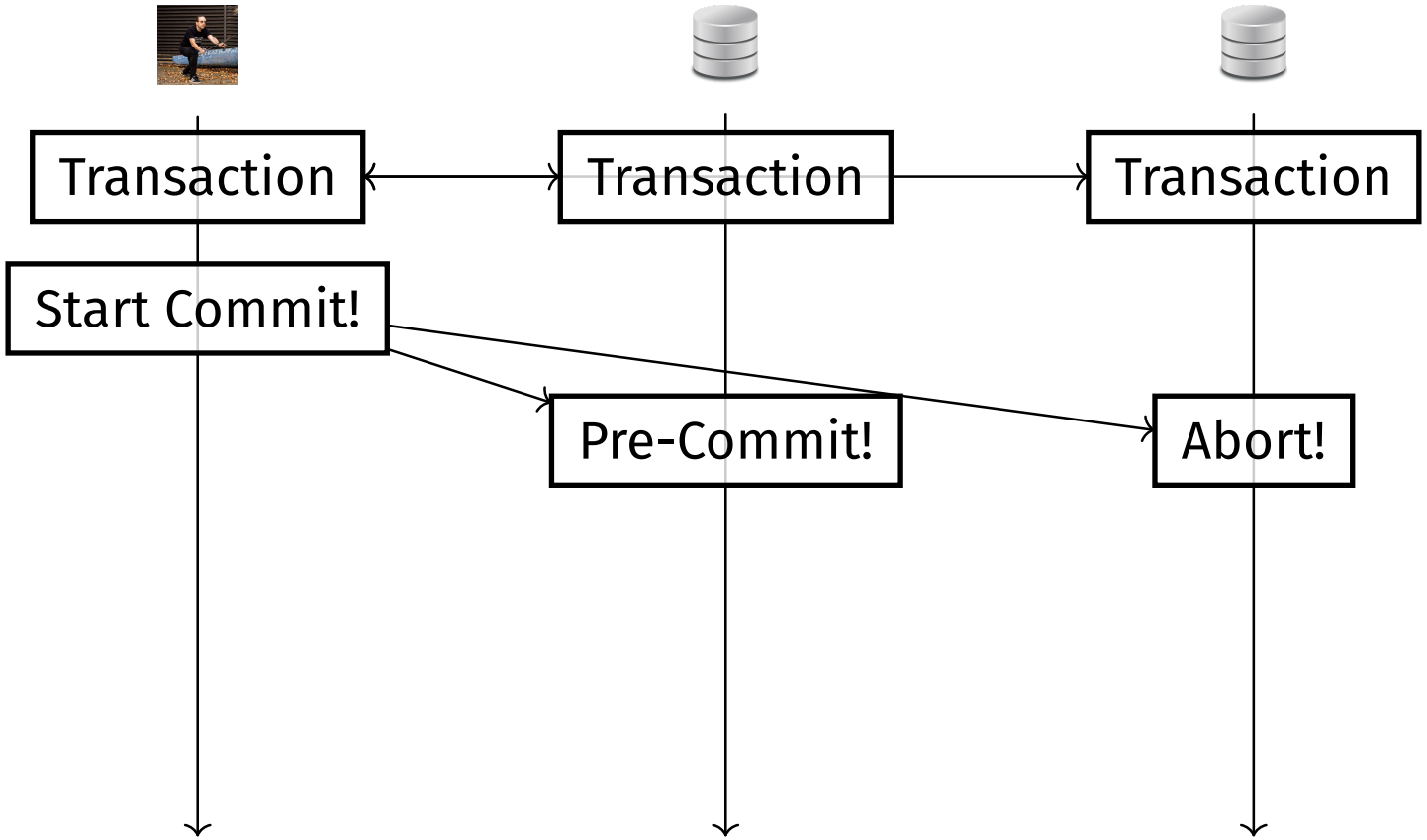
Committing With Validation



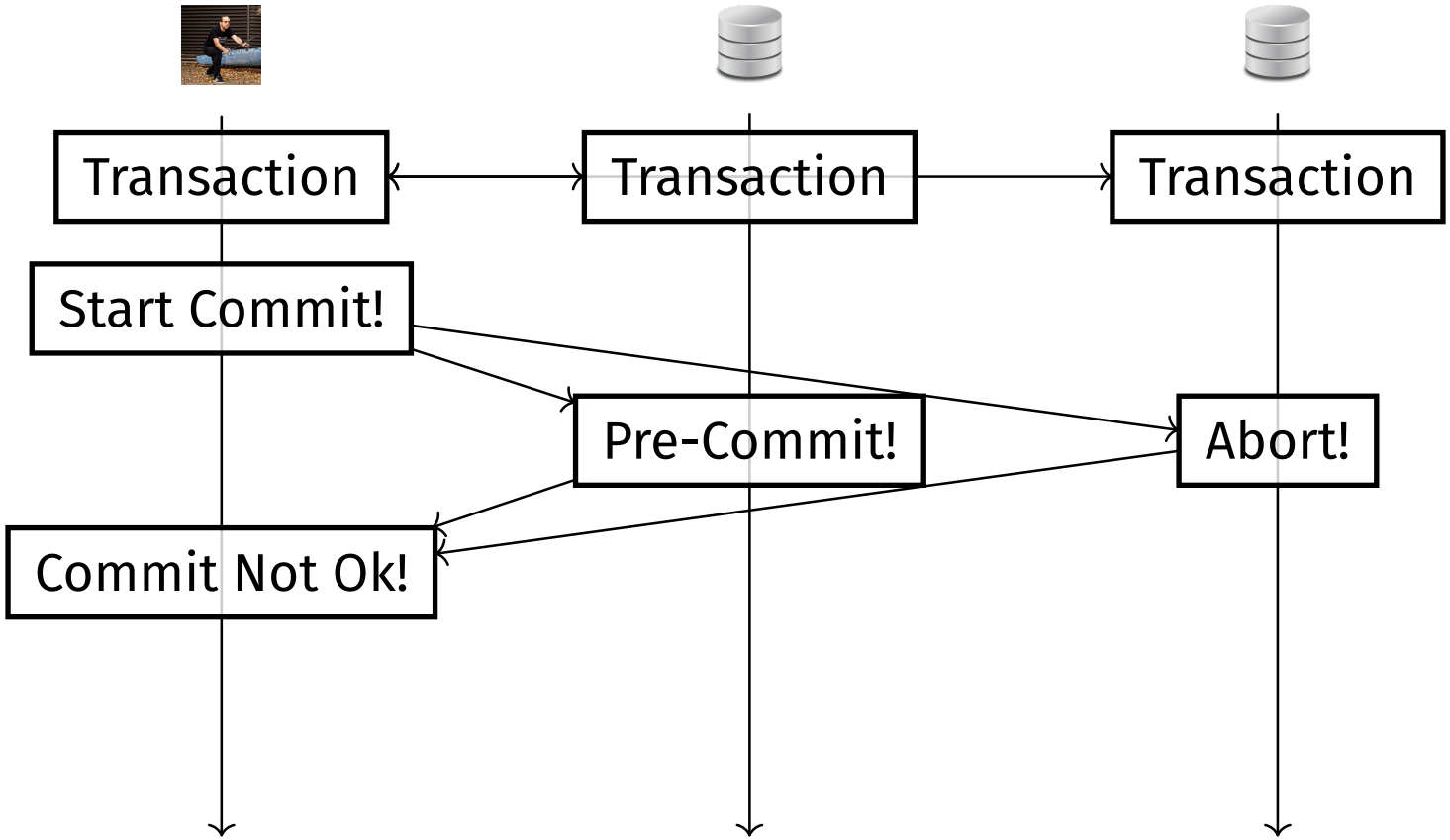
Committing With Validation



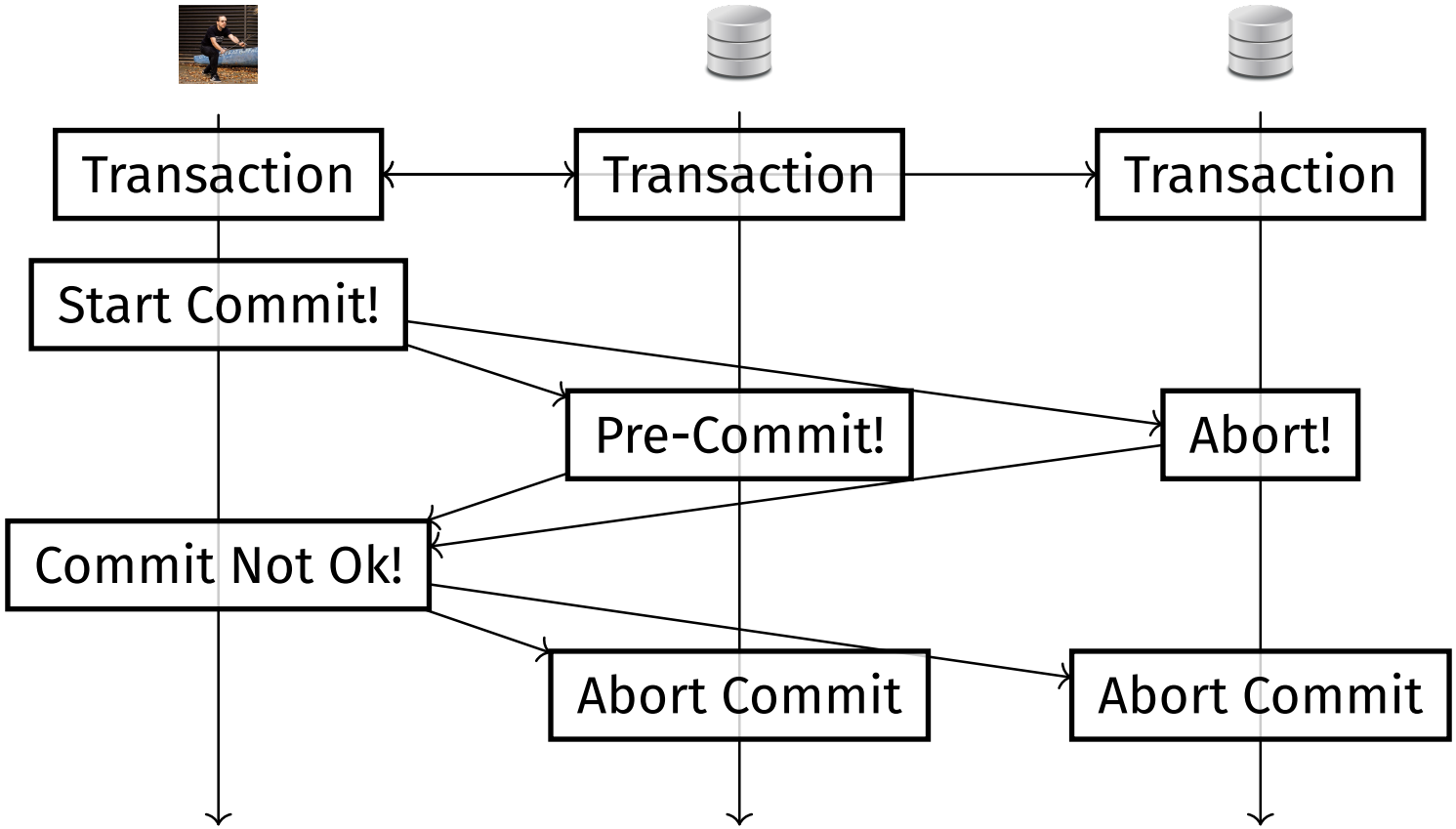
Committing With Validation



Committing With Validation



Committing With Validation



2-Phase Commit

This is called 2-Phase Commit

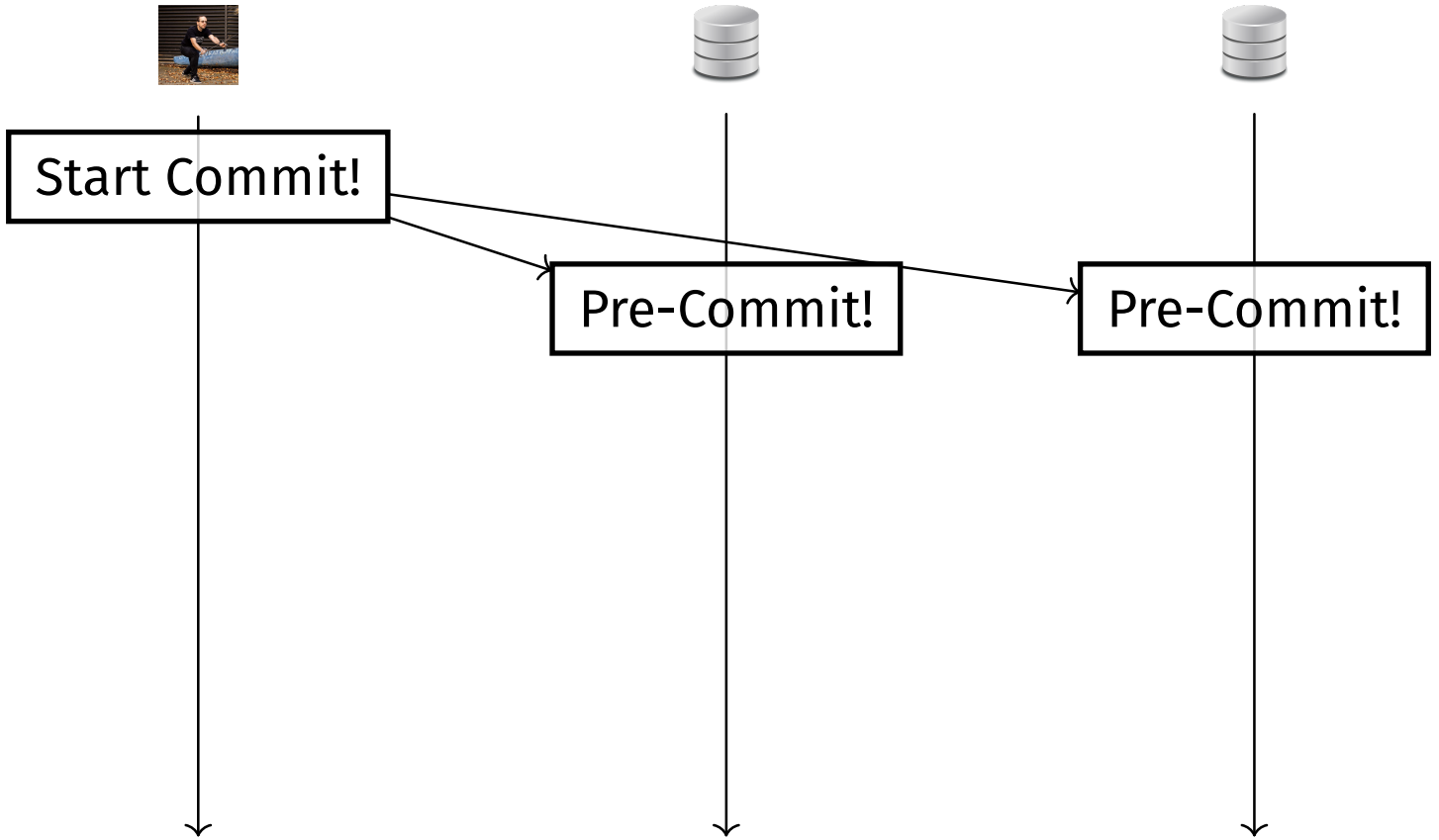
Failure Analysis

What kinds of failures can we tolerate?

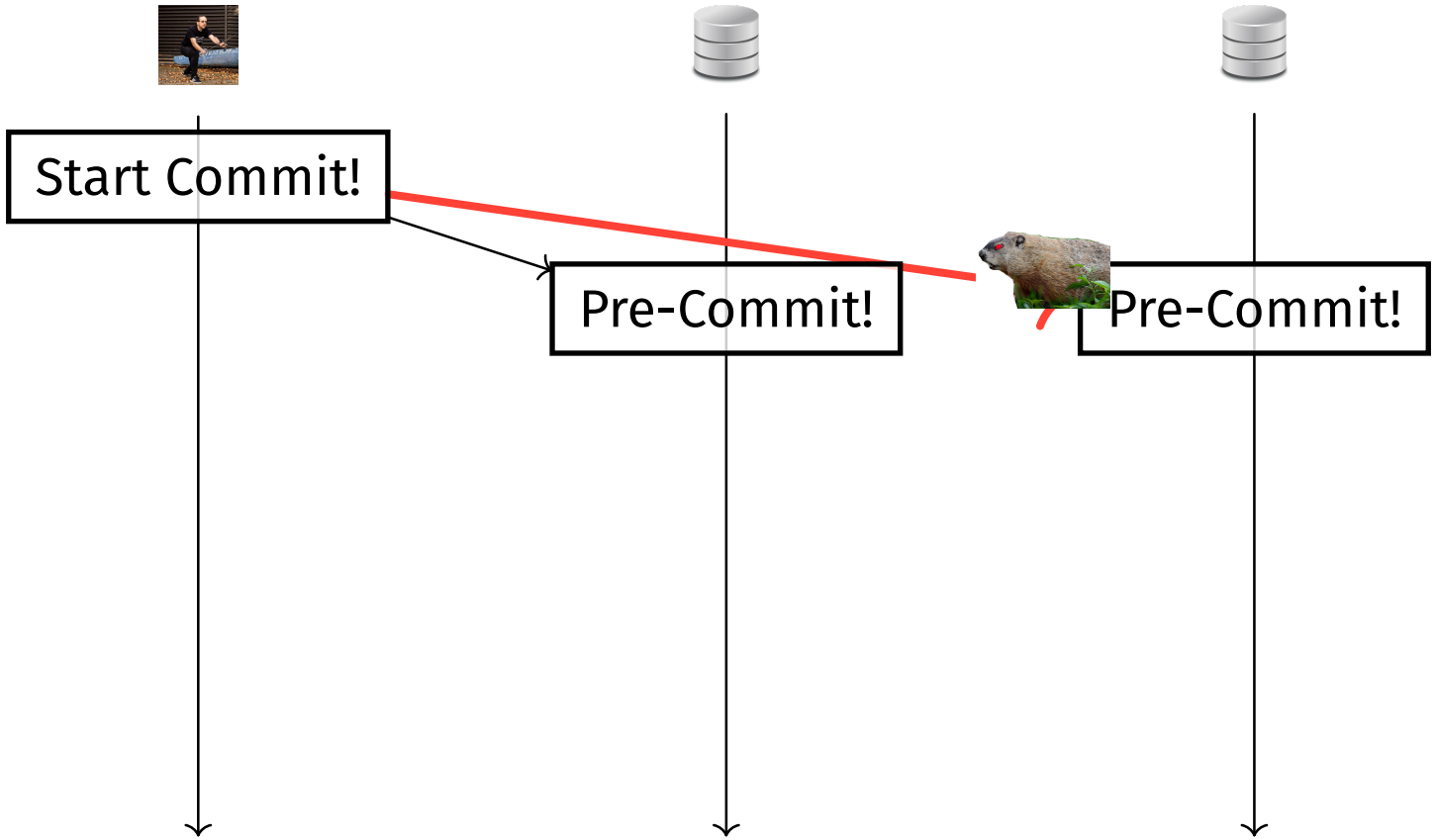
What kinds of failures can we tolerate?

- A message is dropped
- A participant crashes recoverably (power outage)
- A participant crashes non-recoverably (marmots)

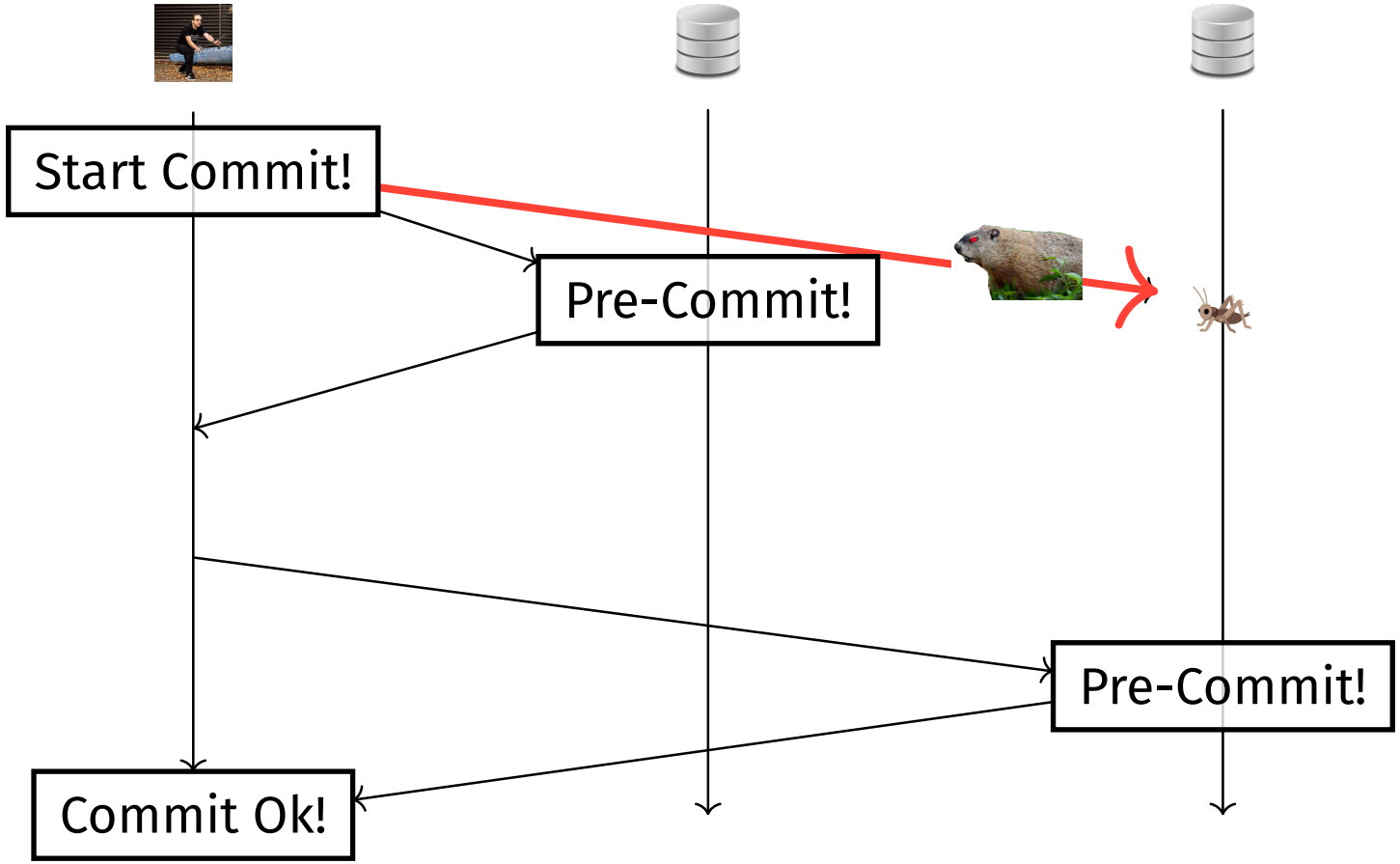
Failure Analysis



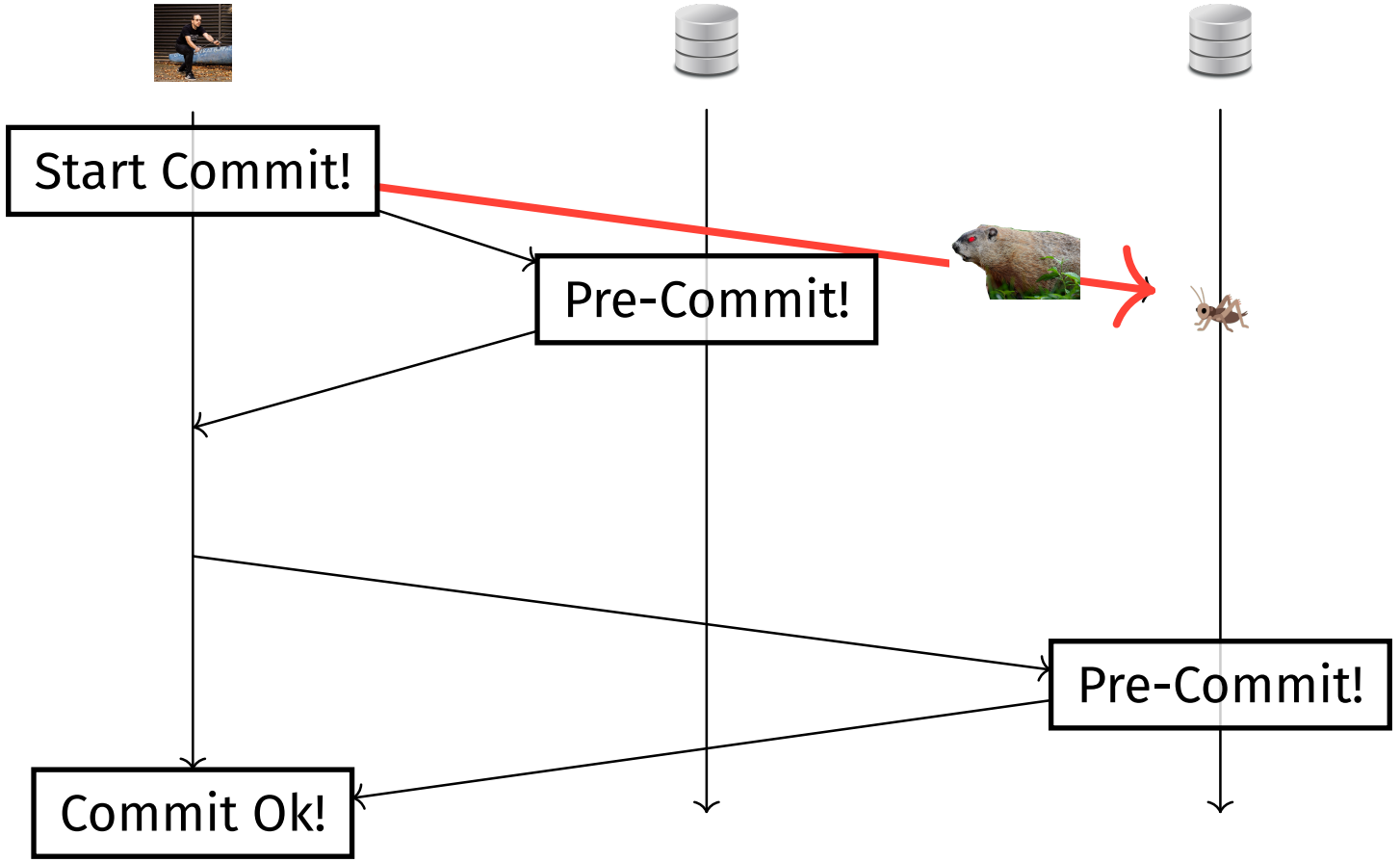
Failure Analysis



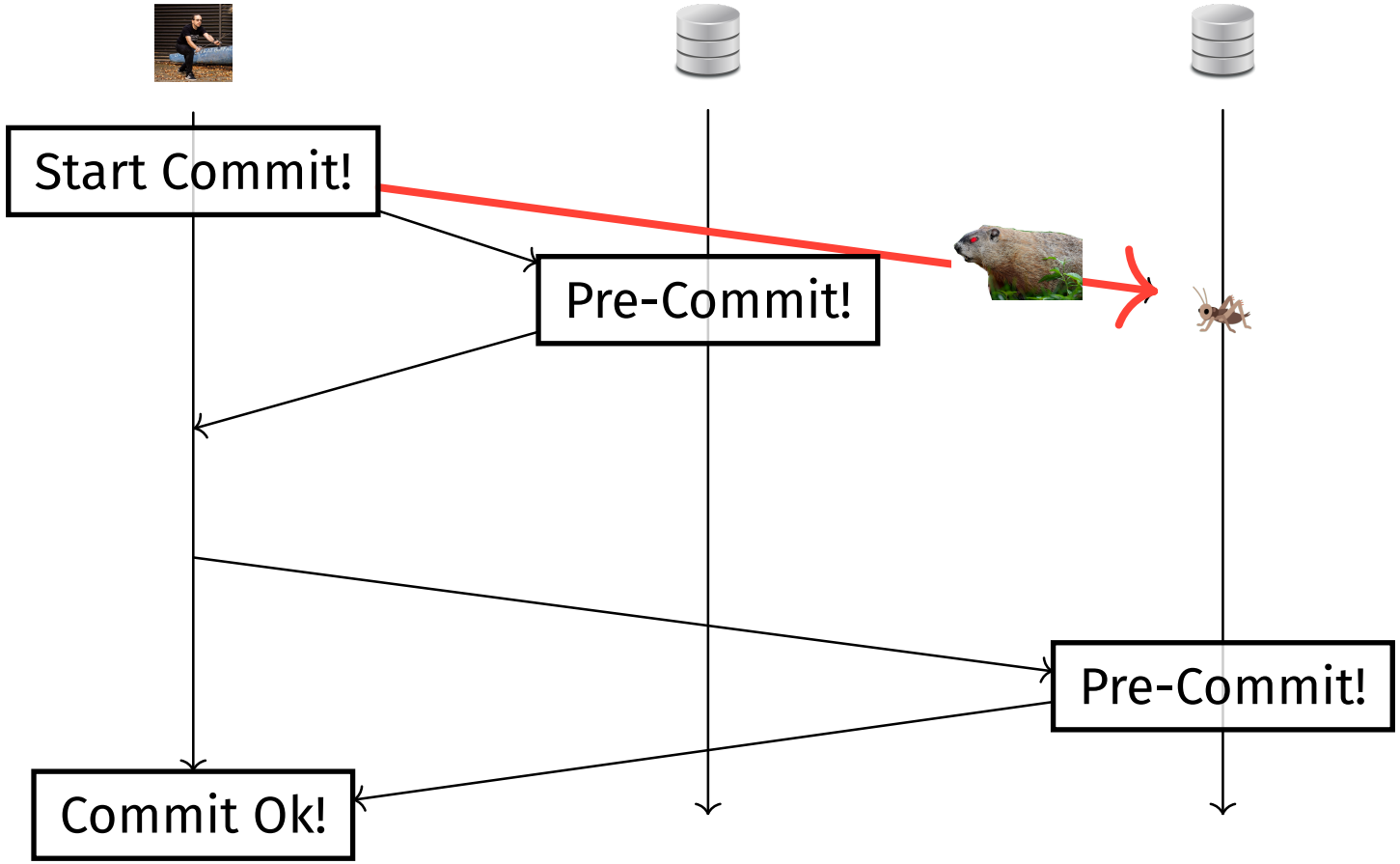
Failure Analysis



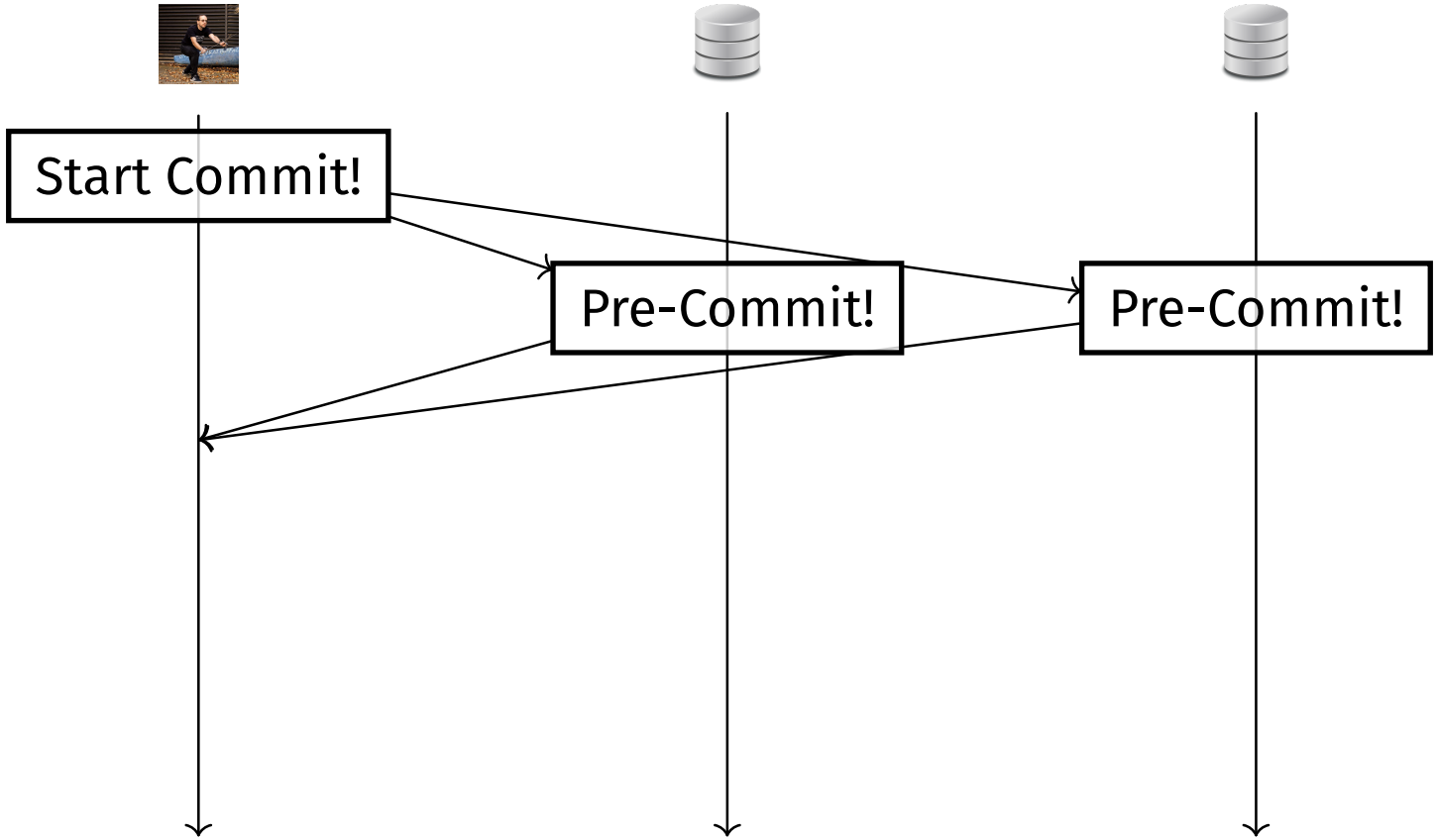
Failure Analysis



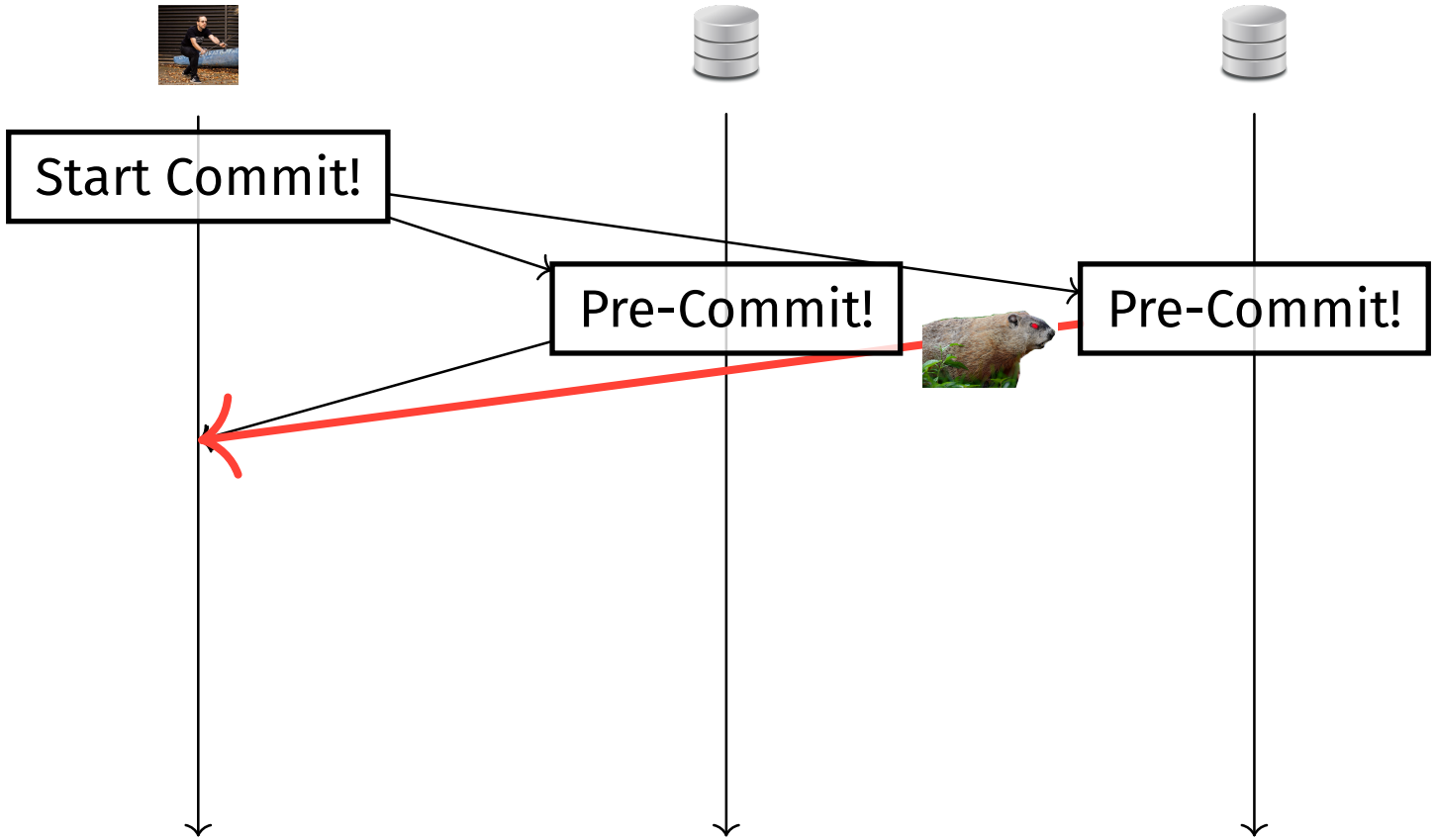
Failure Analysis



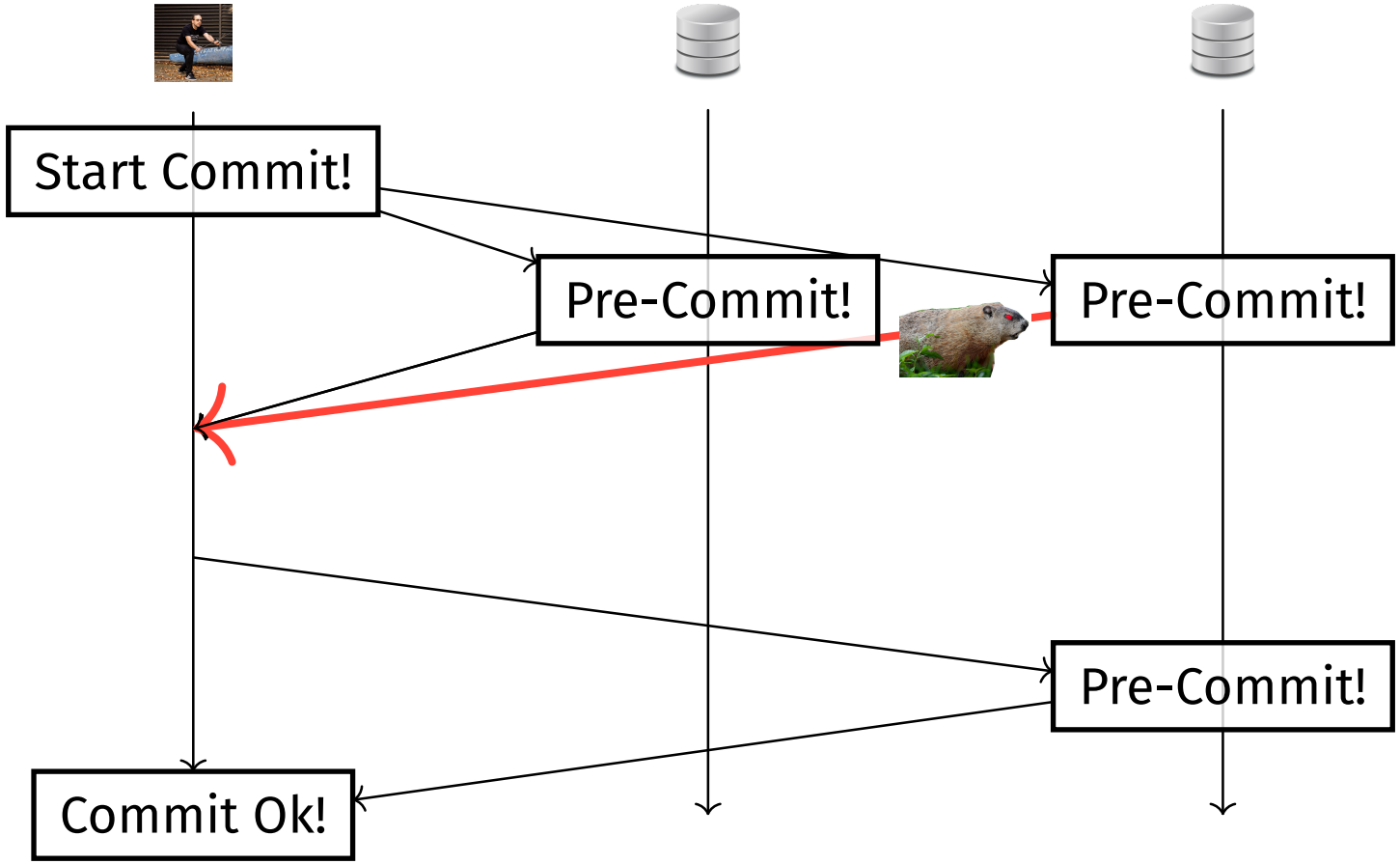
Failure Analysis



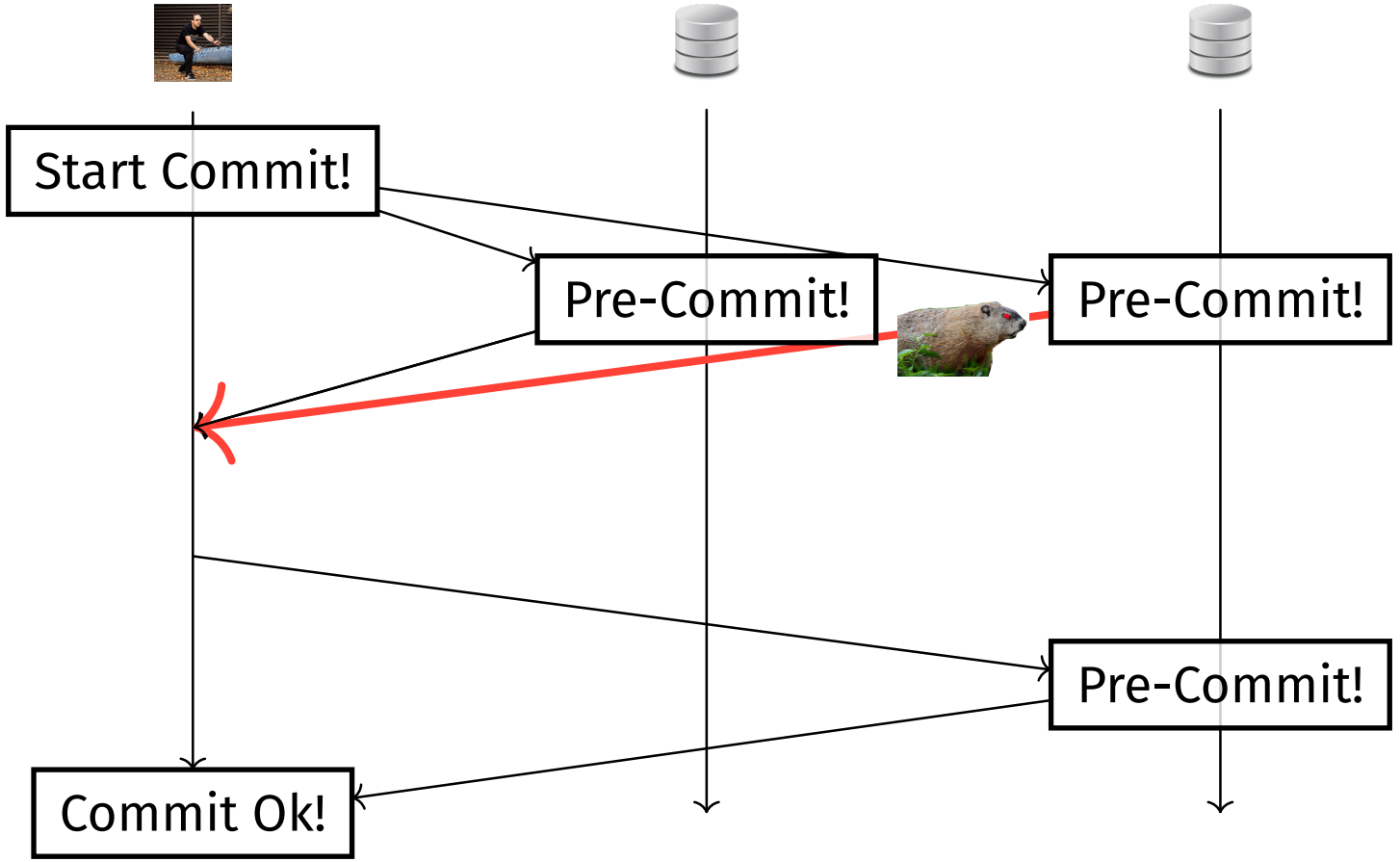
Failure Analysis



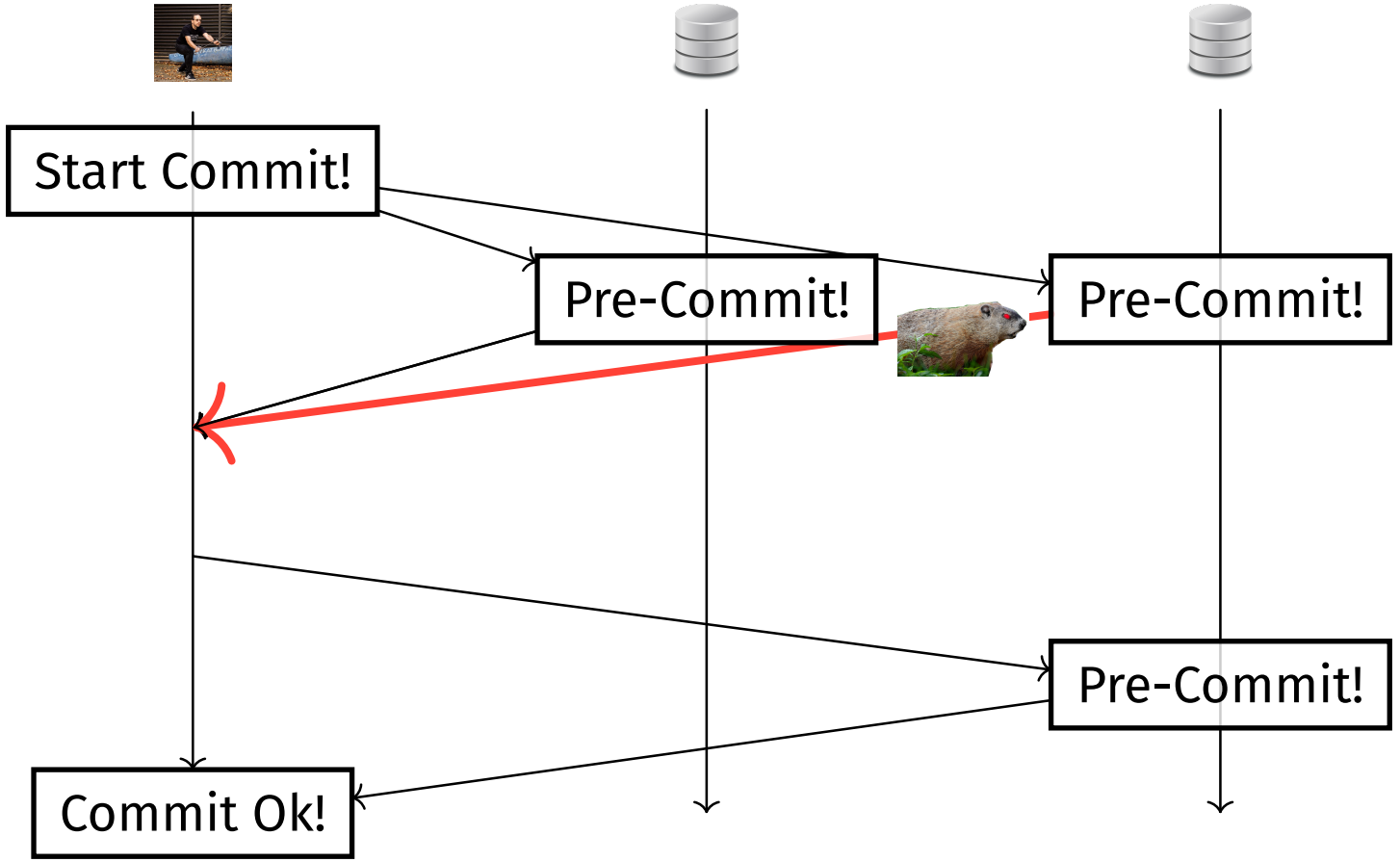
Failure Analysis



Failure Analysis



Failure Analysis



What kinds of failures can we tolerate?

- A message is dropped
- A participant crashes recoverably (power outage)
- A participant crashes non-recoverably (marmots)

Assumptions

1. Each node can enforce ACID **locally**.
2. No malicious nodes.
3. Messages arrive in full or not at all.
4. No global state.

Durability

If the node guarantees **Durability**,
transient failures are no different from message loss.

What kinds of failures can we tolerate?

- A message is dropped
- A participant crashes recoverably (power outage)
- A participant crashes non-recoverably (marmots)

Durability

When is the write durable?

When is the write durable?

Never!

Durability

When is the write durable?

Never!

More durability = More costs

Replication

Idea: Replicate data to multiple nodes/sites/planets...

Simple Replication

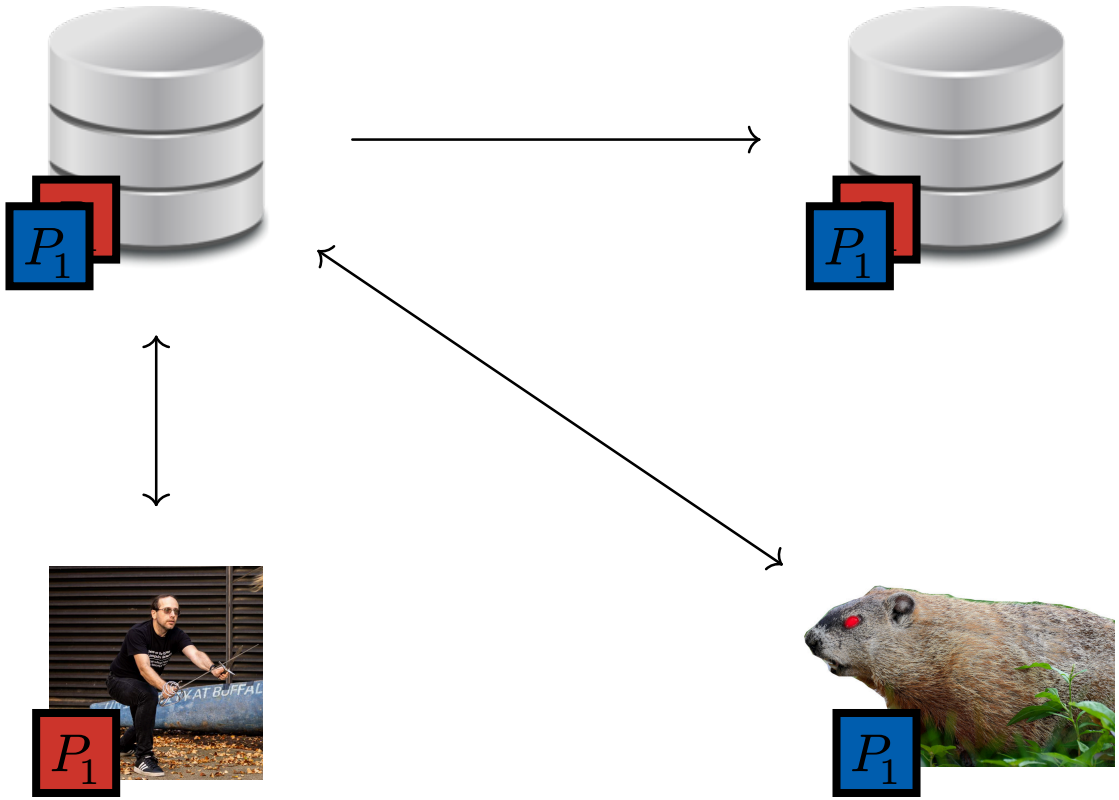


Image credit: Openclipart.org

Simple Replication

Pros

Cons

Pros

- A backup node exists to come back up when a node is eaten by marmots.

Cons

Pros

- A backup node exists to come back up when a node is eaten by marmots.

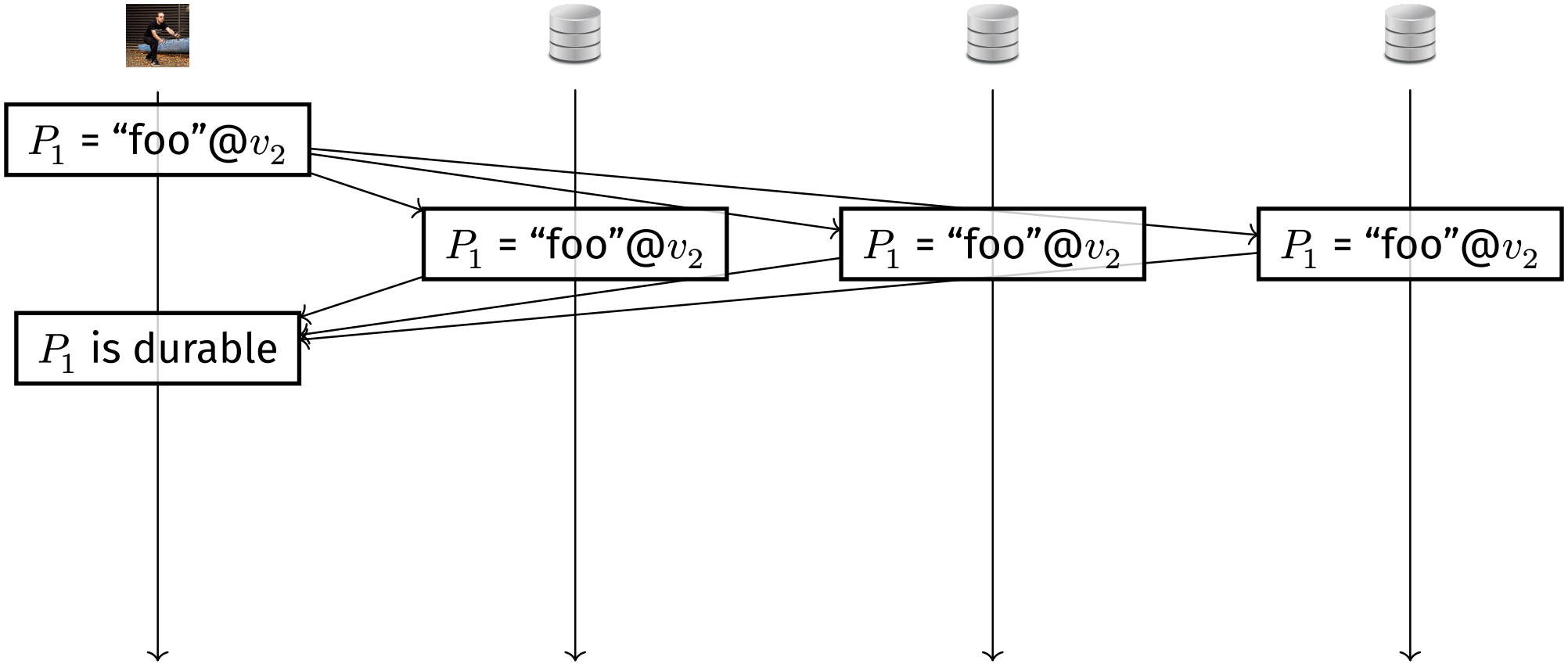
Cons

- Have to wait two message round trips (or more) to commit.

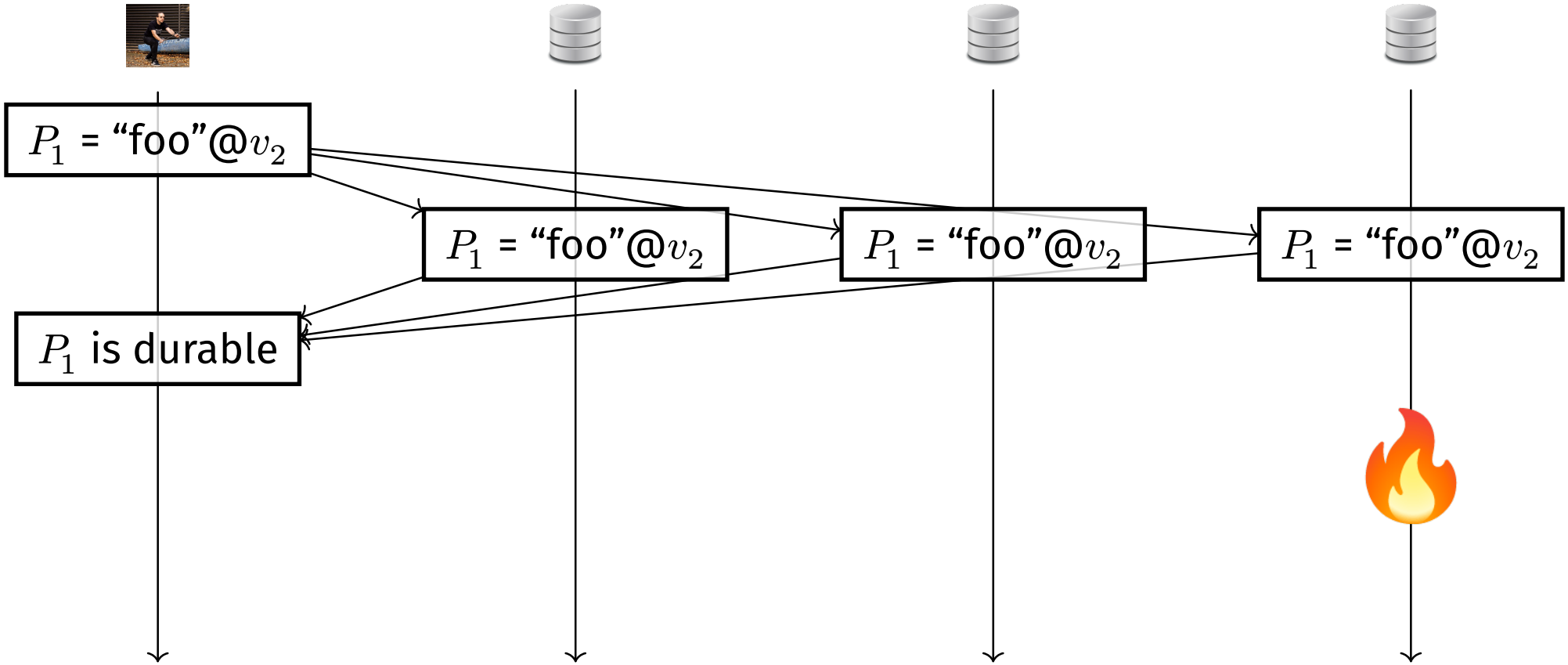
Idea 2: Perform writes in parallel

(Reminder: We're only solving for durability here)

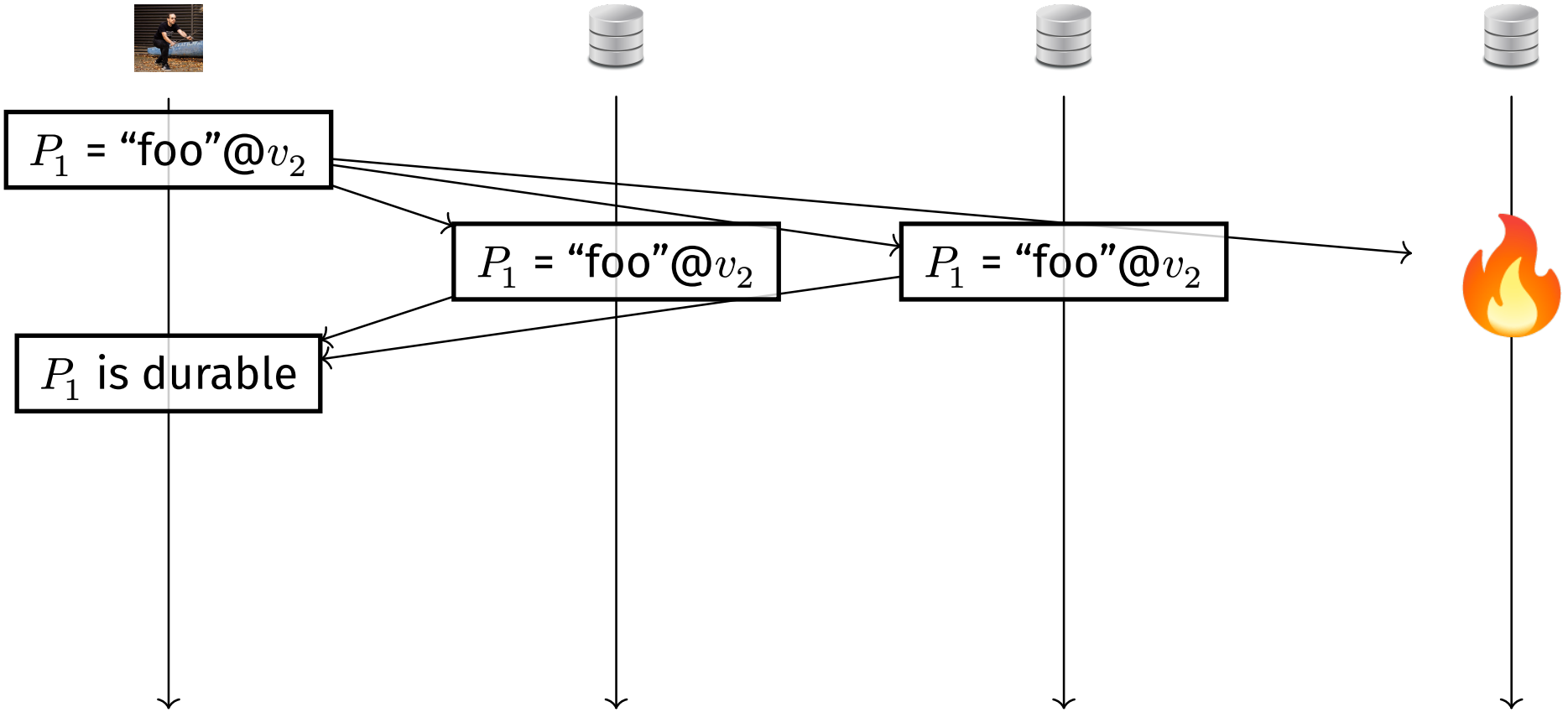
Replication



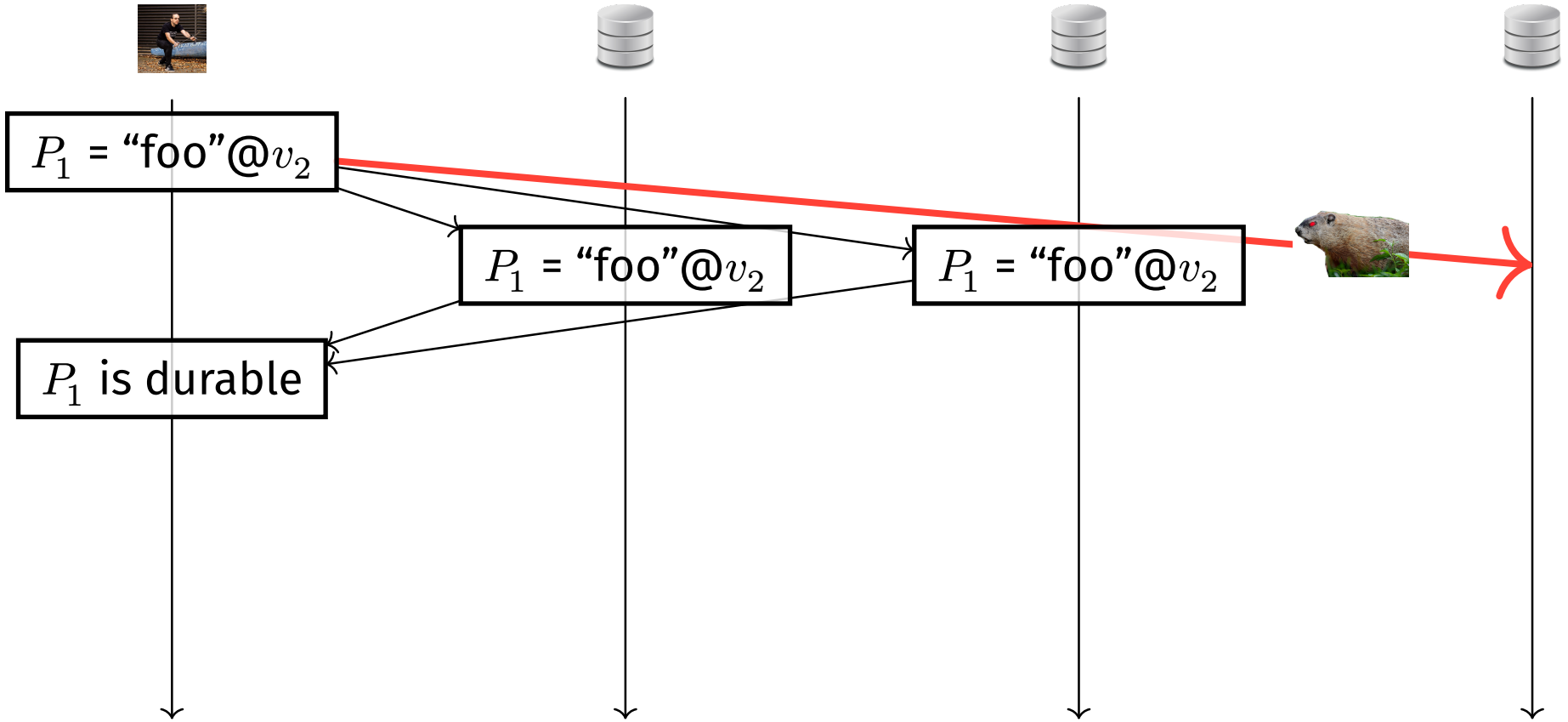
Replication



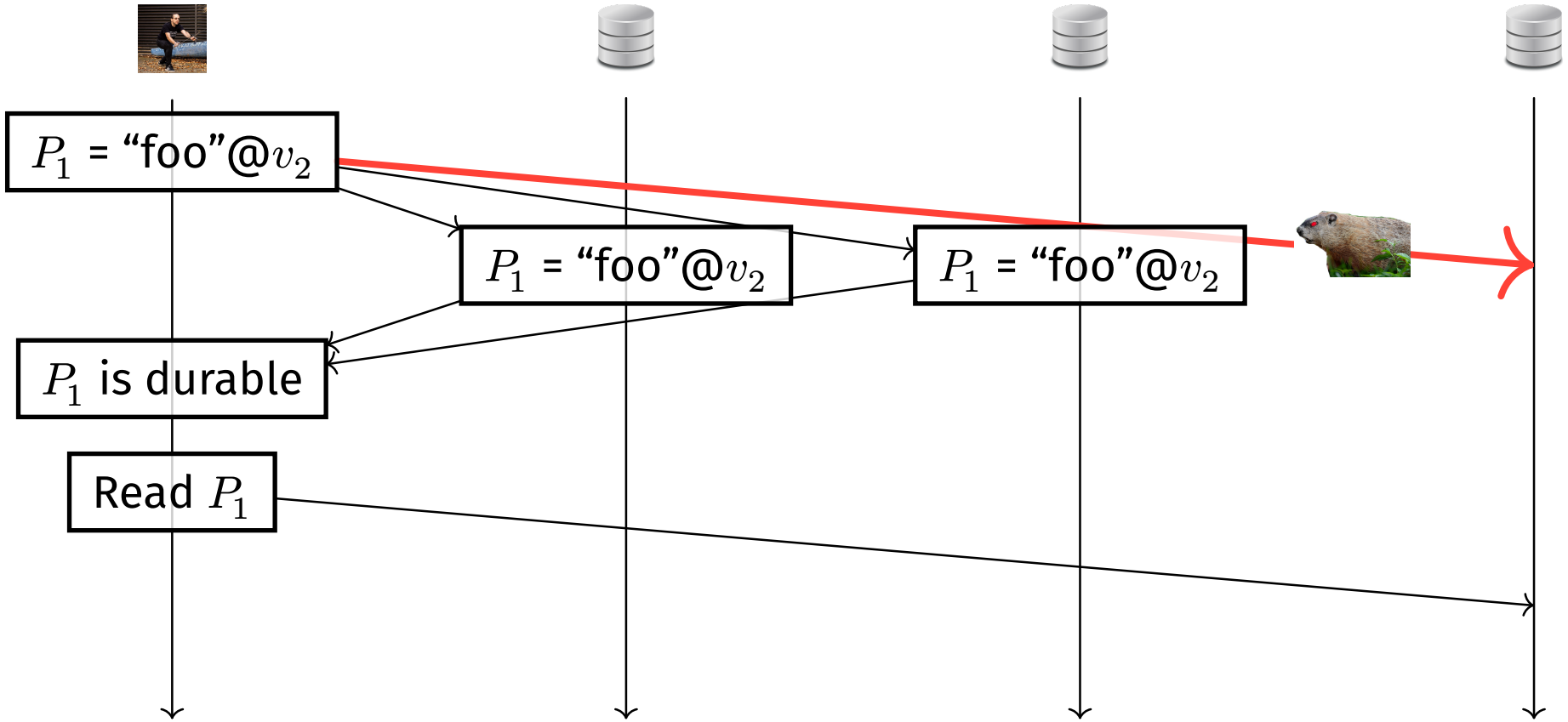
Replication



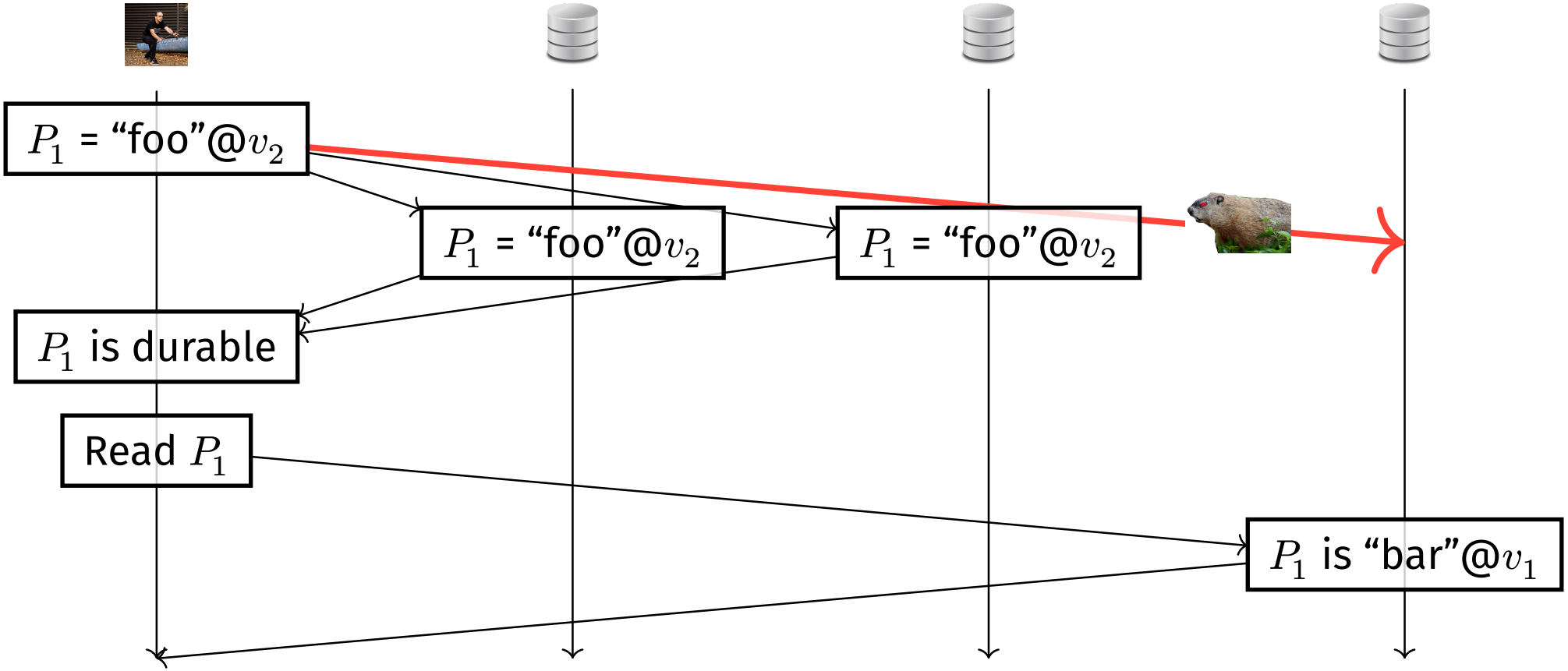
Replication



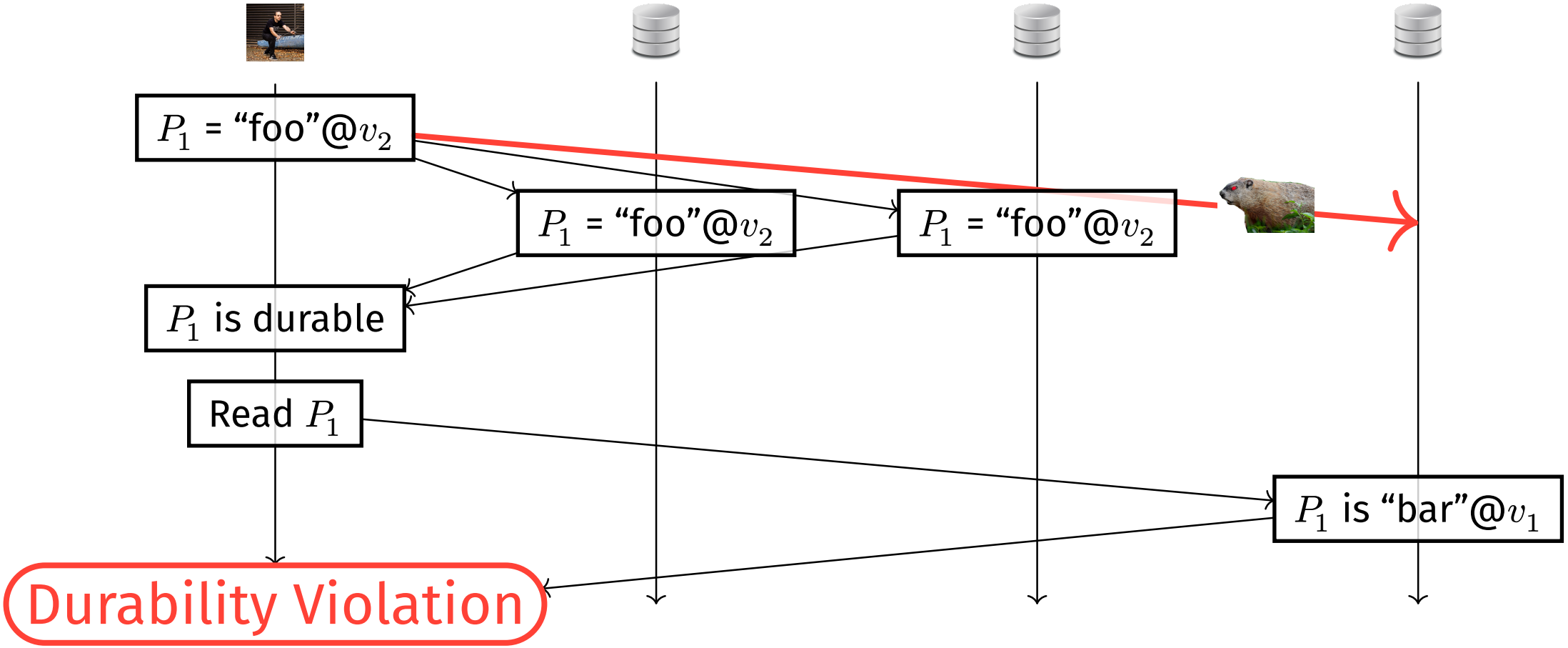
Replication



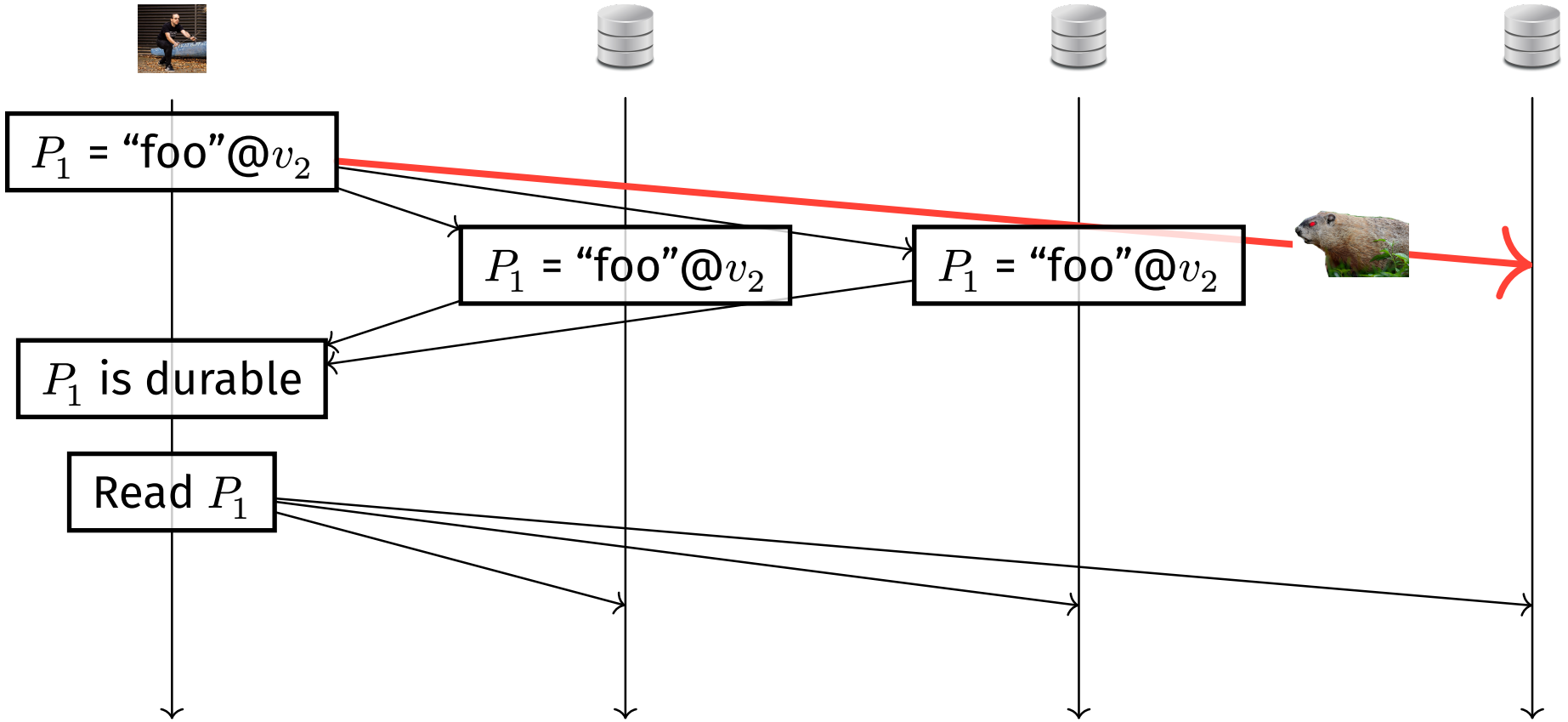
Replication



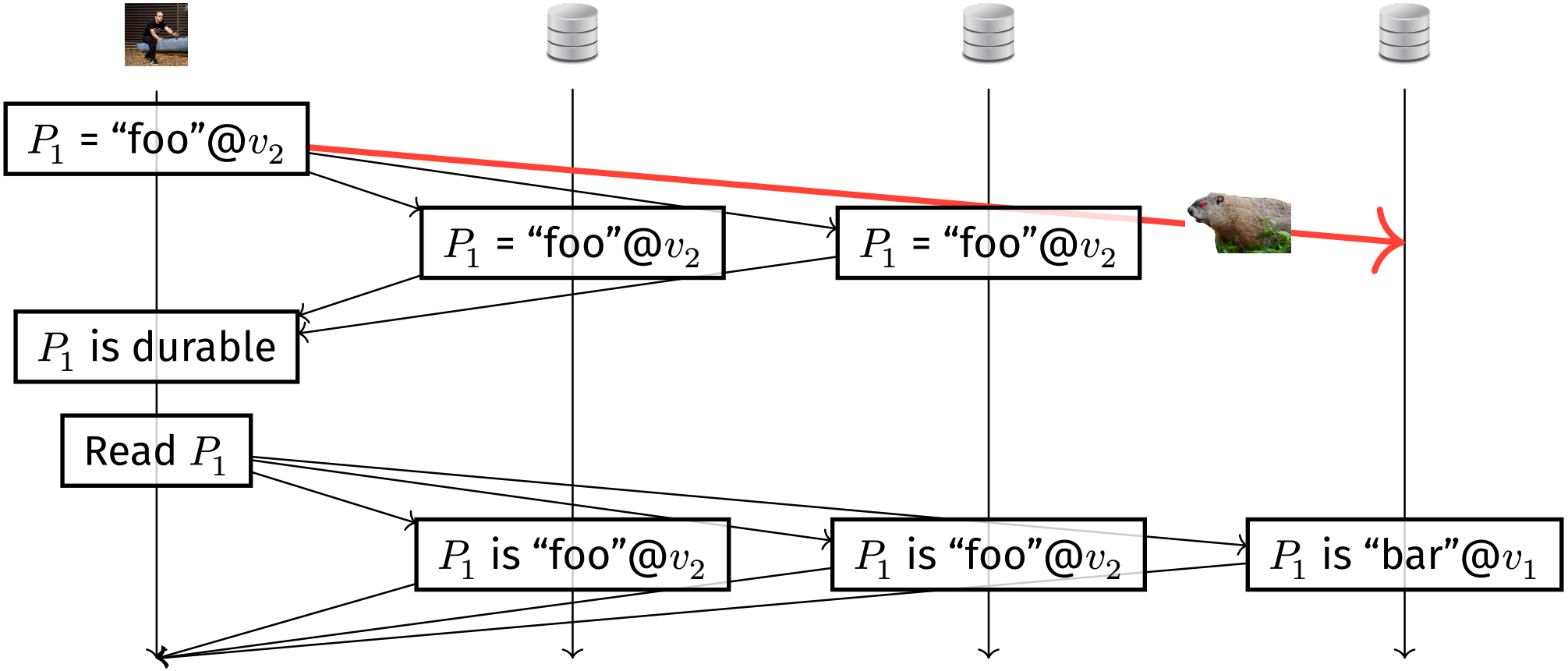
Replication



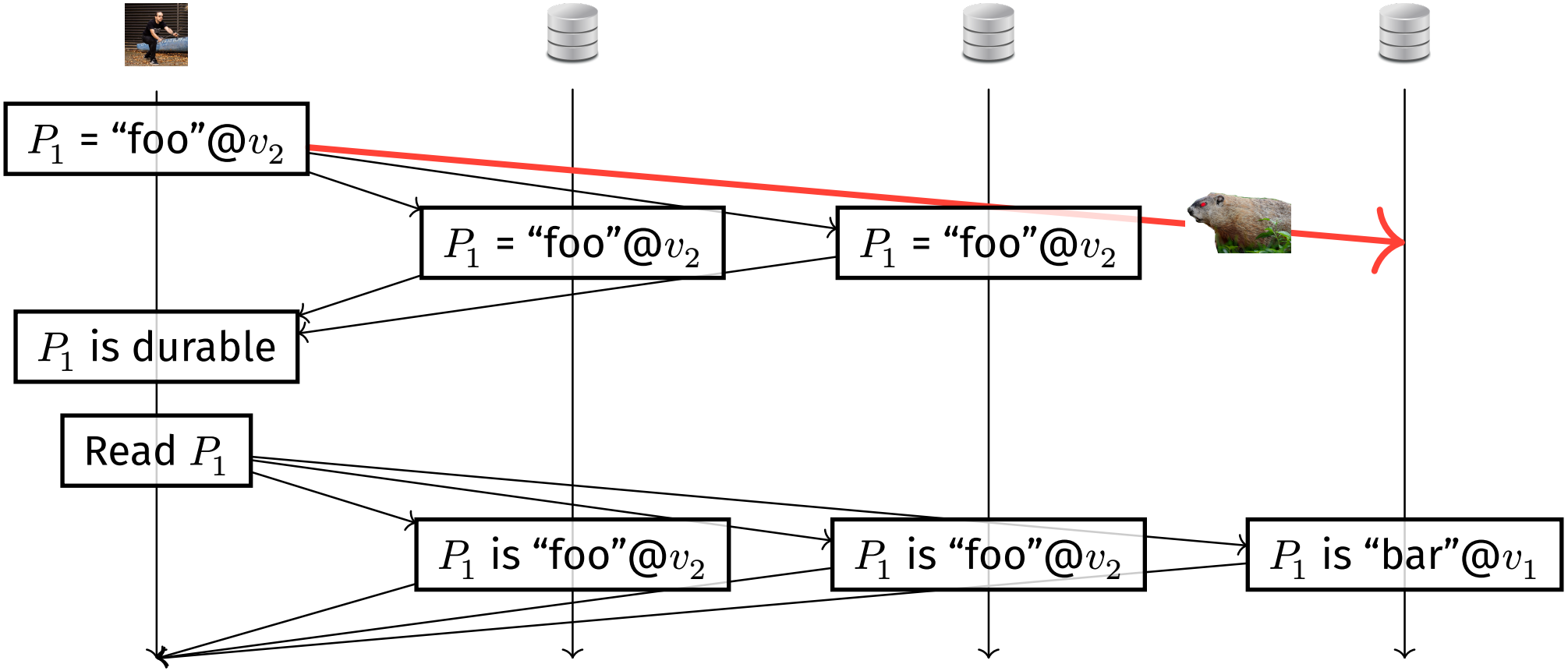
Replication



Replication



Replication



Writer

1. Write data to all replicas
2. Wait for confirmation from W replicas
3. Data is Durable?

Reader

1. Request data from all replicas
2. Receive response from R replicas

How do we guarantee the reader will see at least one copy of every write?

Pigeonhole Principle

$R + W > N$ guarantees that at least one reader and writer are on the same instance

Timeouts

Timeouts

What does a node assume when it fails to receive a message?

Assume transient failure and wait longer

All data remains consistent … but unavailable while we wait, which might be indefinitely.

Assume non-recoverable failure and fail-over to backups

All data remains available … but if it turns out to be transient, there can be inconsistencies.

Consistency **or** **A**vailability **or** **P**artition tolerance

Traditionally: Pick any 2 (‘P’ has to be one)

Consistency **or** **A**vailability **during** **P**artitions

I prefer this phrasing