Verifying Distributed Programs via Canonical Sequentialization

Klaus von Gleissenthall

Joint work with

Alexander Bakst, Ranjit Jhala and Rami Gökhan Kıcı

Writing distributed programs

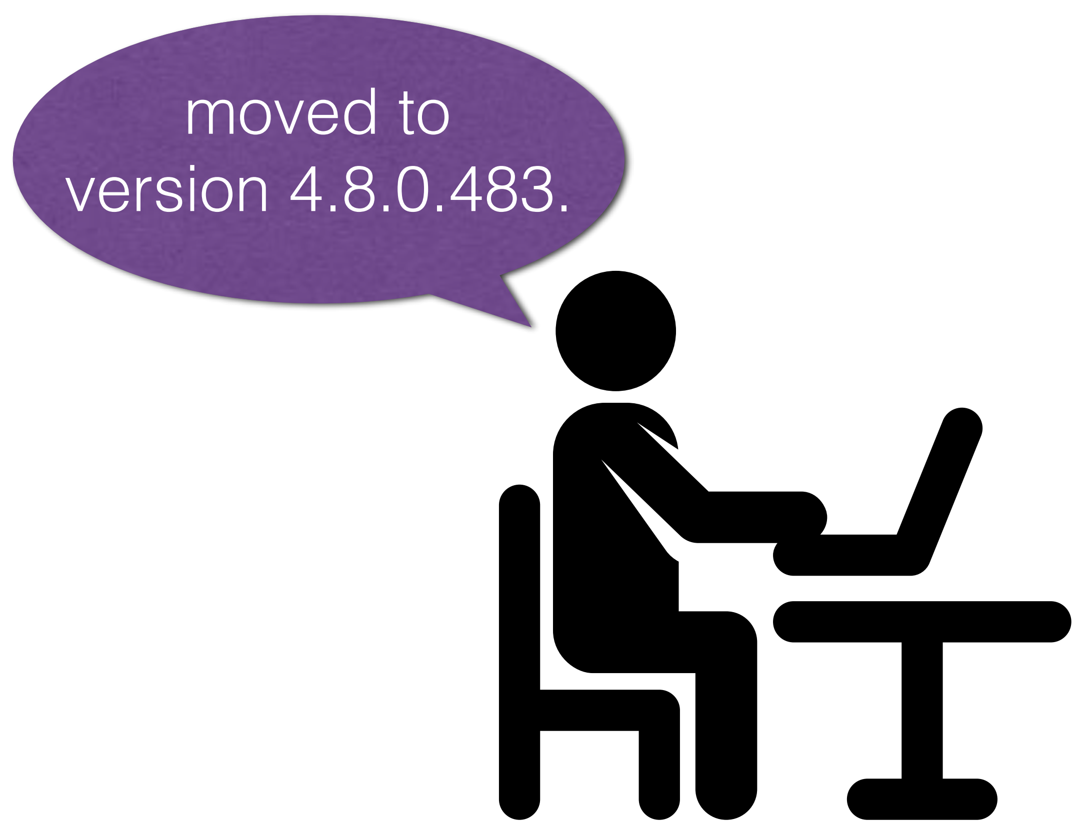
A bug appears…

Writing distributed programs

... haunts you ...

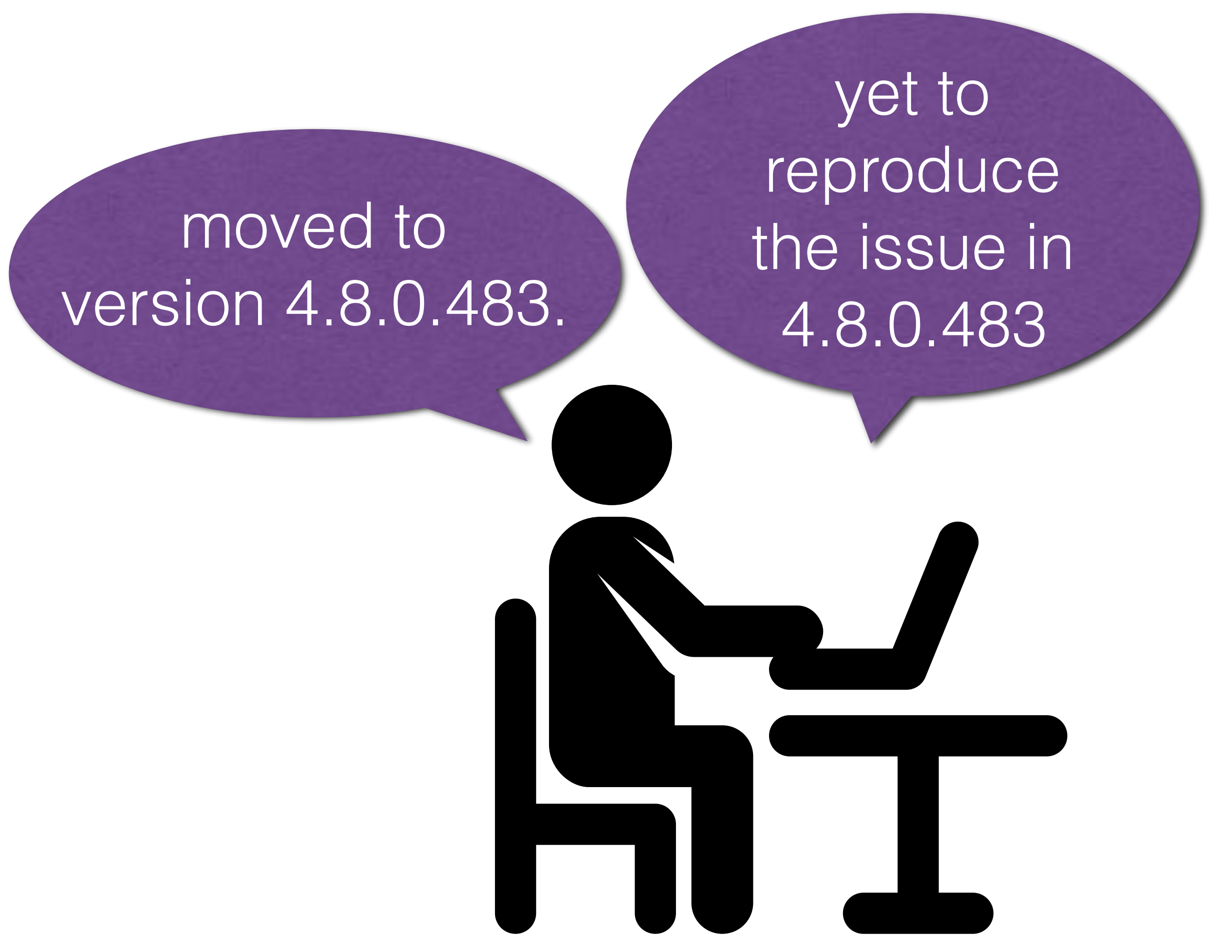
Writing distributed programs

… then you write some more code…



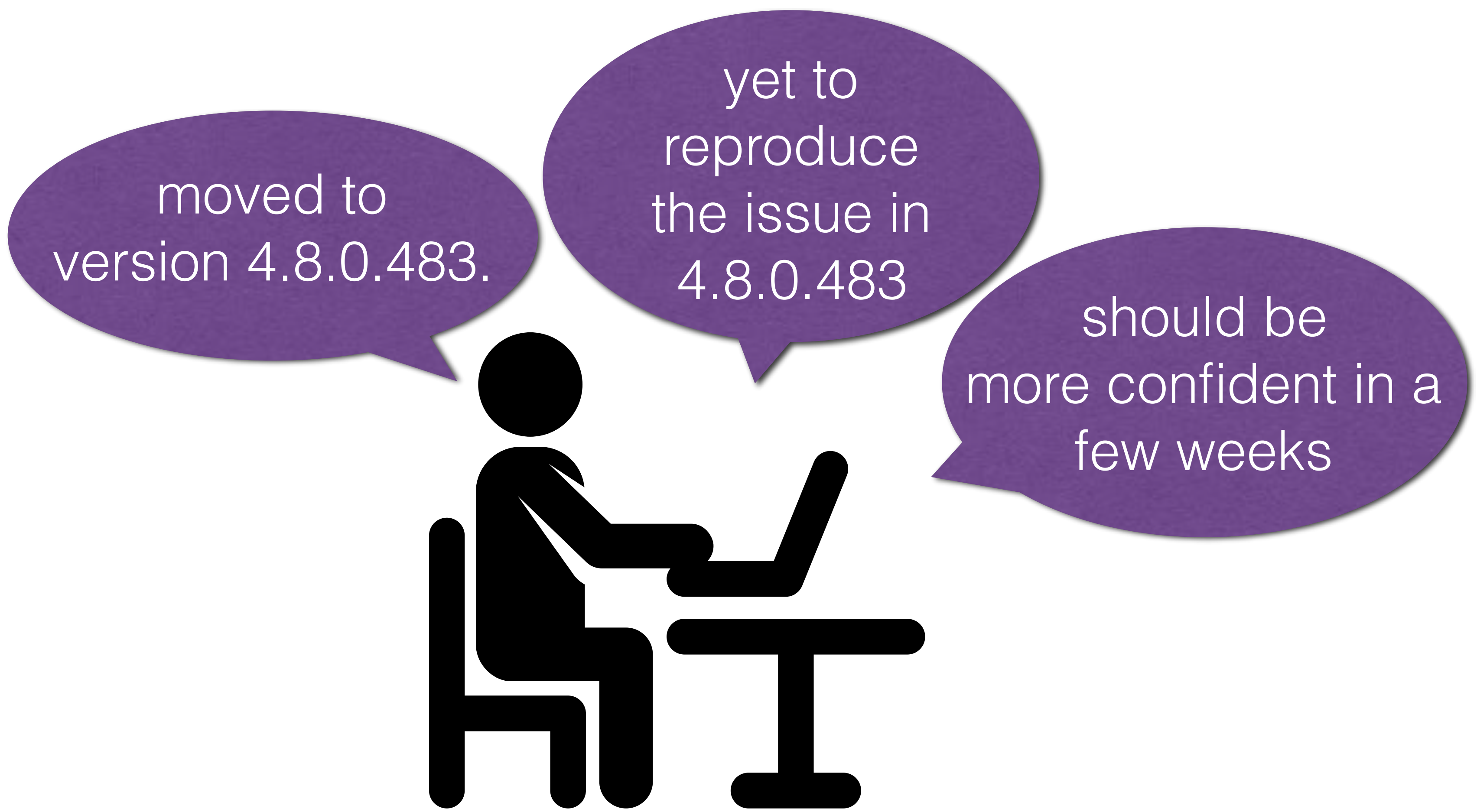
Writing distributed programs

… and the bug disappears…



Writing distributed programs

…leaving you hoping it stays gone.



A better world

Can we catch all deadlocks during compile-edit cycle?

A better world

let's fix it

```
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

          y () pid      = do { me <- myPid; send pid (pid, transaction) }
         untVotes 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
```

sent wrong response address

unmatched receive

unmatched send

check

A better world

proof
No deadlocks can occur!

```
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
```

check

This talk: Brisk

- ⊕ Proves absence of deadlocks
- ⊕ Provides counterexamples
- ⊕ Fast enough for interactive use
- ⊖ Restricted computation model

⊖ Restricted computation model

But Expressive Enough to Implement:

- Work Stealing
- Map Reduce
- Distributed File System

Outline

The Problems

The Key Idea

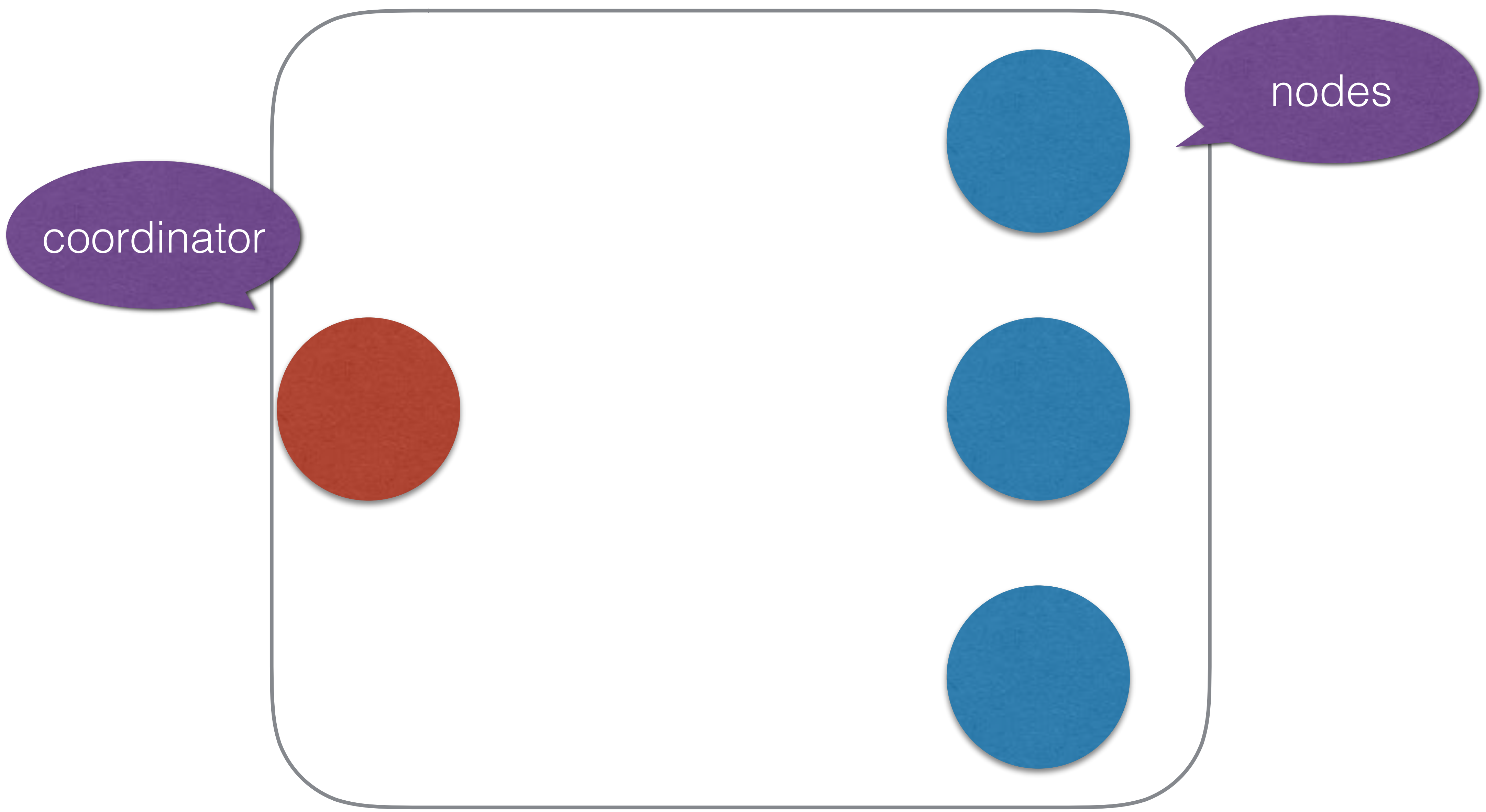
The Implementation

The Evaluation

The Problems

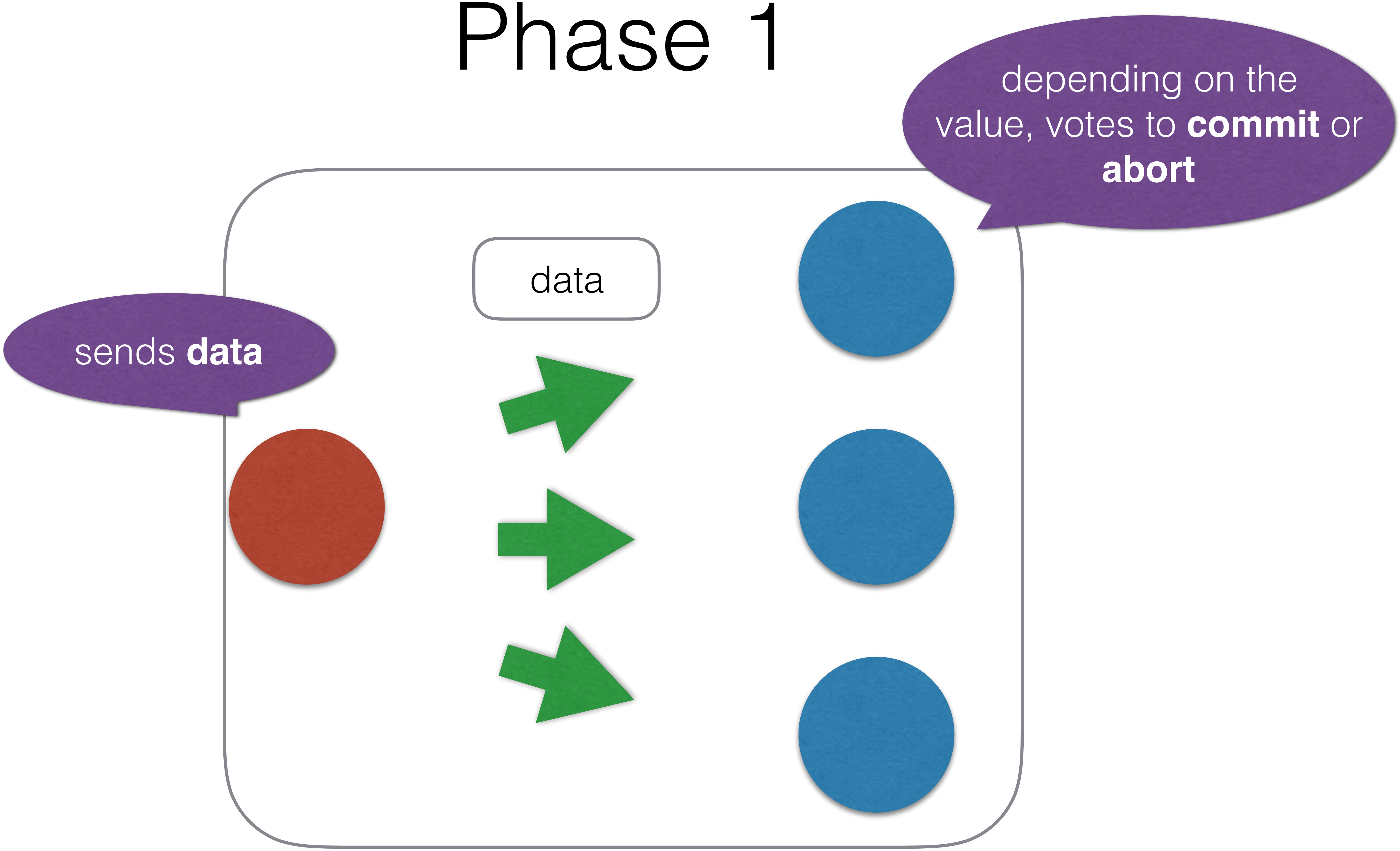
Example: Two phase commit (2PC)

Goal: Commit Transaction to all nodes



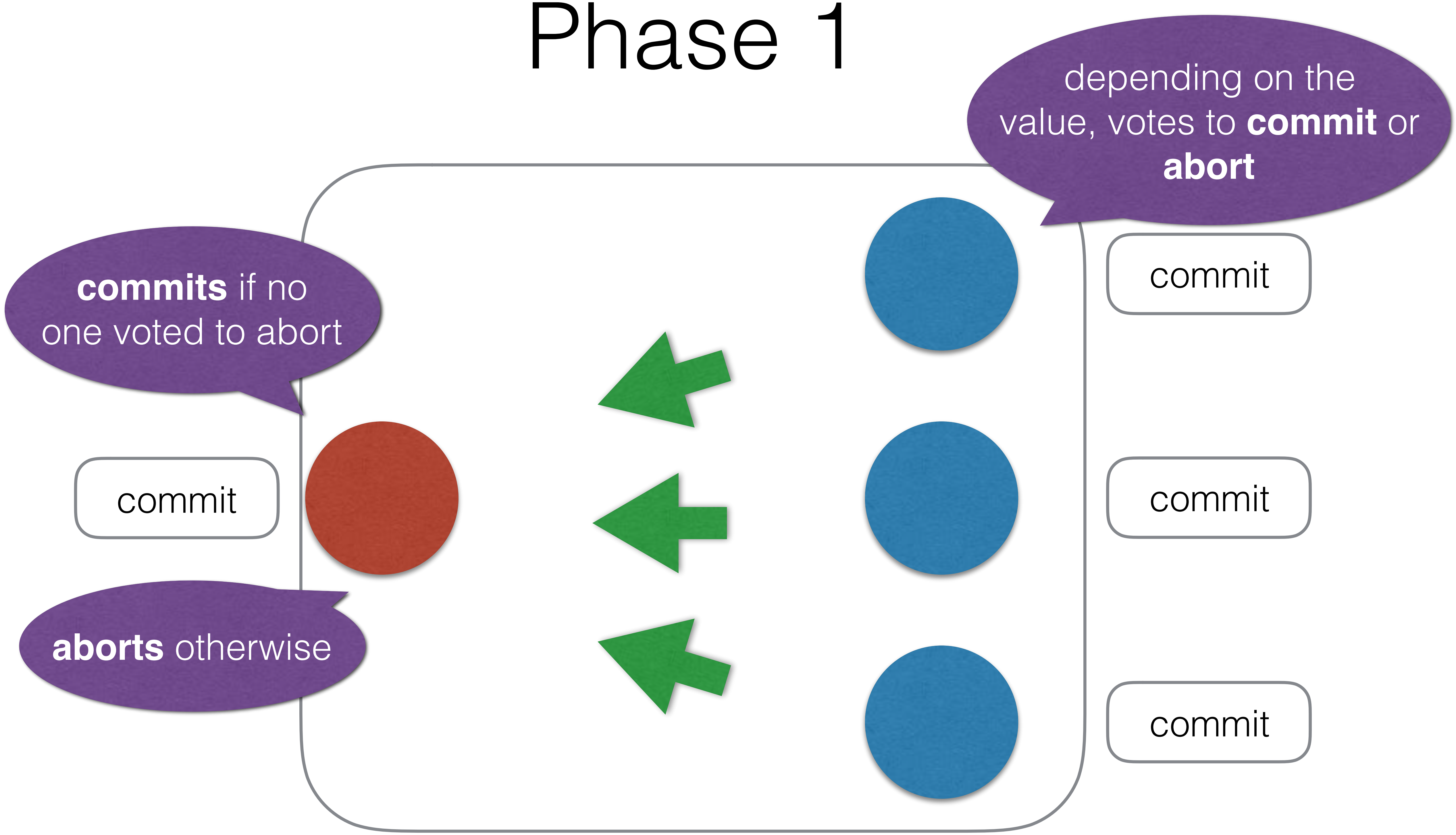
Example: Two phase commit (2PC)

Phase 1



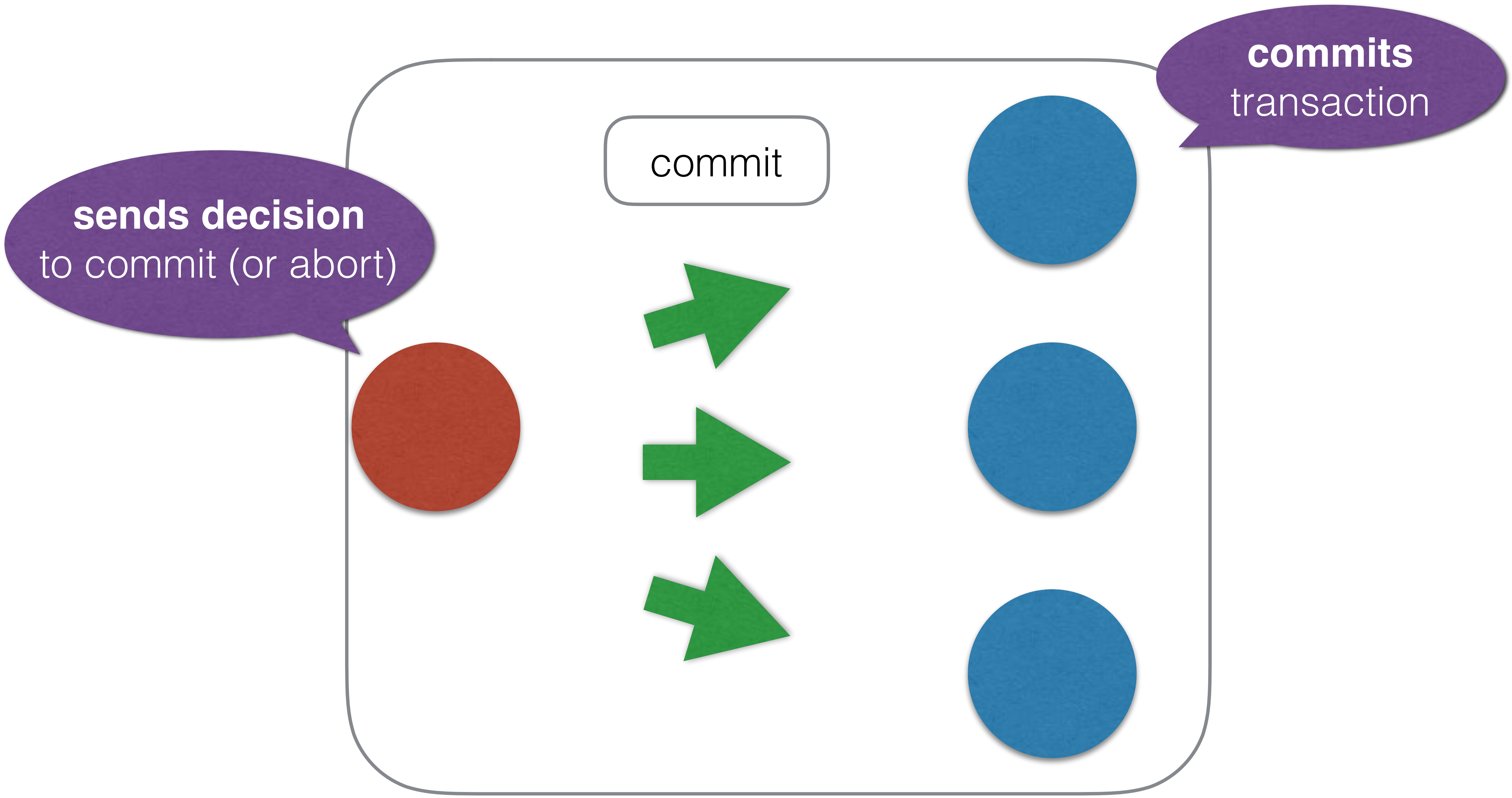
Example: Two phase commit (2PC)

Phase 1



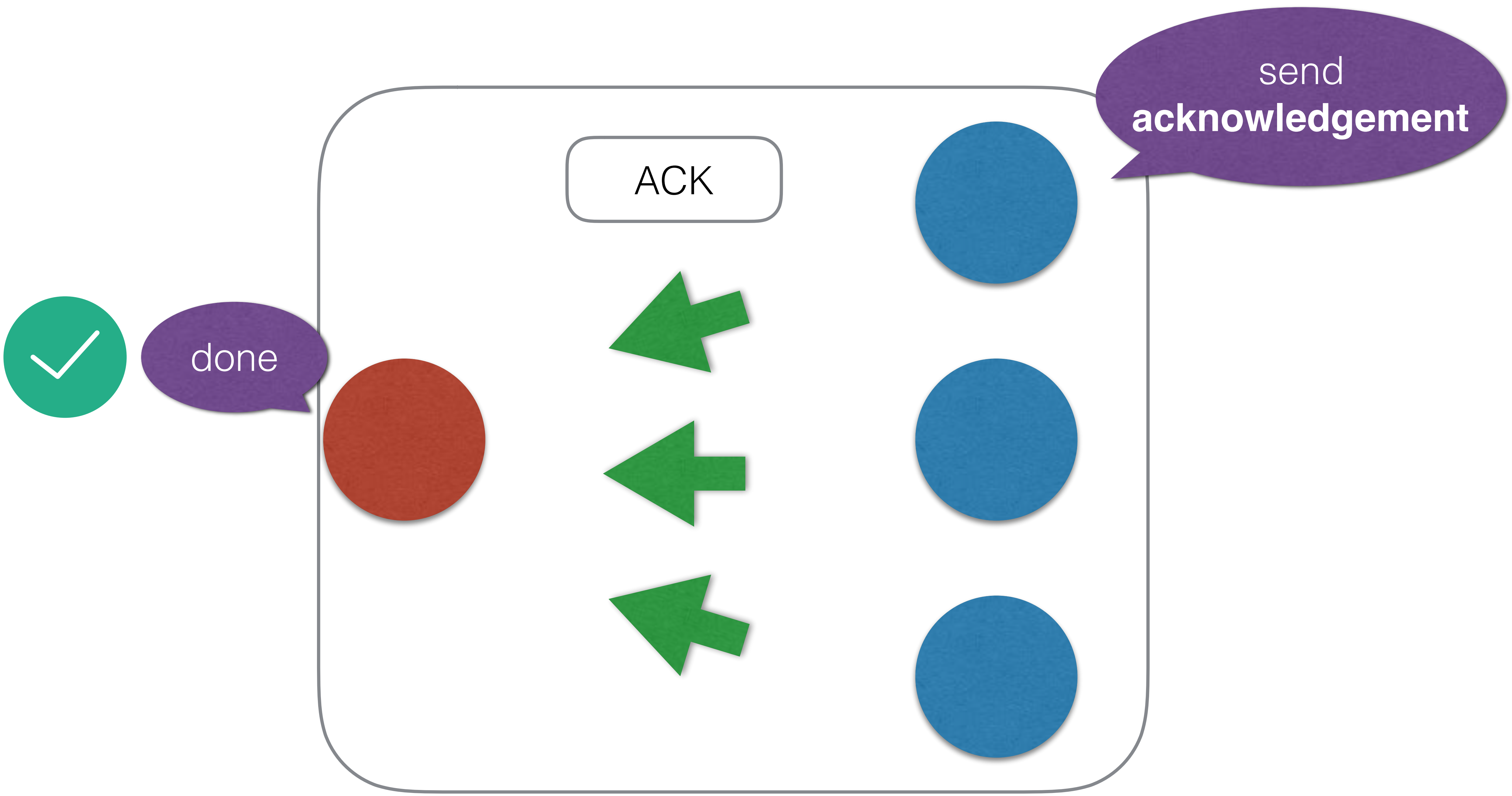
Example: Two phase commit (2PC)

Phase 2



Example: Two phase commit (2PC)

Phase 2



How to verify 2PC?

Sends match **receives?**

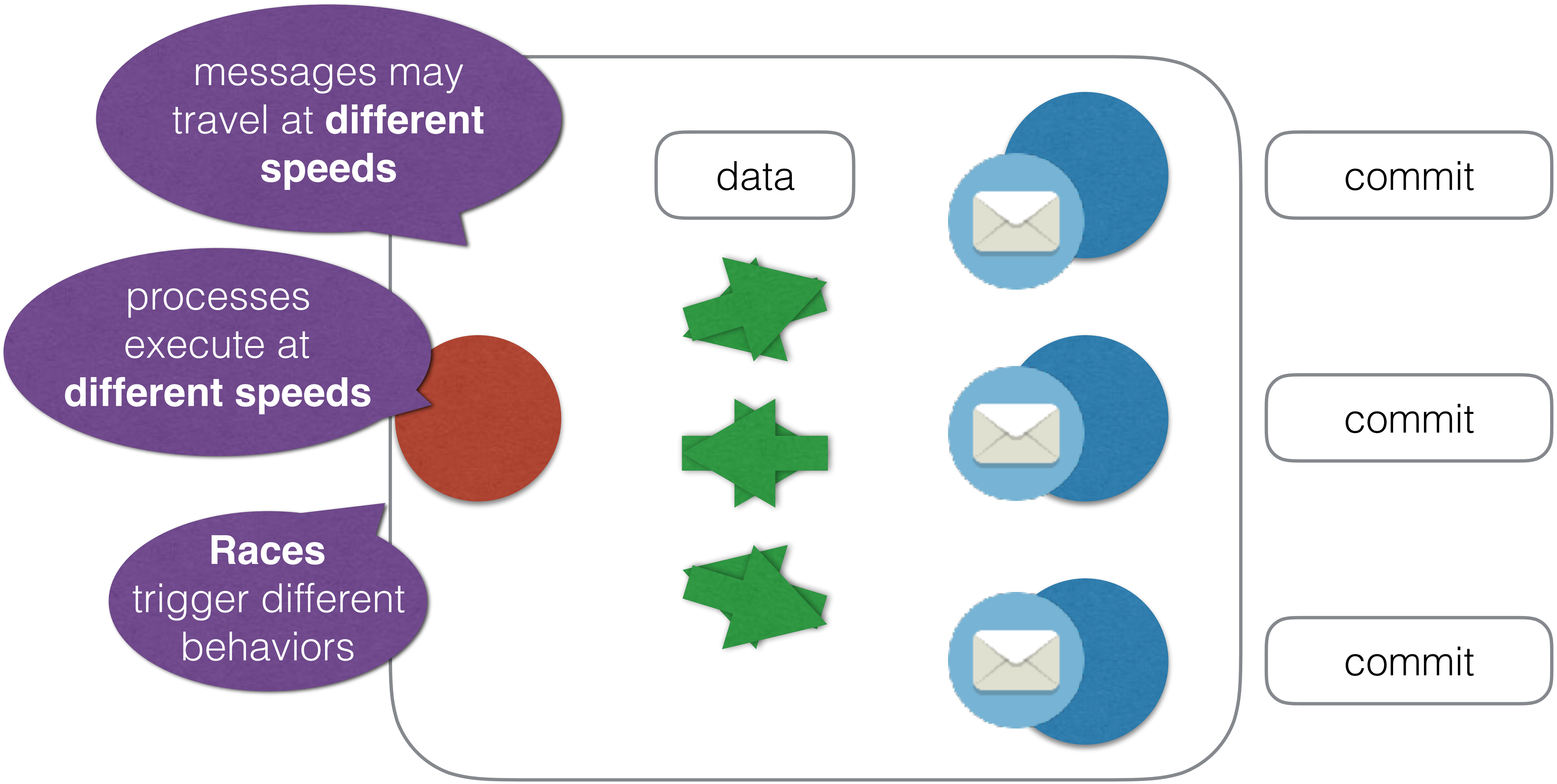
```
coord :: Transaction -> Int -> SymSet ProcessId -> Process ()
coord transaction n nodes = do
                  fold () query nodes
                  n_ <- fold 0 countVotes nodes
   where
        query () pid  = do { me <- myPid; send pid (me, transaction) }
        countVotes c _ = do
         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

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

Does Implementation Deadlock?

How to verify 2PC?

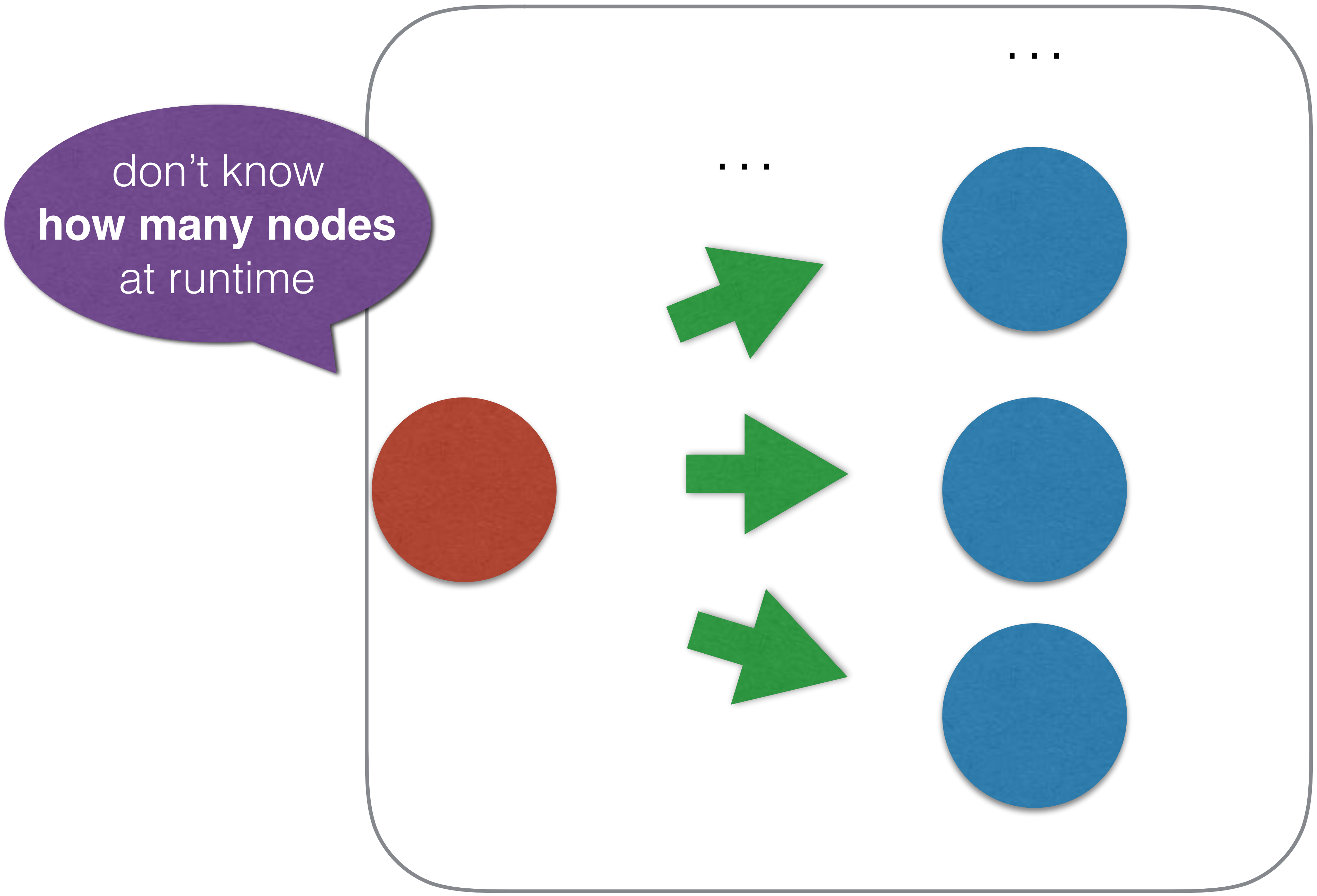
How to verify 2PC?

Problem: Asynchrony



How to verify 2PC?

Problem: Unbounded Processes



How to verify 2PC?

Testing?
No guarantees

Proofs?
High user burden

Model checking…?
Infinite number of states

Outline

The Problems

The Key Idea

The Implementation

The Evaluation

Outline

The Problems

The Key Idea

The Implementation

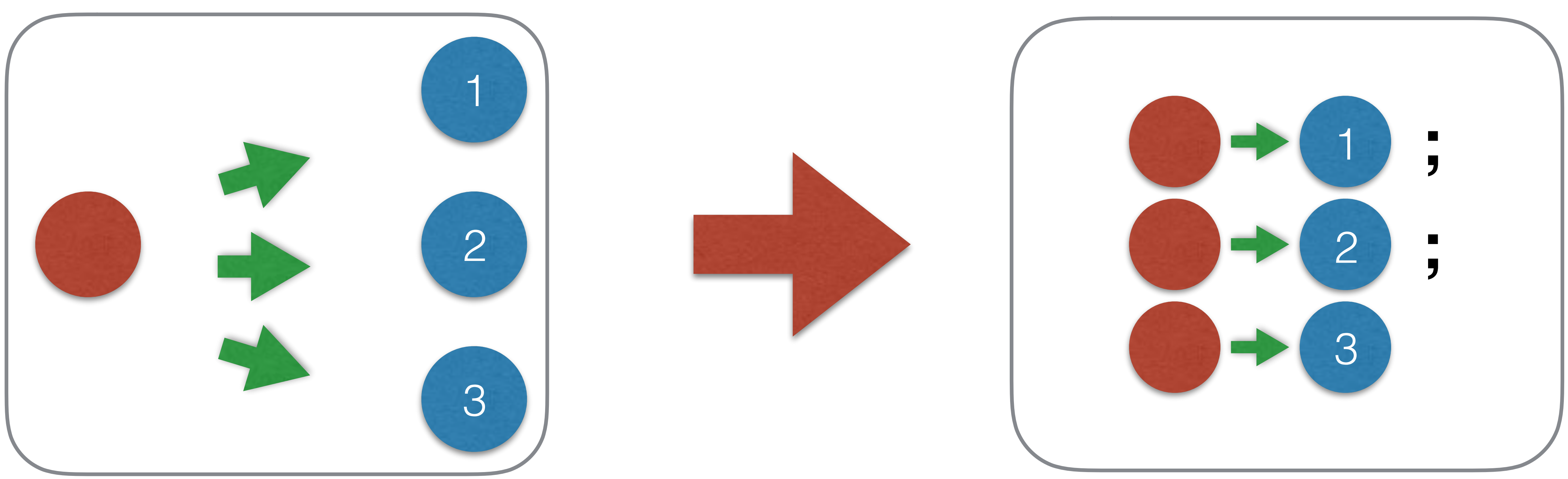
The Evaluation

The Key Idea

Canonical Sequentialization

Canonical Sequentialization

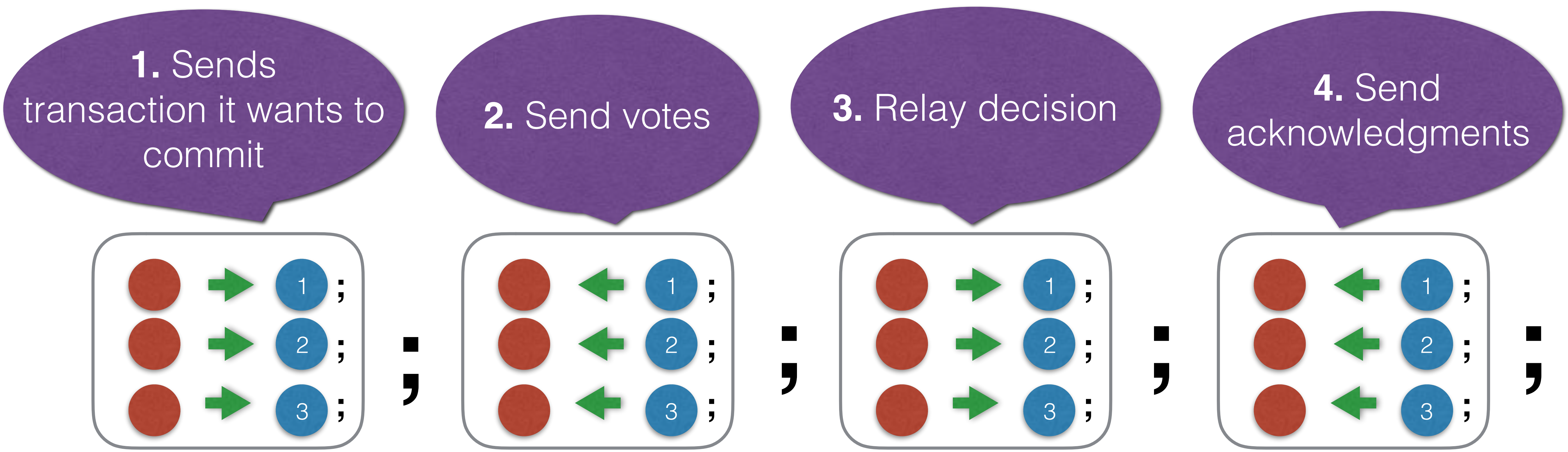
Don't enumerate execution orders…



… **Reason** about **single representative** execution

Canonical Sequentialization

Example 2PC

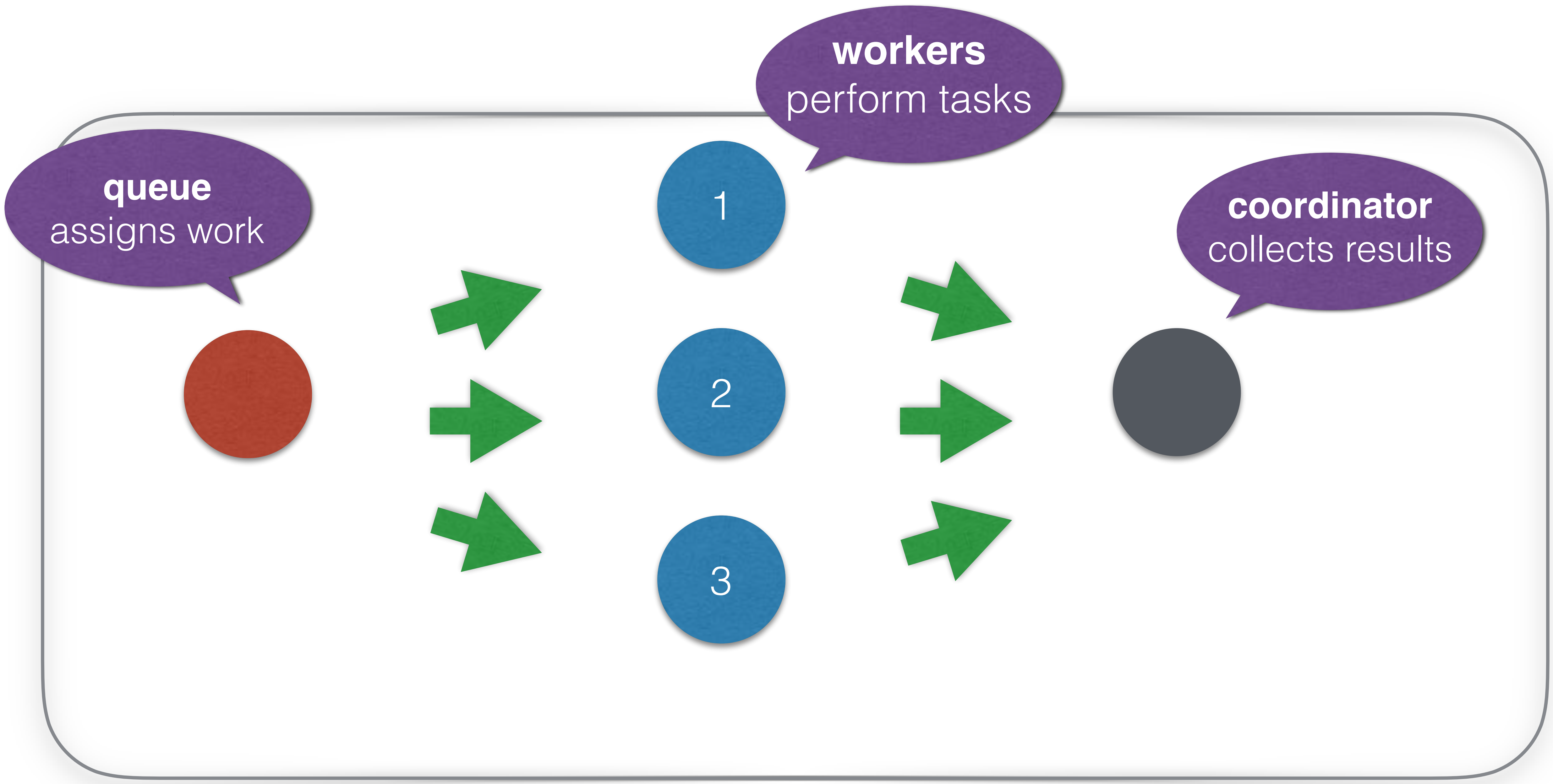


Canonical Sequentialization

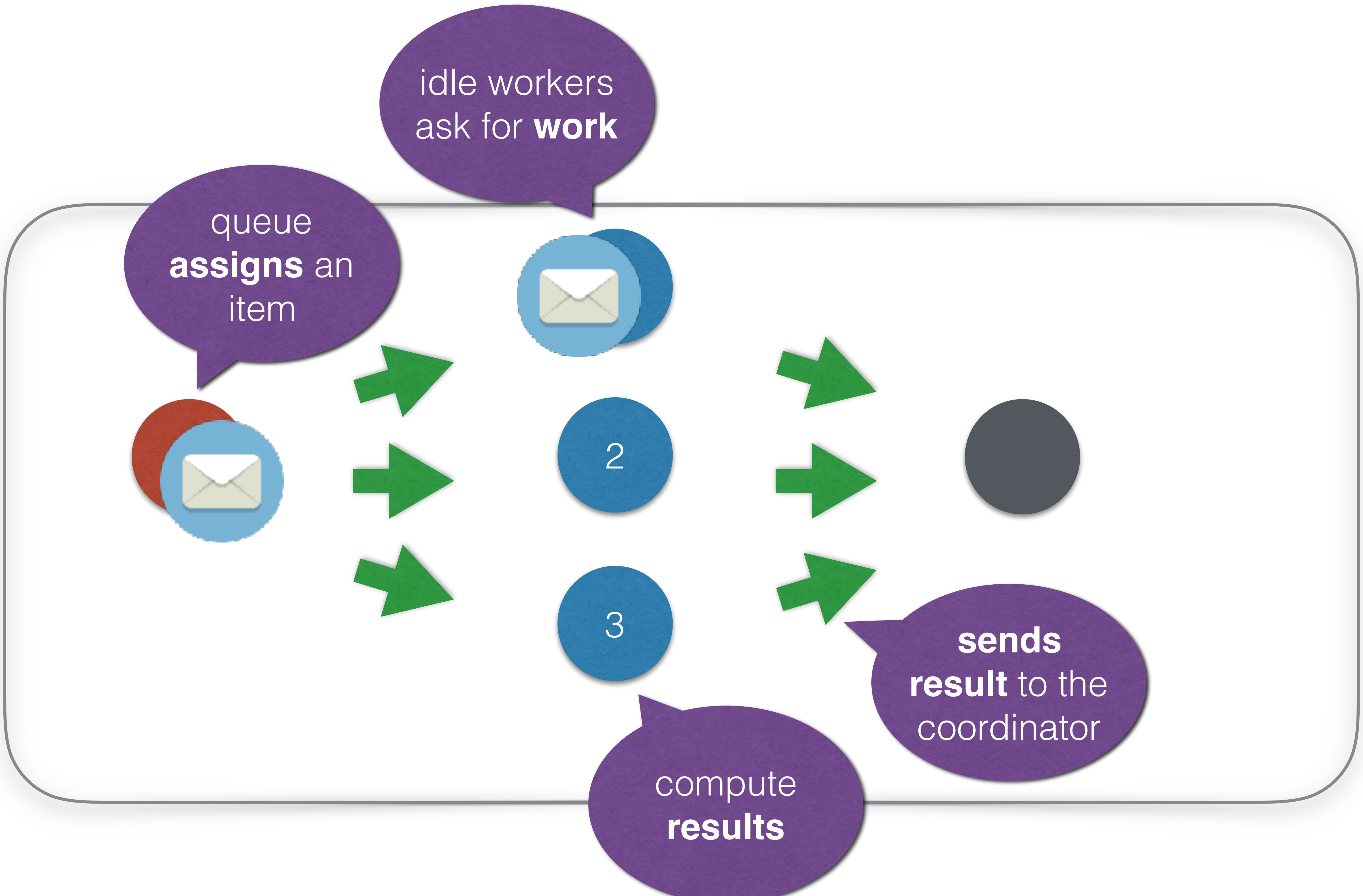
A Trickier Example

Work stealing queue

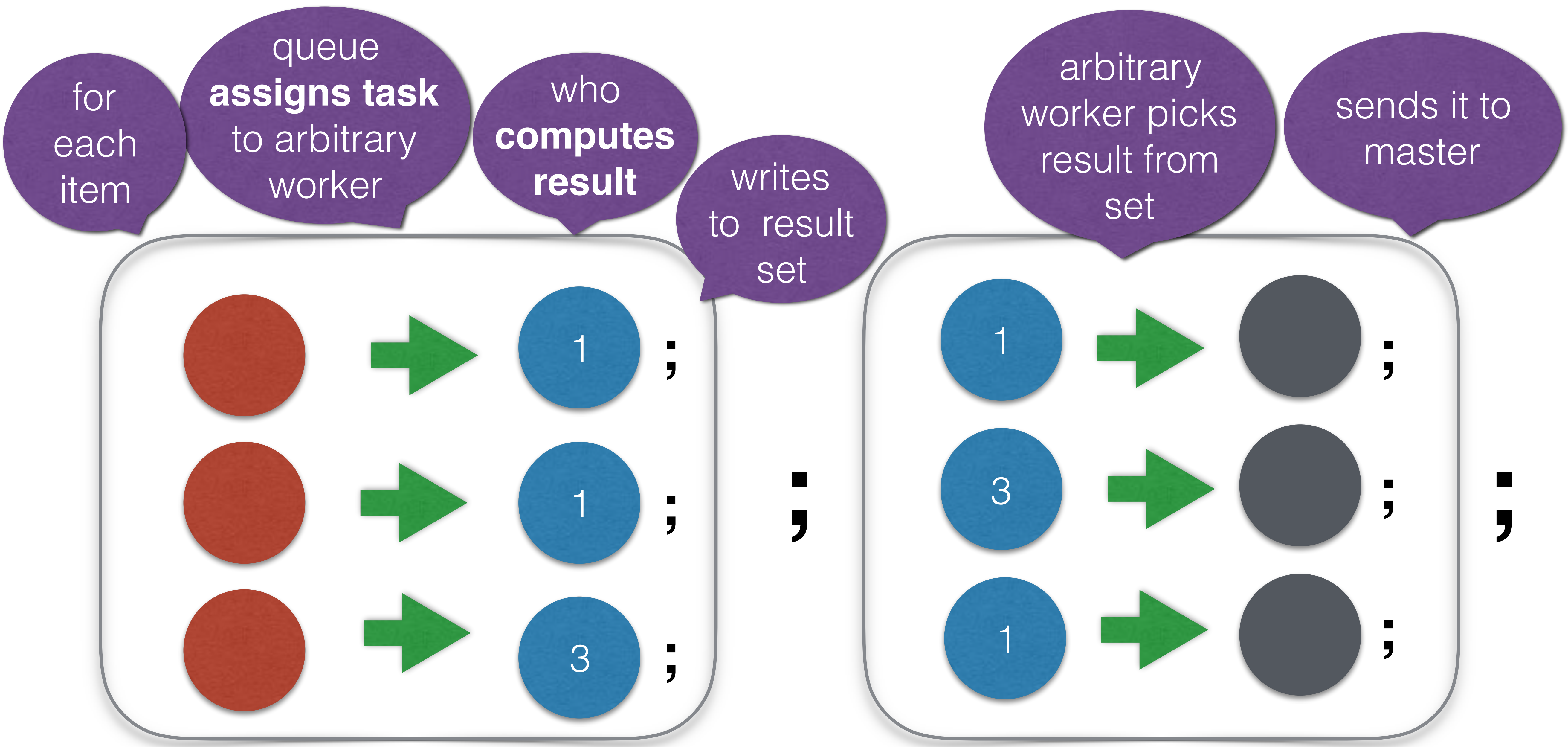
Work stealing queue



Work stealing queue



Sequentialized



How can sequentialization help verify programs?

How can sequentialization help verify programs?

no sequentialization means **likely wrong**

compute its canonical sequentialization

implies **deadlock freedom**

same **halting states**

use to prove **additional properties**

on **simpler, sequential** program

Outline

The Problems

The Key Idea

The Implementation

The Evaluation

Outline

The Problems

The Key Idea

The Implementation

The Evaluation

The Implementation

The Implementation

1. Restrict Computation Model

2. Sequentialize by Rewriting

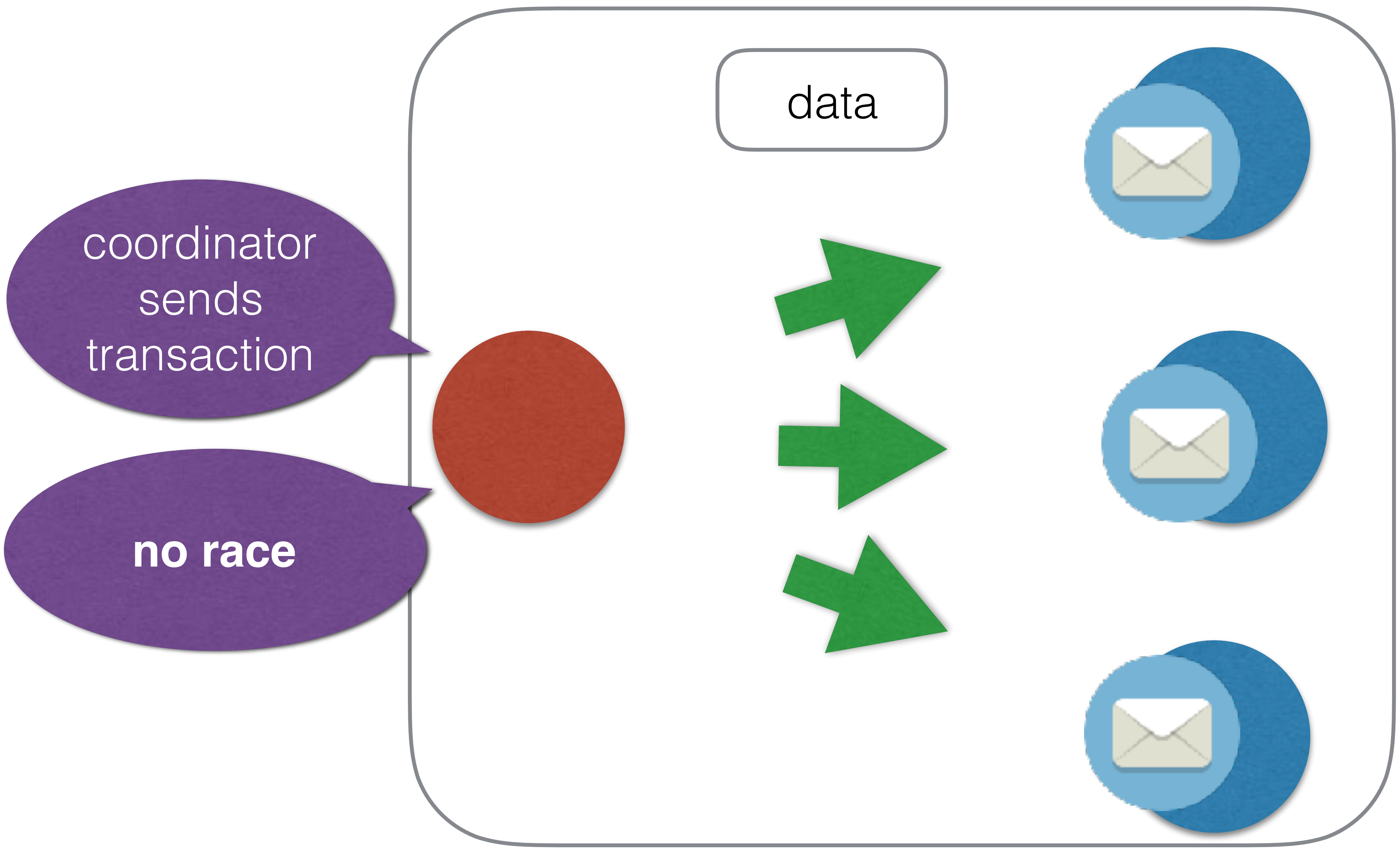
1. Restrict Computation Model

Symmetric Nondeterminism

Races yield equivalent outcomes

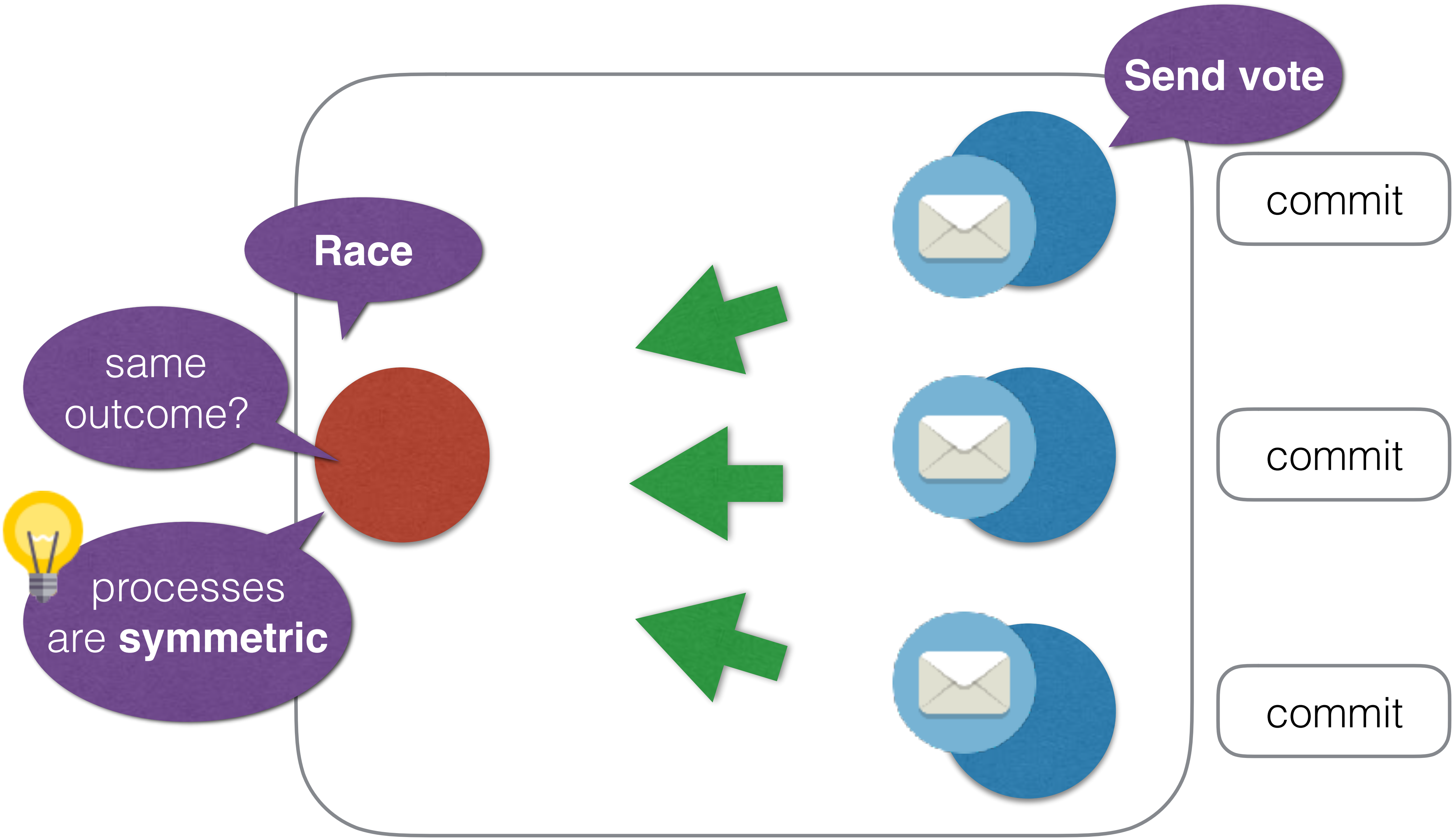
Symmetric Nondeterminism

Example: Phase 1 of 2PC



Symmetric Nondeterminism

Example: Phase 1 of 2PC



Symmetry

Symmetry means **invariance** under transformation

invariant under rotation

not this one

look at from above

Symmetry

In Distributed Systems

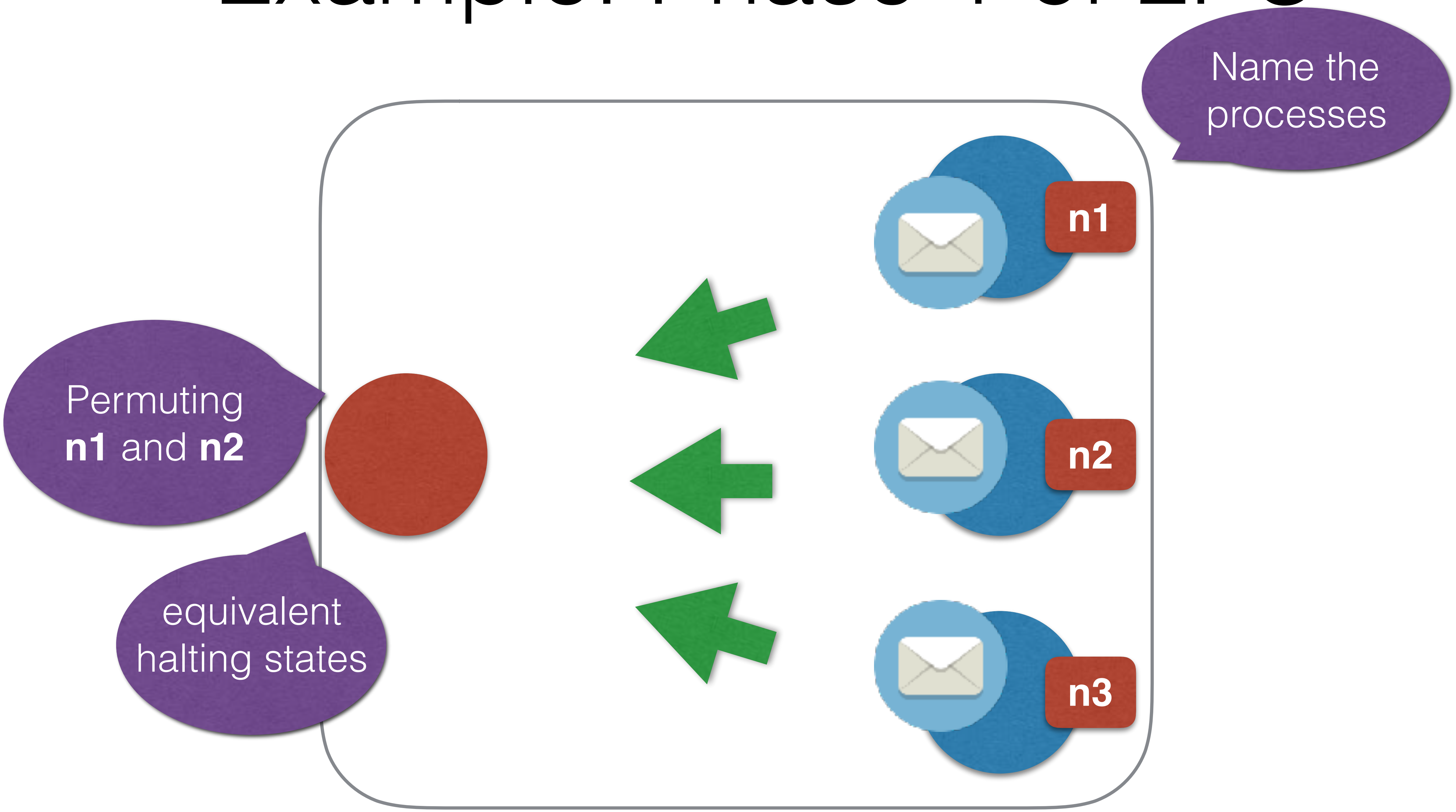
[Norris and Dill 1996]

Permuting Process Identifiers

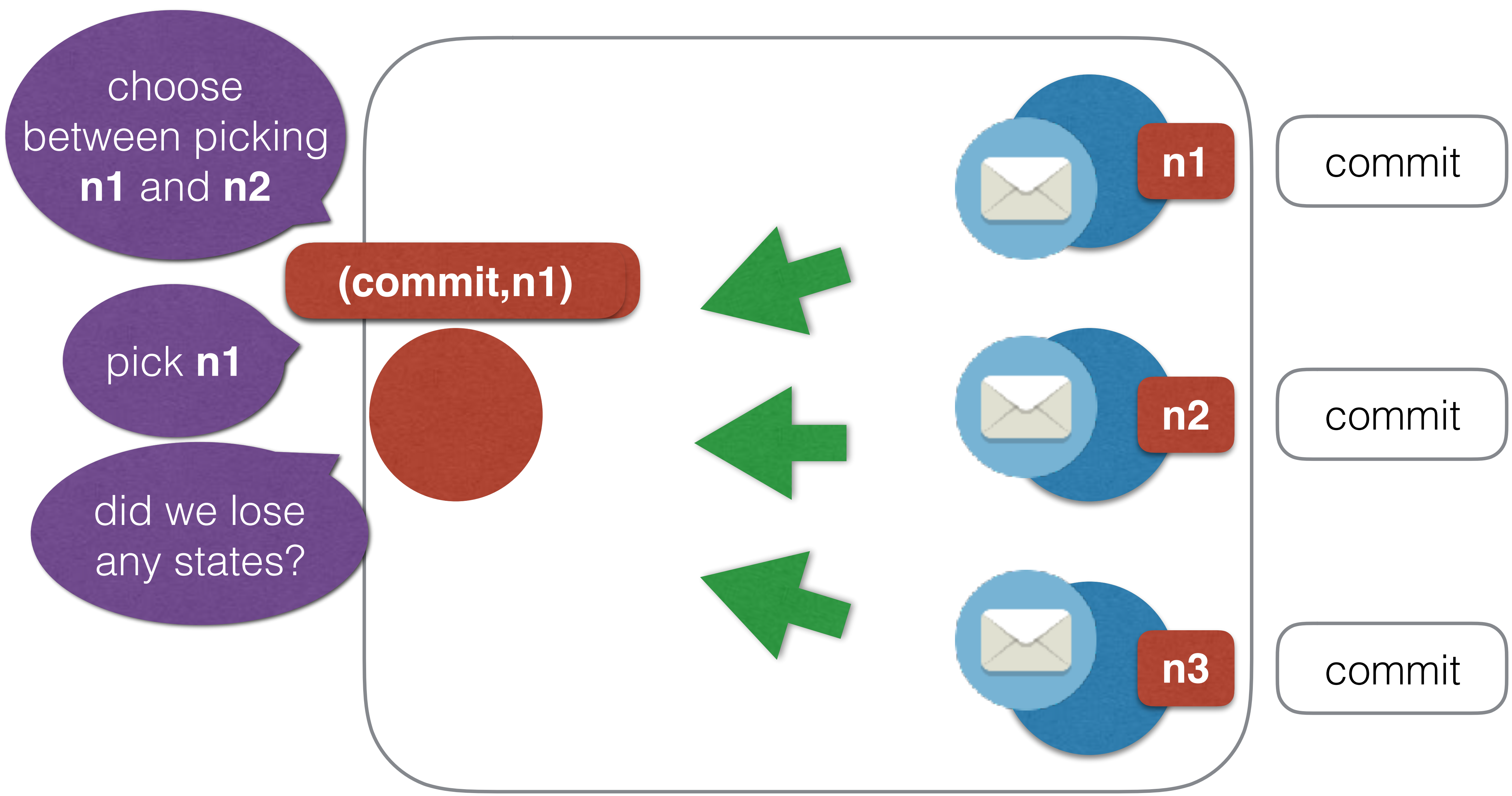
Yields equivalent halting states

Symmetry

Example: Phase 1 of 2PC

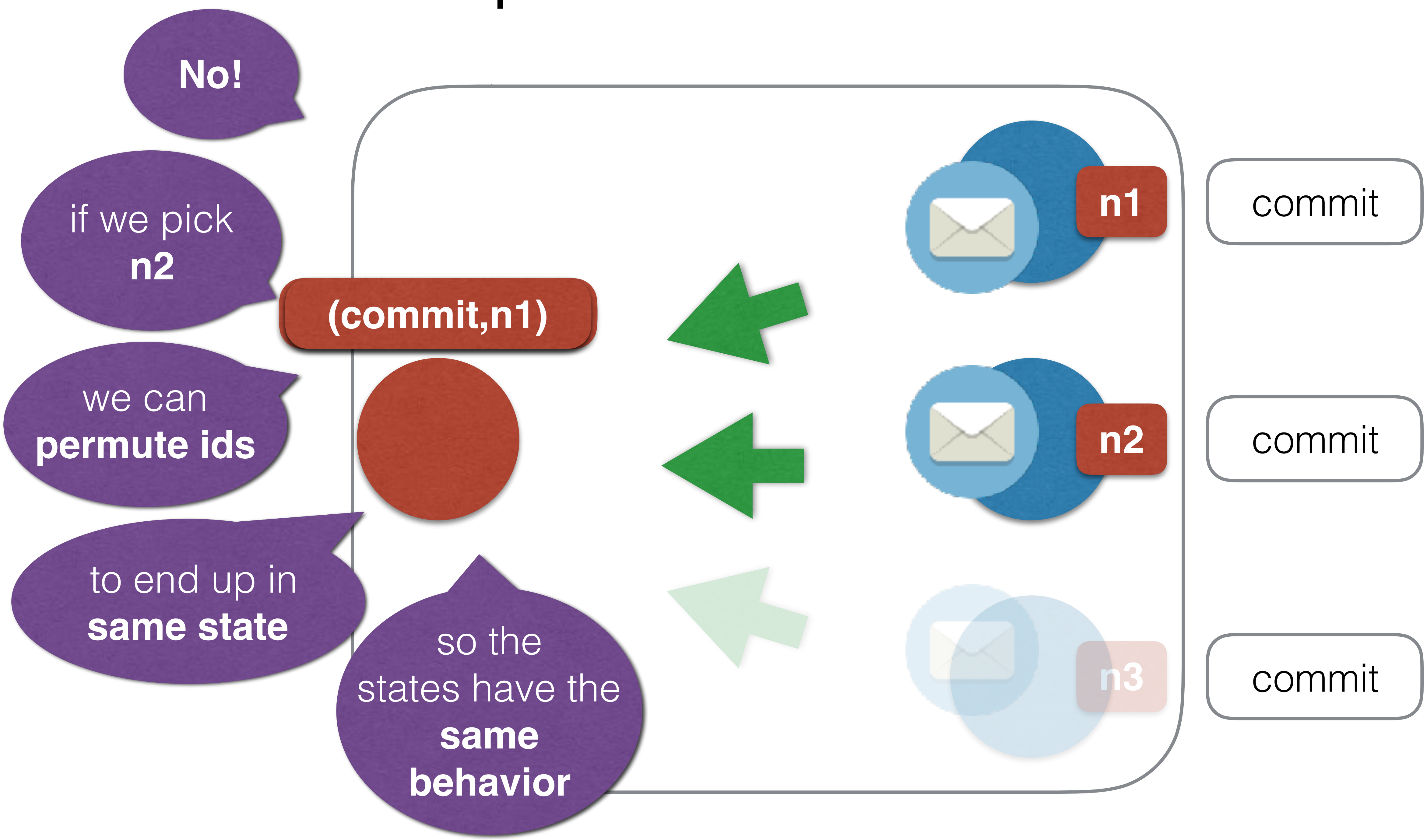


Symmetric Nondeterminism Example: Phase 1 of 2PC



Symmetric Nondeterminism

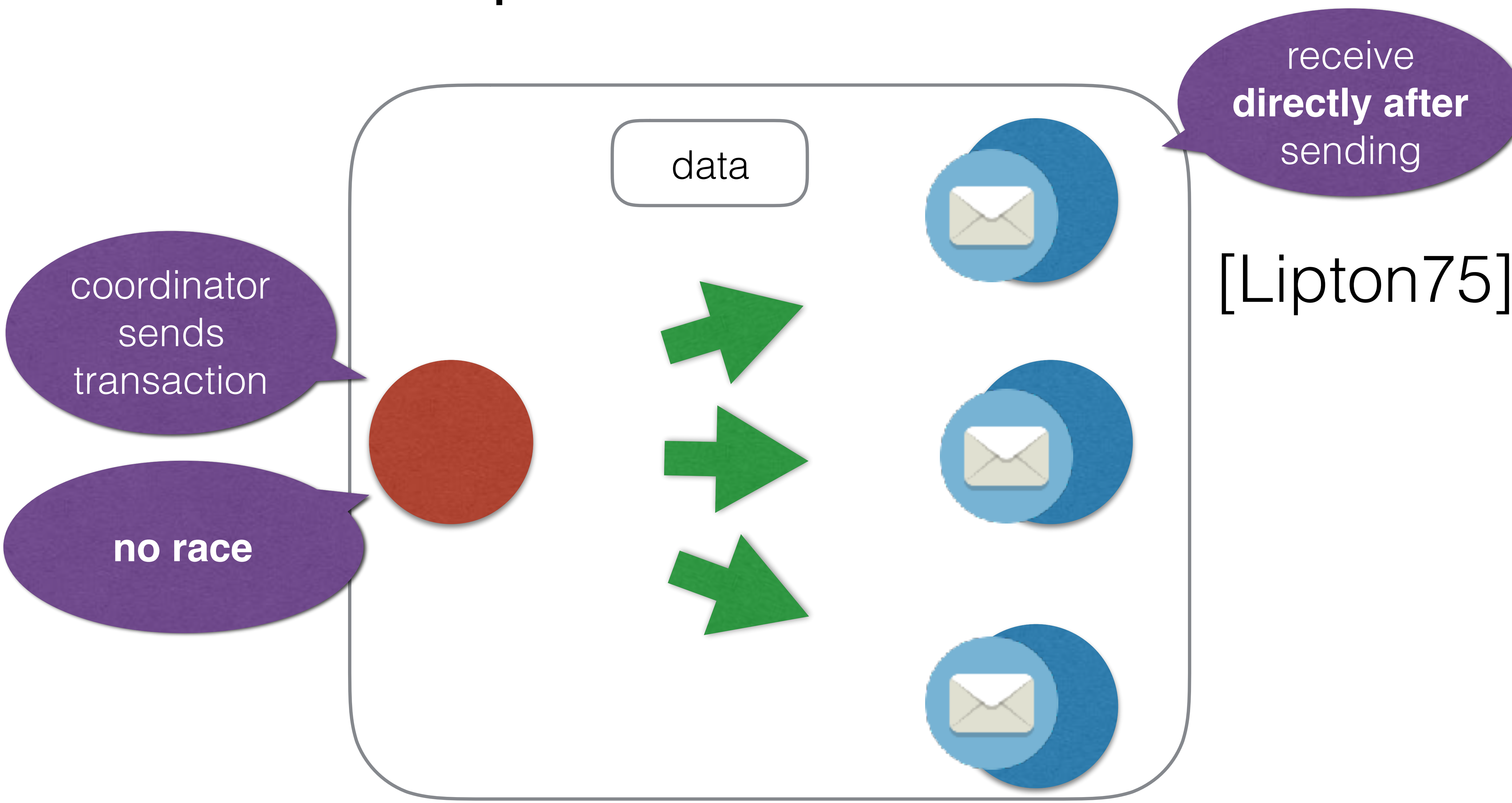
Example: Phase 1 of 2PC



How can we use symmetry to sequentialize?

Symmetric Nondeterminism

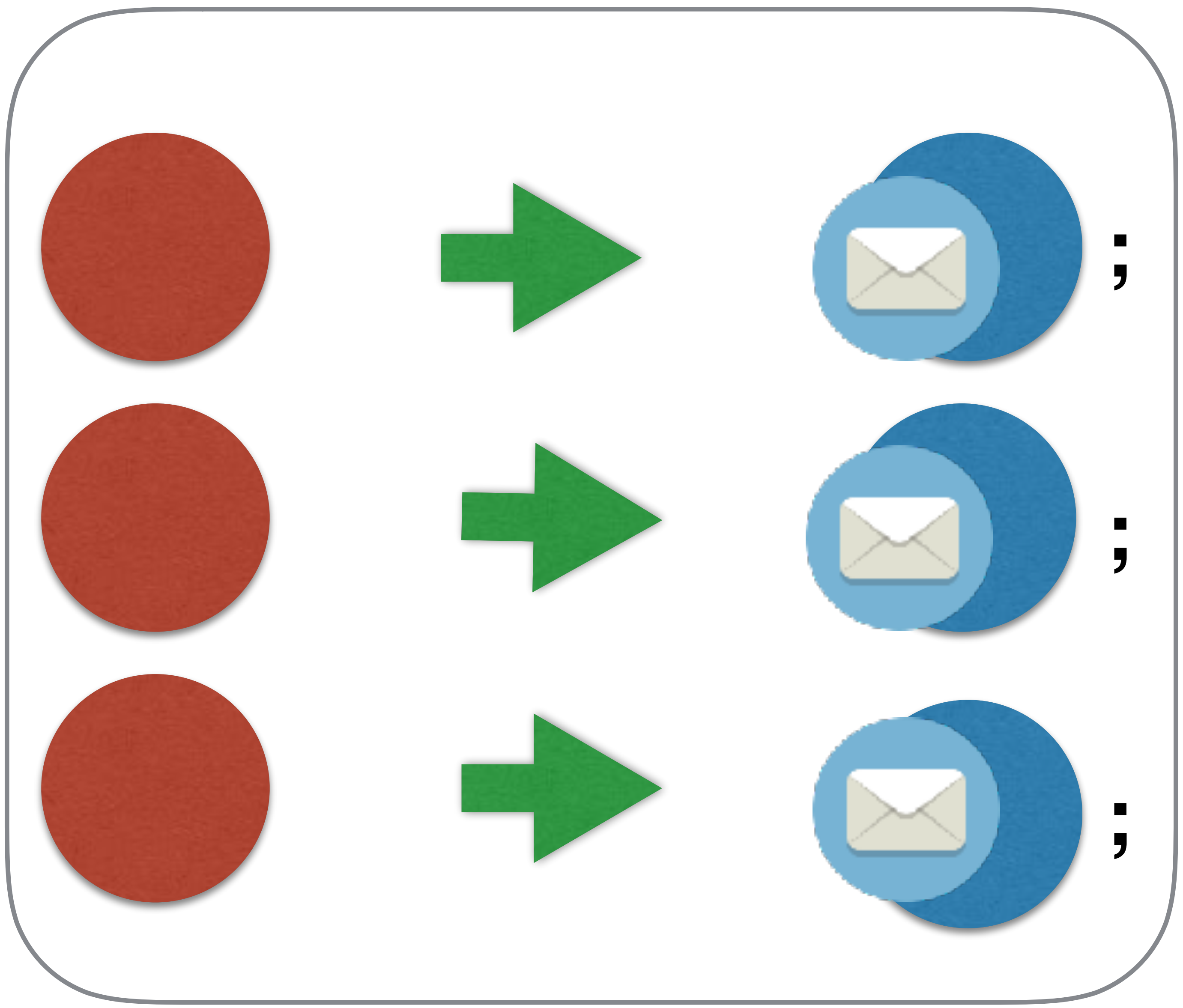
Example: Phase 1 of 2PC



[Lipton75]

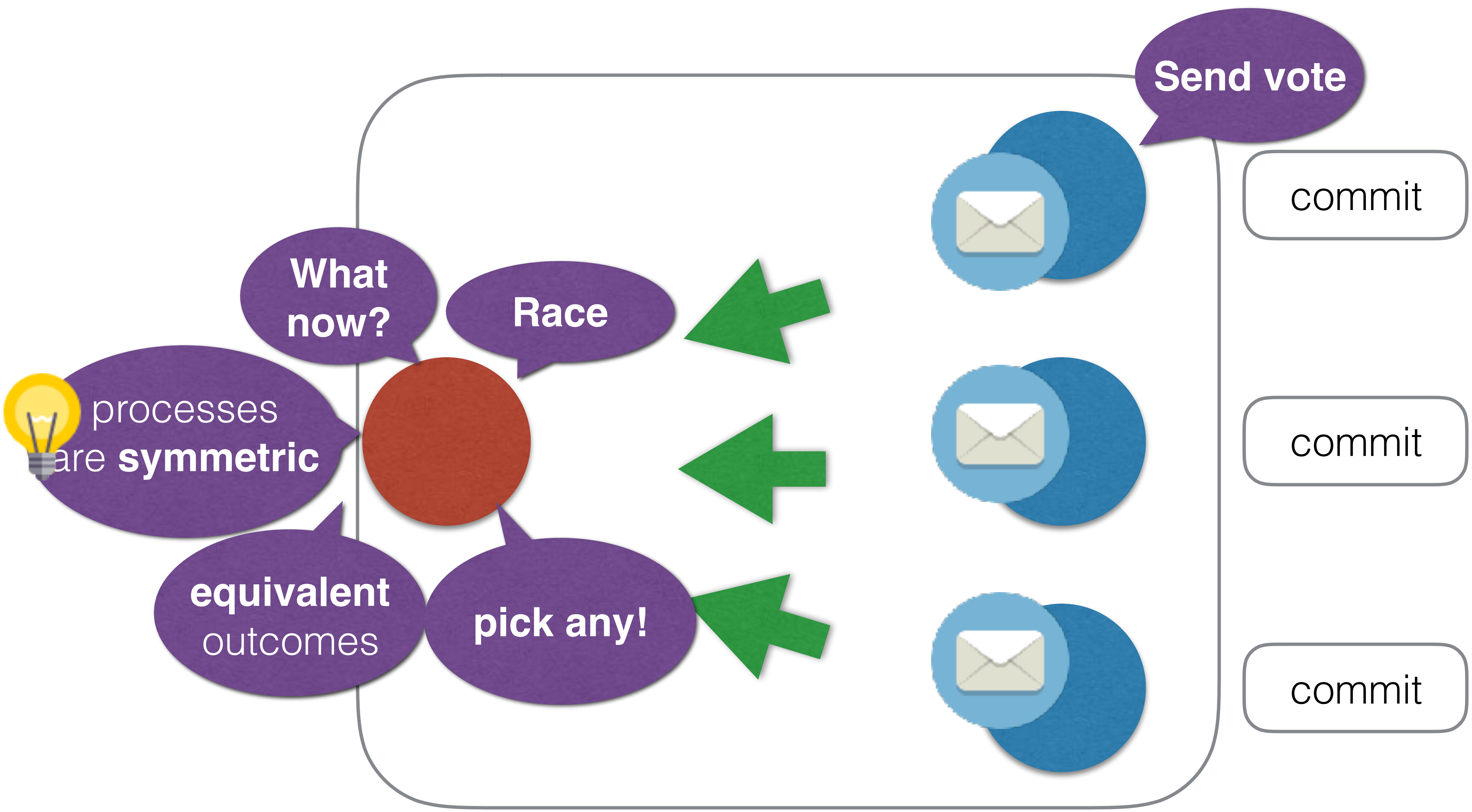
Symmetric Nondeterminism

Example: Phase 1 of 2PC



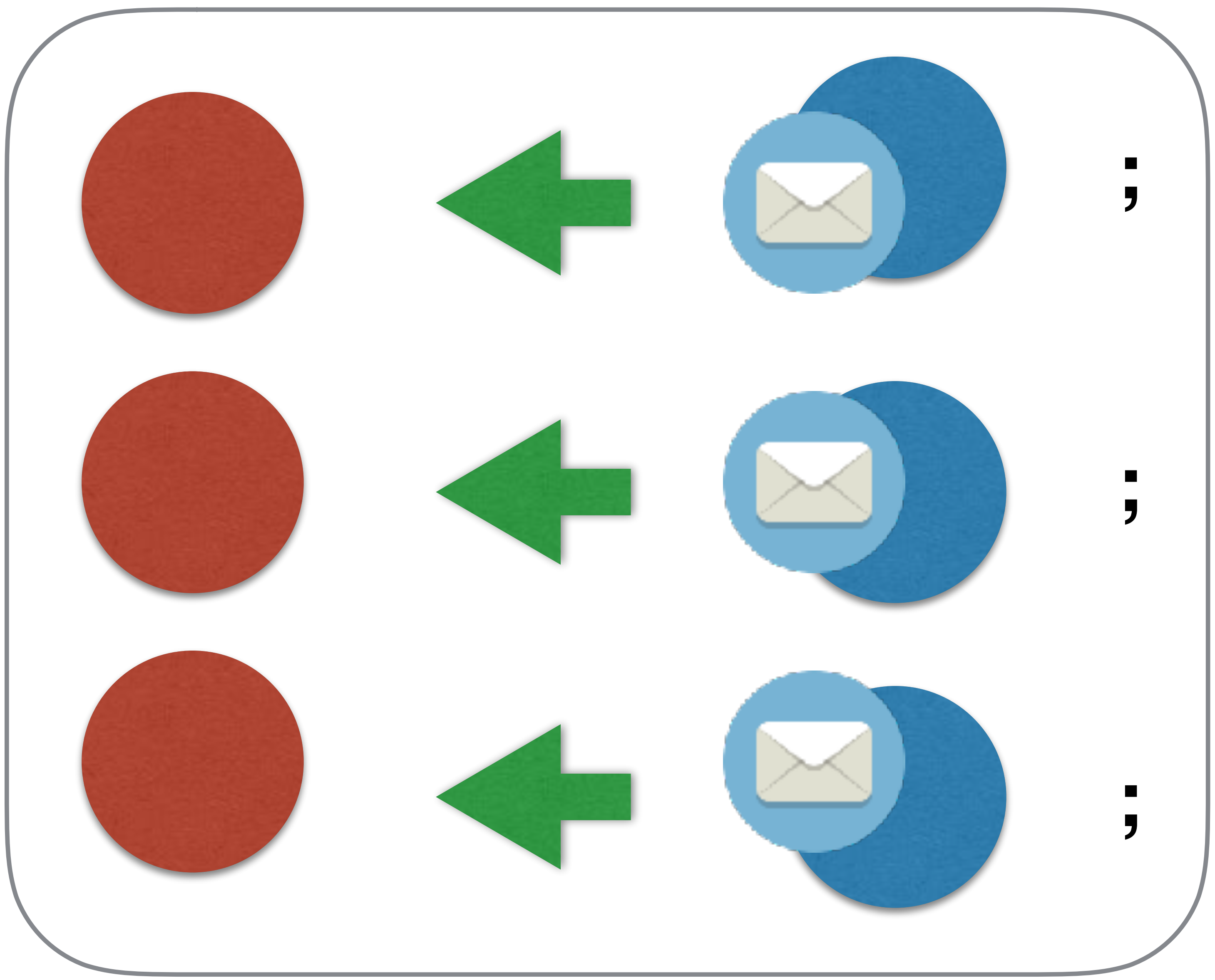
Symmetric Nondeterminism

Example: Phase 1 of 2PC



Symmetric Nondeterminism

Example: Phase 1 of 2PC



The Implementation

1. Restrict Computation Model

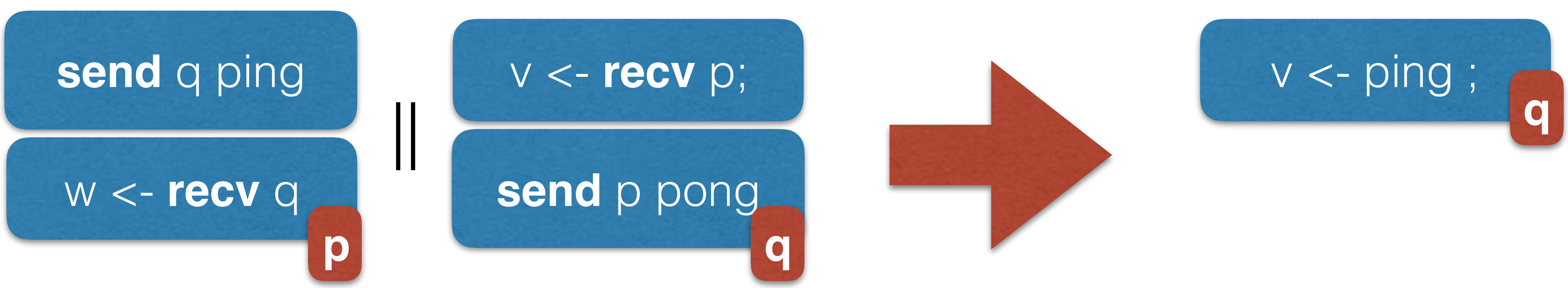
2. Sequentialize by Rewriting

2. Sequentialize by Rewriting

(by example)

2. Sequentialize by Rewriting

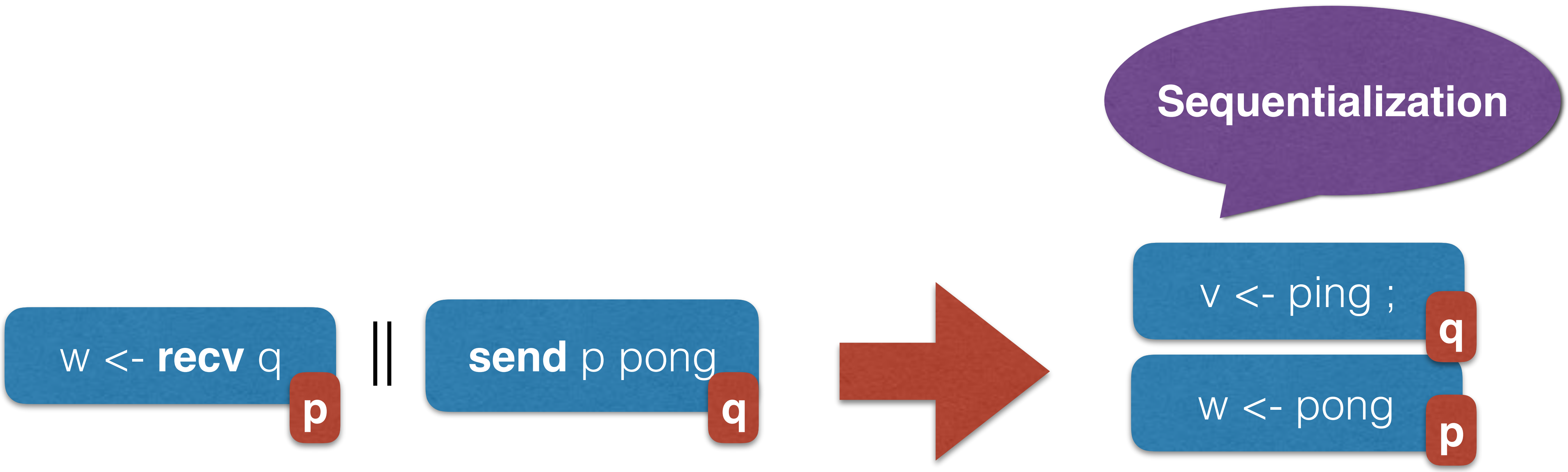
Example 1



p, q are in parallel

2. Sequentialize by Rewriting

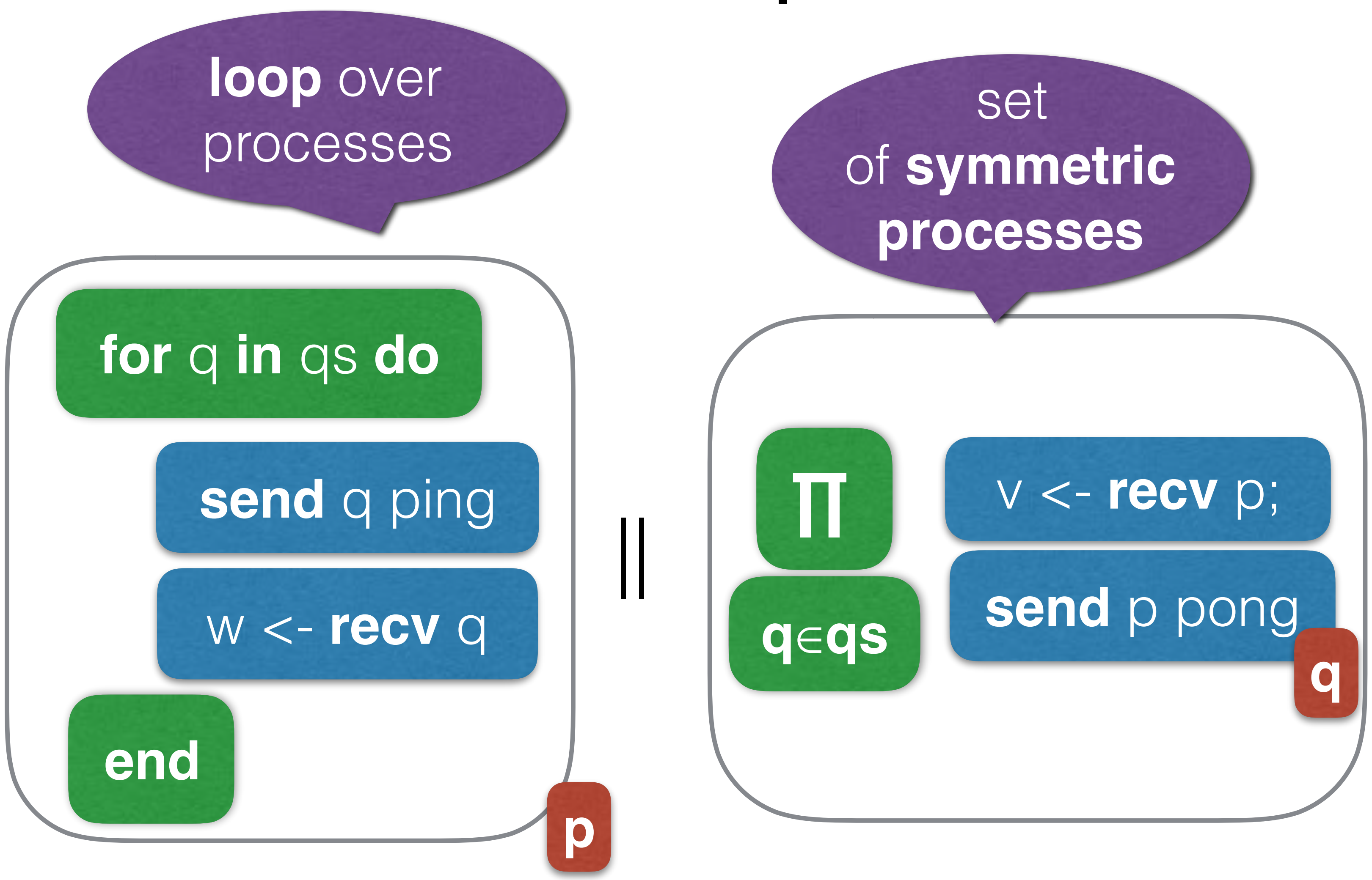
Example 1



p, q are in parallel

2. Sequentialize by Rewriting

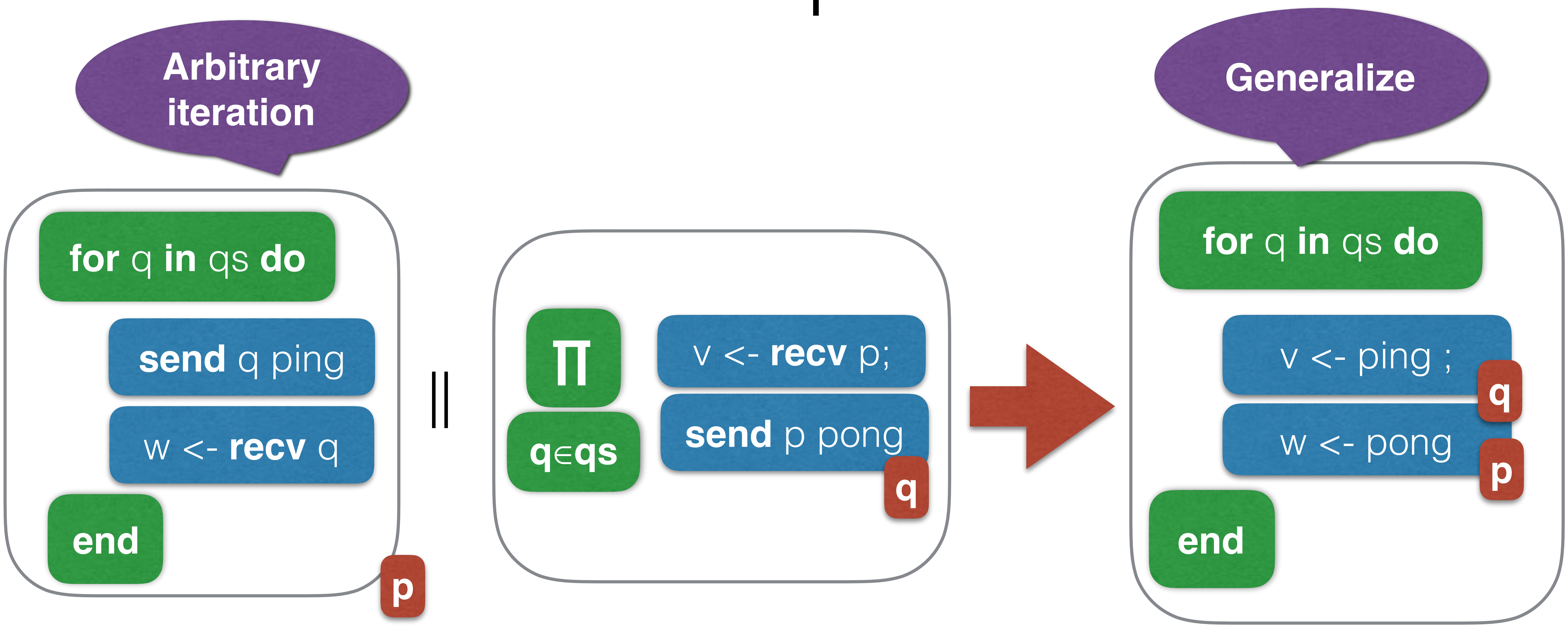
Example 2



p, qs={q1…qn} are in parallel

2. Sequentialize by Rewriting

Example 2



p, qs={q1…qn} are in parallel

2. Sequentialize by Rewriting

Example 3

two loops

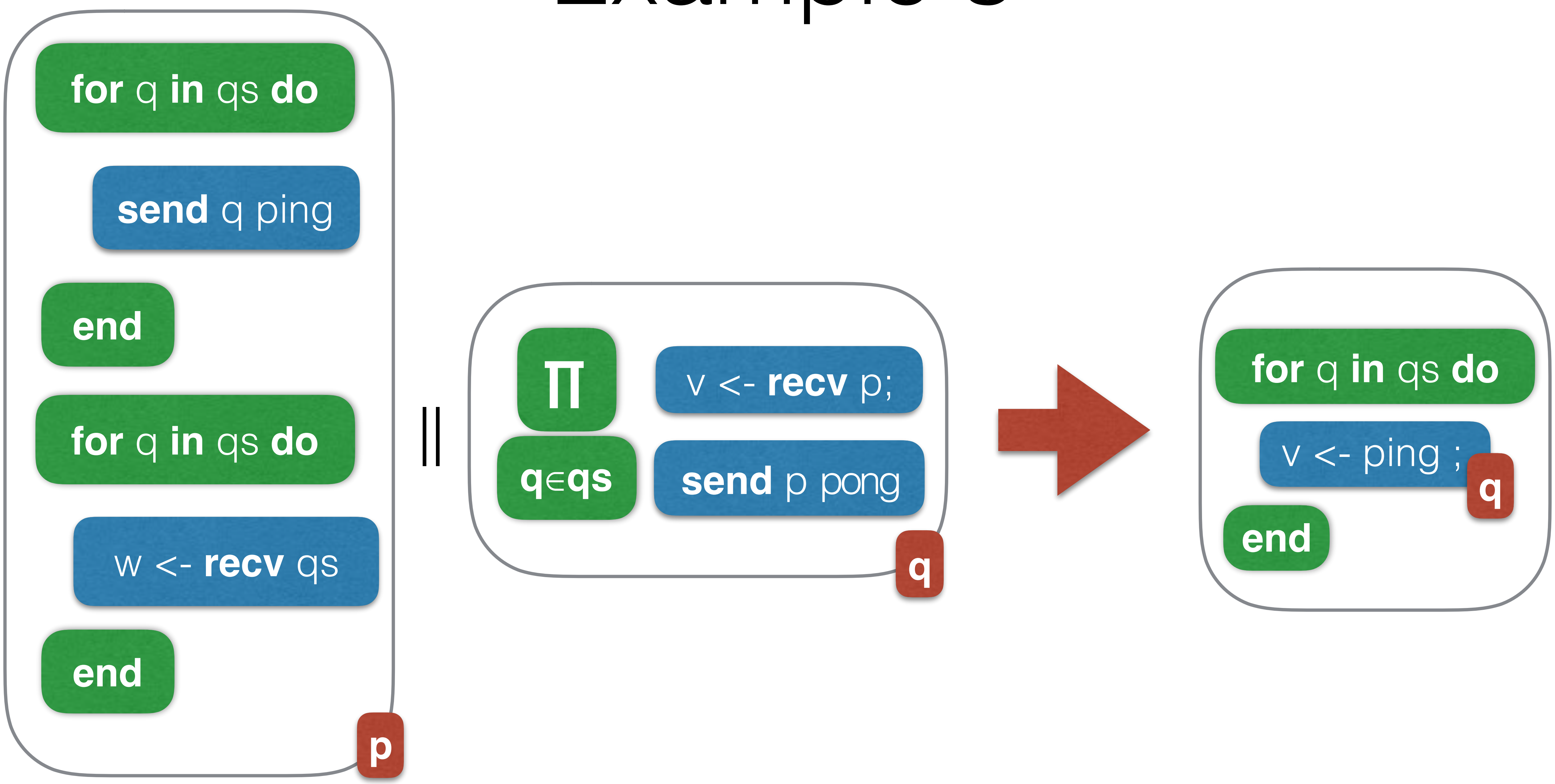
```
for q in qs do
    send q ping
end
for q in qs do
    w <- recv qs
end
```
p

||

```
Π         v <- recv p;
q∈qs      send p pong
```
q

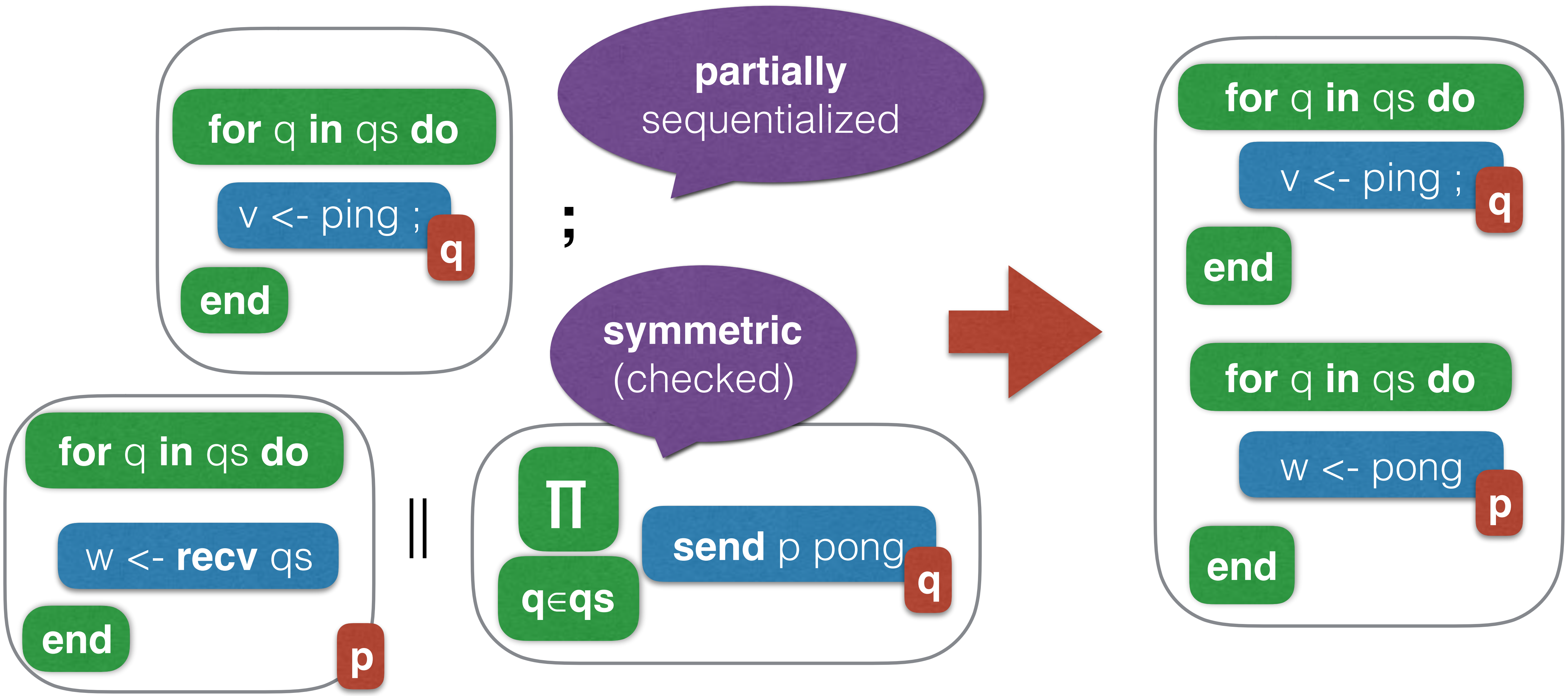
2. Sequentialize by Rewriting

Example 3



2. Sequentialize by Rewriting

Example 3



The Implementation

1. Restrict Computation Model

2. Sequentialize by Rewriting

Outline

The Problems

The Key Idea

The Implementation

The Evaluation

Outline

The Problems

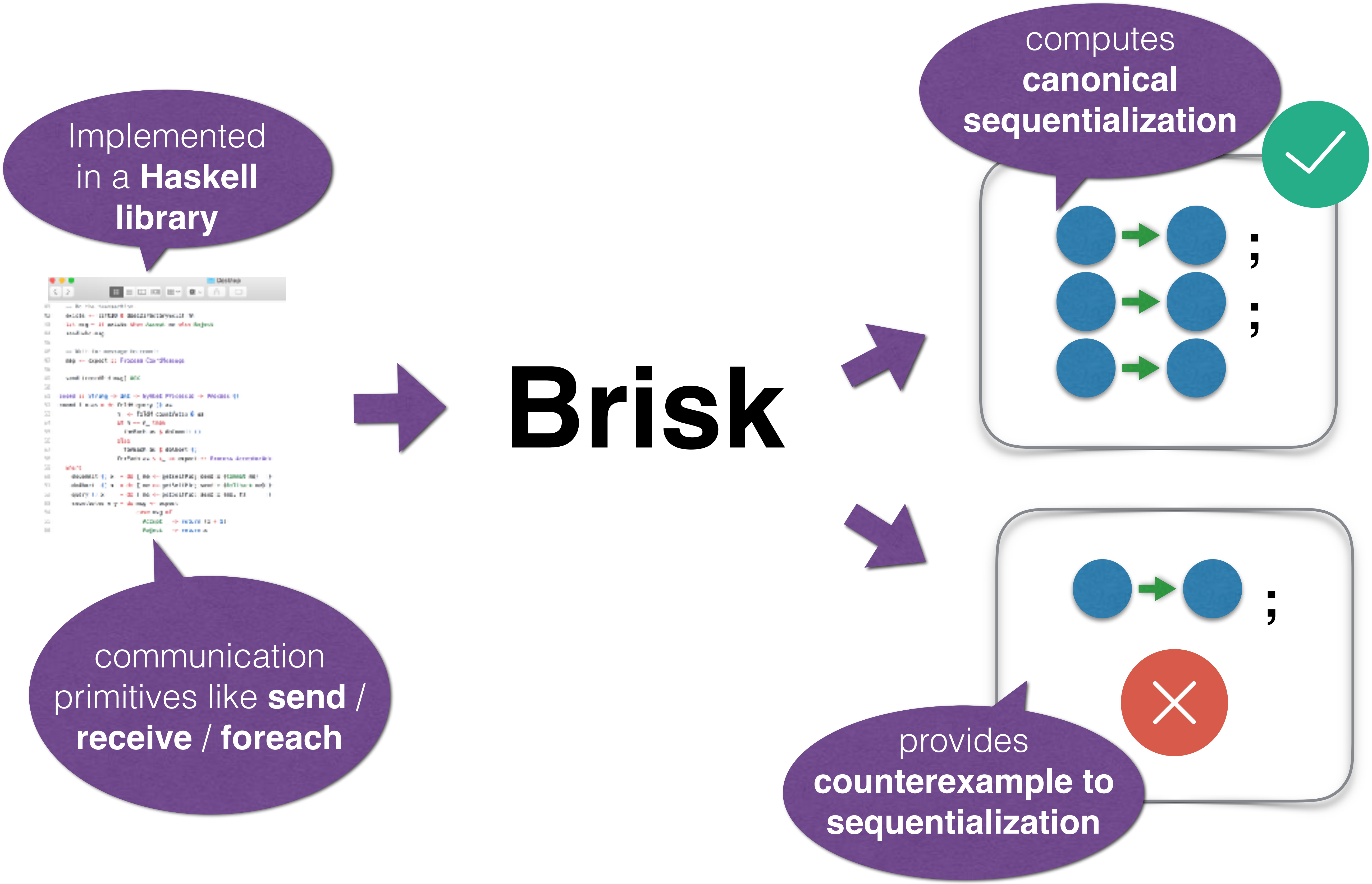
The Key Idea

The Implementation

The Evaluation

The Evaluation

The Evaluation



The Evaluation

Name	Time (ms)
ConcDB	20
DistDB	20
Firewall	30
LockServer	30
MapReduce	30
Parikh	20
Registry	30
TwoBuyers	20
2PC	50
WorkSteal	40
Theque	100

Textbook algorithms

Map/ Reduce framework

Variant of DISCO **distributed filesystem**

fast enough for interactive use

Summary

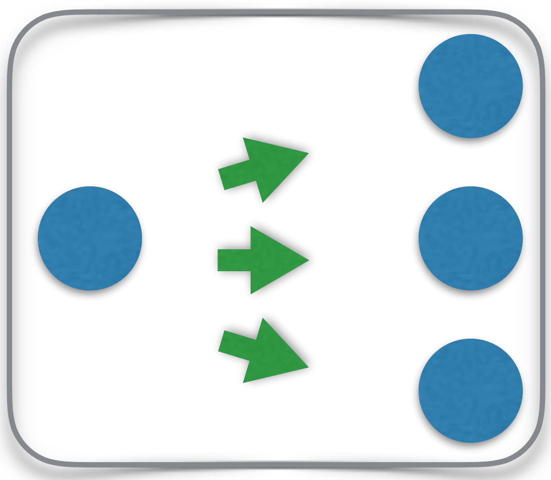
Reason about **representative sequentialization**

symmetric races produce equivalent outcomes

symmetric races + sequentialization =
verify deadlock freedom in tens of milliseconds

What's next

Faults

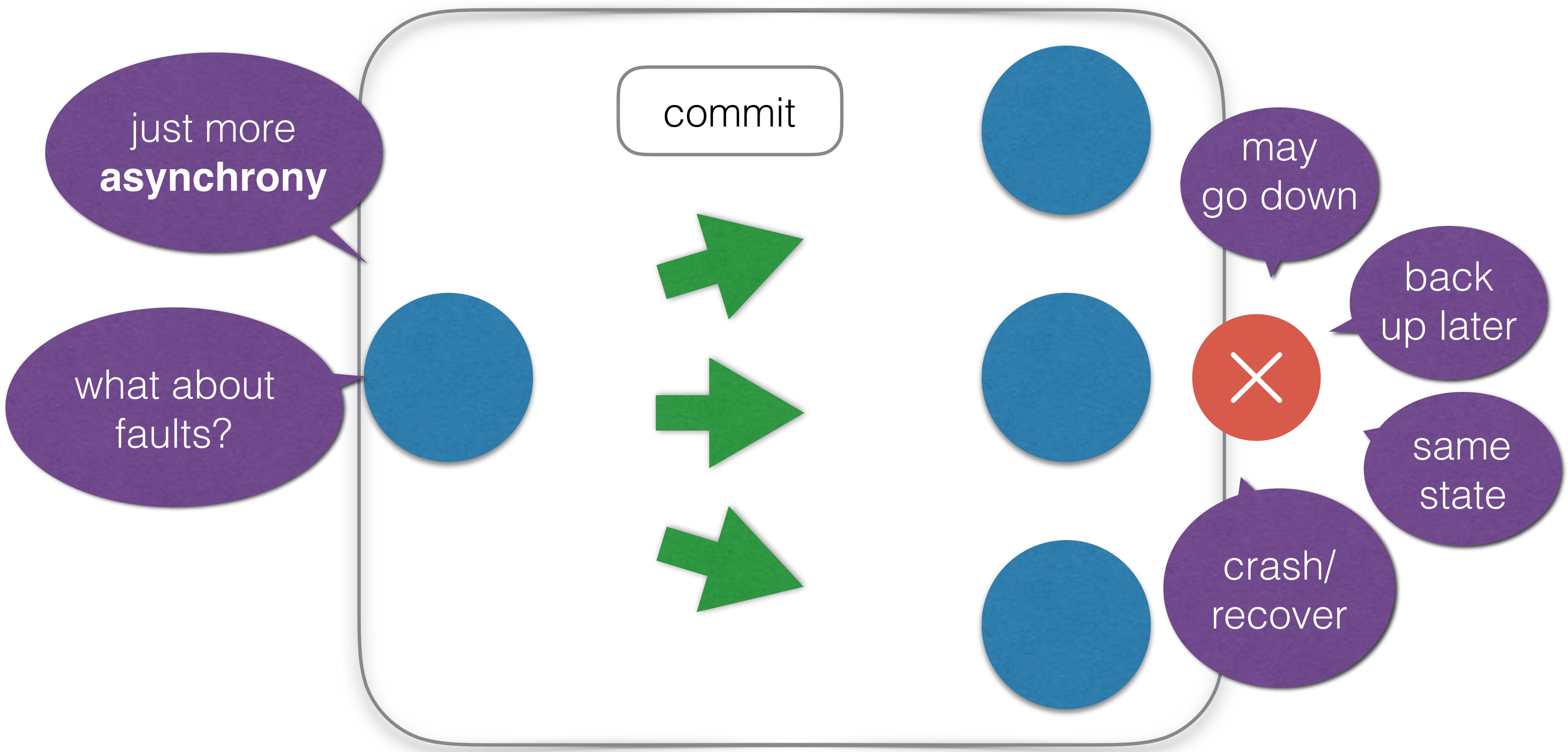


larger program class

larger class of properties

Backup slides

2PC: Faults

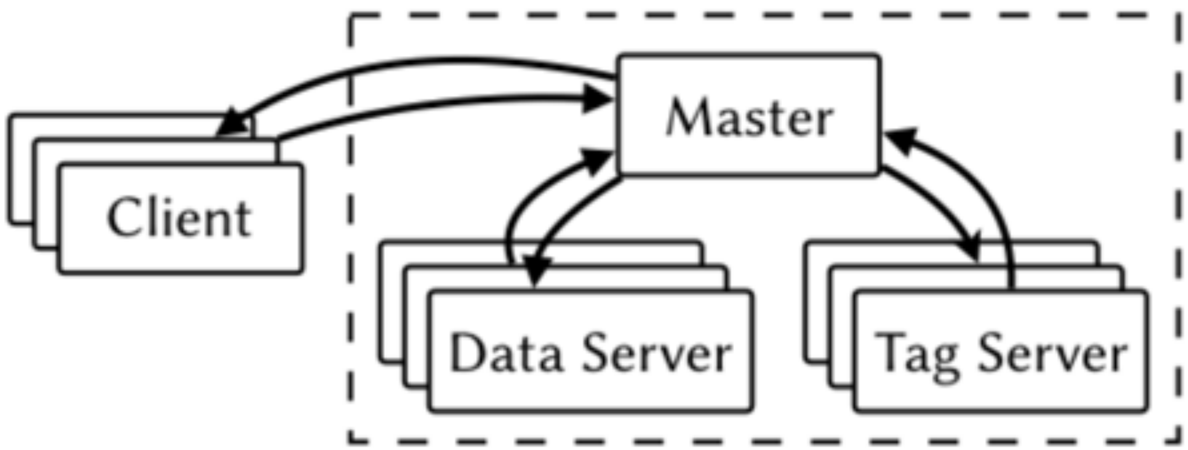

just more **asynchrony**
what about faults?
commit
may go down
back up later
same state
crash/ recover

2PC: Faults

Paxos made simple

- Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.

File System



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

mutable

Tag Server:
- AddTag(name)
- GetTag(tag)

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

immutable

Real consequences

https://bugzilla.xamarin.com/show_bug.cgi?id=42665

What happened to the bug?

however we have yet to reproduce the issue in 4.8.0.483.

Normally, 100,000 runs would hang 20% of the runs.

So far, 100,000 runs has produced no hangs.

This issue still occurs in mono-4.6.2.16

I should be more confident in a few more weeks

(fingers are still crossed right now).

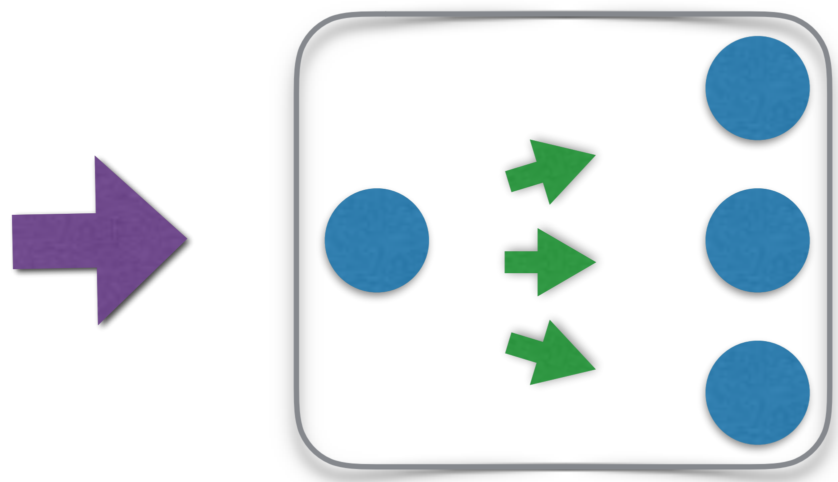
https://bugzilla.xamarin.com/show_bug.cgi?id=42665

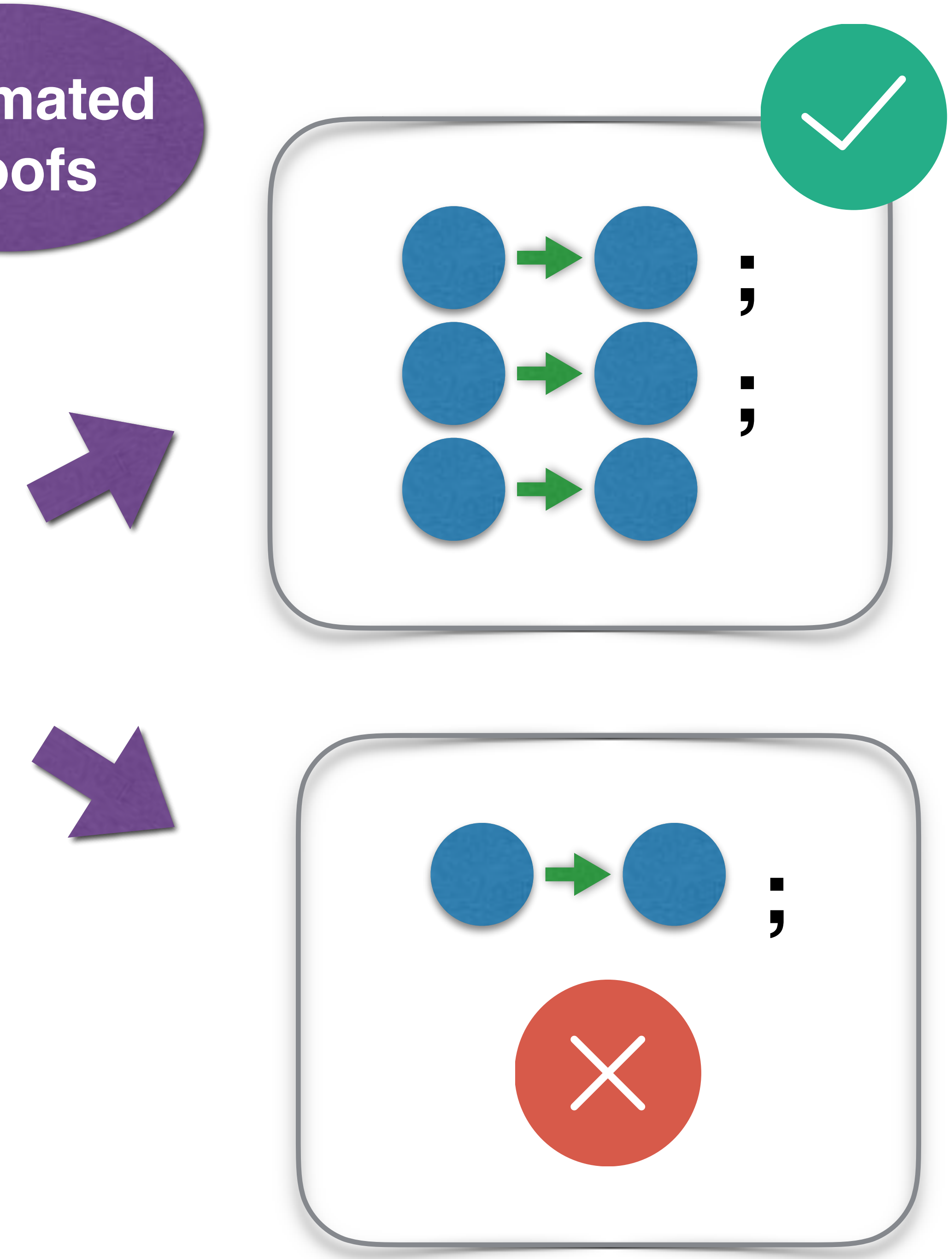
Summary

Programmers **don't case split** on execution orders

fast ~20-100 ms

automated proofs





Correct programs often have an equivalent **sequentialization**

Our approach: **compute sequentialization**

Help writing distributed programs

```haskell
coordPid (Commit p)   = p
coordPid (Rollback p) = p

forEach :: SymSet i -> (i -> Process a) -> Process ()
forEach xs body
  = foldM (\_ i -> body i >> return ()) () xs

acceptor :: Process ()
acceptor = do
  me            <- getSelfPid
  (who, fn)     <- expect :: Process (ProcessId, String)

  -- Do the transaction
  exists <- liftIO $ doesDirectoryExist fn
  let msg = if exists then Accept me else Reject
  send who msg

  -- Wait for message to commit
  msg <- expect :: Process CoordMessage

  send (coordPid msg) ACK
```

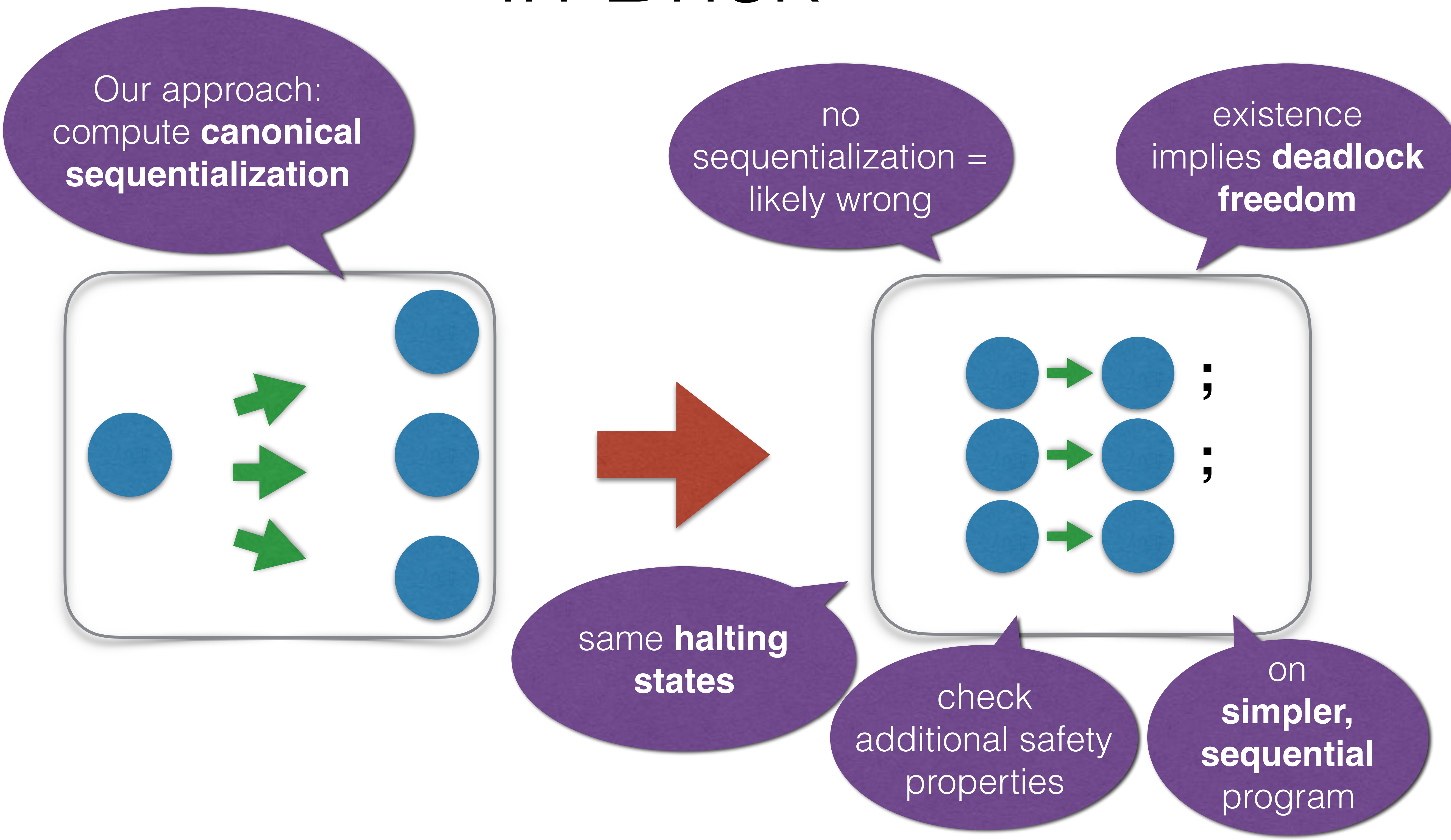
We want to

prove absence of **deadlocks**

quick feedback while compiling

no manual proofs

Canonical Sequentialization in Brisk



Results

<= 100ms

micro benchmarks

Firewall, Map Reduce, 2PC

Distributed file system

Name	#Param	#LOC	Spin N	IceT #Term	Brisk time (ms)
ex2	1	14	-	69	20
ex3	1	13	11	57	20
PingDet	1	17	13	83	20
PingIter	1	19	11	63	20
PingSym	1	13	10	44	30
PingSym2	1	43	7	140	30
ConcDB	1	54	6	265	20
DistDB	2	42	2	218	20
Firewall	1	45	9	201	30
LockServer	1	28	12	109	30
MapReduce	2	64	4	205	30
Parikh	0	35	-	173	20
Registry	1	40	10	171	30
TwoBuyers	0	59	-	332	20
2PCommit	1	47	6	281	50
WorkSteal	2	39	5	141	40
Theque	3	576	3	1443	100

Outline

The Dream

The Problems

The Key Idea

The Implementation

The Evaluation