

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...



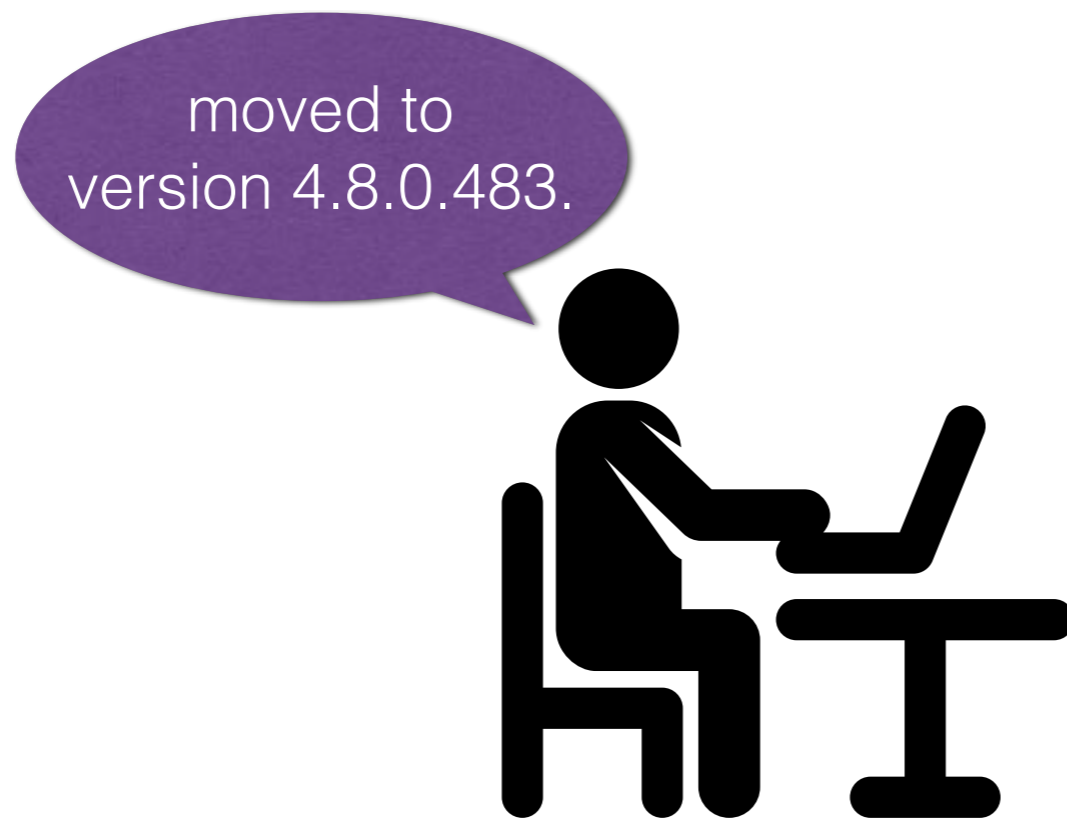
Writing distributed programs

... haunts you ...



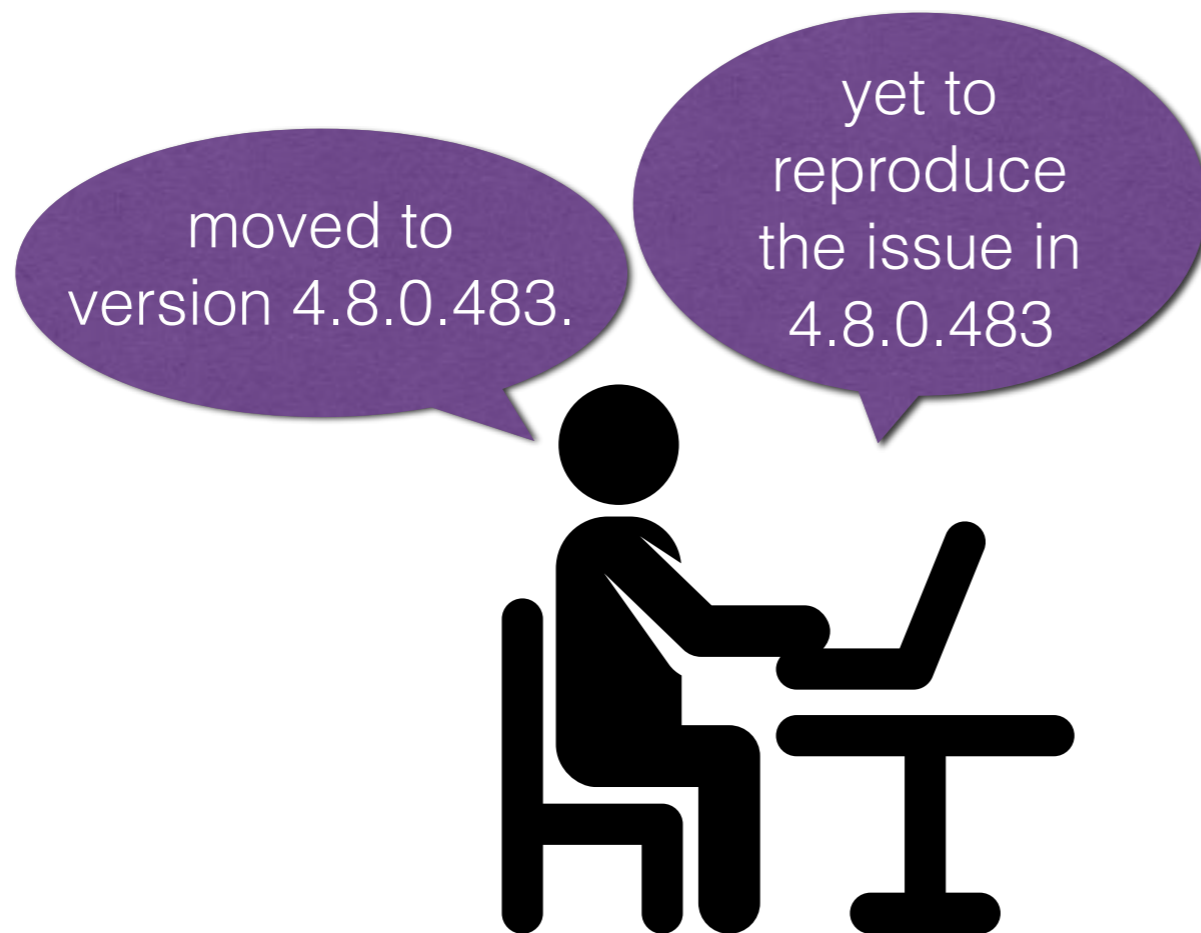
Writing distributed programs

... then you write some more code...



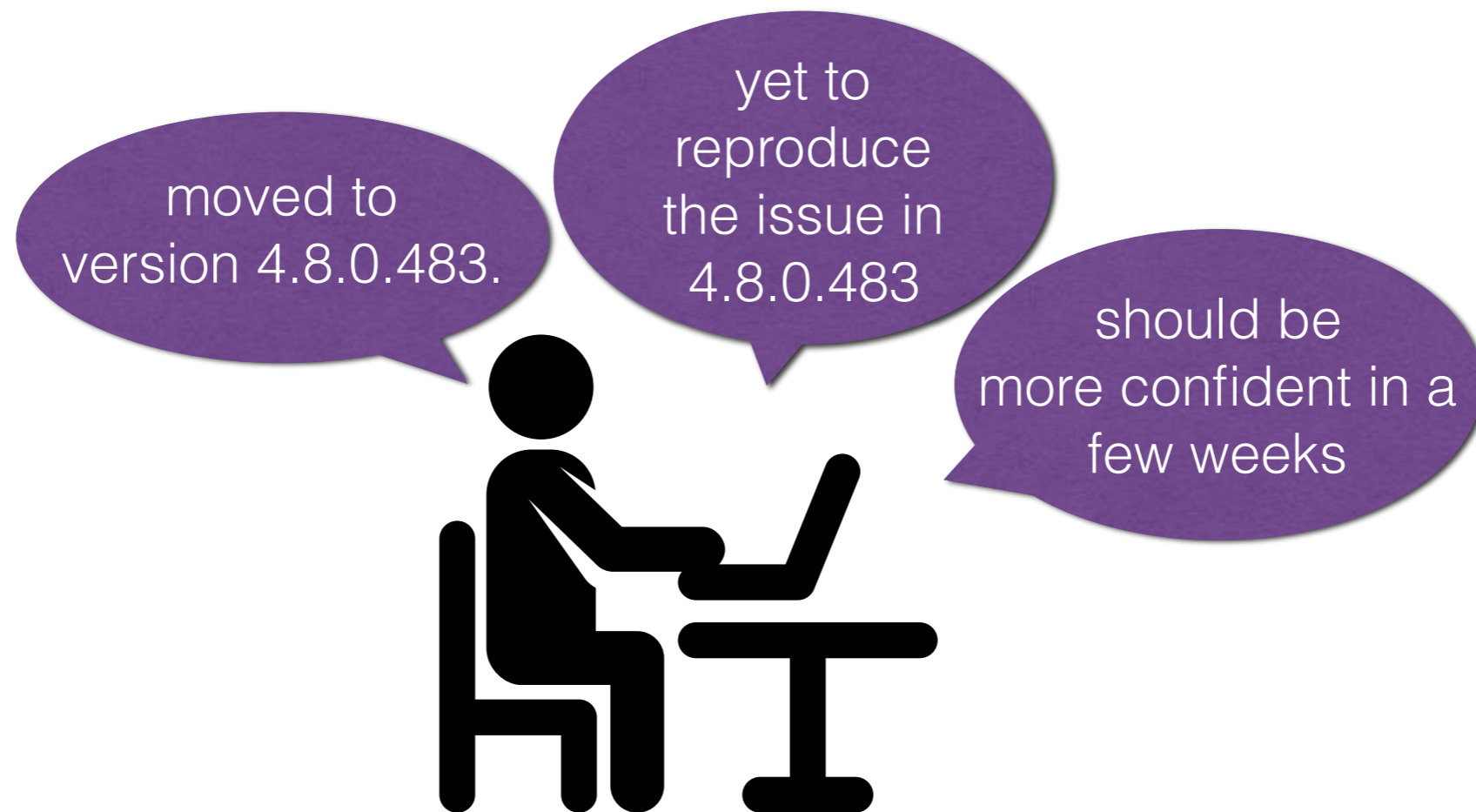
Writing distributed programs

... 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

myPid :: ProcessId -> Process ()
myPid pid = do { me <- myPid; send pid (pid, transaction) }

countVotes :: Int -> SymSet ProcessId -> Int
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
```

unmatched
receive

sent wrong
response address

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



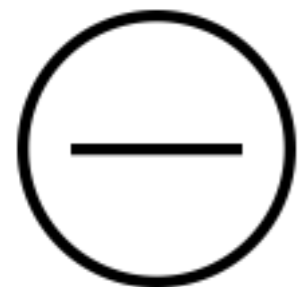
Proves absence of deadlocks



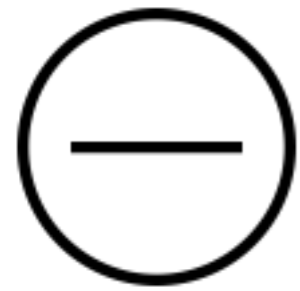
Provides counterexamples



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

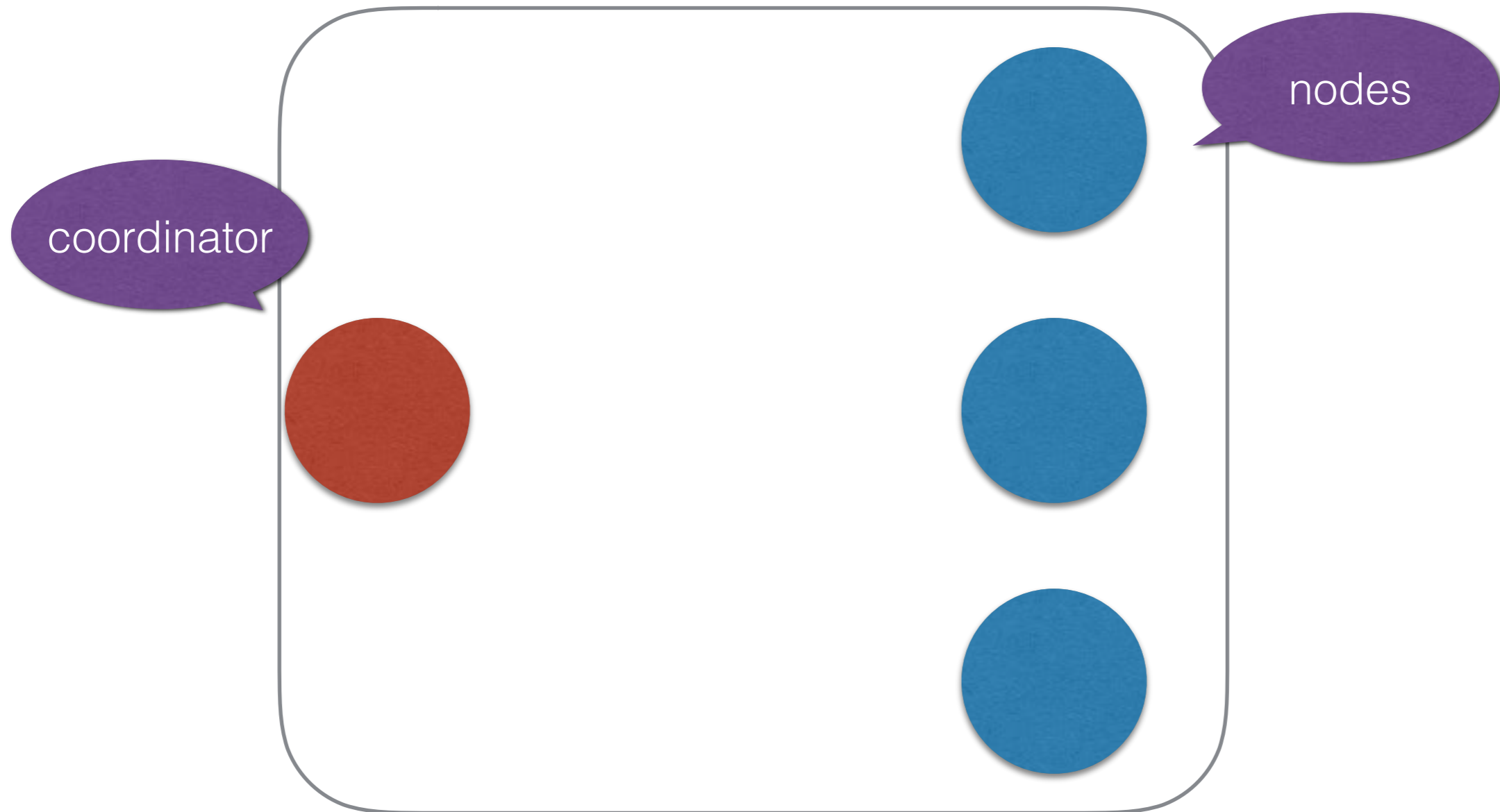
The Implementation

The Evaluation

The Problems

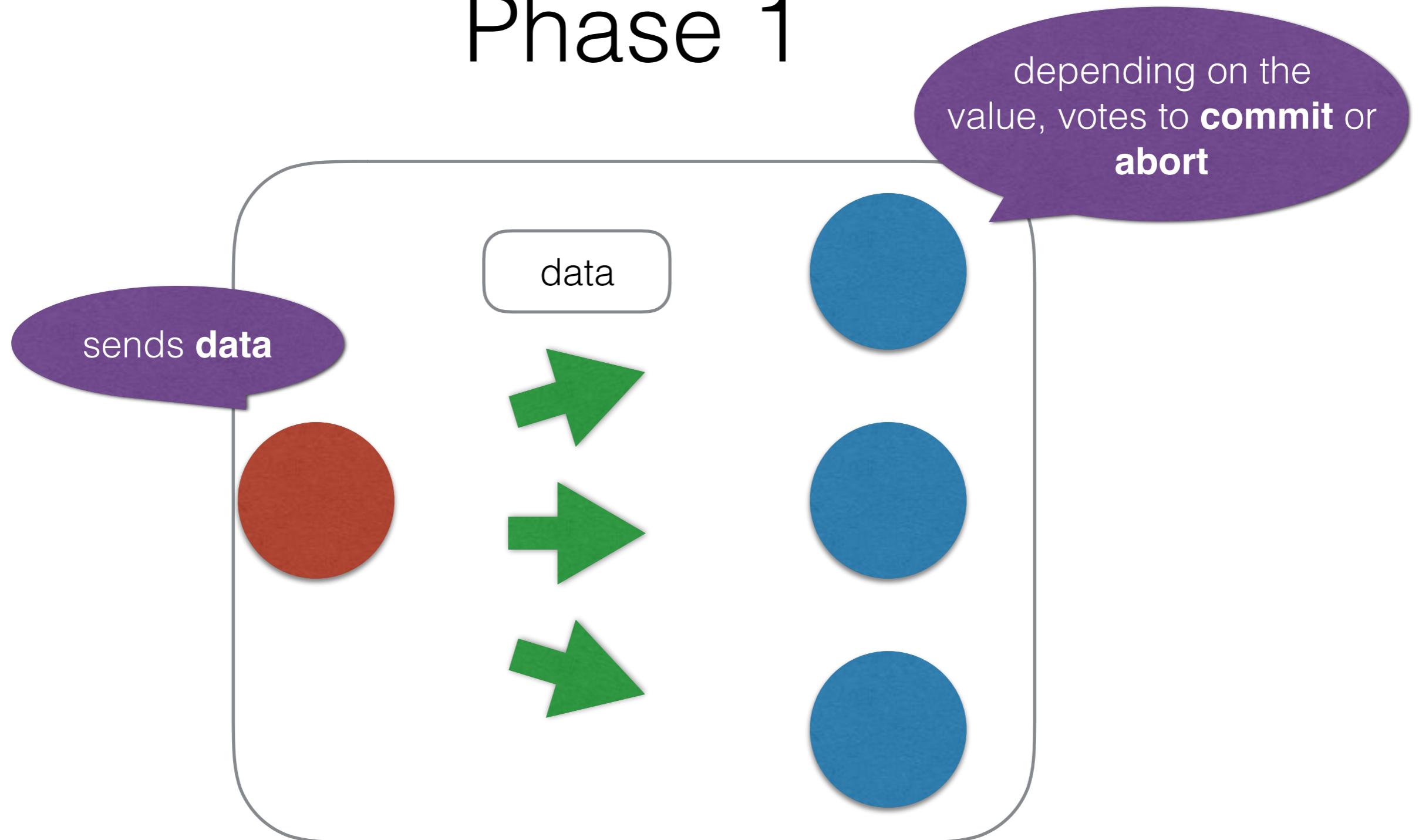
Example: Two phase commit (2PC)

Goal: Commit Transaction to all nodes



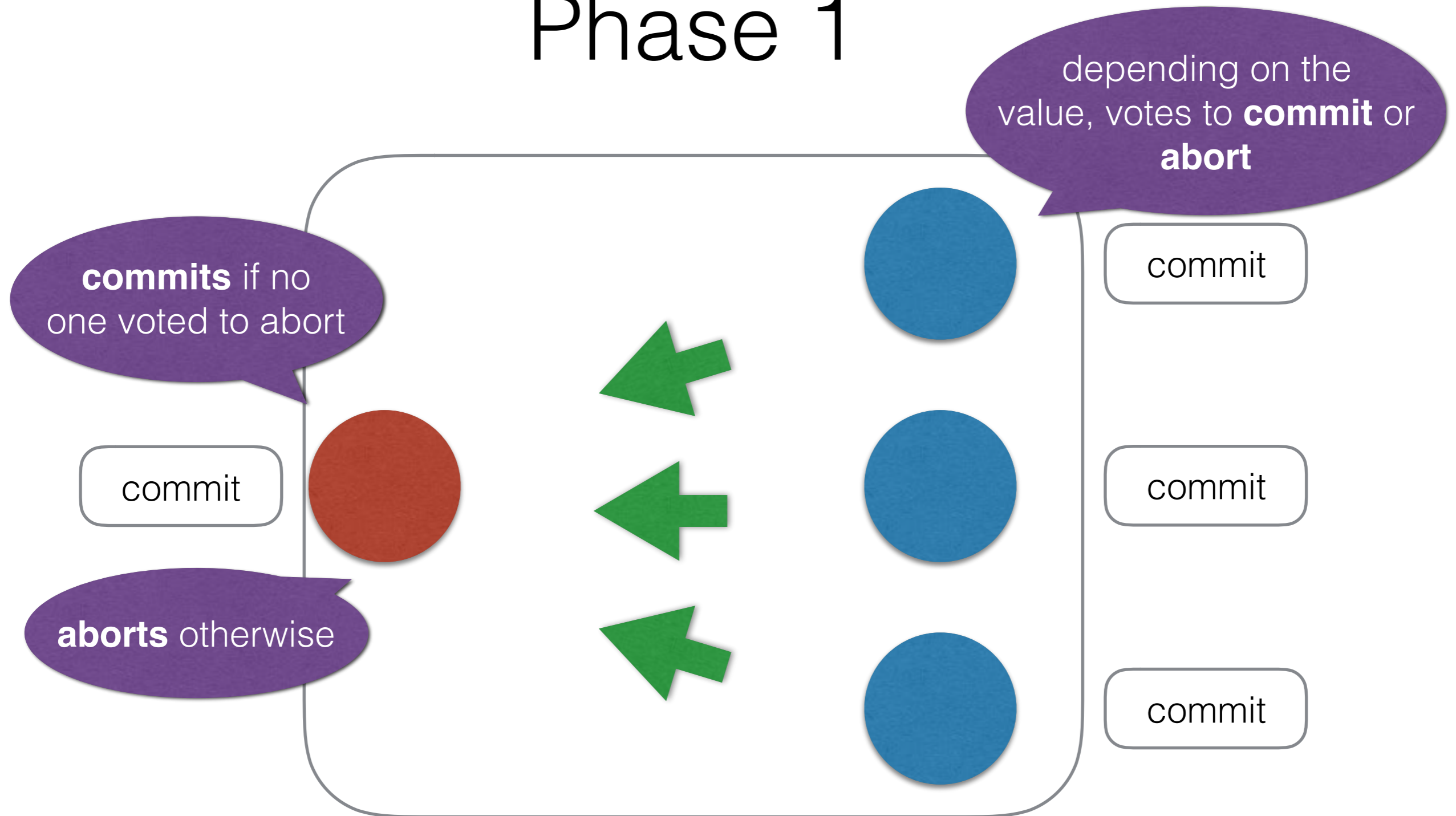
Example: Two phase commit (2PC)

Phase 1



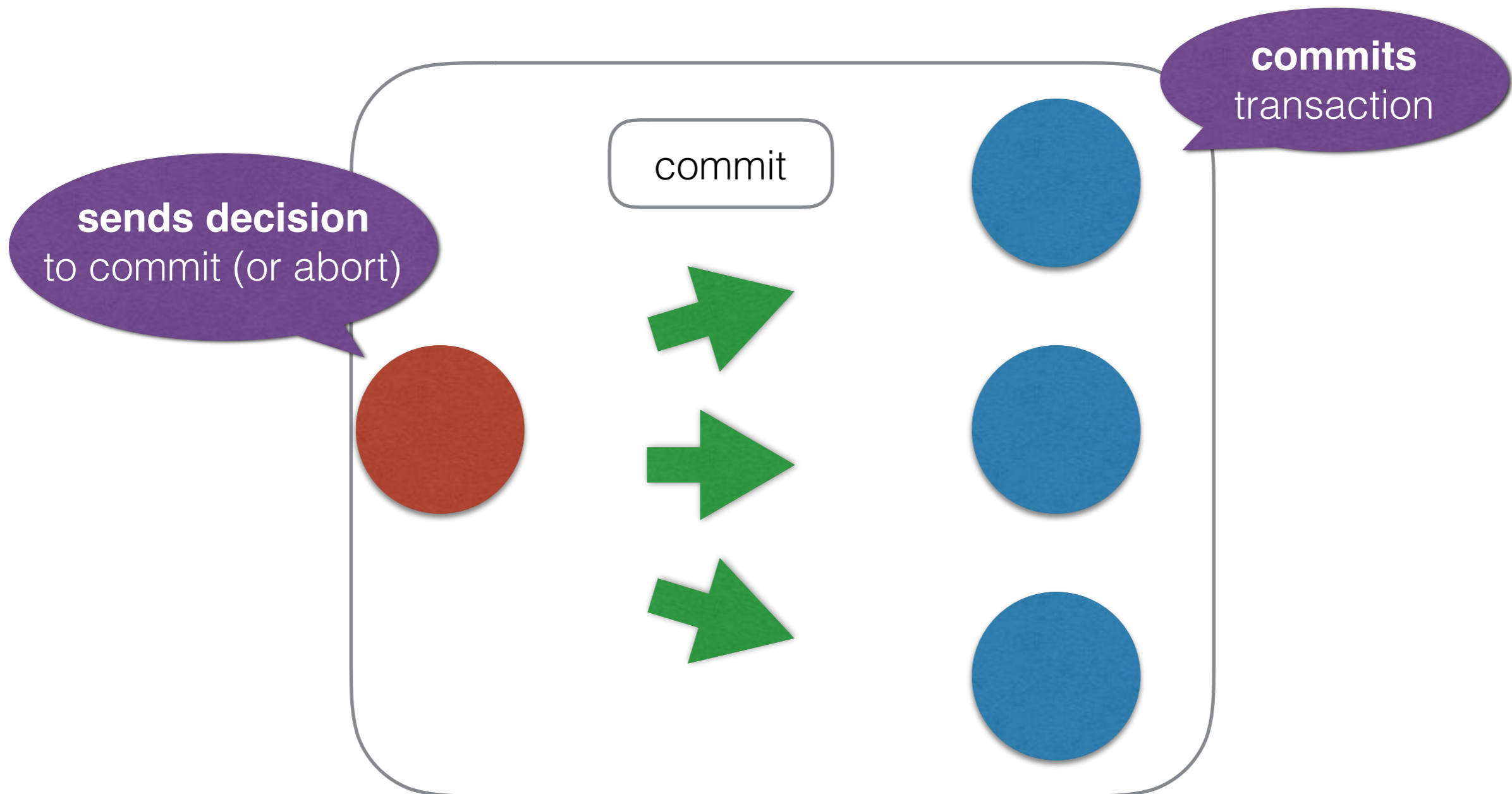
Example: Two phase commit (2PC)

Phase 1



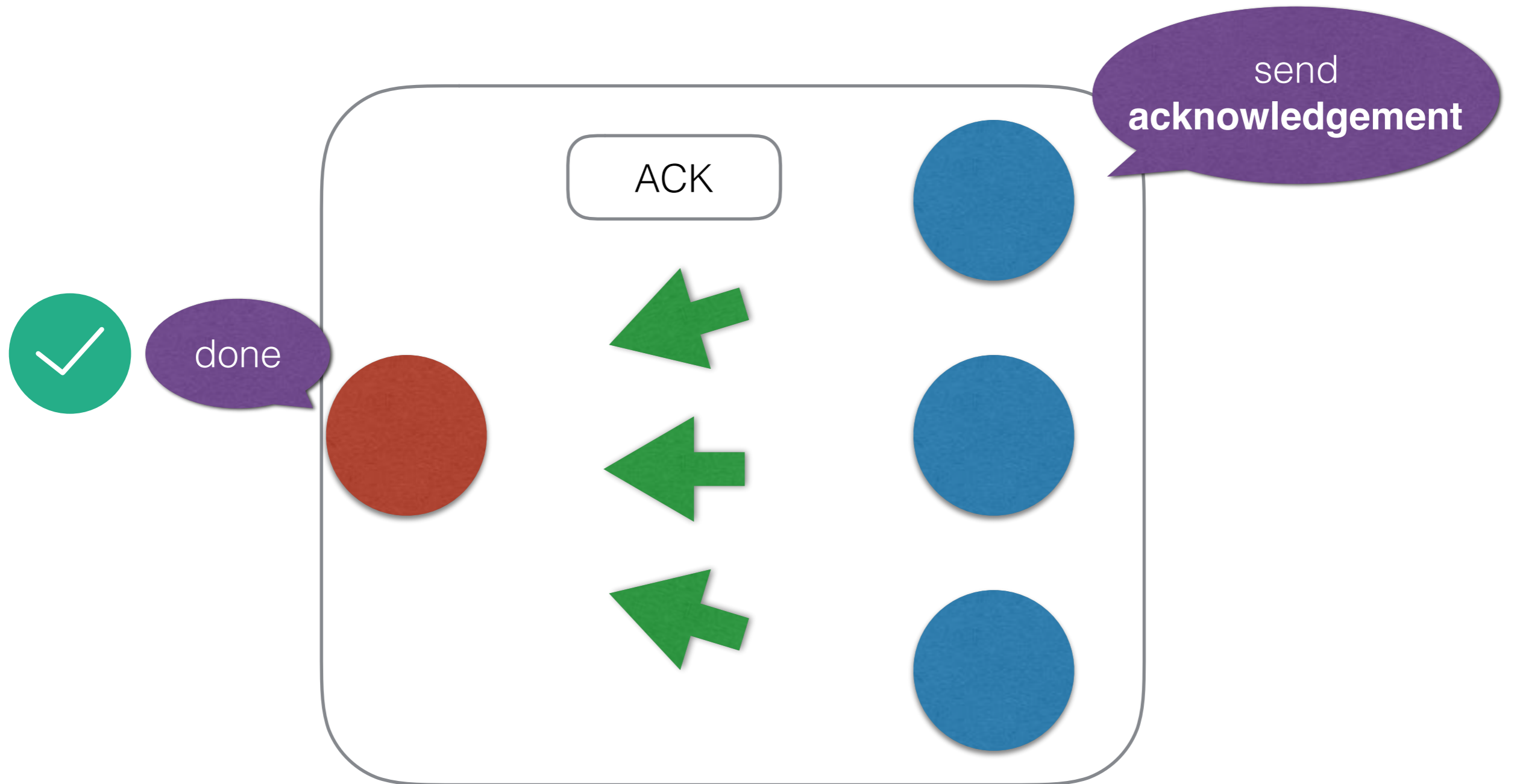
Example: Two phase commit (2PC)

Phase 2



Example: Two phase commit (2PC)

Phase 2



How to verify 2PC?

Sends
match
receives?

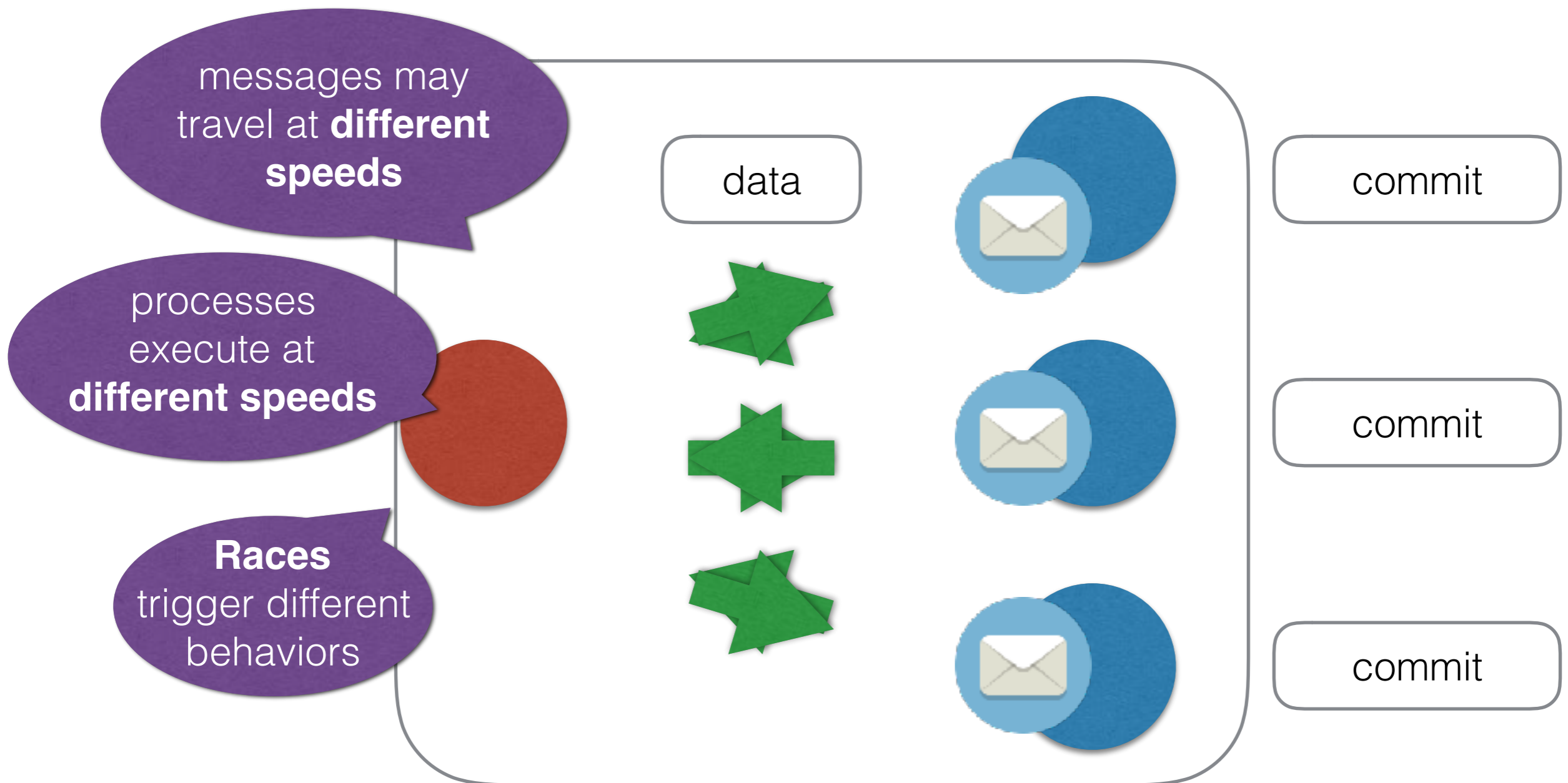
```
1 coord :: Transaction -> Int -> SymSet ProcessId -> Process ()
2 coord transaction n nodes = do
3     fold () query nodes
4     n_ <- fold 0 countVotes nodes
5     where
6         query () pid = do { me <- myPid; send pid (me, transaction) }
7         countVotes c _ = do
8             msg <- expect :: Vote
9             case msg of
10                Accept _ -> return (c + 1)
11                Reject _ -> return c
12
13 acceptor :: Process ()
14 acceptor = do
15     me <- myPid
16     (who, transaction) <- expect :: (ProcessId, Transaction)
17     vote <- chooseVote transaction
18     send who vote
```

Does Implementation Deadlock?

How to verify 2PC?

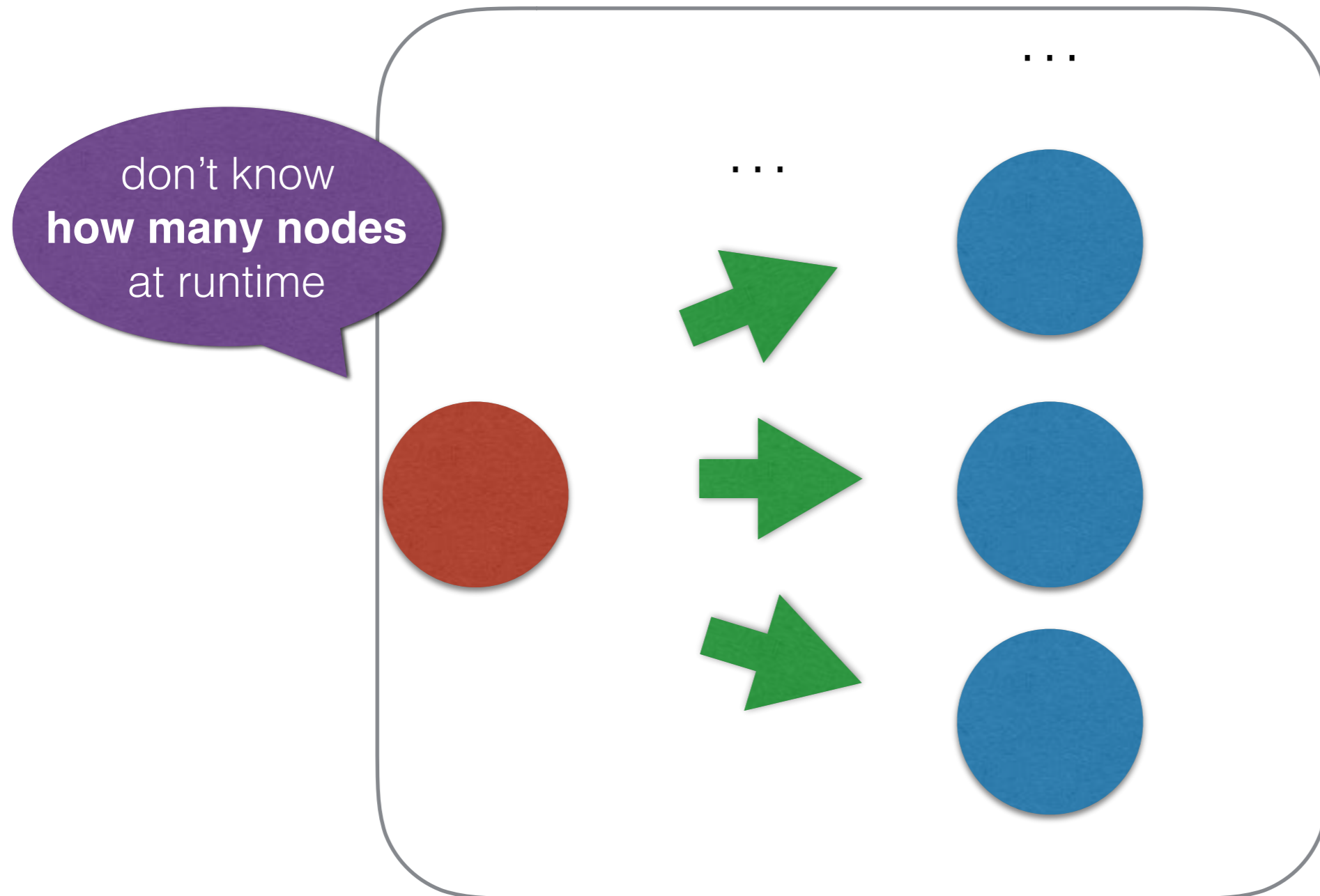
How to verify 2PC?

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

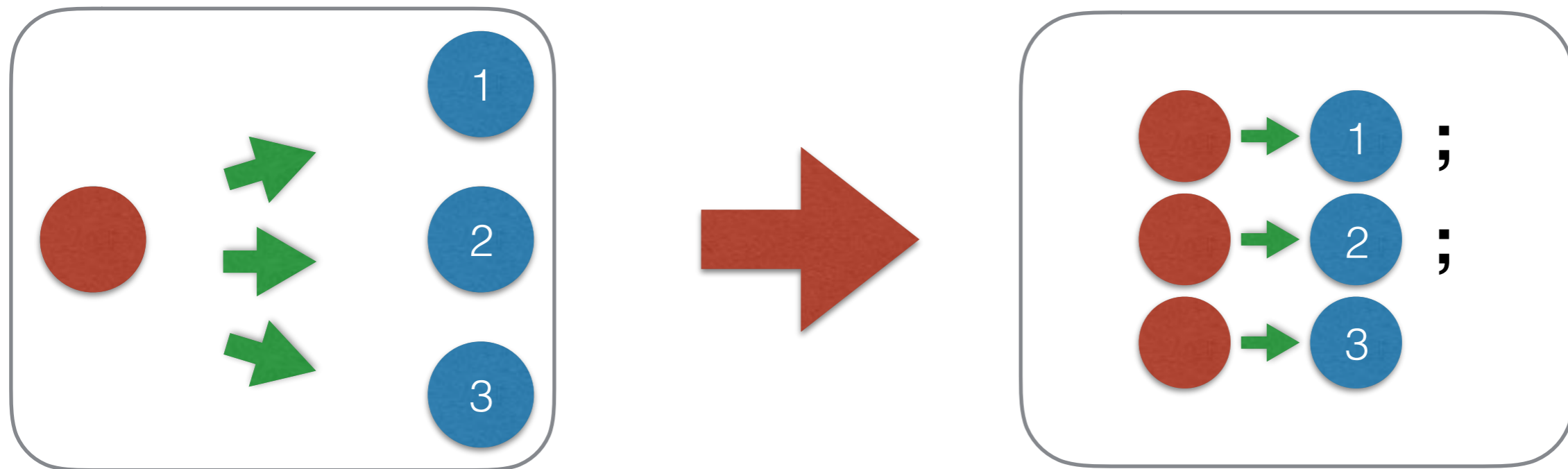
The Evaluation

The Key Idea

Canonical Sequentialization

Canonical Sequentialization

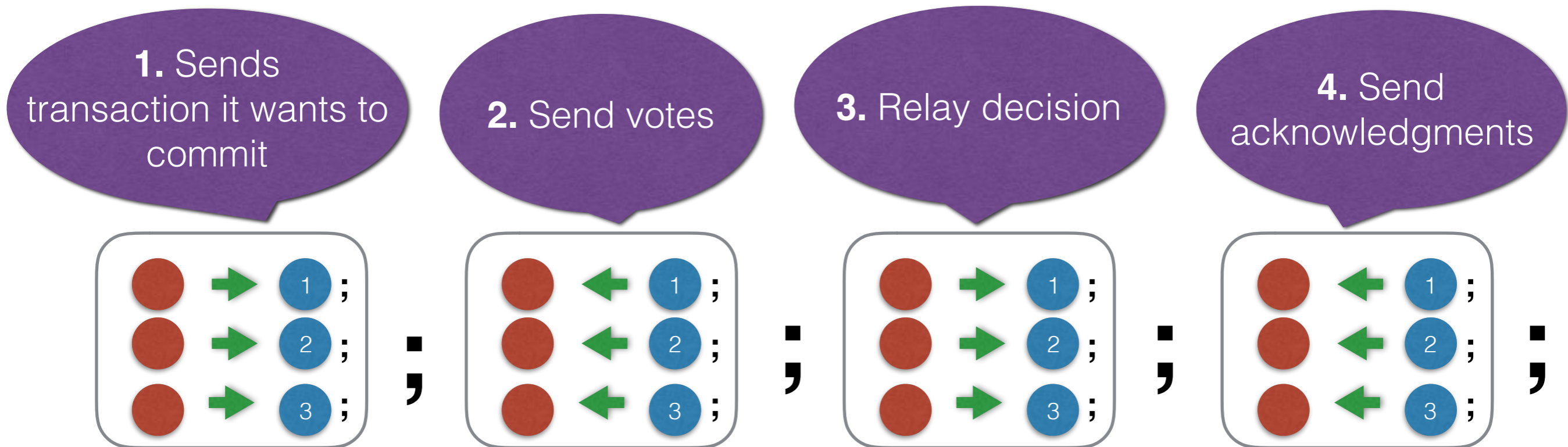
Don't enumerate execution orders...



... **Reason** about **single representative** execution

Canonical Sequentialization

Example 2PC

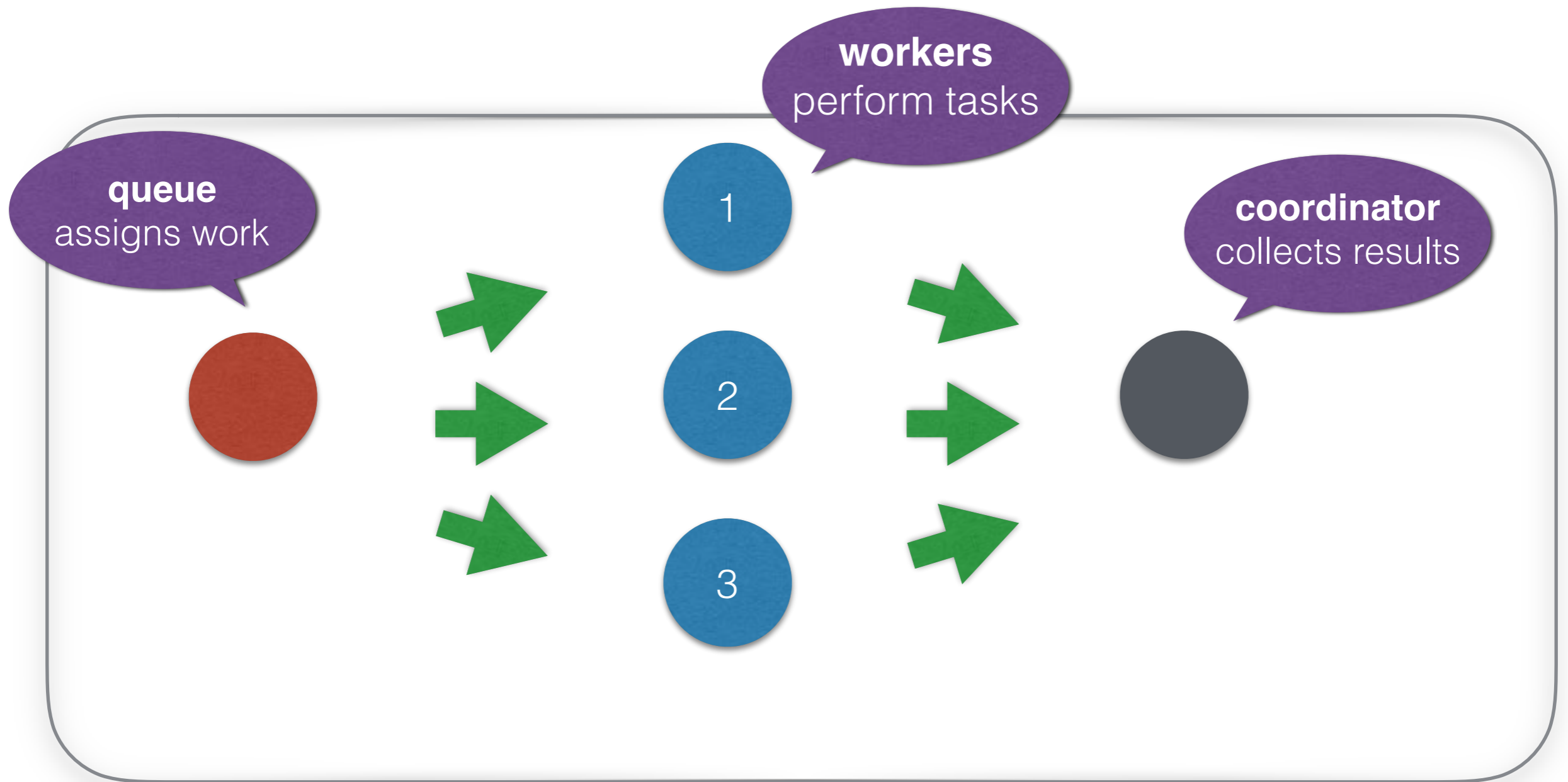


Canonical Sequentialization

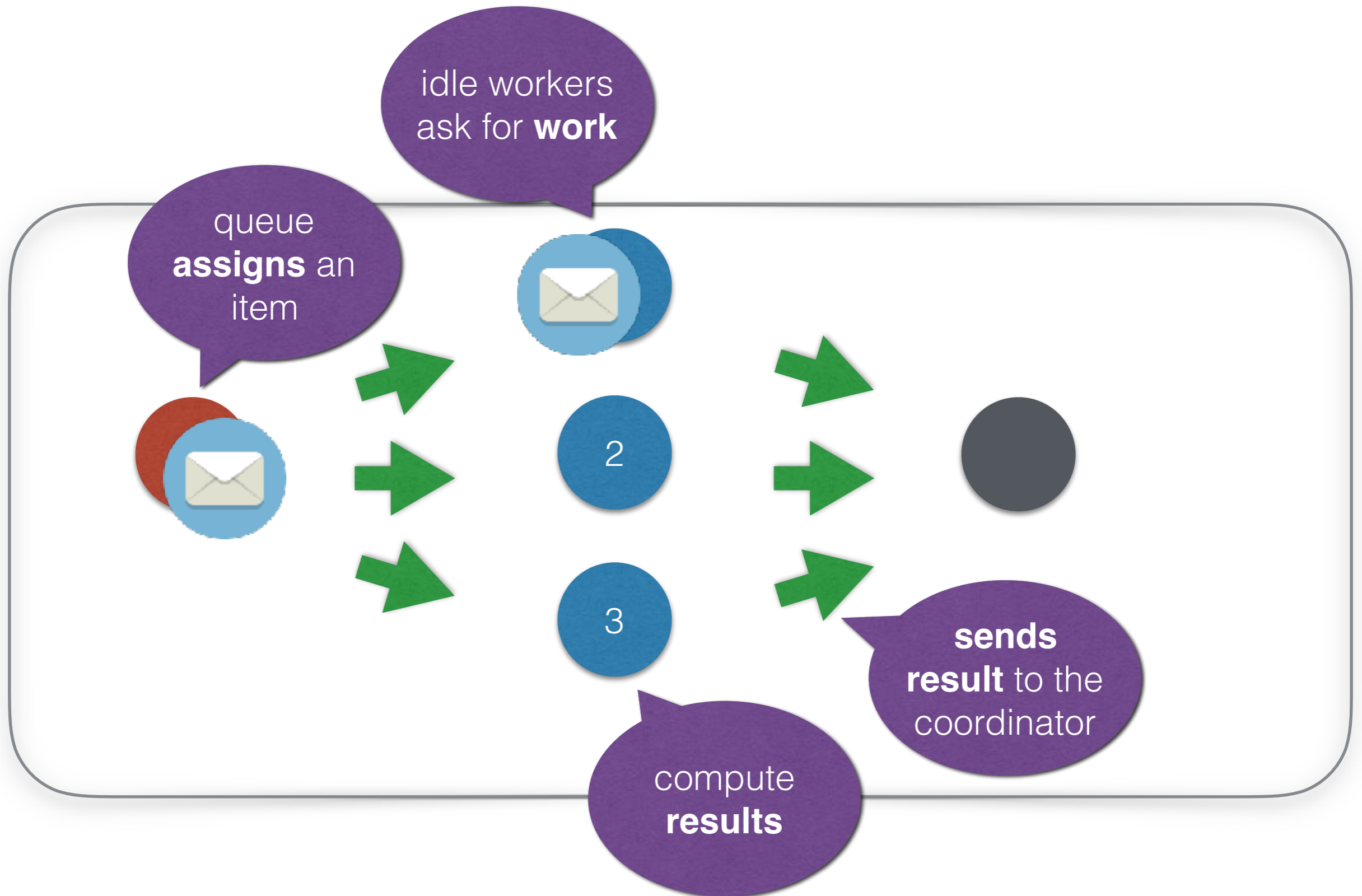
A Trickier Example

Work stealing queue

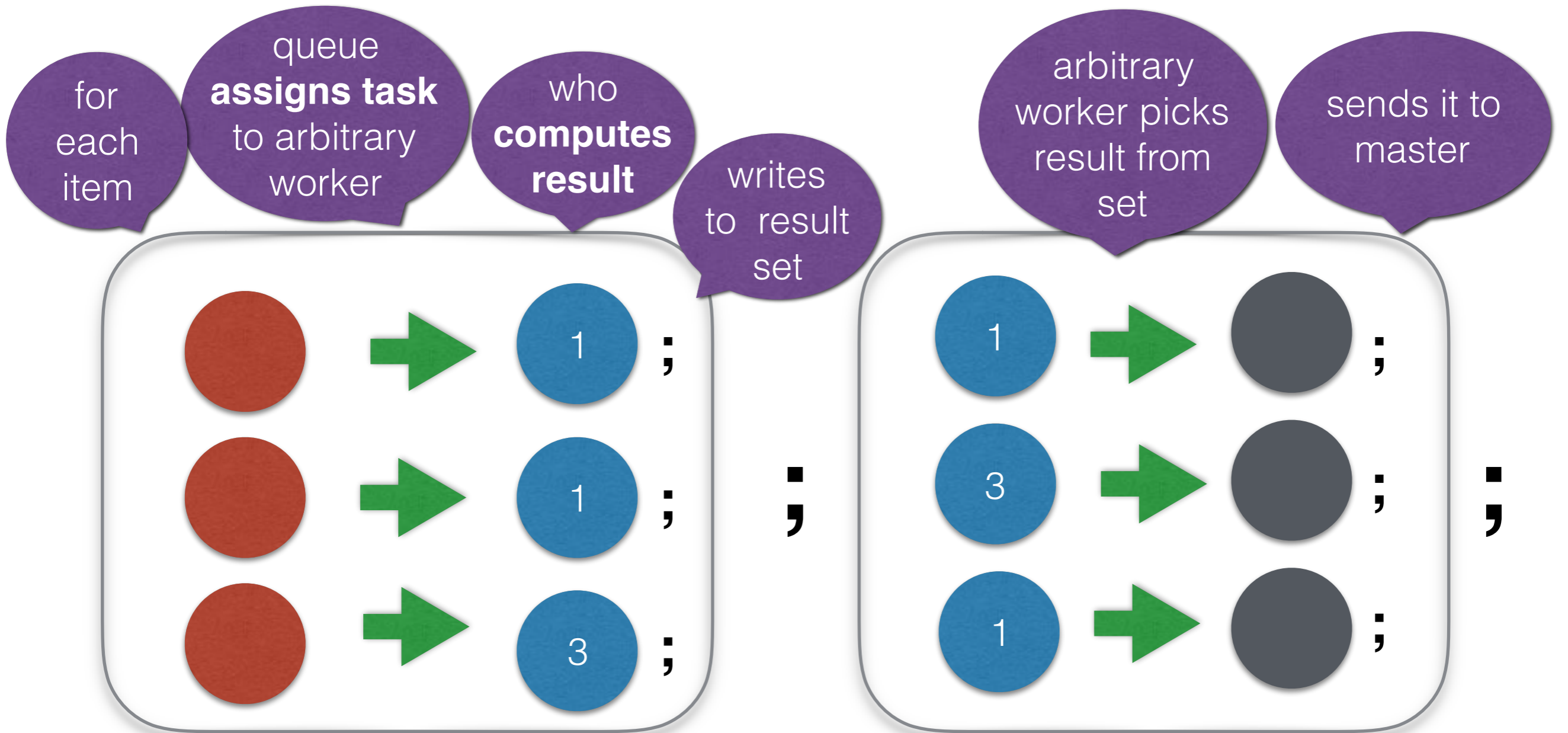
Work stealing queue



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

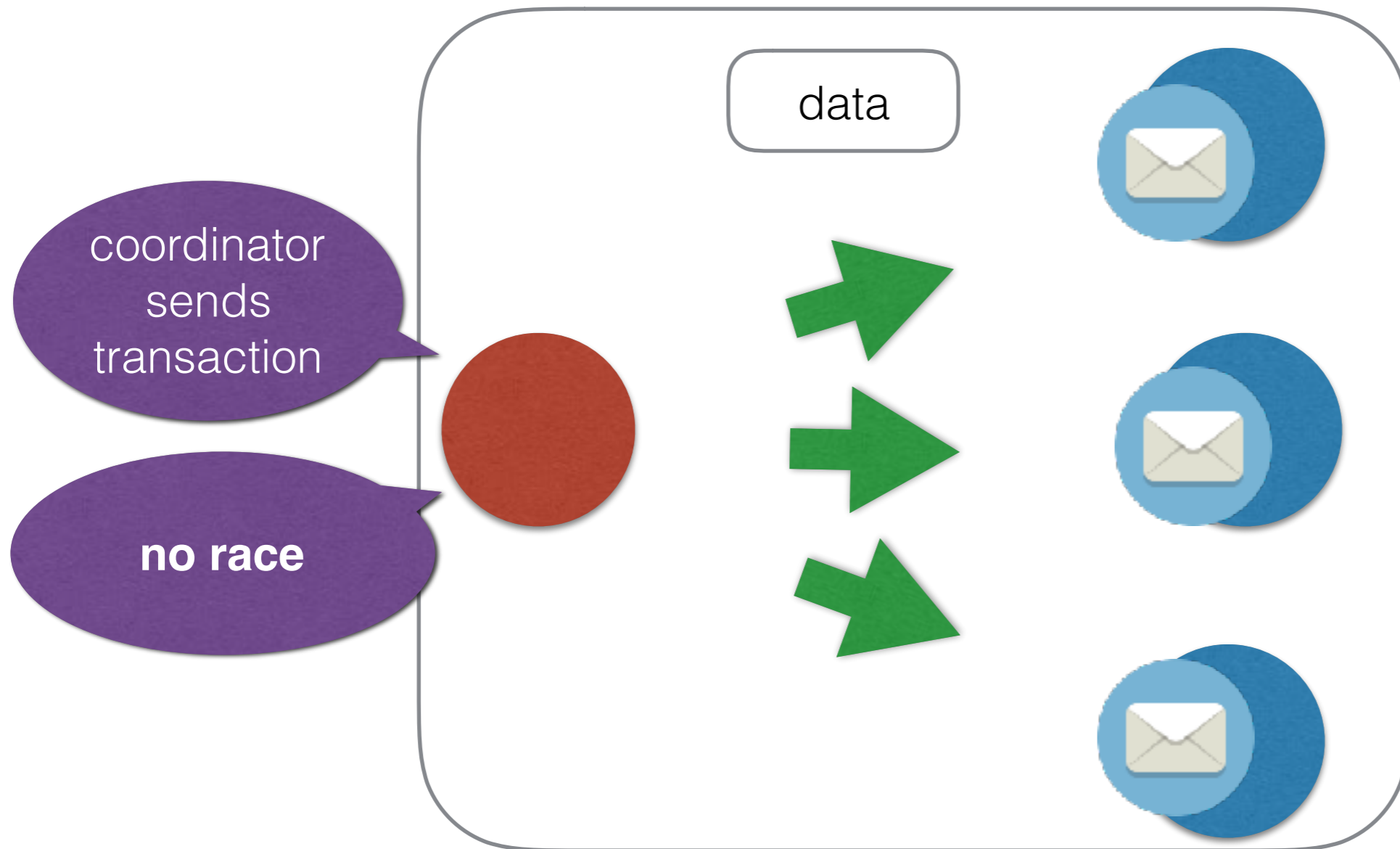
1. Restrict Computation Model

Symmetric Nondeterminism

Races yield equivalent outcomes

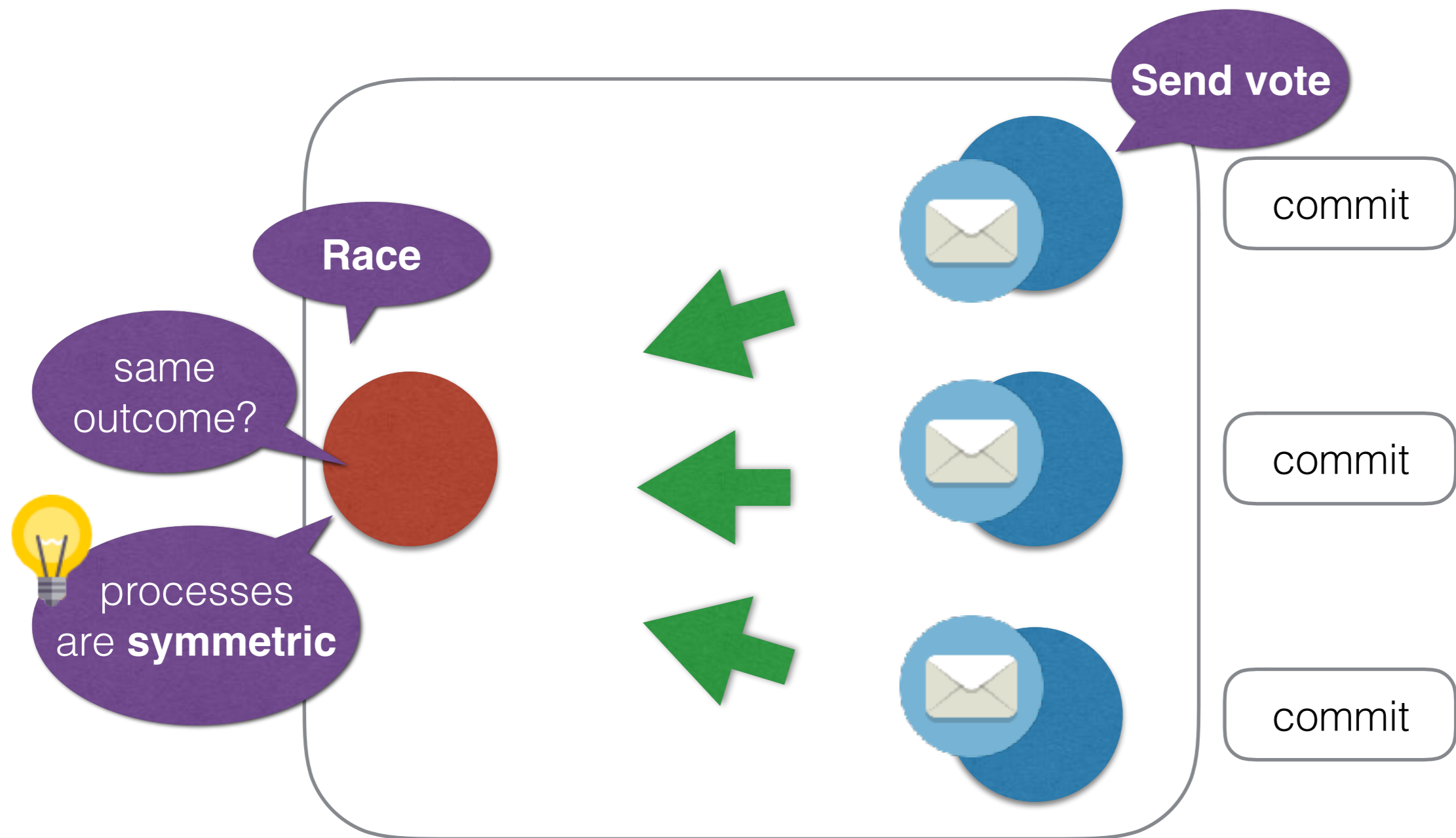
Symmetric Nondeterminism

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



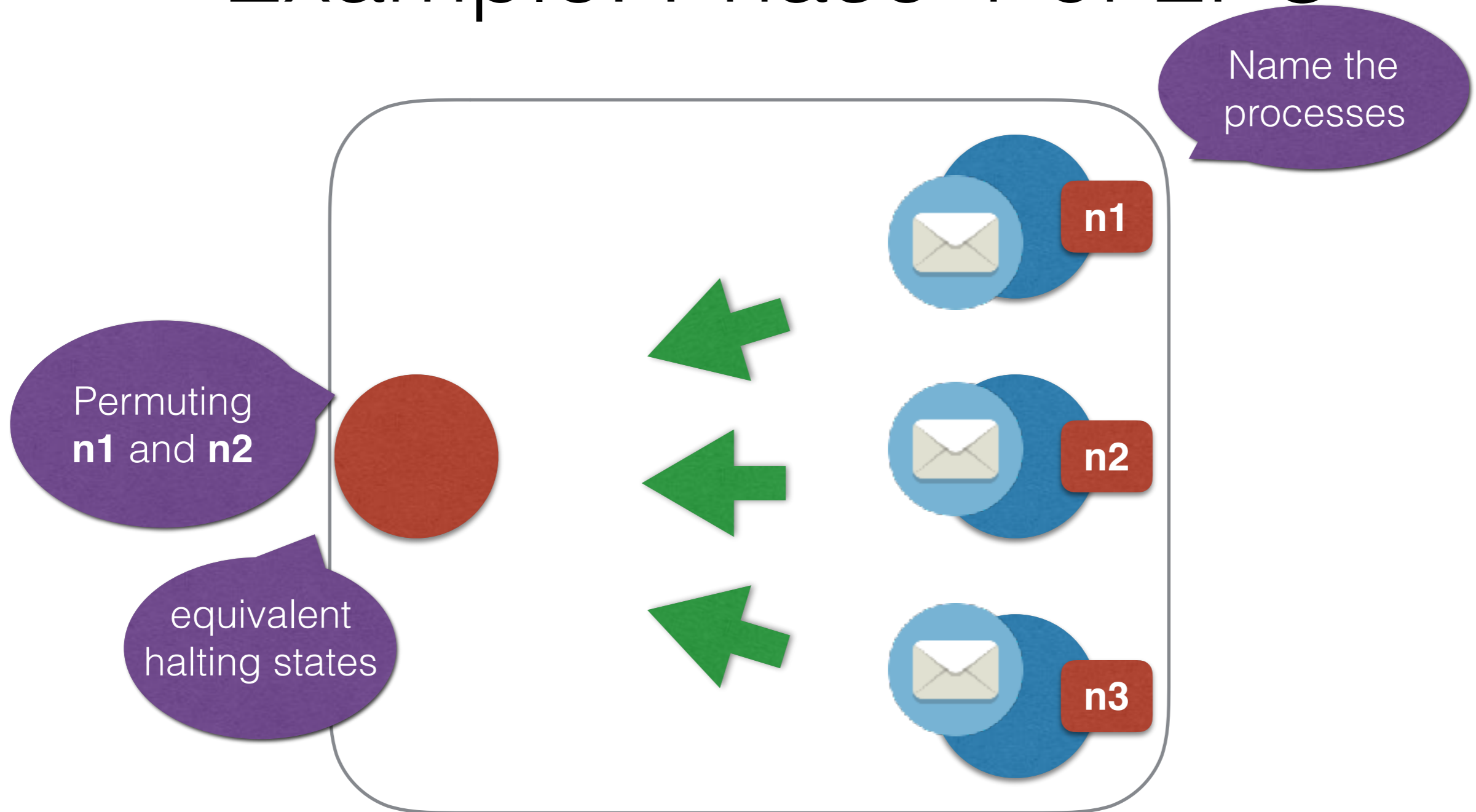
[Norris and
Dill 1996]

Permuting Process Identifiers

Yields equivalent halting states

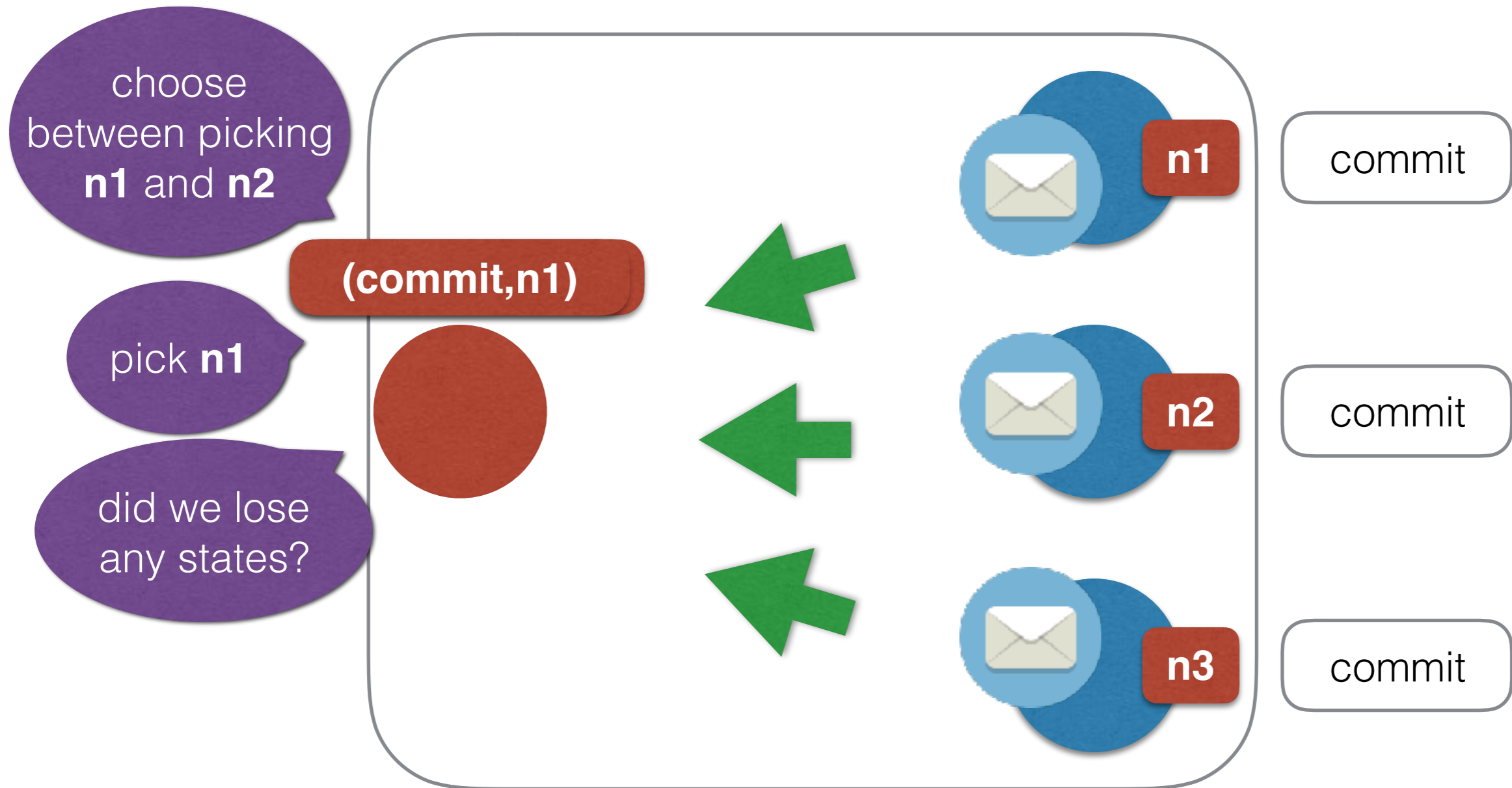
Symmetry

Example: Phase 1 of 2PC



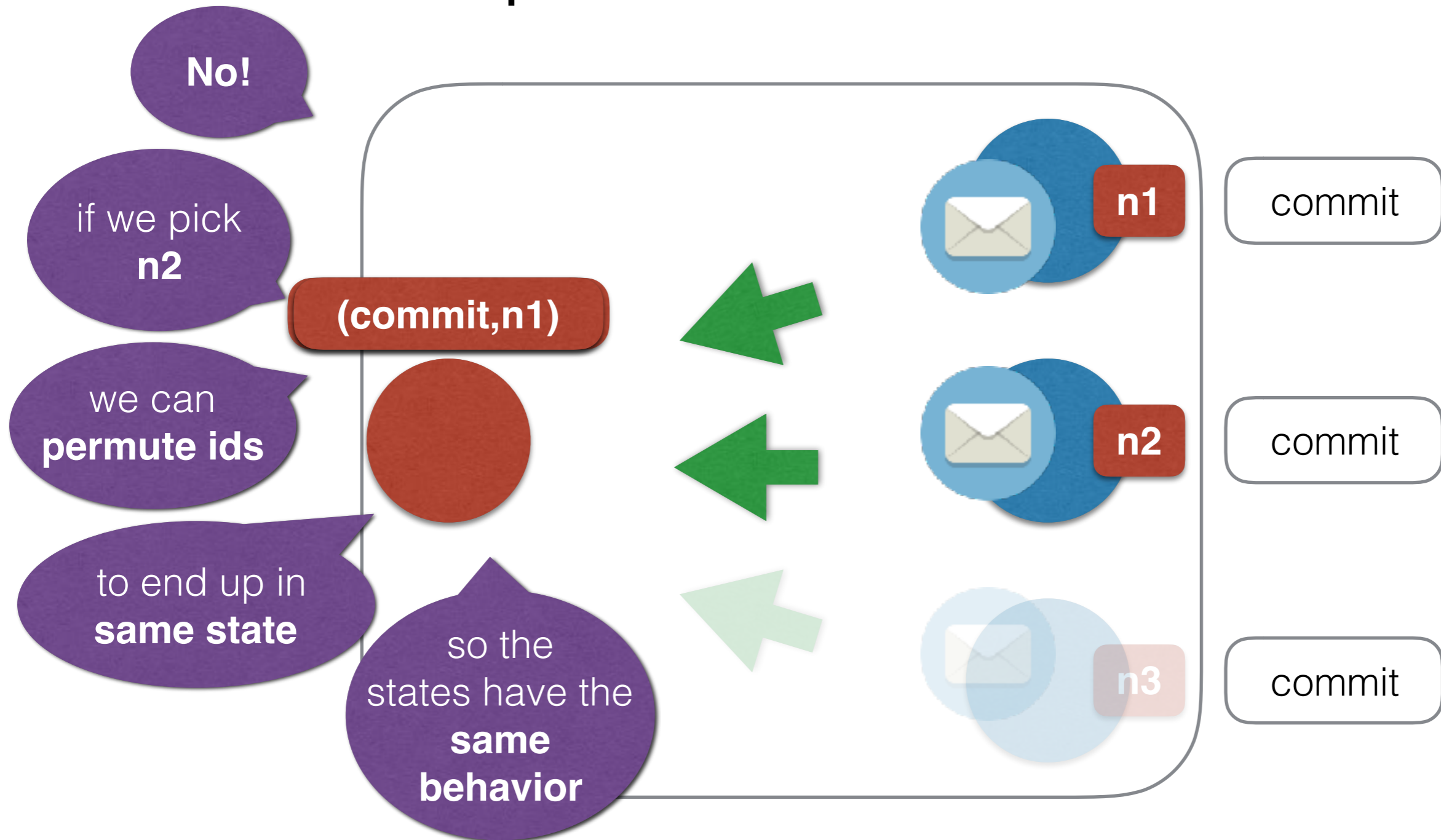
Symmetric Nondeterminism

Example: Phase 1 of 2PC



Symmetric Nondeterminism

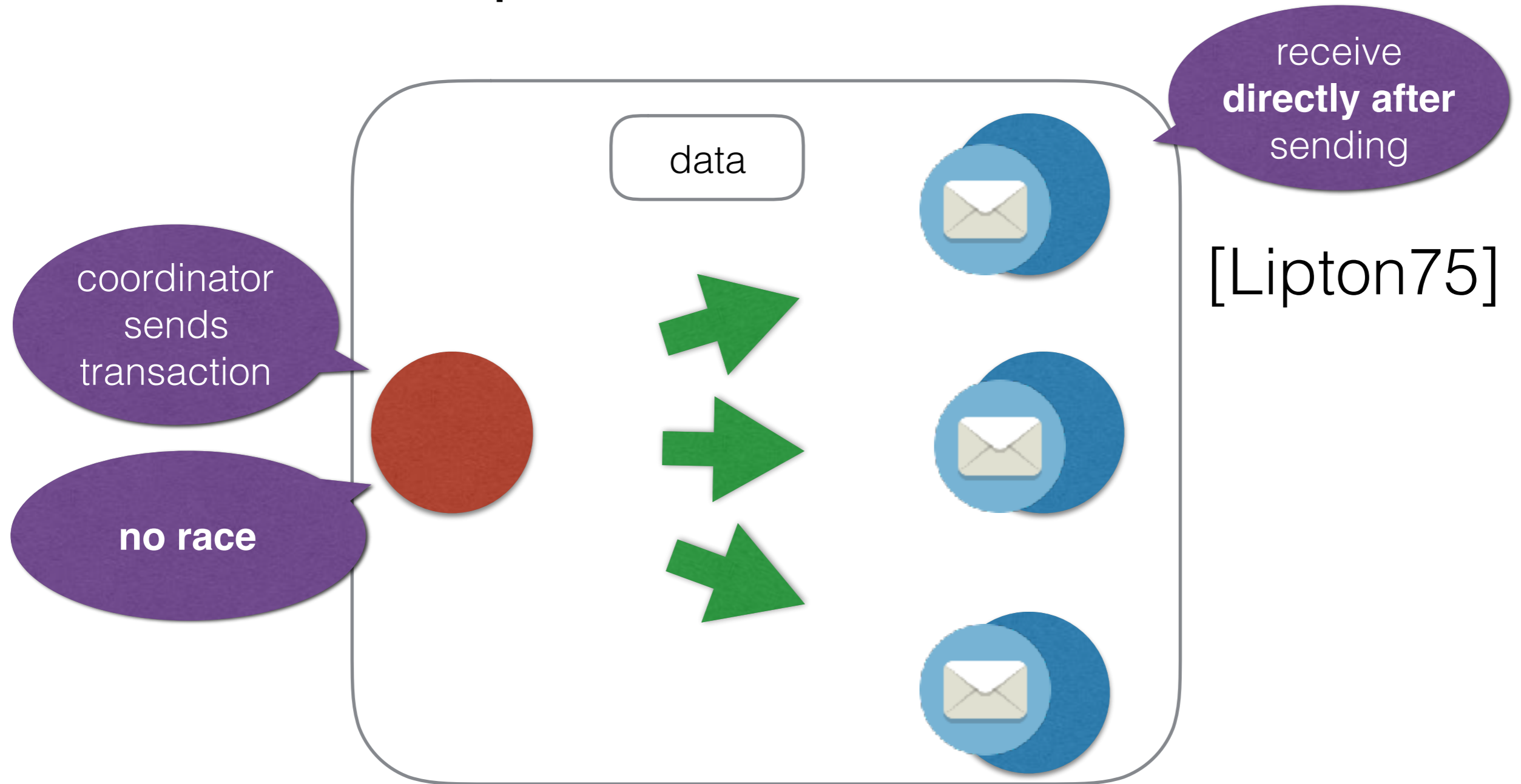
Example: Phase 1 of 2PC



How can we use symmetry
to sequentialize?

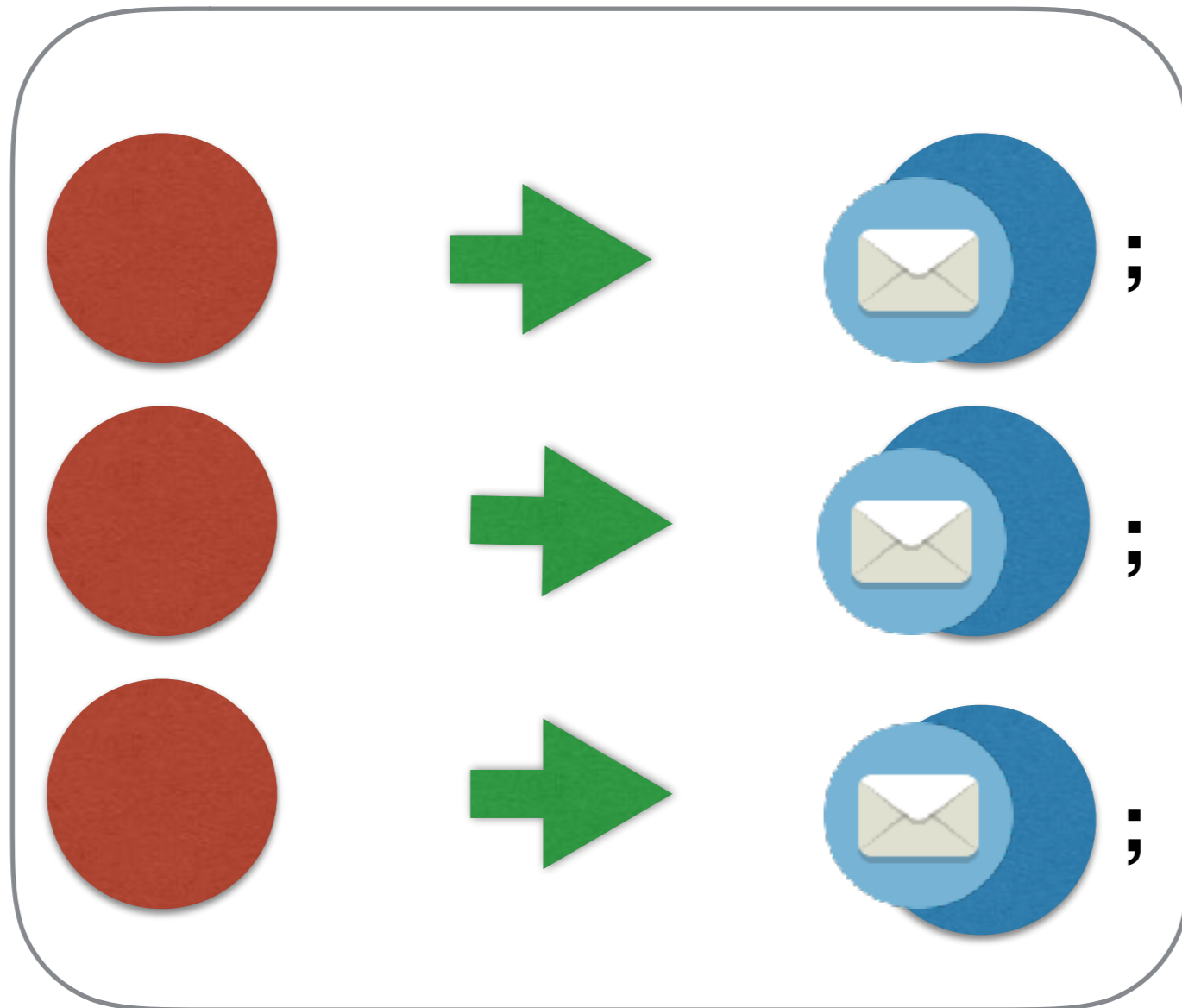
Symmetric Nondeterminism

Example: Phase 1 of 2PC



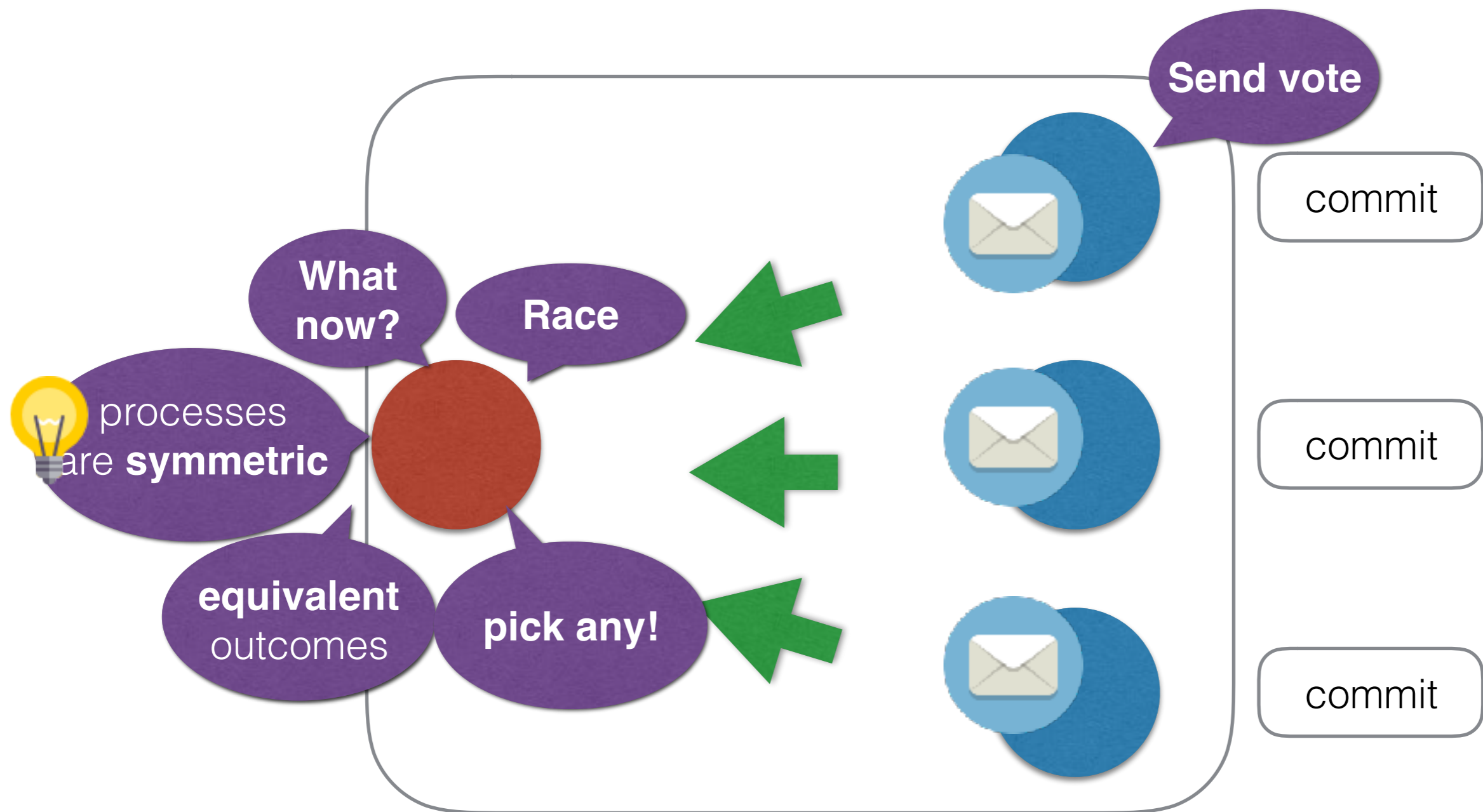
Symmetric Nondeterminism

Example: Phase 1 of 2PC



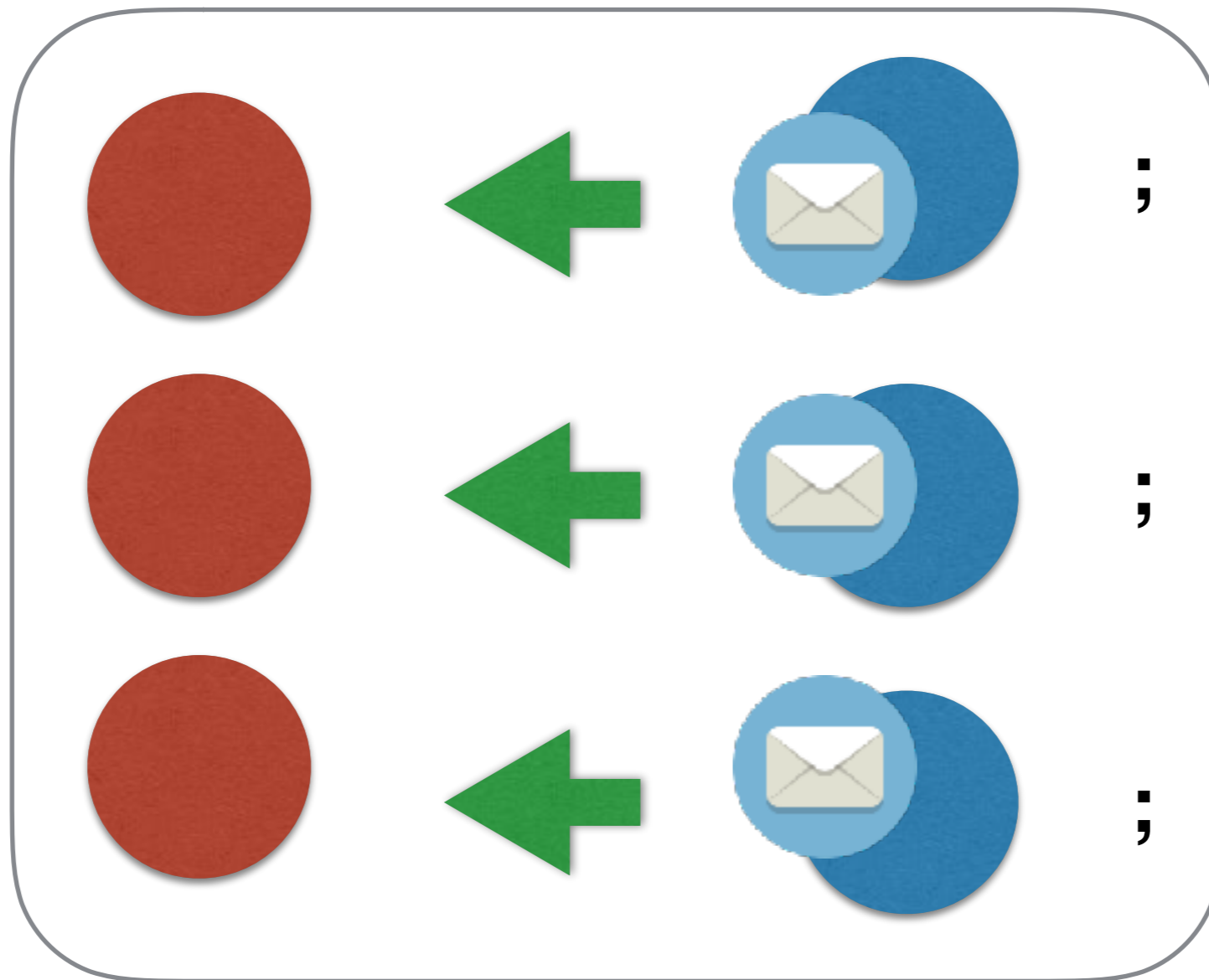
Symmetric Nondeterminism

Example: Phase 1 of 2PC



Symmetric Nondeterminism

Example: Phase 1 of 2PC



The Implementation

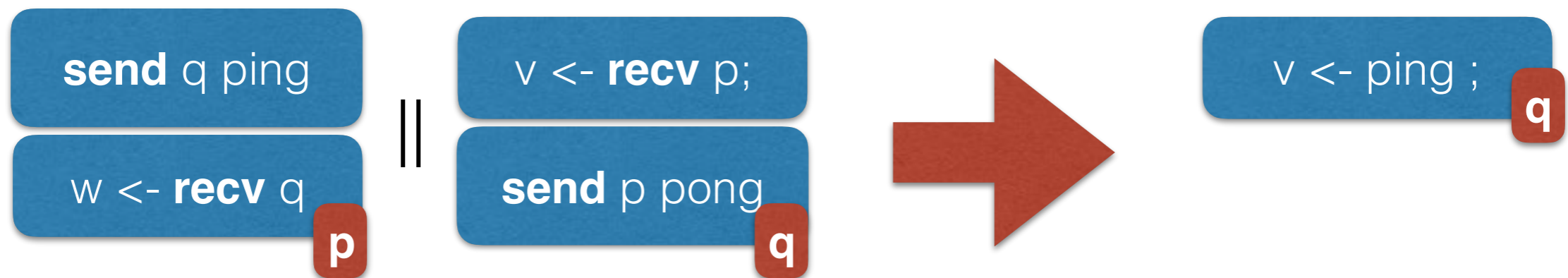
1. Restrict Computation Model
2. Sequentialize by Rewriting

2. Sequentialize by Rewriting

(by example)

2. Sequentialize by Rewriting

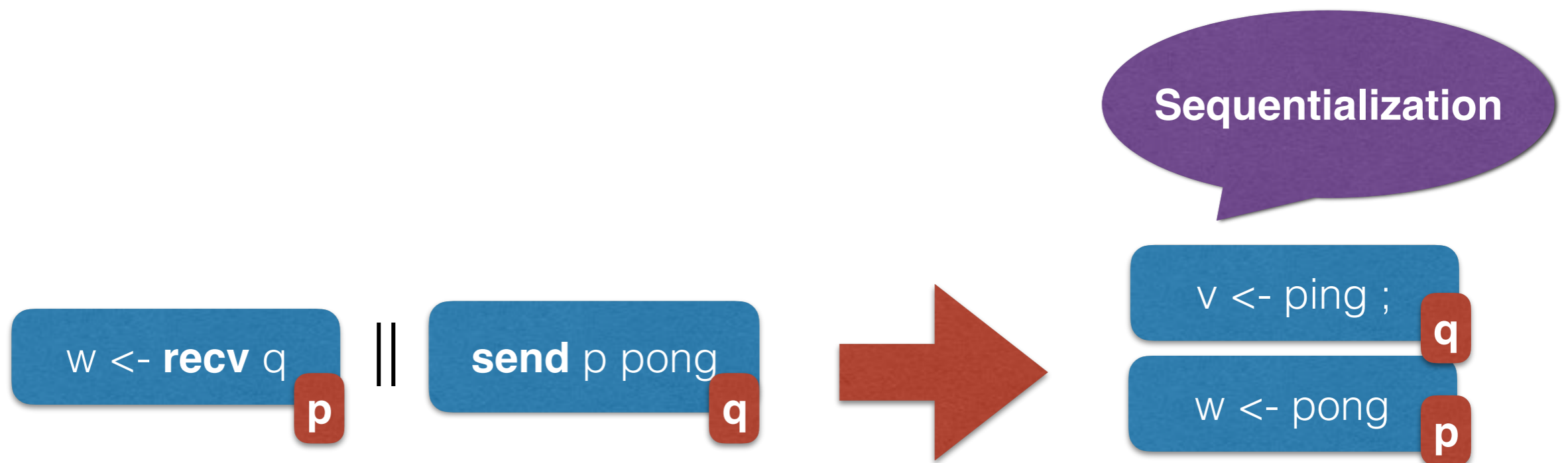
Example 1



p, q are in parallel

2. Sequentialize by Rewriting

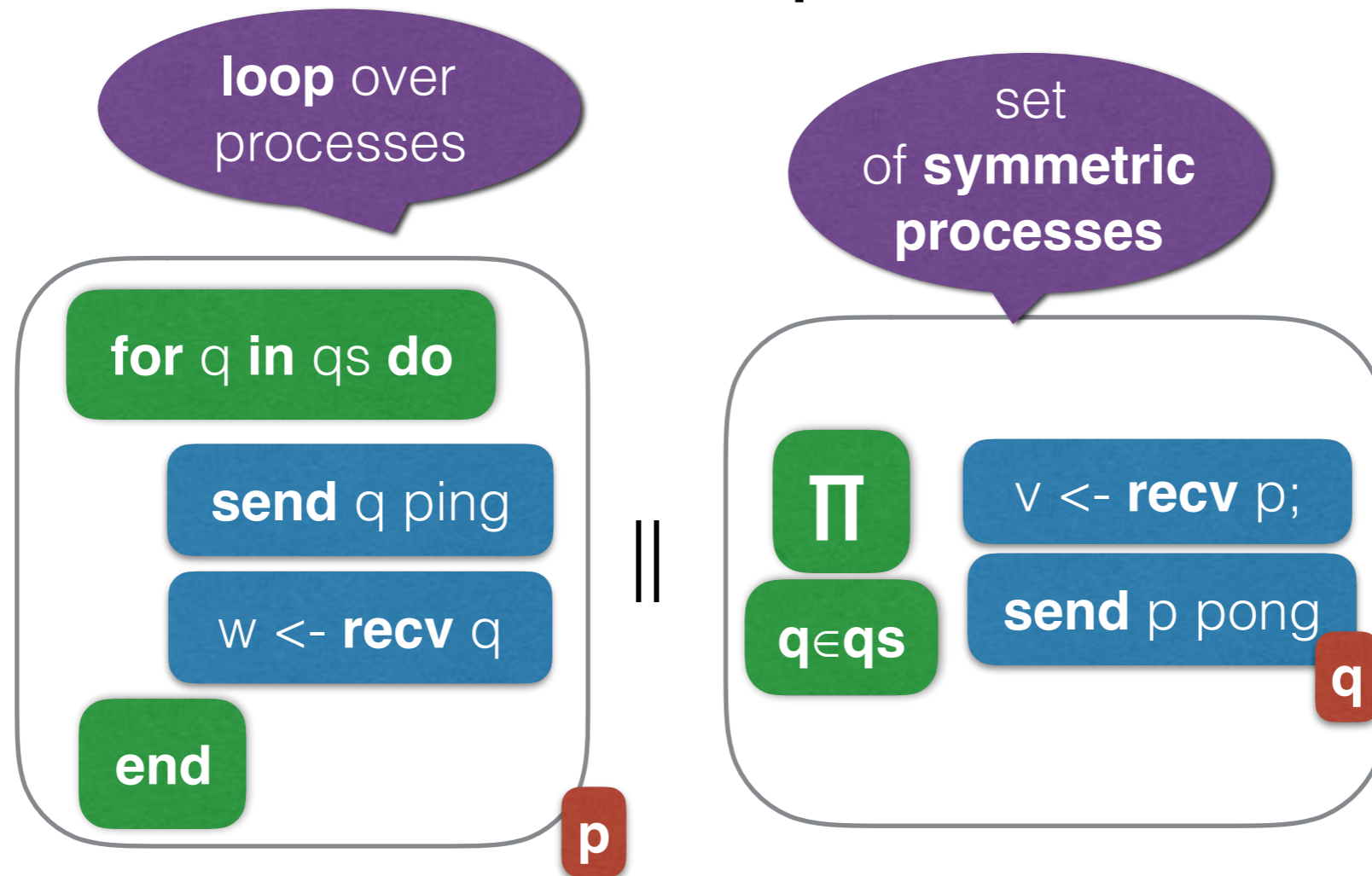
Example 1



p, q are in parallel

2. Sequentialize by Rewriting

Example 2



$p, qs = \{q_1 \dots q_n\}$ are in parallel

2. Sequentialize by Rewriting

Example 2

Arbitrary iteration

```
for q in qs do
```

```
  send q ping
```

```
  w <- recv q
```

```
end
```

p

||

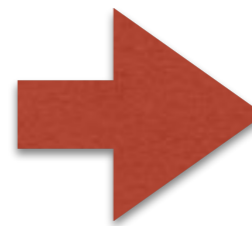
Π

$q \in qs$

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

```
send p pong
```

q



Generalize

```
for q in qs do
```

```
  v <- ping ;
```

```
  w <- pong
```

q

p

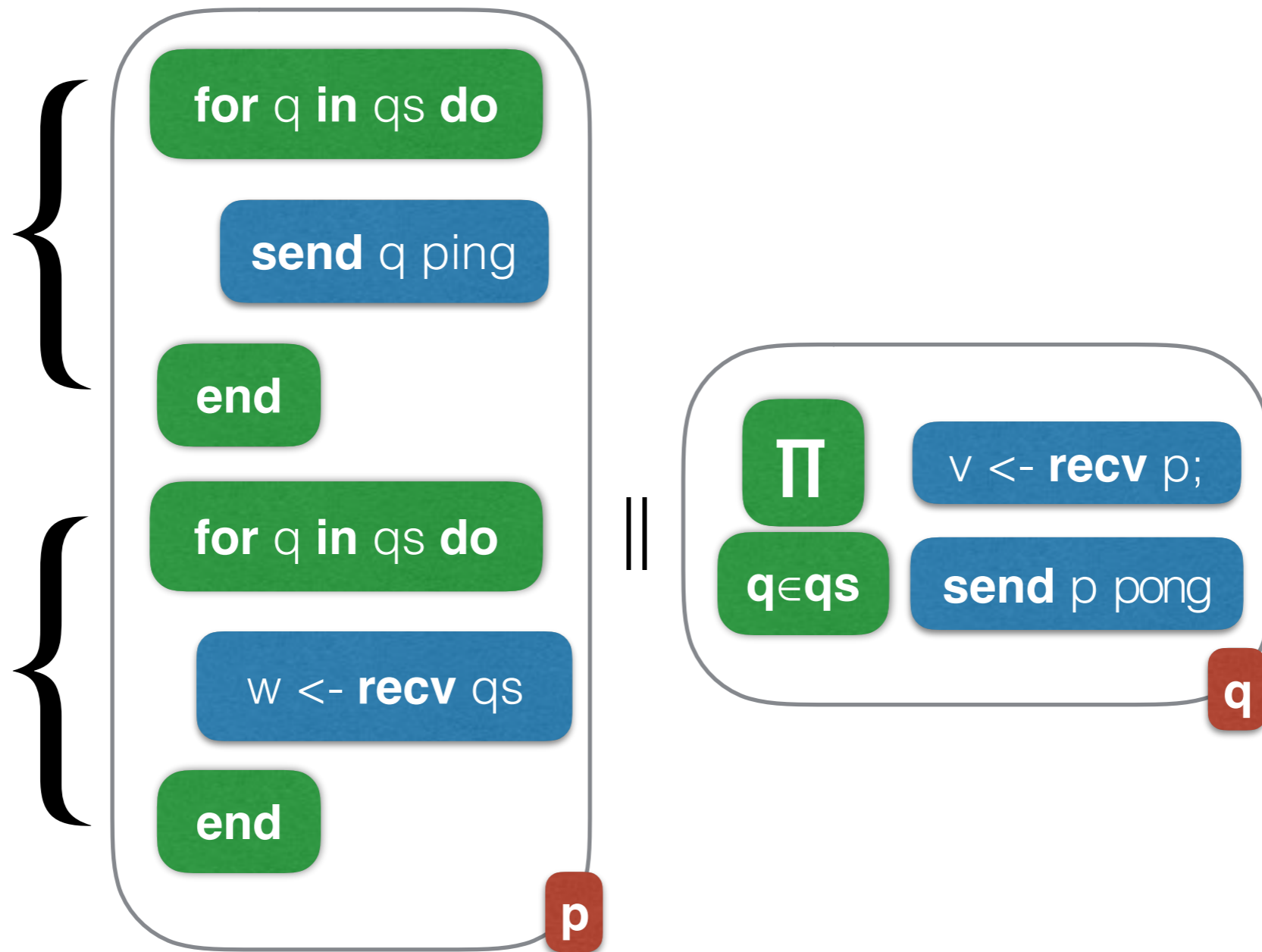
```
end
```

p, qs={q1...qn} are in parallel

2. Sequentialize by Rewriting

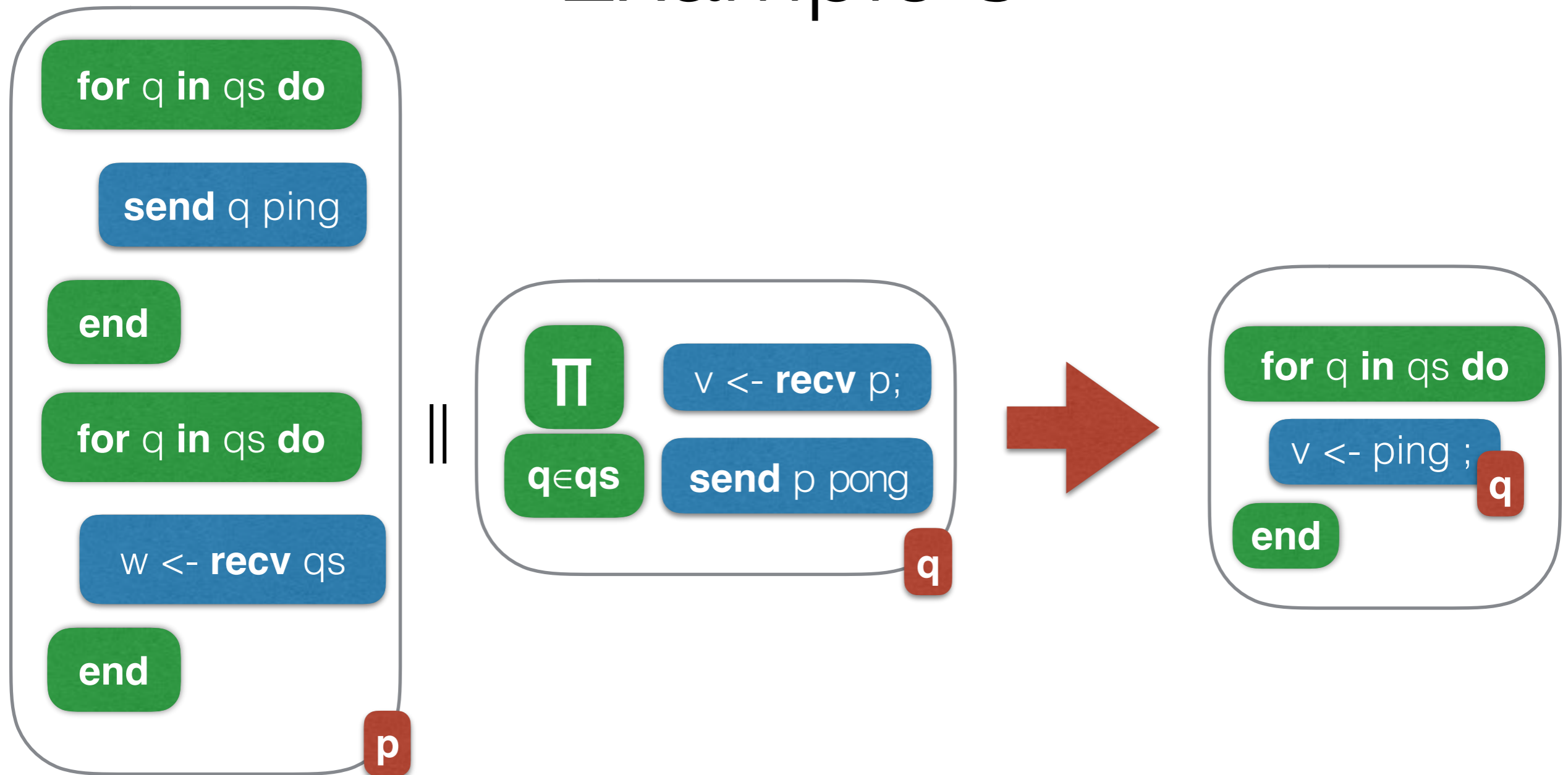
Example 3

two loops



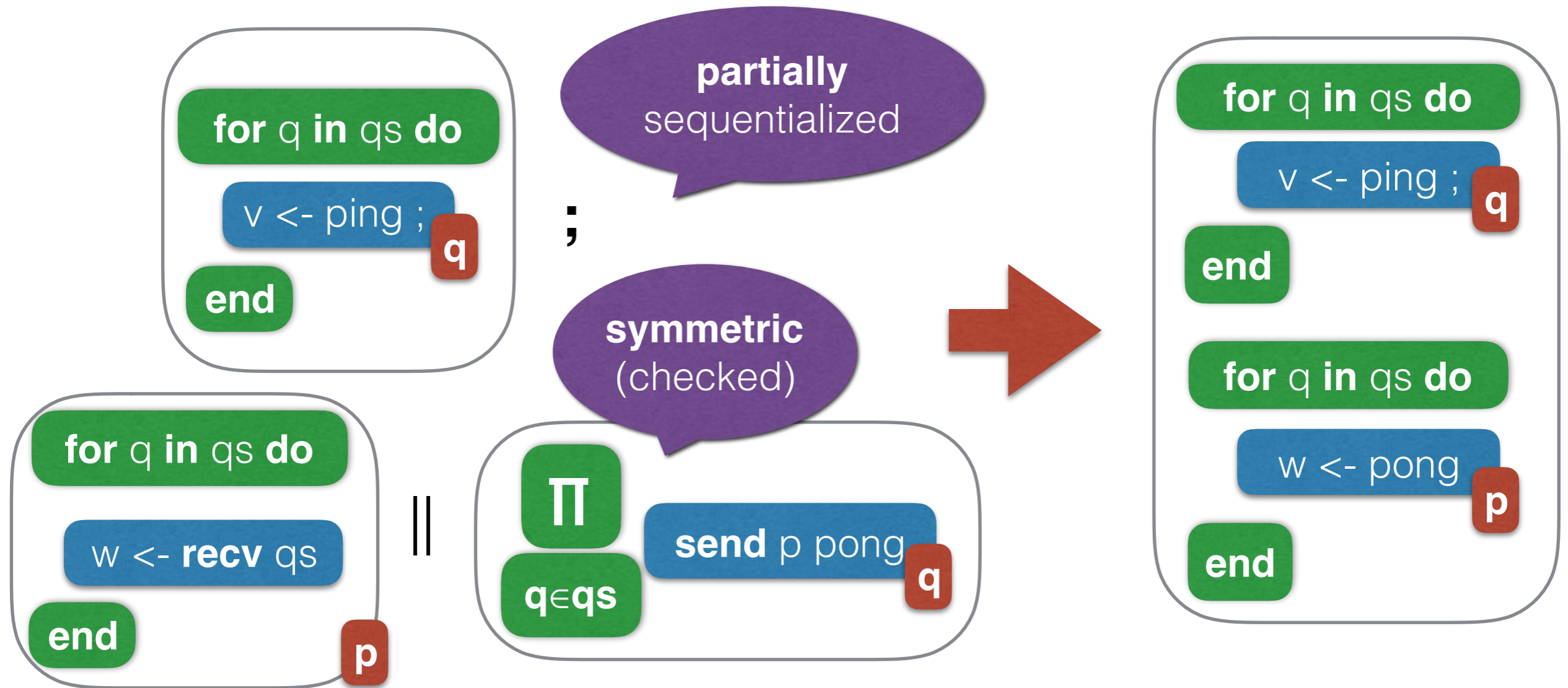
2. Sequentialize by Rewriting

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

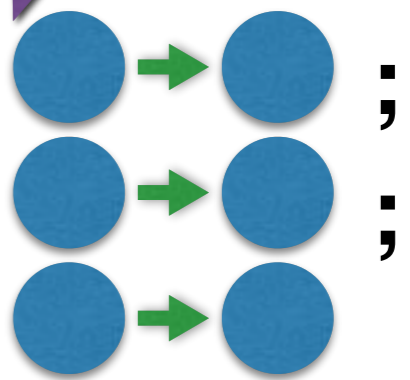
Implemented
in a **Haskell**
library

```
10 -- Brisk Haskell library
11 module Brisk (
12   !*! msg -> !*! Message
13   !*! Message -> !*! Message
14   !*! Message -> !*! Message
15   !*! Message -> !*! Message
16   !*! Message -> !*! Message
17   !*! Message -> !*! Message
18   !*! Message -> !*! Message
19   !*! Message -> !*! Message
20   !*! Message -> !*! Message
21   !*! Message -> !*! Message
22   !*! Message -> !*! Message
23   !*! Message -> !*! Message
24   !*! Message -> !*! Message
25   !*! Message -> !*! Message
26   !*! Message -> !*! Message
27   !*! Message -> !*! Message
28   !*! Message -> !*! Message
29   !*! Message -> !*! Message
30   !*! Message -> !*! Message
31   !*! Message -> !*! Message
32   !*! Message -> !*! Message
33   !*! Message -> !*! Message
34   !*! Message -> !*! Message
35   !*! Message -> !*! Message
36   !*! Message -> !*! Message
37   !*! Message -> !*! Message
38   !*! Message -> !*! Message
39   !*! Message -> !*! Message
40   !*! Message -> !*! Message
41   !*! Message -> !*! Message
42   !*! Message -> !*! Message
43   !*! Message -> !*! Message
44   !*! Message -> !*! Message
45   !*! Message -> !*! Message
46   !*! Message -> !*! Message
47   !*! Message -> !*! Message
48   !*! Message -> !*! Message
49   !*! Message -> !*! Message
50   !*! Message -> !*! Message
51   !*! Message -> !*! Message
52   !*! Message -> !*! Message
53   !*! Message -> !*! Message
54   !*! Message -> !*! Message
55   !*! Message -> !*! Message
56   !*! Message -> !*! Message
57   !*! Message -> !*! Message
58   !*! Message -> !*! Message
59   !*! Message -> !*! Message
60   !*! Message -> !*! Message
61   !*! Message -> !*! Message
62   !*! Message -> !*! Message
63   !*! Message -> !*! Message
64   !*! Message -> !*! Message
65   !*! Message -> !*! Message
66   !*! Message -> !*! Message
67   !*! Message -> !*! Message
68   !*! Message -> !*! Message
69   !*! Message -> !*! Message
70   !*! Message -> !*! Message
71   !*! Message -> !*! Message
72   !*! Message -> !*! Message
73   !*! Message -> !*! Message
74   !*! Message -> !*! Message
75   !*! Message -> !*! Message
76   !*! Message -> !*! Message
77   !*! Message -> !*! Message
78   !*! Message -> !*! Message
79   !*! Message -> !*! Message
80   !*! Message -> !*! Message
81   !*! Message -> !*! Message
82   !*! Message -> !*! Message
83   !*! Message -> !*! Message
84   !*! Message -> !*! Message
85   !*! Message -> !*! Message
86   !*! Message -> !*! Message
87   !*! Message -> !*! Message
88   !*! Message -> !*! Message
89   !*! Message -> !*! Message
90   !*! Message -> !*! Message
91   !*! Message -> !*! Message
92   !*! Message -> !*! Message
93   !*! Message -> !*! Message
94   !*! Message -> !*! Message
95   !*! Message -> !*! Message
96   !*! Message -> !*! Message
97   !*! Message -> !*! Message
98   !*! Message -> !*! Message
99   !*! Message -> !*! Message
100  !*! Message -> !*! Message
```

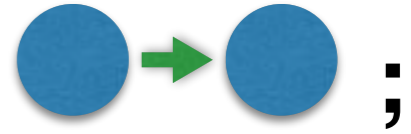
communication
primitives like **send** /
receive / **foreach**

→ **Brisk**

computes
**canonical
sequentialization**



provides
**counterexample to
sequentialization**



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**

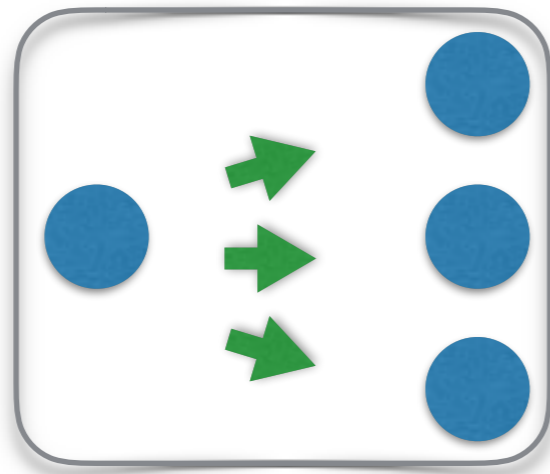
symmetric races produce equivalent outcomes

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

What's next



Faults



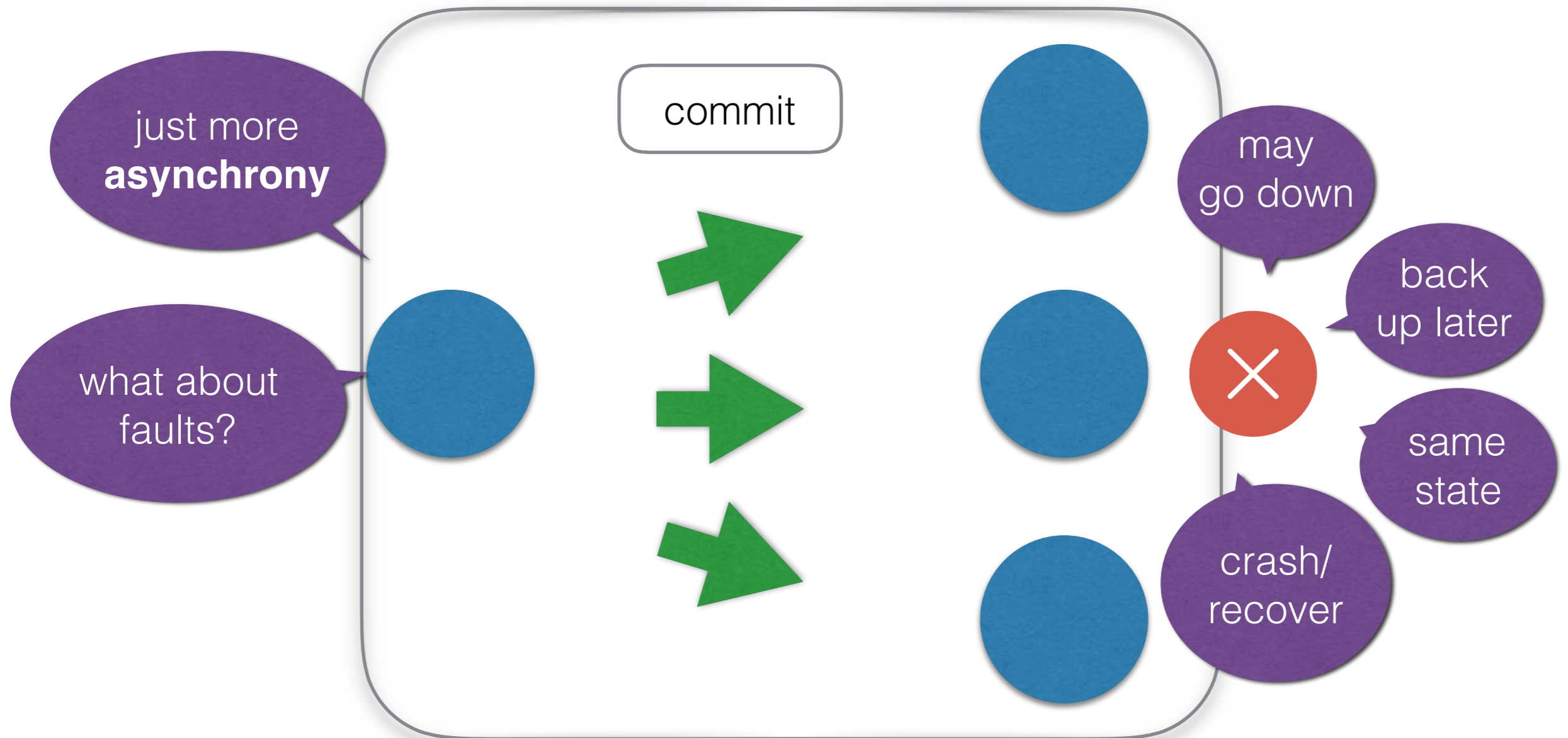
larger
program
class



larger class
of properties

Backup slides

2PC: Faults



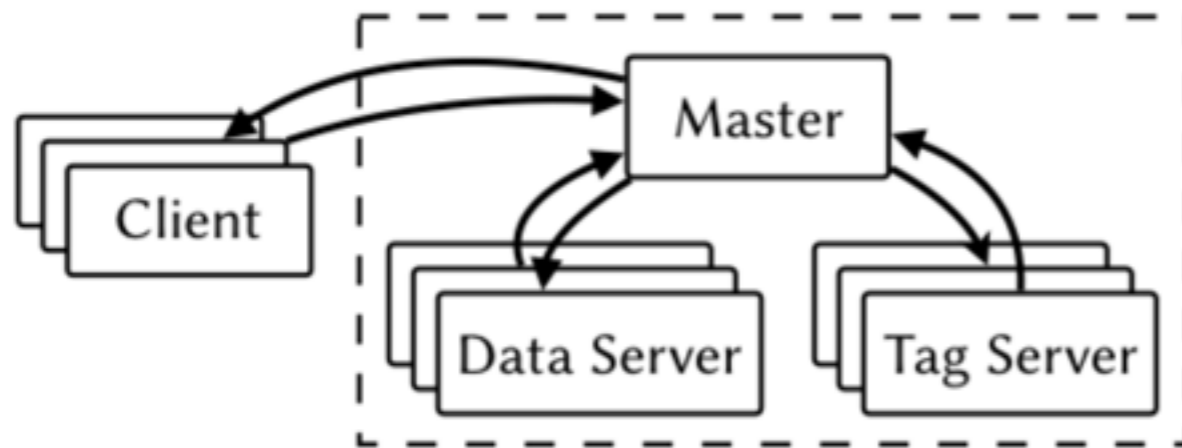
2PC: Faults



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)

Tag Server:

AddTag(name)
GetTag(tag)

Data Server:

PutBlob(name, data)
GetBlob(name)

mutable

immutable

Real consequences



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.

I should be more confident in a few more weeks

This issue still occurs in mono-4.6.2.16

(fingers are still crossed right now).



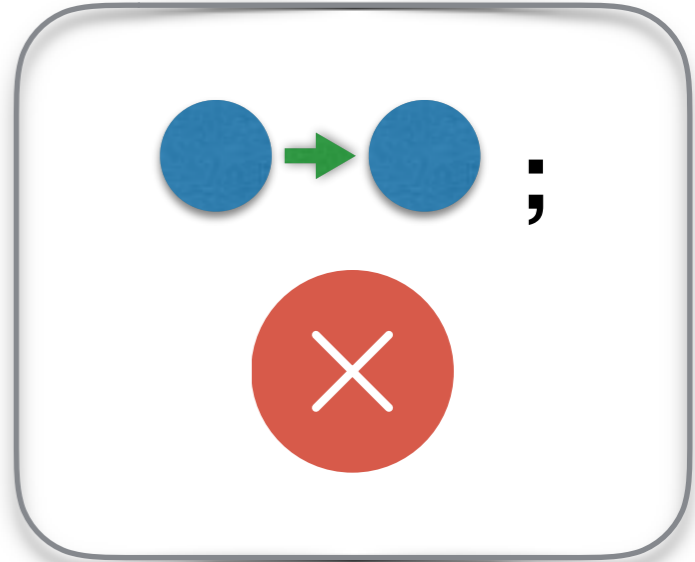
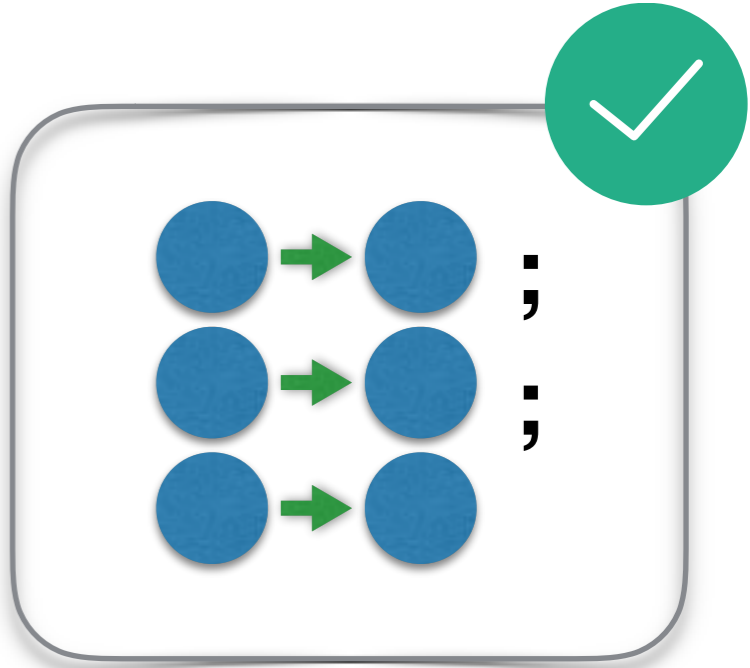
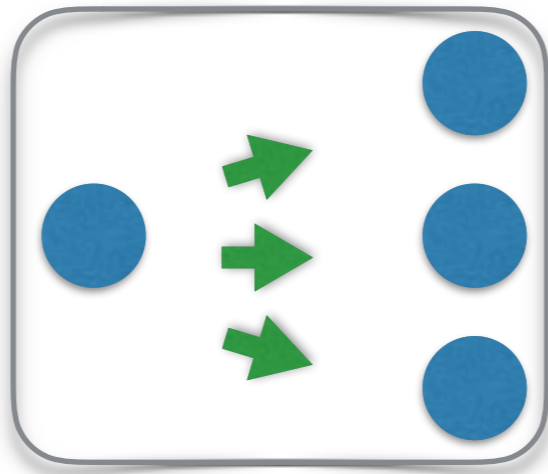
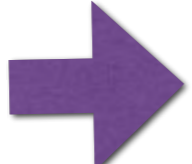
Summary

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

fast
~20-100
ms

automated
proofs

```
10  -- By the constructor
11  @Override @Override @Override @Override @Override
12  int msg = 11; @Override @Override @Override @Override @Override
13  @Override @Override
14
15  -- By the message handler
16  @Override @Override @Override @Override @Override
17  void handleMessage() {
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```



Correct
programs often have
an equivalent
sequentialization

Our approach:
compute
sequentialization

Help writing distributed programs

```
20
29 coordPid (Commit p) = p
30 coordPid (Rollback p) = p
31
32 forEach :: SymSet i -> (i -> Process a) -> Process ()
33 forEach xs body
34   = foldM (\_ i -> body i >> return ()) () xs
35
36 acceptor :: Process ()
37 acceptor = do
38   me          <- getSelfPid
39   (who, fn)    <- expect :: Process (ProcessId, String)
40
41   -- Do the transaction
42   exists <- liftIO $ doesDirectoryExist fn
43   let msg = if exists then Accept me else Reject
44   send who msg
45
46   -- Wait for message to commit
47   msg <- expect :: Process CoordMessage
48
49   send (coordPid msg) ACK
50
```

We want to

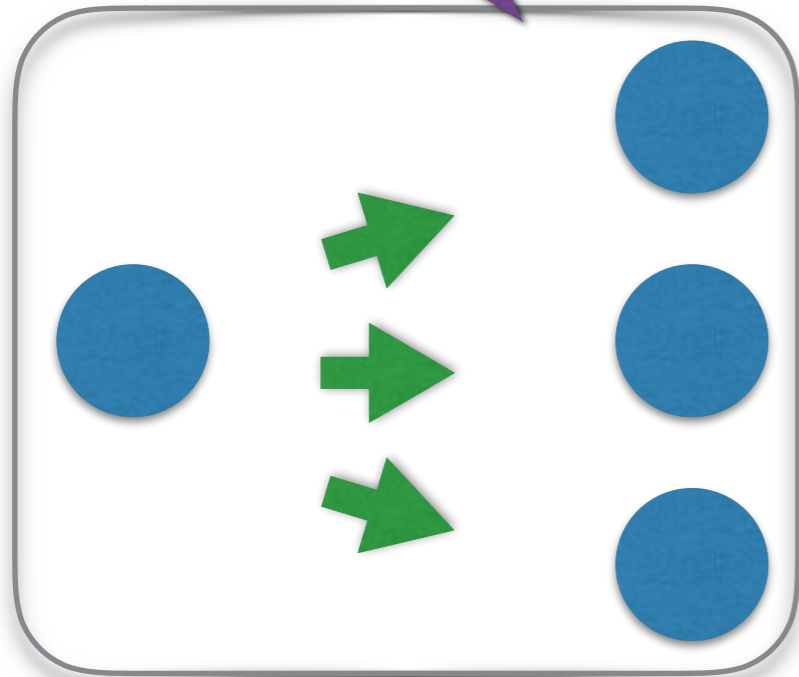
prove
absence of
deadlocks

quick
feedback
while compiling

no manual
proofs

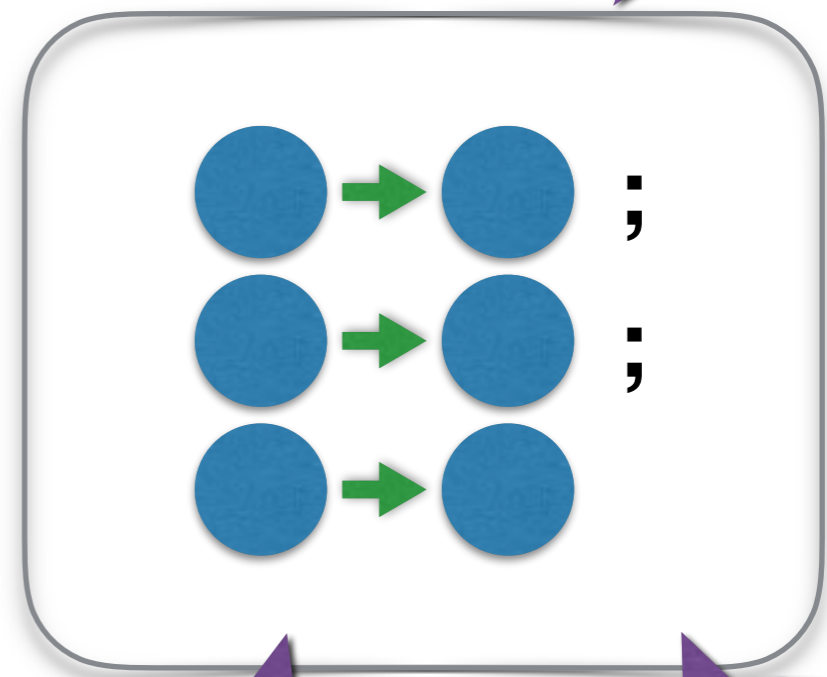
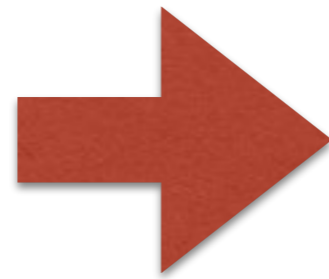
Canonical Sequentialization in Brisk

Our approach:
compute **canonical
sequentialization**



no
sequentialization =
likely wrong

existence
implies **deadlock
freedom**



same **halting
states**

check
additional safety
properties

on
**simpler,
sequential
program**

Results

$\leq 100\text{ms}$

micro
benchmarks

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

Firewall, Map
Reduce, 2PC

Distributed file
system

Outline

The Dream

The Problems

The Key Idea

The Implementation

The Evaluation