Pretend Synchrony

Klaus von Gleissenthall

UC San Diego

Software Systems Shouldn't Fail

Ariane 5 Crash

Crashed due to float to int conversion bug

1996

Marriott Breach

Data from 500 Mio customers (2018)

Name, Passport No., Credit Card Numbers

The Nightmare

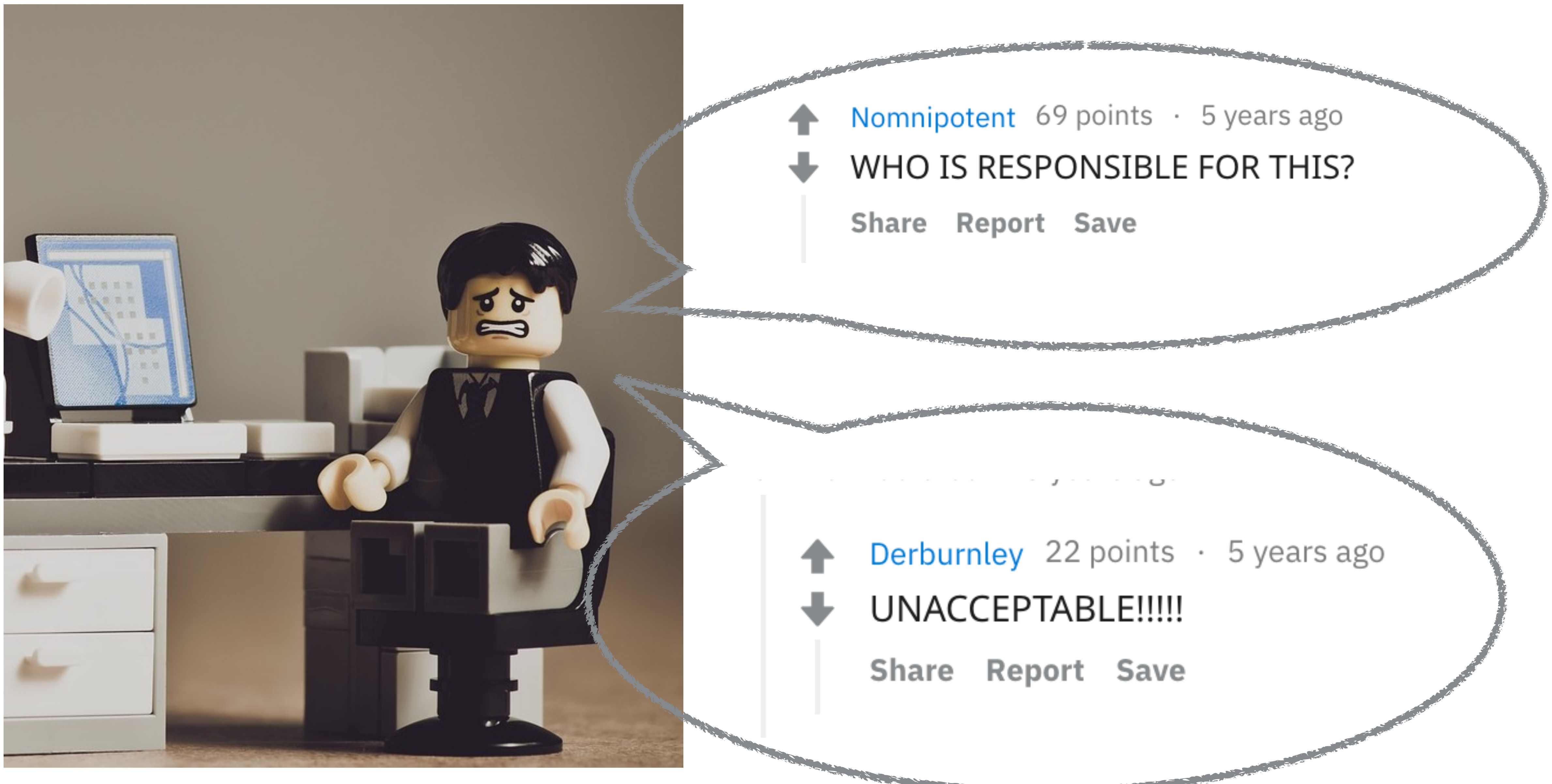
The Nightmare

Katie Warren @KayEllDoubleYoo · 5 Oct 2017

@NetflixUK is down! What do I do now? 😩What kind of life is this? #netflixdown

2

The Nightmare



The Nightmare

elguitarro 49 points · 5 years ago

Hello darkness, my old friend..

Share Report Save

ToxicSandwich 2 points · 4 years ago

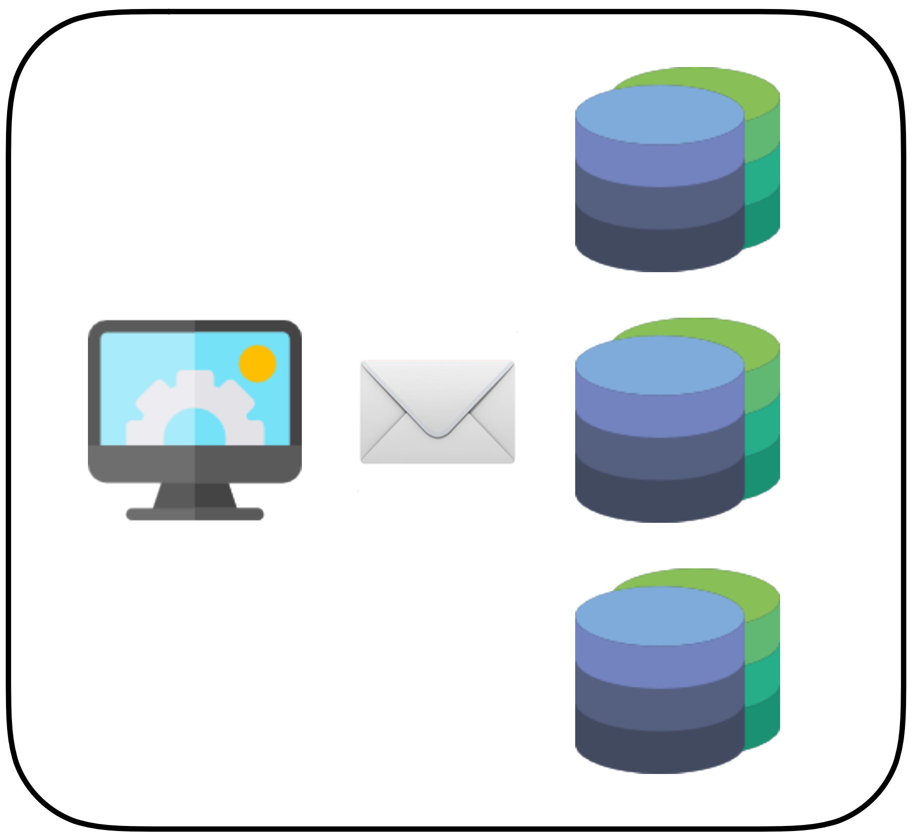
I feel so damn lost.

Share Report Save

Software Systems Shouldn't Fail

How to Make Sure *Distributed* Systems Don't Crash?

Distributed Systems



Nodes run protocol

Send & receive asynchronously

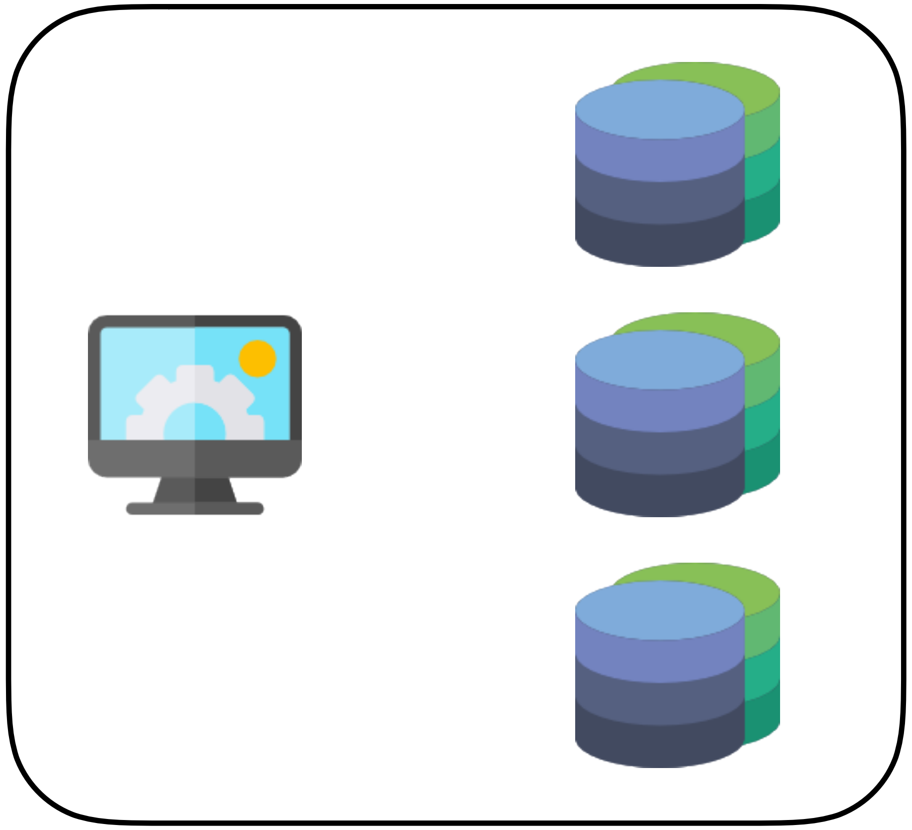
Testing?

Testing?

To fix a run, we need

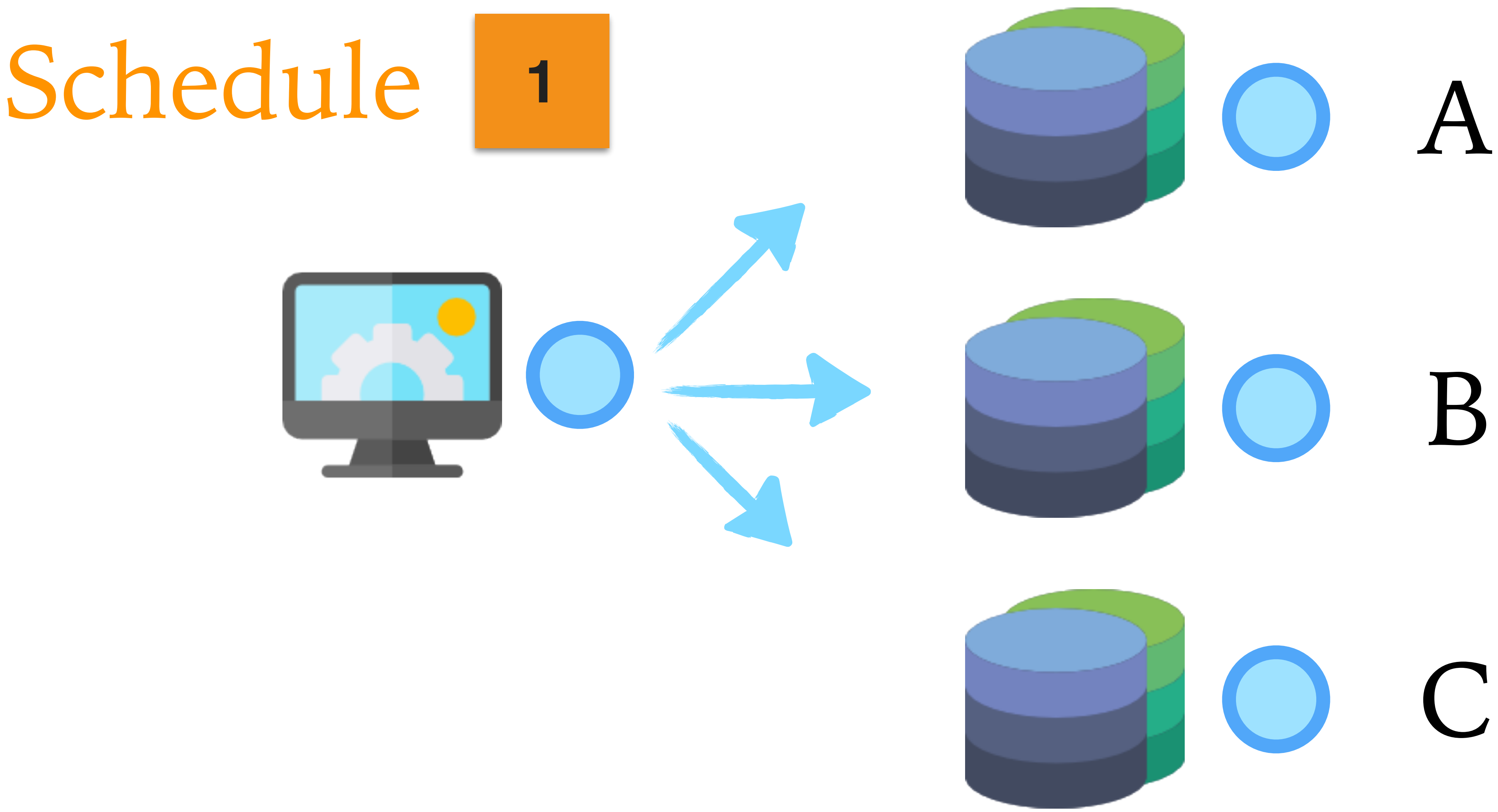
Input

Schedule



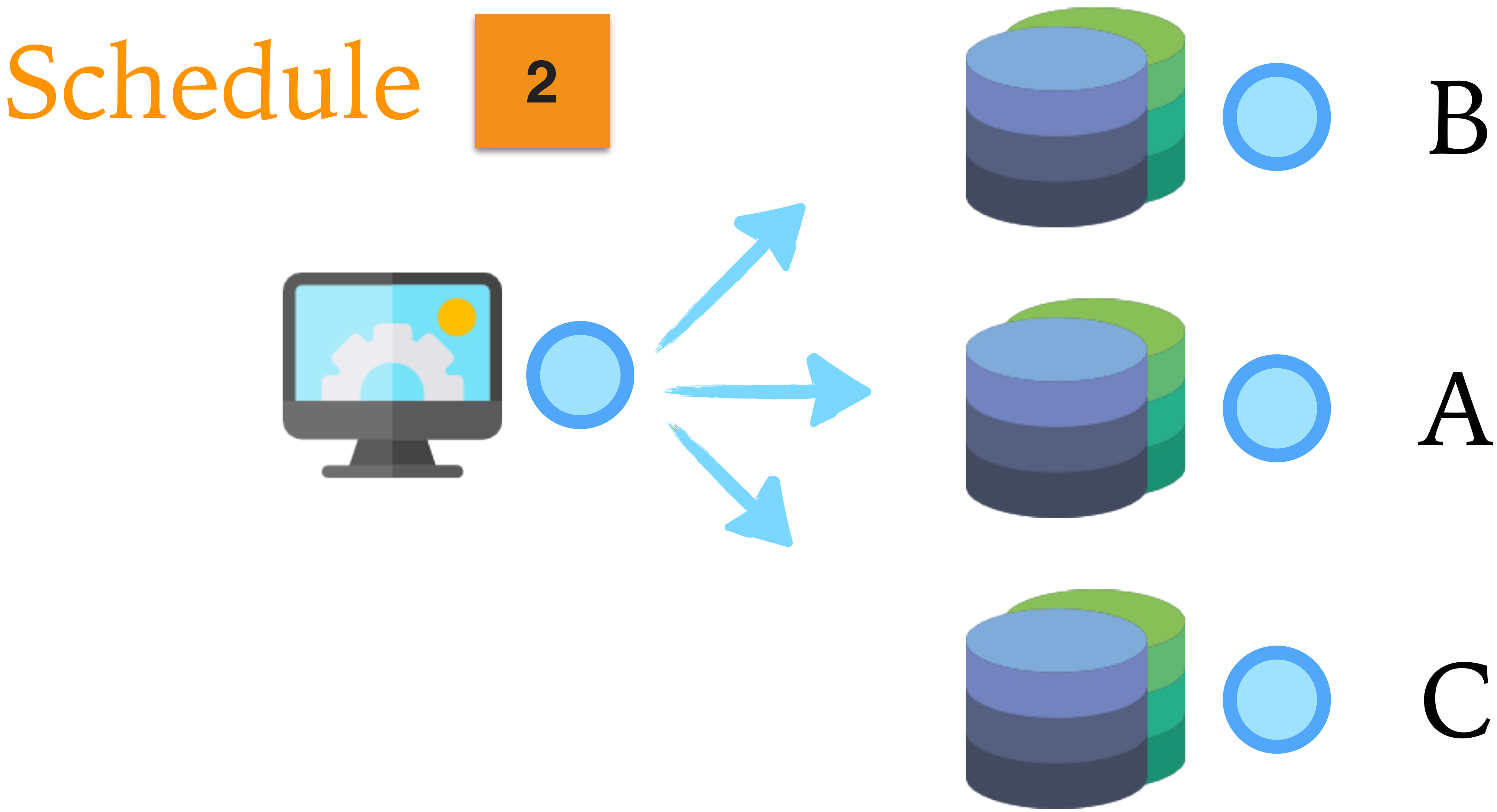
Testing?

Schedule: order on message delivery



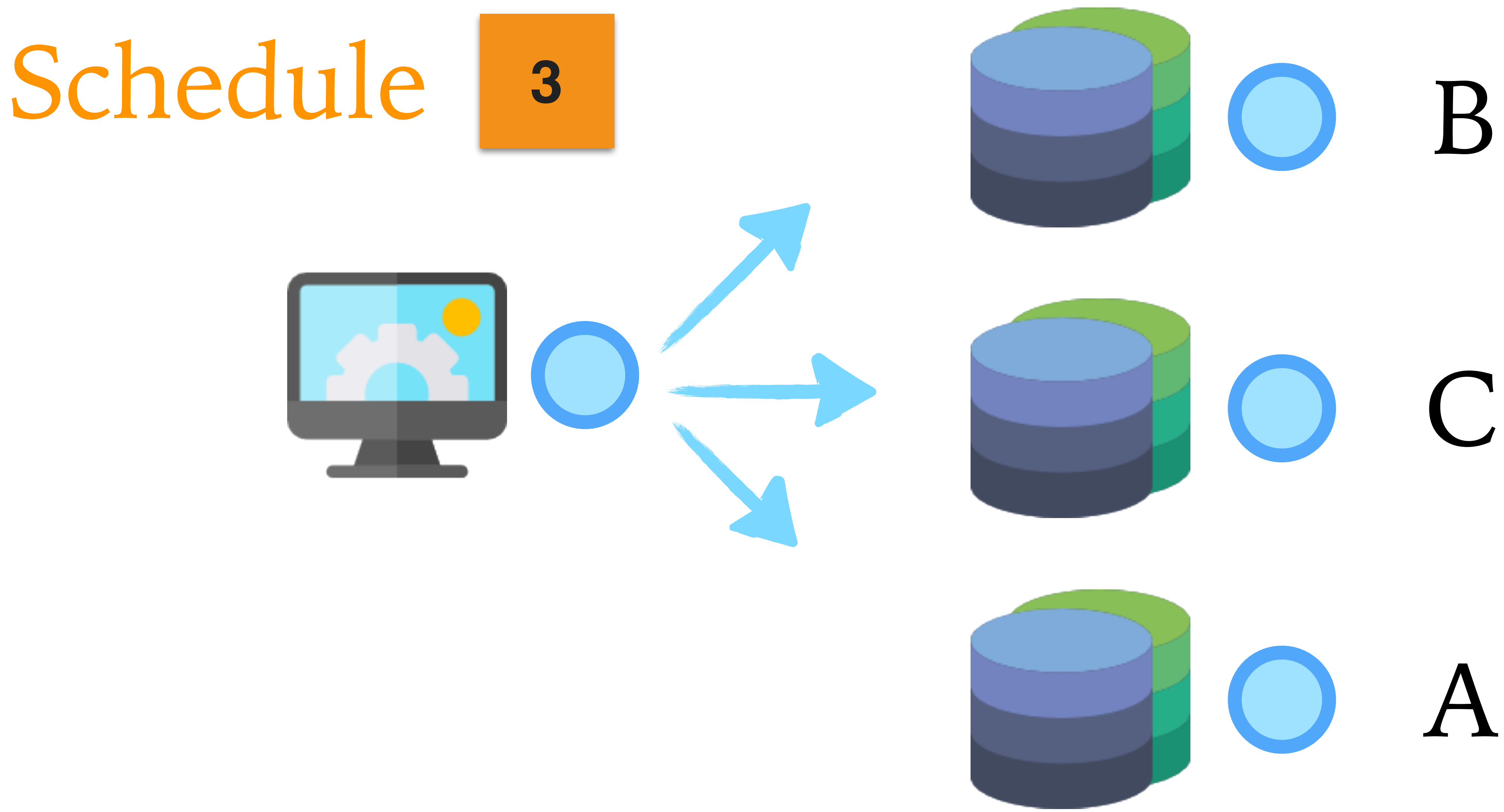
Testing?

Schedule: order on message delivery



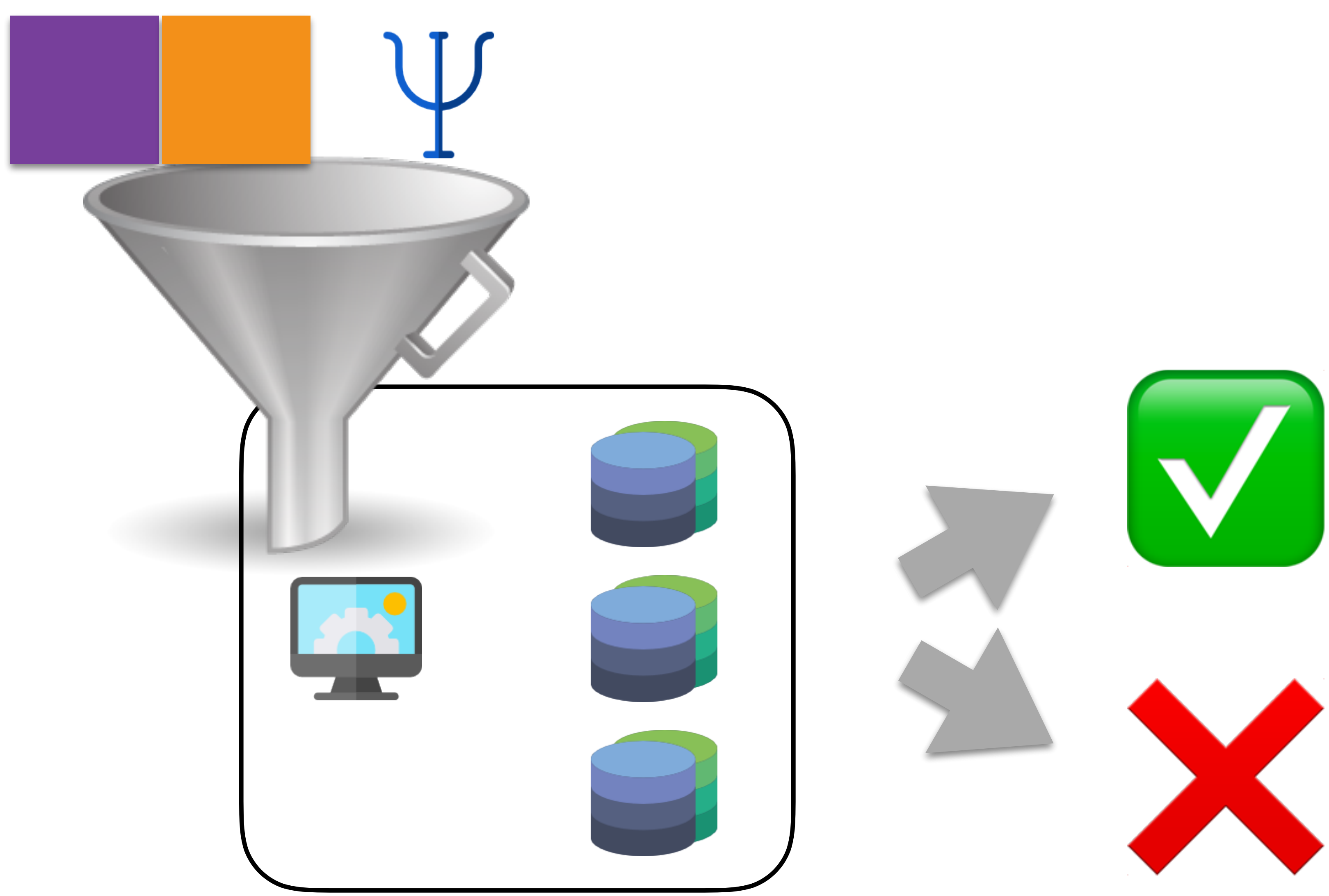
Testing?

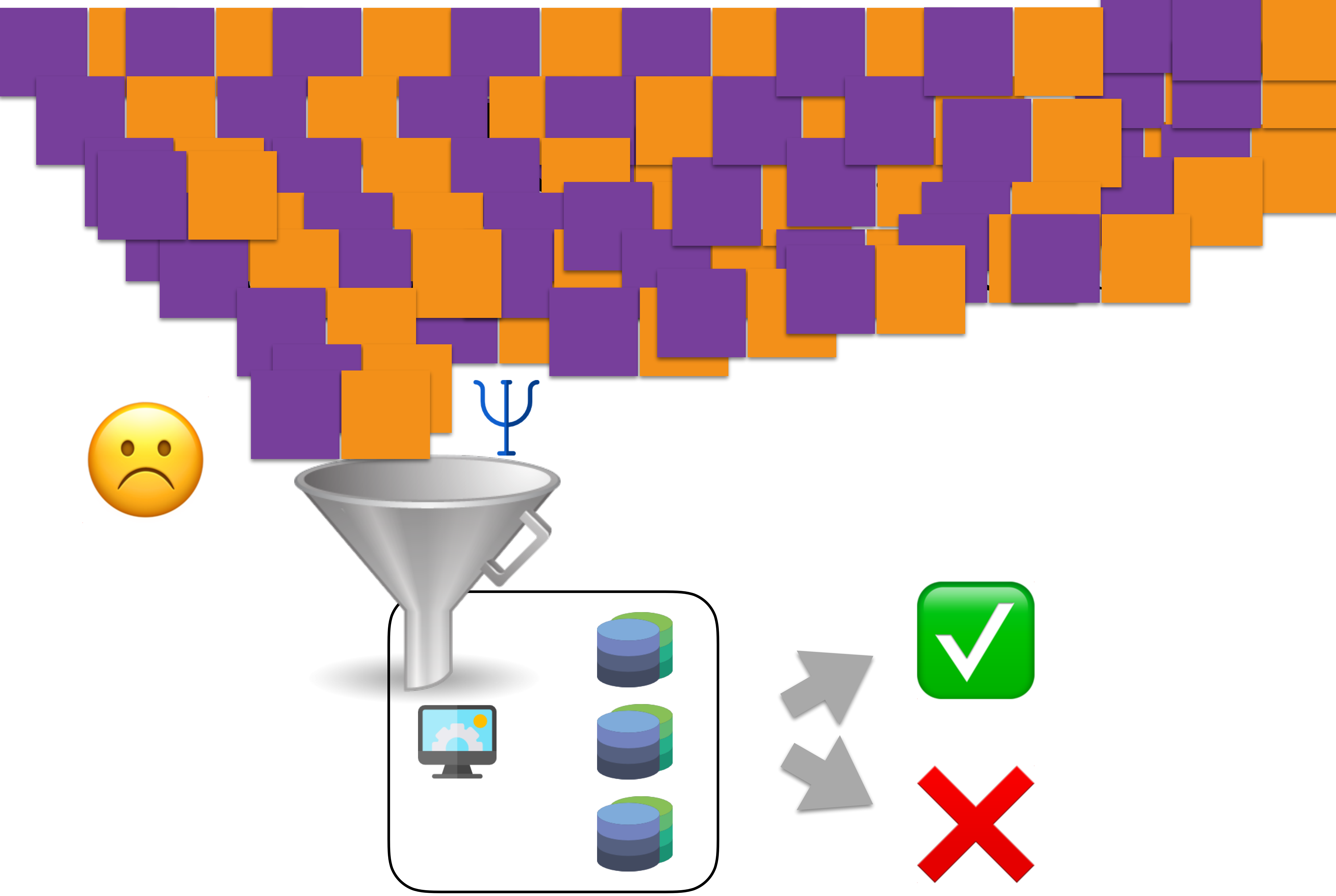
Schedule: order on message delivery



Testing?

Given *input* & *schedule* check property Ψ




Ψ

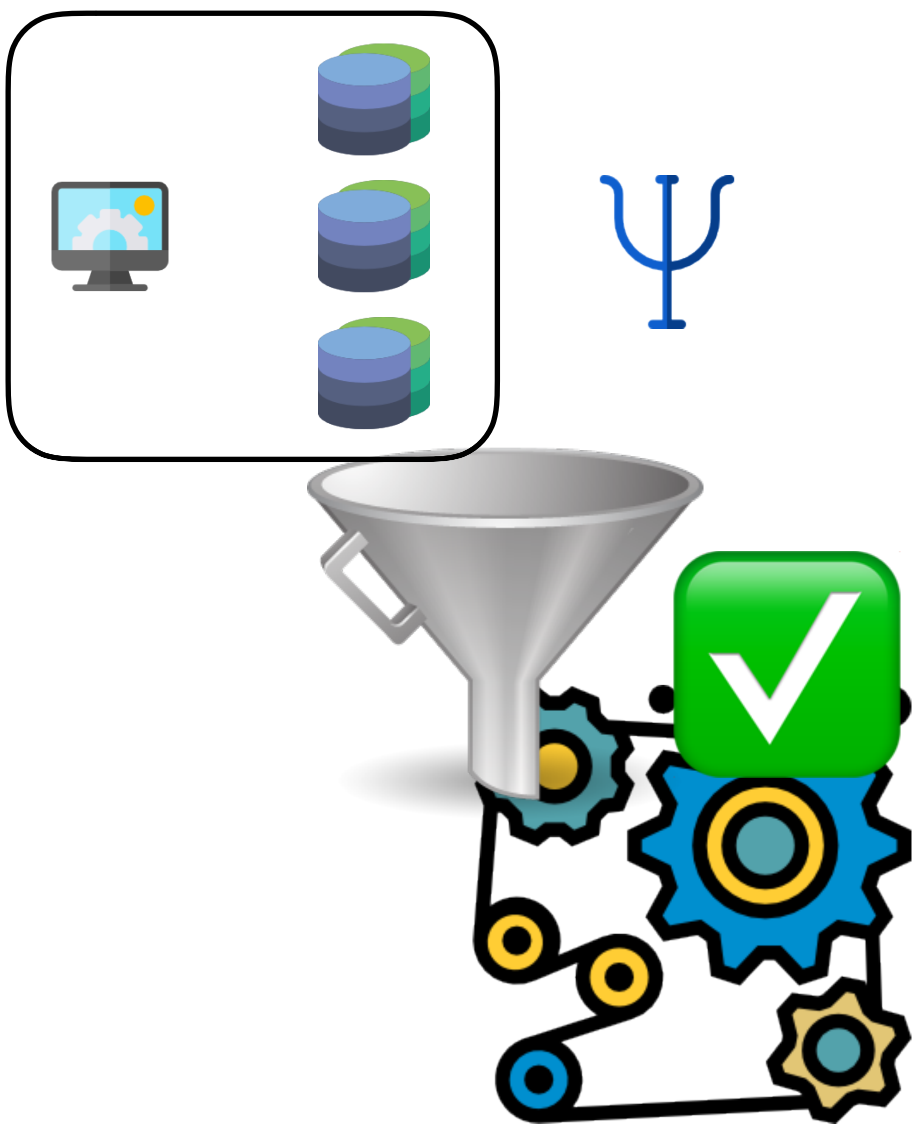
Testing?

Asynchrony: too many schedules

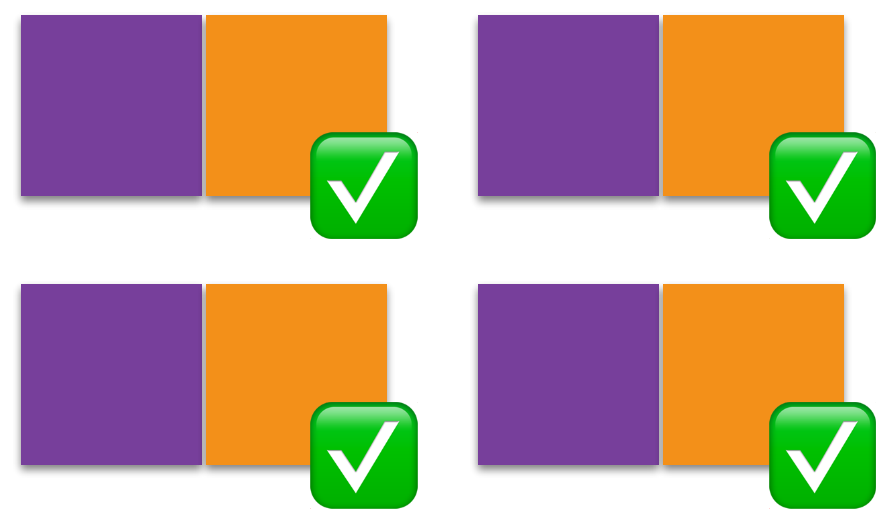
Model Checking?

Model Checking?

Enumerate all *inputs* & *schedules*



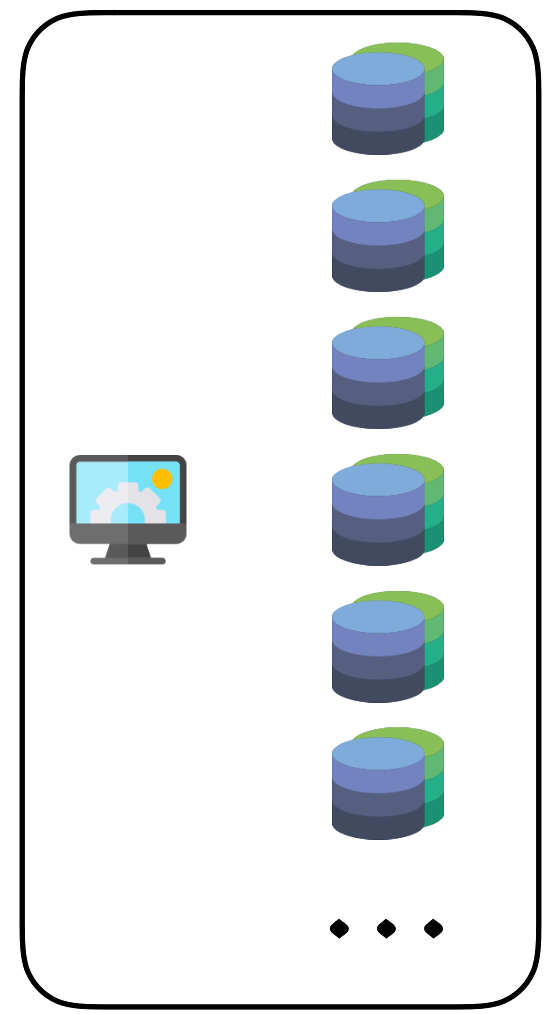
MC



Model Checking?

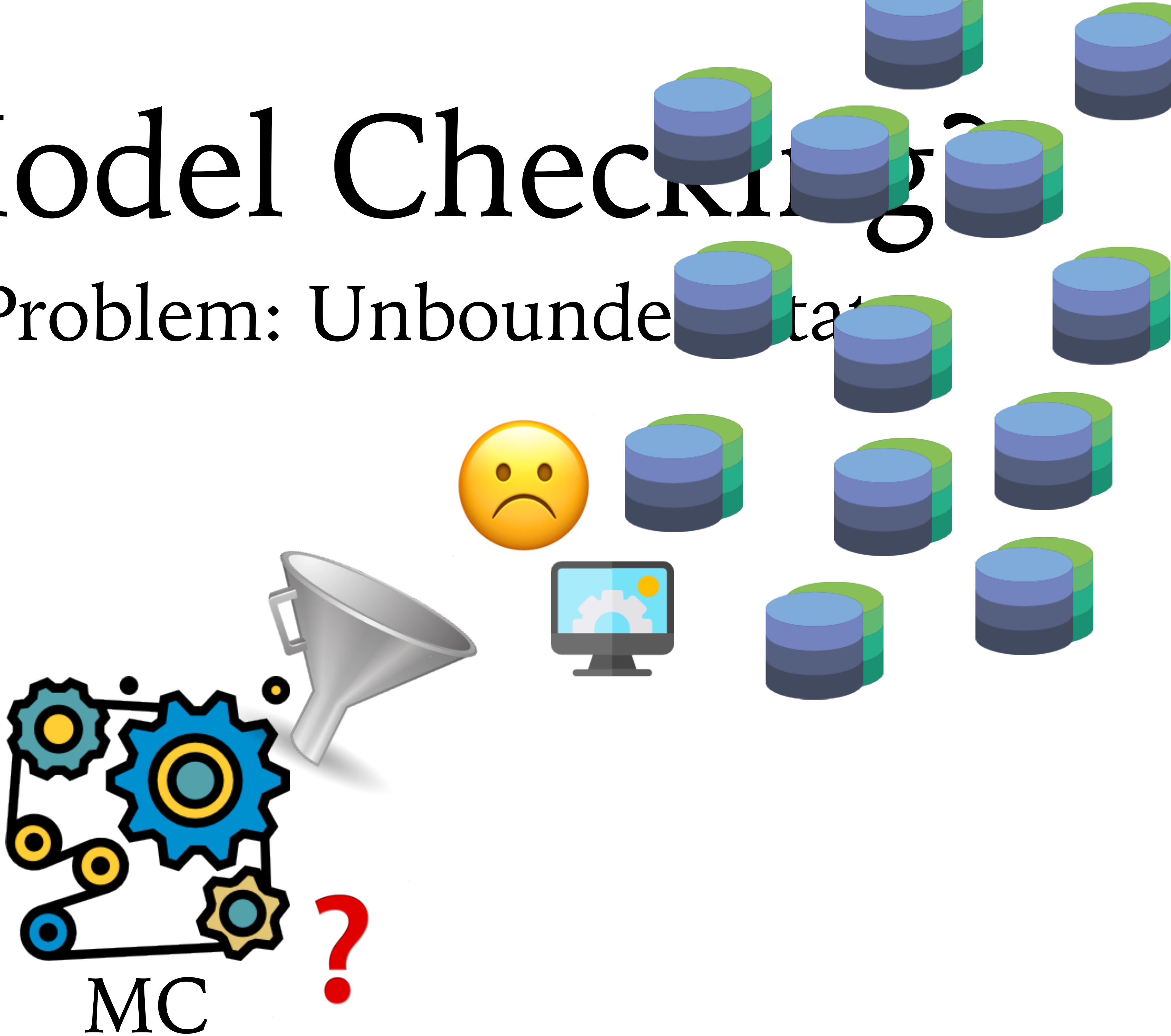
Problem: Unbounded State


MC



Model Checking?

Problem: Unbounde tat



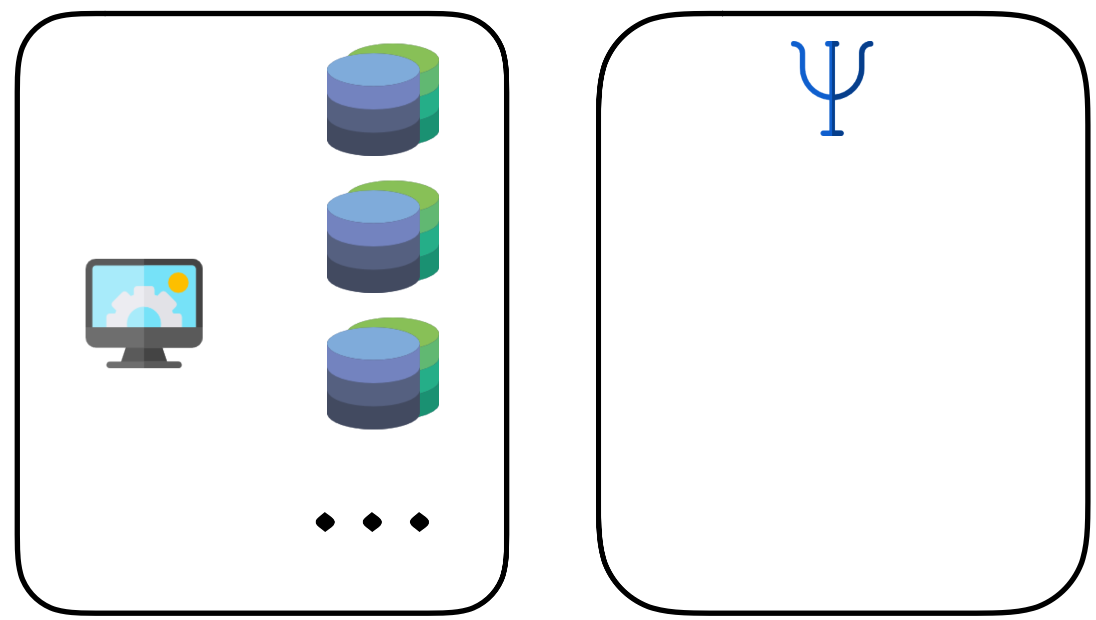
Model Checking?

Problem: Unbounded State

Deductive Verification?

Deductive Verification?

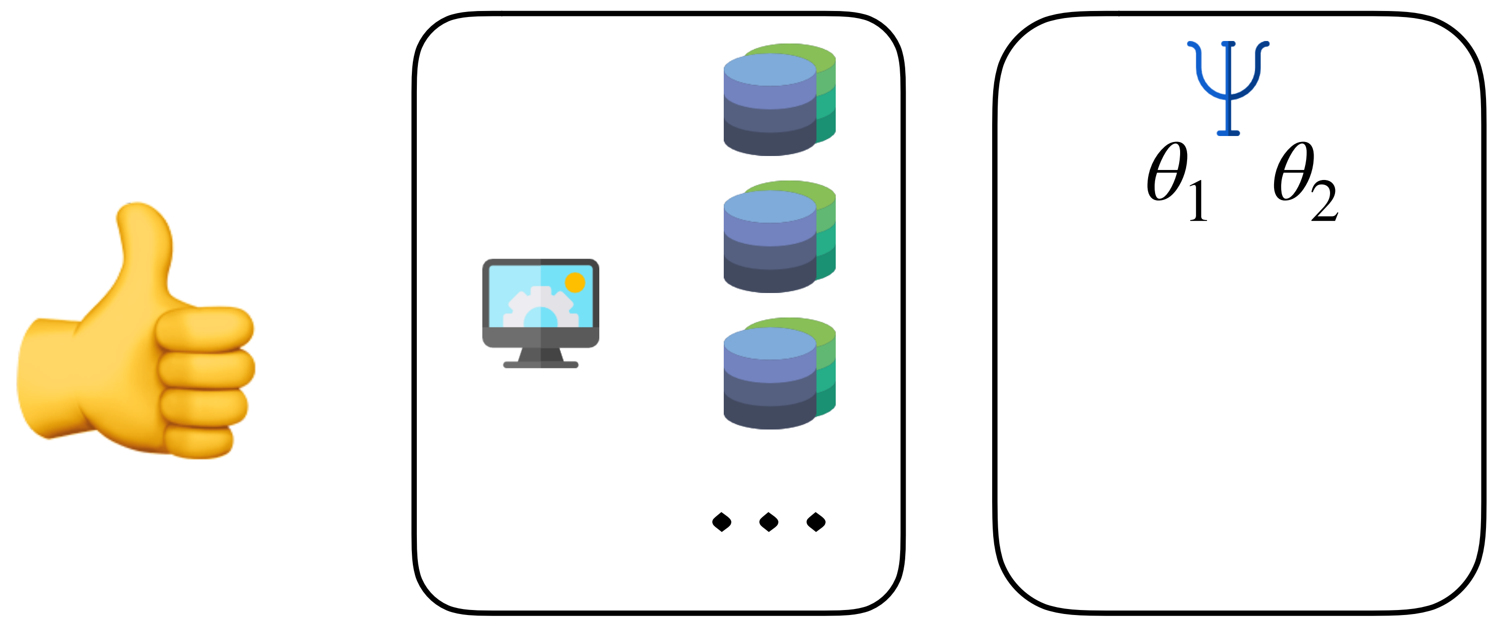
Prove Protocol Correctness



SMT ✅ ❌

Deductive Verification?

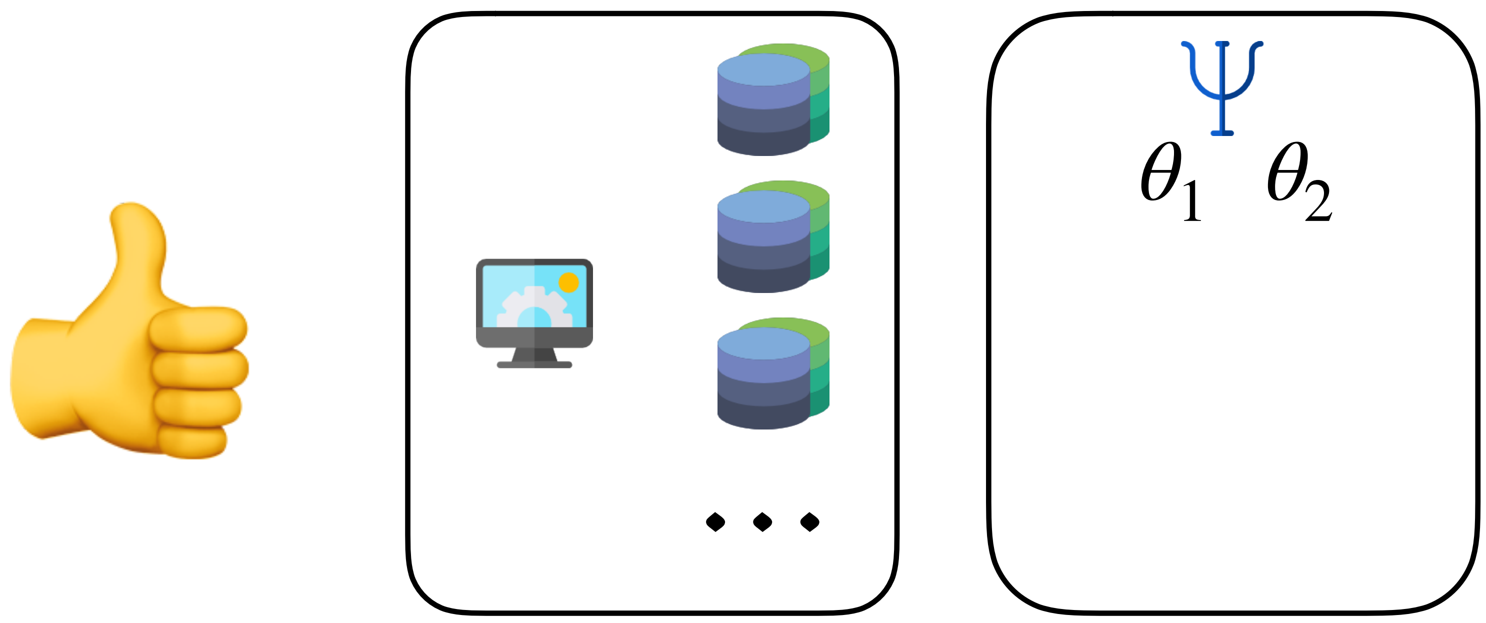
Can handle Unbounded State



… but needs Auxiliary Invariants

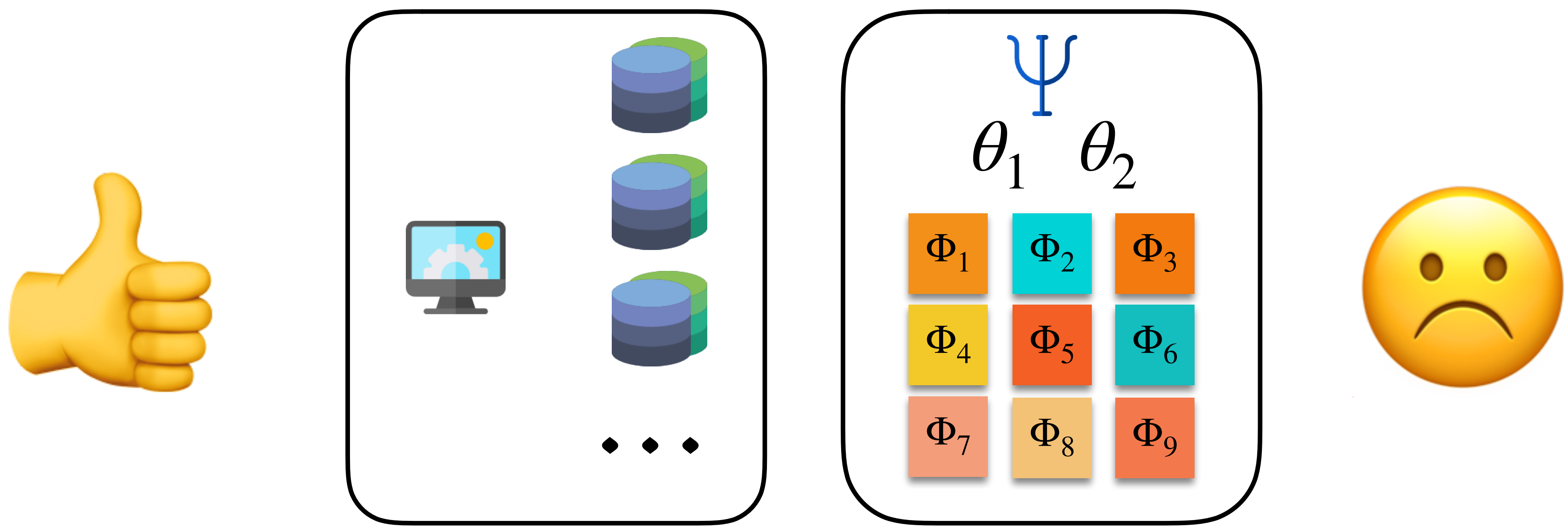
Deductive Verification?

… but needs Auxiliary Invariants



Deductive Verification?

… but needs Auxiliary Invariants



that enumerate

schedules and network state

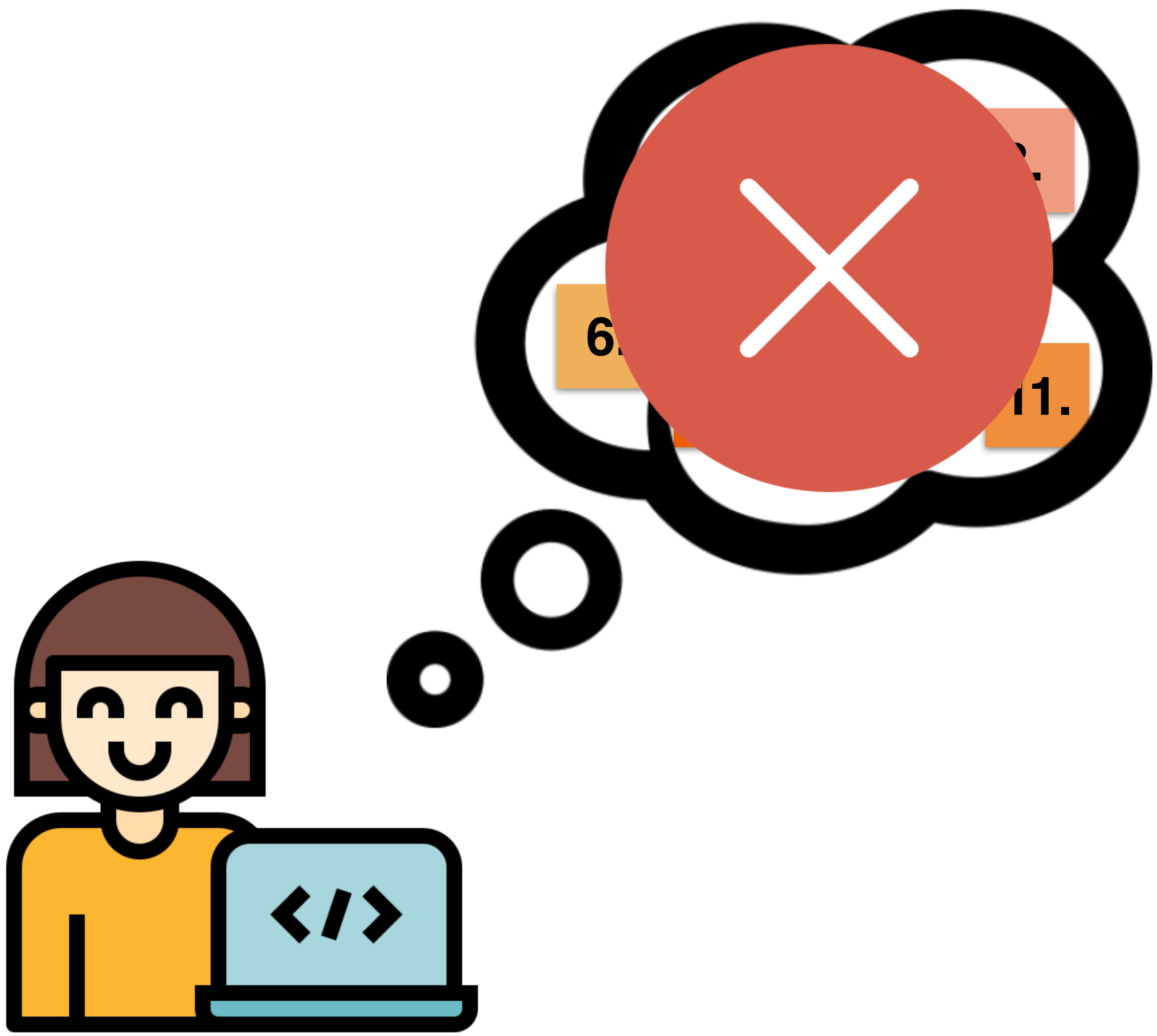
Deductive Verification?

Too many Invariants!

Pretend Synchrony: Make Proofs Easier!

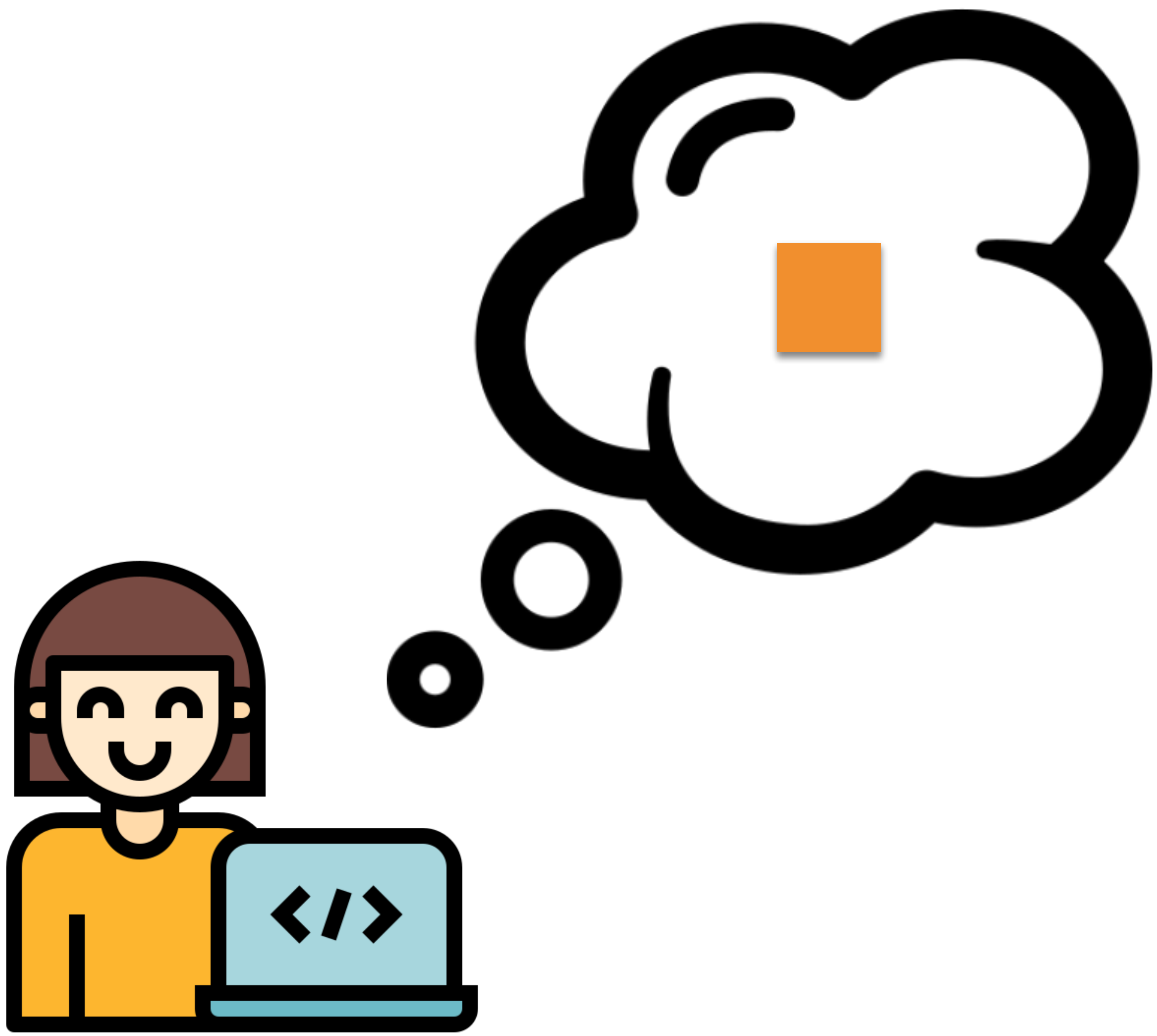
Pretend Synchrony

Programmers *don't case-split* on *schedules* & *network*



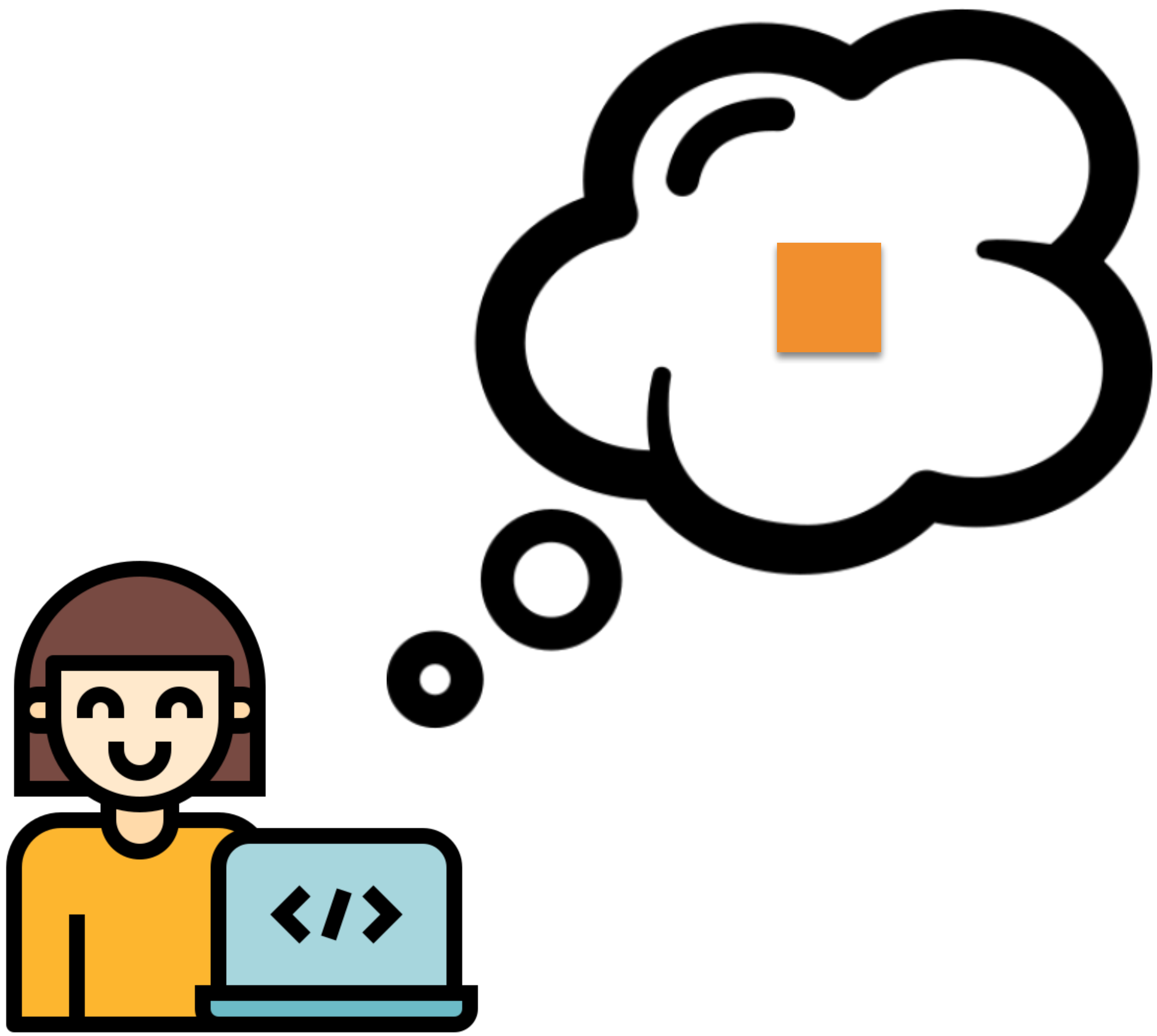
Pretend Synchrony

… they think about a *representative* schedule



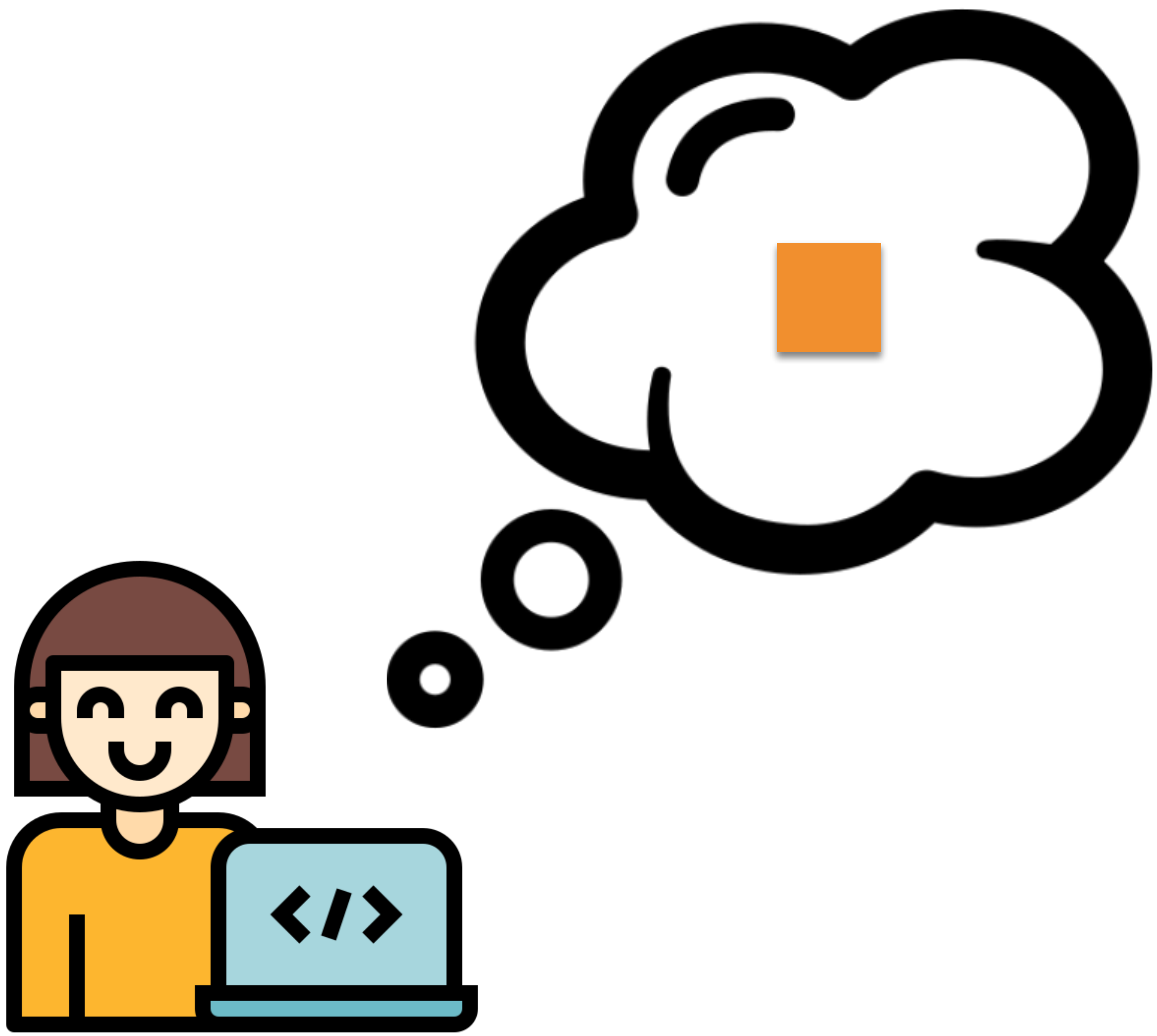
Pretend Synchrony

… where messages are delivered *instantaneously*



Pretend Synchrony

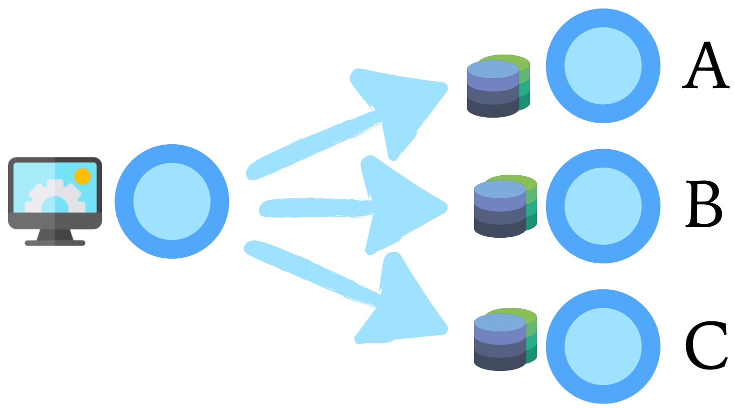
we call this schedule *synchronization*



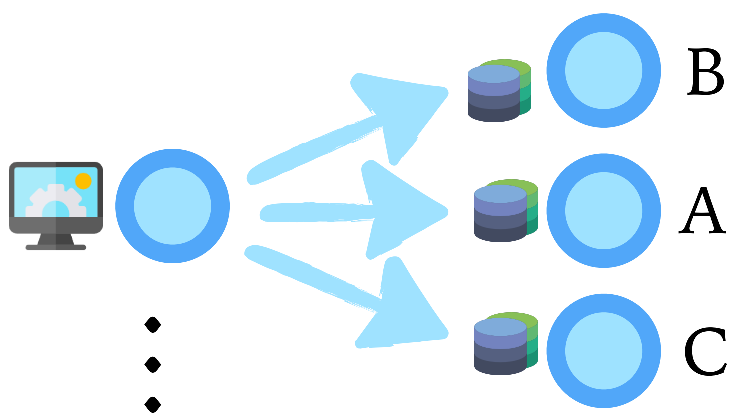
Pretend Synchrony

To verify a protocol

1.

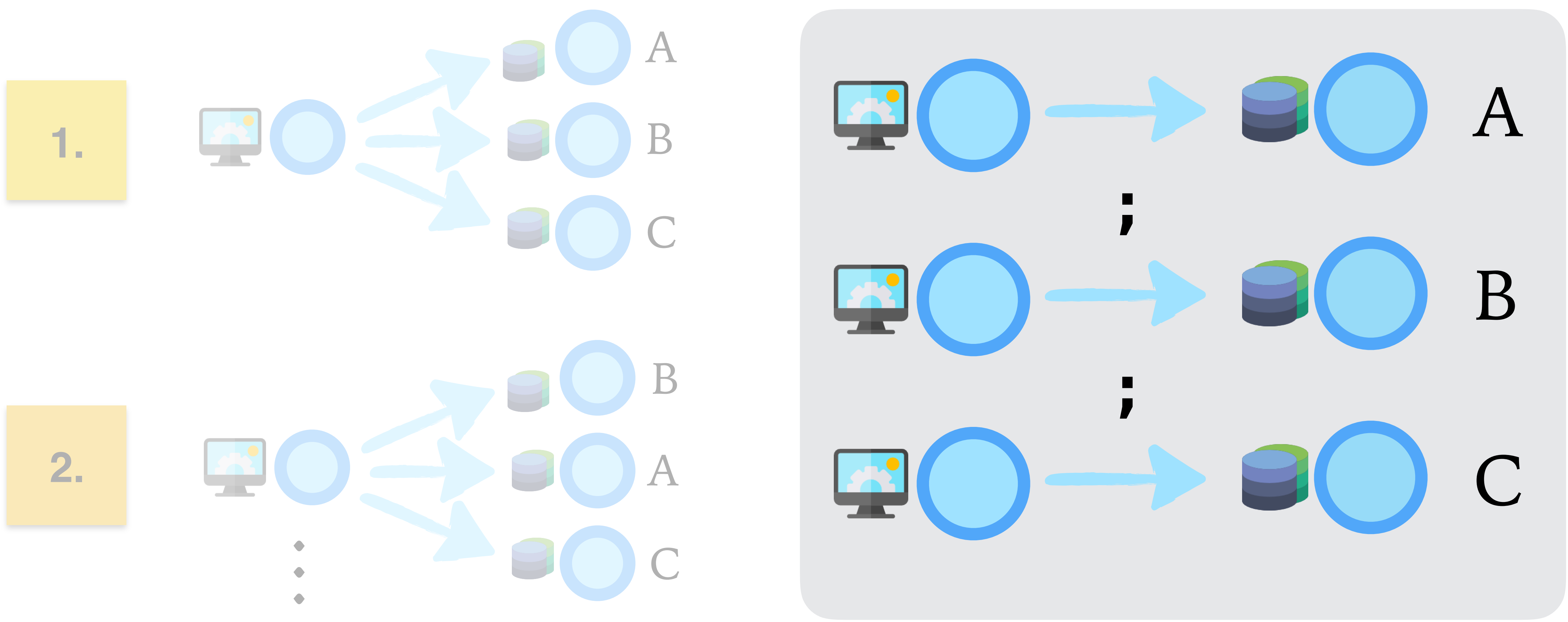


2.



Pretend Synchrony

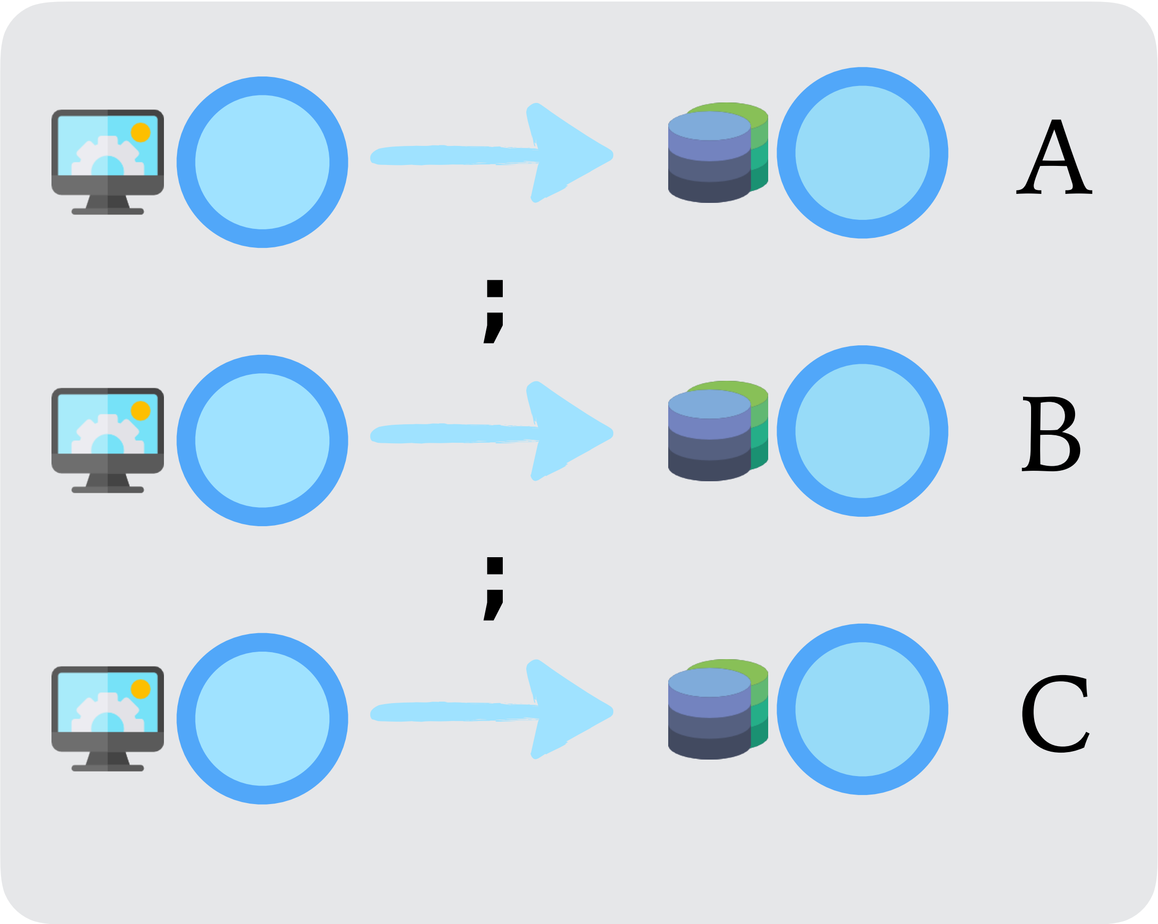
To verify a protocol



… 1. compute its synchronization

Pretend Synchrony

To verify a protocol

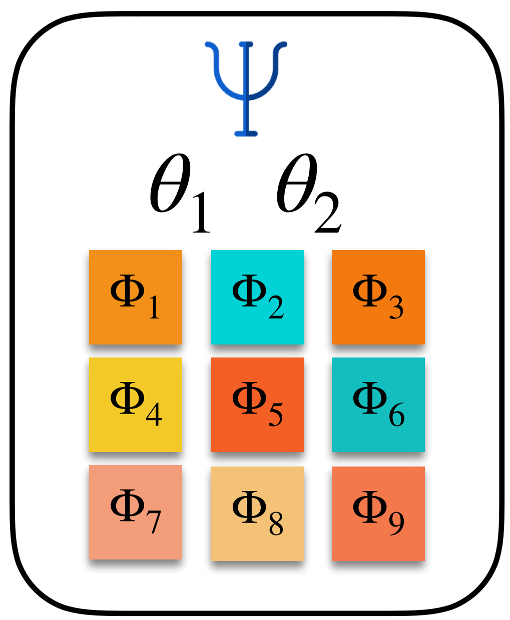


… 2. verify synchronization

Pretend Synchrony

Synchronizations *don't case-split* on

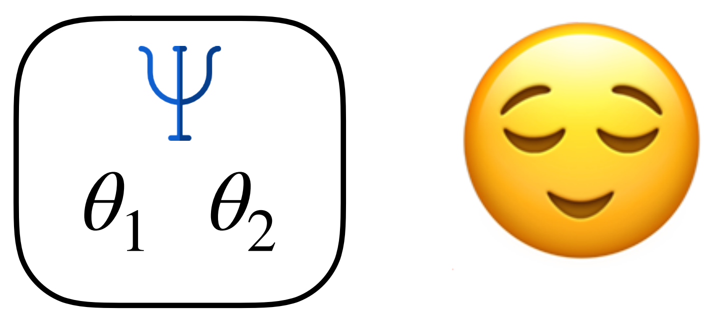
schedules & *network*



Pretend Synchrony

Synchronizations *don't case-split* on

schedules & *network*



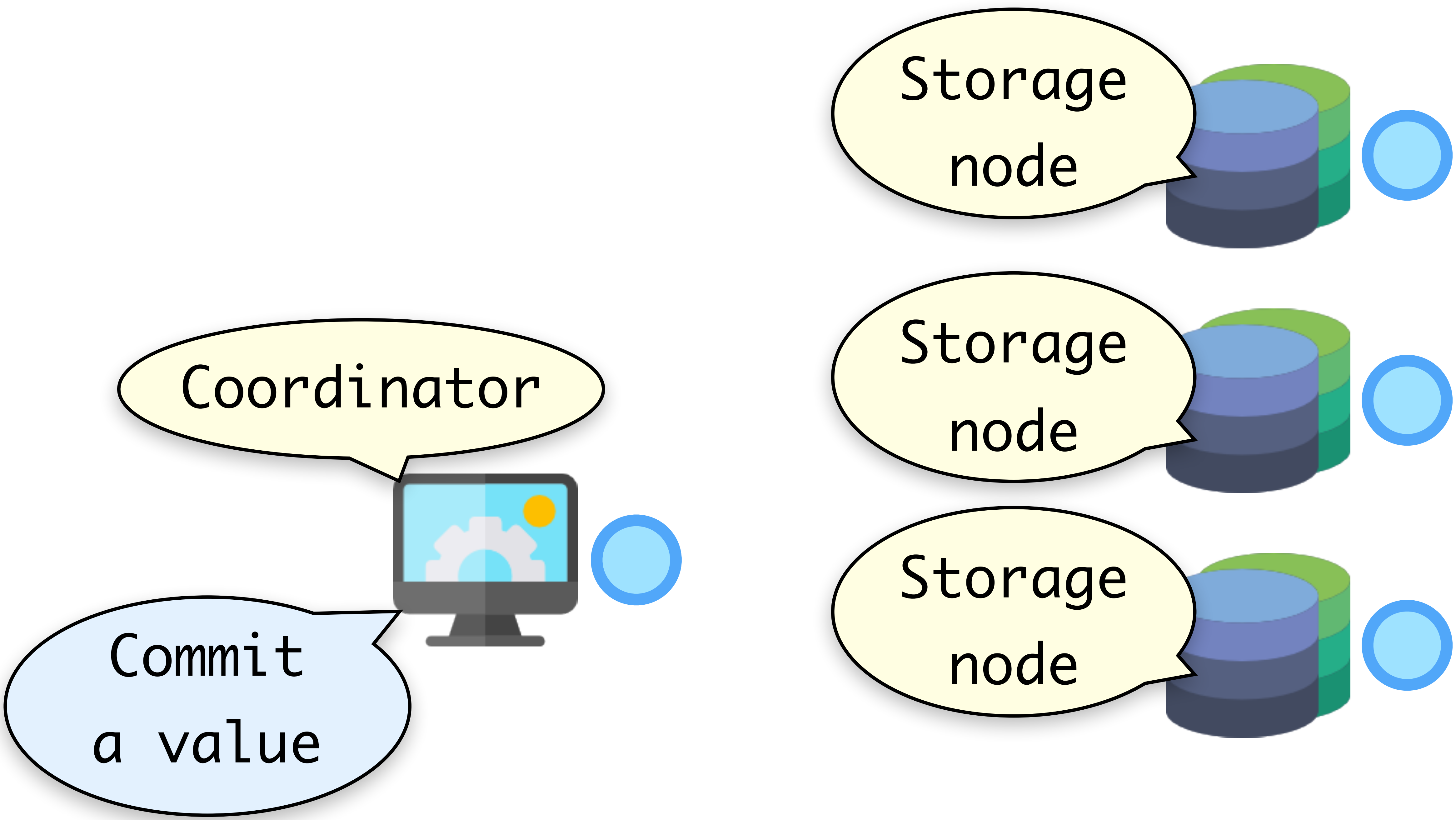
… which significantly *reduces* invariants

Synchronizations

Example 1: Two-Phase Commit

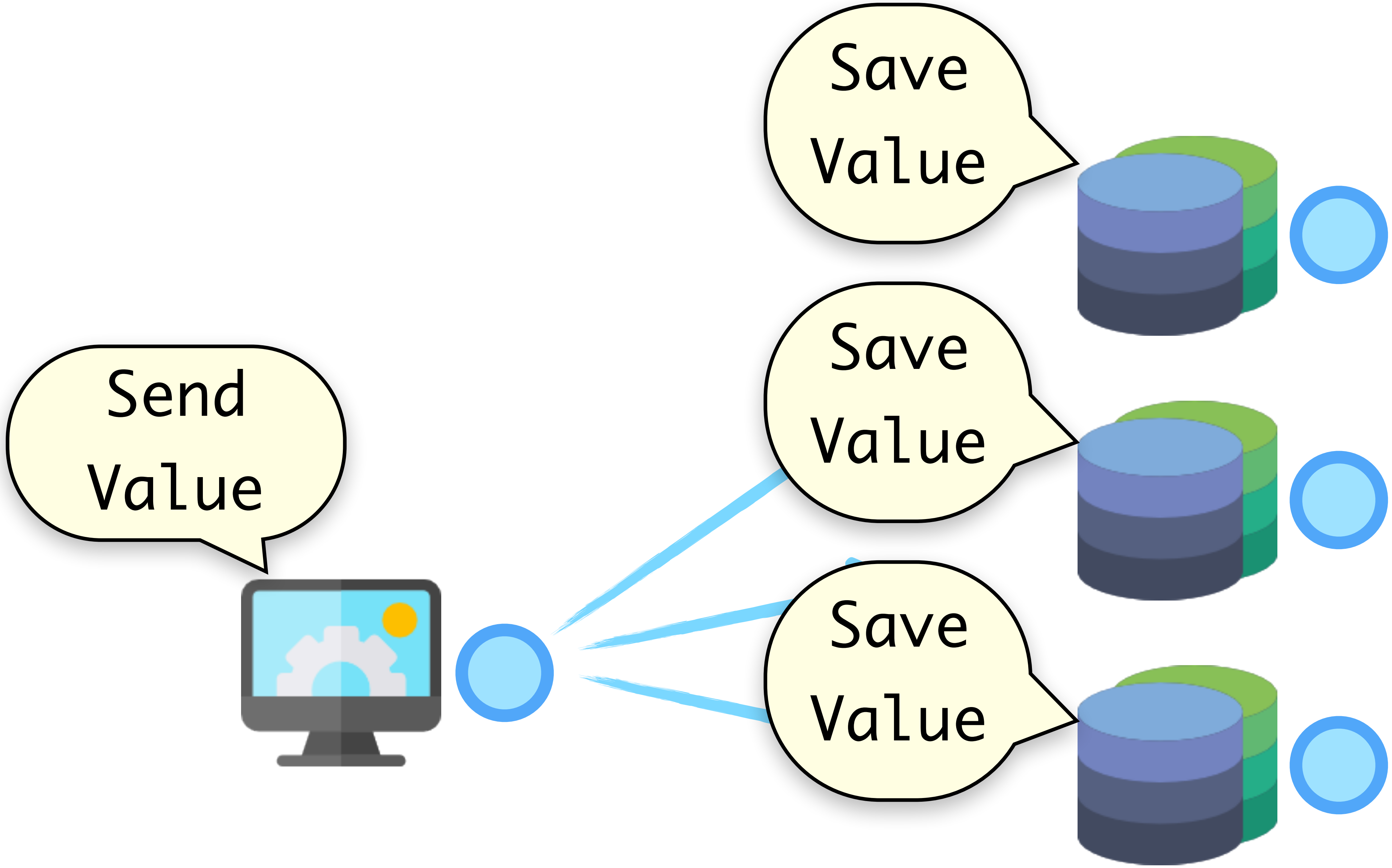
Two-Phase Commit

Phase 1



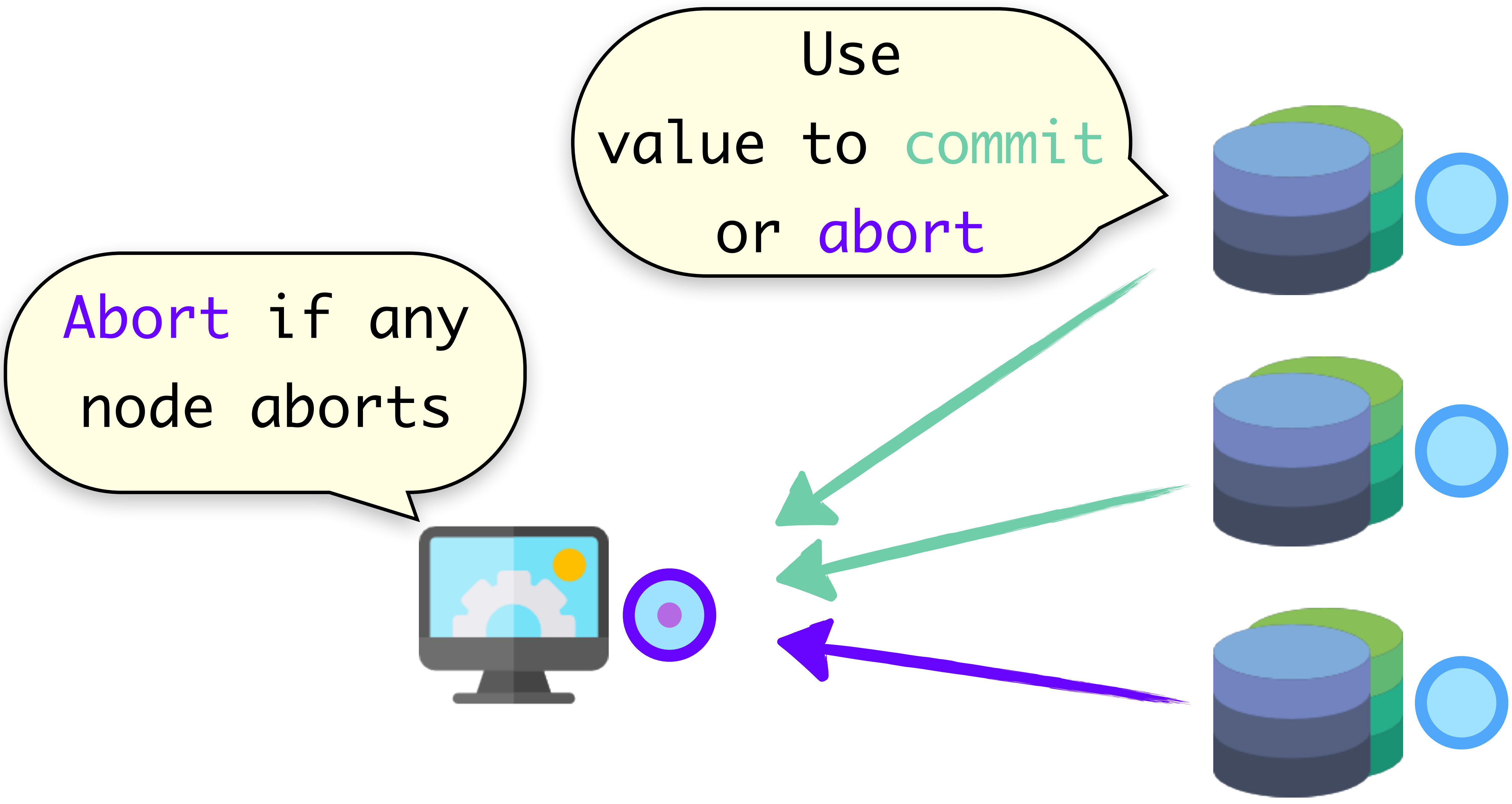
Two-Phase Commit

Phase 1



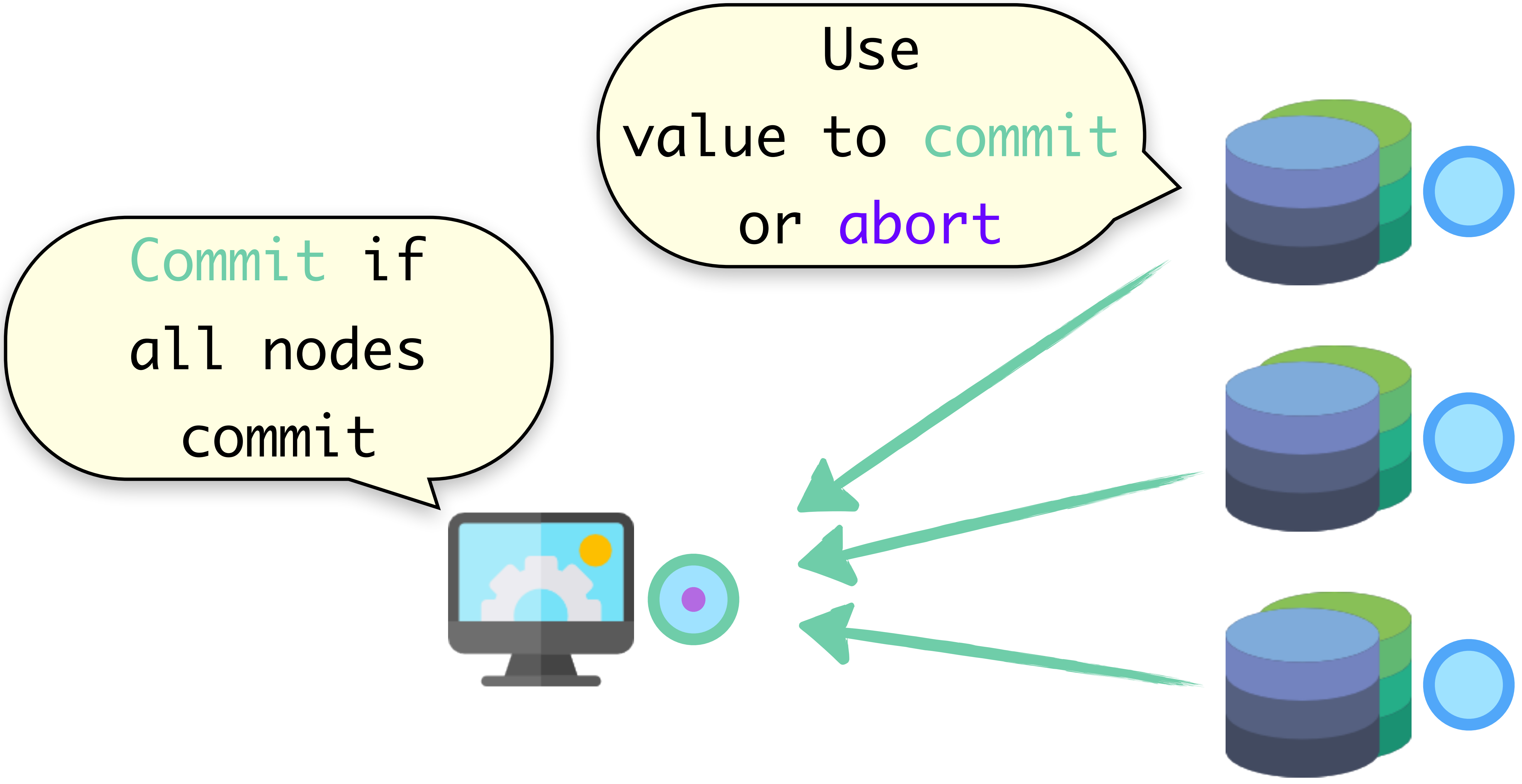
Two-Phase Commit

Phase 1



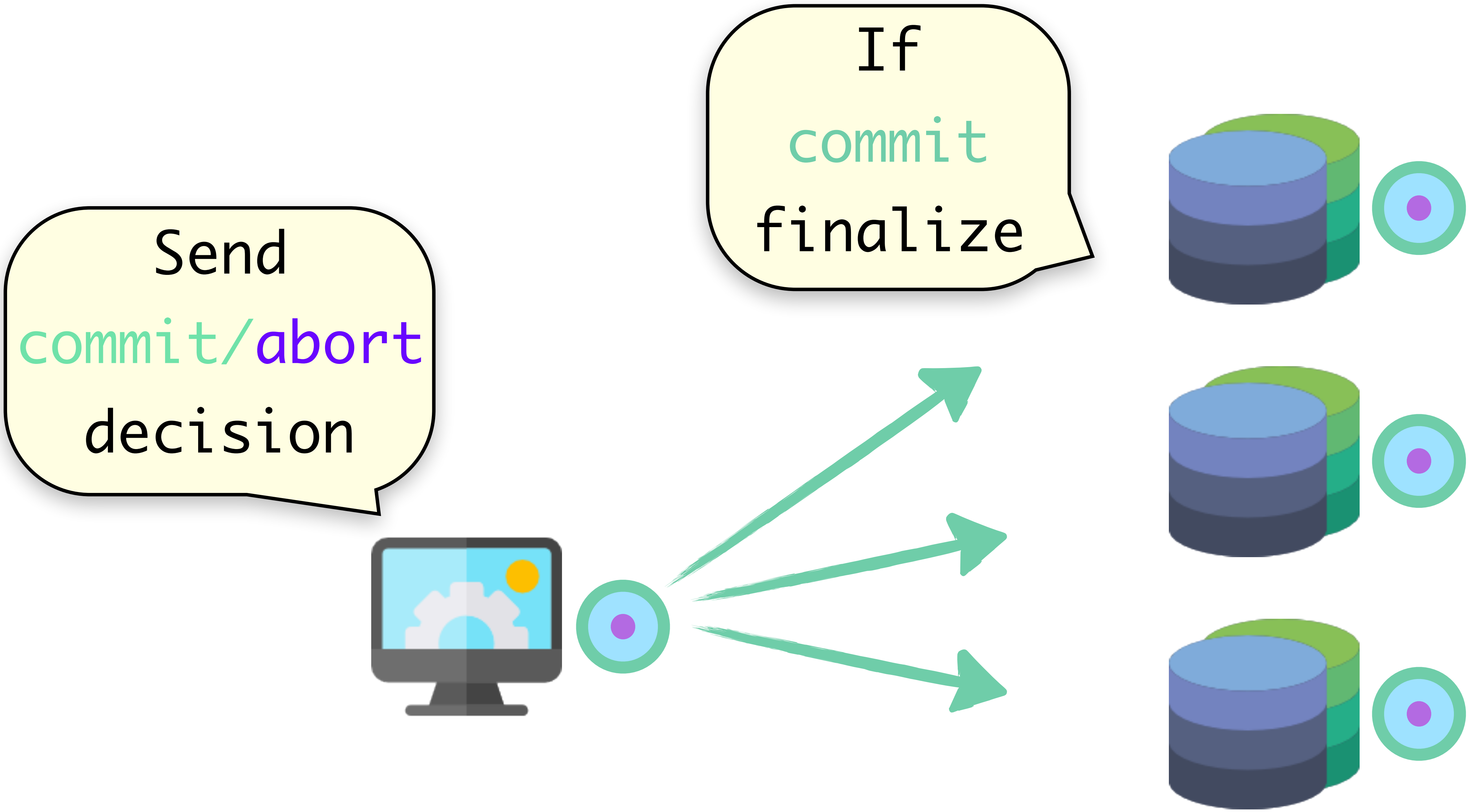
Two-Phase Commit

Phase 1



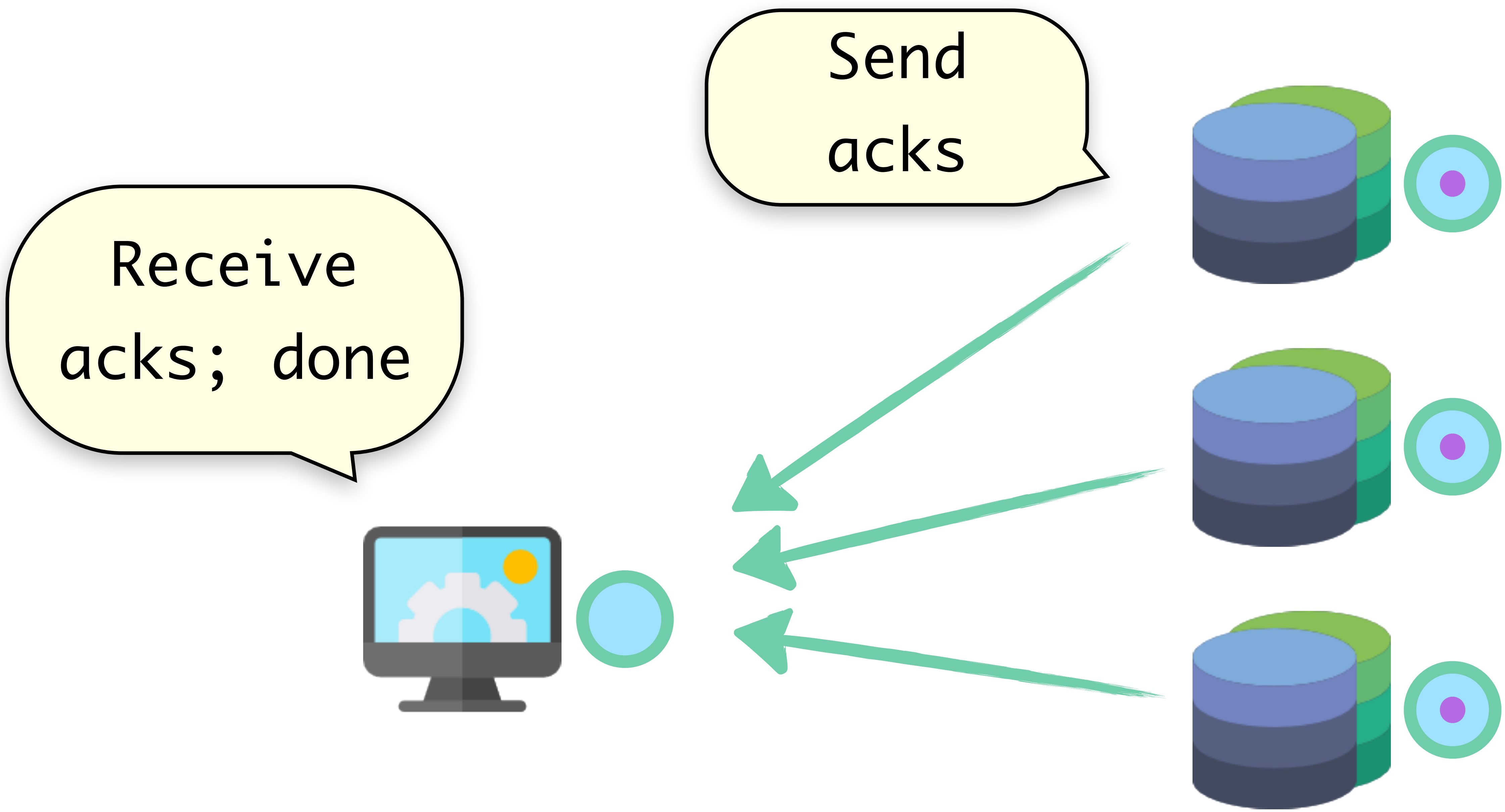
Two-Phase Commit

Phase 2



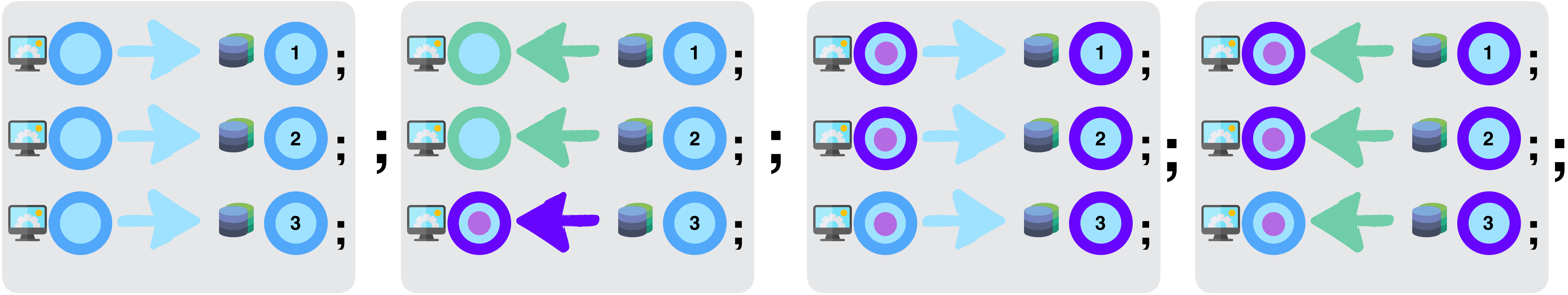
Two-Phase Commit

Phase 2



Two-Phase Commit

Synchronization



1.
Send value
to nodes

2.
Respond
commit/abort

3.
Relay decision

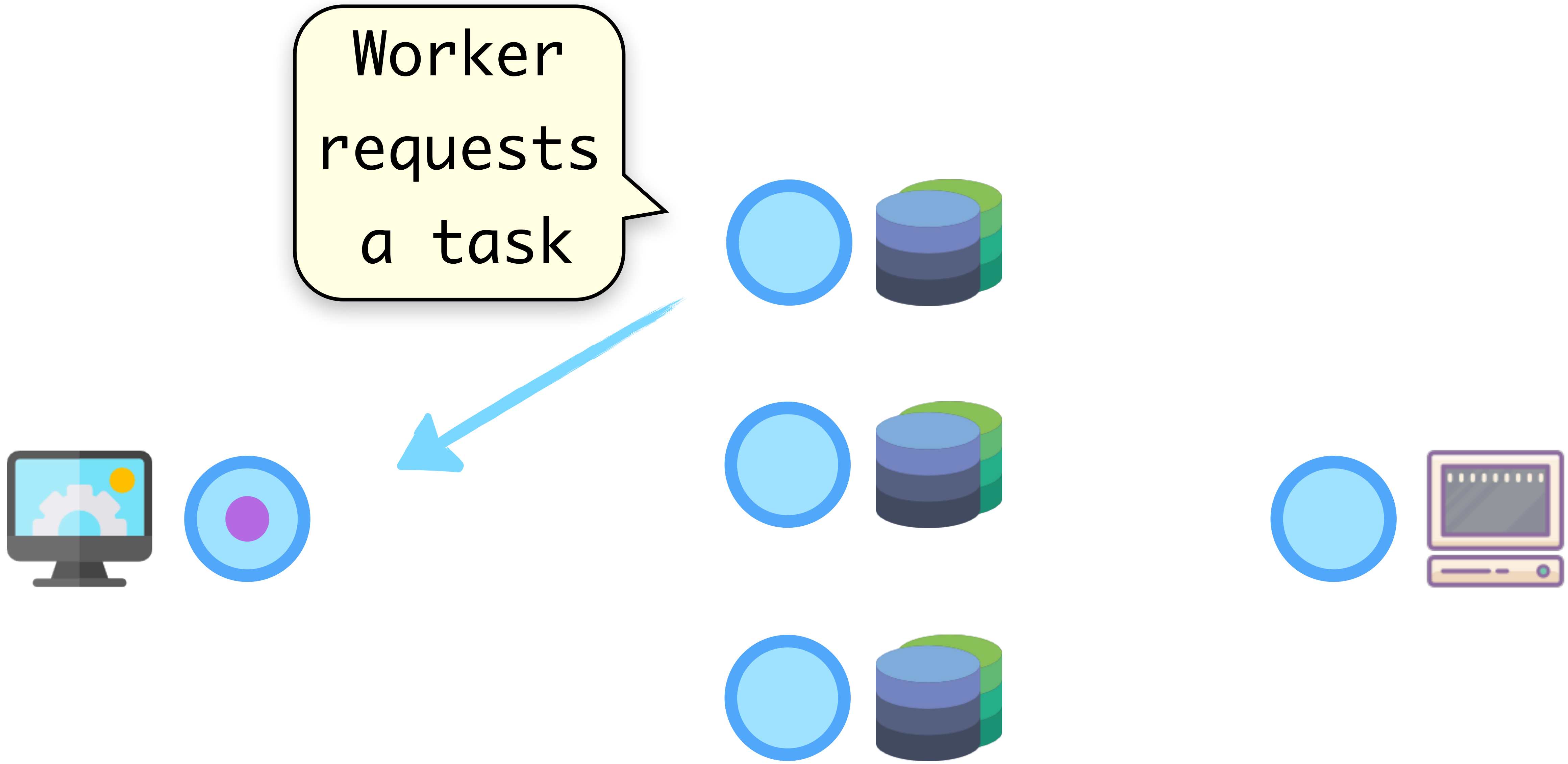
4.
Gather acks

Synchronizations

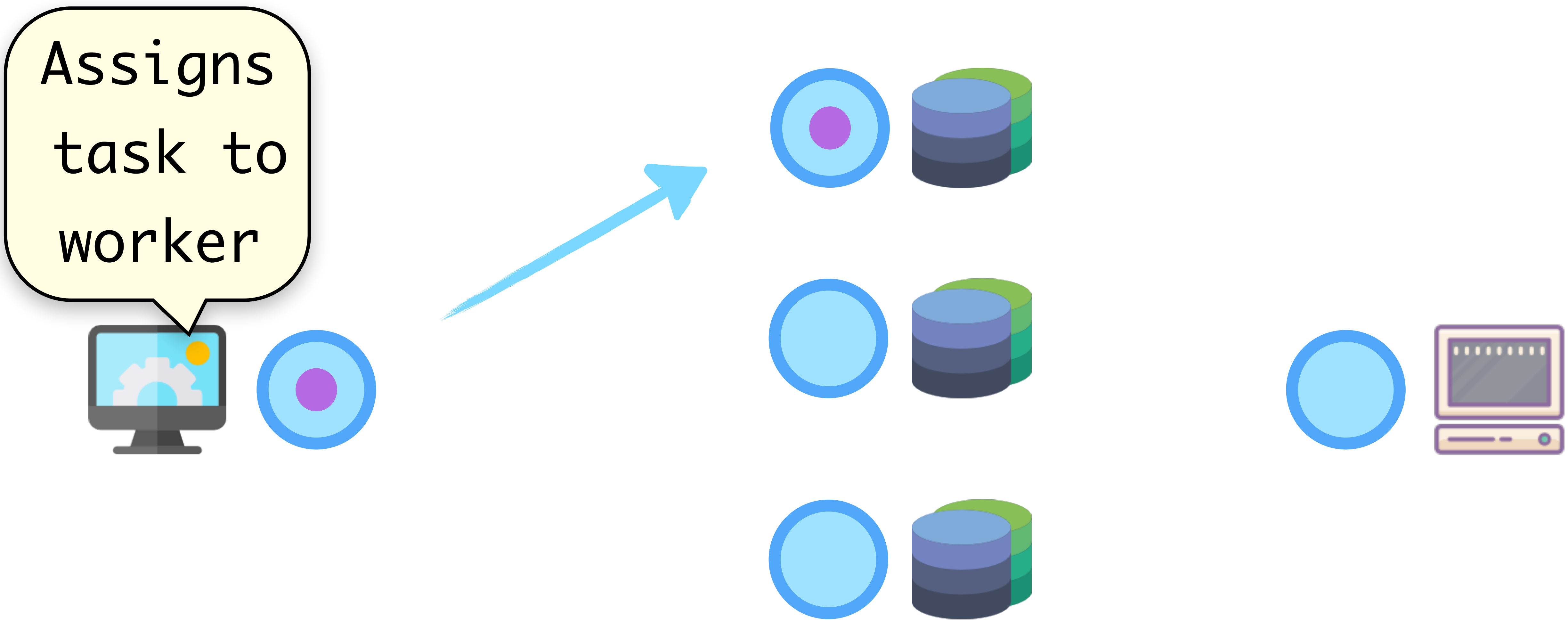
Example 2: Work stealing Queue

Work stealing Queue

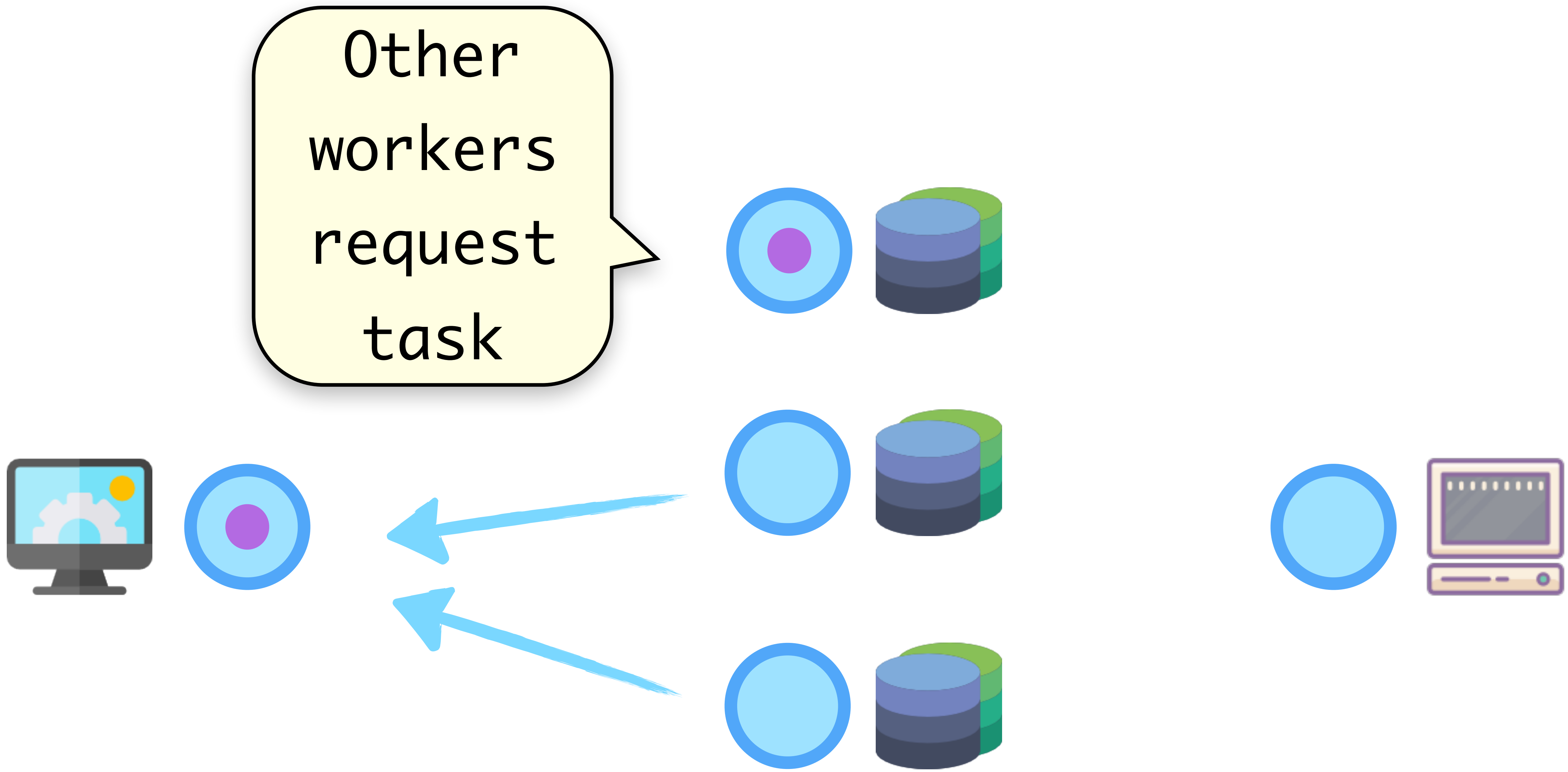
Work stealing Queue



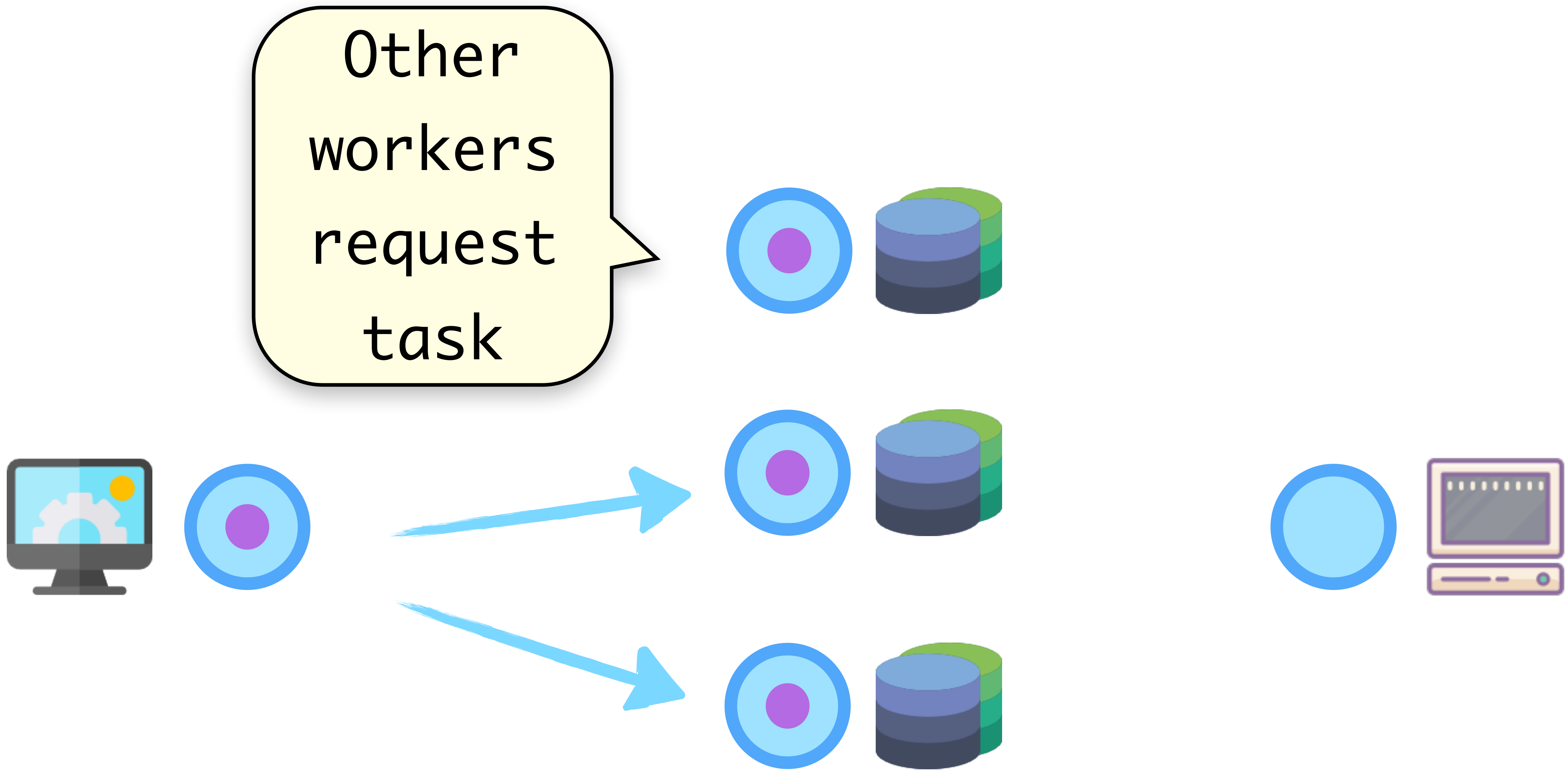
Work stealing Queue



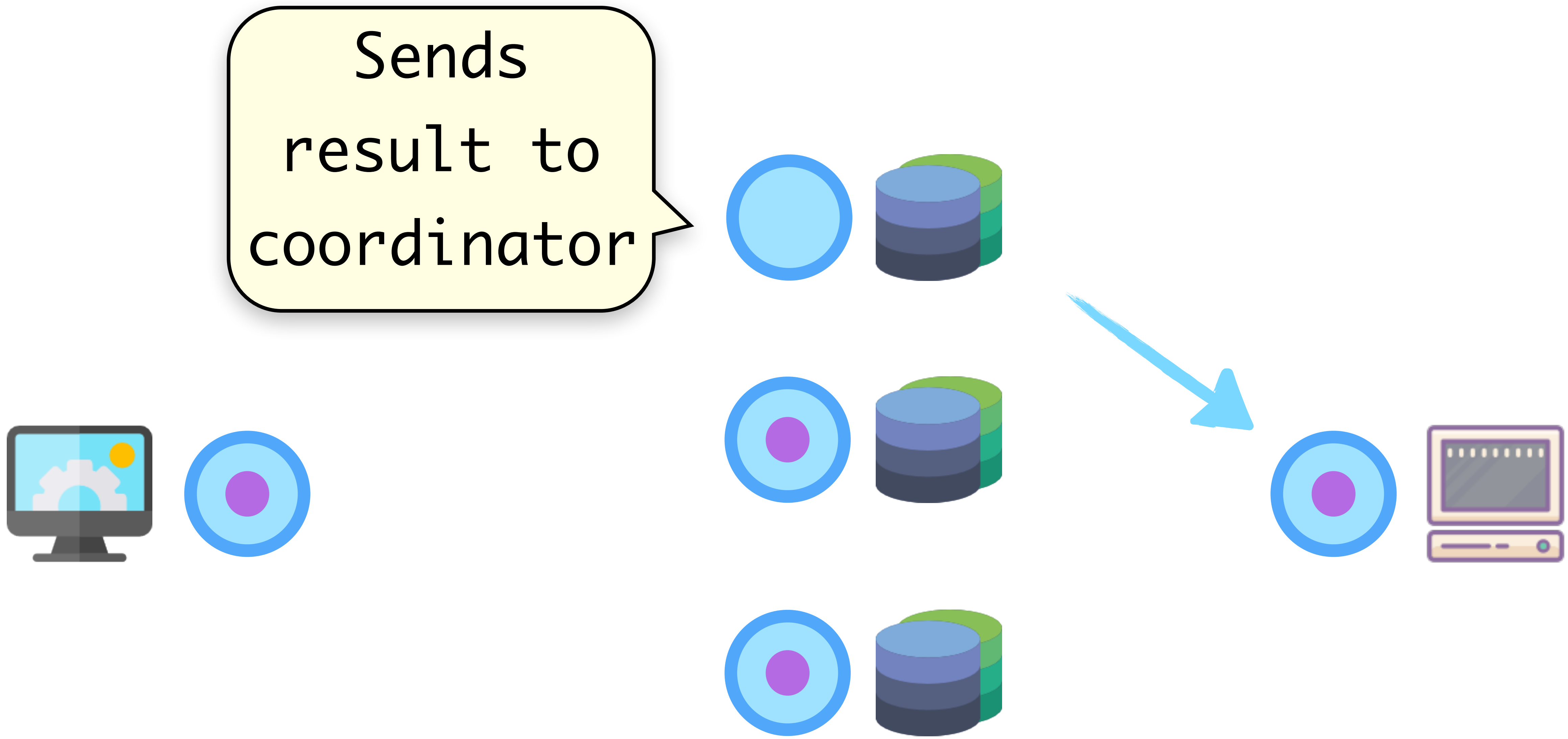
Work stealing Queue



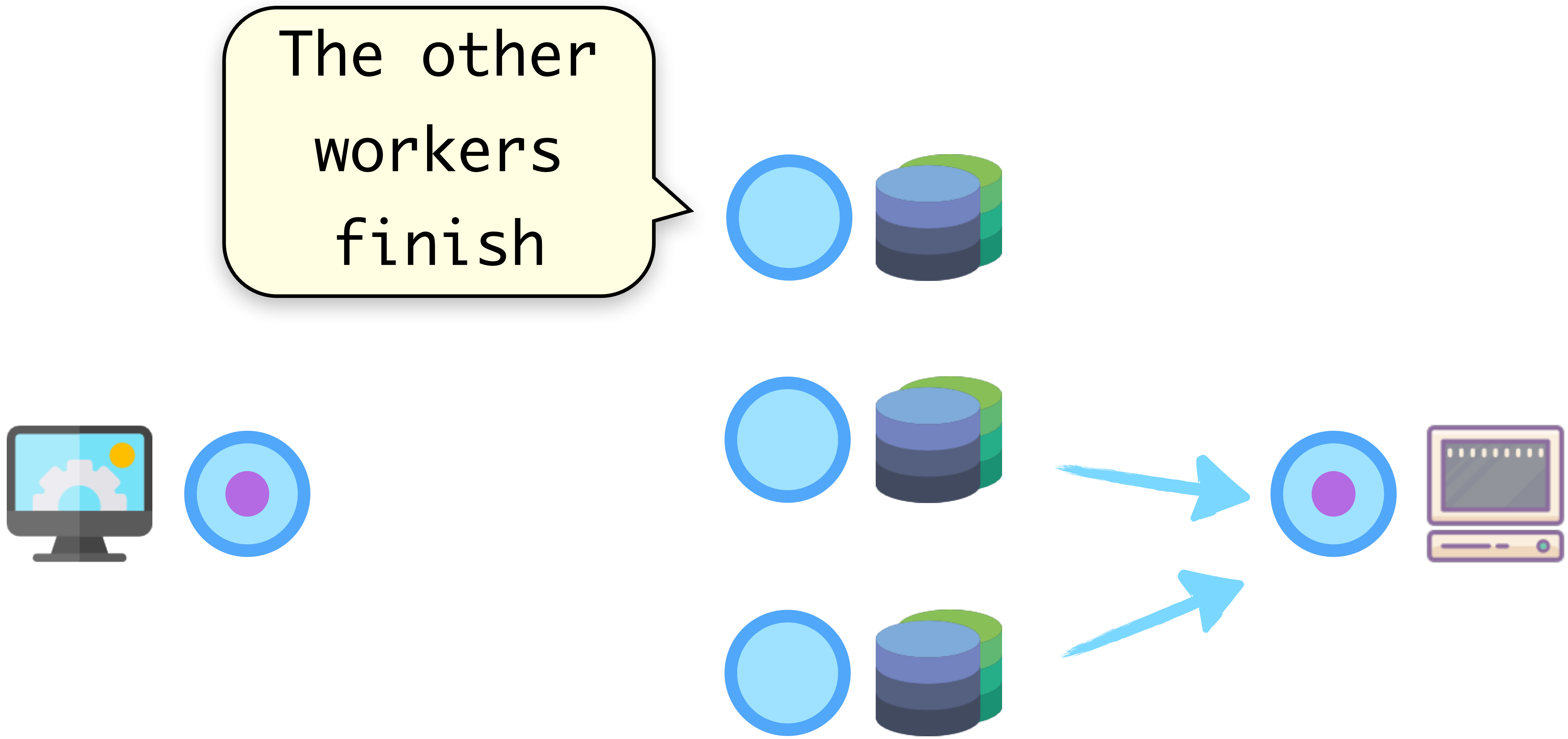
Work stealing Queue



Work stealing Queue



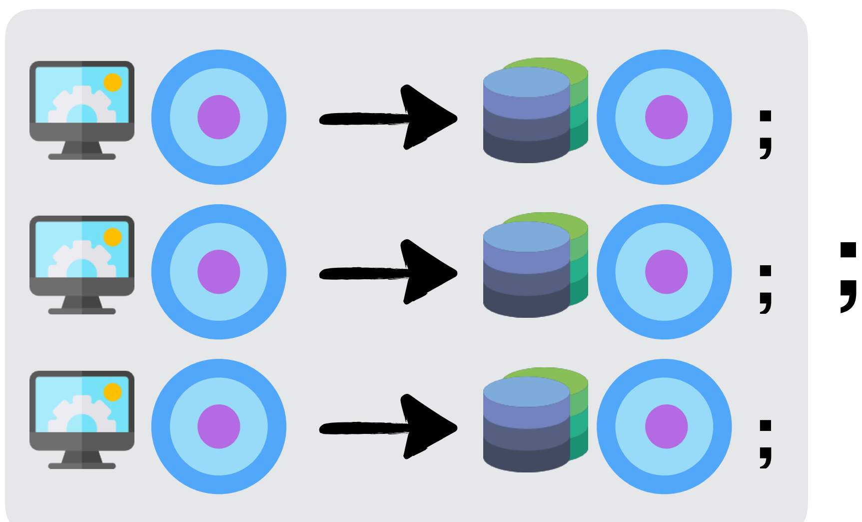
Work stealing Queue

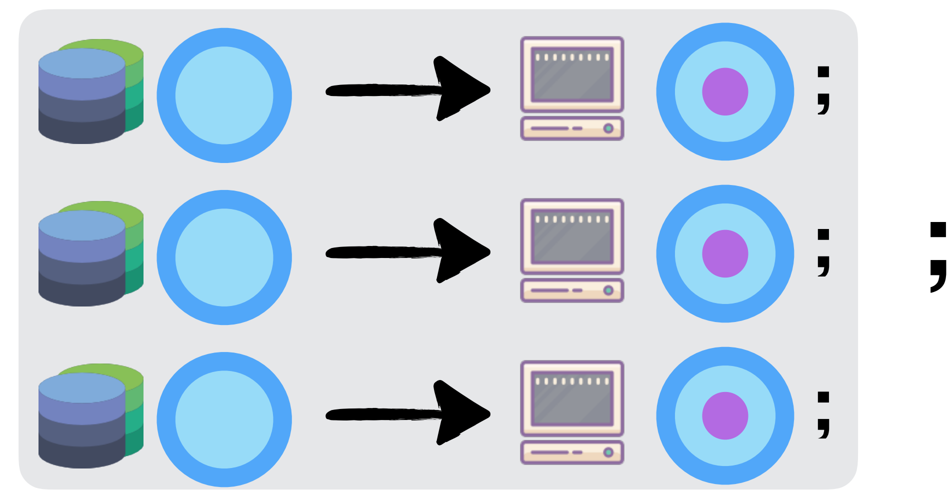


Work stealing Queue

Work stealing Queue

Synchronization





1.
Queue assigns tasks to workers that write result to *set*

2.
Workers pick results from *set* and send to collector

Outline

Key Idea: Pretend Synchrony

1. Computing Synchronizations
2. Verifying the Synchronization

Extensions

Evaluation

1. Computing Synchronizations

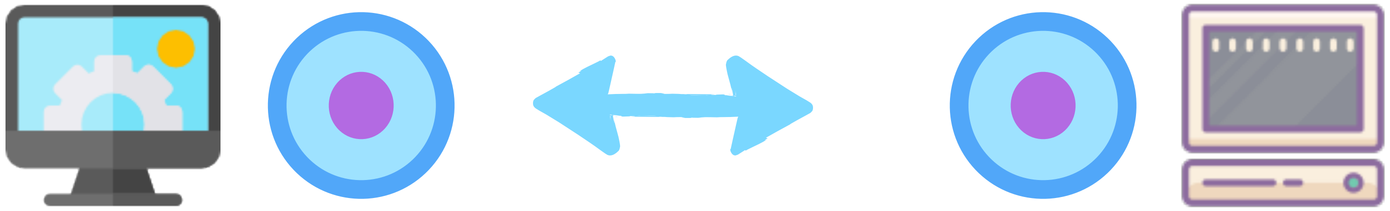
(By Rewriting and By Example)

Synchronize by Rewriting

Example 1: Loop Free

Synchronize by Rewriting

Example 1



Synchronize by Rewriting

Example 1

process p

process q

```
send q Ping;      ||   v <- recv p;
w <- recv q;           send p Pong;
```

Synchronize by Rewriting

Example 1

Since there is only a *single order*

```
send q Ping;        v <- recv p;
                ||
w <- recv q;        send p Pong;
```

Synchronize by Rewriting

Example 1

Since there is only a *single order*

```
send q Ping;
v <- recv p;
```

```
w <- recv q;   ||   send p Pong;
```

… we can sequentialize *the send & receive (Lipton'75)*

Synchronize by Rewriting

Example 1

… we can *sequentialize the send & receive (Lipton'75)*

```
send q Ping;
```

```
v <- recv p;
```

```
w <- recv q;
```

||

```
send p Pong;
```

Synchronize by Rewriting

Example 1

… we can *sequentialize the send & receive (Lipton'75)*

```
q.v <- Ping;
```

```
w <- recv q;  ||  send p Pong;
```

… and replace them by an *assignments*

Synchronize by Rewriting

Example 1

… we can *sequentialize the send & receive (Lipton'75)*

```
q.v <- Ping;
```

```
send p Pong;
```

```
w <- recv q;
```

… and replace them by an *assignments*

Synchronize by Rewriting

Example 1

… we can *sequentialize the send & receive (Lipton'75)*

```
q.v <- Ping;
p.w <- Pong;
```

… and replace them by an *assignments*

Synchronize by Rewriting

Example 1

Synchronization

```
q.v <- Ping;
```

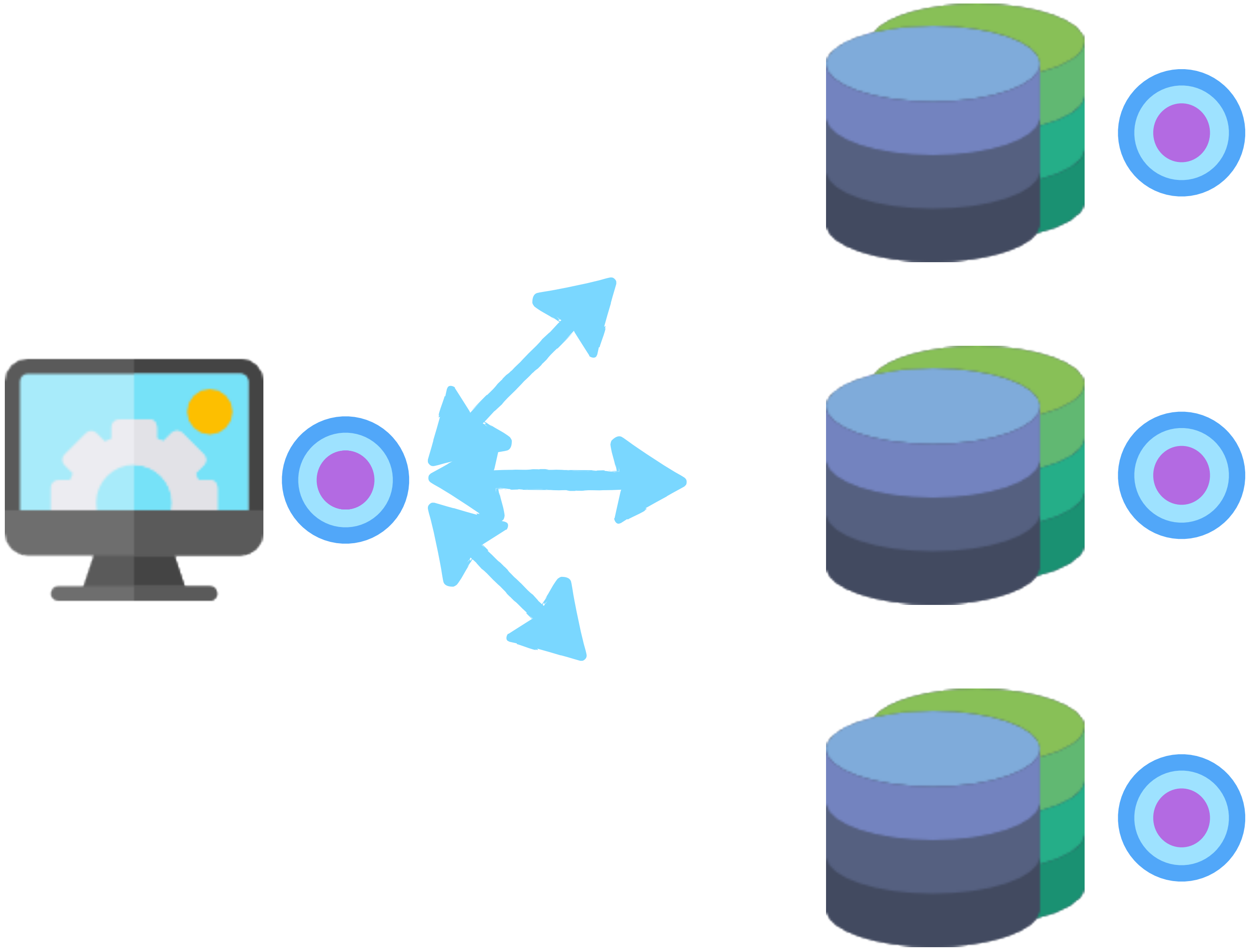
```
p.w <- Pong;
```

Synchronize by Rewriting

Example 2 : Loop over Processes

Synchronize by Rewriting

Example 2



Synchronize by Rewriting

Example 2

Synchronize by Rewriting

Example 2

p loops over qs

```
for q in qs do
  send q Ping;
  w <- recv q;
end
```

$\Big\|\ \prod_{q \in qs}$

```
v <- recv p;
send p Pong;
```

Synchronize by Rewriting

Example 2

Since iterations are *sequential*

```
for q in qs do
    send q Ping;
    w <- recv q;
end
```

$\|$ $\prod_{q \in qs}$

```
v <- recv p;
send p Pong;
```

… and each iteration talks to *a single process*

Synchronize by Rewriting

Example 2

… *focus* on *arbitrary* iteration

```
for q in qs do
  send q Ping;
  w <- recv q;
end
```

$\parallel \;\; \prod_{q \in qs}$

```
v <- recv p;
send p Pong;
```

Synchronize by Rewriting

Example 2

… *focus* on *arbitrary* iteration, *synchronize*

```
for q in qs do
    q.v <- Ping;
    p.w <- Pong;
end
```

Synchronize by Rewriting

Example 2

… *focus* on *arbitrary* iteration, *synchronize*

```
for q in qs do
    q.v <- Ping;
    p.w <- Pong;
end
```

… and *generalize* (Materialization, Sagiv'99)

Synchronize by Rewriting

Example 2

```
for q in qs do
    q.v <- Ping;
    p.w <- Pong;
end
```

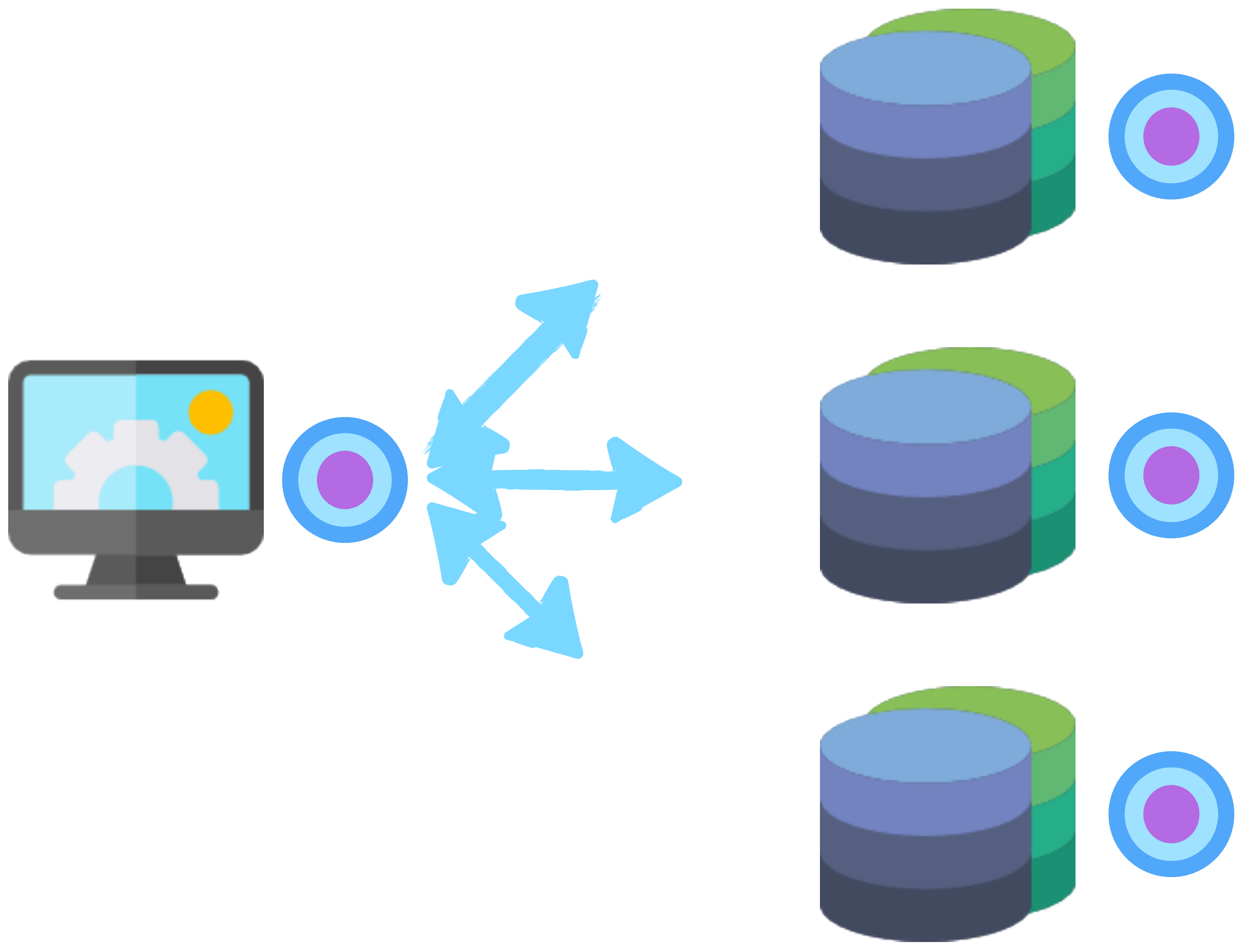
Synchronization

Synchronize by Rewriting

Example 3 : Symmetric Races

Synchronize by Rewriting

Example 3



Synchronize by Rewriting

Example 3

two loops

```
for q in qs do
  send q Ping;
end
for _ in qs do
  w <- recv qs;
end
```

||

$\prod_{q \in qs}$

```
v <- recv p;
send p Pong;
```

Synchronize by Rewriting

Example 3

Split the rewrite into two *sequential steps*

```
for q in qs do
  send q Ping;
end
for _ in qs do
  w <- recv qs;
end
```

$$\| \prod_{q \in qs}$$

```
v <- recv p;
send p Pong;
```

… the first loop and its *receive*, then the rest

Synchronize by Rewriting

Example 3

… some *qs* might send during the first loop

```
for q in qs do
  send q Ping;
end
for _ in qs do
  w <- recv qs;
end
```

$\|$ $\prod_{q \in qs}$

```
v <- recv p;
send p Pong;
```

we can *pretend* the sends happen later (*Lipton'75*)

Synchronize by Rewriting

Example 3

Since the loop is *sequential*

```
for q in qs do
  send q Ping;
end
for _ in qs do
  w <- recv qs;
end
```

$\|\ \prod_{q \in qs}$

```
v <- recv p;
send p Pong;
```

Synchronize by Rewriting

Example 3

Since the loop is *sequential*

```
for q in qs do
    q.v <- Ping;          ||   Π       send p Pong;
end                           q∈qs
for _ in qs do
    w <- recv qs;
end
```

… synchronize arbitrary an iteration

Synchronize by Rewriting

Example 3

Since the loop is *sequential*

```
for q in qs do
  q.v <- Ping;
end
```

```
for _ in qs do
  w <- recv qs;
end
```

$\|$ $\prod_{q \in qs}$ `send p Pong;`

… and generalize

Synchronize by Rewriting

Example 3

Problem: Iterations are no longer *sequential*

```
for _ in qs do
  w <- recv qs;
end
```

$\|$ $\prod_{q \in qs}$ `send p Pong;`

… there is a *race* between the processes in *qs*

Synchronize by Rewriting

Example 3

Problem: Iterations are no longer *sequential*

```
for _ in qs do
  w <- recv qs;
end
```

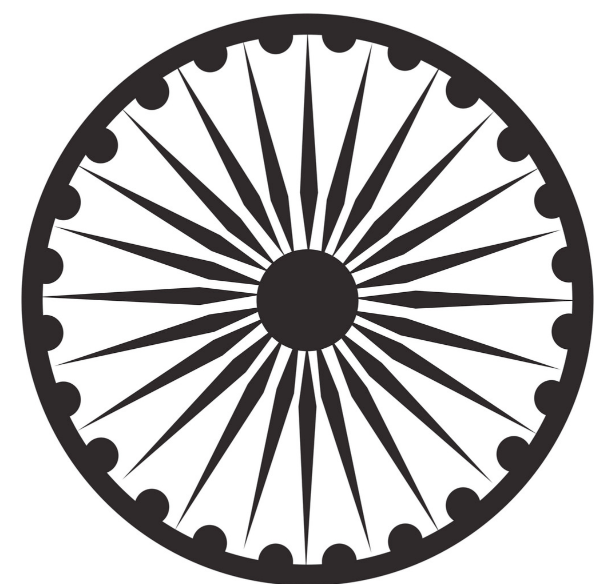
$\| \prod_{q \in qs}$ `send p Pong;`

… how can we synchronize?

Synchronize by Rewriting

Exploit

Sequence

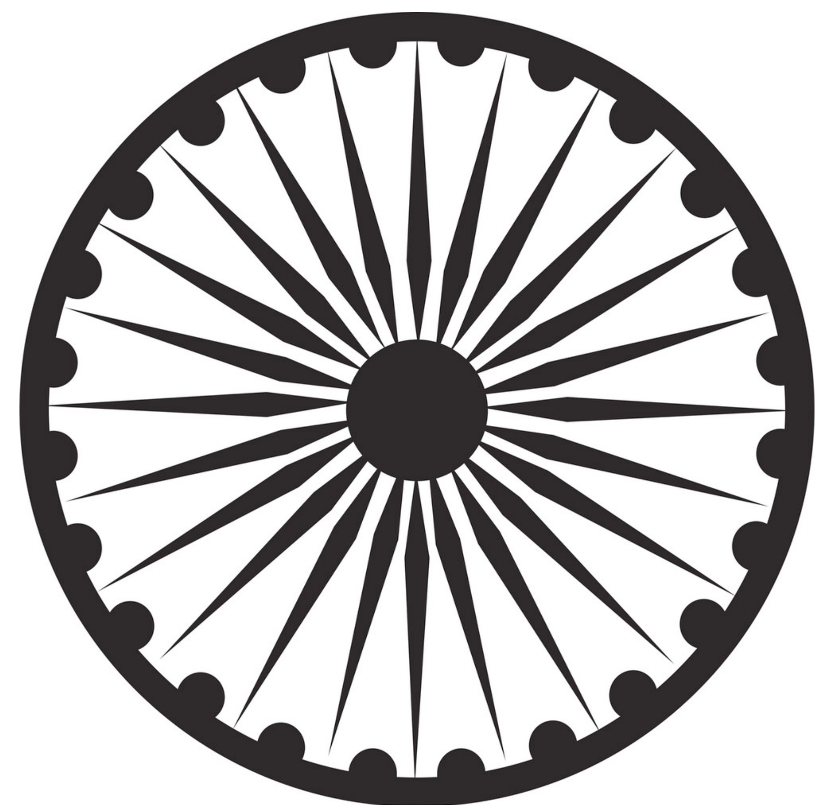


Symmetry

Synchronize by Rewriting

Exploit

Sequence

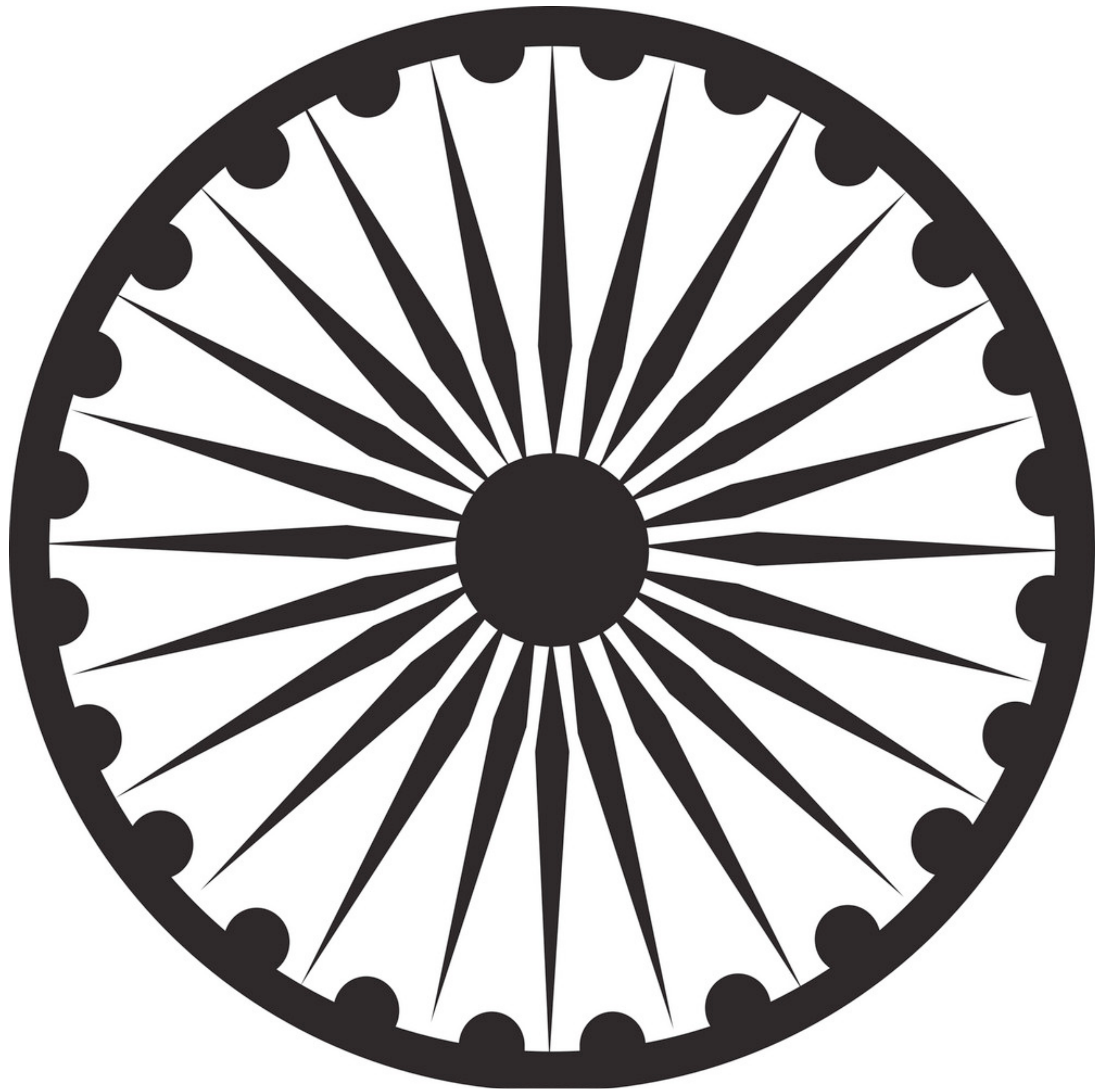


Symmetry

Symmetry

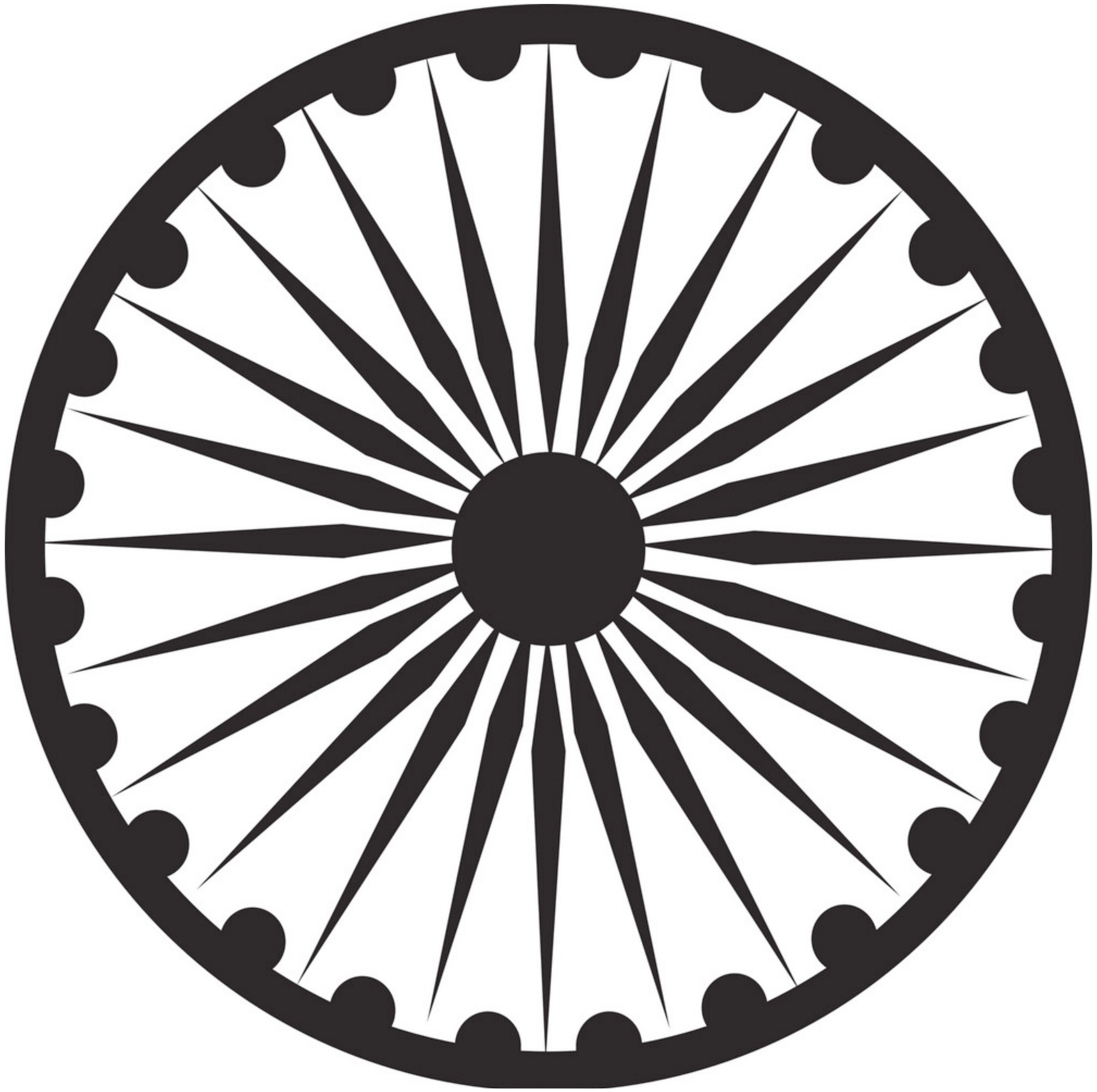
Invariance under *transformation*

Symmetry



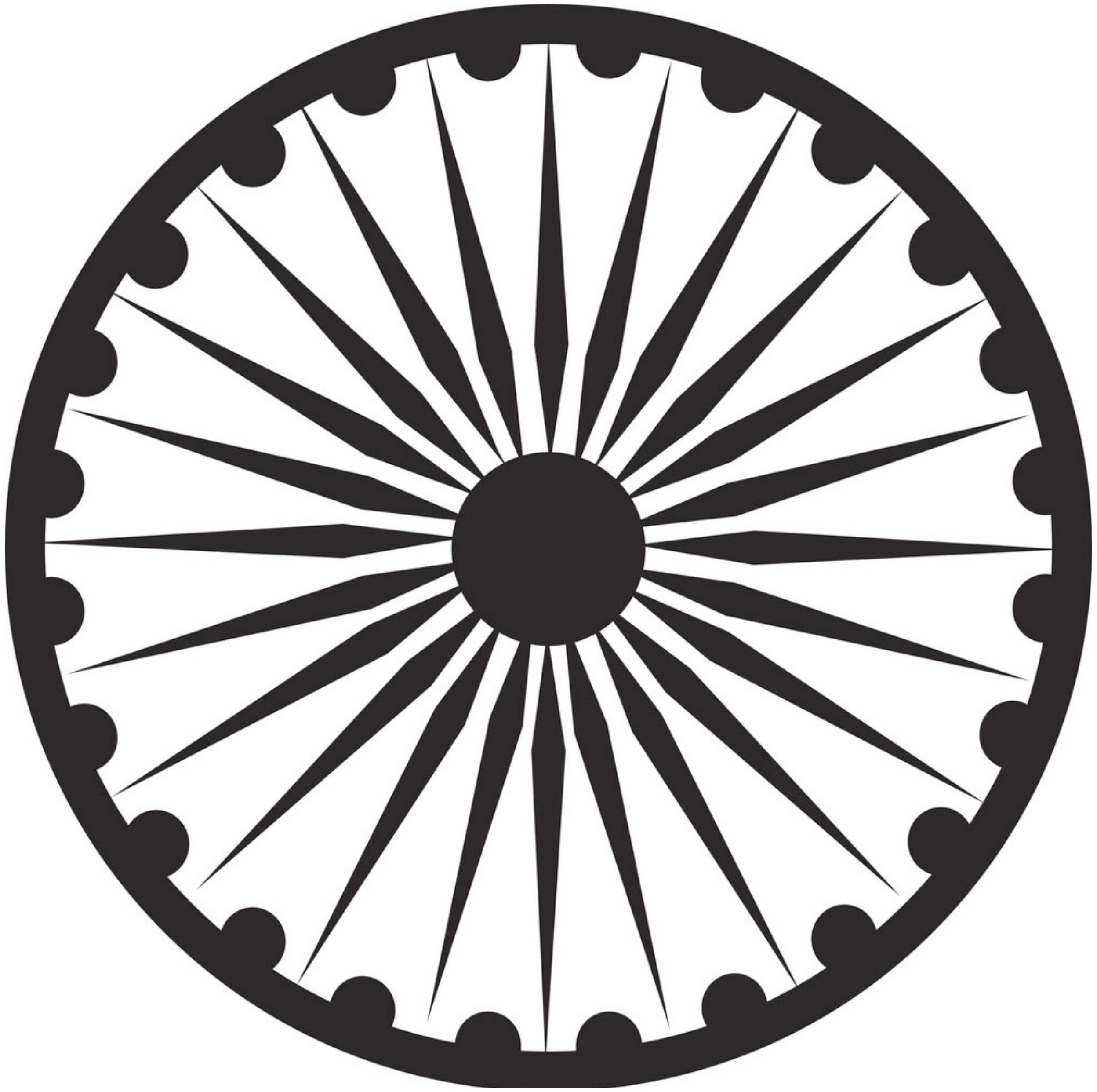
Invariance under *transformation*

Symmetry



Invariance under *rotation*

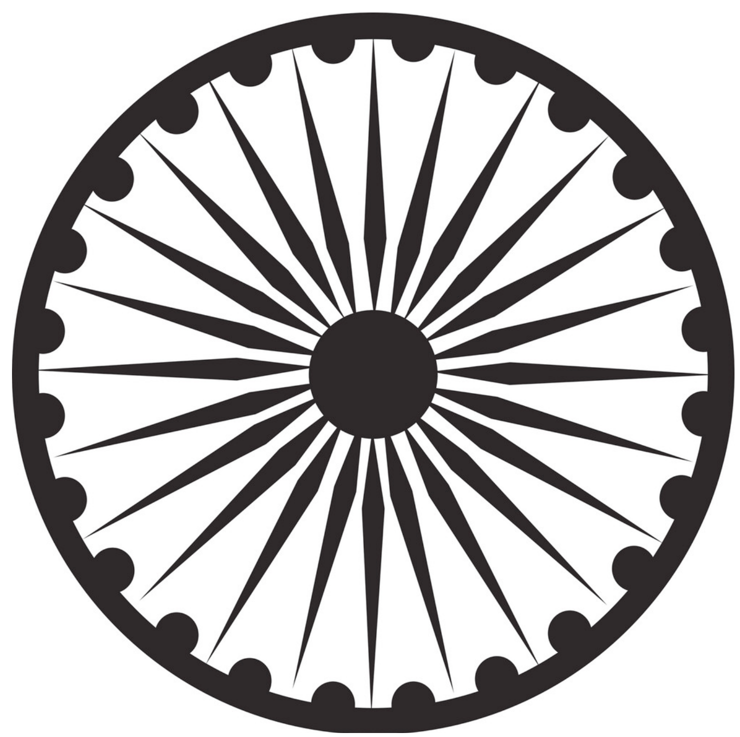
Symmetry



Invariance under *rotation*

Symmetry in Distributed Systems

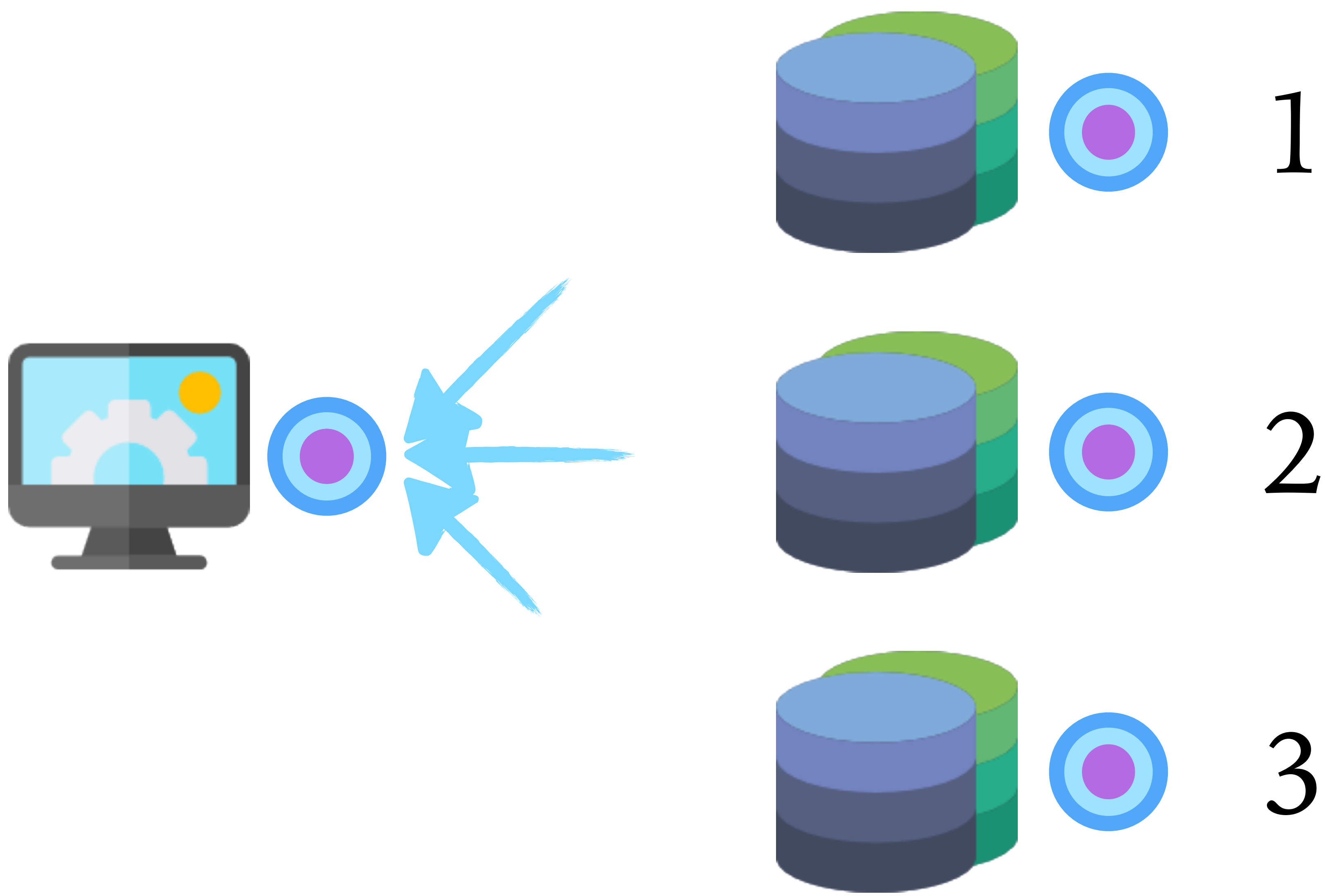
Invariance under process-id permutation



… doesn't affect halting states (Ip &Dill 1996)

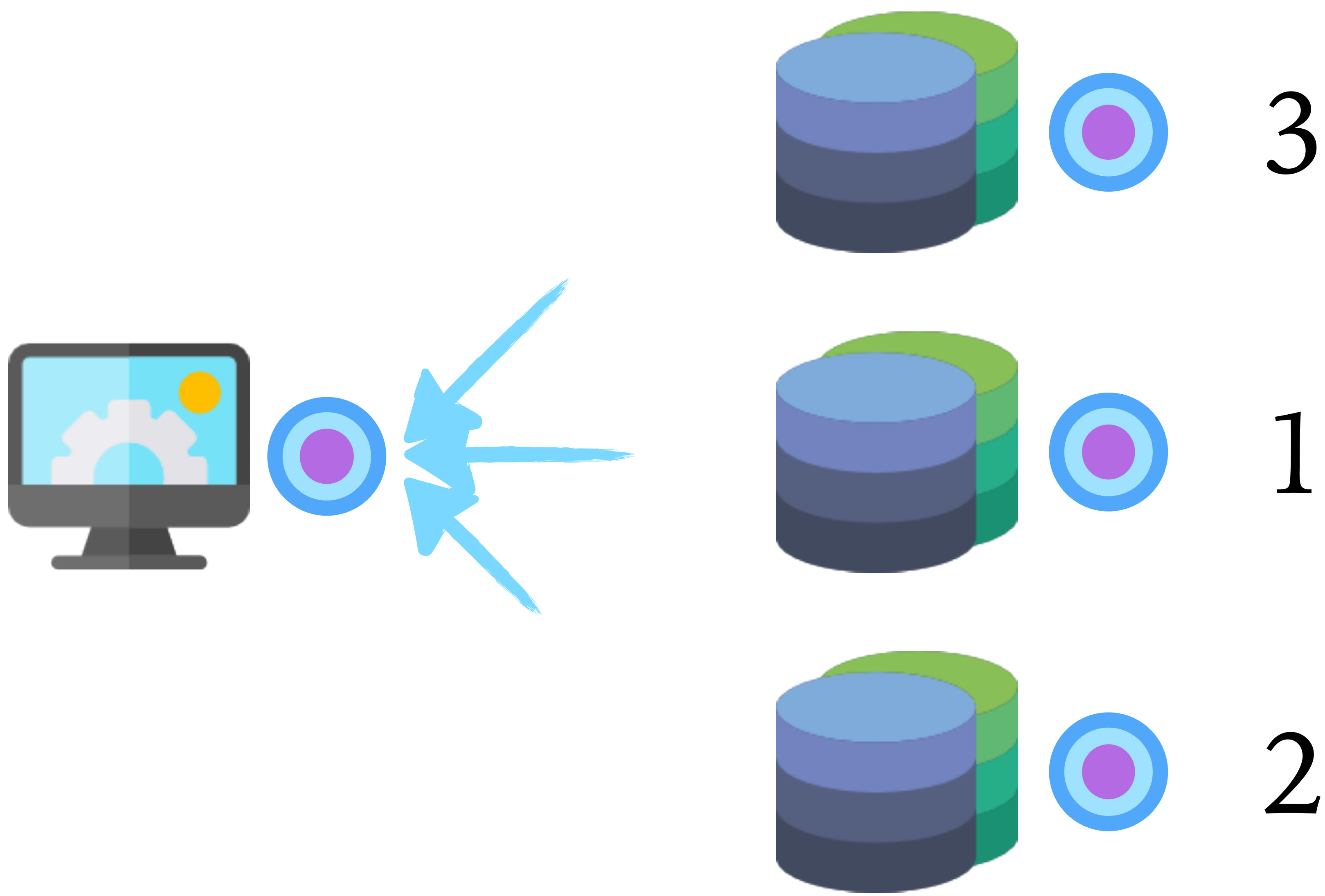
Symmetry in Distributed Systems

Name nodes …



Symmetry in Distributed Systems

permuting process names



… does not affect halting states

Use Symmetry to Synchronize!

Synchronize by Rewriting

Example 3

Since the processes in *qs* are *symmetric*

```
for _ in qs do
  w <- recv qs;
end
```

$\|$ $\prod_{q \in qs}$ `send p Pong;`

… we can receive the messages in *any* order

Synchronize by Rewriting

Example 3

… we can receive the messages in *any* order

```
for _ in qs do
  w <- recv qs;
end
```

$||$ $\prod_{q \in qs}$ `send p Pong;`

Synchronize by Rewriting

Example 3

… we can receive the messages in *any* order

```
for q in qs do
  w <- recv q;
end
```

$\parallel \; \prod_{q \in qs}$ `send p Pong;`

… in particular, we the *iteration order* of the loop

Synchronize by Rewriting

Example 3

… in particular, we the *iteration order* of the loop

```
for q in qs do
  w <- recv q;
end
```

|| $\prod_{q \in qs}$ `send p Pong;`

… since the loop is now *sequential*

Synchronize by Rewriting

Example 3

… since the loop is now *sequential*

```
for q in qs do
  w <- recv q;
end
```

$$\Big\Vert \prod_{q \in qs}$$

```
send p Pong;
```

Synchronize by Rewriting

Example 3

… since the loop is now *sequential*

```
for _ in qs do
  p.w <- Pong;
end
```

… we can synchronize, as before

Synchronize by Rewriting

Example 3

… since the loop is now *sequential*

```
for _ in qs do
    p.w <- Pong;
end
```

… we can synchronize, as before

Synchronize by Rewriting

Example 3

… we can synchronize, as before

```
for _ in qs do
  p.w <- Pong;
end
```

Synchronize by Rewriting

Example 3

… we can synchronize, as before

```
for q in qs do
  q.v <- Ping;
end
for _ in qs do
  p.w <- Pong;
end
```

… and get the overall synchronization

Synchronize by Rewriting

Example 3

```
for q in qs do
    q.v <- Ping;
end
for _ in qs do
    p.w <- Pong;
end
```

Synchronization

Synchronize by Rewriting

Example 4: Two Phase Commit

Synchronize by Rewriting

Two Phase Commit: Phase 1

```
for p in dbs do
  send p (c,val)
```

```
abort <- False
```

```
for p in dbs do
  msg <-recv
  if msg == A
    abort <- True
```

$\Bigg|\Bigg| \quad \prod_{p \in dbs}$

```
(id,val) <-recv
```

```
vote <- * ? C : A
```

```
send id vote
```

Synchronize by Rewriting

Two Phase Commit: Phase 2

```
dec <- abort ? A : C

for p in dbs do
  send p dec

for p in dbs do
  _ <-recv
```

$\|\ \prod_{p \in dbs}$

```
dec <- recv

if dec == C
  value <- val

send id Ack
```

Synchronize by Rewriting

Two Phase Commit : Synchronizing Phase 1

```
for p in dbs do
  send p (c,val)
```

```
abort <- False
```

```
for p in dbs do
  msg <-recv
  if msg == A
    abort <- True
```

$||$ $\prod_{p \in dbs}$

```
(id,val) <-recv
```

```
vote <- * ? C : A
```

```
send id vote
```

… by example 3

Synchronize by Rewriting

Two Phase Commit : Synchronizing Phase 1

```
for p in dbs do
  p.id<-c;
  p.val<-c.val;
```

```
abort <- False
for p in dbs do
  msg <-recv
  if msg == A
    abort <- True
```

$\|\ \prod_{p \in dbs}$

```
vote <- * ? C : A
send id vote
```

… by example 3

Synchronize by Rewriting

Two Phase Commit : Synchronizing Phase 1

```
for p in dbs do
   p.id<-c;
   p.val<-c.val;
c.abort<-False;
```

```
for p in dbs do
  msg <-recv
  if msg == A
    abort <- True
```

|| $\prod_{p \in dbs}$

```
vote <- * ? C : A
send id vote
```

… by example 3

Synchronize by Rewriting

```
for p in dbs do
  p.id<-c;
  p.val<-c.val;
c.abort<-False;
for p in dbs do
  vote <-* ? C : A
  c.msg<-p.vote;
  if msg == A
    abort <- True
```

Synchronized Phase 1

Outline

Key Idea: Pretend Synchrony

1. Computing Synchronizations

2. Verifying the Synchronization

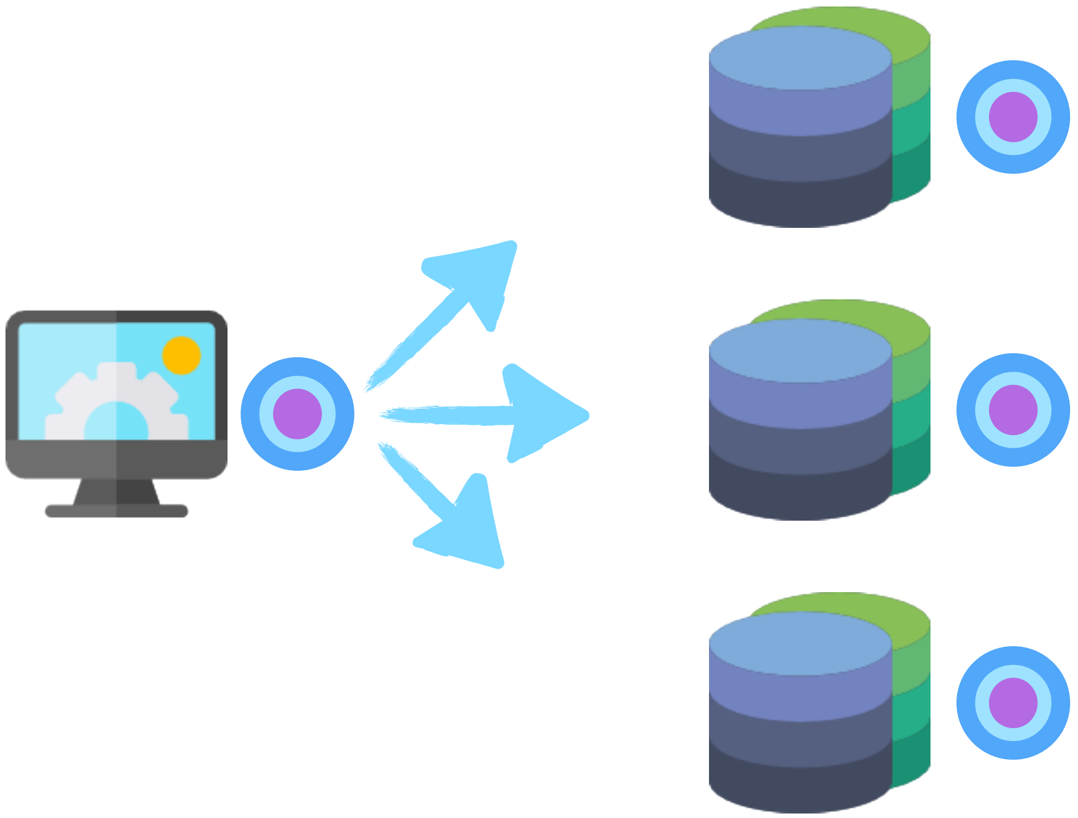
Extensions

Evaluation

2. Verifying the Synchronization

Synchronous Proofs Are Easy!

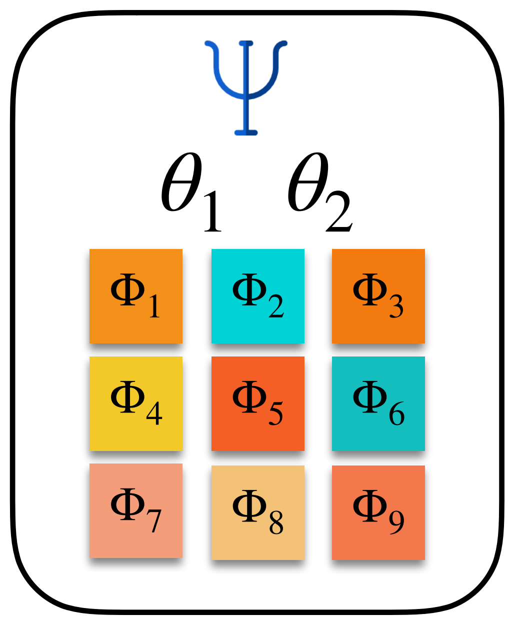
2PC: Correctness



Ψ = Nodes *agree* on *same value*

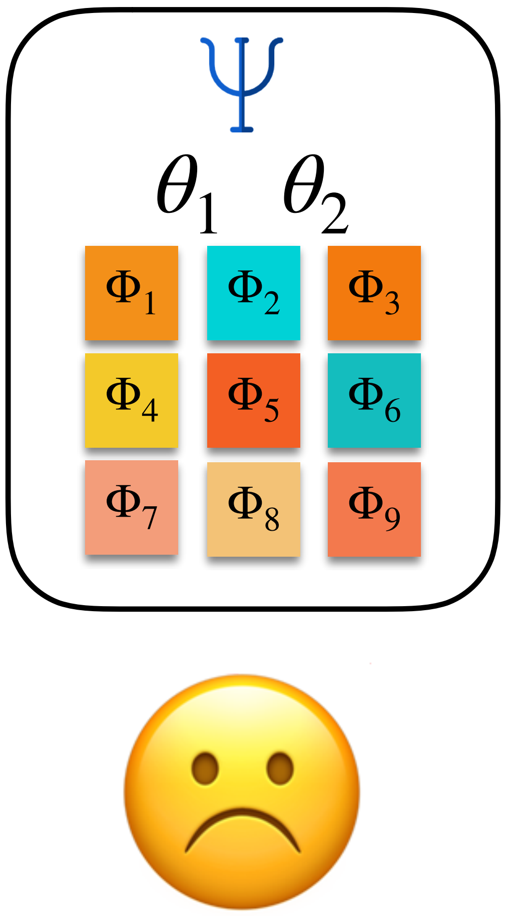
Asynchronous Proofs are *Ugly*!

Asynchronous Proofs



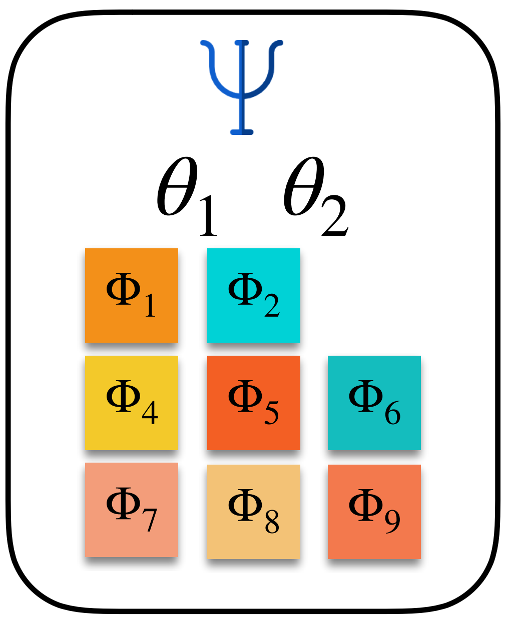
Asynchronous Proofs

Enumerate schedules and network state



Asynchronous Proofs

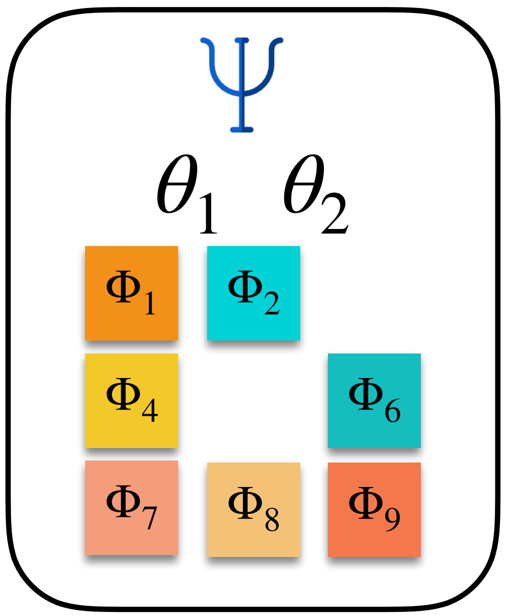
Enumerate schedules and network state



Φ_3  did send to 

Asynchronous Proofs

Enumerate schedules and network state

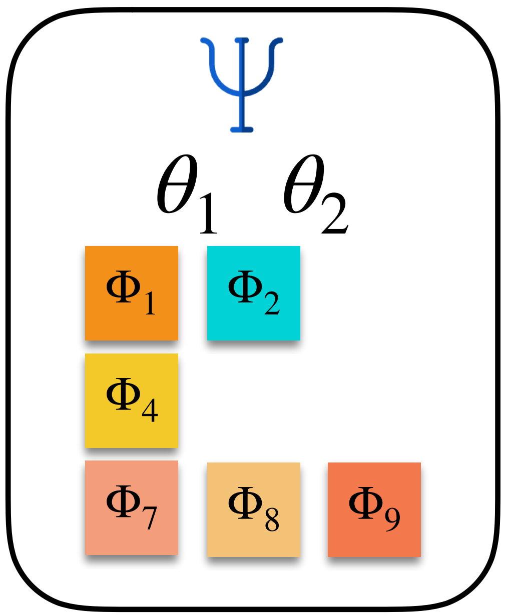


Φ_3  did send to 

$\vee$ Φ_5  didn't execute its receive

Asynchronous Proofs

Enumerate schedules and network state



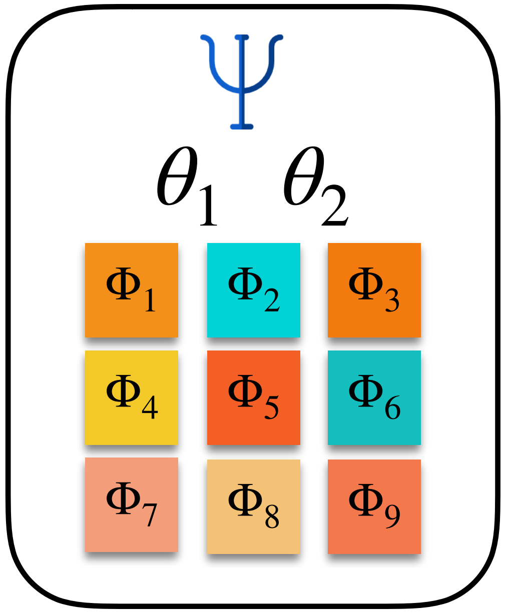
Φ_3 did send to

$\vee$ Φ_5 didn't execute its receive

$\vee$ Φ_6 there is messages from to containing 's ID and value

Synchronous Proofs are *Nice*!

Asynchronous Proofs



Synchronous Proofs

$$\boxed{\begin{array}{c}\Psi \\ \theta_1 \quad \theta_2\end{array}}$$

Synchronous Proofs

$$\boxed{\begin{matrix} \Psi \\ \theta_1 \quad \theta_2 \end{matrix}}$$

Synchronous Proofs

Ψ

$\theta_1 \quad \theta_2$

```
for p in dbs do
  p.id<-c;
  p.val<-c.val;
```

$$\theta_1 = \forall p \in dbs\,.\,p \in done \rightarrow p\,.\,val = c\,.\,val$$

Synchronous Proofs

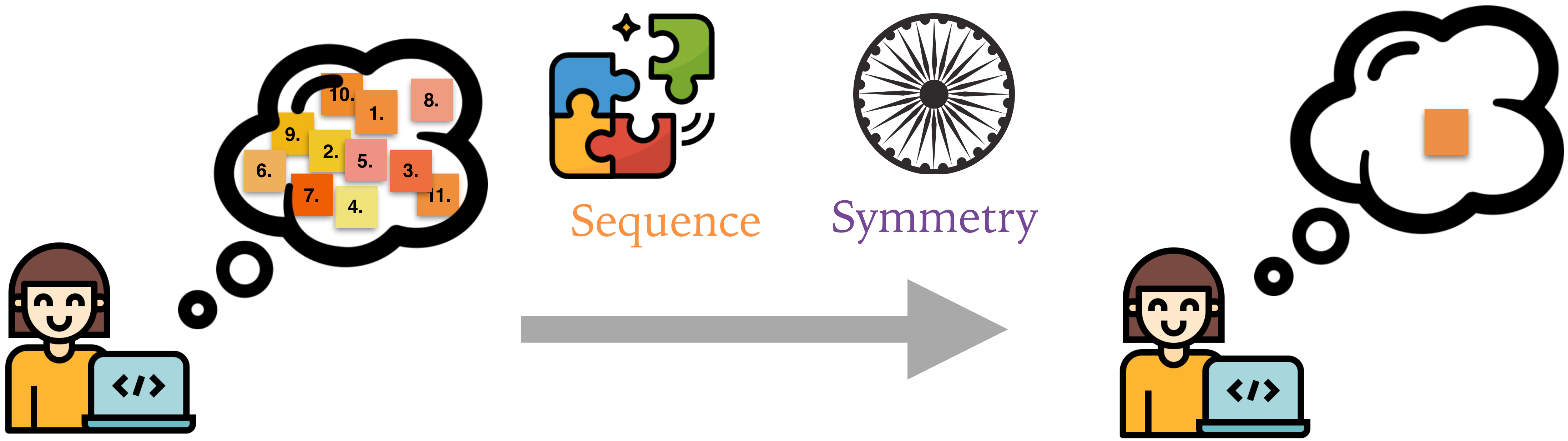
Ψ

$\theta_1 \quad \theta_2$

```
for p in dbs do
    if msg == C
      p.value <- p.val;
    _<-Ack;
```

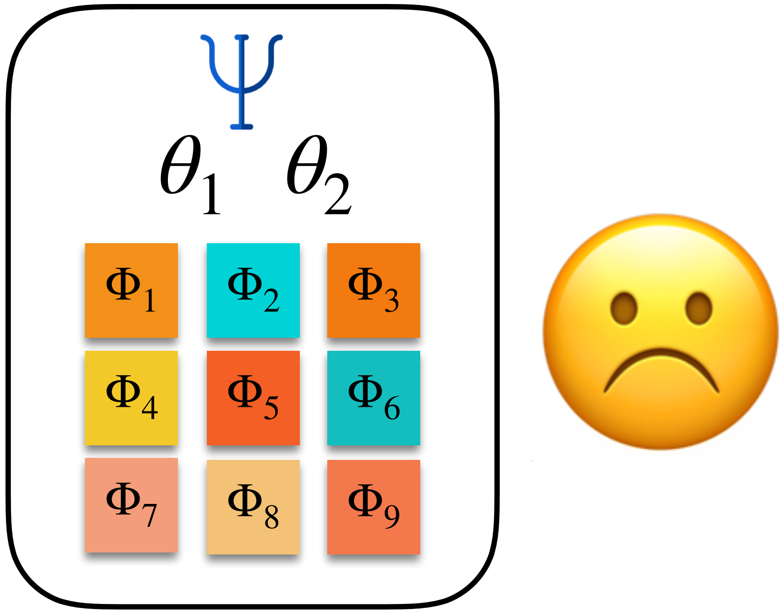
$$\theta_2 = \forall p \in dbs \,.\, p \in done \wedge c \,.\, dec = C \Rightarrow p \,.\, value = c \,.\, val$$

Recap



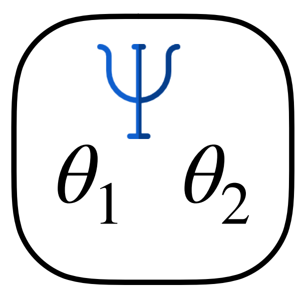
Compute synchronizations using *sequence* and *symmetry*

Recap



Makes Deductive Proofs Easier

Recap



😌

Makes Deductive Proofs Easier

Outline

Key Idea: Pretend Synchrony

1. Computing Synchronizations

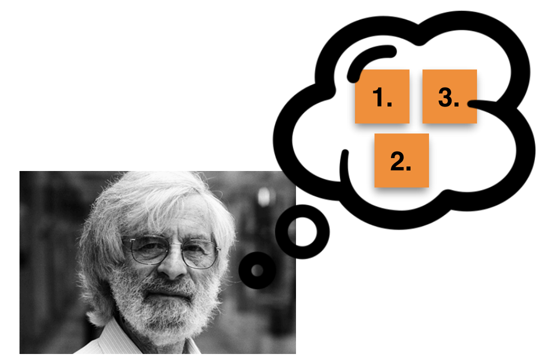
2. Verifying the Synchronization

Extensions

Evaluation

Extensions

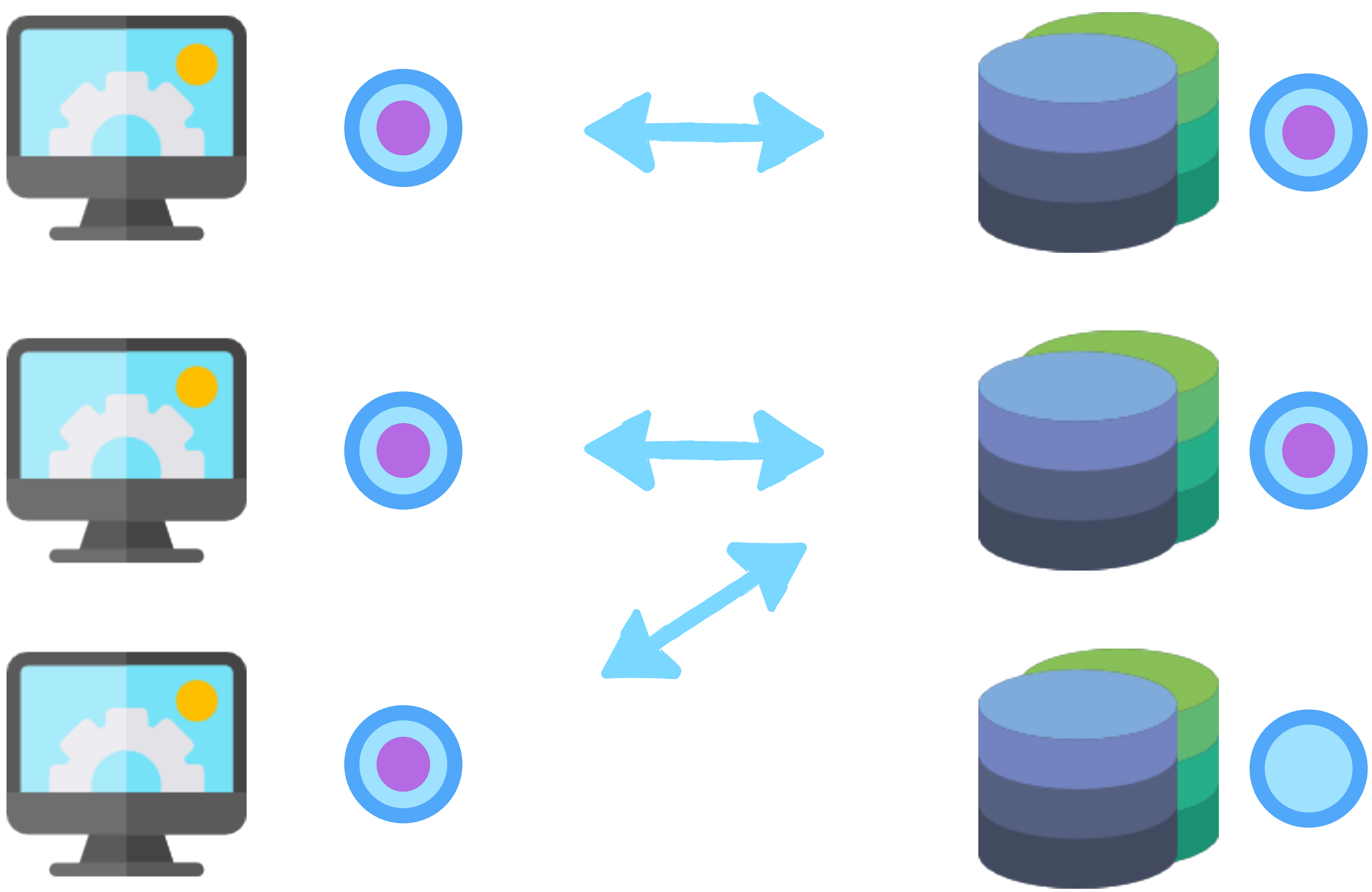
Multicasts



Message Drops

Rounds

Multicasts



Multicasts

Multicasts

Problem: neither sequential nor symmetric

$\Pi_{p \in ps}$

```
for q in qs do
  send q (Ping,p);
  w <- recv q;
end
```

$\|$

$\Pi_{q \in qs}$

```
for _ in ps do
  (v,id) <- recv ps;
  send id Pong;
end
```

… can't compose *in sequence?* Compose *in parallel*!

Multicasts

… can't compose *in sequence?* Compose *in parallel*!

$\Pi_{p \in ps}$

```
for q in qs do
  send q (Ping,p);
  w <- recv q;
end
```

||

$\Pi_{q \in qs}$

```
for _ in ps do
  (v,id) <- recv ps;
  send id Pong;
end
```

Multicasts

… can't compose *in sequence?* Compose *in parallel*!

$\prod_{p \in ps}$

```
for q in qs do
    send q (Ping,p);
    w <- recv q;
end
```

$\|$ $\prod_{q \in qs}$

```
for _ in ps do
    (v,id) <- recv p;
    send id Pong;
end
```

… *focus* on *arbitrary* process *p*

Multicasts

… *focus* on *arbitrary* process *p*

```
Π p∈ps
for q in qs do
    send q (Ping,p);
    w <- recv q;
end

||

Π q∈qs
for _ in ps do
    (v,id) <- recv p;
    send id Pong;
end
```

… the interaction is *sequential!*

Multicasts

… the interaction is *sequential!*

$\prod_{p \in ps}$

```
for q in qs do
  send q (Ping,p);
  w <- recv q;
end
```

$\|$

$\prod_{q \in qs}$

```
for _ in ps do
  (v,id) <- recv p;
  send id Pong;
end
```

Multicasts

… the interaction is *sequential!*

π
p∈ps

```
for q in qs do
    (q.v,q.id)<-(Ping,p);
    p.w<-Pong
end
```

… synchronize (by example 2)

Multicasts

… synchronize (by example 2)

Π $p \in ps$

```
for q in qs do
  (q.v,q.id)<-(Ping,p);
  p.w<-Pong
end
```

Multicasts

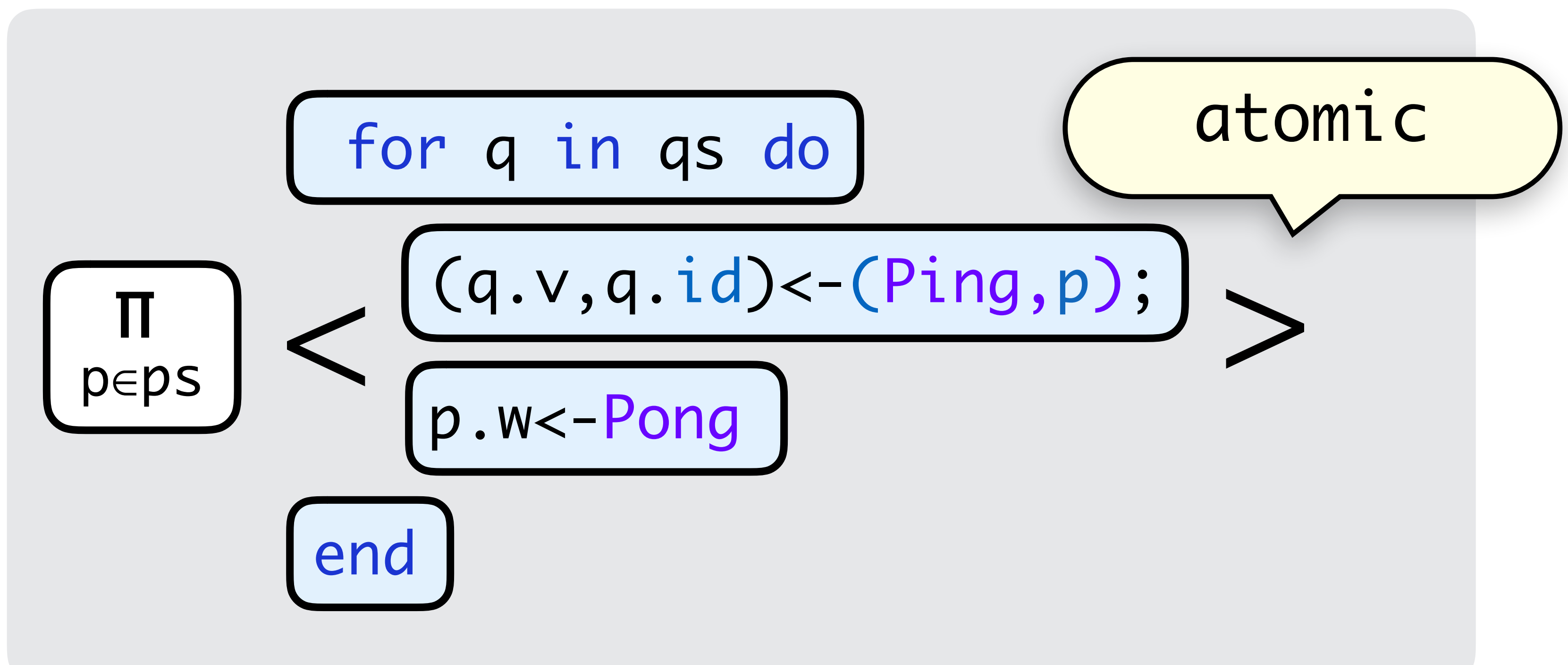
… synchronize (by example 2)

```
for q in qs do
  Π      < (q.v,q.id)<-(Ping,p);
  p∈ps     p.w<-Pong              >
end
```

… and *generalize*!

Multicasts

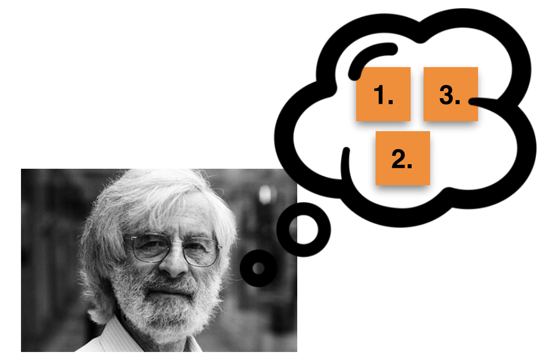
… and *generalize*!



results in a *concurrent*, *shared memory* program

Extensions

Multicasts

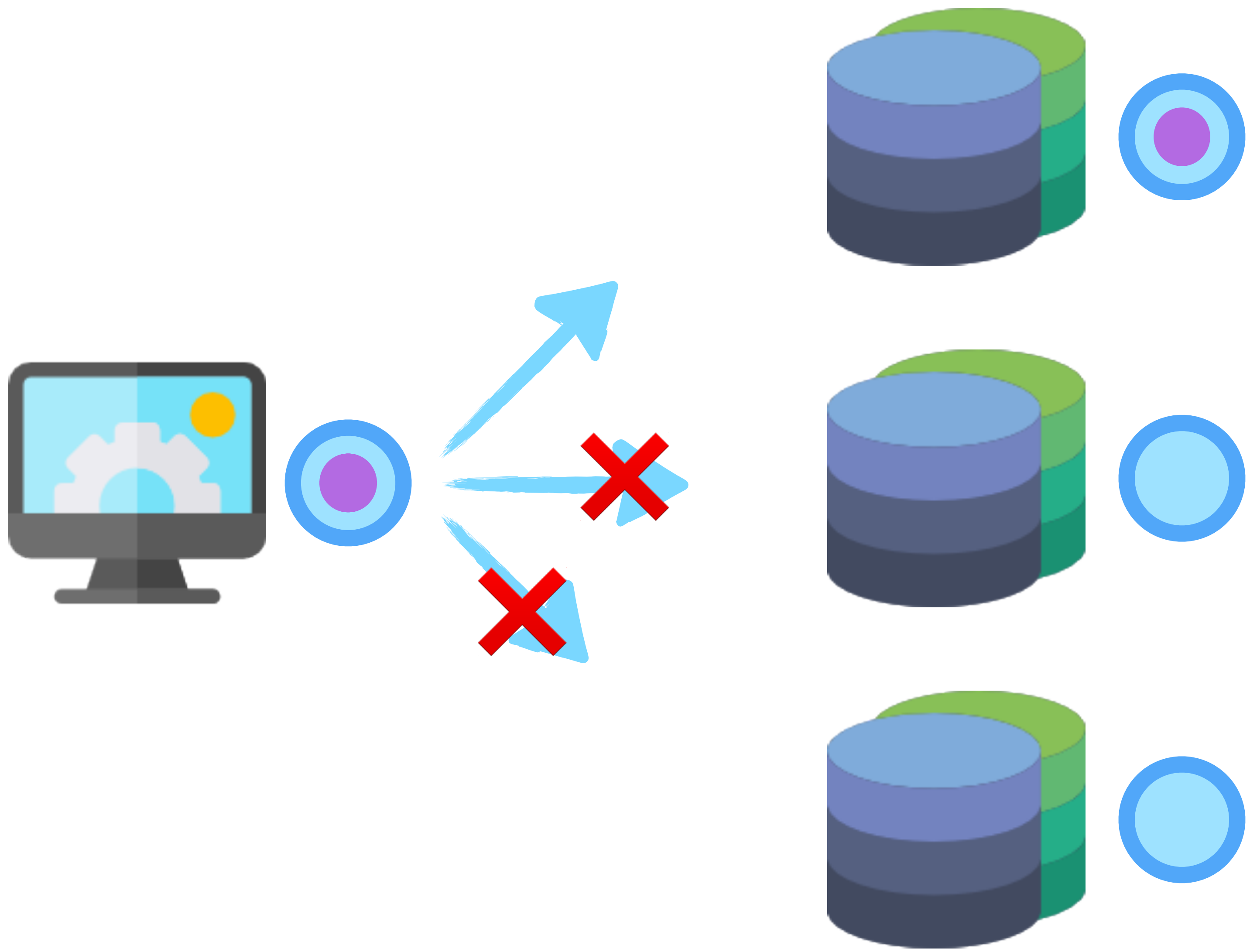


Message Drops

Rounds

Extensions

Message Drops



Extensions

Message Drops

Receive non-deterministically times out

```
for q in qs do
  send q Ping;
end
```

$\parallel \prod_{q \in qs}$

```
v <- recvTO p;
```

Extensions

Message Drops

The interaction is *sequential*

```
for q in qs do
  send q Ping;
end
```

$\|$ $\prod_{q \in qs}$

```
v <- recvTO p;
```

Extensions

Message Drops

The interaction is *sequential*

```
for q in qs do
  send q Ping;
end
```

$\|\ \prod_{q \in qs}$

```
v <- recvTO p;
```

… focus on a single iteration

Extensions

Message Drops

… focus on a single iteration

```
for q in qs do
    send q Ping;          ||   Π q∈qs   v <- recvTO p;
end
```

Extensions

Message Drops

… focus on a single iteration

```
for q in qs do
    send q Ping;
    v <- recvTO p;
end
```

… match up the send and receive

Extensions

Message Drops

… match up the send and receive

```
for q in qs do
    send q Ping;
    v <- recvTO p;
end
```

Extensions

Message Drops

… match up the send and receive

```
for q in qs do
  q.v <-* ?
      Just Ping:
      None
end
```

… *case-split* whether the message was received

Extensions

Message Drops

… *case-split* whether the message was received

```
for q in qs do
  q.v <-* ?
      Just Ping:
      None
end
```

Extensions

Message Drops

… *case-split* whether the message was received

```
for q in qs do
  q.v <-* ?
      Just Ping:
      None
end
```

… *and generalize*

Extensions

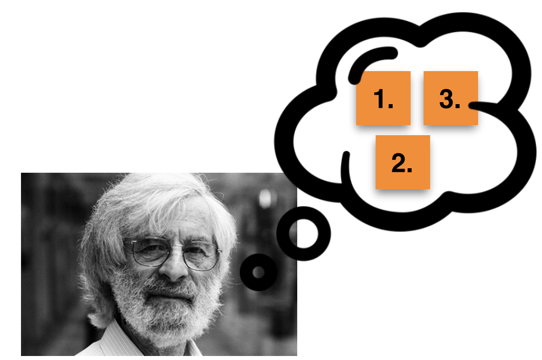
Message Drops

```
for q in qs do
  q.v <-* ?
      Just Ping:
      None
end
```

Synchronization

Extensions

Multicasts

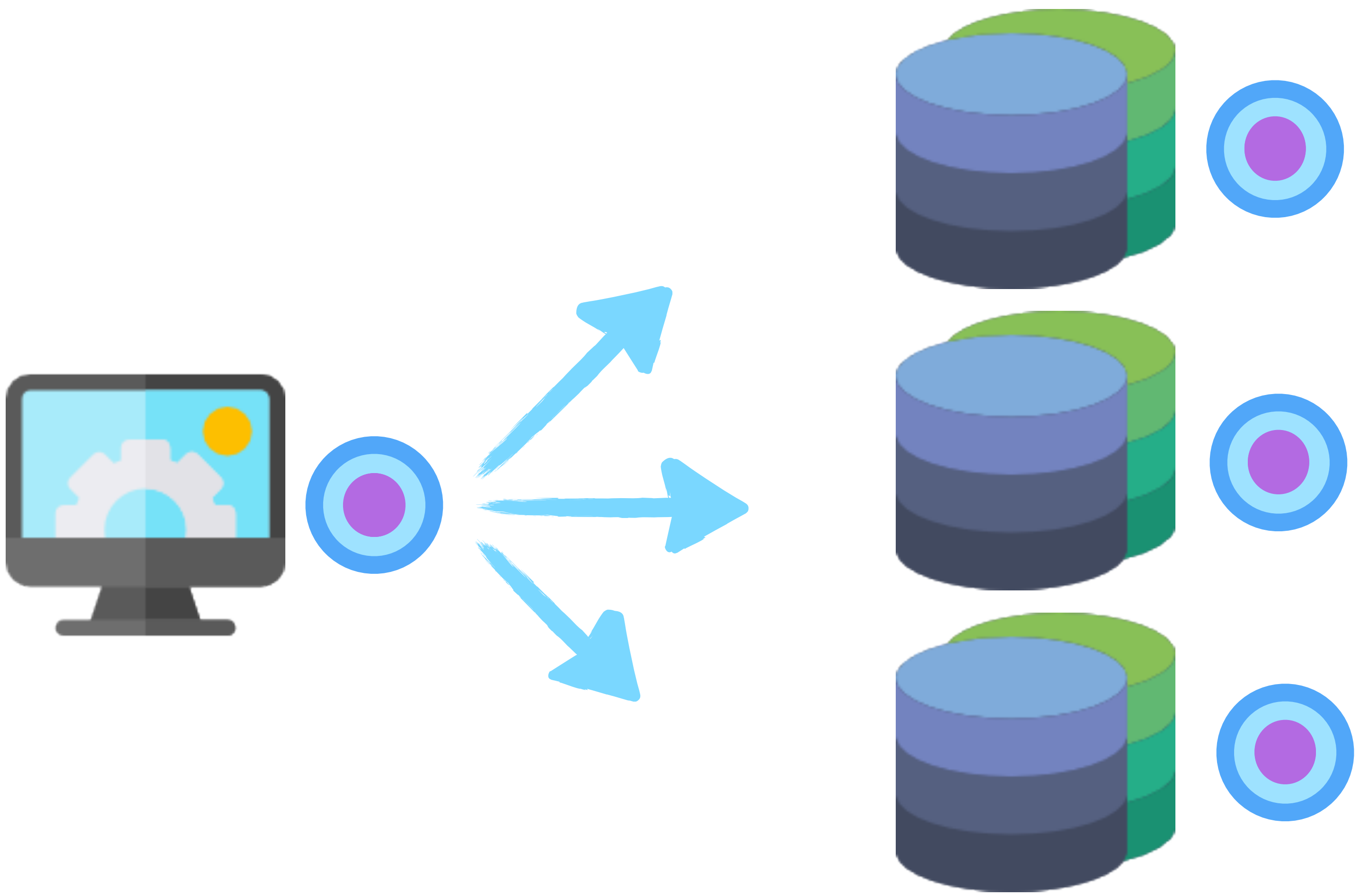


Message Drops

Rounds

Extensions

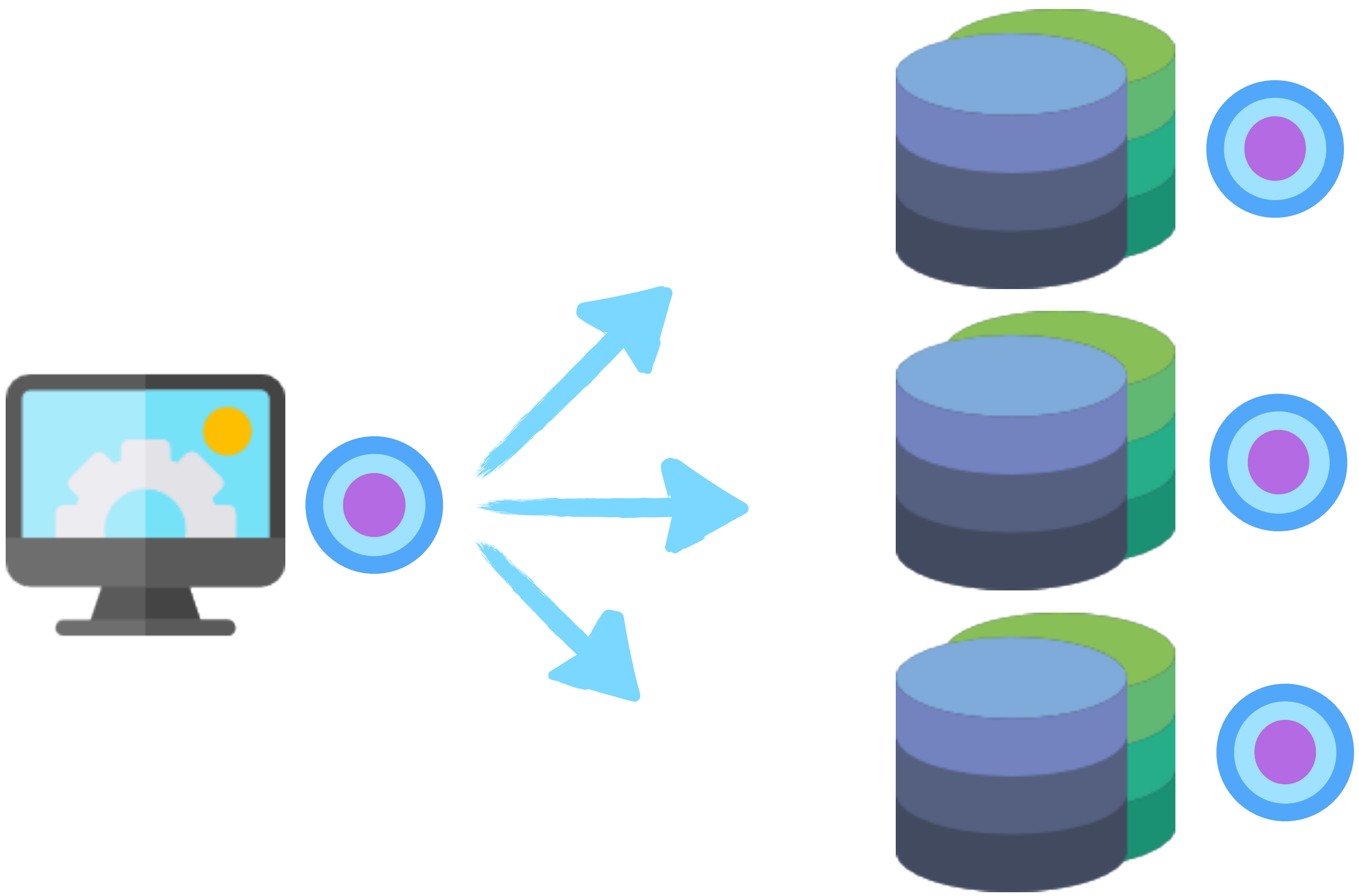
Rounds



Instead of running only *once*

Extensions

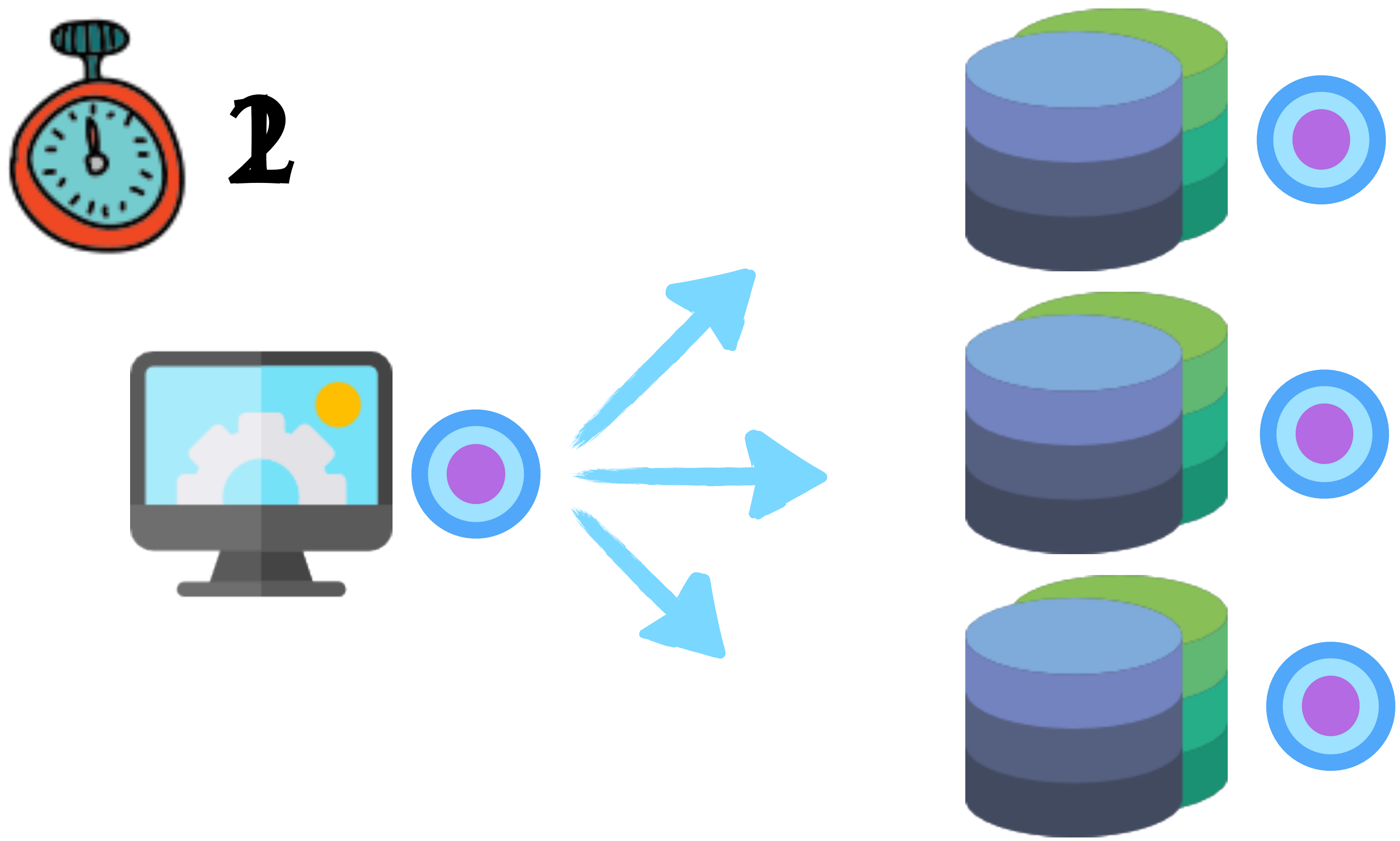
Rounds



… *repeat* protocol in *multiple rounds*

Extensions

Rounds



… *repeat* protocol in *multiple rounds*

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
for q in qs do
    send q Ping
    _<- recv q;
end
```

$\|$ $\prod_{q \in qs}$

```
w<-recv p;
send p Ack;
```

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
for q in qs do
    send q (Ping, r)
    _<- recv q;
end
```

$\|$ $\prod_{q \in qs}$

```
w<-recv p;
send p Ack;
```

… we send a round number *r*

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
for q in qs do
  send q (Ping, r)
  _<- recv q;
end
```

$\|\ \prod_{q \in qs}$

```
(w,r)<-recv p;
send p Ack;
```

… *bind* the round number at the receive

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
for q in qs do
  send q (Ping, r)
  _<- recv q;
end
```

$\|$ $\prod_{q \in qs}$

```
(w[r],r)<-recv p;
send p Ack;
```

… and turn *w* into an *array*

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
for q in qs do
    send q (Ping, r)
    _<- recv q;
end
```

$$\Big\| \quad \prod_{q \in qs}$$

```
(w[r],r)<-recv p;
send p r Ack;
```

… *reply* with a message *for round number r*

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
 for q in qs do
   send q (Ping, r)
   _<- recv q r;
end
```

$\parallel \prod_{q \in qs}$

```
(w[r],r)<-recv p;
send p r Ack;
```

… receive for round r

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

$\| \quad \prod_{q \in qs}$

```
(w[r],r)<-recv p;
send p r Ack;
```

… repeat the for-loop, *once for each round*

Extensions

Rounds

To repeat the protocol (from *Ex. 2*)

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

$$\Big\| \prod_{q \in qs}$$

```
repeat do
  (w[r],r)<-recv p;
  send p r Ack;
end
```

… and *repeat* each process q, *indefinitely*

Extensions

Rounds

Show $\forall r \in rounds, \forall q \in qs : q\,.\,w[r] = Ping$

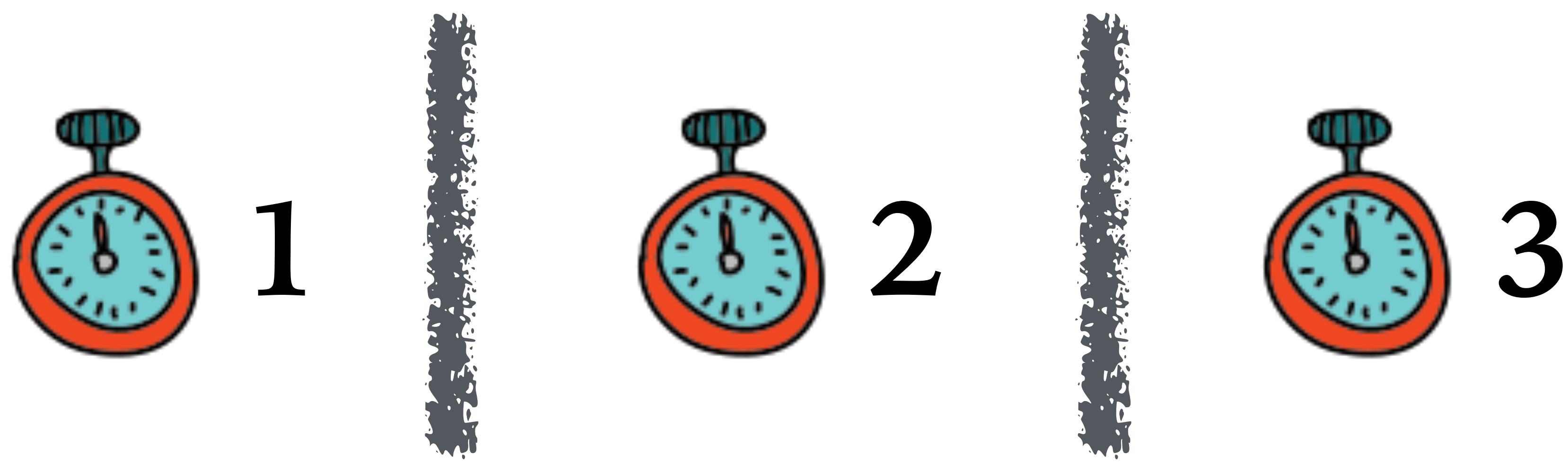
```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

||

```
Π q∈qs
repeat do
  (w[r],r)<-recv p;
  send p r Ack;
end
```

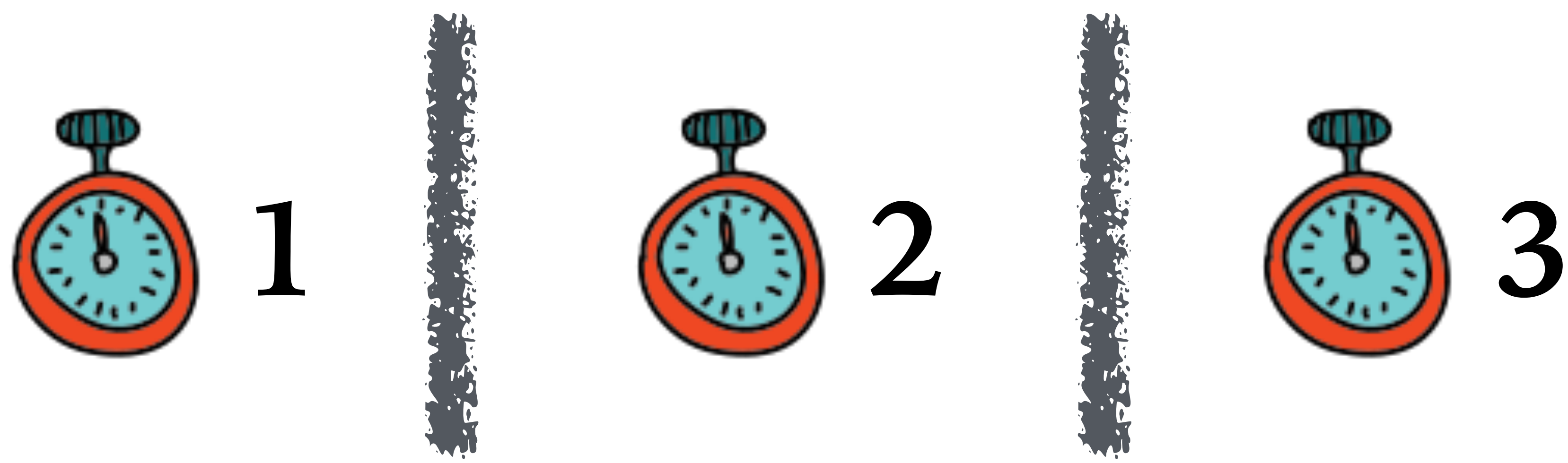
Idea: *Round Non-Interference*

Idea: *Round Non-Interference*



No *shared state* or *communication* between rounds

Idea: *Round Non-Interference*



Show $\forall r \in rounds : \varphi(r)$ by showing $\varphi(r^*)$ for an arbitrary round r^*

Extensions

Rounds

Check Round Non-interference *via Syntax*

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

$||$ $\prod_{q \in qs}$

```
repeat do
  (w[r],r)<-recv p;
  send p r Ack;
end
```

Extensions

Rounds

Check Round Non-interference *via Syntax*

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

||

```
        repeat do
Π         (w[r],r)<-recv
q∈qs      send p r Ack;
        end
```

… round id only bound *once*

Extensions

Rounds

Check Round Non-interference *via Syntax*

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

$\Vert$ $\prod_{q \in qs}$

```
repeat do
  (w[r],r)<-recv p;
  send p r Ack;
end
```

… send *only for bound round*

Extensions

Rounds

Check Round Non-interference *via Syntax*

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

$\|$ $\prod_{q \in qs}$

```
repeat do
  (w[r],r)<-recv p;
  send p r Ack;
end
```

… receive *only for bound round*

Extensions

Rounds

Check Round Non-interference *via Syntax*

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

||

```
Π q∈qs
repeat do
  (w[r],r)<-recv p;
  send p r Ack;
end
```

… array indexed *only for bound round*

Extensions

Rounds

To show $\forall r \in rounds : q.w[r] = Ping$

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

$\|$ $\prod_{q \in qs}$

```
repeat do
  (w[r],r)<-recv p;
  send p r Ack;
end
```

Extensions

Rounds

To show $\forall r \in \mathit{rounds} : q\,.\,w[r] = \mathit{Ping}$

```
for q in qs do
  send q (Ping, r*)
  _<- recv q r*;
end
```

$\| \; \prod_{q \in qs}$

```
(w[r*],r*)<-recv p;
send p r* Ack;
```

… show $\forall q \in qs : q\,.\,w[r^*] = \mathit{Ping}$ as before

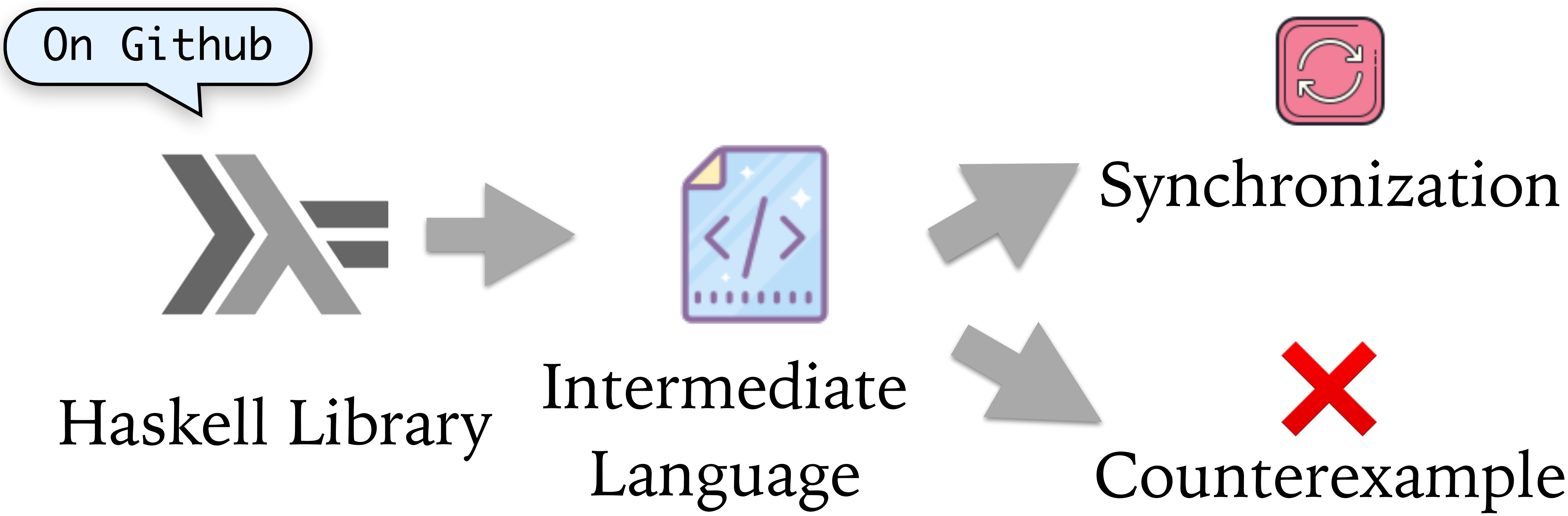
Evaluation

Evaluation

Brisk

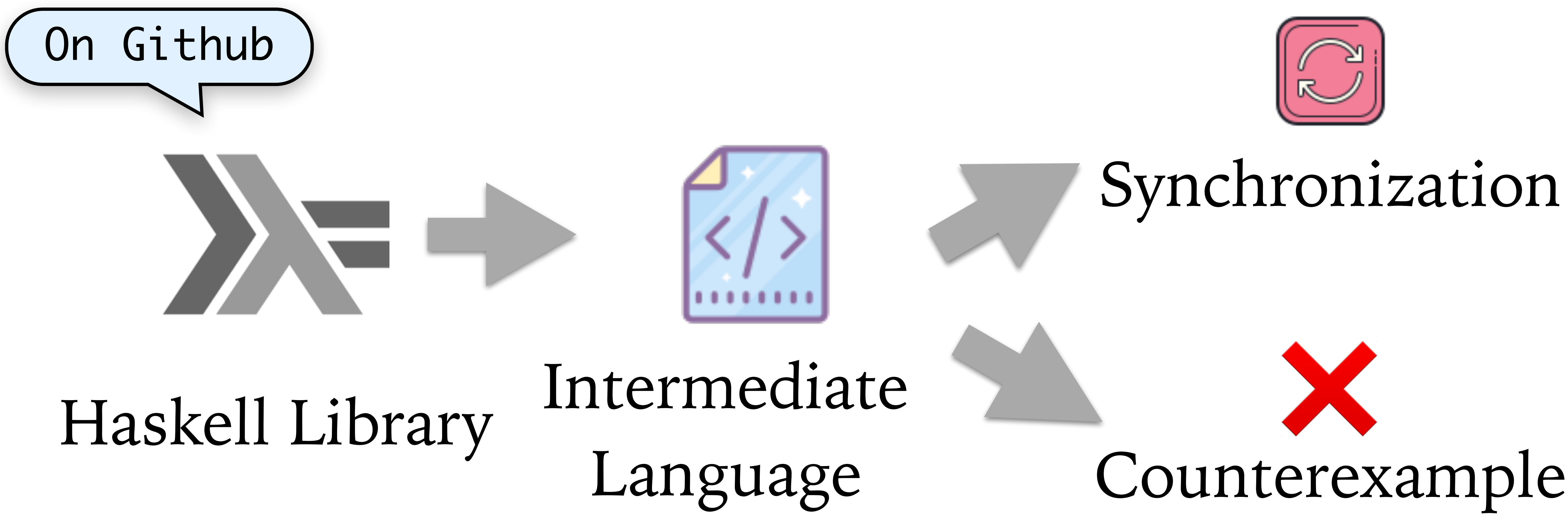
Goolong

Brisk: OOPSLA'17



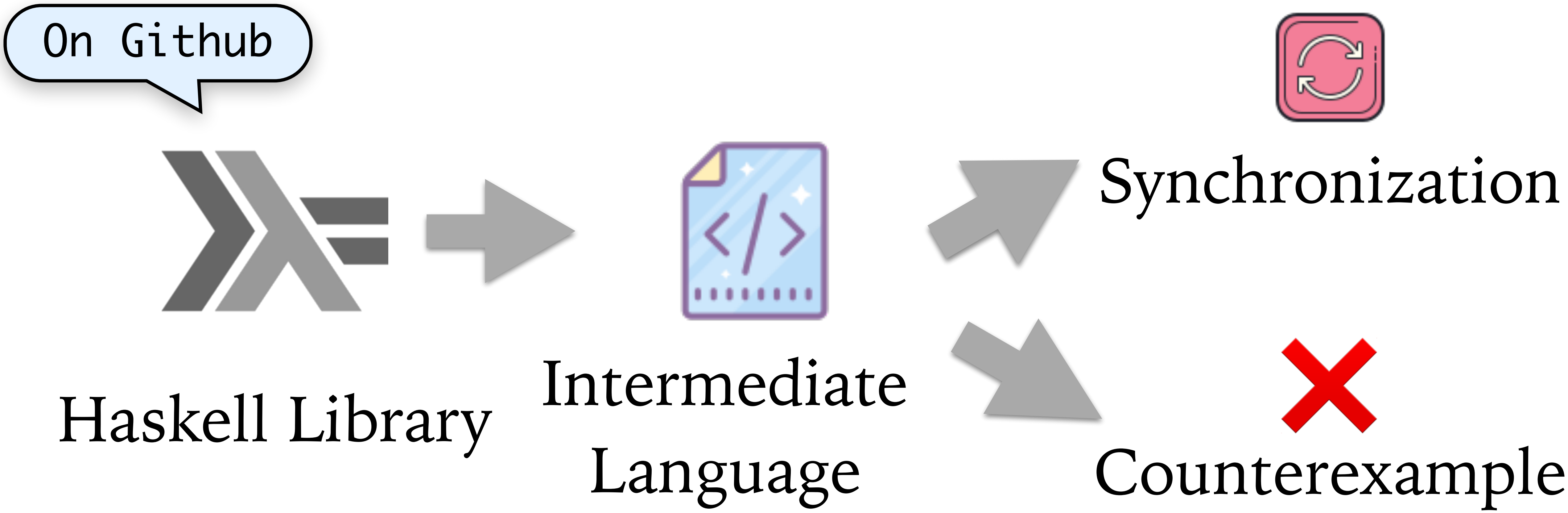
checks if a *synchronization* exists

Brisk: OOPSLA'17



…through *rewrites* implemented in Prolog

Brisk: OOPSLA'17



… no *deadlocks*, spurious sends, etc.

Brisk: OOPSLA'17

From the literature

Use interactively

Name	Time Brisk	Spin TO	#n
ConcDB	20 ms		6
DistDB	20 ms		2
Firewall	30 ms		2
LockServer	30 ms		12
MapReduce	30 ms		4
Parikh	20 ms		-
Registry	30 ms		10
TwoBuyers	20 ms		-
2PC	50 ms		6

Really fast

Brisk: OOPSLA'17

Name	Time Brisk	Spin	TO	#n
Map/Reduce	40 ms			5
Theque Filesystem	100 ms			3

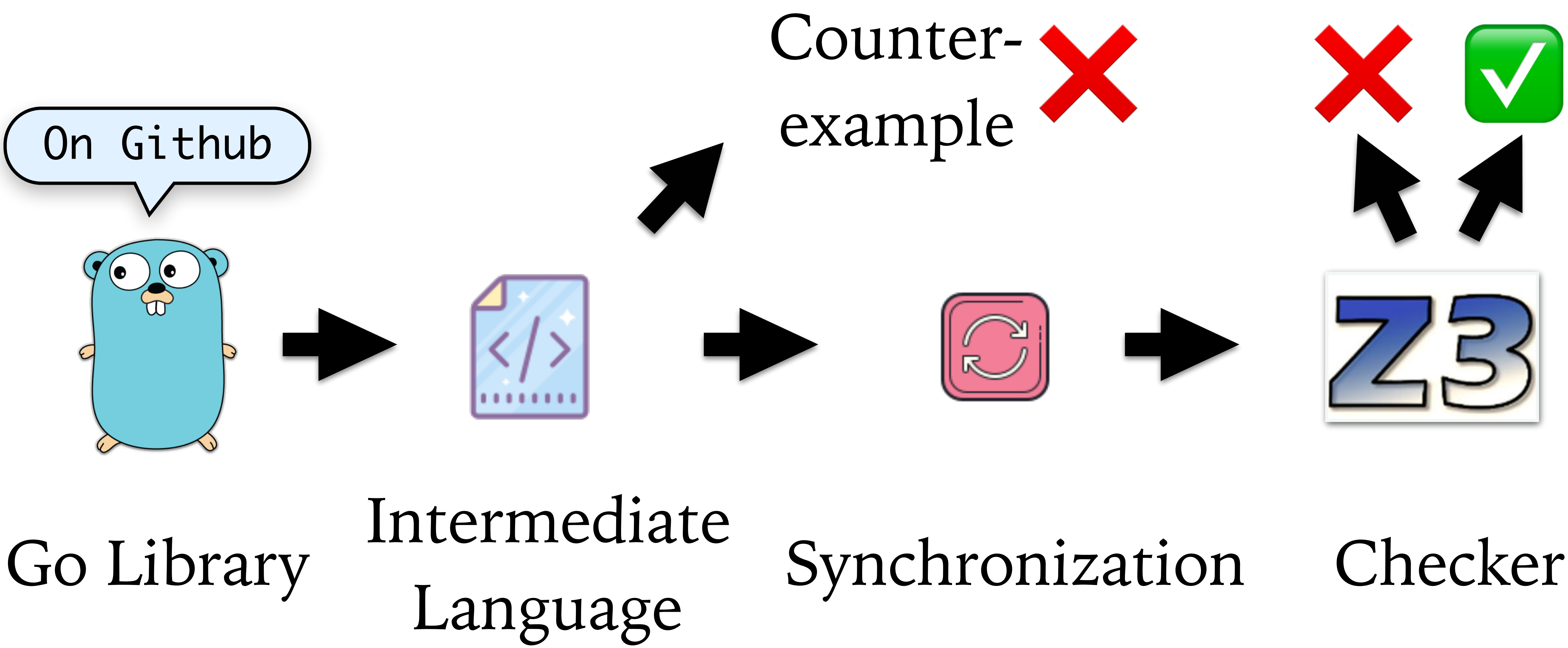
Case studies

Evaluation

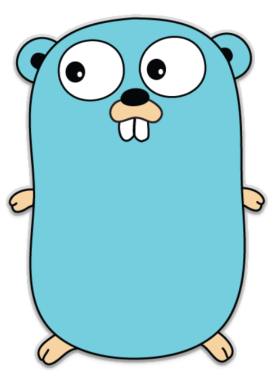
Brisk

Goolong

Goolong: POPL'19



Goolong: POPL'19



Go Library

Declare protocol variables

```
x<-NewVar()
v<-x.Get()
x.Set(v)
```

Iteration over sets/ Invariants

```
for q in qs @Inv do
```

Communication

```
send q v
w<- recv q;
```

Goolong: 2PC Phase 1

```
proposal := gochai.NewVar()
vote := gochai.NewVar()
reply := gochai.NewVar()
abort := gochai.NewVar()
committed := gochai.NewVar()
ack := gochai.NewVar()

committed.Assign(0)
abort.Assign(0)

/*{-@ invariant: forall([decl(i,int)], implies( and([ elem(i,done) ]),ref(val,i)=proposal) ) -@}*/
    for ID := range n.PeerIds {
        n.Send(ID, proposal)
    }

    for ID := range n.PeerIds {
        vote = n.RecvAll()
        if vote.Get() == 0 {
            abort.Assign(1)
        }
    }
```

Declarations

Send proposals

Receive Votes

Goolong: POPL'19

Case-Studies

2PC

Raft
Leader
Election

Single Decree
Paxos

Multi-Paxos
KV Store

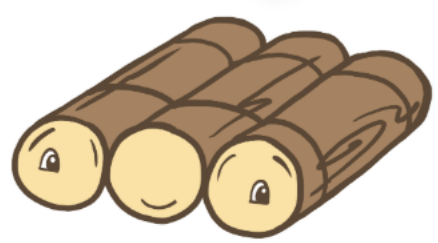
Goolong: POPL'19

Case-Studies

If committed, all nodes have same value

2PC

At most one candidate elected leader, per term



Raft Leader Election

Proposers agree on same value

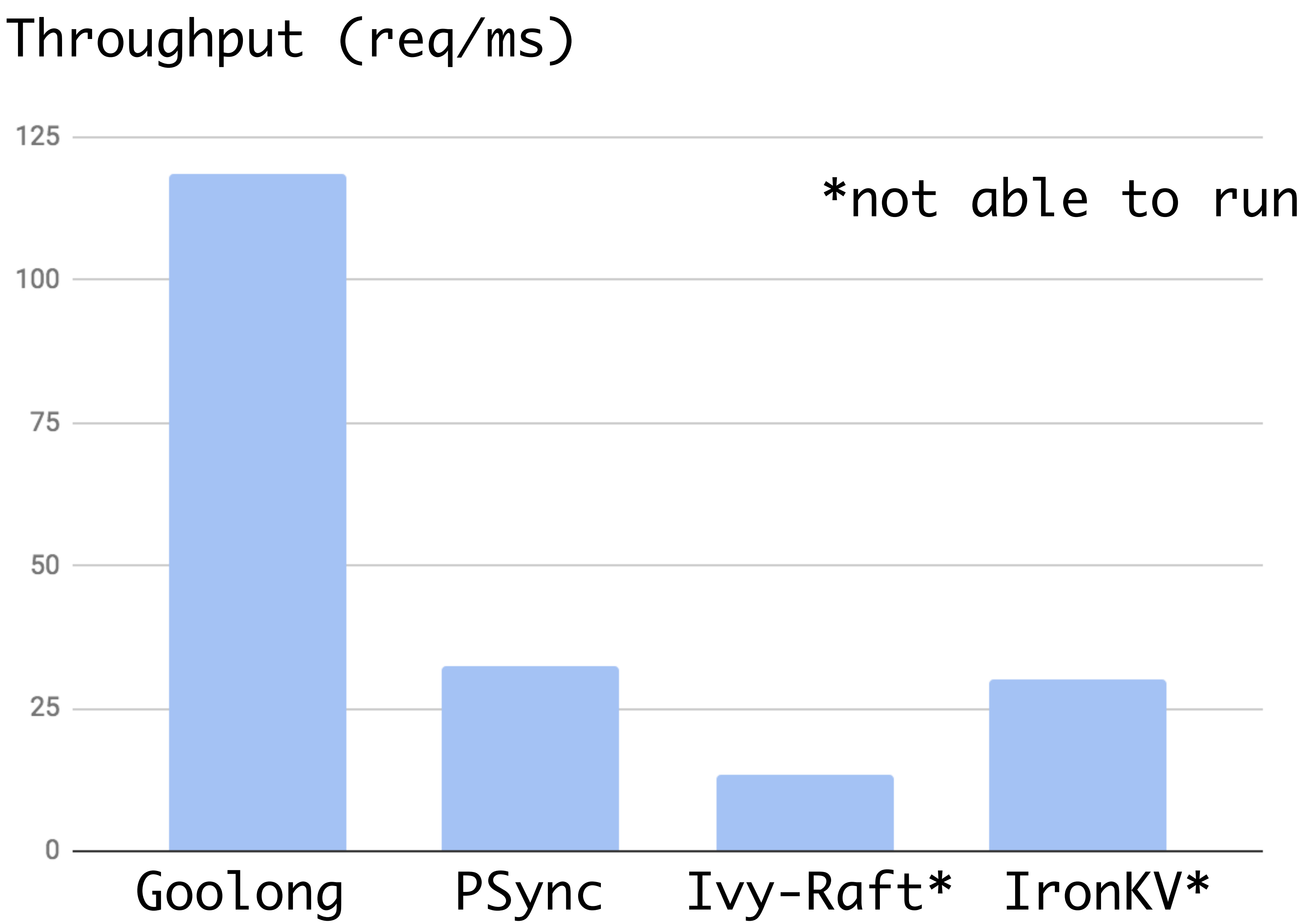
Single Decree Paxos

Proposers agree on same value, per instance

Multi-Paxos KV Store

Goolong: POPL'19

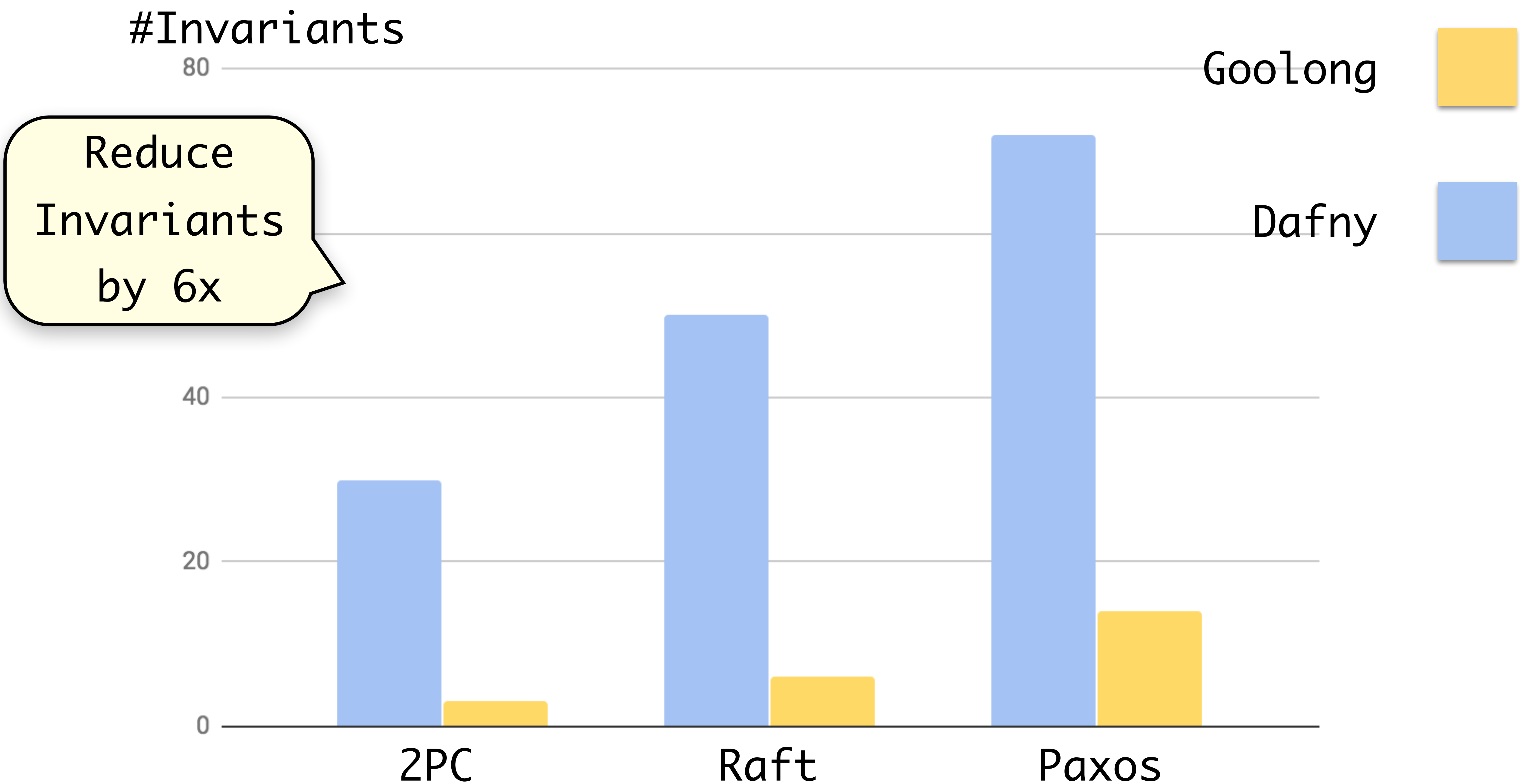
Multi-Paxos
KV Store



Goolong vs. other verified KVstores

Does Synchrony Simplify Proofs?

Goolong: POPL'19



Number of Invariants Dafny vs. Goolong

Goolong: POPL'19

Name	Time Dafny	Time Goolong
2PC	12.8s	0.04s
Raft	301.6s	0.18s
Paxos	1141.3s	1.51s
Total	1455.8s	1.73s

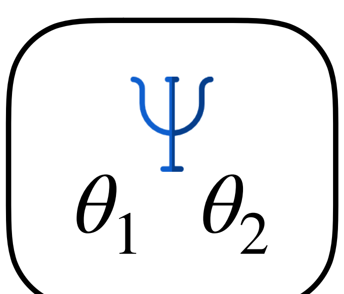
Reduce checking time by 3 orders of magnitude

Recap: Evaluation

Brisk: Synchronization in ms

Goolong: Go Library

Competitive with verified KV-stores



Reduces Invariants and Checking Time

?

Inference Rules

R-SEND

$$\frac{\Delta, \Sigma \models x = q \qquad q \text{ is a PID} \qquad m \text{ fresh} \qquad \Gamma' = \Gamma \cup \{(p, q, t, r, m)\}}{\Gamma, \Delta, \Sigma, [\mathsf{send}(x, n, t, r)]_p \rightsquigarrow \Gamma', (\Delta; [\mathsf{m} \leftarrow \mathsf{n}]_p), \Sigma, \mathsf{skip}}$$

R-RECV

$$\frac{\Delta, \Sigma \models x = p \qquad p \text{ is a PID} \qquad (p, q, t, r, m) \in \Gamma \qquad \Gamma' = \Gamma - \{(p, q, t, r, m)\}}{\Gamma, \Delta, \Sigma, [\mathsf{y} \leftarrow \mathsf{recv}(\mathsf{t}, \mathsf{x}, \mathsf{r})]_q \rightsquigarrow \Gamma', (\Delta; [y \leftarrow p.m]_q), \Sigma, \mathsf{skip}}$$

R-RECVTO

$$\frac{\Gamma, \Delta, \Sigma, [\mathsf{y} \leftarrow \mathsf{recv}(\mathsf{t}, \mathsf{x}, \mathsf{r})]_q \rightsquigarrow \Gamma', (\Delta; [y \leftarrow p.m]_q), \Sigma, \mathsf{skip}}{\Gamma, \Delta, \Sigma, [\mathsf{y} \leftarrow \mathsf{recvTO}(\mathsf{t}, \mathsf{x}, \mathsf{r})]_q \rightsquigarrow \Gamma', (\Delta; [y \leftarrow \mathsf{Just}\ p.m]_q \oplus [y \leftarrow \mathsf{None}]_q), \Sigma, \mathsf{skip}}$$

R-CHOICE

$$\frac{\Gamma, \Delta, \Sigma, A \rightsquigarrow \Gamma, (\Delta; \Delta_A), \Sigma, \mathsf{skip} \qquad \Gamma, \Delta, \Sigma, B \rightsquigarrow \Gamma, (\Delta; \Delta_B), \Sigma, \mathsf{skip}}{\Gamma, \Delta, \Sigma, A \oplus B \rightsquigarrow \Gamma, (\Delta; \Delta_A \oplus \Delta_B), \Sigma, \mathsf{skip}}$$

R-FALSE

$$\frac{\Delta \models \mathit{false}}{\Gamma, \Delta, \Sigma, A \rightsquigarrow \Gamma, \Delta, \Sigma, \mathsf{skip}}$$

R-CONTEXT

$$\frac{\Gamma, \Delta, \Sigma, A \rightsquigarrow \Gamma', \Delta', \Sigma', A'}{\Gamma, \Delta, \Sigma, A \circ B \rightsquigarrow \Gamma', \Delta', \Sigma', A' \circ B}$$

R-LOOP

$$\frac{\begin{array}{l} (1)\ u, x \text{ fresh} \\ (2)\ \Gamma_0 \triangleq \Gamma \cup \{\mathsf{unfold}(u, x, ps)\} \text{ and } \Delta_0 \triangleq \mathsf{assume}(I_C) \\ (3)\ \Delta, \Sigma \models I_C \text{ and } (\Delta_0; \langle \Delta^u \rangle), \Sigma \models I_C \\ \Gamma_0, \Delta_0, \Sigma, [A]_u \parallel B[x/p] \rightsquigarrow \Gamma, (\Delta_0; \Delta^u), \Sigma, \mathsf{skip} \end{array}}{\begin{array}{l} \Gamma, \Delta, \Sigma, \prod (p \in ps).\ [A]_p \parallel [\mathsf{for}\ p \in ps\ \{I_S\}\ \mathsf{do}\ B\ \mathsf{end}]_q \rightsquigarrow \\ \Gamma, [\mathsf{for}\ p \in ps\ \mathsf{do}\ \langle I_S \triangleright \Delta^u[p/u] \rangle\ \mathsf{end}], \Sigma, \mathsf{skip} \end{array}}$$

R-SEND-UNFOLD

$$\frac{\Gamma \vdash \mathsf{unfold}(u, x, ps) \qquad \Gamma' \triangleq \Gamma - \{\mathsf{unfold}(u, x, ps)\}}{\Gamma, \Delta, \Sigma, \mathsf{send}(t, x, n) \rightsquigarrow \Gamma', (\Delta\ ;\ \mathsf{assume}(x = u)), \Sigma, \mathsf{send}(t, x, n)}$$

R-RECV-UNFOLD

$$\frac{\Gamma \vdash \mathsf{unfold}(u, x, ps) \qquad \Gamma' \triangleq \Gamma - \{\mathsf{unfold}(u, x, ps)\}}{\Gamma, \Delta, \Sigma, \mathsf{y} \leftarrow \mathsf{recv}(\mathsf{ps}, \mathsf{t}) \rightsquigarrow \Gamma', (\Delta\ ;\ \mathsf{x} \leftarrow \mathsf{pick}(\mathsf{ps})), \Sigma', \mathsf{y} \leftarrow \mathsf{recv}(\mathsf{u}, \mathsf{t})}$$

R-FOCUS

$$\frac{\begin{array}{l} (1)\ u \text{ fresh} \\ (2)\ \Gamma_0 \triangleq \Gamma \cup \{\mathsf{unfold}(u, \_, ps)\} \text{ and } \Delta_0 \triangleq \mathsf{assume}(I_C) \\ (3)\ \Sigma' = (\Sigma \parallel \prod (p \in ps).\Delta^u[p/u]) \\ (4)\ \Delta, \Sigma' \models I_C \text{ and } (\Delta_0; \Delta^u), \Sigma' \models I_C \\ \Gamma_0, \Delta_0, \mathsf{skip}, [A]_u \parallel \prod (q \in qs).\ [B]_q \rightsquigarrow \Gamma_0, (\Delta_0; \Delta^u), \mathsf{skip}, \mathsf{skip} \end{array}}{\Gamma, \Delta, \Sigma, \prod (p \in ps).\ [A]_p \parallel \prod (q \in qs).\ [\mathsf{foreach}\ ps\ \mathsf{do}\ B]_q \rightsquigarrow \Gamma, \Delta, \Sigma', \mathsf{skip}}$$

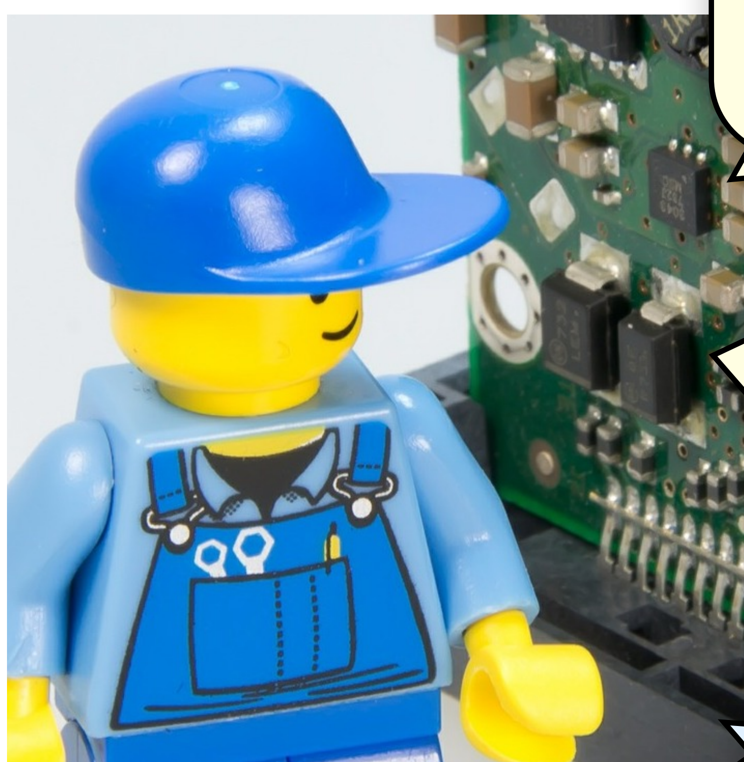
Limitations

Structured Loops

Symmetric Non-Determinism

Round Non-Interference

Limitations: Structured Loops

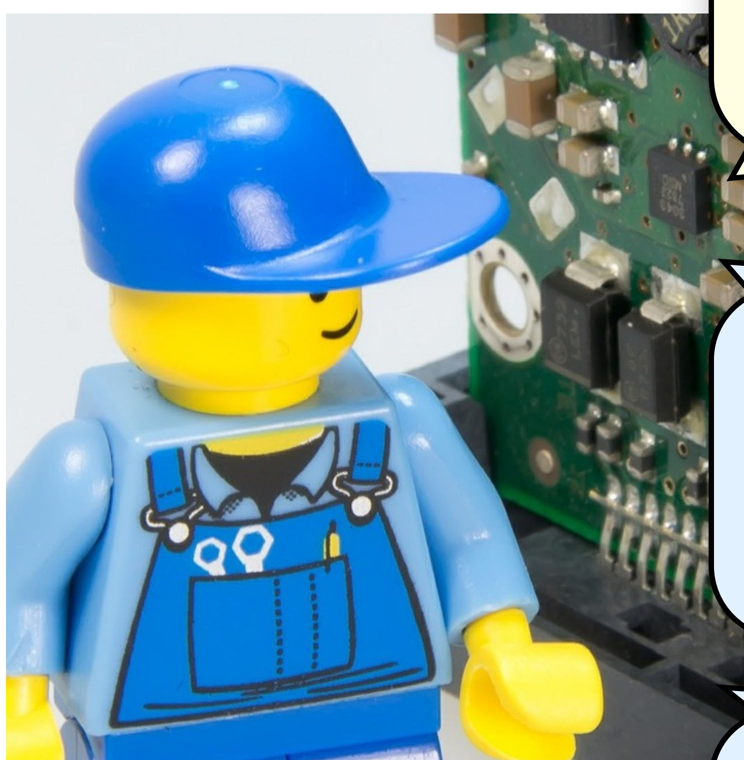


Loops over <u>sets of processes</u>

In each iteration, talk to a <u>single process</u>, only

<u>Easy:</u> transform to broadcast/gather

Limitations: Structured Loops



Loops over <u>sets of processes</u>

<u>Hard:</u> arbitrary loop carried state

Encode as rounds

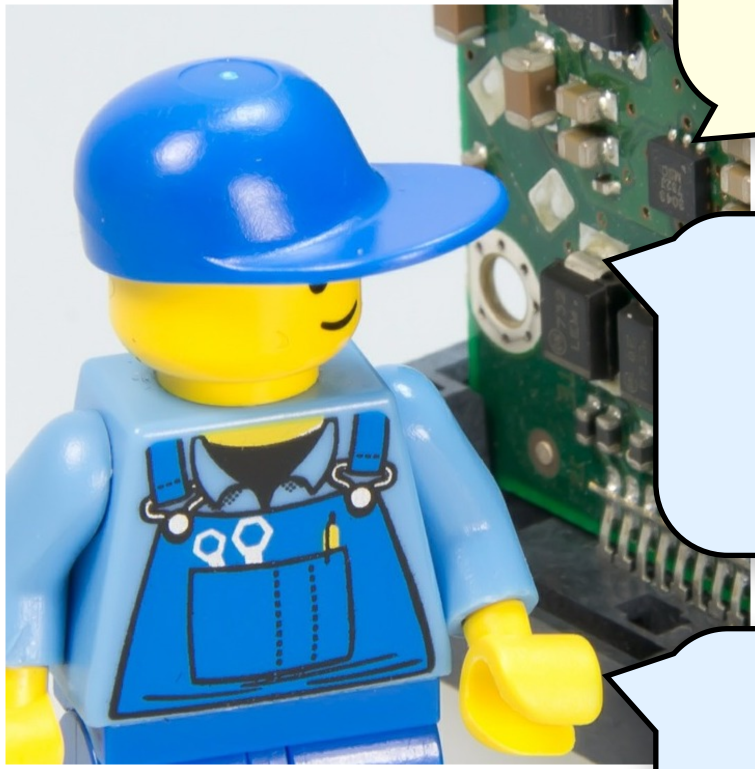
Limitations

Structured Loops

Symmetric Non-Determinism

Round Non-Interference

Limitations:
Symmetric Non-Determinism

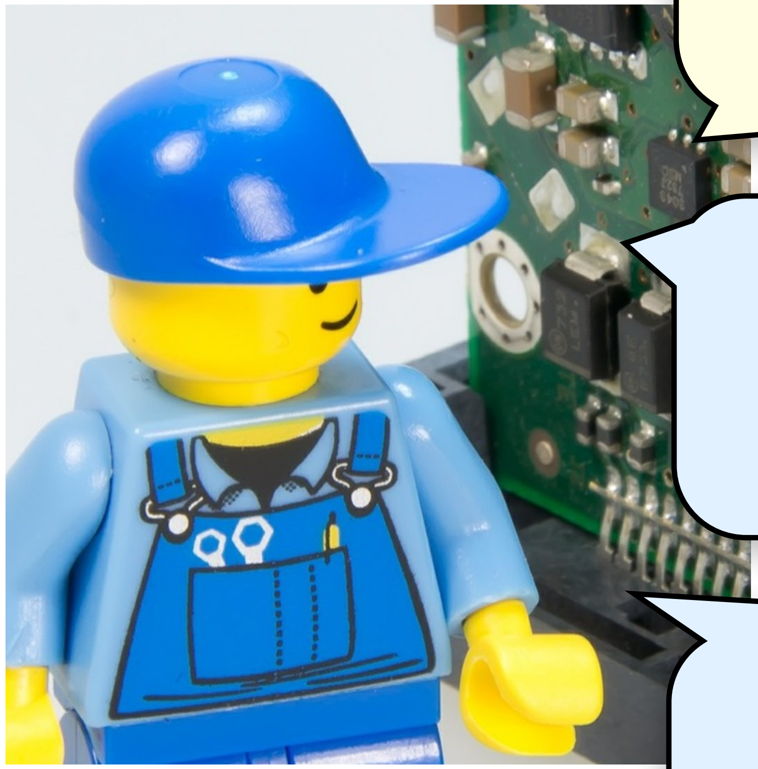


Sends have matching receives!

Easy: check with static analysis

Easy: remove "deadlocks"/ spurious sends

Limitations:
Symmetric Non-Determinism



Sends have matching receives!

Hard: Topologies e.g., Chord, Stoica et al., SIGCOMM ’01.

More inspiration from shape analysis!

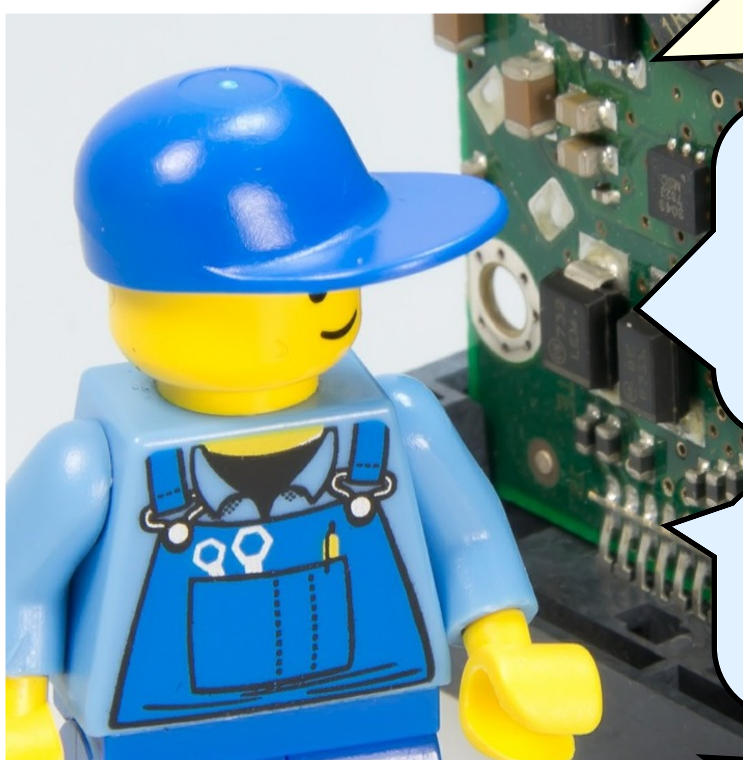
Limitations

Structured Loops

Symmetric Non-Determinism

Round Non-Interference

Limitations:
Round Non-Interference



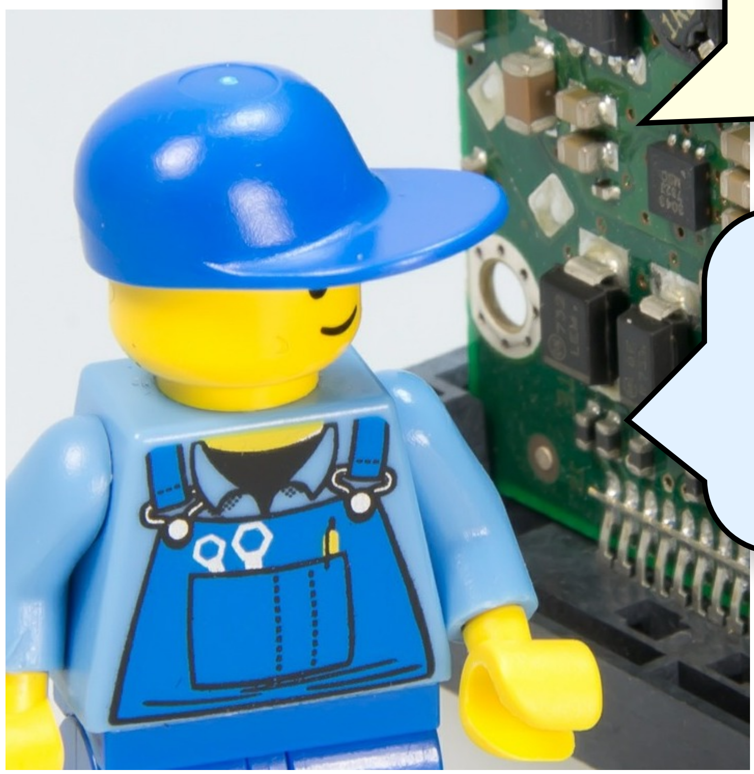
Rounds don't share messages/state

Enough for Multi-Paxos, Raft-Leader Election, Zab

Prevents Optimization such as stable leader

Algorithms like Stoppable Paxos, 2008

Limitations:
Round Non-Interference

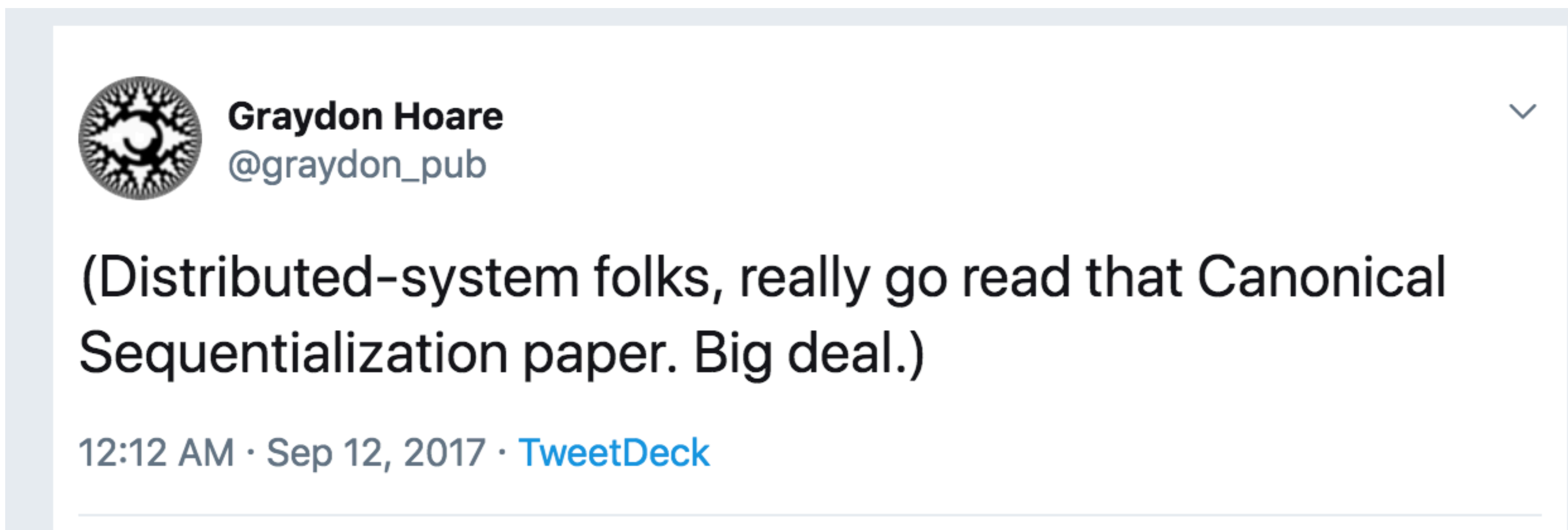


Rounds don't share messages/ state

Generalize: HO-model, communication closed layers?

Future Work

Language Restrictions


Graydon Hoare
@graydon_pub

(Distributed-system folks, really go read that Canonical Sequentialization paper. Big deal.)

12:12 AM · Sep 12, 2017 · TweetDeck

Future Work

Language Restrictions

Graydon Hoare
@graydon_pub

"Figure out how programmers are currently reasoning, formalize as language restricting validity to that model, profit!" is A+ research plan

Graydon Hoare @graydon_pub · Sep 12, 2017
Yeah it's normal in many contexts, I'm just applauding a particularly pleasing example of it (that Canonical Sequentializatoon paper)

2

Goolong: POPL'19

Invariants

Name	#Inv Async	Time Async/ Dafny	#Inv Sync	Time Sync
2PC	30	12.8s	3	0.04s
Raft	50	301.6s	6	0.18s
Paxos	72	1141.3s	14	1.51s
Total	152	1455.8s	23	1.73s

Reduce Invariants by 6x

Reduce checking time by 3 orders of magnitude

Goolong: POPL'19

Case-Studies

1.5-3x slowdown over unverified KV store

System	Throughput (req/ms)
Goolong	118.5
PSync	32.4
Ivy-Raft*	13.5
IronKV*	30

*not able to run

Multi-Paxos KV Store

Brisk: OOPSLA'17

Two Phase Commit in Brisk

```
coord :: Transaction -> Int -> SymSet ProcessId -> Process ()
coord transaction n nodes = do
                    fold query () nodes
                    n_ <- fold countVotes 0 nodes
                    if n == n_ then
                        forEach nodes commit ()
                    else
                         forEach nodes abort ()
                    forEach nodes expect :: Ack
   where
        query () pid     = do { me <- myPid; send pid (me, transaction) }
        countVotes init nodes = do
              msg <- expect :: Vote
              case msg of
                 Accept _ -> return (x + 1)
                 Reject   -> return x

acceptor :: Process ()
acceptor = do
      me <- myPid
      (who, transaction)   <- expect :: (ProcessId, Transaction)
      vote <- chooseVote transaction
      send who vote
```

Leftovers

Synchronous Proofs

```
for p in dbs do
  p.id<-c;
  p.val<-c.val;
for p in dbs do
  c.abort<-False;
  vote <-* ?
       Commit : Abort
  c.msg<-p.vote;
  if msg == Abort
     abort <- True
```

Synchronous Proofs

```
for p in dbs do @Inv1
  p.id<-c;
  p.val<-c.val;
for p in dbs do @Inv2
  c.abort<-False;
  vote <-* ?
      Commit : Abort
  c.msg<-p.vote;
  if msg == Abort
    abort <- True
```

Synchronous Proofs

```
for p in dbs do @Inv1
  p.id<-c;
  p.val<-c.val;
for p in dbs do @Inv2
  c.abort<-False;
  vote <-* ?
       Commit : Abort
  c.msg<-p.vote;
  if msg == Abort
    abort <- True
```

@Inv1=

∀p∊dbs:p∊done=>p.val=c.val

@Inv2= true

Synchronous Proofs

```
c.dec <- c.abort ?
   Abort : Commit
```

```
for p in dbs do @Inv3
```

```
p.dec<-c.dec;
```

```
for p in dbs do @Inv4
```

```
if msg == Commit
  p.value <- p.val;
```

```
_<-Ack;
```

Synchronous Proofs

```
c.dec <- c.abort ?
  Abort : Commit
for p in dbs do @Inv3
    p.dec<-c.dec;
for p in dbs do @Inv4
    if msg == Commit
      p.value <- p.val;
    _<-Ack;
```

No case-splits!

@Inv3 = true

No network state!

@Inv4=

∀p∈dbs: (p∈done ∧ c.dec= Commit)
=> p.value=c.val

Outline

Key Idea: Pretend Synchrony

Implementation

From 2PC to Paxos

Evaluation

Limitations

Implementation

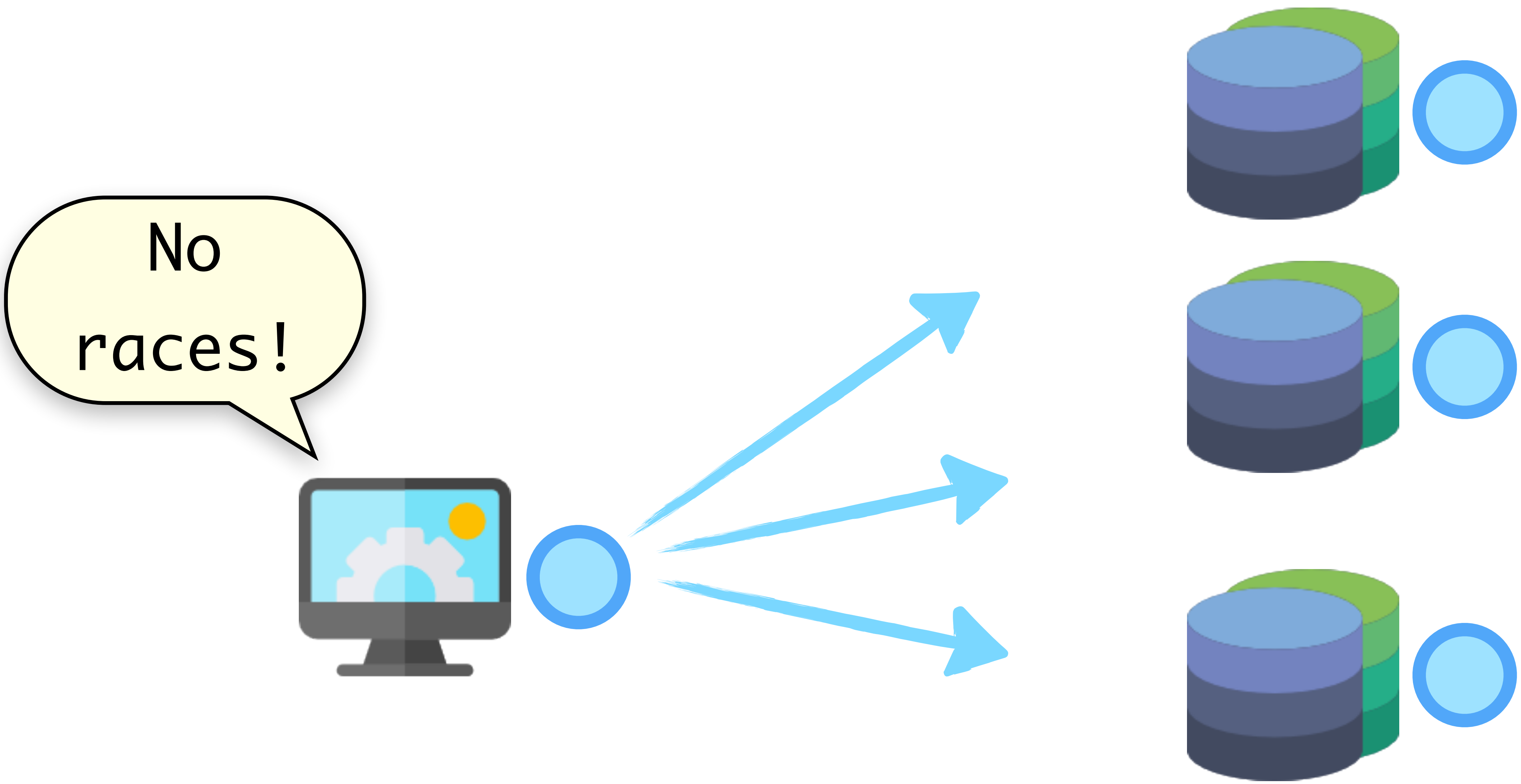
Implementation

Key ingredients

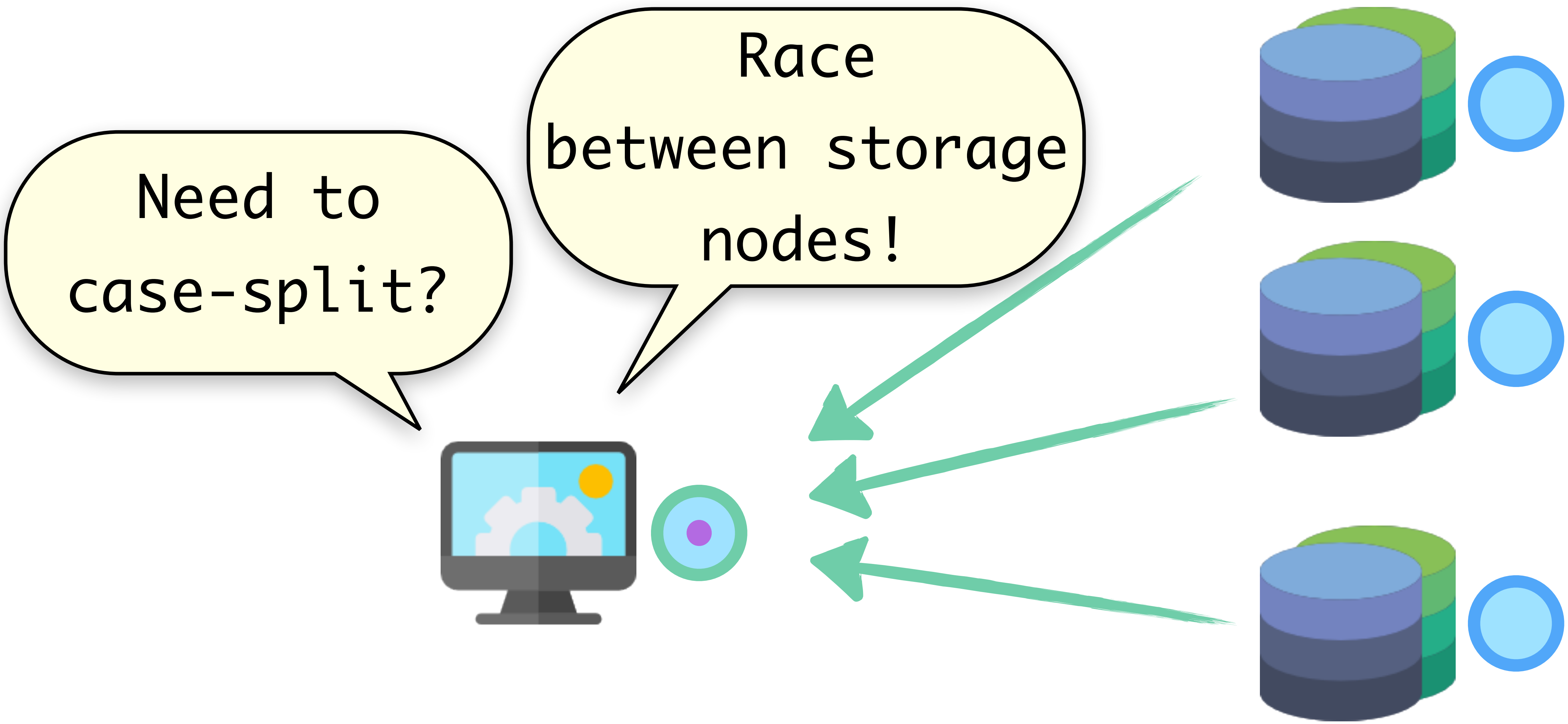
Symmetric Nondeterminism

Synchronize by Rewriting

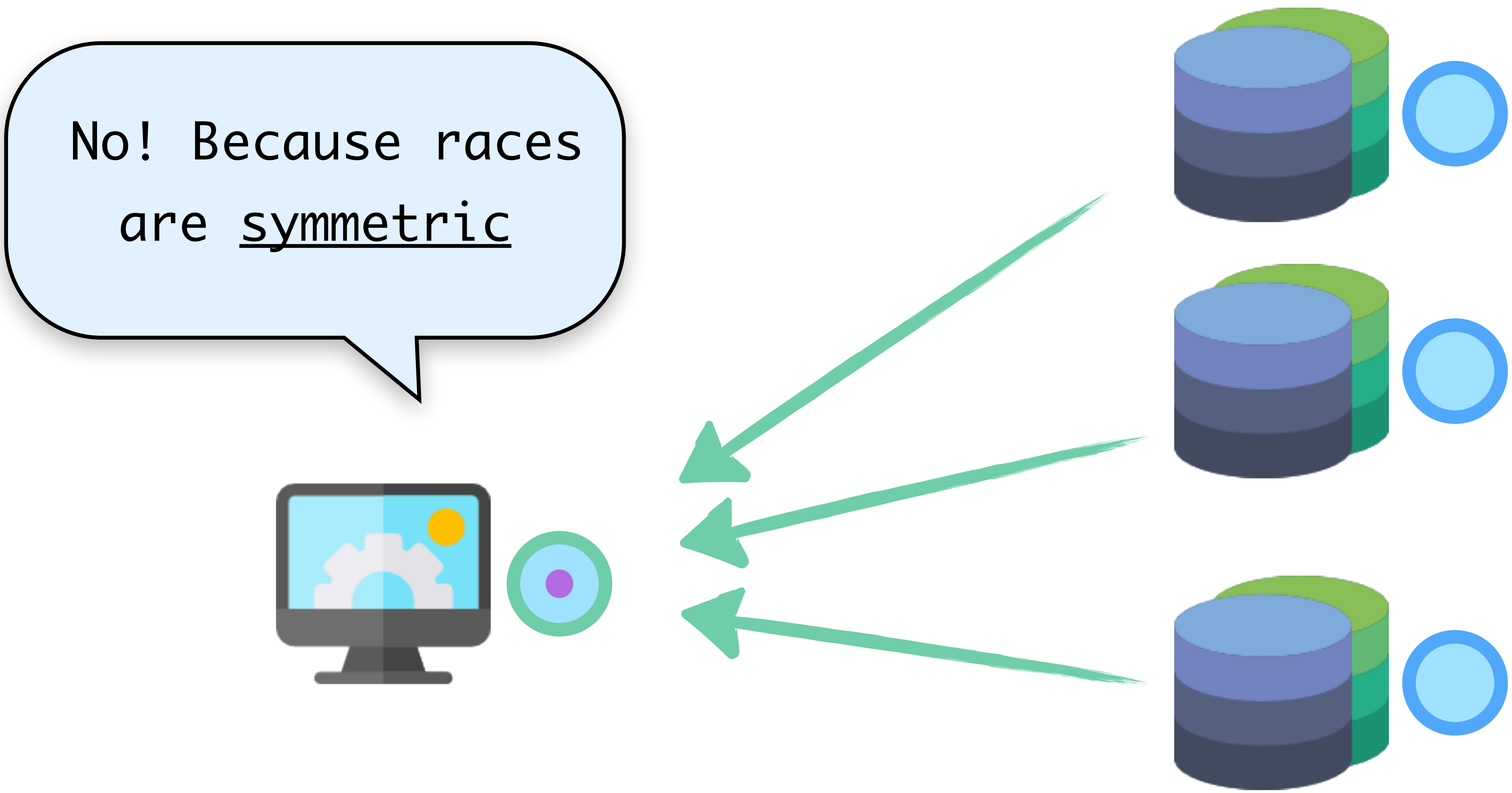
Two-phase Commit: Phase 1



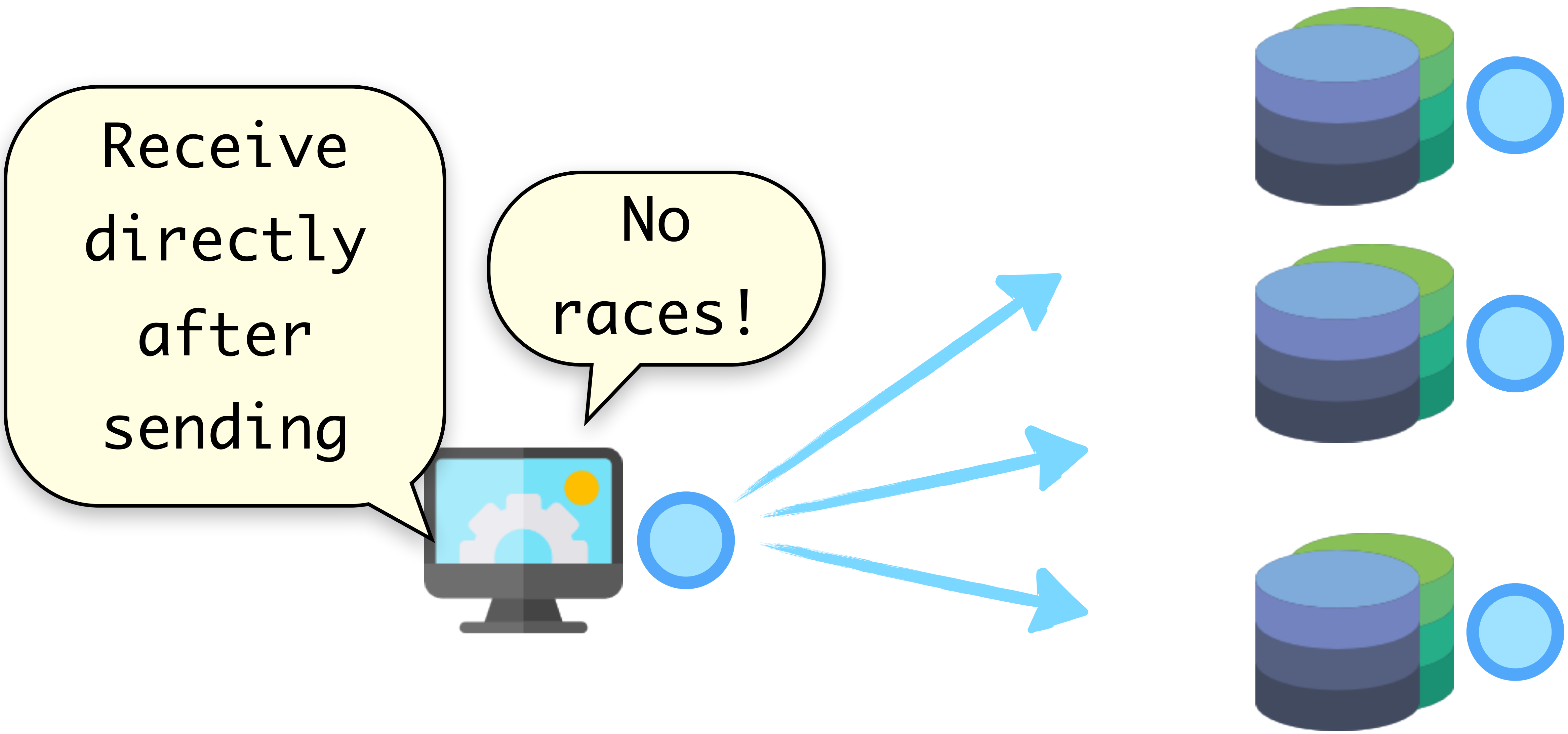
Two-phase Commit: Phase 1



Two-phase Commit: Phase 1

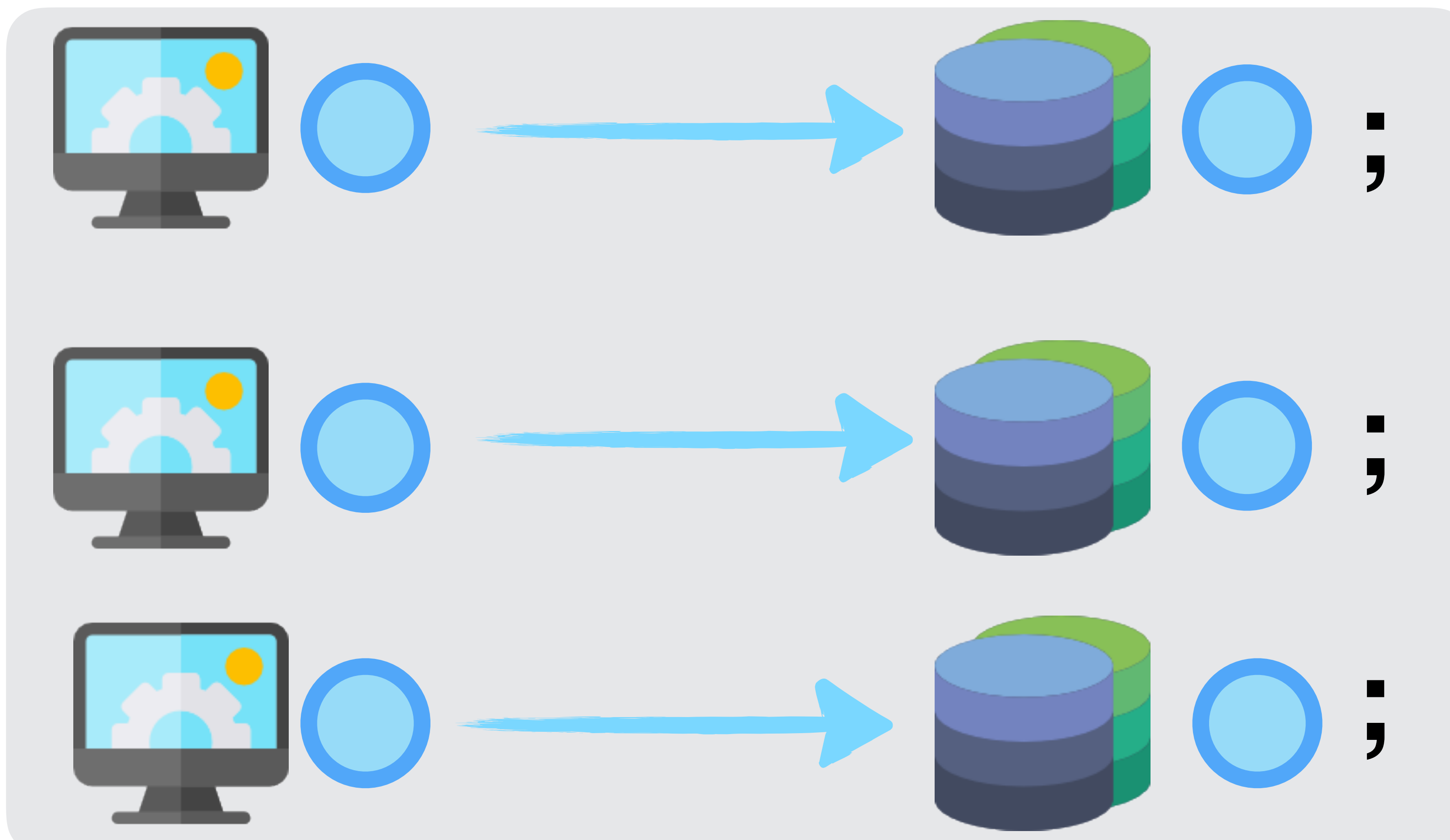


Two-phase Commit: Phase 1



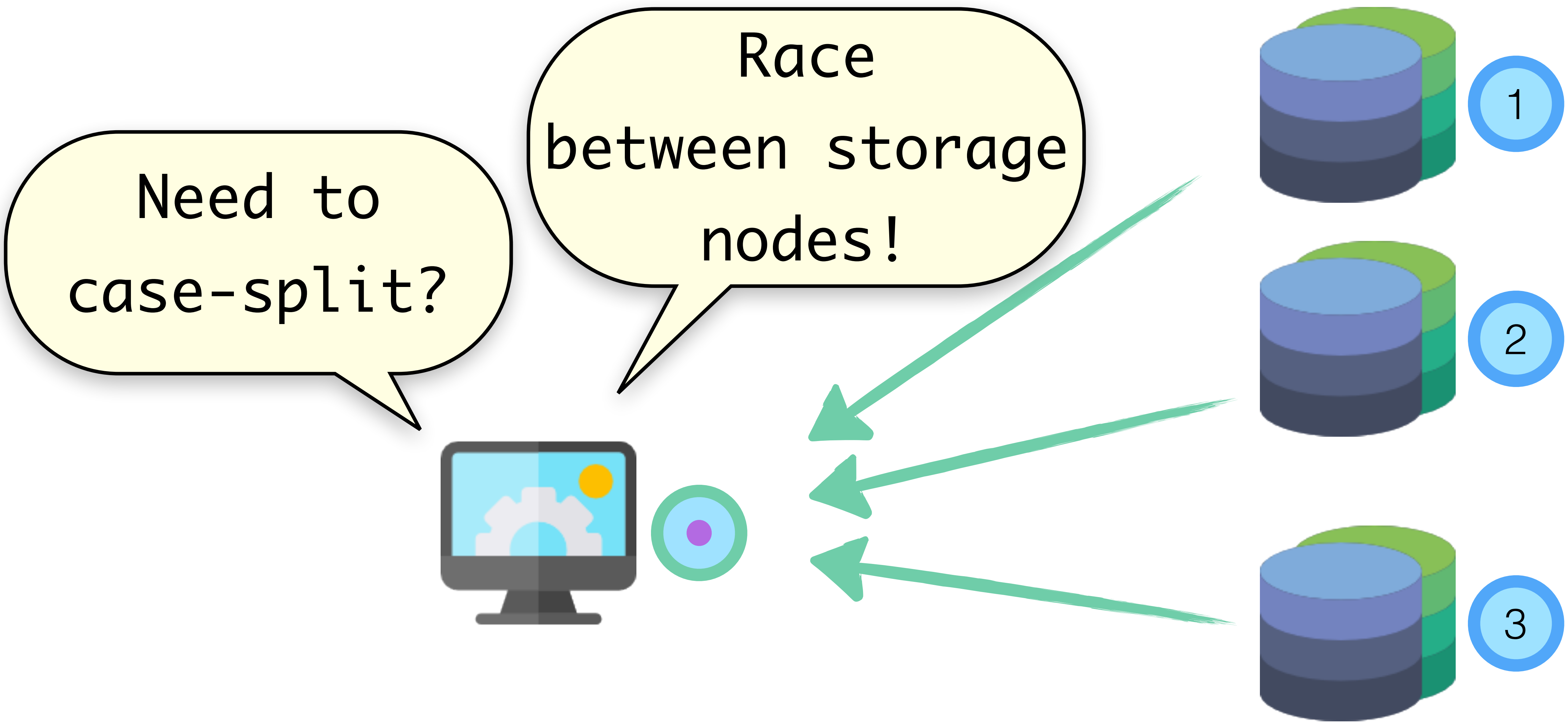
Theory of Movers [Lipton 1975]

Two-phase Commit: Phase 1

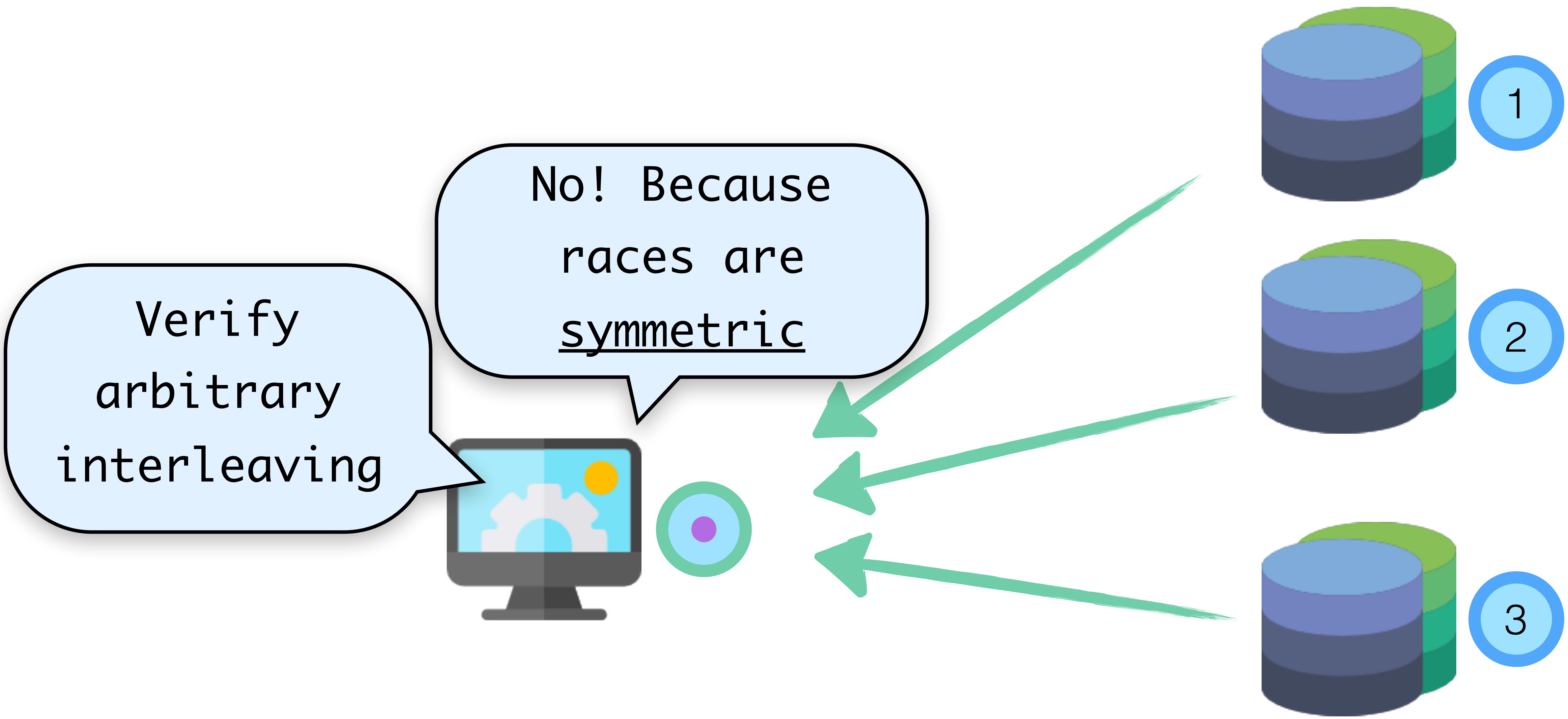


Theory of Movers [Lipton 1975]

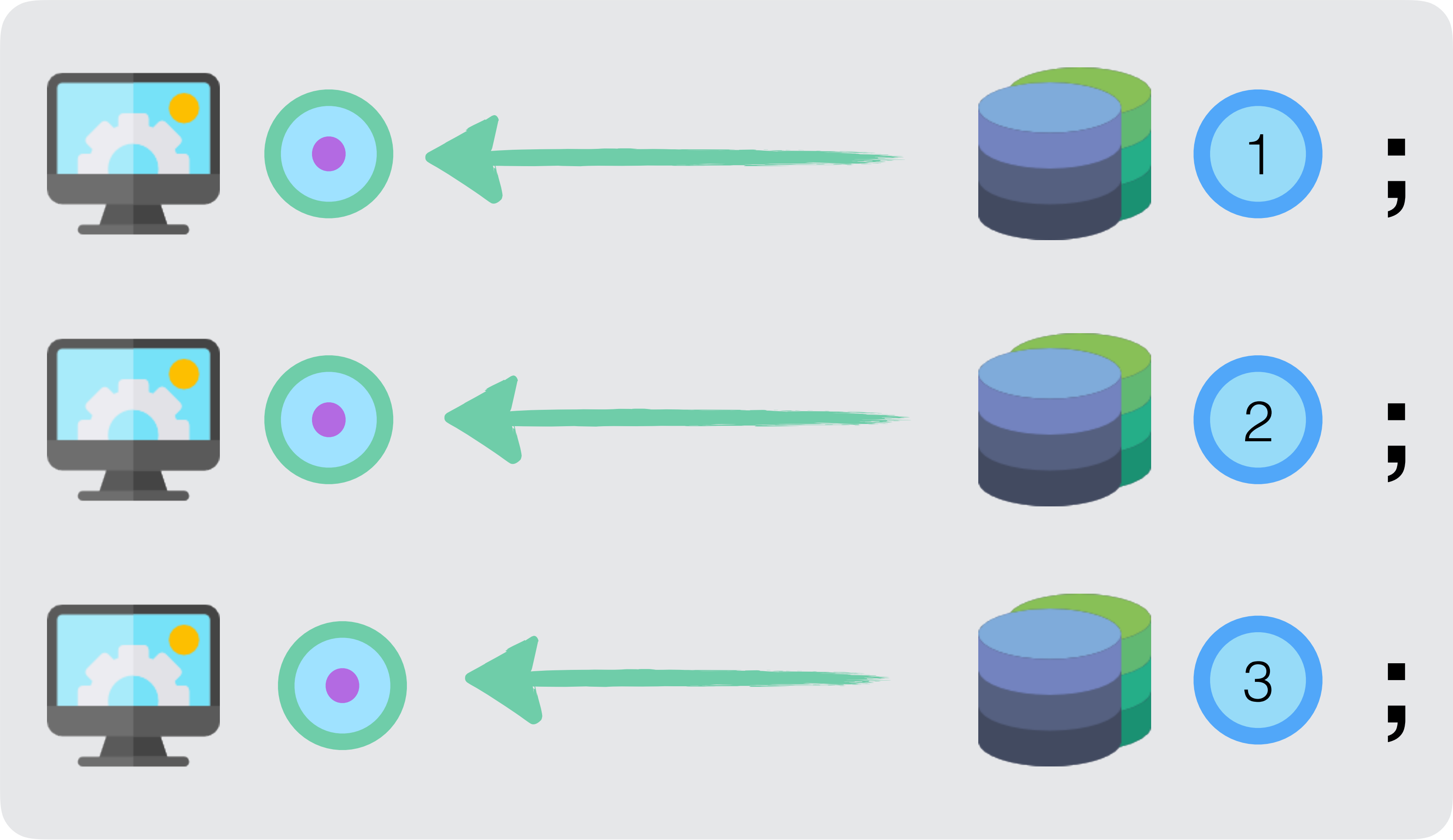
Two-phase Commit: Phase 1



Two-phase Commit: Phase 1



Two-phase Commit: Phase 1



Implementation

Key ingredients

Symmetric Nondeterminism

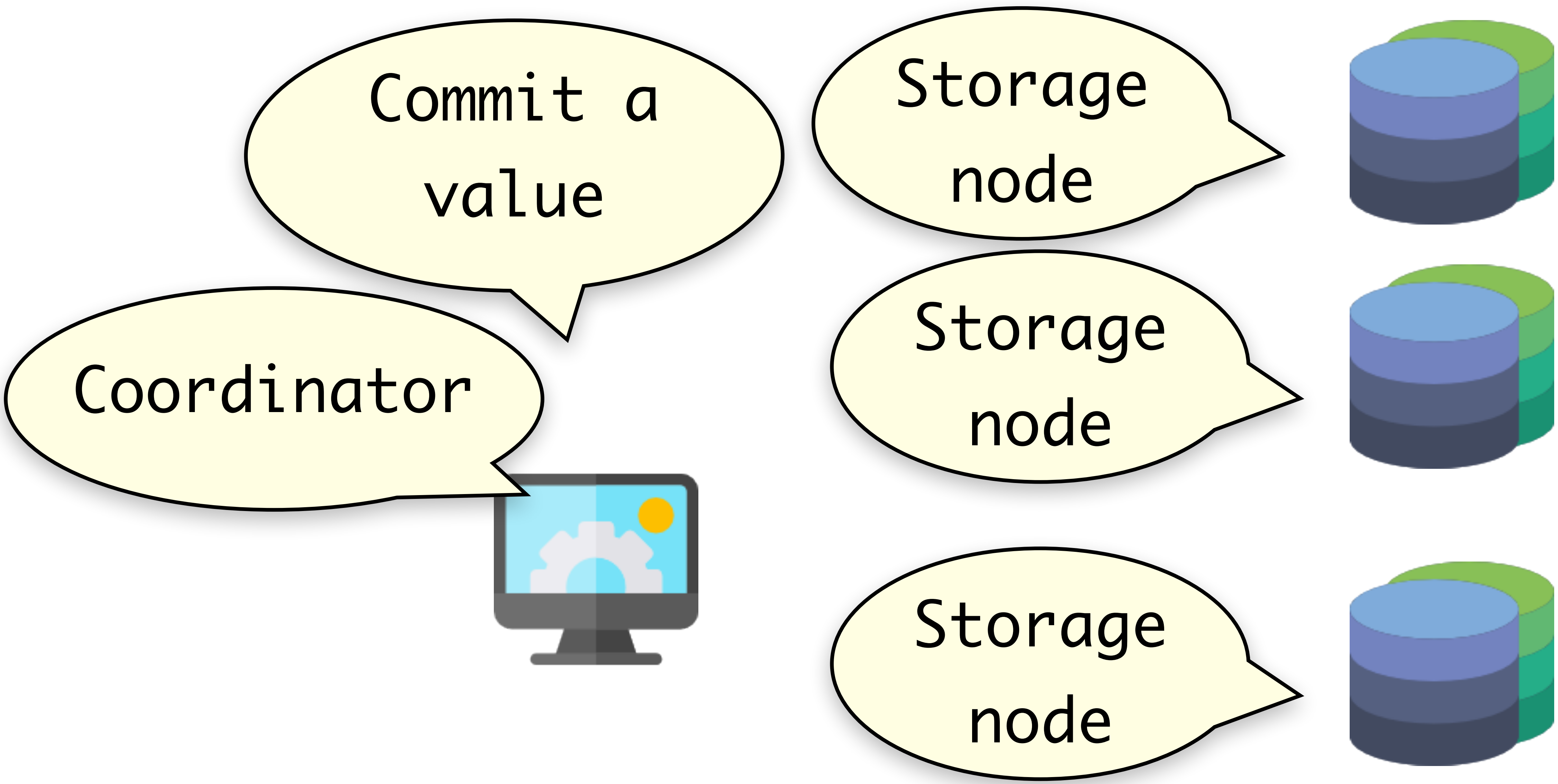
Synchronize by Rewriting

Implementation

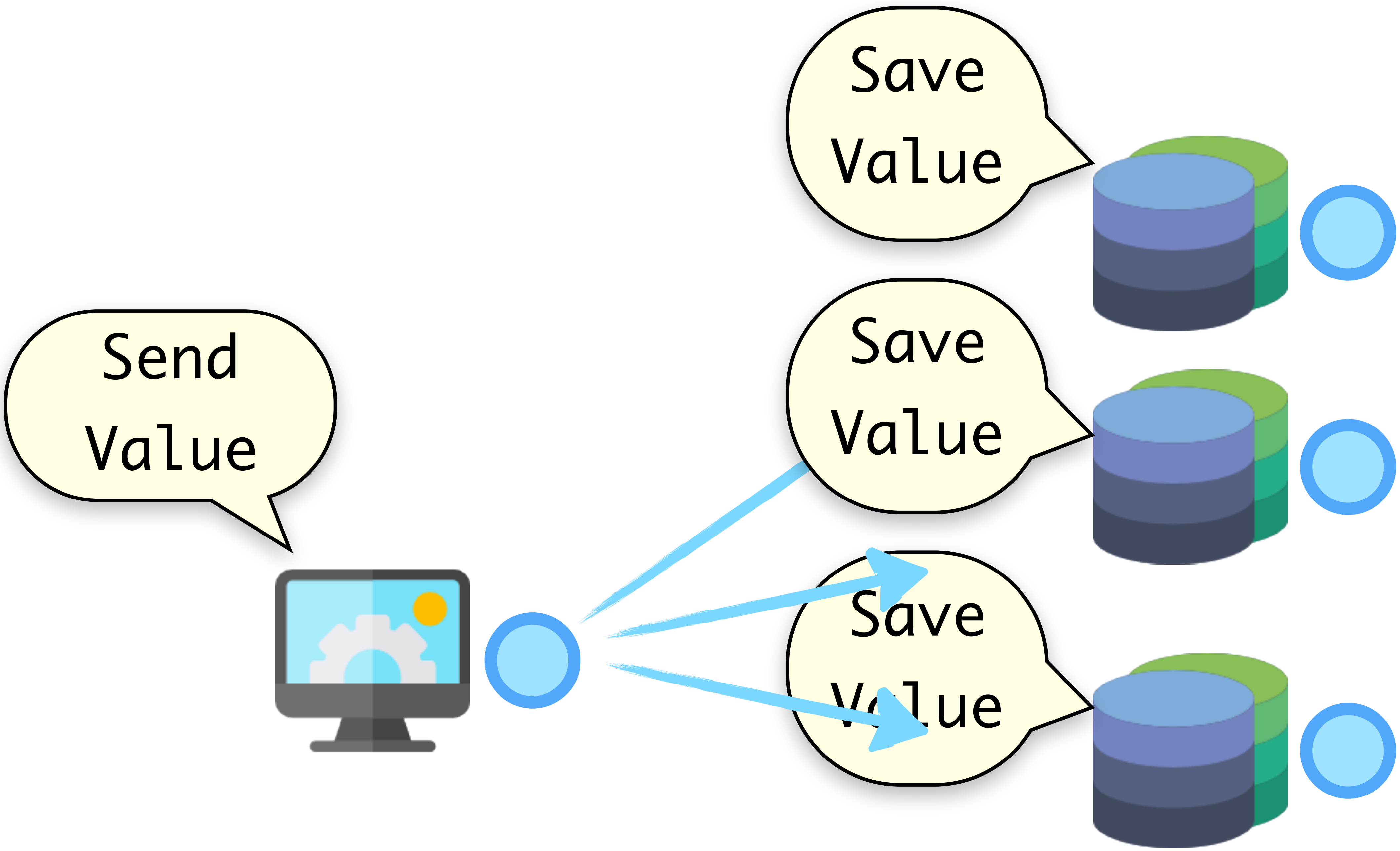
Example 2PC

Example: Two-phase Commit

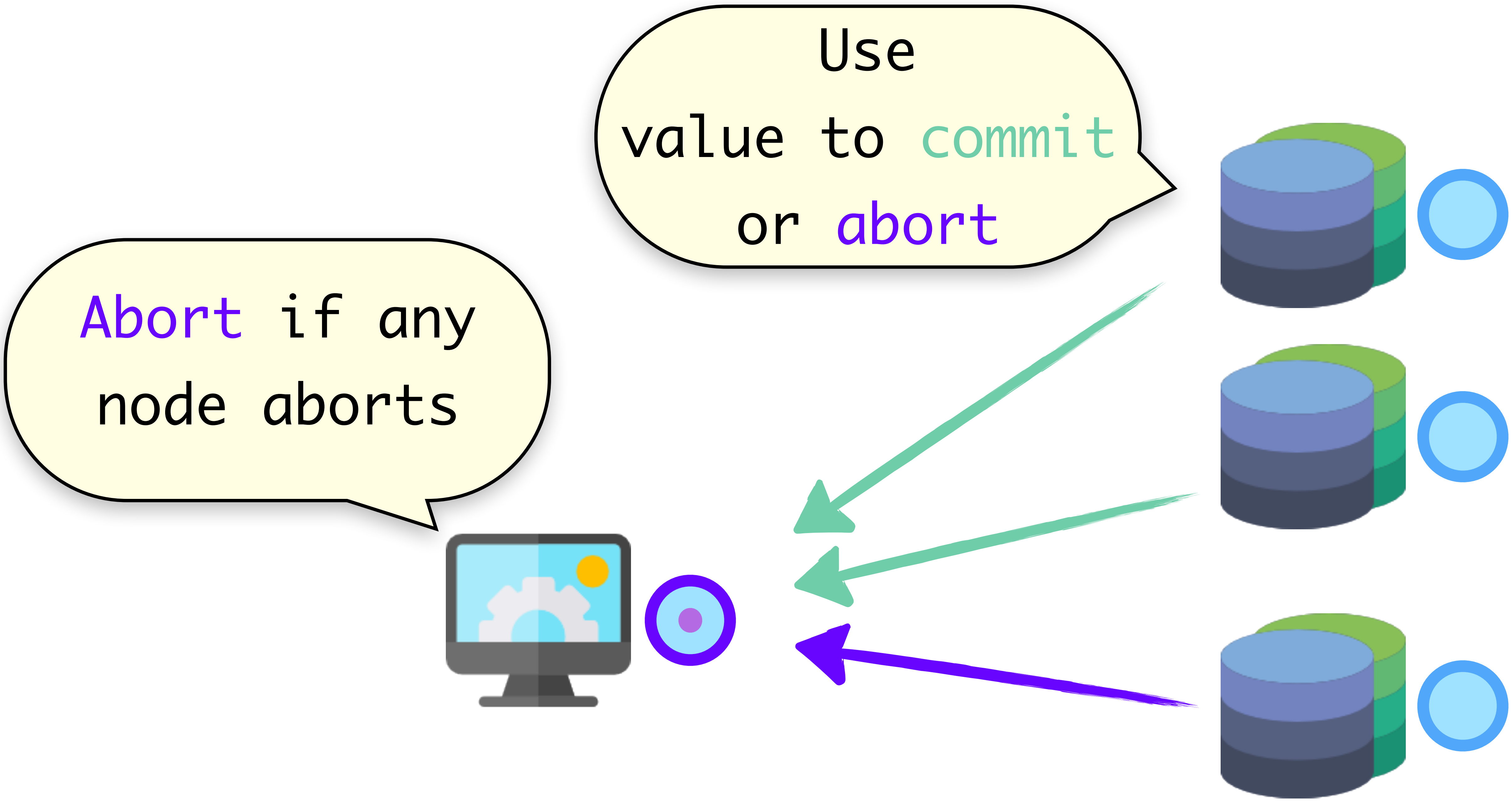
Example: Two-phase Commit



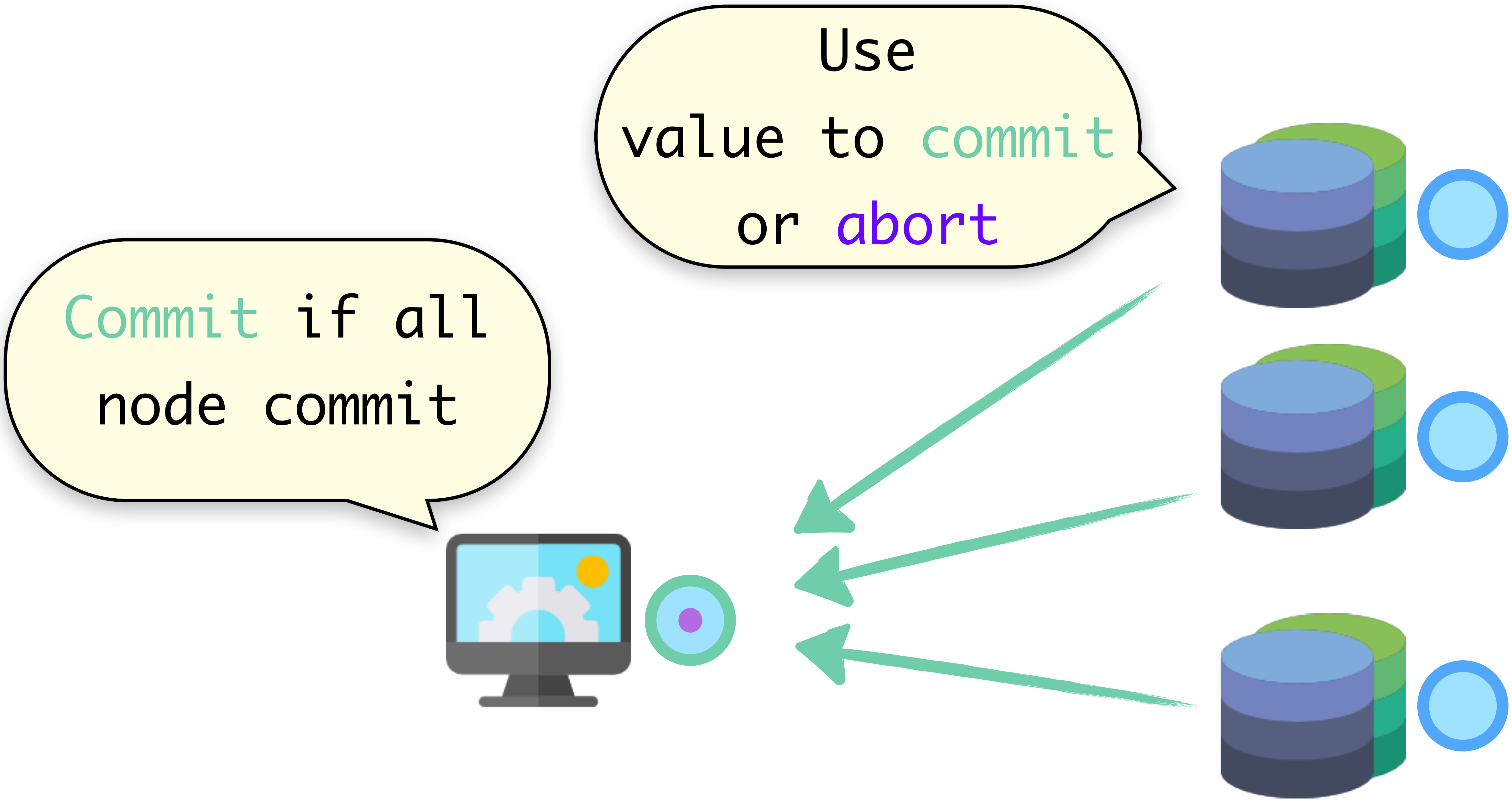
Two-phase Commit: Phase 1



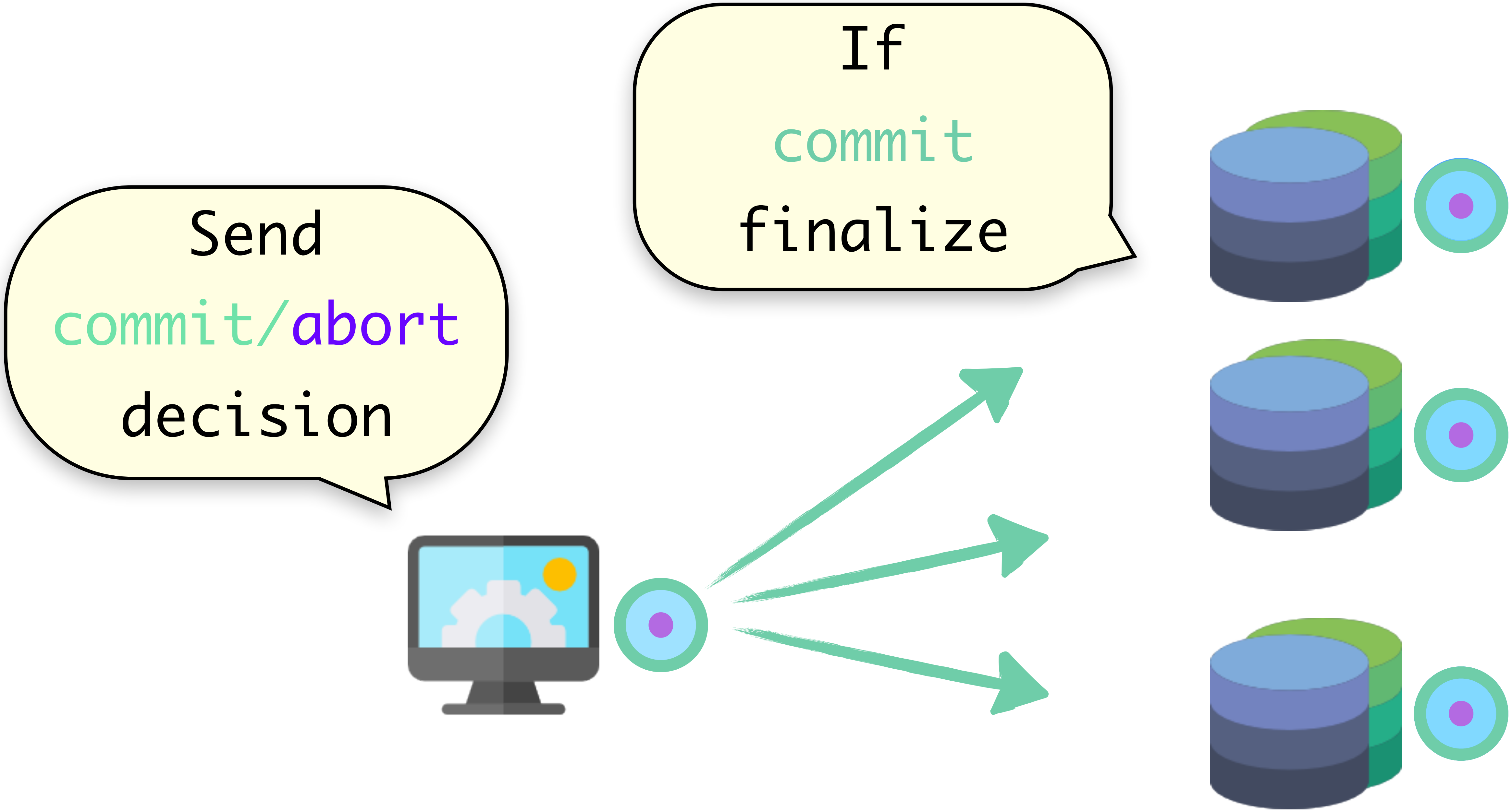
Two-phase Commit: Phase 1



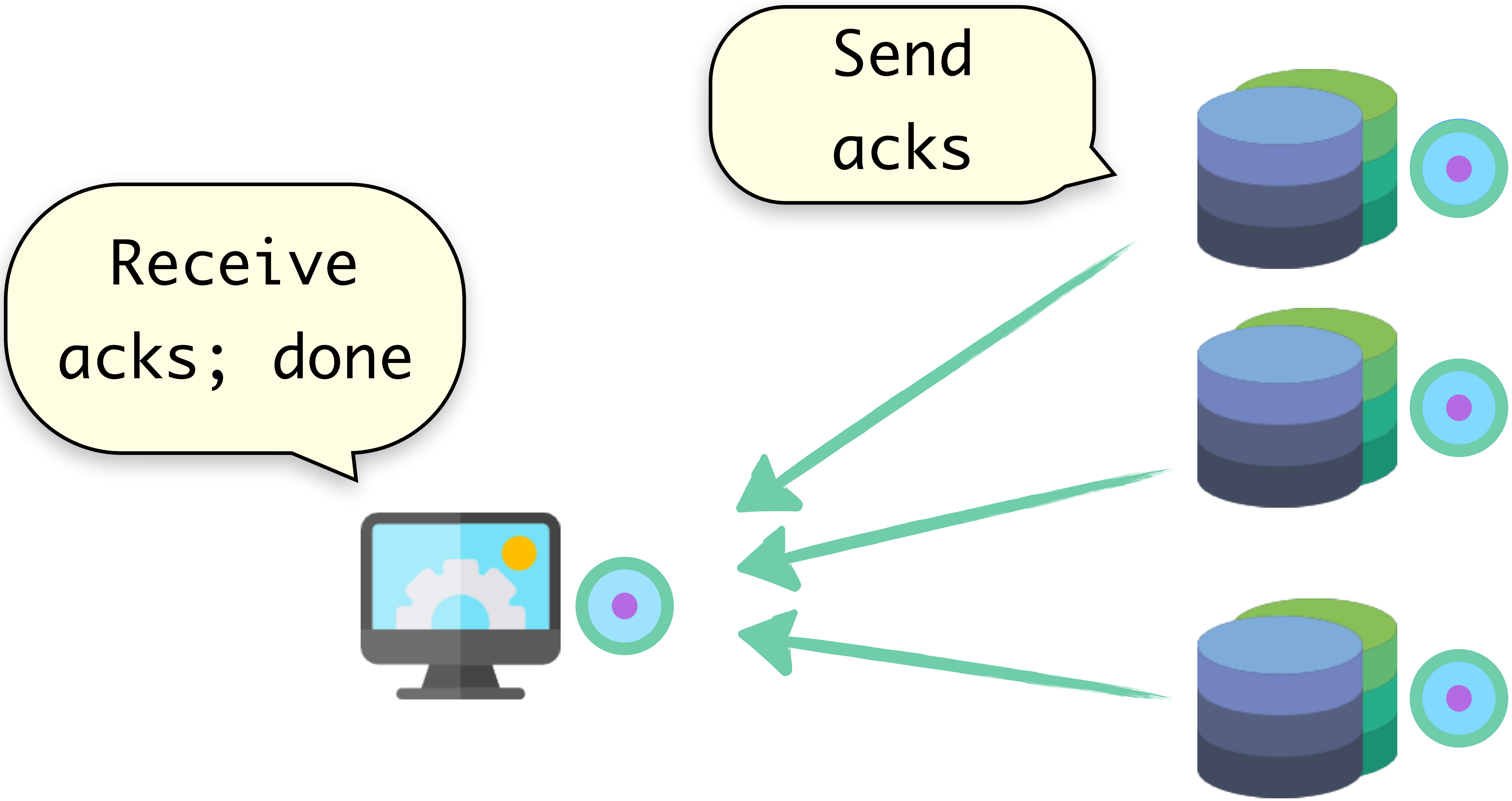
Two-phase Commit: Phase 1



Two-phase Commit: Phase 2



Two-phase Commit: Phase 2



Problem: *Asynchronous Proofs are Hard*

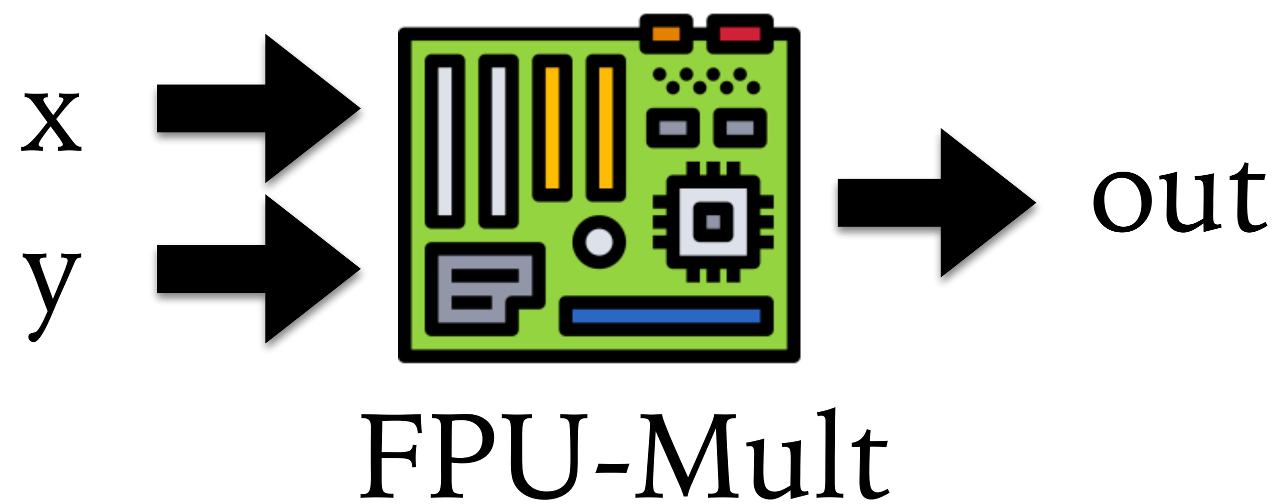
Make Proofs Easier!

Synchronous Proofs are Easy

Verilog Primer

Example FPU

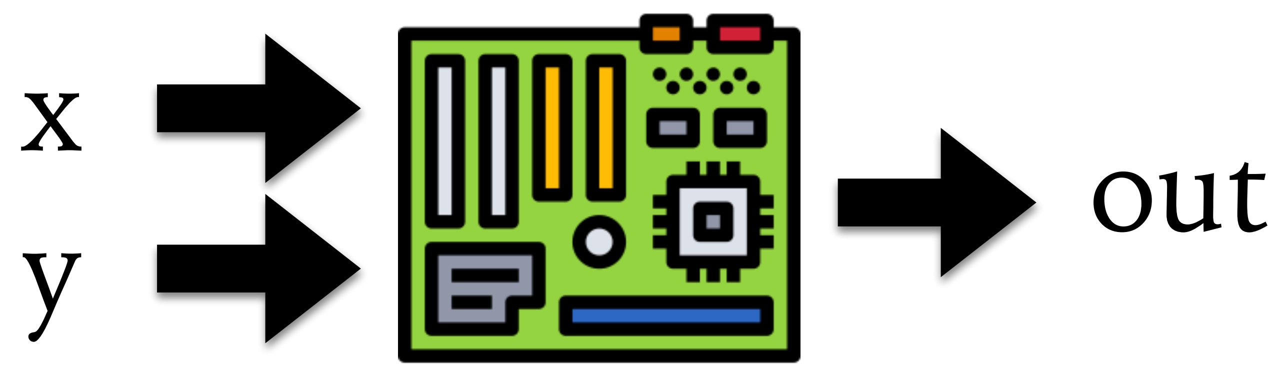
Given floats x and y



… compute x*y

Example FPU

Given floats x and y



FPU-Mult

… but exhibits timing variability

Example FPU

```
always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else if (isNaN)
        ...
    else
        out <= flp_res;
end
```

Example FPU

```
always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else if (isNaN)
        ...
    else
        out <= flp_res;
end
```

```
always @(posedge clk) begin
    ...
    flp_res <= // x*y;
end
```

Influence set:
cycles that influenced value

Example FPU

Influence set: for x=0

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}
…				
k-1	{k-1}	{k-1}	{0}	{k-2}
k	{k}	{k}	{1}	{k-1}

Example FPU

Influence set: x=1

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}
…				
k-1	{k-1}	{k-1}	{0}	{k-2}
k	{k}	{k}	{1}	{0,k-1}

sets for out at k differ: **timing variability**

Example FPU

How to verify?

Produce product program, track eqivalence of influence sets through equivalence of membership

Example FPU

The FPU takes a fast path, if x is 0

cycle	x	y	flp_res	out
0	0	1	X	X
1	0	1	X	0
…				
k-1	0	1	0	0
k	0	1	0	0

Example FPU

The FPU takes the slow path, if x is 1

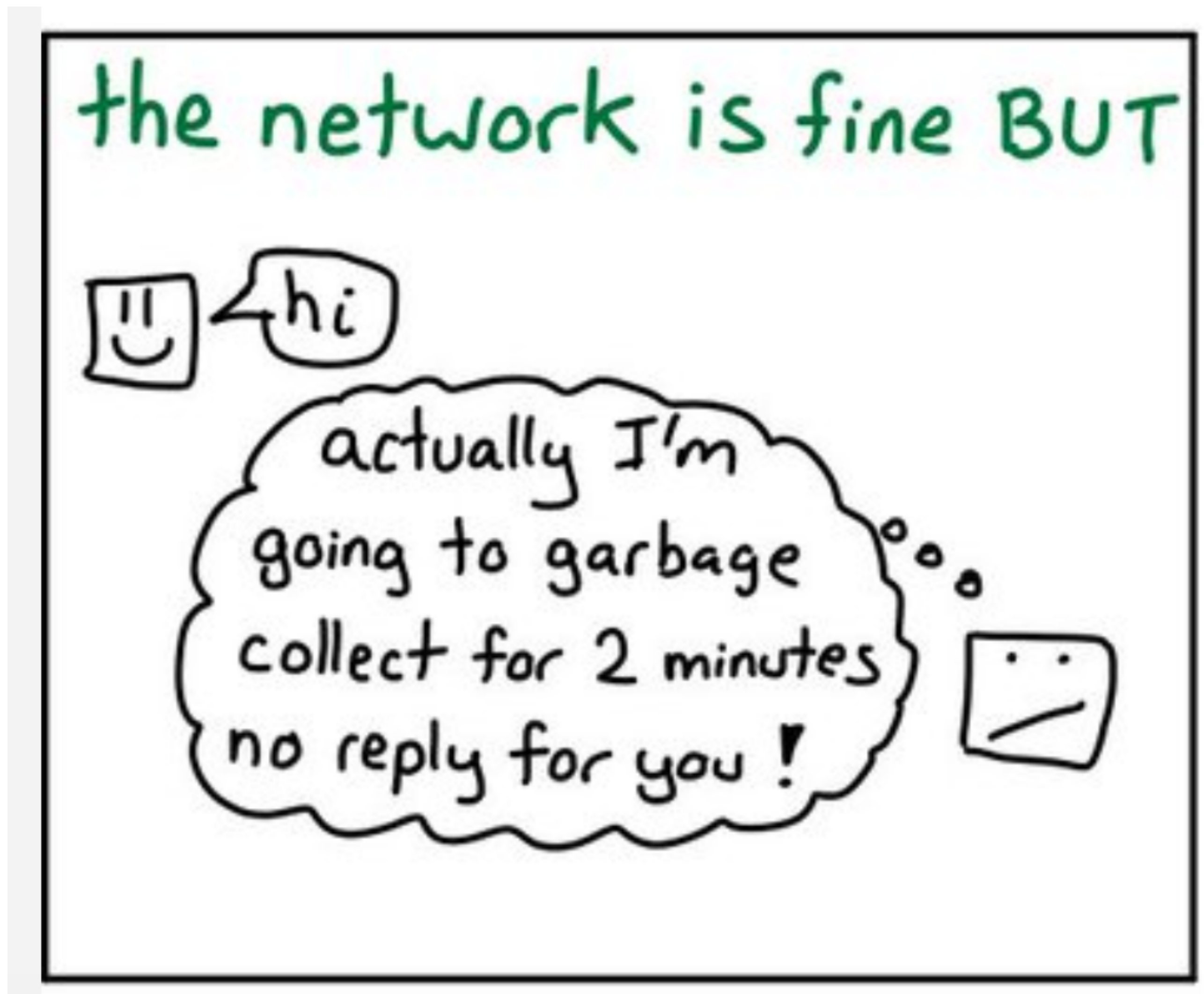
cycle	x	y	flp_res	out
0	1	1	X	X
1	0	1	X	0
…				
k-1	0	1	1	0
k	0	1	0	1

Example FPU

Influence set: all cycles that influenced value

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}
…				
k-1	{k-1}	{k-1}	{0}	{k-2}
k	{k}	{k}	{1}	{k-1}

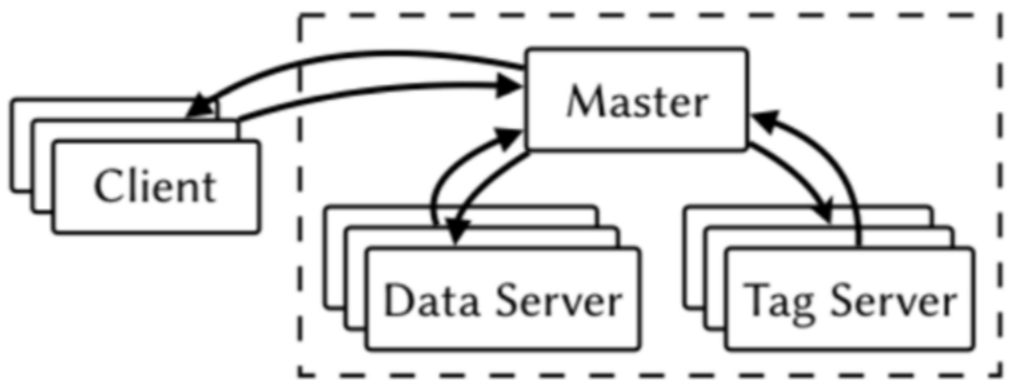
BACKUP


the network is fine BUT
hi
actually I'm going to garbage collect for 2 minutes no reply for you!

Results

Benchmark	Goolong #LG	#LI	#A	#I	RW (s)	Chk (s)	Dafny #LI	#A	#I	#H	Chk (s)
Two-Phase Commit	102	49	2	3	0.17	0.04	55	8	30	62	12.81
Raft Leader Election	138	44	17	6	0.19	0.18	40	20	50	73	301.68
Single-Decree Paxos	504	65	23	14	0.25	1.51	69	50	72	63	1141.35
Total	744	158	42	23	0.61	1.73	164	78	152	198	1455.84
Multi-Paxos KV	847	100	21	14	0.24	1.64	-	-	-	-	-

BACKUP



Master:
AllocBlob(name)
PutBlob(name, data)
GetBlob(name)
AddTag(tag, refs)
GetTag(tag)

Tag Server:
AddTag(name)
GetTag(tag)

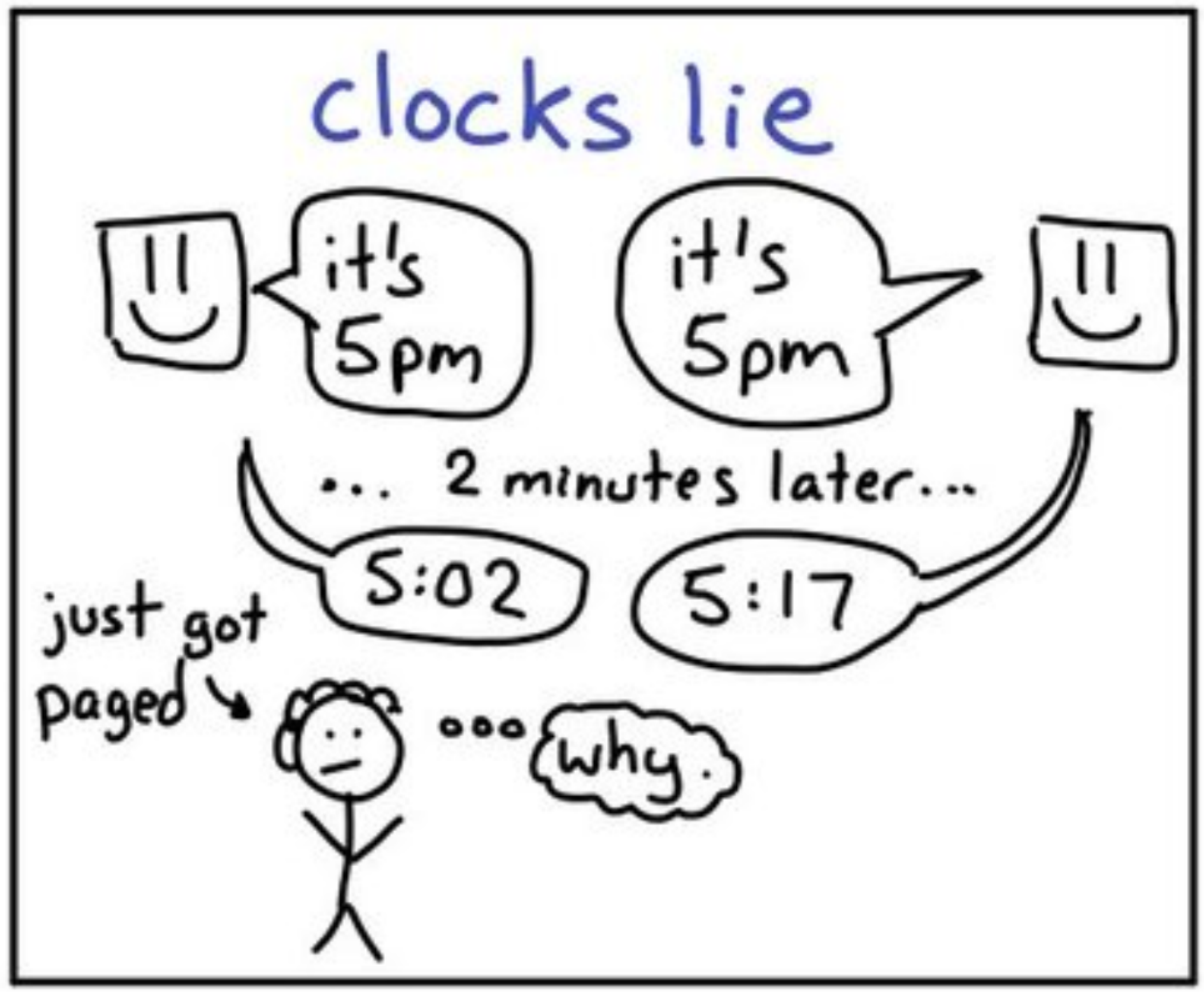
Data Server:
PutBlob(name, data)
GetBlob(name)

Tags	**Data**
Mutable	Immutable

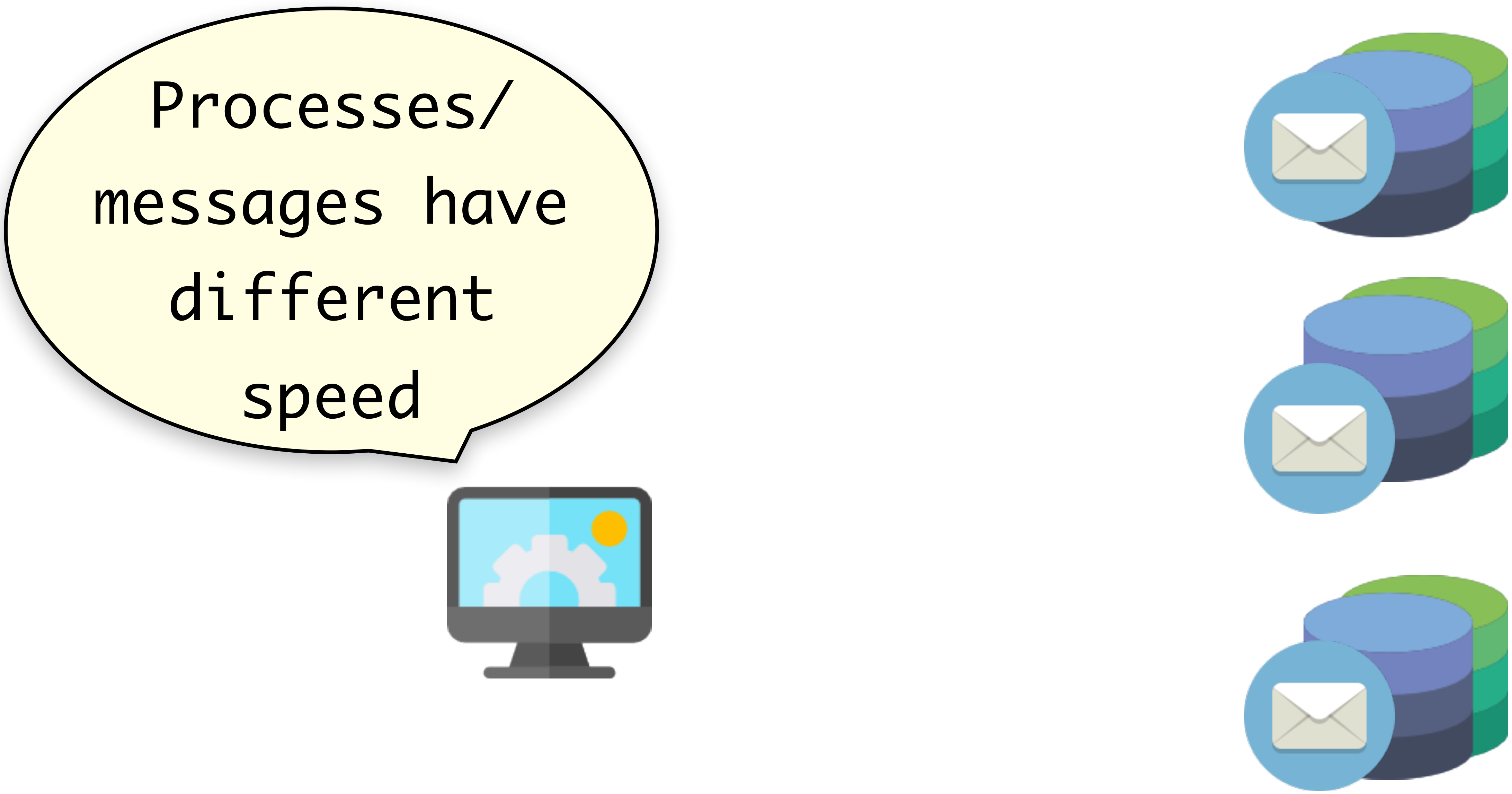
BACKUP

System	**Throughput (req/ms)**
GOOLONG	118.5
PSync	32.4
Ivy-Raft*	13.5
IronKV*	~ 30

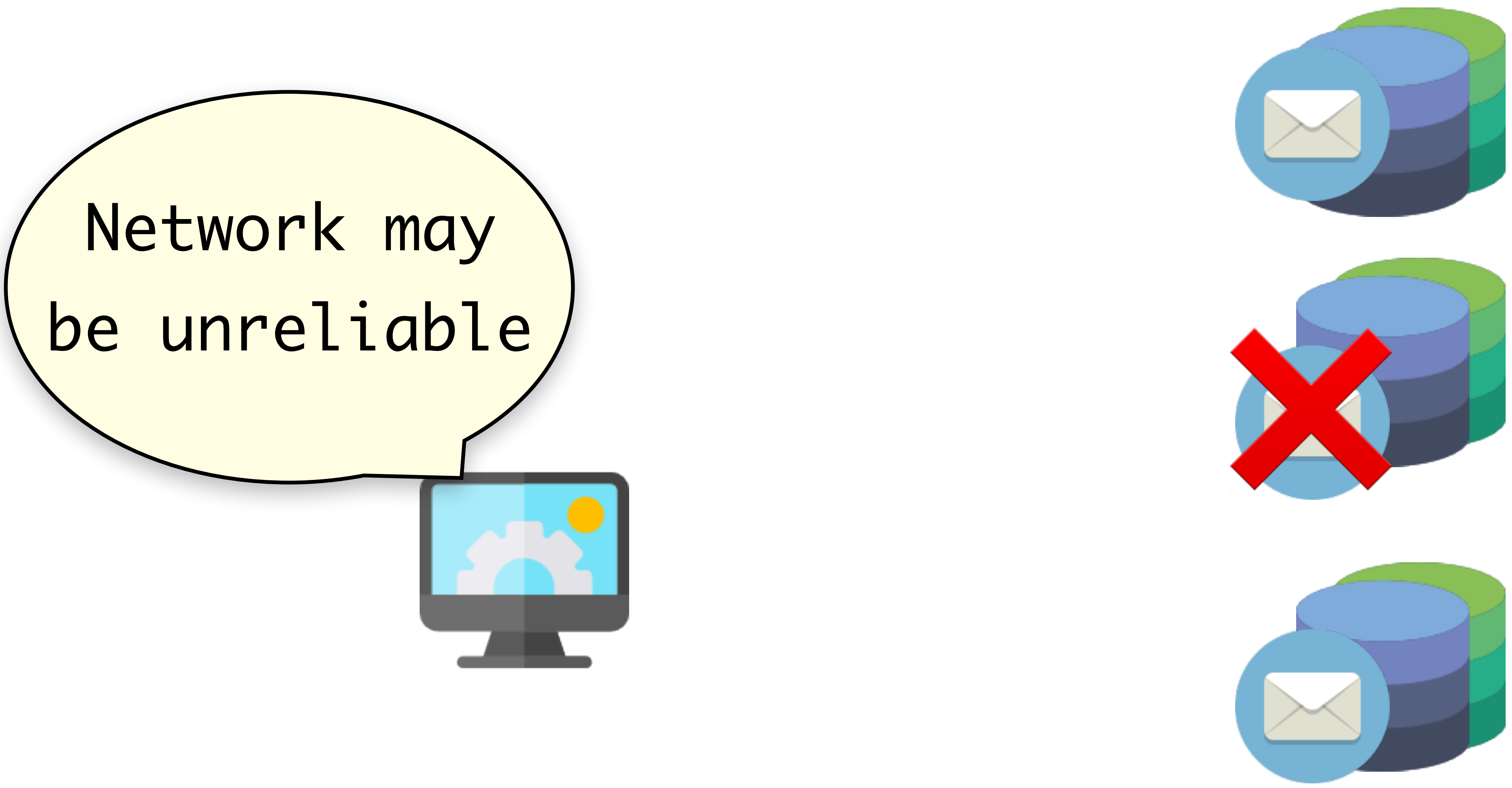
BACKUP


clocks lie
it's 5pm
it's 5pm
... 2 minutes later...
5:02
5:17
just got paged
why.

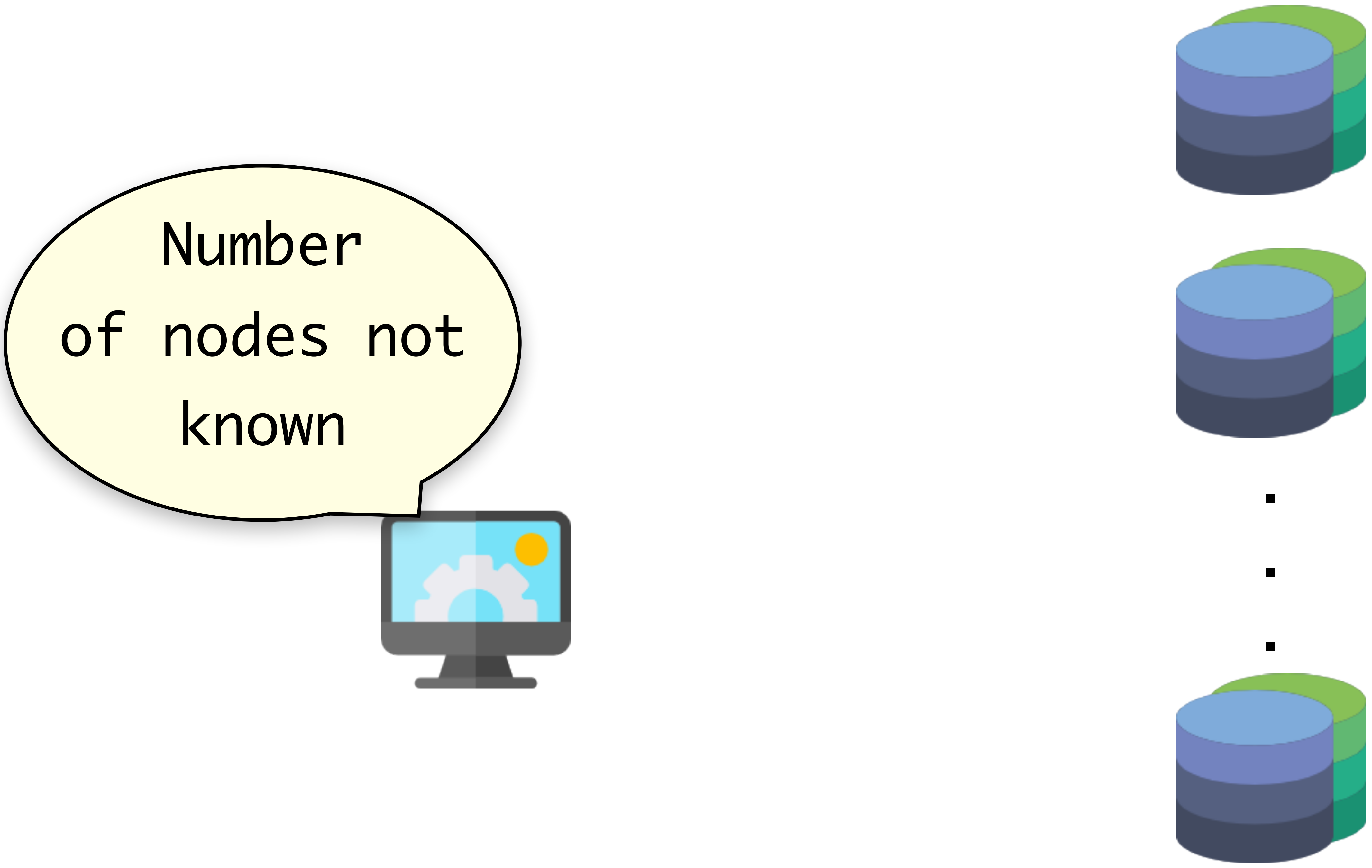
Problem 1: Asynchrony



Problem 2: Message Drops



Problem 3: Parametrized



How to make sure they don't?

Too

many

possibilitie

Testing

Model Checking

Deductive Proofs

Infinite Number

of States:

No guarantees