COMP3221
Distributed Systems

Blockchain

Presented by A/Prof. Vincent Gramoli

THE UNIVERSITY OF SYDNEY

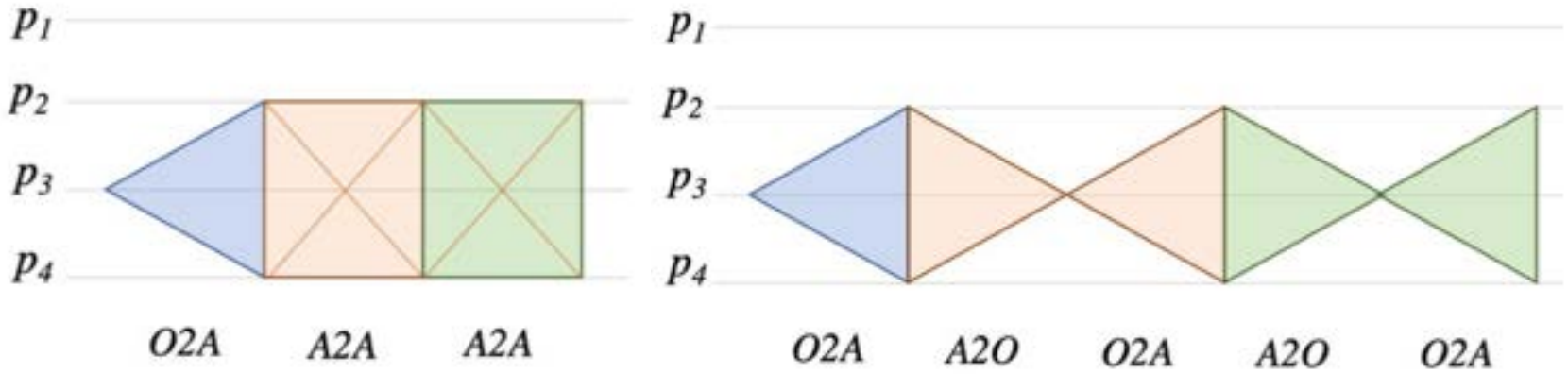
https://www.bankrate.com/investing/best-blockchain-etfs/

Robustness

Leader-based Consensus

Leader-based consensus algorithms

Most blockchain consensus algorithms are leader-based in that a single process has to play a leader role, broadcasting to all nodes or aggregating results from all nodes.



They are typically variants of PBFT [CL02].

[CL02] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst., 20(4):398-461, Nov. 2002.

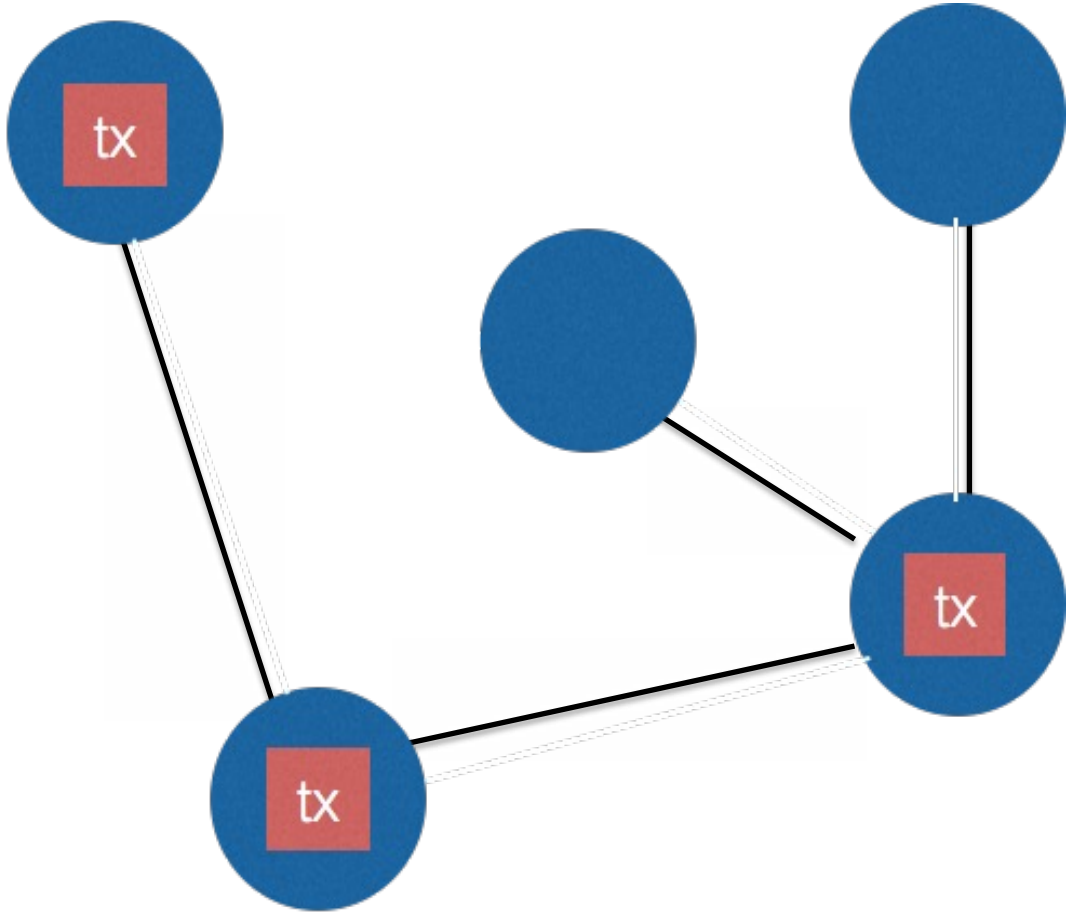
Leader-based consensus algorithms

The process that plays the role of the leader changes over time.

It is hard for a client (e.g. wallet) to know which process is the leader to send a request to.

Easier to:

- send transaction requests to the processes that will run the consensus
- send their request to the processes that are the closest geographically



Leader vs. leaderless consensus algorithms

Consider a consensus algorithm that decides a *set of transactions* of B bits

We want to compare the performance of:

- A leader-based solution
- A leaderless solution

In a leader-based algorithm, the set of transactions is proposed by the leader

In a leaderless algorithm, the set of transactions is proposed by all the participating processes

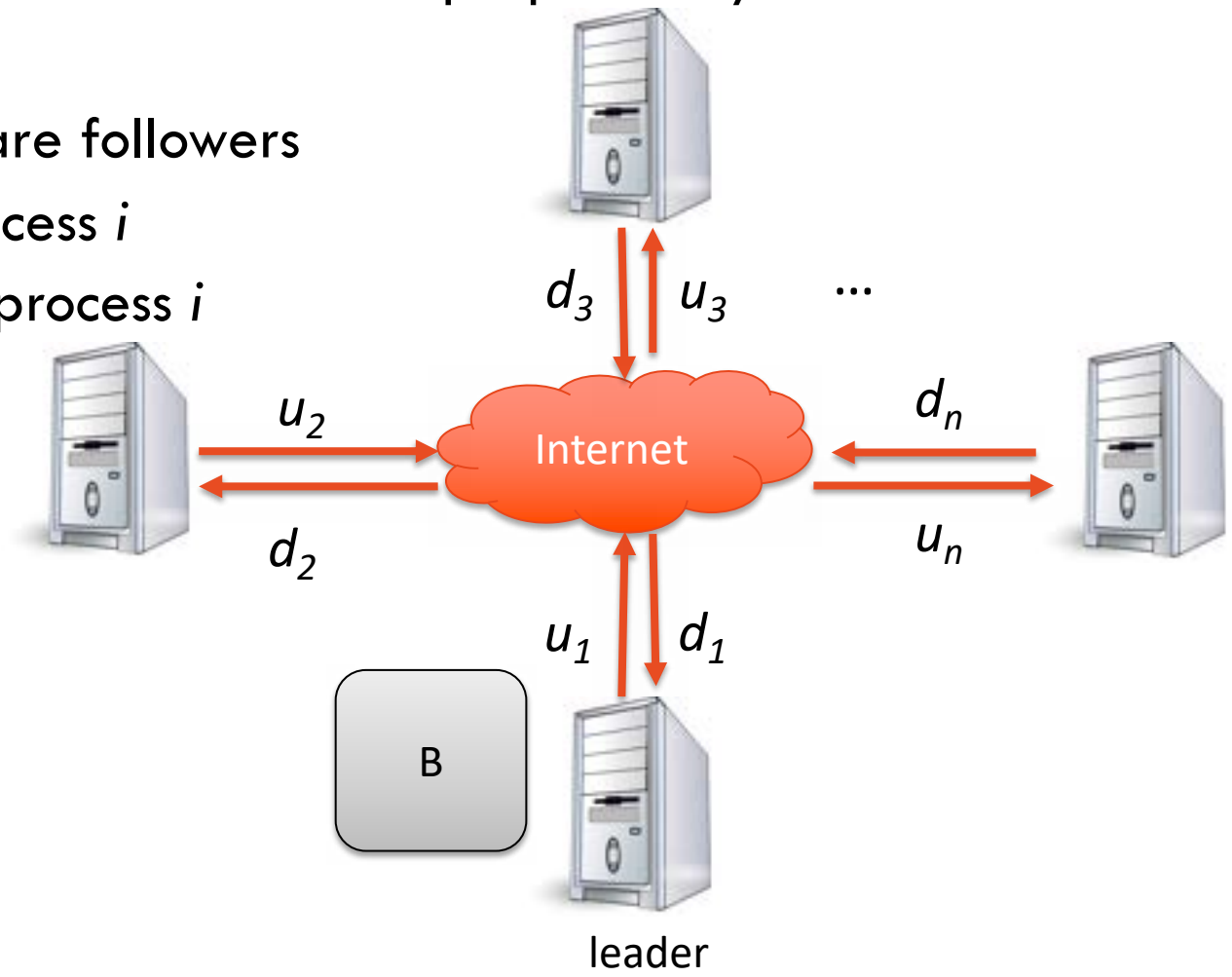
Block exchange time in leader-based algorithms

In a leader-based algorithm, the set of transactions is proposed by the leader

Let process 1 be the leader, others are followers

Let u_i be the upload capacity of process i

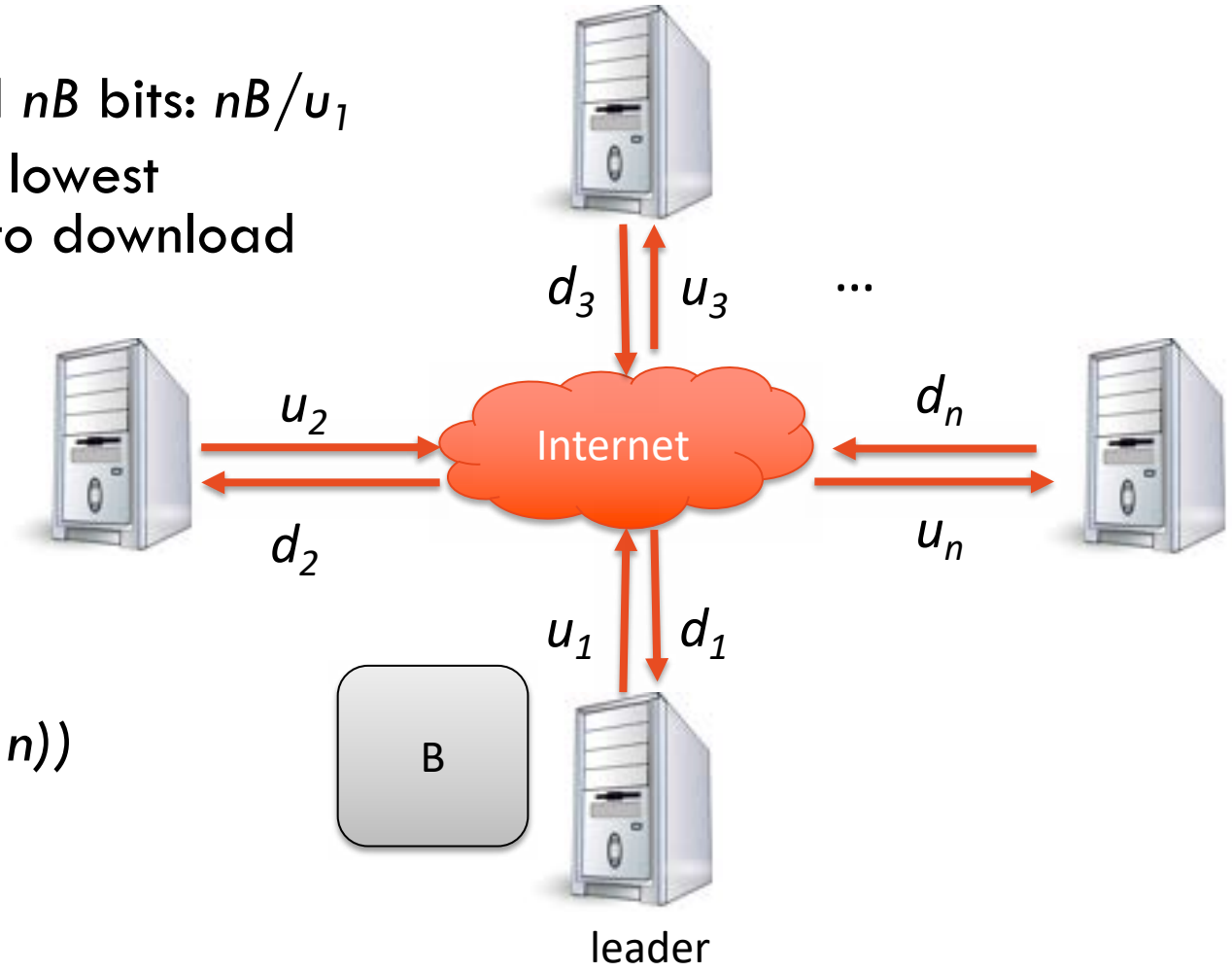
Let d_i be the download capacity of process i



Block exchange time in leader-based algorithms (con't)

The time P it takes to exchange the set of transactions from the leader to all other processes is the maximum between:

- The time for the leader to upload nB bits: nB/u_1
- The time for the follower with the lowest download rate $min(d_i: 1 \leq i \leq n)$ to download B bits: $B/min(d_i: 1 \leq i \leq n))$



$\Rightarrow P = max(nB/u_1, B/min(d_i: 1 \leq i \leq n))$
$= O(nB)$

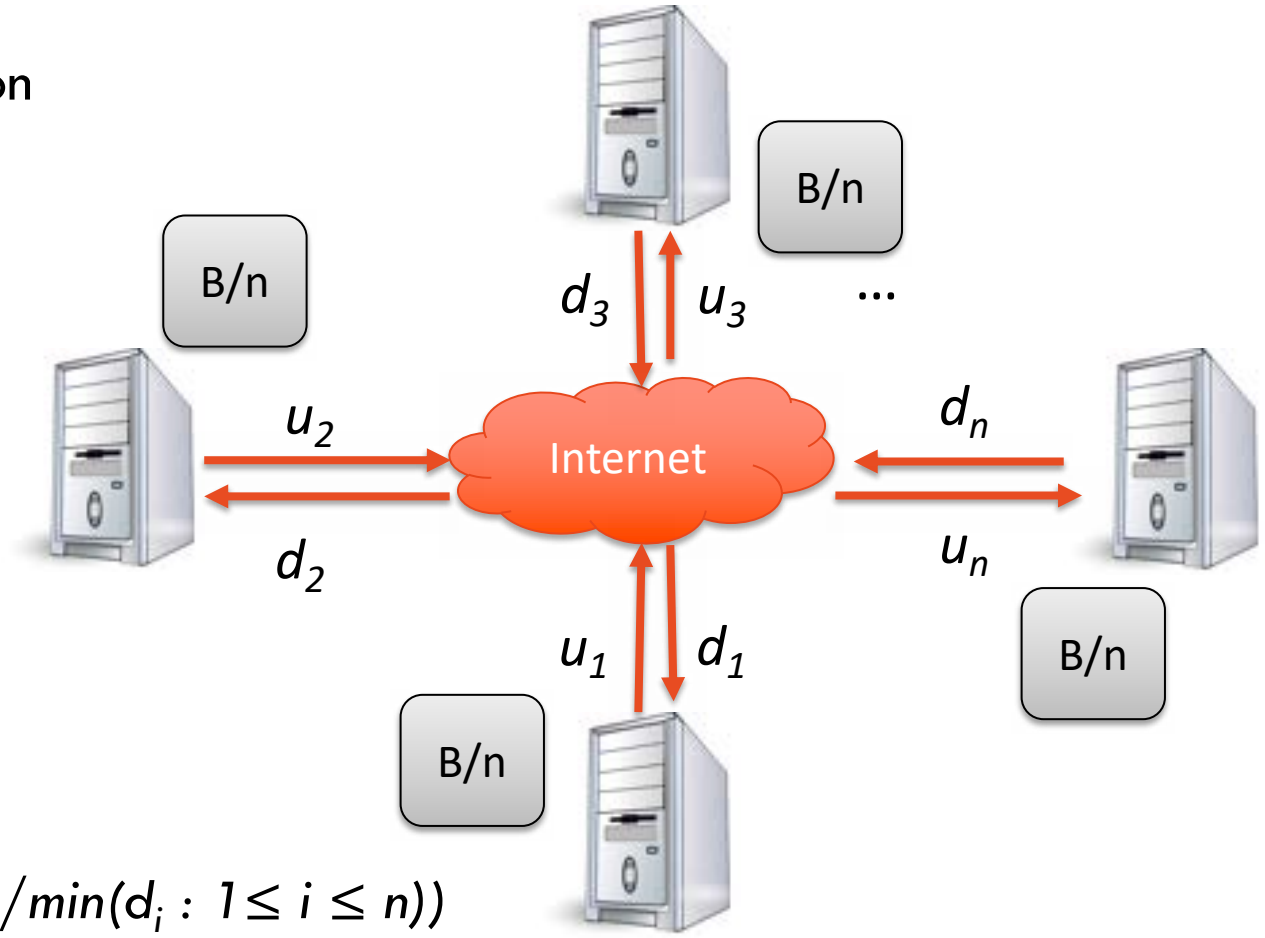
Block exchange time in leaderless algorithms

In a leaderless algorithm, the set of transactions is proposed by all the participating processes

Consider that each process has a portion B/n of the total set of transactions

The time it takes to exchange these sets of transactions is the max of:

- The time for the slowest process to upload B/n bits
- The time for the slowest process to download B bits



$\Rightarrow P = max((B/n)/min(u_i : 1 \leq i \leq n), (B/min(d_i : 1 \leq i \leq n))$

$= O(B)$

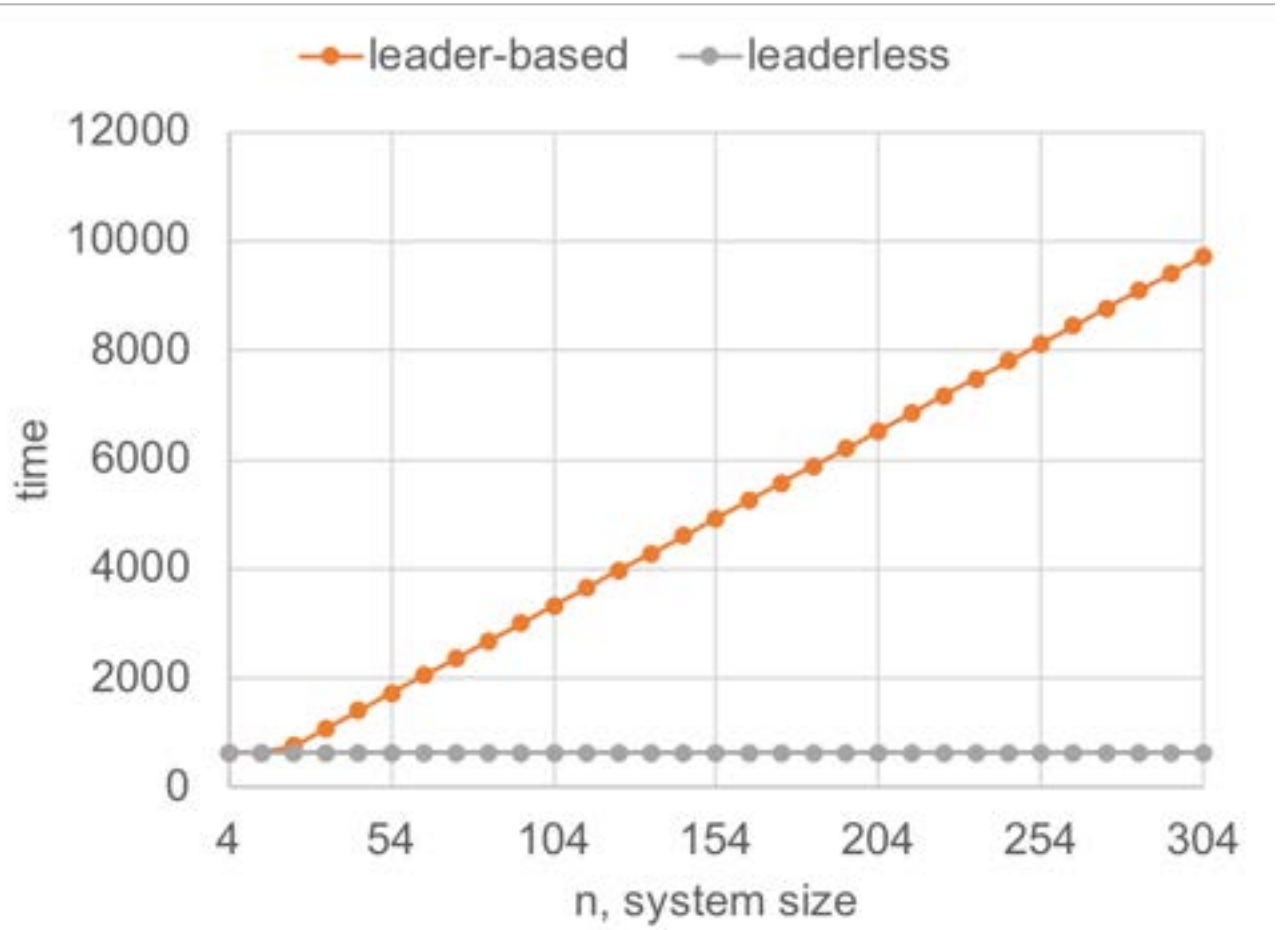
Leader vs. leaderless block exchange time

Consider that the leader has 10 times higher upload rate than other processes

And that every process download 2 times faster than it uploads

The time it takes for all processes to be aware of the set of transactions is the same on a small network

When the number of processes reaches 21, then the leaderless algorithm starts being faster and faster



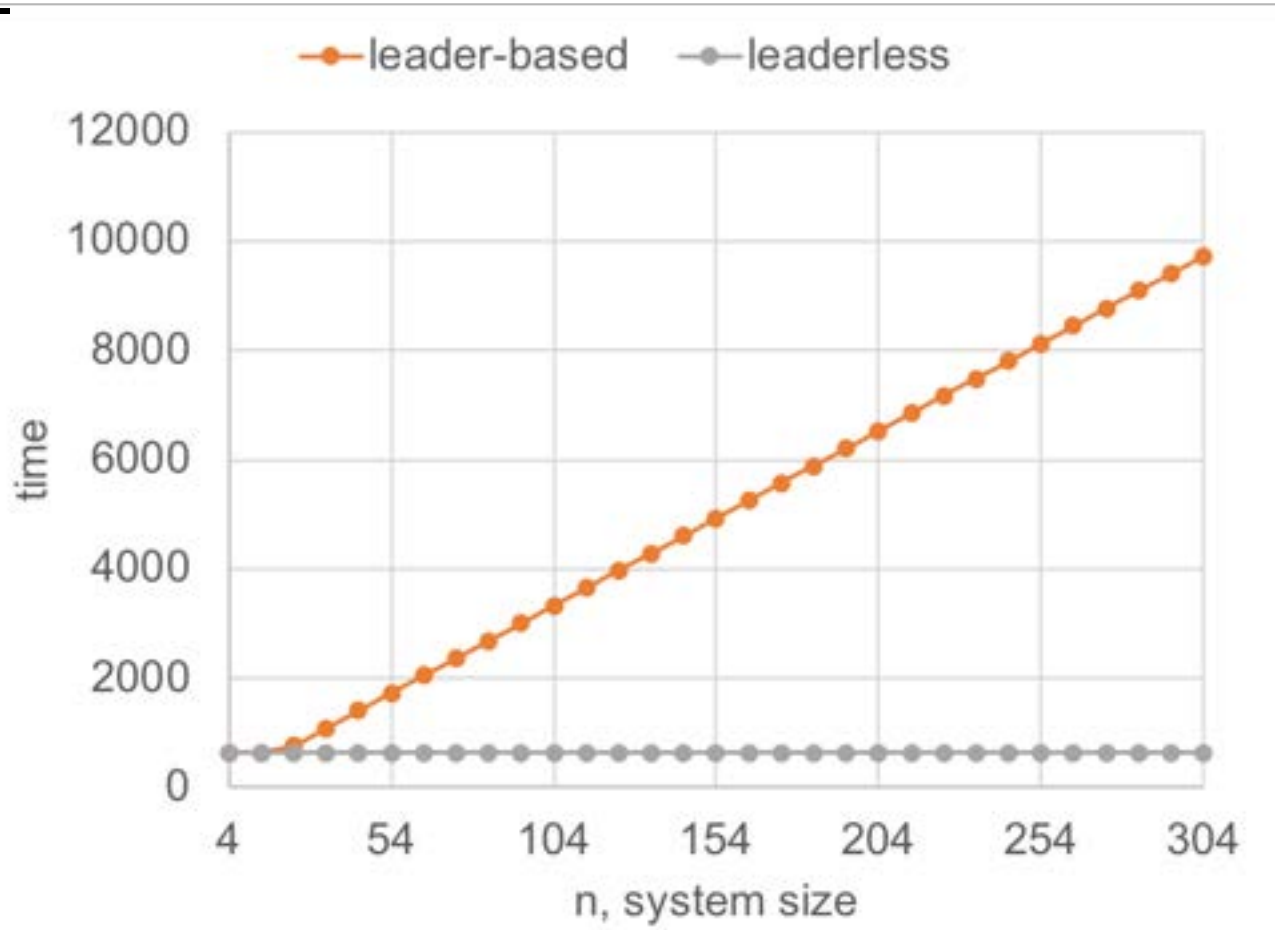
Leader vs. leaderless block exchange time

To conclude, the performance of a leader-based algorithm would typically not grow with the amount of the resources used in the system.

Because the time to propagate the block to the followers is limited by:

- the upload capacity of the leader and
- the lowest download capacity of the followers

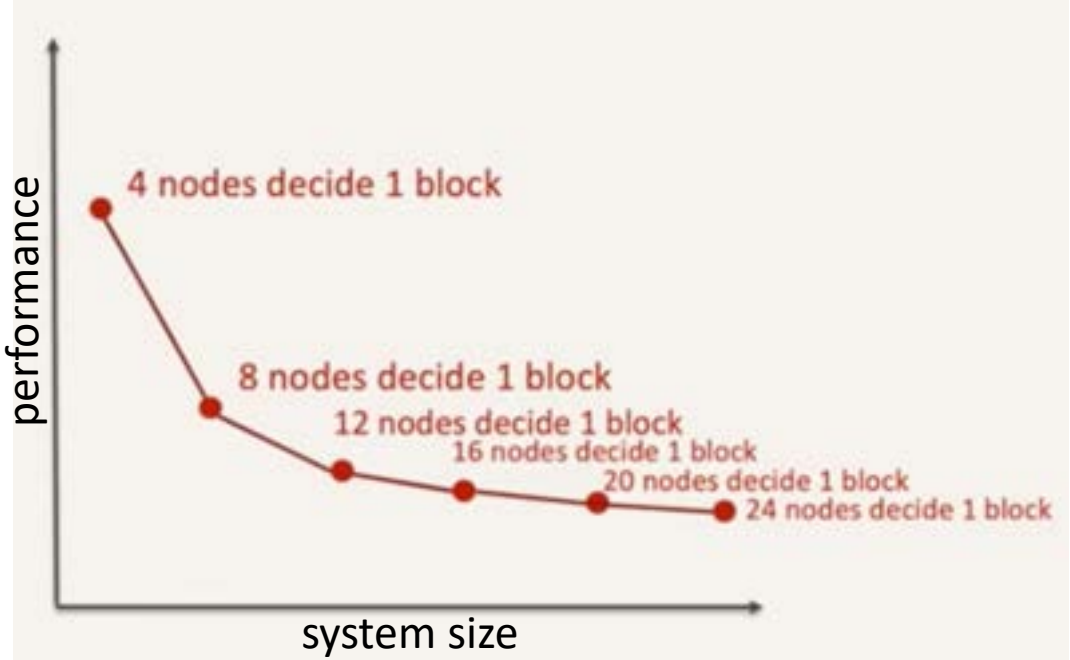
Leaderless algorithms do not have this limitation.



The Set Consensus Problem

Traditional Consensus does not Scale

Traditional blockchain consensus algorithms are leader-based. They inherit the same scalability limitation of other leader-based algorithms.



In addition, the leader tries to impose its block, disallowing other nodes from proposing blocks. This means that the amount of blocks/transactions appended to the blockchain at the end of the consensus does not change (regardless of the number of validators).

Example: Facebook and IBM Blockchains

Facebook developed a blockchain called Libra, later renamed into Diem
It relies on the HotStuff Byzantine Fault Tolerant (BFT) consensus algorithms

IBM helped developed HyperledgerFabric whose secure version relies on BFTSmart consensus algorithm.

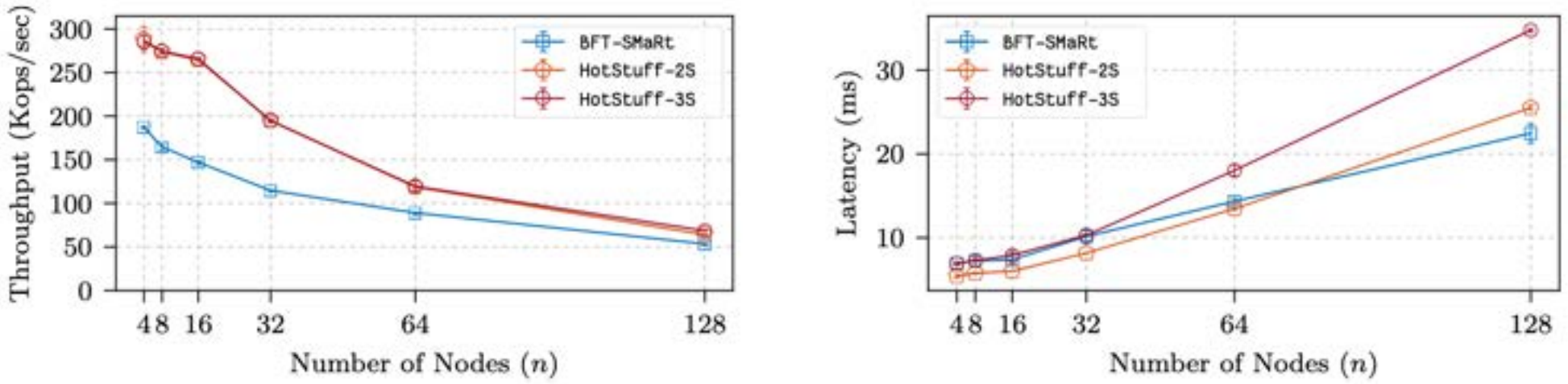


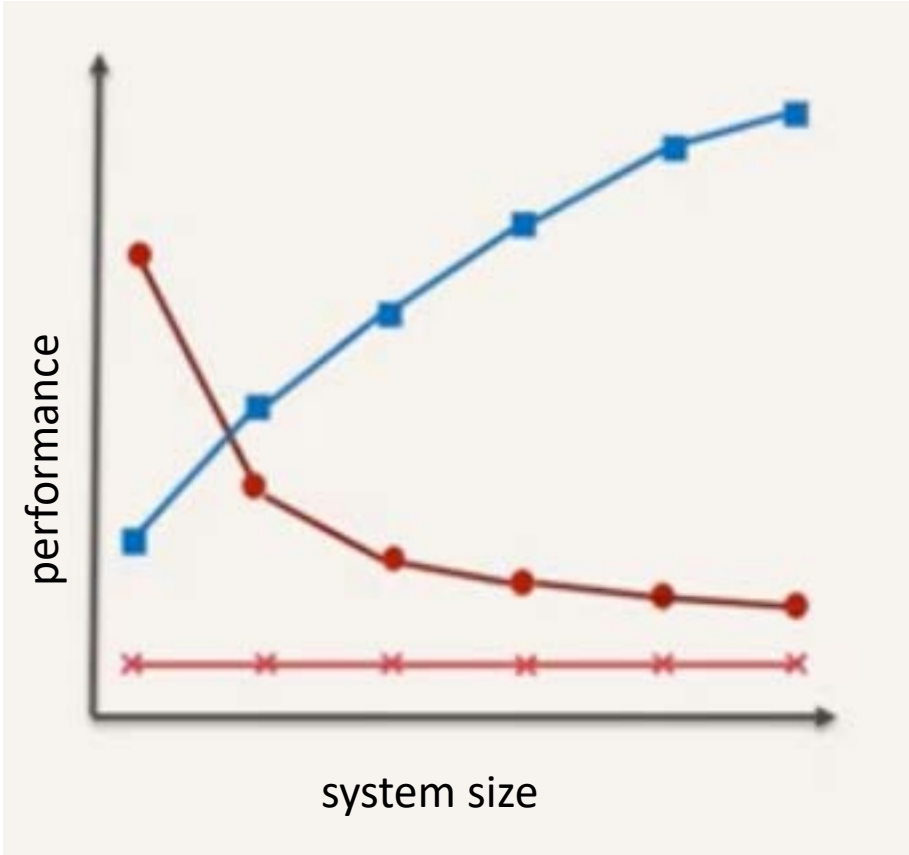
Figure 6: Scalability with 0/0 payload, batch size of 400.

Both are leader-based, when the system size increases, their throughput drops and their latency raises.

A New Consensus Problem Definition is needed for Blockchains

Instead of proposing blocks, processes propose sets of transactions that will be included in the block:

1. All processes propose a set of transactions
2. They exchange these sets among each other
3. They decide a block that:
   - Is not necessarily what any process has proposed
   - Can result from the union of several proposed sets of transactions



The Set Consensus Problem

A set of transactions is *non-conflicting* if it does not contain conflicting transactions.

Assuming that each correct node proposes a proposal s, the Set Byzantine Consensus (SBC) problem is for each of them to decide on a set in such a way that the following properties are satisfied:

- SBC-Termination: every correct node eventually decides a set of transactions;
- SBC-Agreement: no two correct nodes decide on different sets of transactions;
- SBC-Validity: a decided set of transactions is a non-conflicting set of valid transactions taken from the union of the proposed sets; and if all nodes are correct and propose a common valid non-conflicting set of transactions, then this subset is the decided set.

Errors in Byzantine Fault Tolerant Consensus Algorithms

HoneyBadger BFT Consensus Algorithm

The *HoneyBadger Byzantine Fault Tolerant (BFT)* consensus algorithm solves the consensus problem probabilistically by invoking multiple instances of a randomized binary consensus algorithm.

Unfortunately, it does not terminate as a bug was found in the randomized binary consensus algorithm presentation. It is thus possible for the algorithm to enter an infinite loop.

Let us see the algorithm to understand how this algorithm cannot terminate.

HoneyBadger BFT Consensus Algorithm

- Relies on randomized Byzantine fault tolerant consensus protocol [PODC'14]

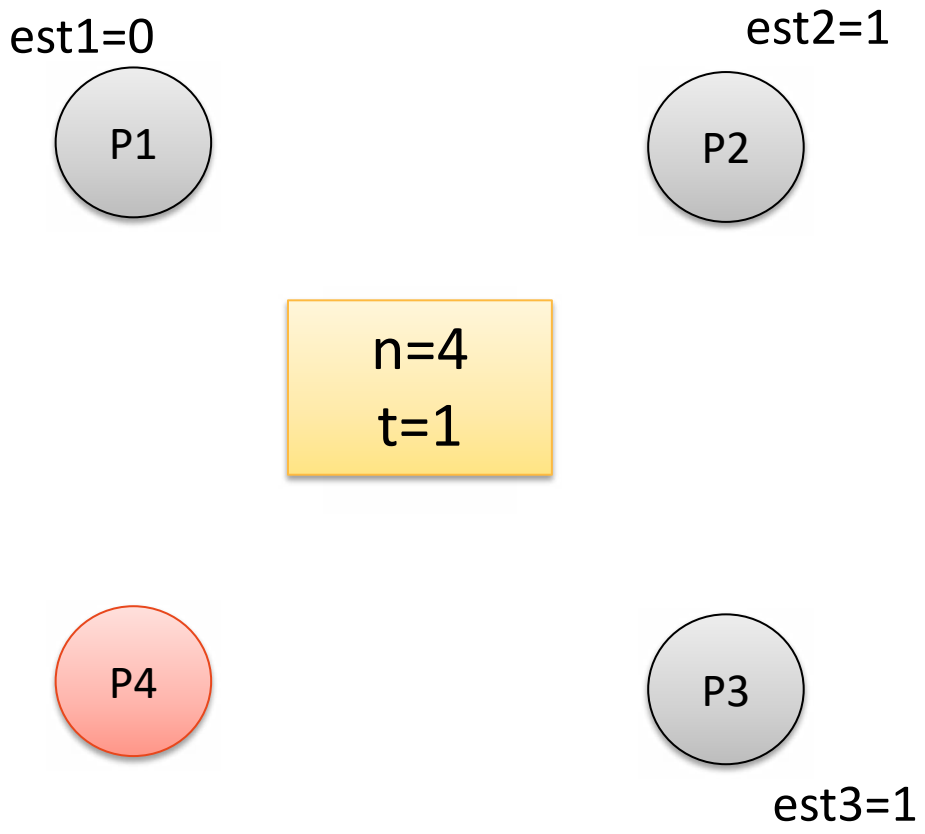
```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.  if (vals = {v}) {
9.    est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

- That in turns relies on
  - Binary value broadcast bv-broadcast() for processes to exchange binary values
  - Common coin common-coin-random() that returns the same random results at all processes
  - Point-to-point reliable channels, n > 3t, asynchrony

Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

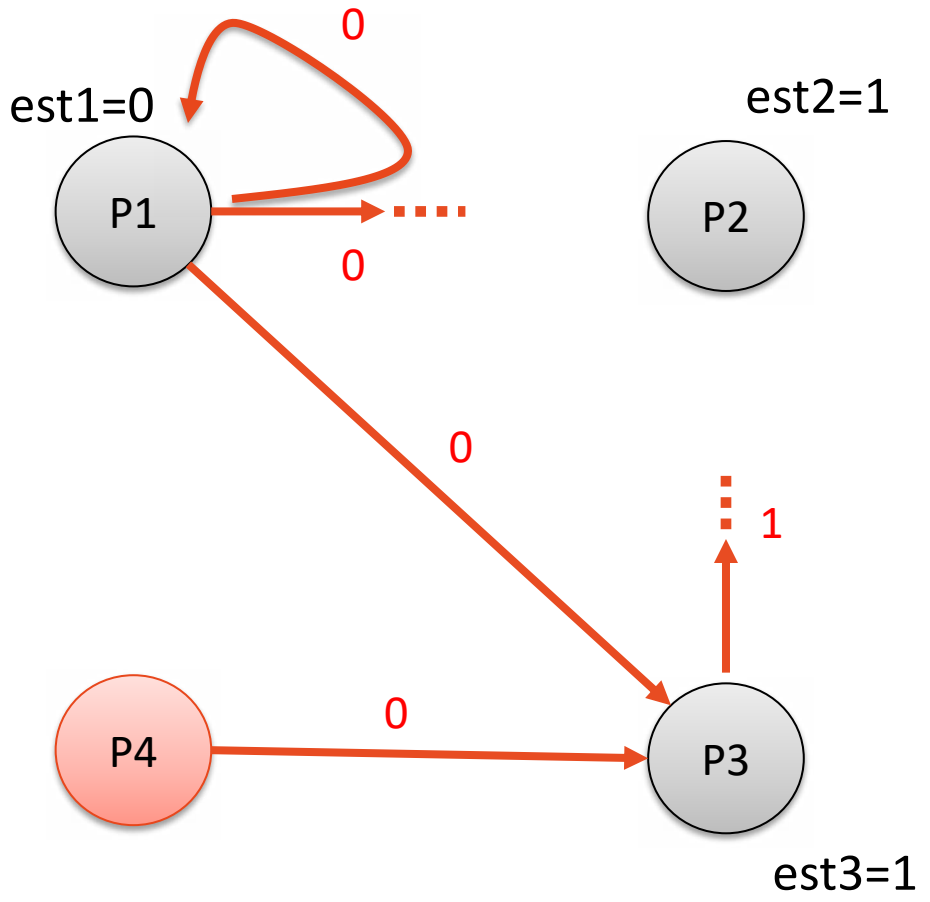
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

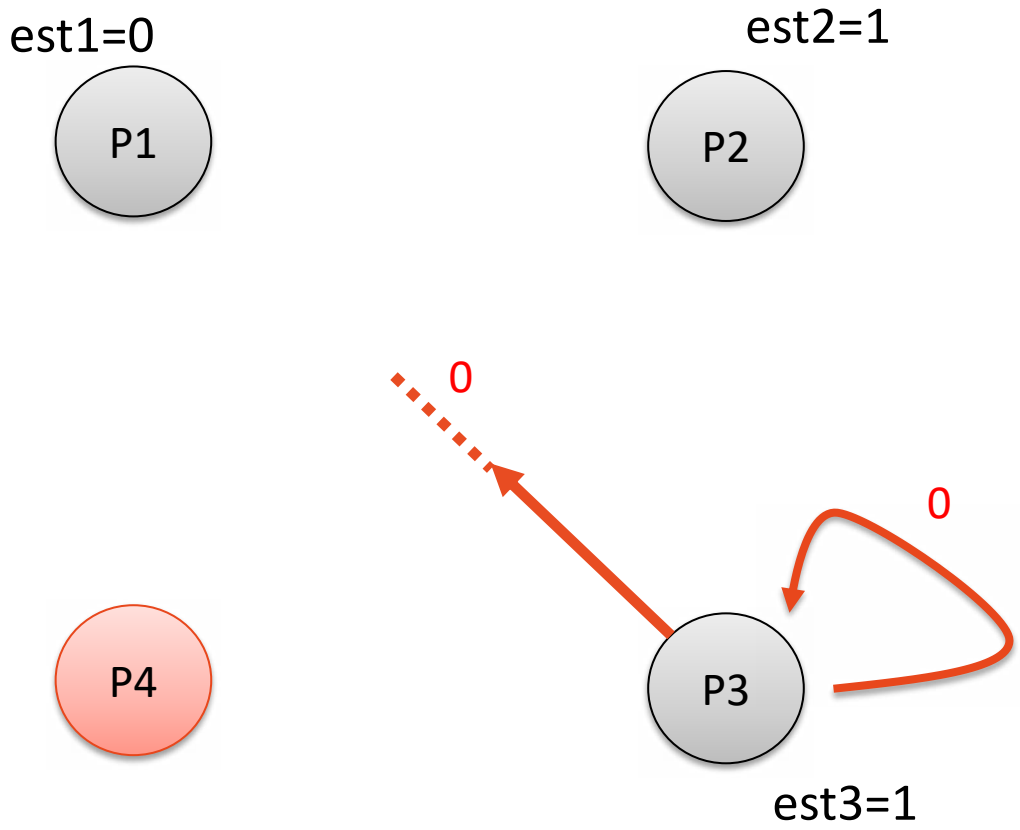
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

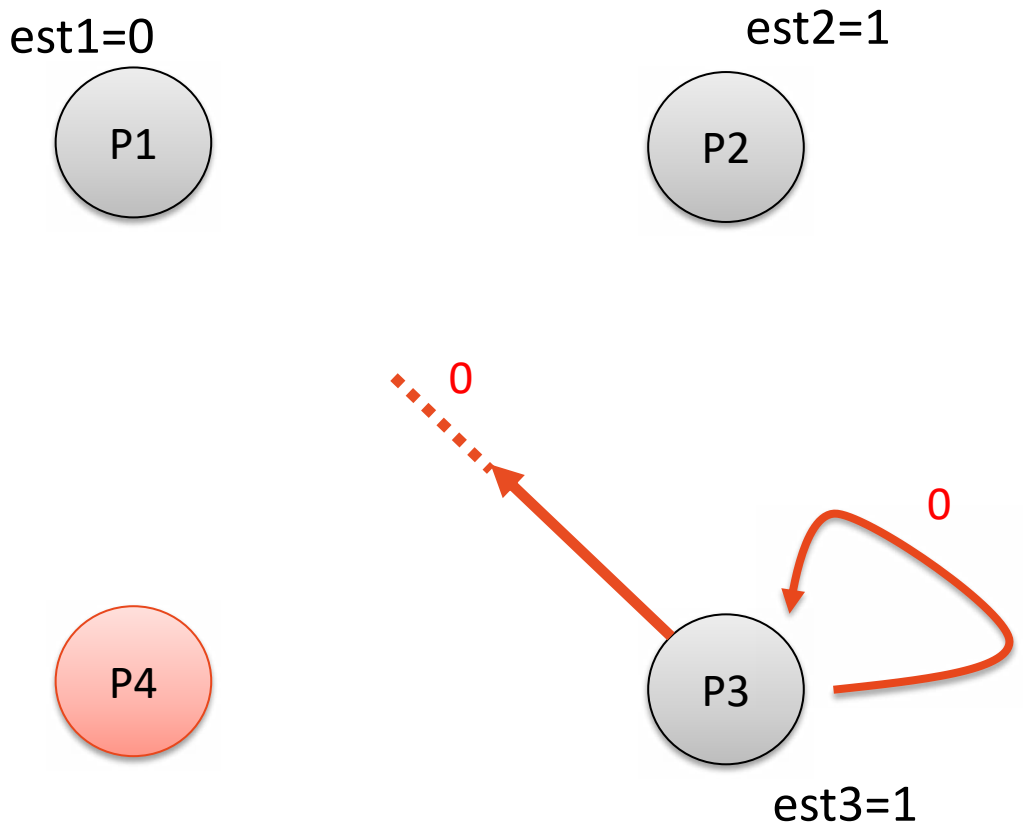
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

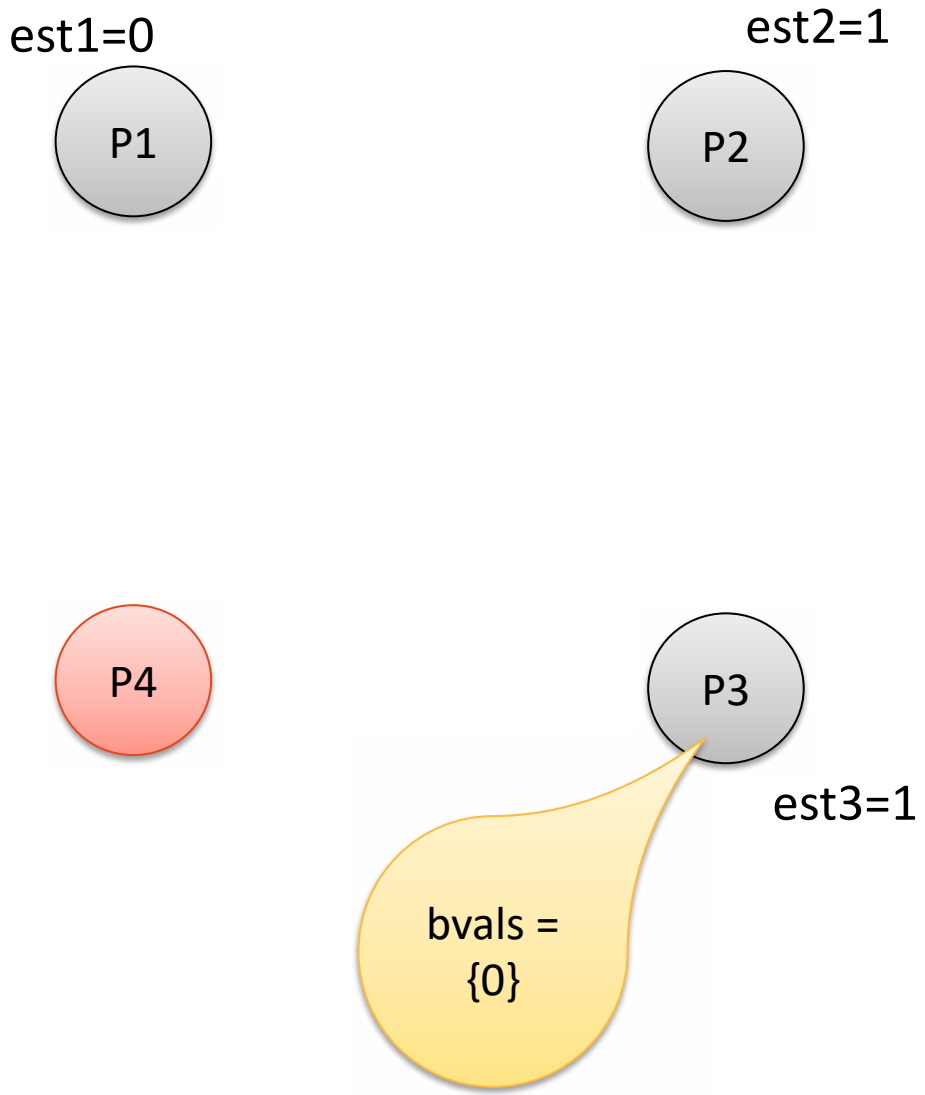
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

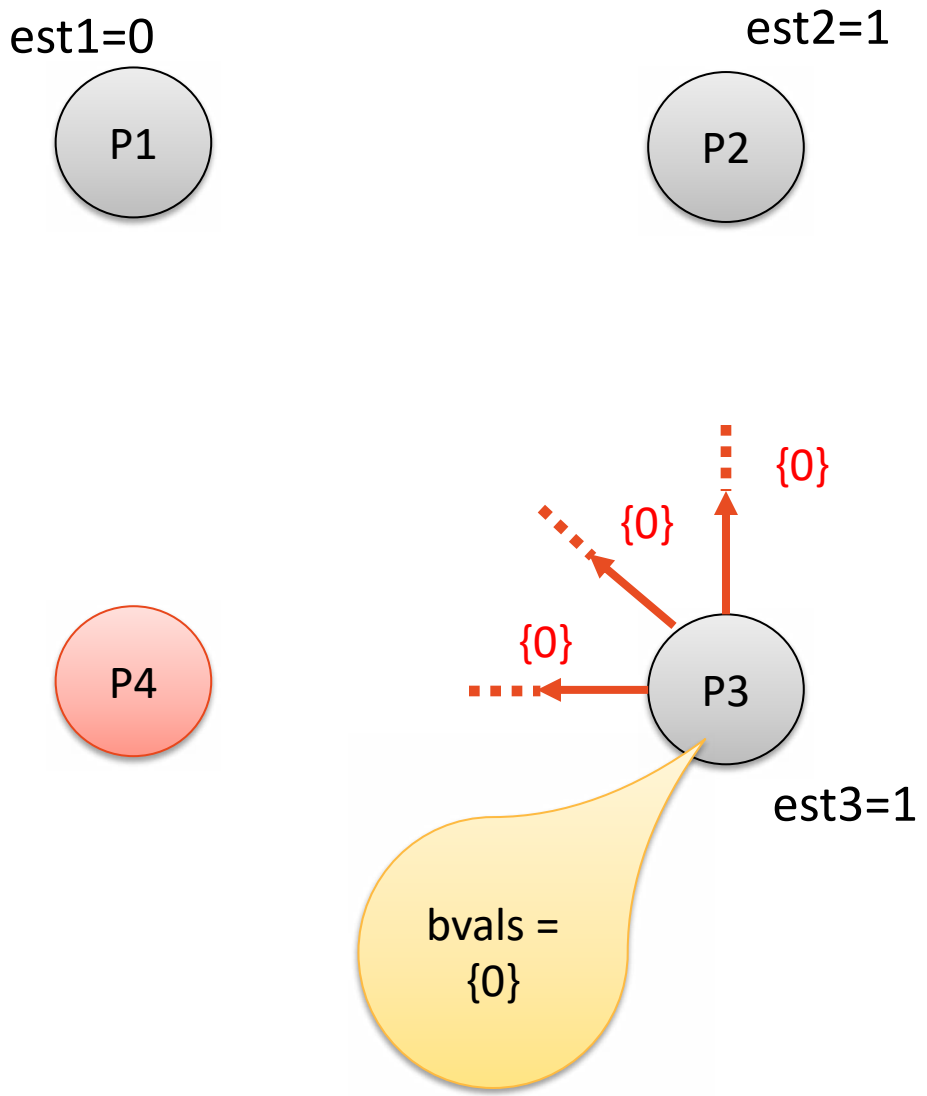
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

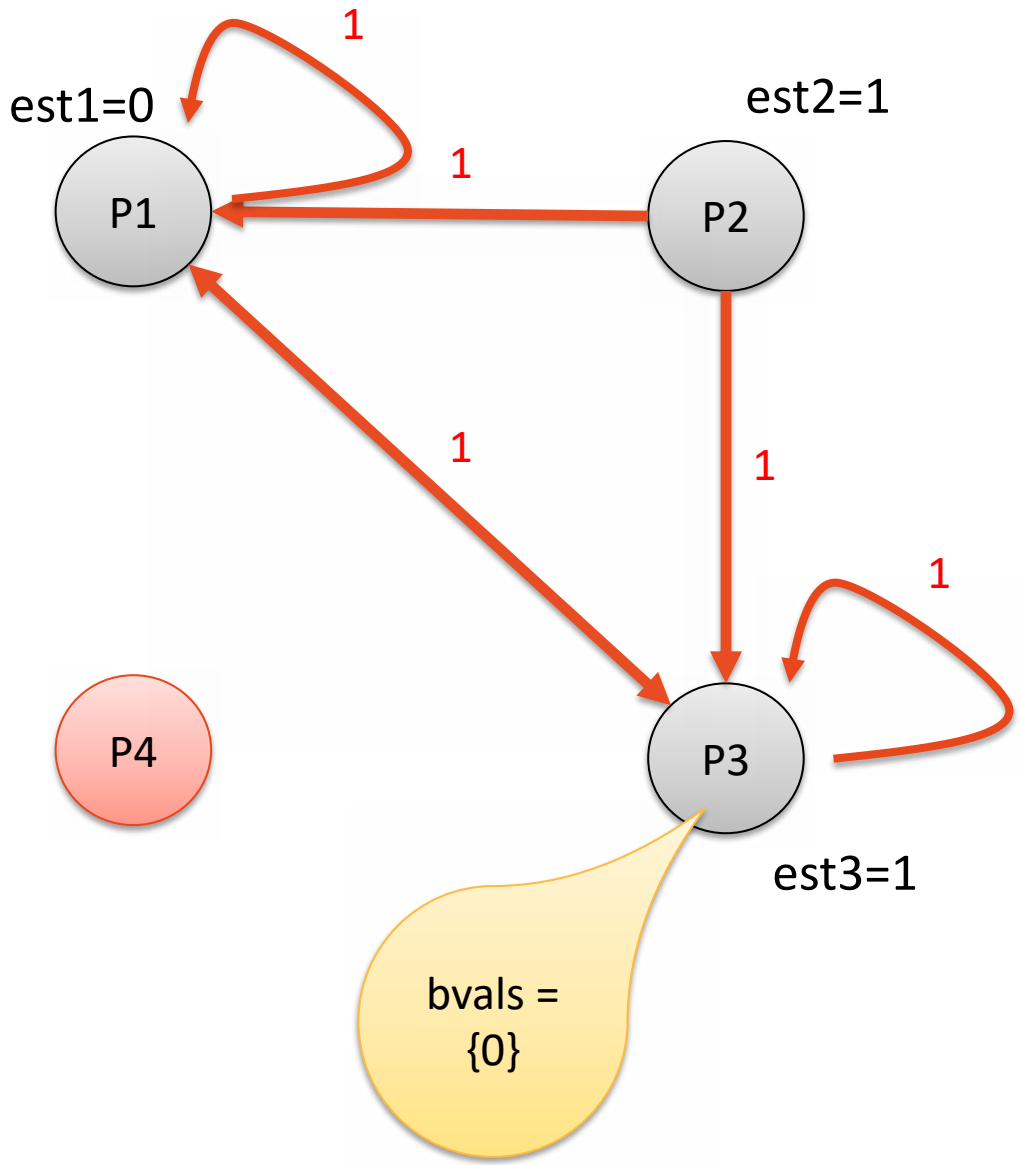
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

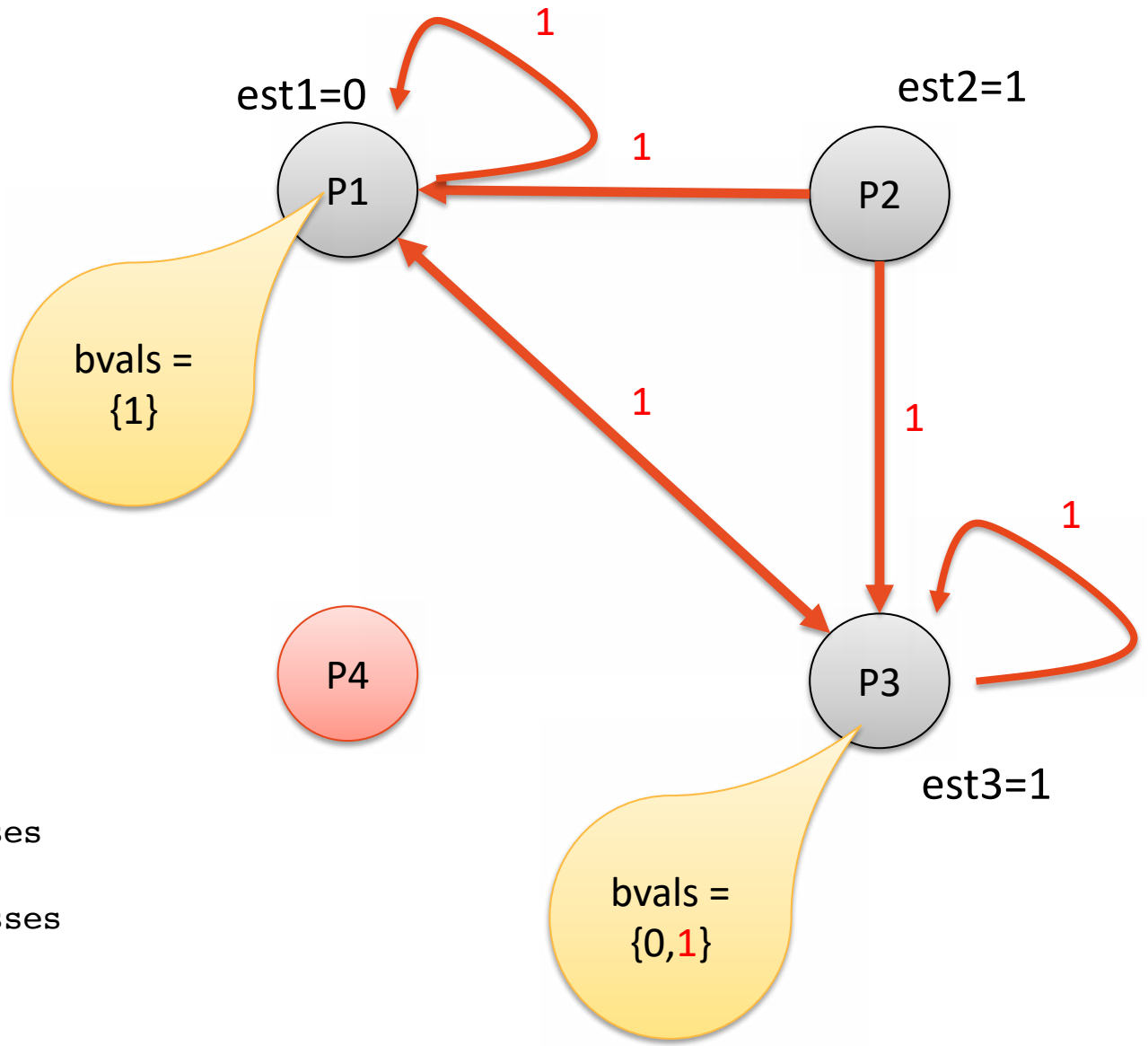
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

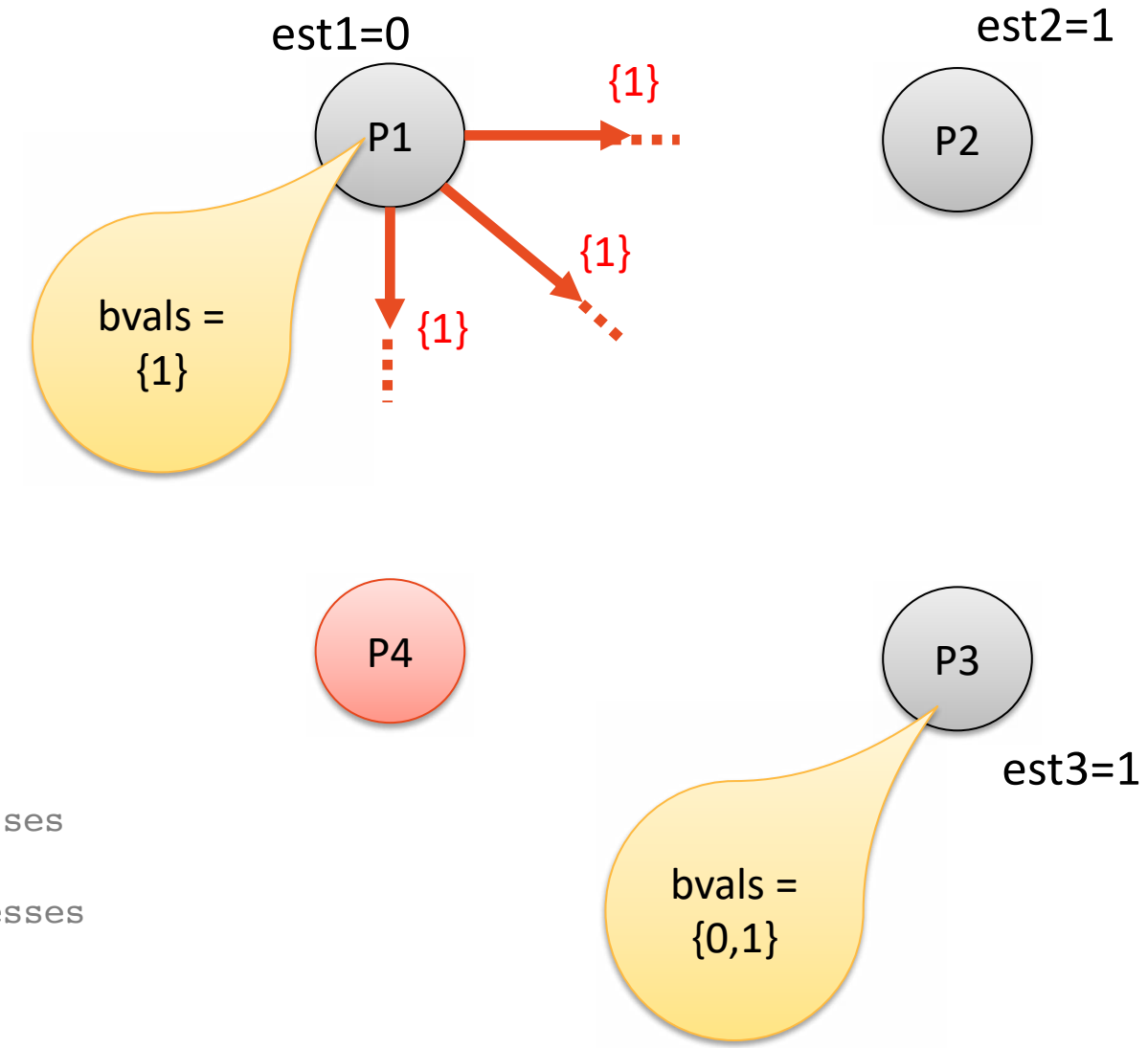
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

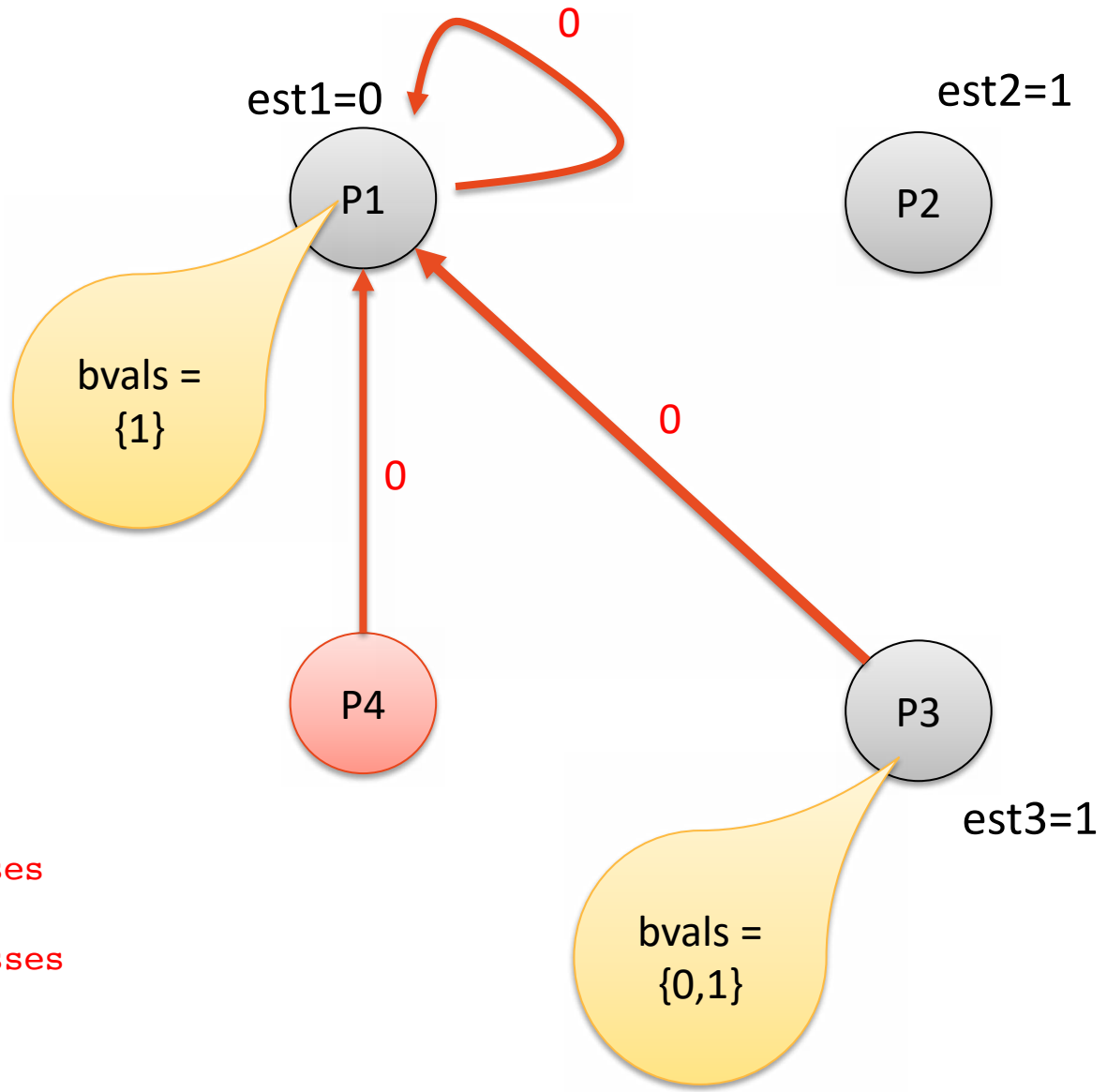
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

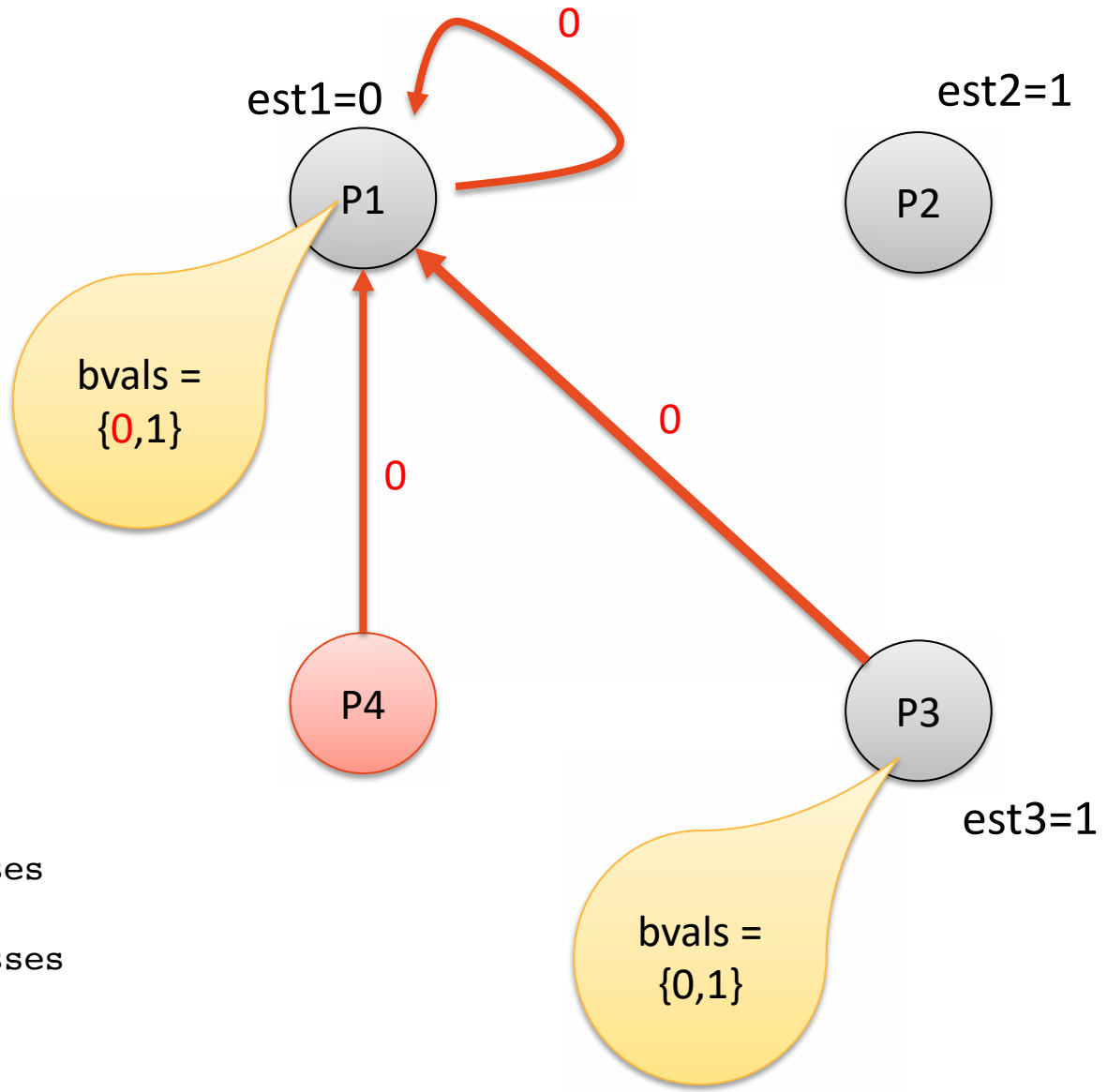
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

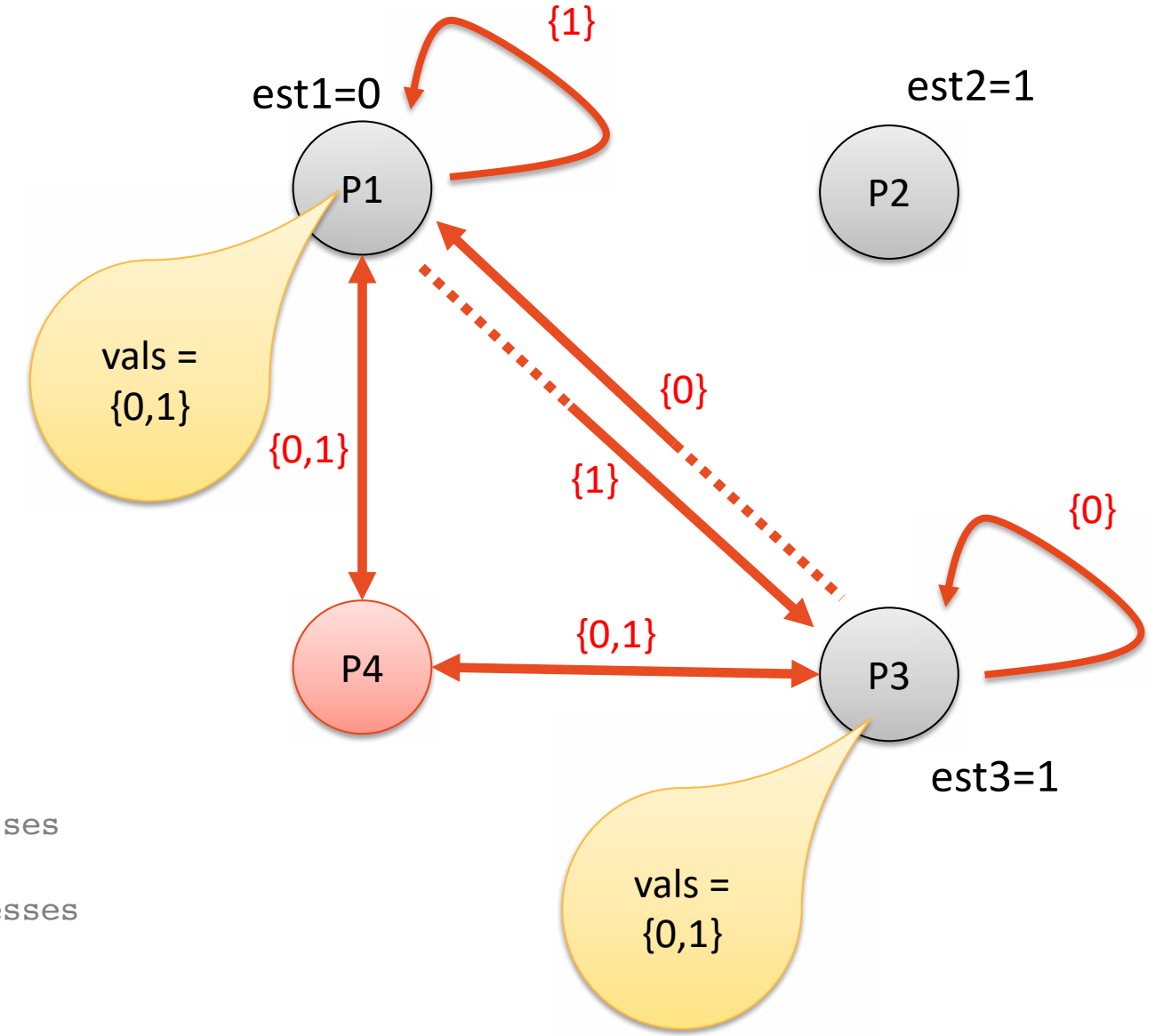
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.   if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.   if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

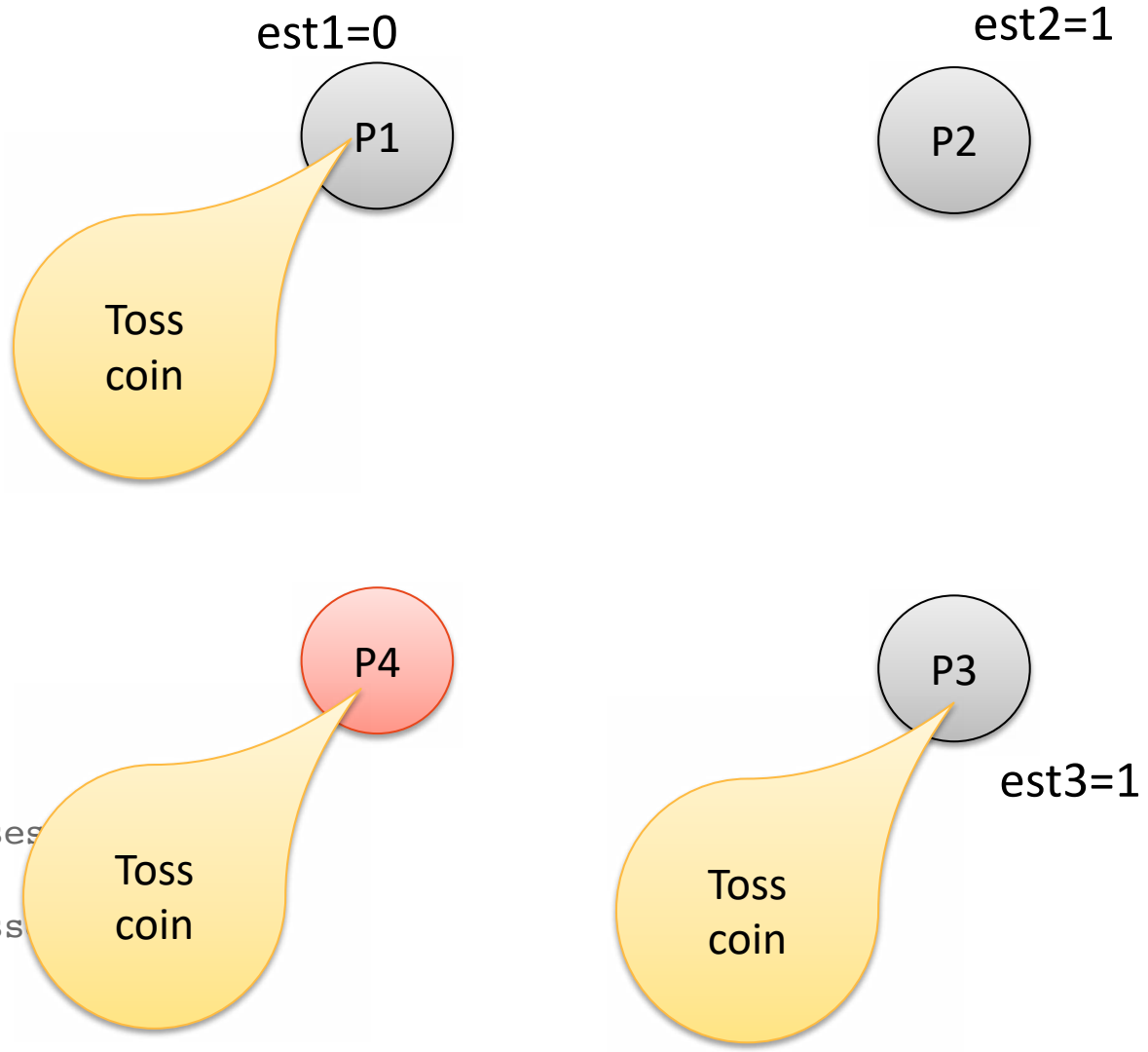
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



Termination counter-example

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

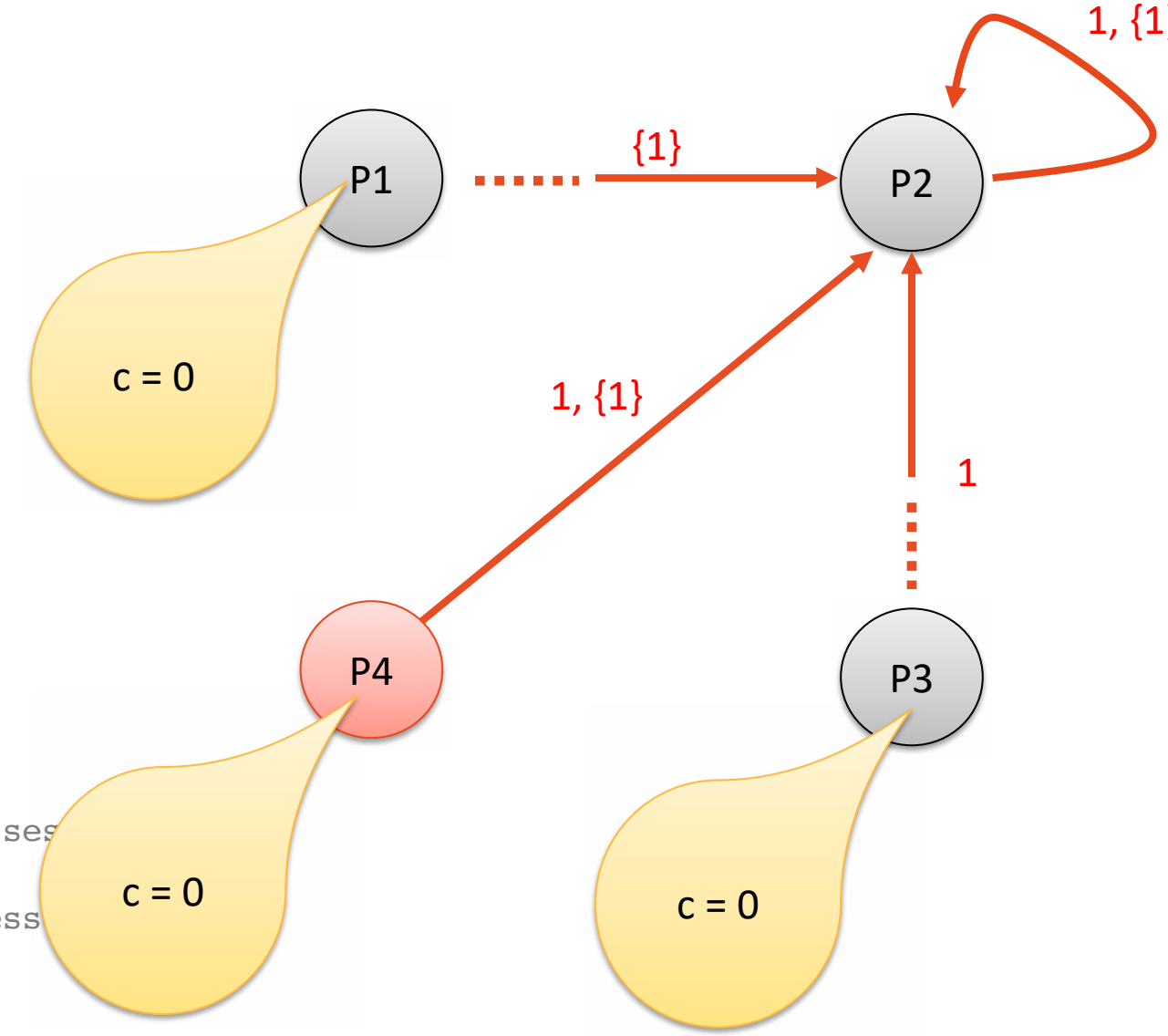
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct process
20.   then bvals ← bvals U {v}
```



If the coin returns 0

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

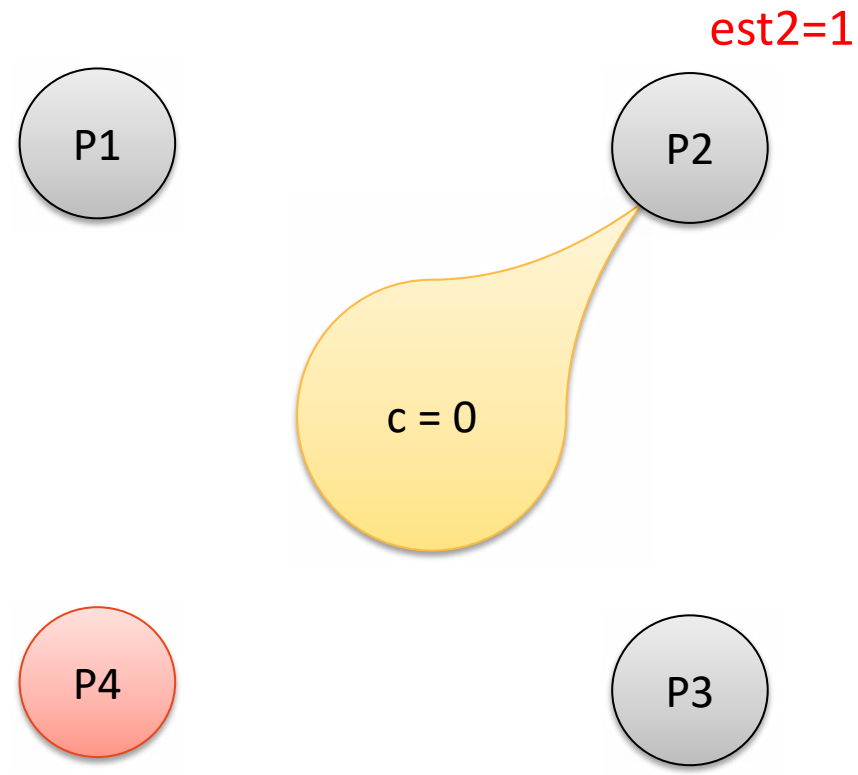
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct process
20.   then bvals ← bvals U {v}
```



If the coin returns 0

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

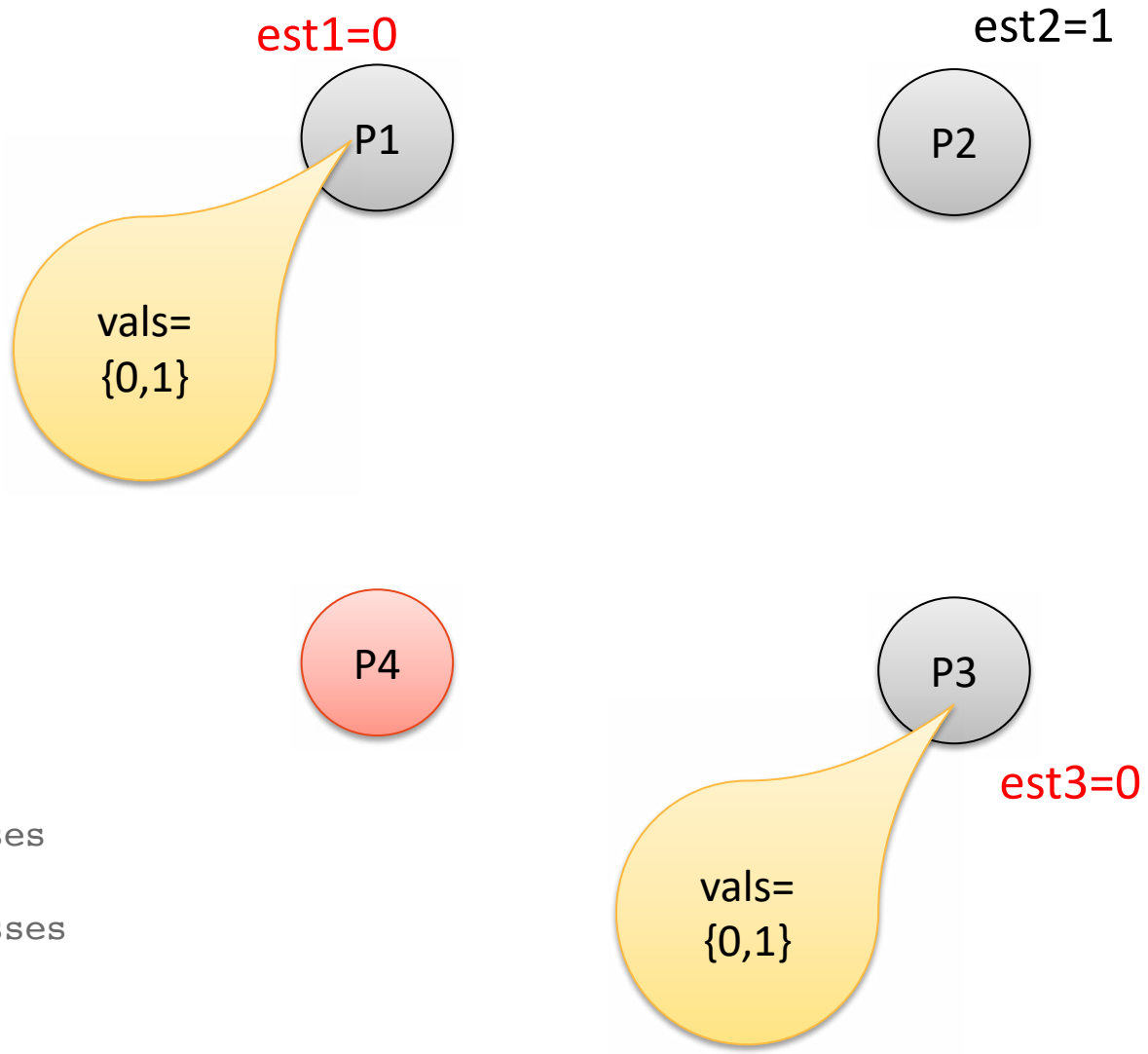
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



If the coin returns 0

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

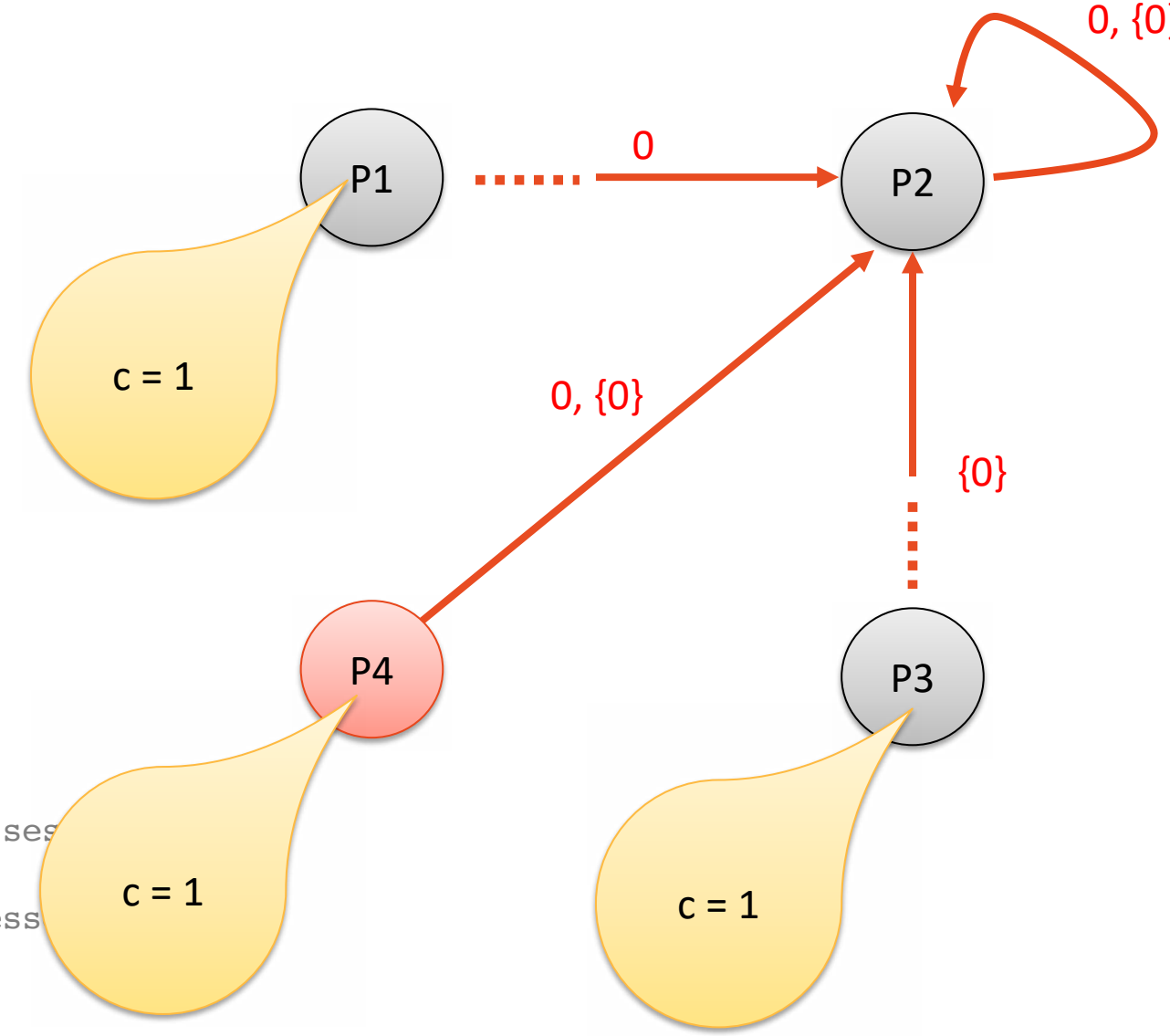
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



If the coin returns 1

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

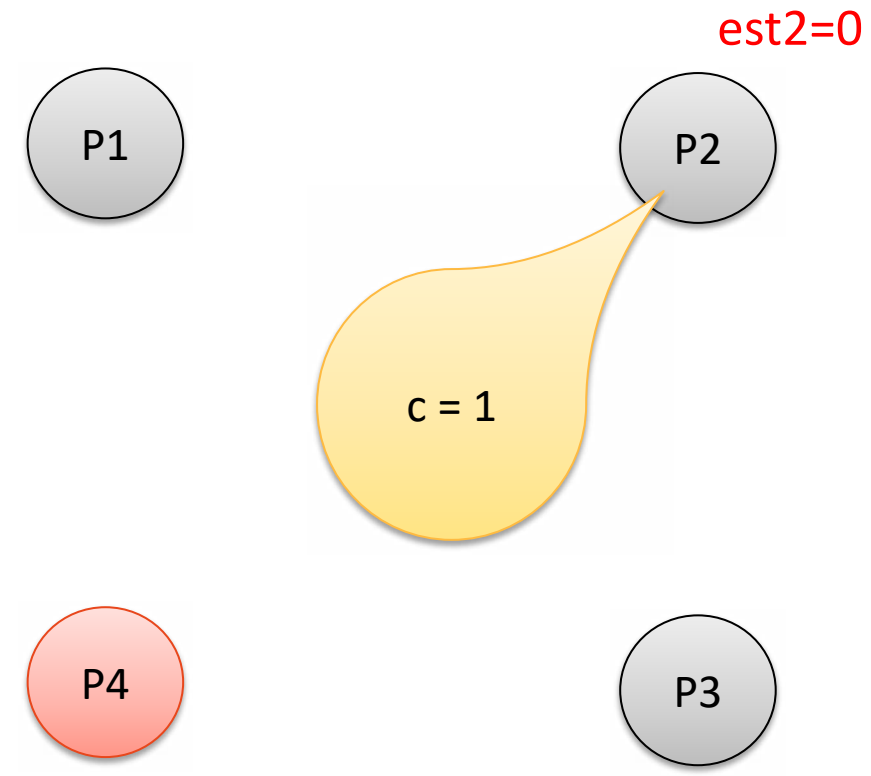
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct process
20.   then bvals ← bvals U {v}
```



If the coin returns 1

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

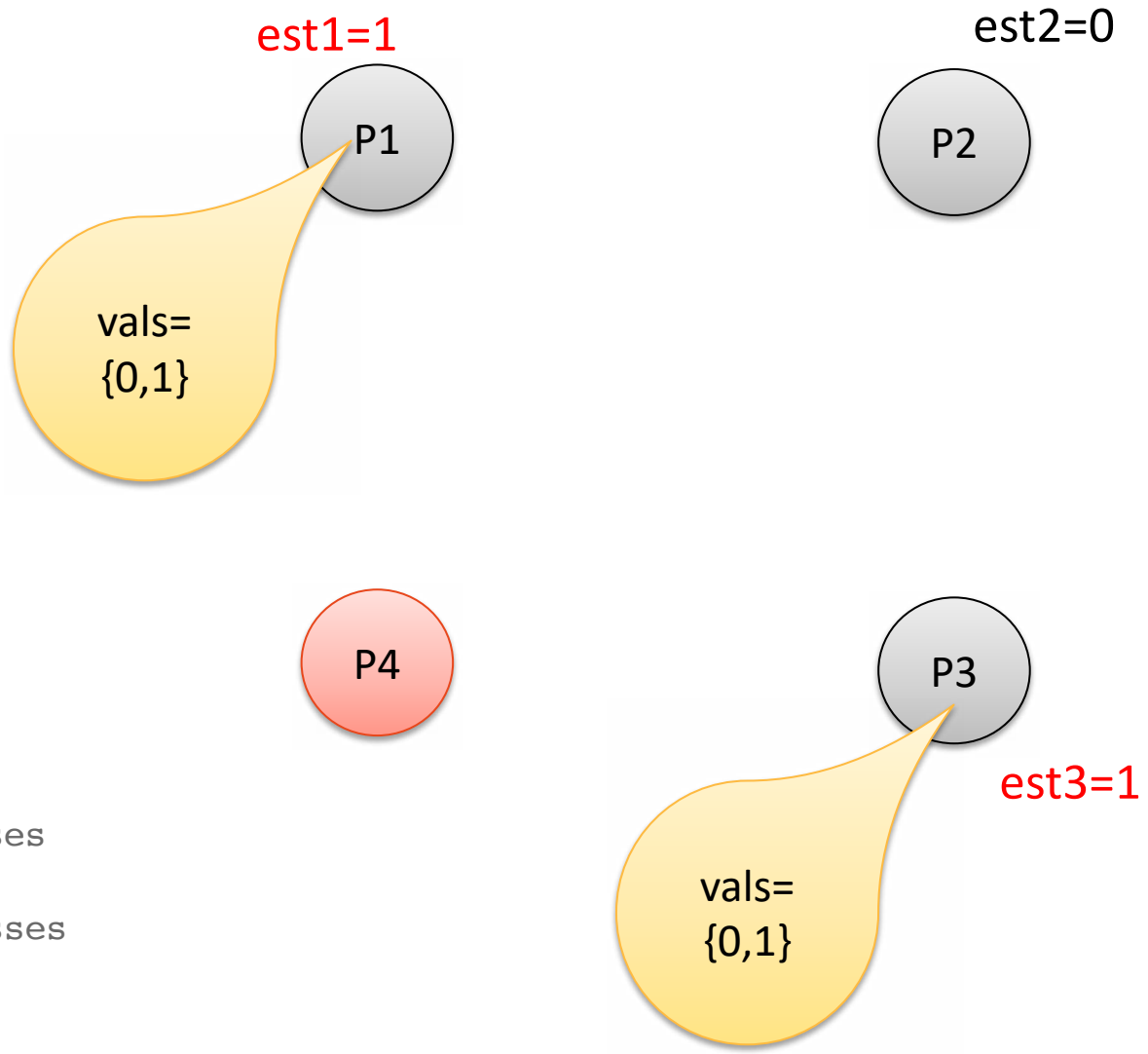
```
15. bv-broadcast(EST, est, i) {
16.   broadcast(BV, est)
17.    if <BV, v> received from t+1 distinct processes
18.   then broadcast <BV,v>
19.    if <BV, v> received from 2t+1 distinct processes
20.   then bvals ← bvals U {v}
```



If the coin returns 1

```
1. est ← v
2. while (true) {
3.   bv-broadcast(EST[r], est, i) → bvals
4.   wait until (bvals[r] ≠ ∅)
5.   broadcast(ECHO[r], bvals[r]) → msgs
6.   wait until (vals = compute(msgs, bvals)) ≠ ∅
7.   c ← common-coin-random()
8.   if (vals = {v}) {
9.     est ← v
10.    if (v = c) then decide(v) and exit
11.  } else {
12.    est ← c
13.  }
14. }
```

```
16. bv-broadcast(EST, est, i) {
17.   broadcast(BV, est)
18.   if <BV, v> received from t+1 distinct processes
19.   then broadcast <BV,v>
20.   if <BV, v> received from 2t+1 distinct processes
21.   then bvals ← bvals U {v}
```



Errors in BFT consensus algorithms

Consensus algorithms are complicated

Byzantine fault tolerant (BFT) consensus algorithms are particularly complicated. One has to make sure that they solve consensus despite the arbitrary behaviour of some participants. Even though brilliant researchers write the proofs of correctness of these algorithms, there is a risk that they make mistakes.

Several errors have been identified in blockchain consensus protocols and some of them can be exploited by hackers.

Democratic Byzantine Fault Tolerant Consensus Algorithm

DBFT Consensus Algorithm

The *Democratic Byzantine Fault Tolerant (DBFT)* consensus algorithm solves the consensus problem (deterministically).

- It solves the Set Consensus Problem
- It invokes multiple instances of a binary consensus algorithm

In addition to be proven by hand, it has been proven correct using formal verification. This means that a software called a model checker tested that any execution of the protocol solves the consensus problem.

Let us now see a simplified version (the safe version) of the binary consensus algorithm at the heart of DBFT by looking at the differences with the previous probabilistic consensus algorithm.

Safe DBFT Binary Consensus of Process p_i [NCA'18]

```
1. est ← v
2. r ← 0
3. while (true) {
4.   r ← r+1
5.   bv-broadcast(EST, est) → bvals
6.   wait until (bvals ≠ ∅)
7.   broadcast(ECHO, bvals) → msgs
8.   wait until (vals = compute(msgs, bvals)) ≠ ∅
9. s ← global-coin-random()
9.   if (vals = {v}) {
10.    est ← v
11.    if (v = r mod 2) then decide(v) and exit
12.  } else {
13.    est ← r mod 2
14.  }
15. }

16. bv-broadcast(EST, est, i) {
17.   broadcast(BV, est)
18.   if <BV, v> received from t+1 distinct processes
19.   then broadcast <BV,v>
20.   if <BV, v> received from 2t+1 distinct processes
21.   then bvals ← bvals U {v}
```

Agreement proof

BV-broadcast [JACM'15] ensures:

- BV-justification: If p_i is correct and $v \in bvals_i$, then v has been BV-broadcast by some correct process.

- BV-obligation: If at least $t+1$ correct processes BV-broadcast the same value v, then v is eventually added to $bvals_i$ of each cprrect process p_i

- BV-Uniformity: If a value is added to the set $bvals_i$ of a correct process p_i, then eventually
  $v \in bvals_j$ at every correct process p_j.

- BV-Termination: Eventually the set $bvals_i$ of each correct process p_i is not empty.

Agreement proof

Lemma 1: If at the beginning of a round, all correct processes have the same estimate, they never change their estimate thereafter.

Proof sketch. Assume all correct have the same estimate v at round r. BV-Obligation and BV-Justification => bin_values[r] = {v}. Hence, at every correct we have values = {v}, so that est becomes v.

Agreement proof (con't)

Lemma 2: Let pi and pj be correct. If their values are singletons, then they are the same.

Proof sketch. If a correct process has values = {v} then it received AUX[r]{v} from n-t distinct processes and t+1 correct. For two correct processes to have w and v as this singleton, it would mean they received these distinct values from n-t processes each. As (n-t)+(t+1) > n, one correct process must have sent the same value to both, and we have v=w.

Agreement proof (con't)

Theorem [Agreement]: No two correct processes decide different values.

Proof sketch. Let r be the 1st round where a correct process decides, hence values[r] = {v=r mod 2}. If another correct process decides w in the same round, then v=w by Lemma 2. Let pj be a correct that does not decide in round r. By Lemma 2, $values_j \neq \{w\}$ so $values_j = \{0,1\}$. Thus $est_j = v$. All correct have thus the same estimate in round r+1, and stick to it (Lemma 1).

How do we ensure that the DBFT binary consensus algorithm terminate?

There is no solution to the consensus problem in asynchrony.

We need to assume partial synchrony.

We will also slightly change the Safe DBFT binary consensus algorithm with:

- A weak coordinator that breaks symmetry
- A timer that allows to catchup with the unknown bound on the delay of messages

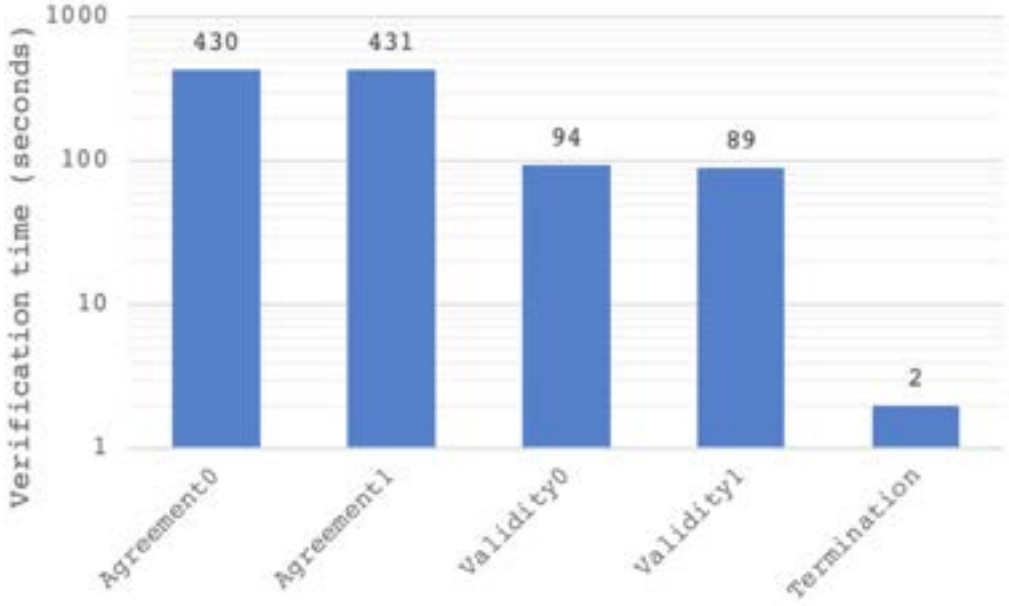
DBFT Binary Consensus of Process p_i [NCA'18]

```
1.  est ← v
2.  r ← 0
1.  while (true) {
2.      bv-broadcast(EST, val, i) → cvals
4.      start-timer()
5.      if (i = r mod n) then
6.        wait until (cvals = {w})
5.         broadcast(COORD, r, w) → c
6.      wait until (cvals ≠ ∅ ∧ timer expired)
7.      if (c ∈ cvals) then e ← {c} else e ← cvals
8.      broadcast(AUX, e) → bvals
9.      wait until (∃s ⊆ bvals) where:
10.      * s contains the values of AUX messages from n-f distinct proc.
11.      * ∀ m ∈ s, m ∈ cvals
12.     if (s = {v}) {
13.       val ← v
14.       if (v = r mod 2) and undecided then decide(v)
15.     else {val ← r mod 2}
16.     if decided in round r - 2 then exit()
17.     r ← r + 1
18.   }
```

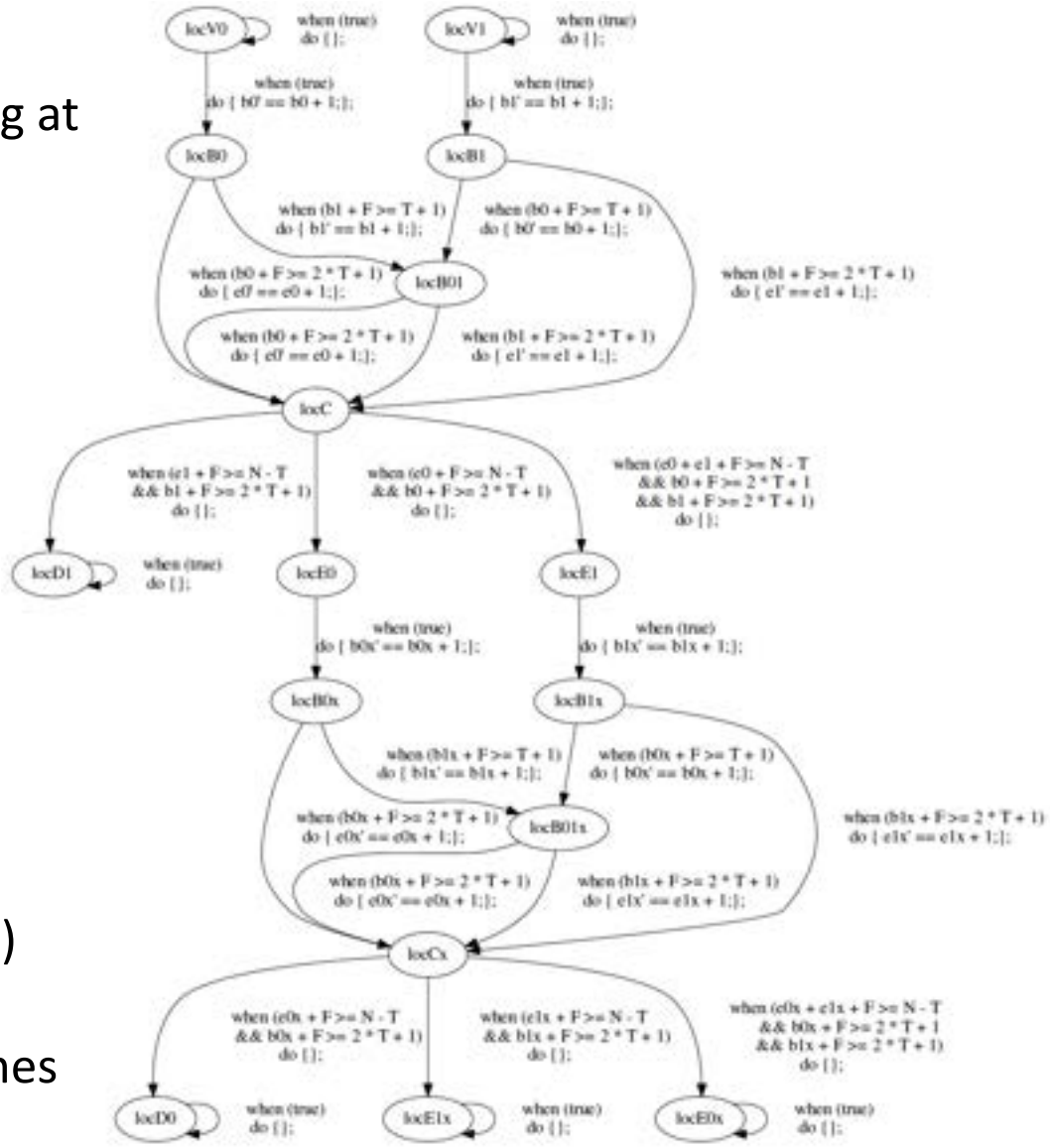
The actual DBFT binary consensus algorithm needs a weak coordinator and some timer in order to terminate.

Formal Verification of the DBFT binary consensus algorithm

On a simple laptop with an Intel Core i5-7200U CPU running at 2.50GHz, verifying all the correctness properties for BV-broadcast took less than 40 seconds.



Parallelized version of the Byzantine Model Checker (ByMC) with MPI on a 4 AMD Opteron 6276 16-core CPU with 64 cores at 2.3 GHz with 64GB of memory. The verification times for the 5 properties was 17 minutes and 26 seconds.



Democratic Byzantine Fault Tolerance

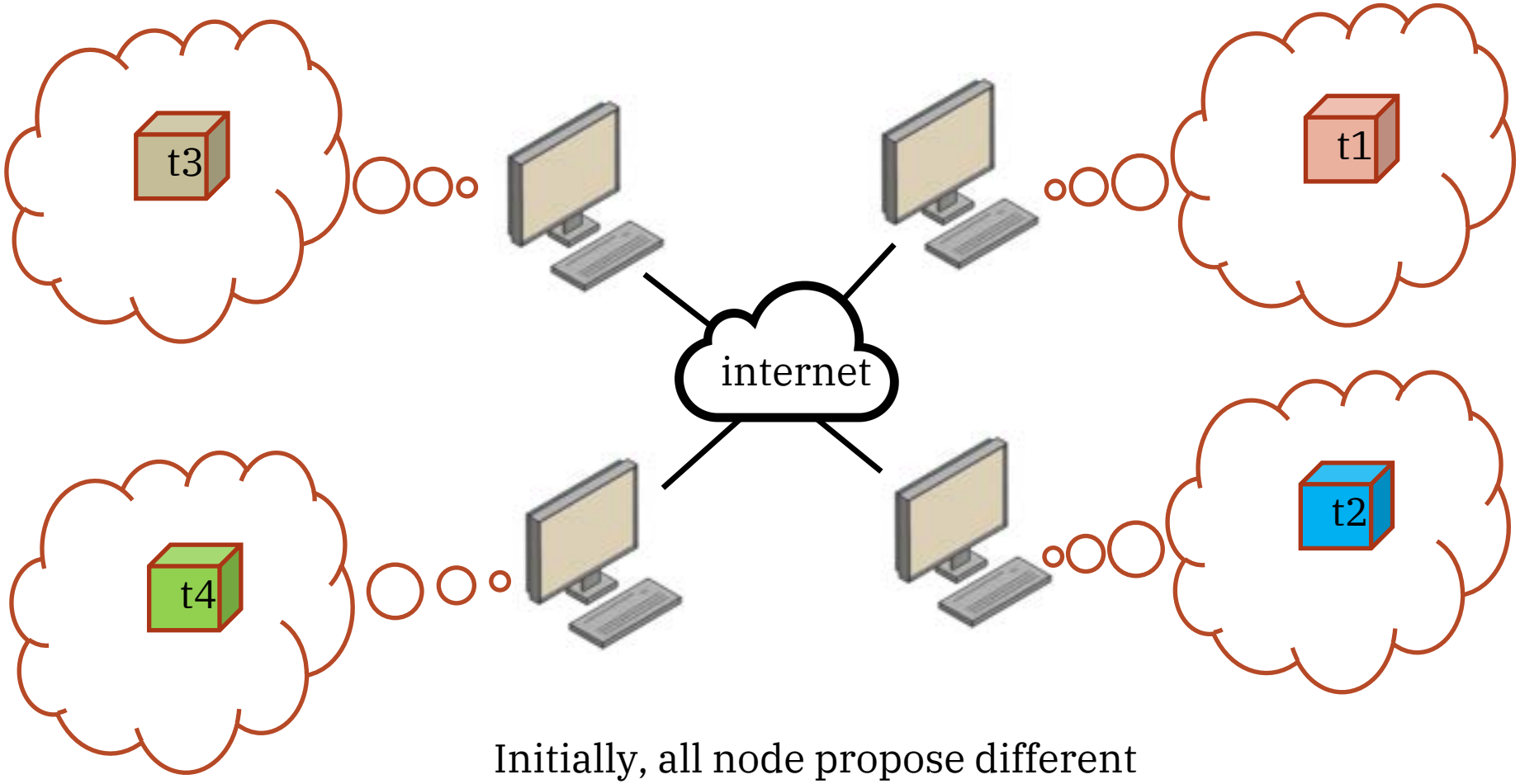
Reminder: Blockchain Consensus

A set of transactions is *non-conflicting* if it does not contain conflicting transactions.

Assuming that each correct node proposes a proposal s, the Set Byzantine Consensus (SBC) problem is for each of them to decide on a set in such a way that the following properties are satisfied:

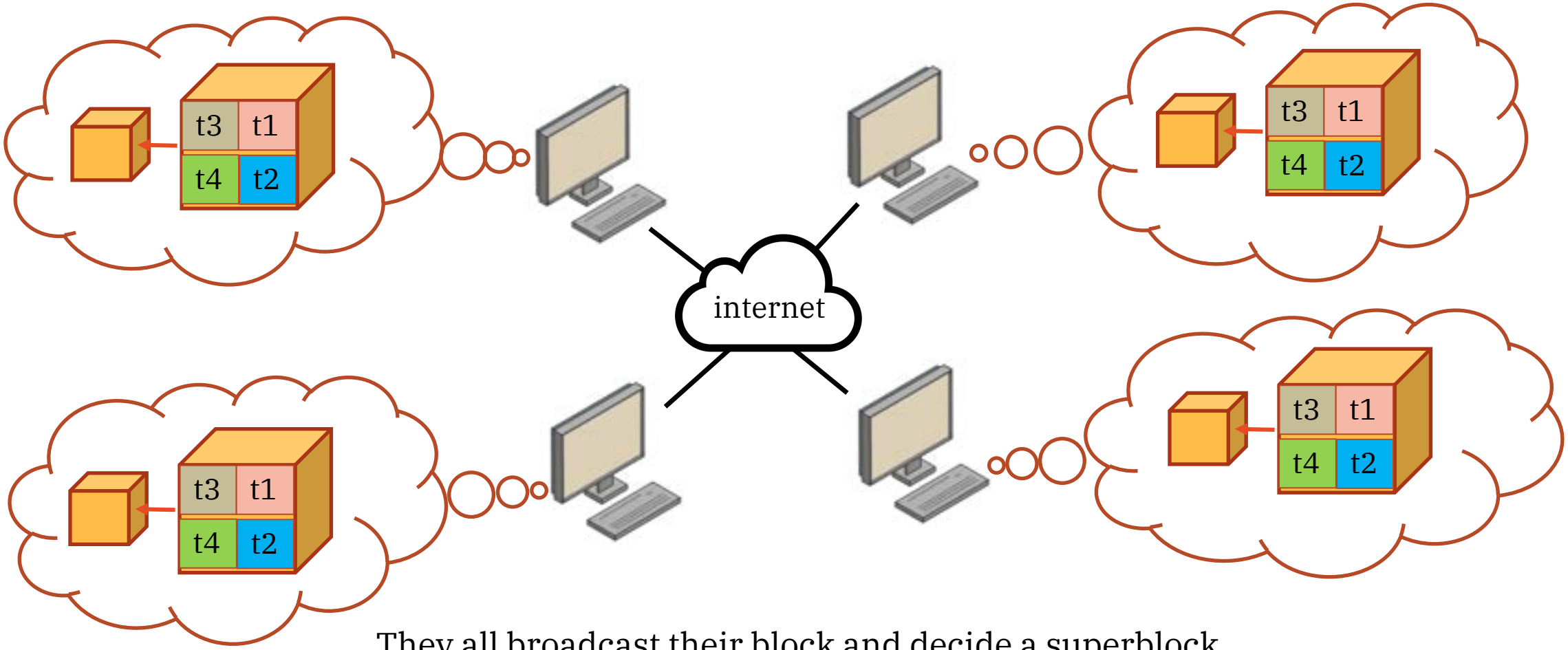
- SBC-Termination: every correct node eventually decides a set of transactions;
- SBC-Agreement: no two correct nodes decide on different sets of transactions;
- SBC-Validity: a decided set of transactions is a non-conflicting set of valid transactions taken from the union of the proposed sets; and if all nodes are correct and propose a common valid non-conflicting set of transactions, then this subset is the decided set.

Superblock



Initially, all node propose different blocks

Superblock



They all broadcast their block and decide a superblock

Additional Material

- Module 4 of the MOOC: https://www.coursera.org/learn/blockchain-scalability
- Chapter 5 of the textbook https://link.springer.com/book/10.1007/978-3-031-12578-2
- DBFT [NCA'18] https://ieeexplore.ieee.org/document/8548057