COMP2121: Principles of Distributed Systems and Networks

Consensus

Unit coordinator Dr. Vincent Gramoli
School of Information Technologies

THE UNIVERSITY OF SYDNEY

Failures

A distributed system is a system where I can't get my work done because a computer has failed that I've never even heard of.

Leslie Lamport

Introduction

- Previously:
  - We have essentially considered message losses as potential failures
- Now:
  - There are several kind of message losses
  - Computing entities might fail in multiple ways
  - We cover here what this implies on the distributed system

Outline

- Distributed Implementation of Blockchain
- Failure taxonomy
- Synchronous Consensus
- Crash-Tolerant Consensus

Blockchain

THE UNIVERSITY OF
SYDNEY

Payment with Blockchain

Bitcoin is both a currency (lower-case b) and a payment system (big B).

It implements a distributed ledger that is

- Immutable: transactions can only be appended to the ledger
- Pseudonymous: participants are identified by their address (pseudonym)

It uses:

- A Gossip-based protocol for maintaining an overlay
- Peer-to-peer network where peers are both clients and servers (miners)
- Public key cryptosystem, participant sign their transactions
- Consensus for the distributed system to decide a block at index I

Market Capitalization

#	Name	Market Cap	Price	Available Supply	Volume (24h)	% Change (24h)	Price Graph (7d)
1	Bitcoin	$9,674,432,316	$609.29	15,878,076 BTC	$65,948,500	-0.10%	
2	Ethereum	$1,149,758,382	$13.66	84,158,631 ETH	$25,720,400	5.74%	

The Blockchain abstraction

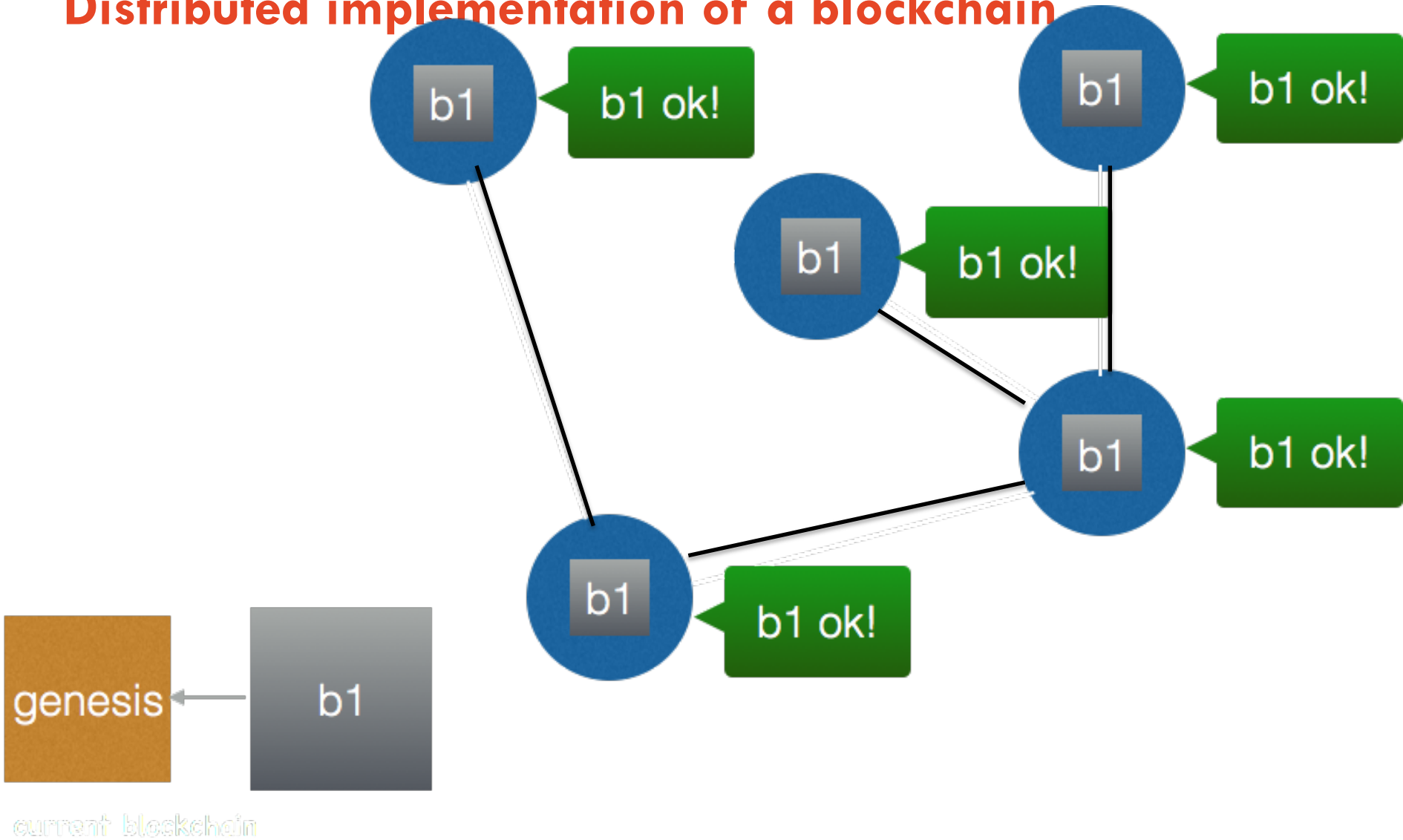
Let $G = \langle B, P \rangle$ a directed acyclic graph (DAG) where blocks B point to each other with pointers P

$\langle b0, b1 \rangle \in P$ is a pointer from current block b1 to previous block b0

The *pointer* is a representation of a hash of the destination block that the source block contains

The *genesis block* is a special block known initially by all participants

Distributed implementation of a blockchain



Failure Taxonomy

THE UNIVERSITY OF SYDNEY

Failure taxonomy

- *Failure*: a system *fails* when it cannot meet its promises
- An *error* is part of a system state that may lead to a failure
- A *fault* is the cause of an error
  - Transient fault: a fault that occurs once and then disappear
  - Intermittent fault: a fault that occurs, then vanishes, then reappears
  - Permanent fault: a fault that continues to exist until faulty component is replaced

Failure taxonomy

- *Dependability* implies the following:
  - *Availability*: the probability that the system is operating correctly at any given moment
    E.g., a system that is unavailable 1ms per hour is available >99.9999%
  - *Reliability*: the ability for a system to run continuously without failures
    E.g., a system may be reliable even though it is shut down during December
  - *Safety*: is a property of a system in which nothing bad happens if it fails
    E.g., a system is safe is its failure does not corrupt it
  - *Maintainability*: The easiness for a system to be repaired
    Note. may lead to high availability if the system recovers automatically after a failure

Failure taxonomy

- Failure model

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

- The most serious failure is arbitrary (a.k.a. Byzantine), it can be of any type.

Consensus

THE UNIVERSITY OF SYDNEY

Consensus

Problem definition

Consider a system in which each entity p_i

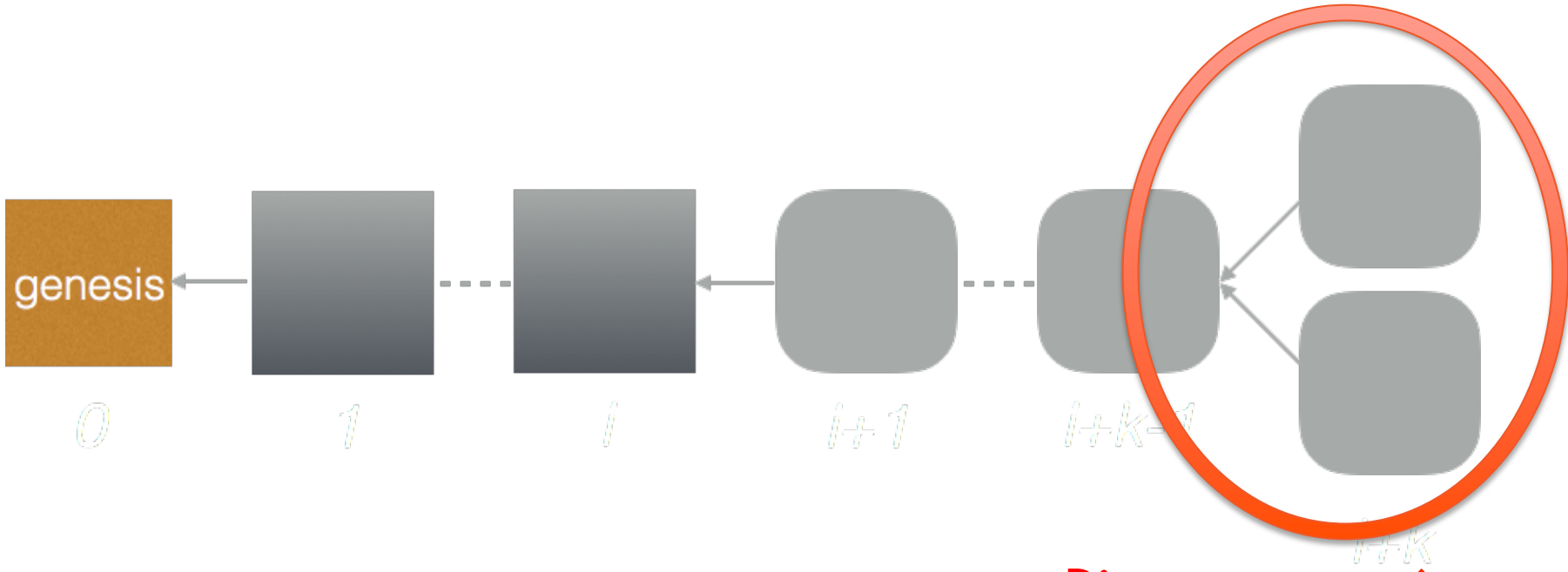
- receives an input x initially a value from a well-ordered set of inputs
- outputs y, initially undefined ($\perp$) but that can be assigned only once

Solving the consensus problem aims at guaranteeing that:

- *Termination*: in every admissible execution, y_i is eventually assigned a value for every non-faulty processor p_i. A correct process decides.
- *Agreement*: in every execution, if y_i and y_j are assigned, then $y_i = y_j$, for all non-faulty processes p_i and p_j. All correct processes decide the same.
- *Validity*: in every execution, if, for some value v, $x_i = v$ for all processes p_i, and if y_i is assigned for some non-faulty process p_i, then $y_i = v$. The value decided has been proposed.

Why is consensus needed in blockchain?

Consensus is necessary to totally order the blocks, maintaining the *chain*

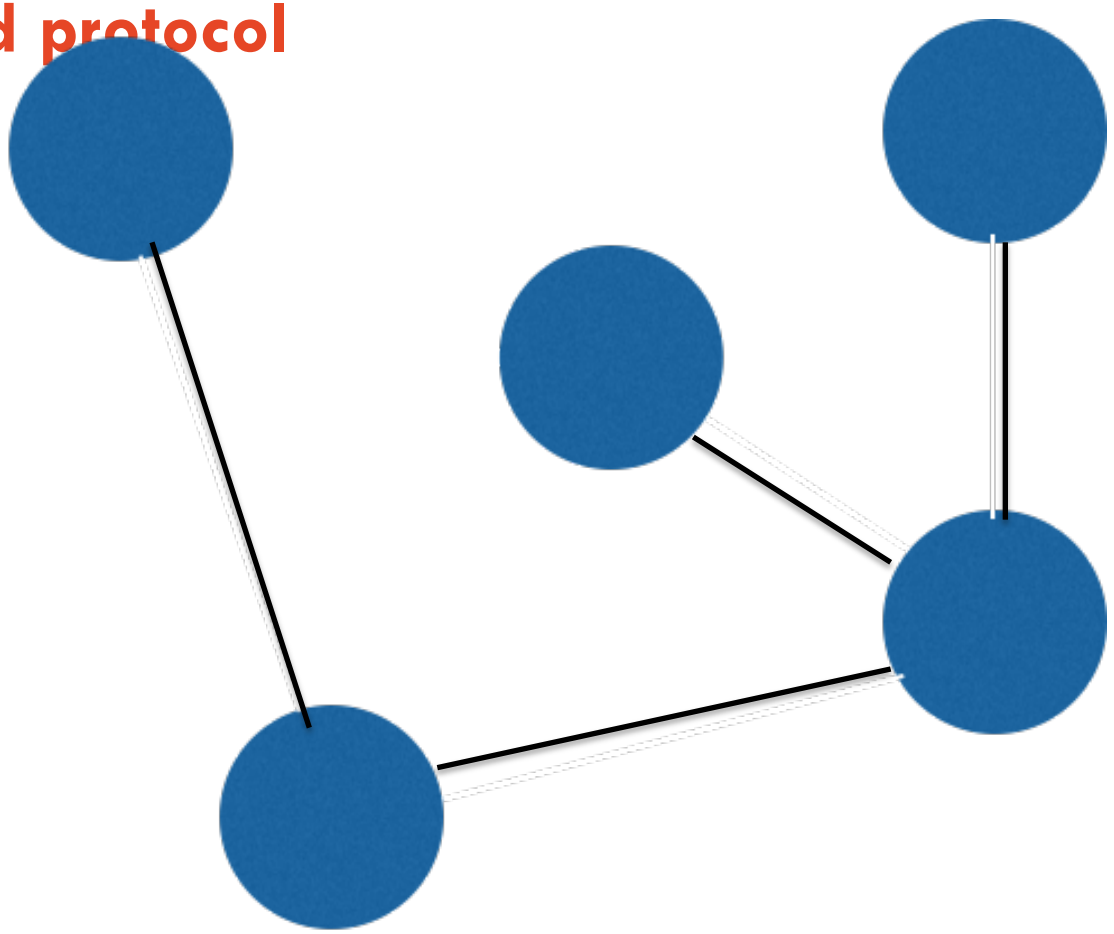


Disagreement ⇒ no chain but a DAG

Example of a Blockchain Execution

THE UNIVERSITY OF SYDNEY

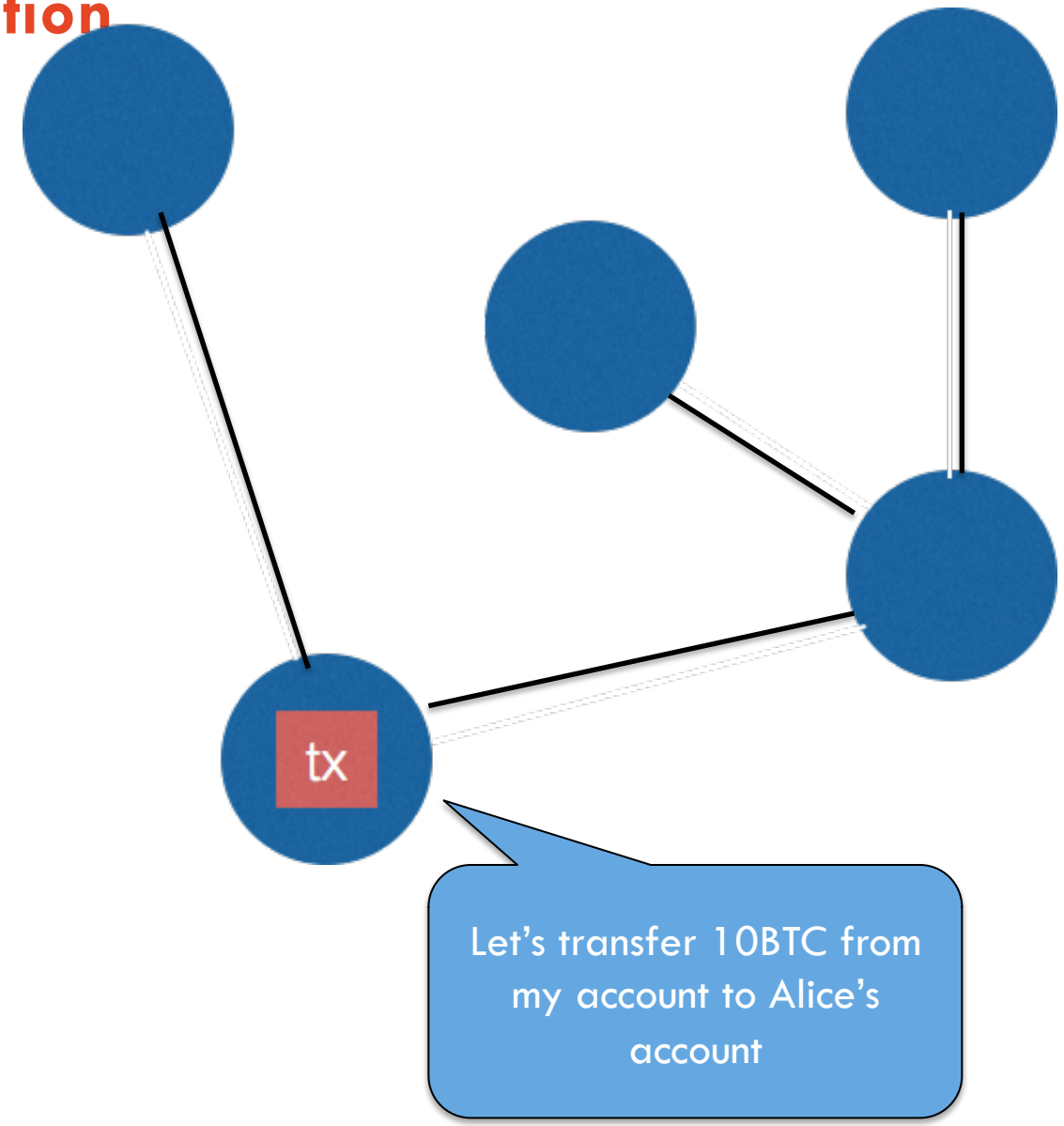
Gossip-based protocol



genesis

Current blockchain state

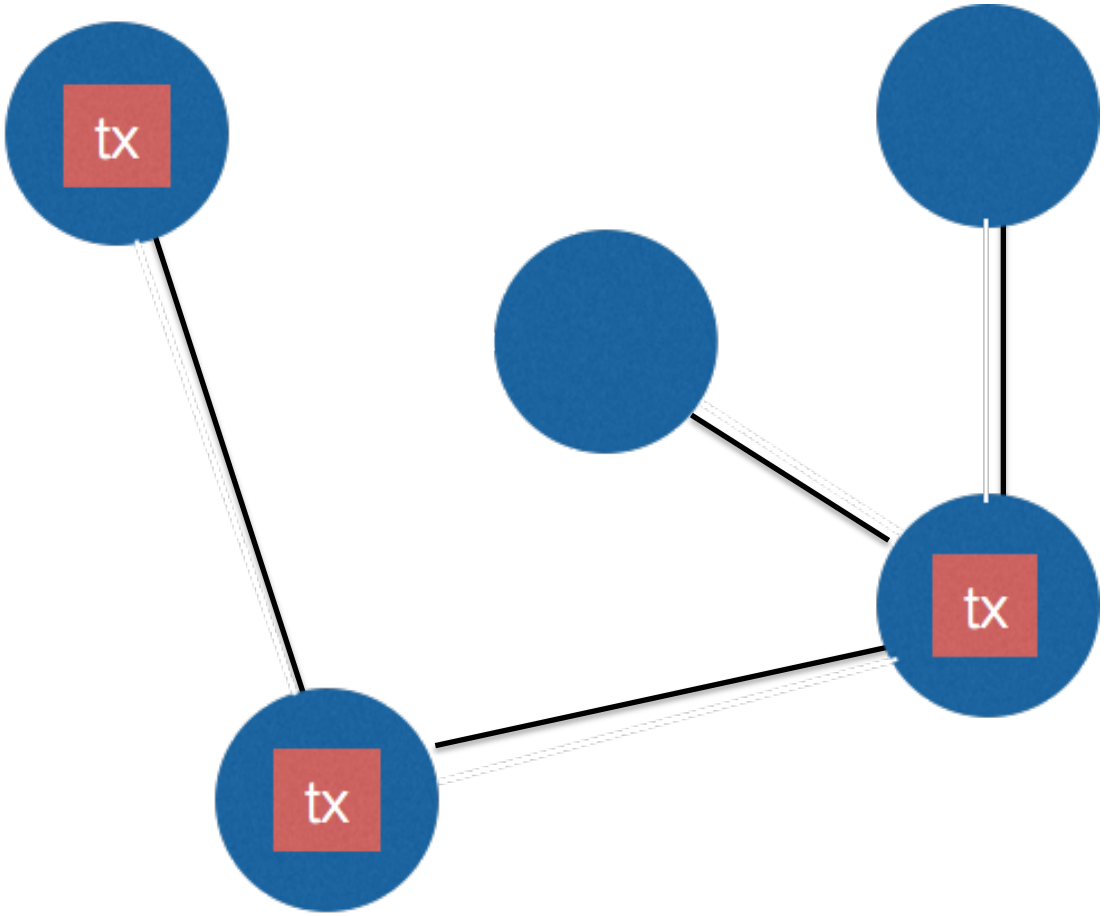
New transaction



genesis

current blockchain

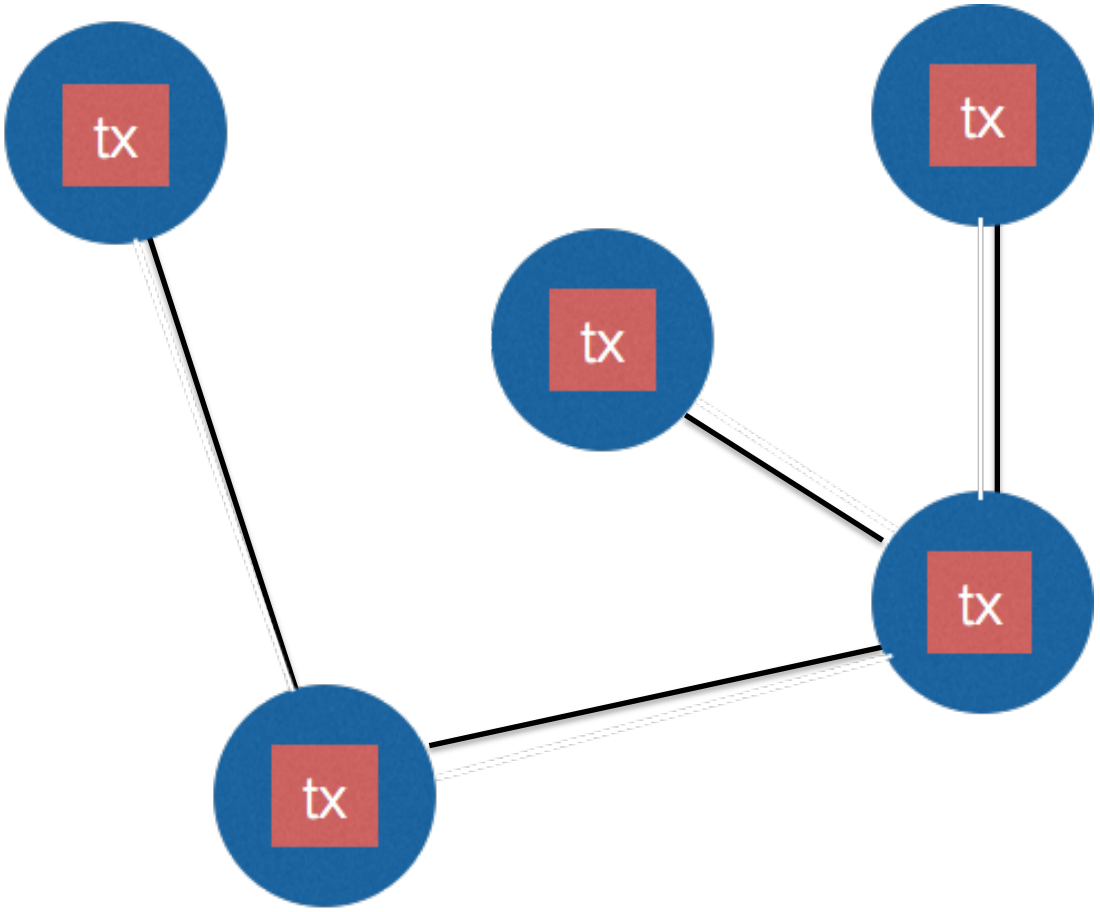
Broadcast



genesis

current blockchain

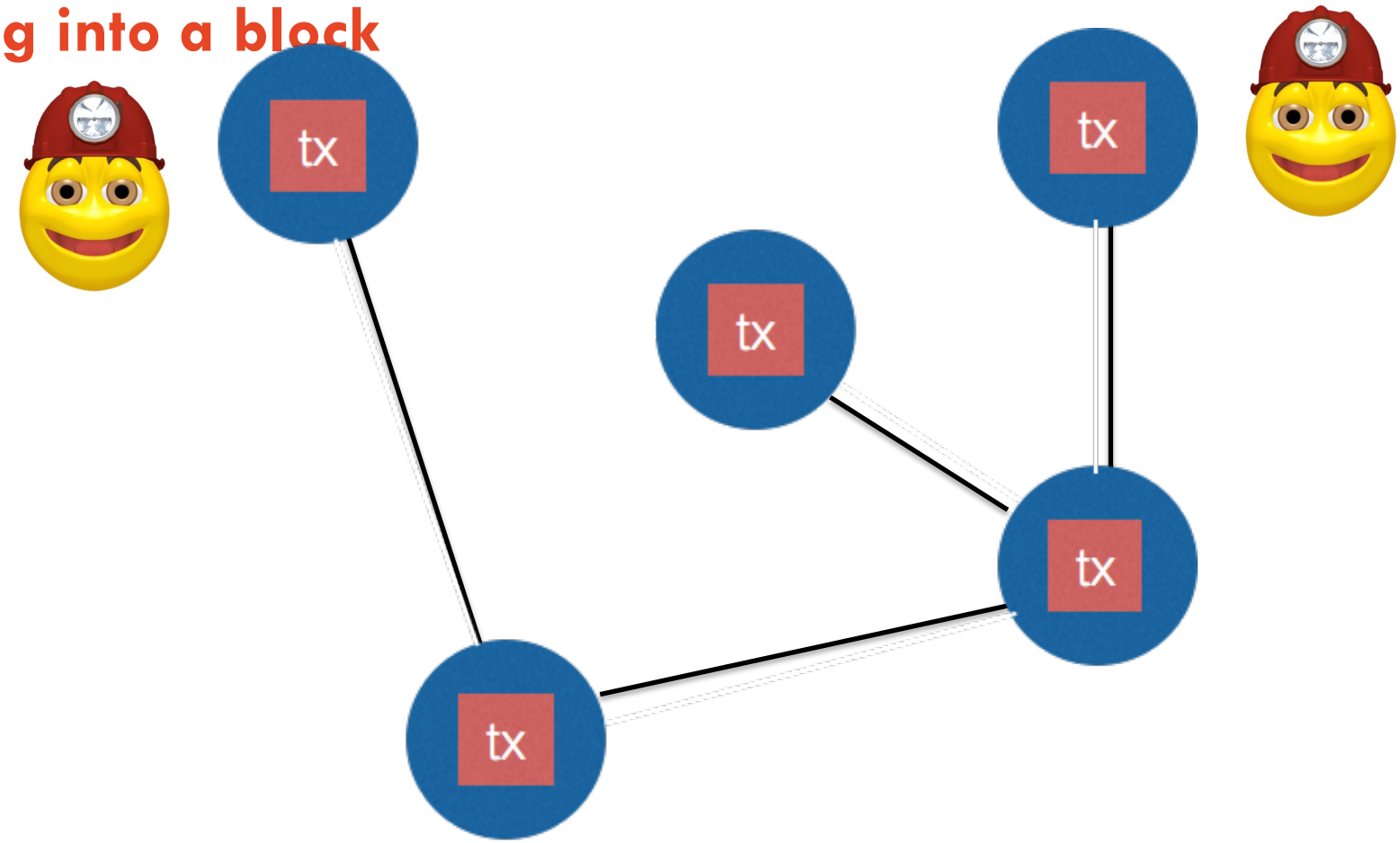
Broadcast



genesis

current blockchain

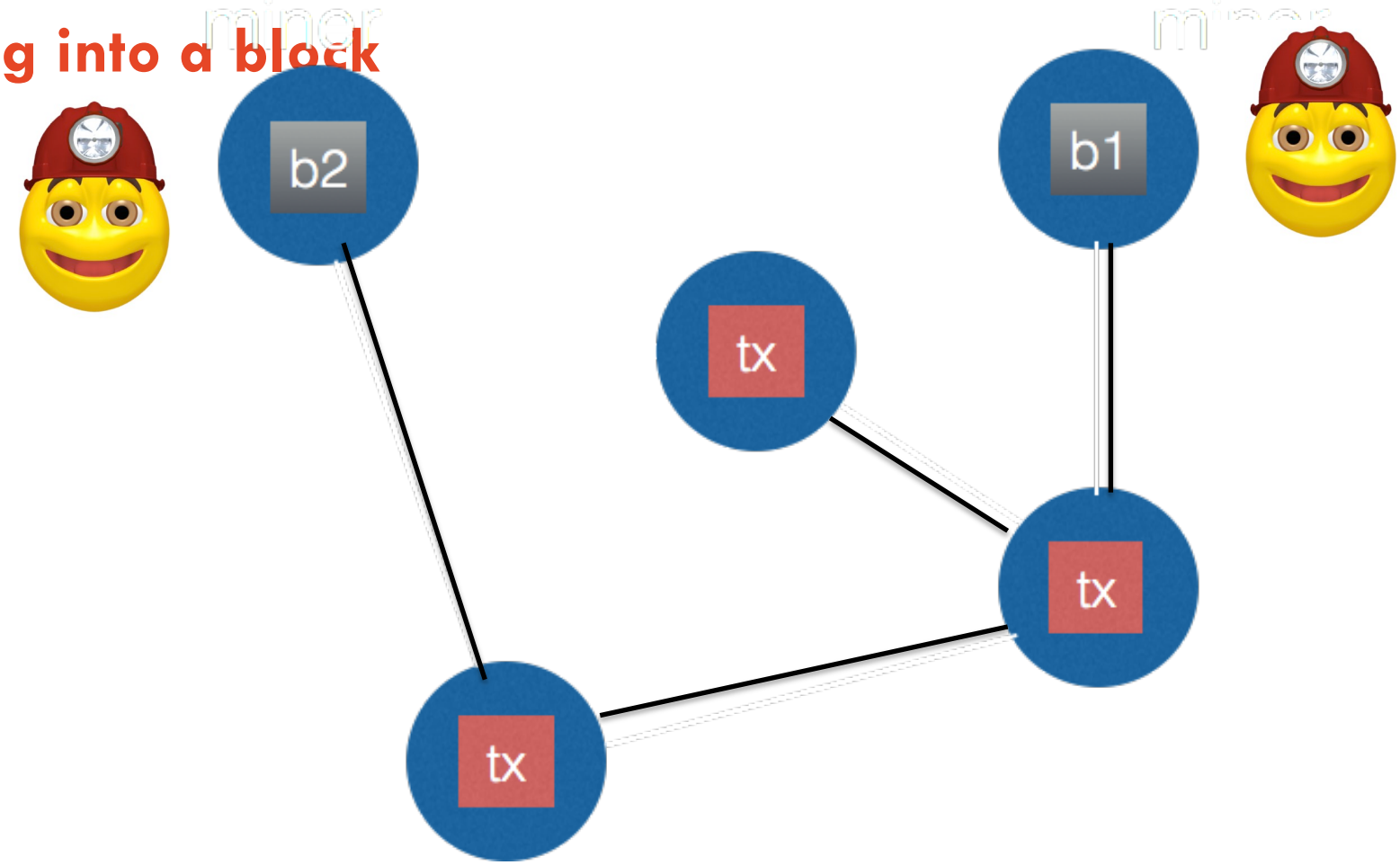
Mining into a block



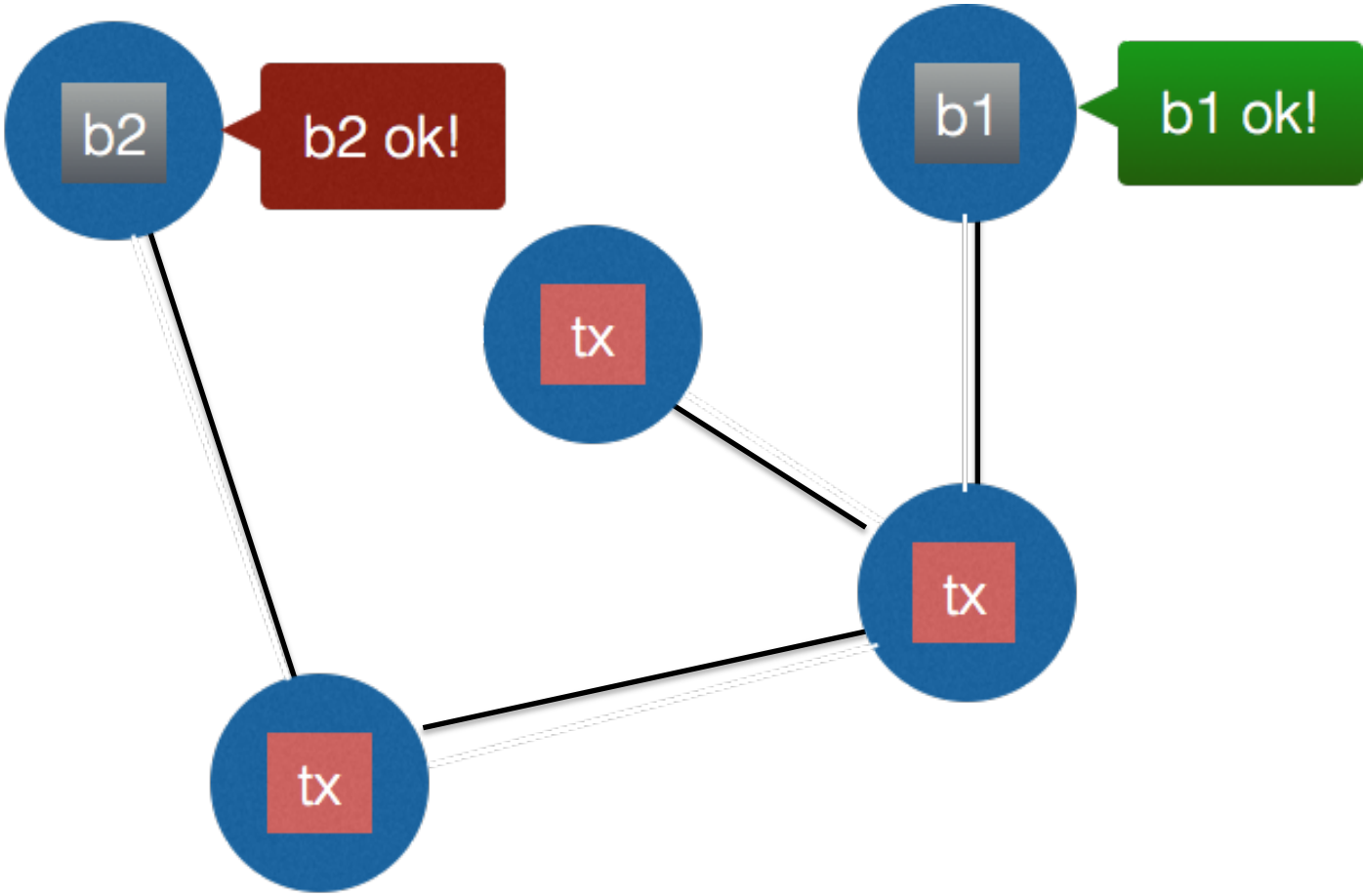
genesis

current blockchain

Mining into a block



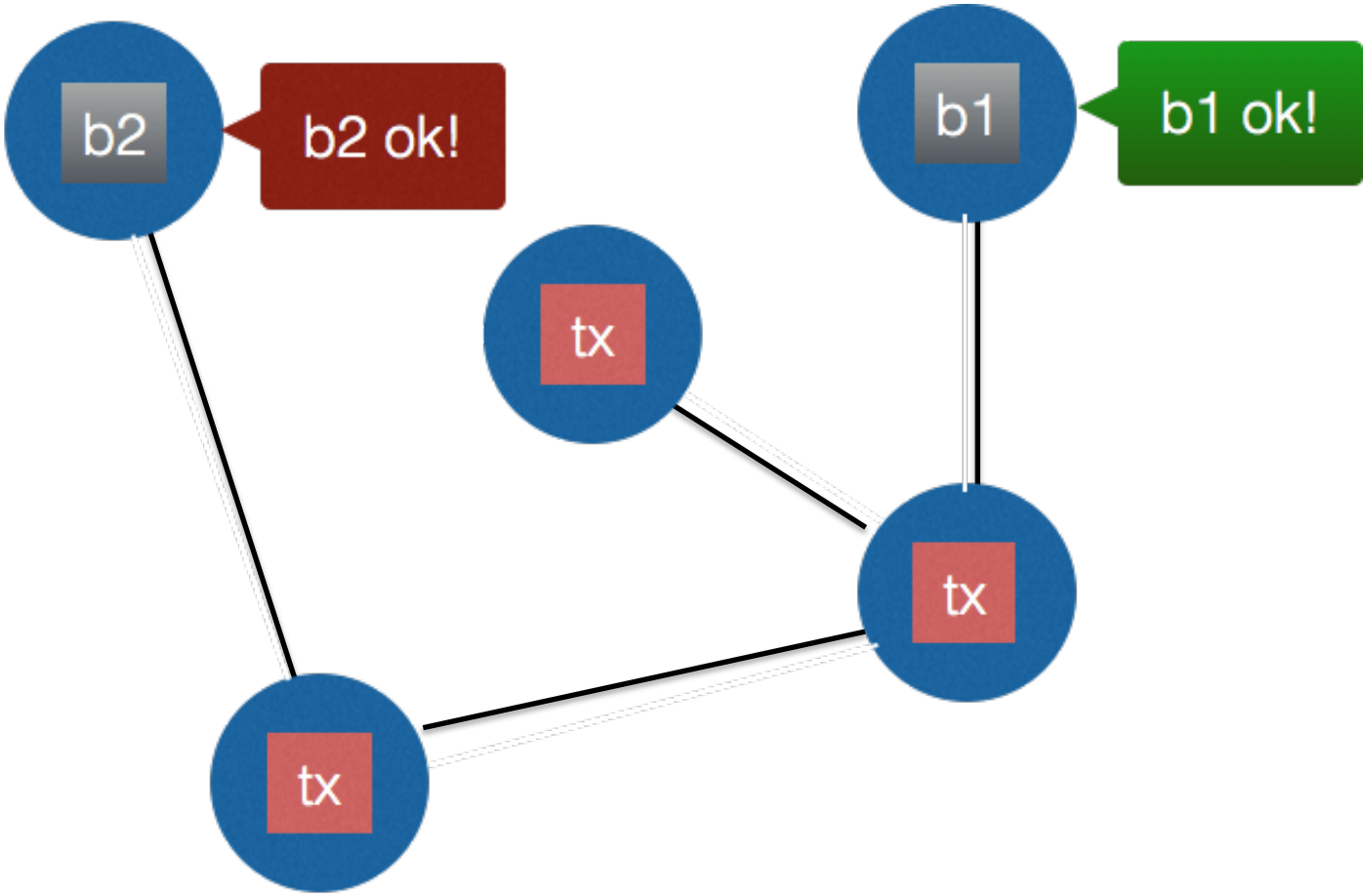
Consensus



genesis

current blockchain

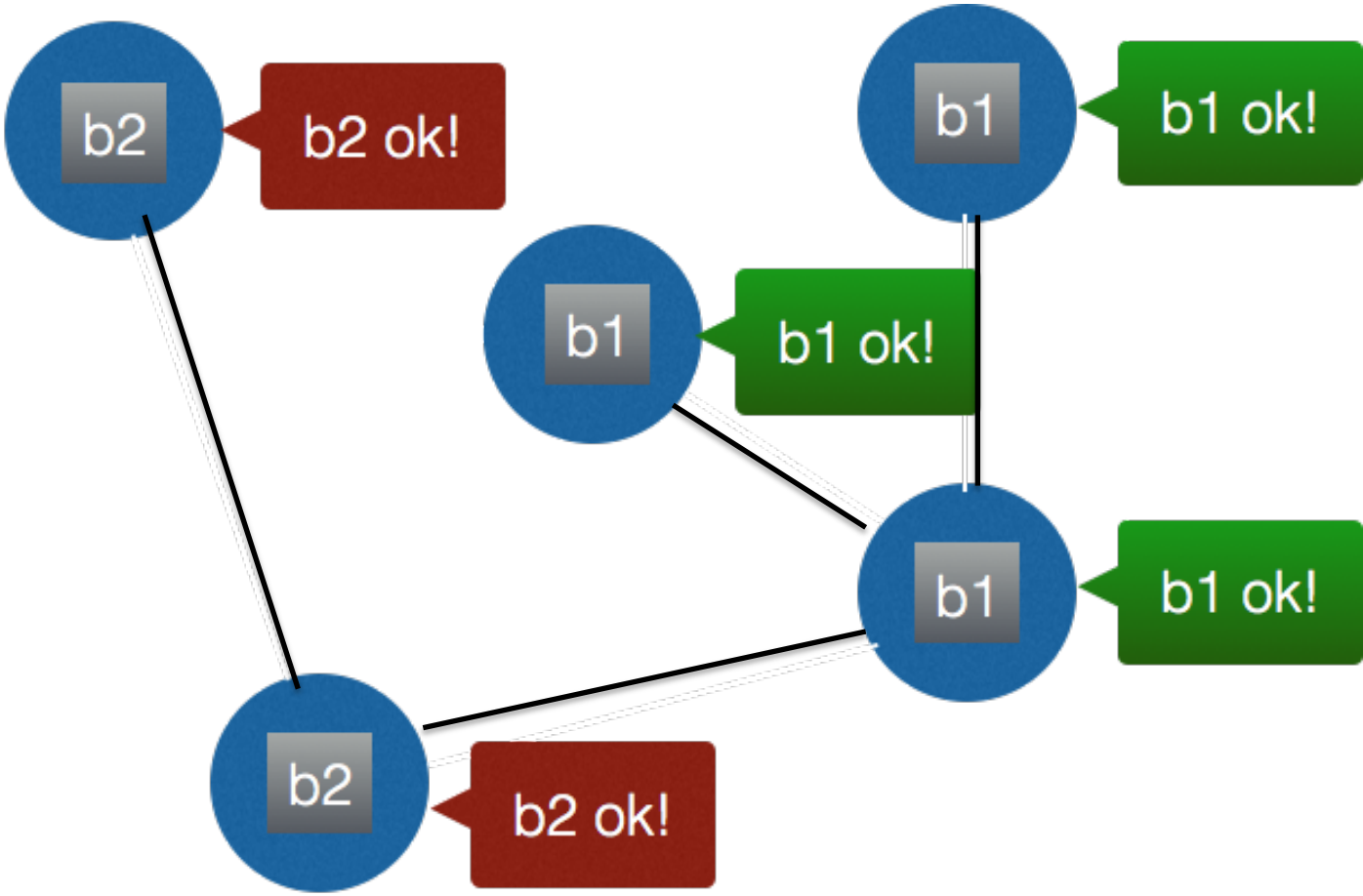
Consensus



genesis

current blockchain

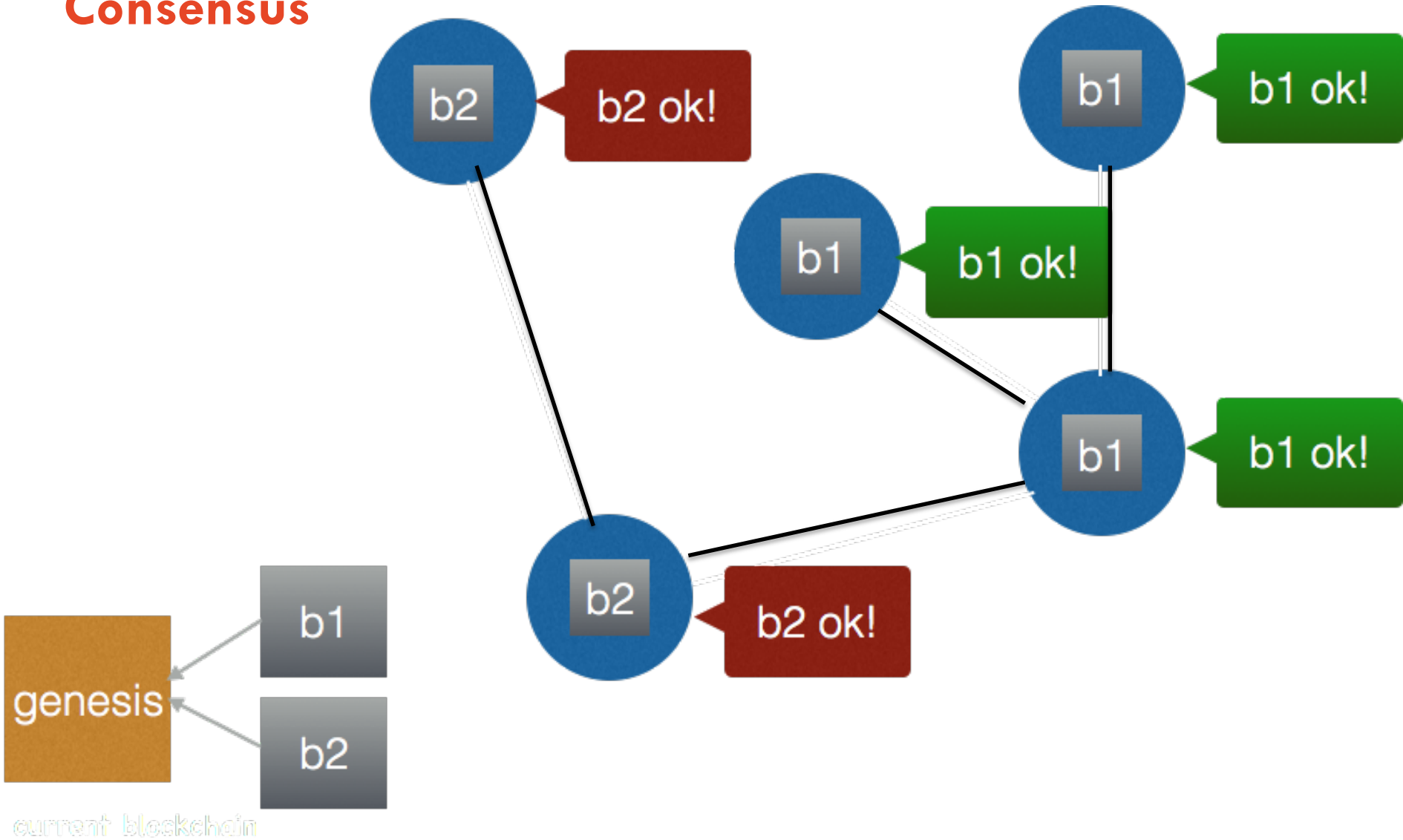
Consensus



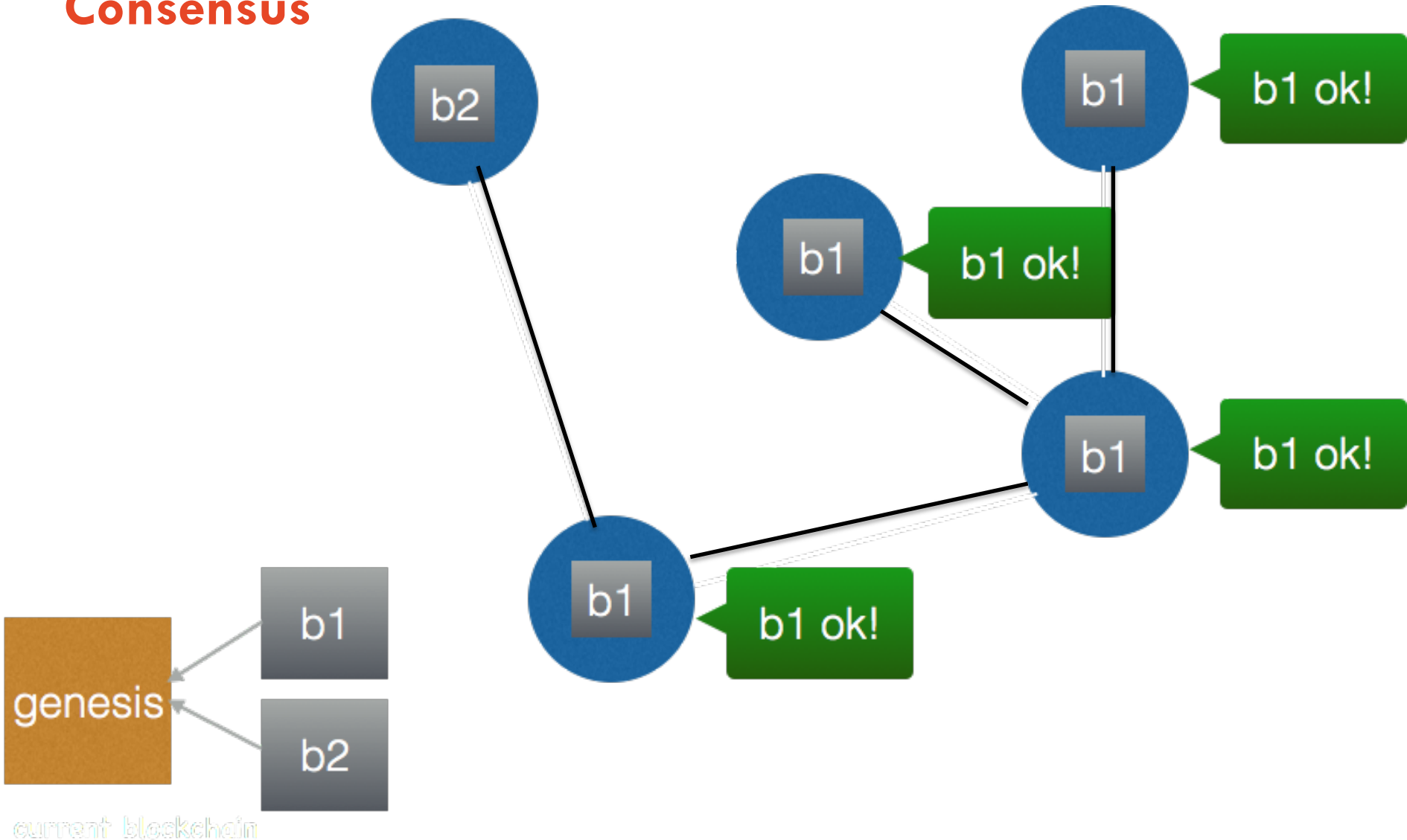
genesis

current blockchain

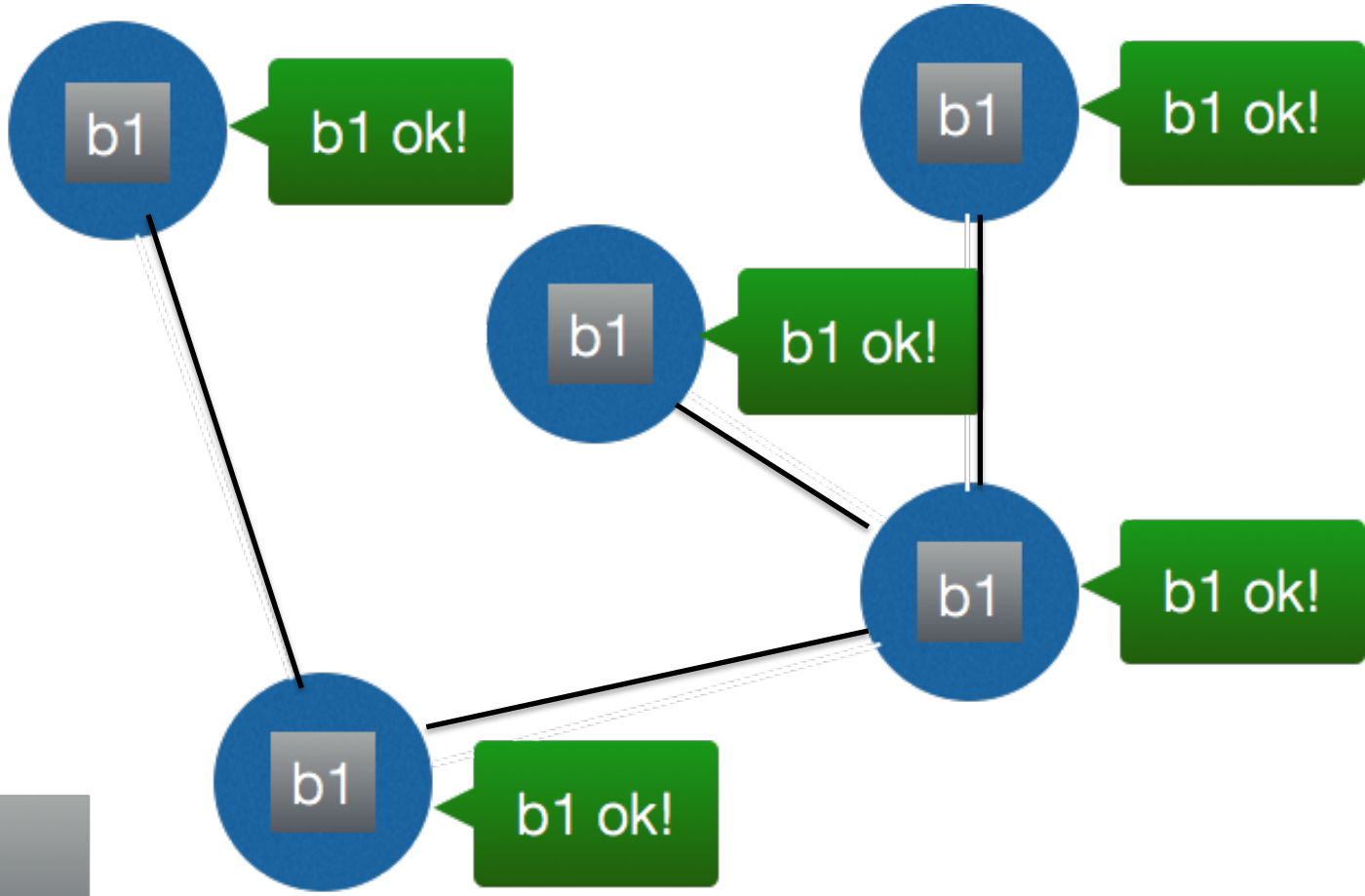
Consensus

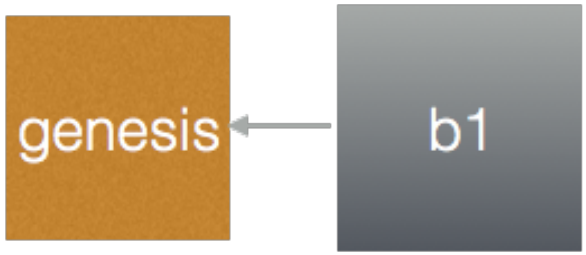

b2
b2 ok!
b1
b1 ok!
b1
b1 ok!
b1
b1 ok!
b2
b2 ok!
genesis
b1
b2
current blockchain

Consensus


b2
b1
b1 ok!
b1
b1 ok!
b1
b1 ok!
b1
b1 ok!
genesis
b1
b2
current blockchain

Consensus





current blockchain

Consensus

Characteristics of the distributed system may impact the feasibility (or at least the easiness) of solving the consensus problem:

- A system is *synchronous* (as opposed to asynchronous) if there exists some constant $c \geq 1$, such that if any processor has taken $c+1$ steps, every other process has taken at least 1 step.
- *Synchronous communication*: Every message is delivered with a globally predetermined maximum time. (This time is known by the algorithm.)
- Message delivery is *ordered* (in real time) means that messages from different senders are delivered in the order they were sent in real global time.
- Message transmission may be done to **unicasting** (to a single destination) or **multicasting** (to multiple destinations)

Crash-Tolerant Synchronous Consensus

THE UNIVERSITY OF SYDNEY

Consensus

Crash-tolerant synchronous solution

- Assumptions
  - Synchronous communication using unicast (known bound on the delay of msgs)
  - Crash failures
  - A maximum of f processes can fail (by crashing only)

State of process p_i:
V, the set of known values, initially {x} the proposed value of p_i
f, the maximum number of processors that can fail

Basic consensus algorithm for p_i:

```
for (k=1; k ≤ f+1; k++) {                  // in each f+1 round
  send {v in V that p_i has not sent yet} to all p_j     // send values just found
  receive S_j from p_j j≠i                 // receive values
  V = V ∪ S_j for all j                    // records the values
}
y := min(V)                                // at the end, output min
```

Consensus

Crash-tolerant synchronous solution (con't)

- Intuition of the proof: at least one iteration (i.e., *round*) without failure
  - Termination: the loop has a bounded number of iterations, so y is assigned
  - Agreement: at the end we have $V_i = V_j$ for any i and j so $y_i = y_j$
  - Validity: there exists a p_j such that its input x_j is the decided value

State of process p_i:
V, the set of known values, initially {x} the proposed value of p_i
f, the maximum number of processors that can fail

Basic consensus algorithm for p_i:

```
for (k=1; k ≤ f+1; k++) {                  // in each f+1 round
  send {v in V that p_i has not sent yet} to all p_i     // send values just found
  receive S_j from p_j j≠i                 // receive values
  V = V ∪ S_j for all j                    // records the values
}
y := min(V)                                // at the end, output min
```

Byzantine-Tolerant Synchronous Consensus

THE UNIVERSITY OF SYDNEY

Consensus

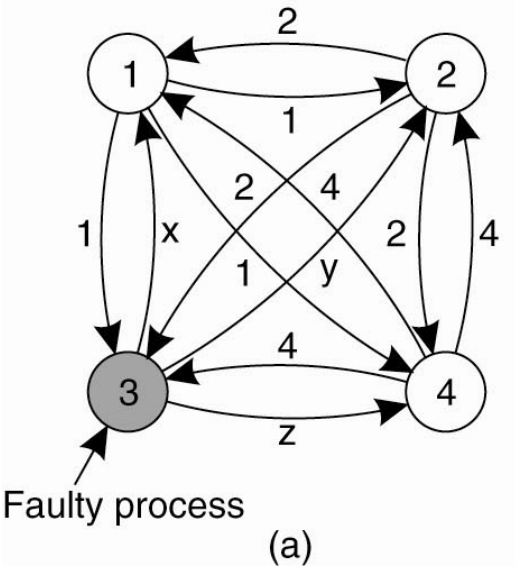
Byzantine-tolerant synchronous solution

- Assumptions with a different type of failures
  - Synchronous communication using unicast
  - A maximum of $f < n/3$ processes can fail among n
  - Byzantine failures (f processes may misbehave, e.g., send fake messages)

- General result:

In a system of n processes among which f are Byzantine, there is no solution to the consensus problem if $n \leq 3$.

Consensus



Byzantine-tolerant synchronous solution (con't)

- Assumptions with a different type of failures
  - Synchronous communication using unicast
  - A maximum of $f < n/3$ processes can fail among n
  - Byzantine failures (f processes may misbehave, e.g., send fake messages)

Example: n=4, f=1, every correct pi proposes i, except one Byzantine proposes i or letters…

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

(b)

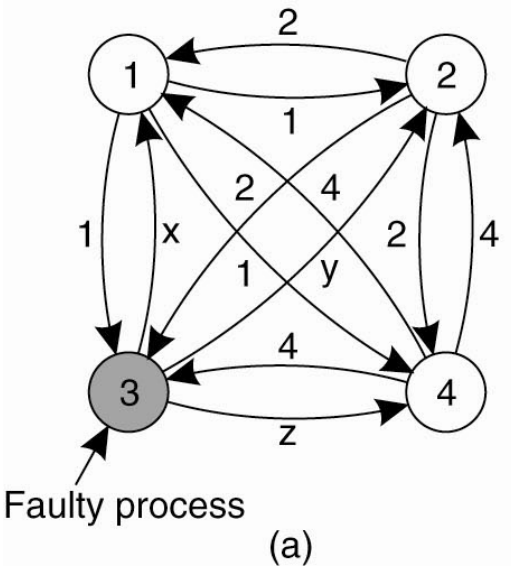
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Consensus

Byzantine-tolerant synchronous solution (con't)

- Example of solution for n=4 and f=1
  - (a) every process sends its input to all others
  - (b) the collected vectors (i^{th} coordinate received from p_i)
  - (c) every process sends the collected vector and obtains these 3 new vectors
  - A majority vector is computed using for each coordinate the one appearing a majority of times (or $\perp$ if there is none)
  - $(1, 2, \perp, 4)$



(a)

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Consensus

Fault-tolerant asynchronous solution

- What if we change the assumptions
  - Asynchronous communication using unicast
  - failures
  - A maximum of f processes can fail

- General result:

  There is no solution to the consensus problem in an asynchronous system even with a single "crash" failure

Distributed Commit

THE UNIVERSITY OF SYDNEY

Distributed Commit

Distributed commit problem

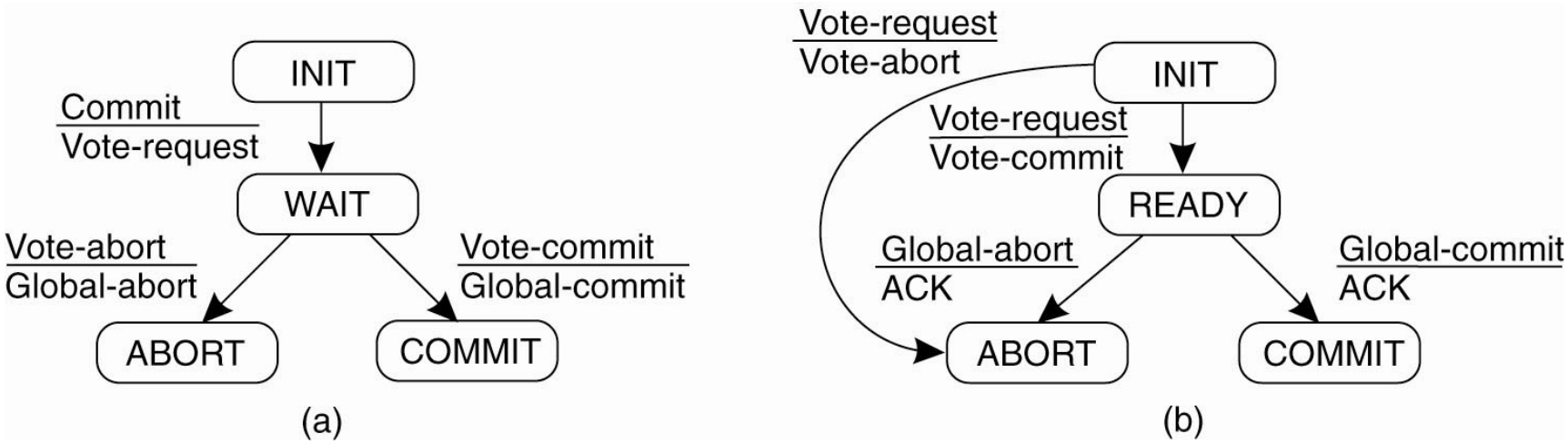
- The *distributed commit* problem involves having an operation being performed by each member of a process group, or none at all.
- If this operation is the delivery of a message, then this problem is called *reliable multicast*.
- This operation could be the *commit* of a transaction, hence the name.
- Usually solved using a special process called a *coordinator*.

Distributed Commit

Two phase commit (2PC)

Assumptions: no failure occur

- (a) The finite state machine for the coordinator in 2PC
- (b) The finite state machine for a participant



Distributed Commit

2PC (con't)

- Actions taken by a participant P when residing in state READY and having contacted another participant Q

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Distributed Commit

2PC (con't)

1. Voting phase:
   1. Coordinator sends an invite to commit request to all
   2. The participants answer the coordinator by a COMMIT or an ABORT message

2. Commit phase:
   1. The coordinator commits if it received only COMMIT message from all, otherwise it aborts (if timeout expires or if some ABORT message was received) and sends the appropriate action to the participants
   2. Upon reception, the participants abort if the coordinator aborted or commit if the coordinator committed

Distributed Commit

2PC (con't)

1. Voting phase:
   1. Coordinator sends an invite to commit request to all
   2. The participants answer the coordinator by a COMMIT or an ABORT message

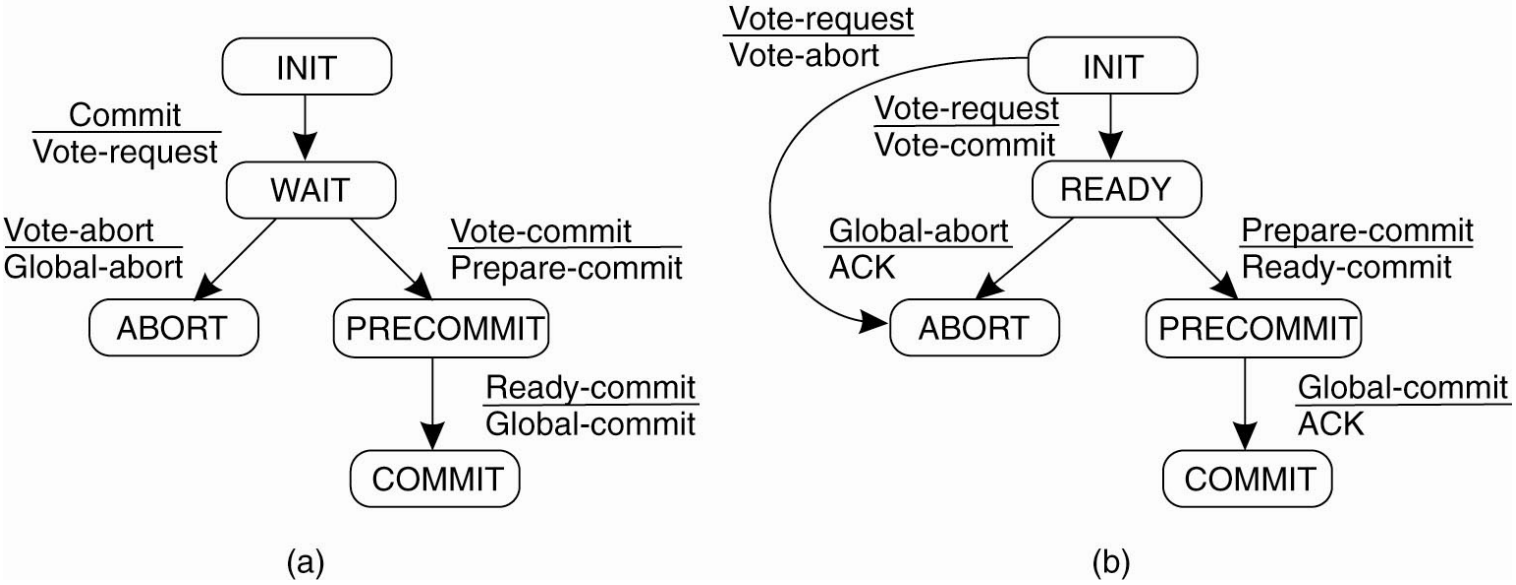
2. Commit phase:
   1. The coordinator commits if it received only COMMIT message from all, otherwise it aborts (if timeout expires or if some ABORT message was received) and sends the appropriate action to the participants
   2. Upon reception, the participants abort if the coordinator aborted or commit if the coordinator committed

- Limitation: only works in the absence of failures, otherwise if the coordinator fails after inviting to commit, then the other processes cannot cooperatively decide whether to COMMIT or ABORT (as they do not know what the coordinator decided)

Distributed Commit

Three phase commit (3PC)

- Problem with 2PC: the participants may have to wait for the coordinator to recover to reach a decision
- 3PC:
  - (a) The finite state machine for the coordinator in 3PC
  - (b) The finite state machine for a participant



Distributed Commit

3PC (con't)

- Assumptions:

1. Voting phase:
   1. Coordinator sends an invite to commit request to all
   2. The participants answer the coordinator by a PREPARE-COMMIT or an ABORT message

2. Pre-commit phase:
   1. The coordinator PRE-COMMITs if it received PREPARE-COMMIT message from all, otherwise it aborts (if timeout expired or if some ABORT message was received) and sends the appropriate action to the participants
   2. The participants abort if the coordinator aborted or a timeout expired or acknowledge the PRE-COMMIT if the coordinator PRE-COMMITted

3. Commit phase:
   1. If acknowledged, the coordinator sends a GLOBAL-COMMIT message
   2. Upon reception, the participants commit

Distributed Commit

3PC (con't)

- The states of the coordinator and each participant satisfy the following two conditions:
  - There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state
  - There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made

- If any operational process is in its READY state, no crashed process will recover to a state other than INIT, ABORT, PRECOMMIT
- ⇒ participants can always come to a final decision

Paxos

THE UNIVERSITY OF
SYDNEY

Paxos

Part-time Parliament (original version)

- Greek legislators working part-time
- Legislator would have hourglass timers, ledger w/ pen and ink and would communicate with messengers
- If a majority of the legislators were in the Chamber and no one entered or left the Chamber for a sufficient long period of time then any decree proposed by a legislator in the Chamber would be passed, and every decree that had been passed would appear in the ledger of every legislator in the Chamber

- Ballot b
- Vote ok/nok
- Decree v
- Decision d

Paxos

Paxos

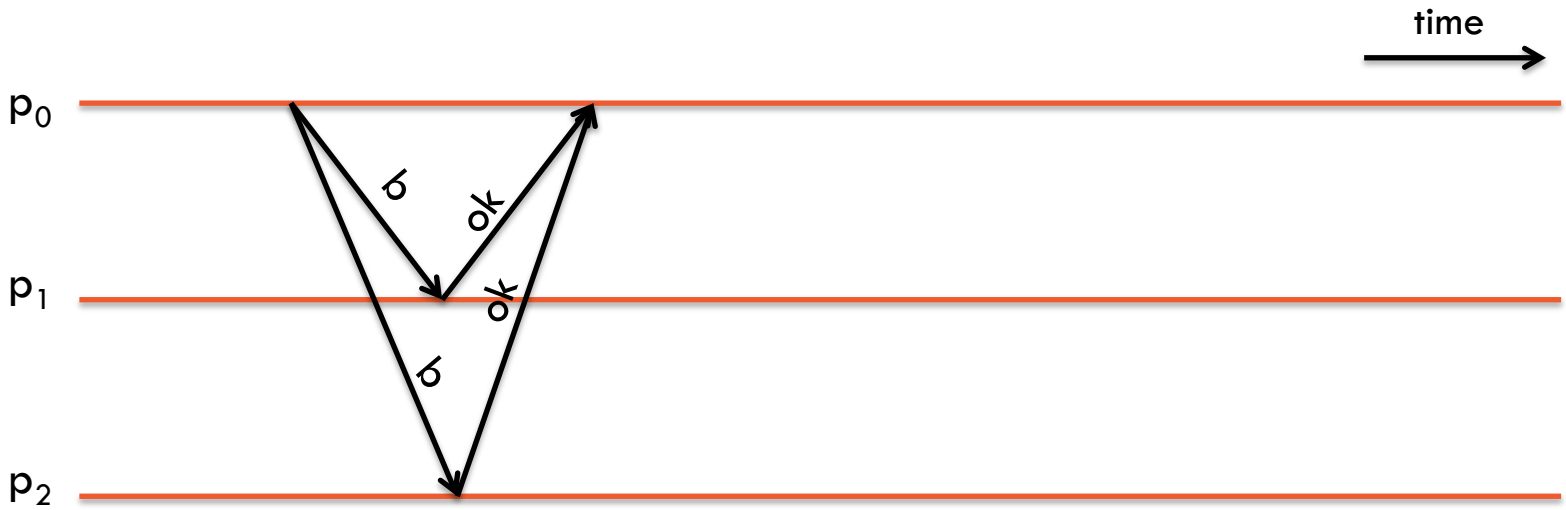
- Historical note
  - In the 1980s, a fault-tolerant distributed file system called "Echo" was built
  - According to the developers, it achieves "consensus" despite any number of failures as long as a majority of nodes is alive
  - The steps of the algorithm are simple if there are no failures and quite complicated if there are failures
  - Leslie Lamport thought that it is impossible to provide guarantees in this model and tried to prove it
  - Instead of finding a proof, he found a much simpler algorithm that works: the Paxos algorithm

- Paxos is a consensus algorithm that does not rely on a particular coordinator
  - Can tolerate some degree of asynchrony: there is an unknown bound on the delay of messages
  - All nodes may crash at any time and may recover

Paxos

Simplified Paxos

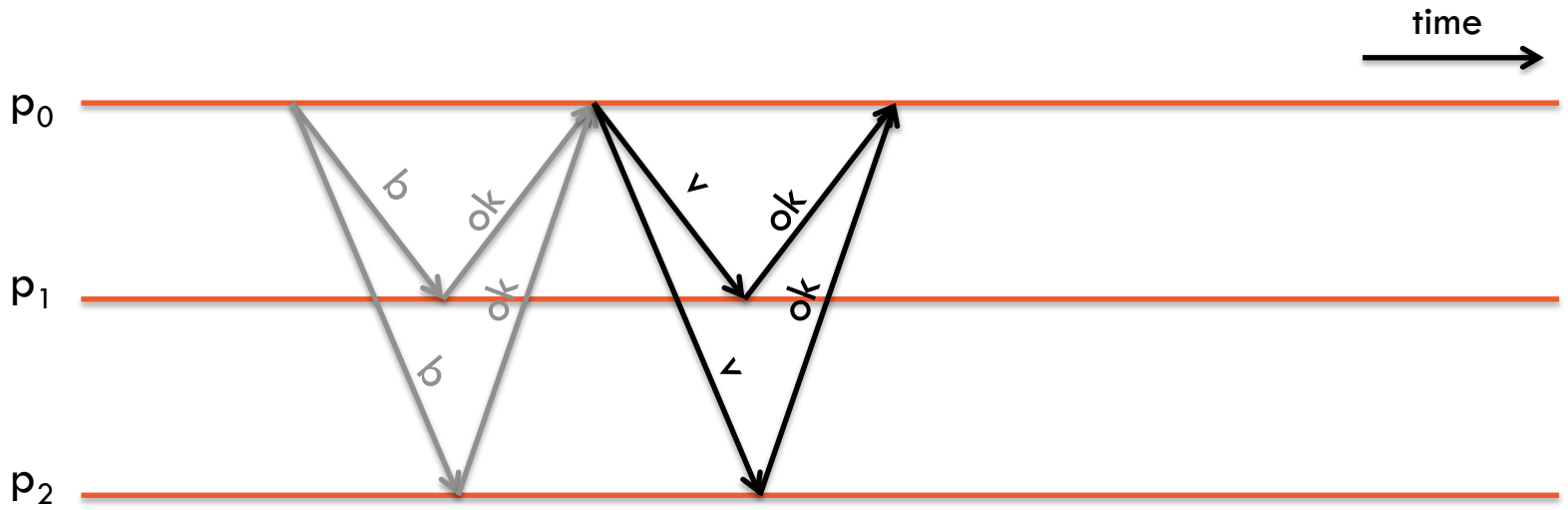
- 1st phase: prepare ballot b



Paxos

Simplified Paxos (con't)

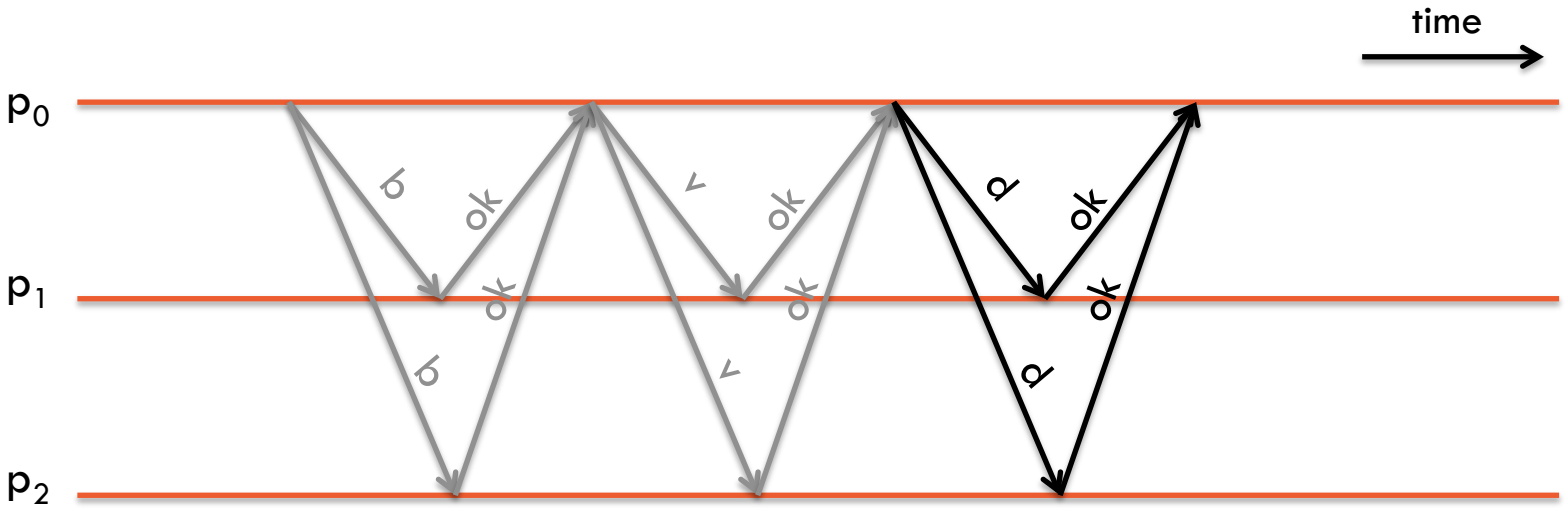
- 2nd phase: propose value v



Paxos

Simplified Paxos (con't)

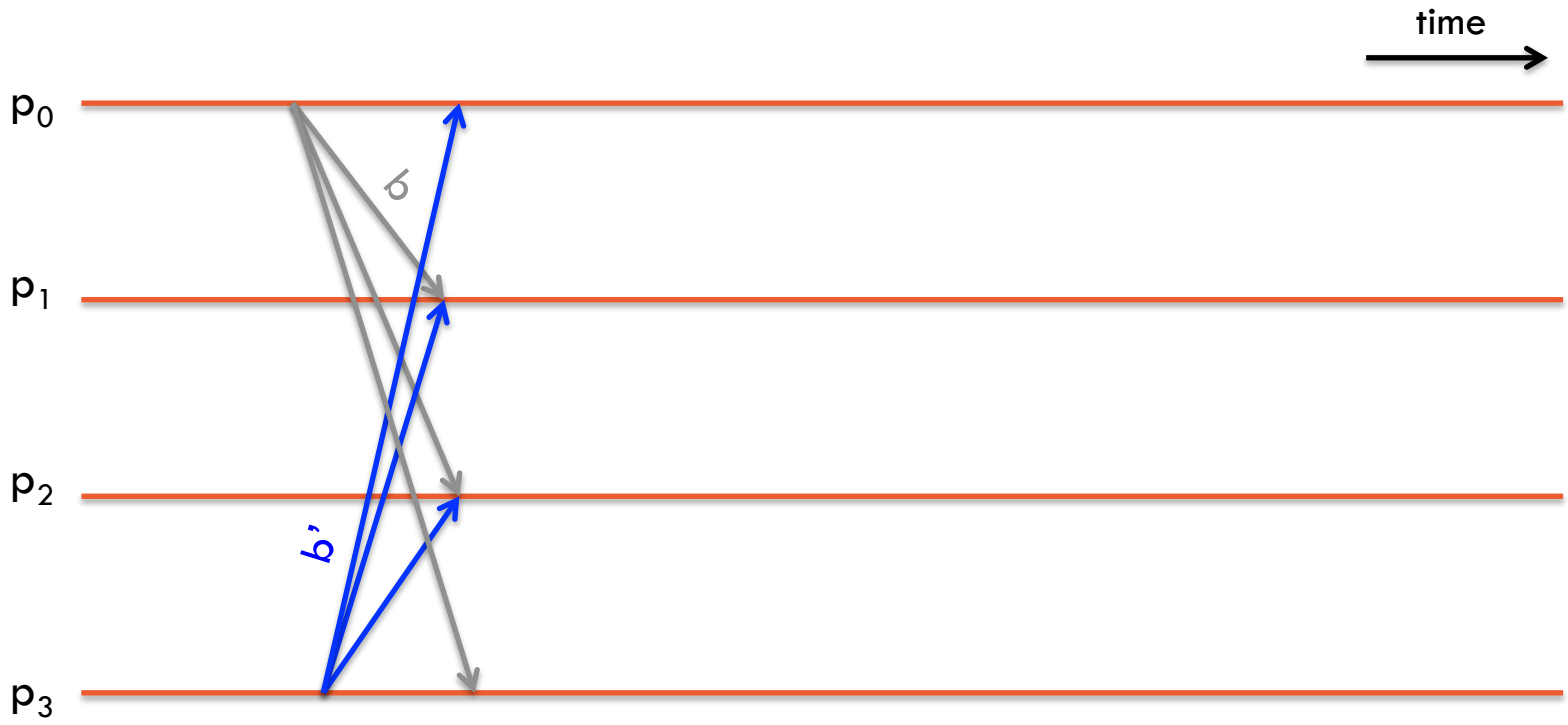
- 3rd phase: propagate decision d



Paxos

Prepare phase in detail

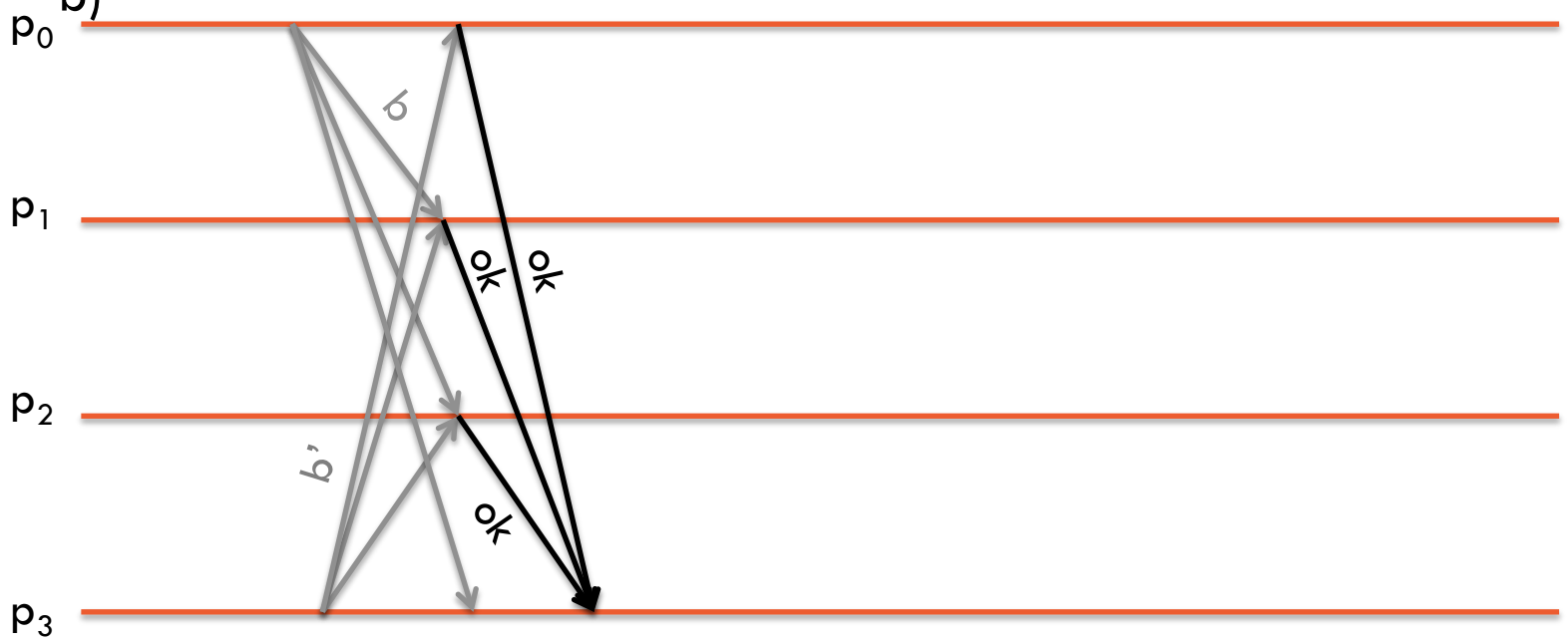
- What if two leaders prepare two ballots concurrently?



Paxos

Prepare phase in detail (con't)

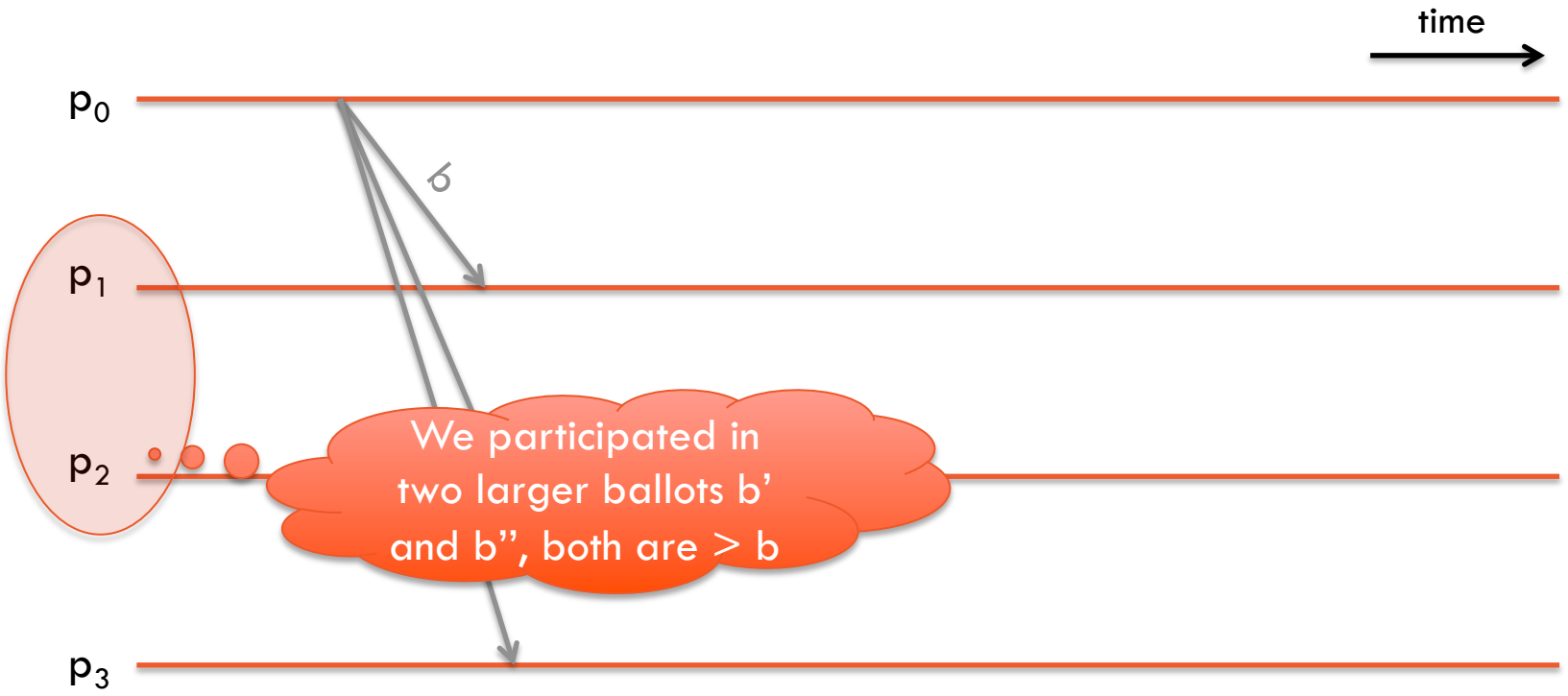
- What if two leaders prepare two ballots concurrently?
  - only participate in ballot with greatest number and ignore lower num (b' > b)



Paxos

Prepare phase in detail (con't)

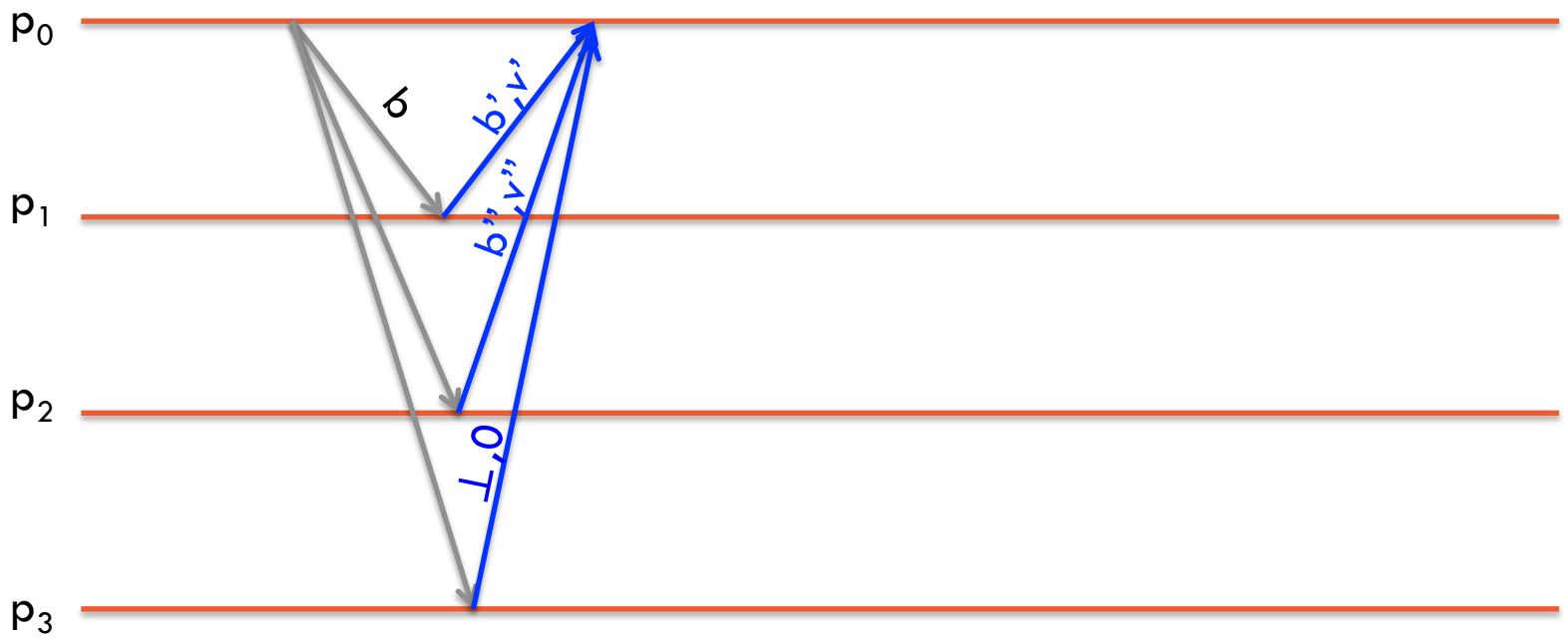
- How to decide upon an ongoing ballot?



Paxos

Prepare phase in detail (con't)

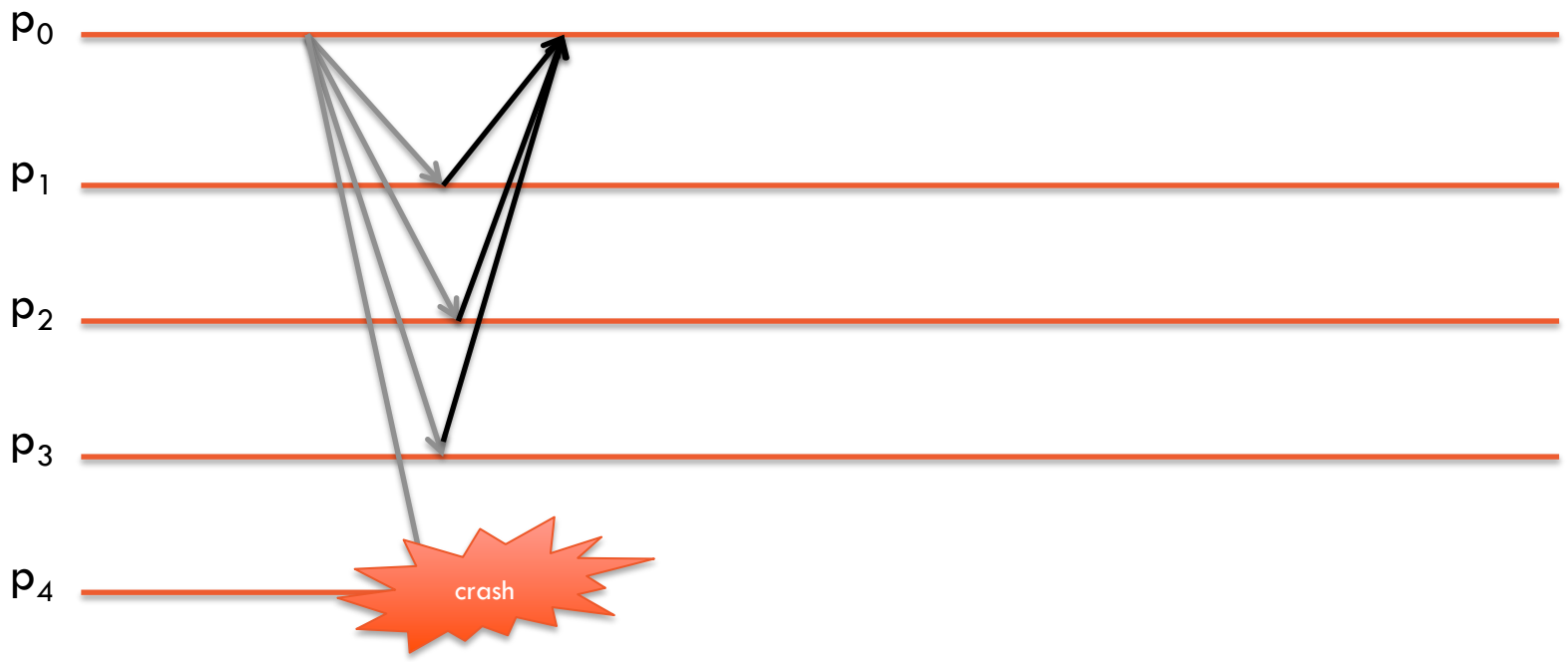
- How to decide upon an ongoing ballot?
  - responses include highest ballot number seen and voted value (b">b'>b)



Paxos

Prepare phase in detail (con't)

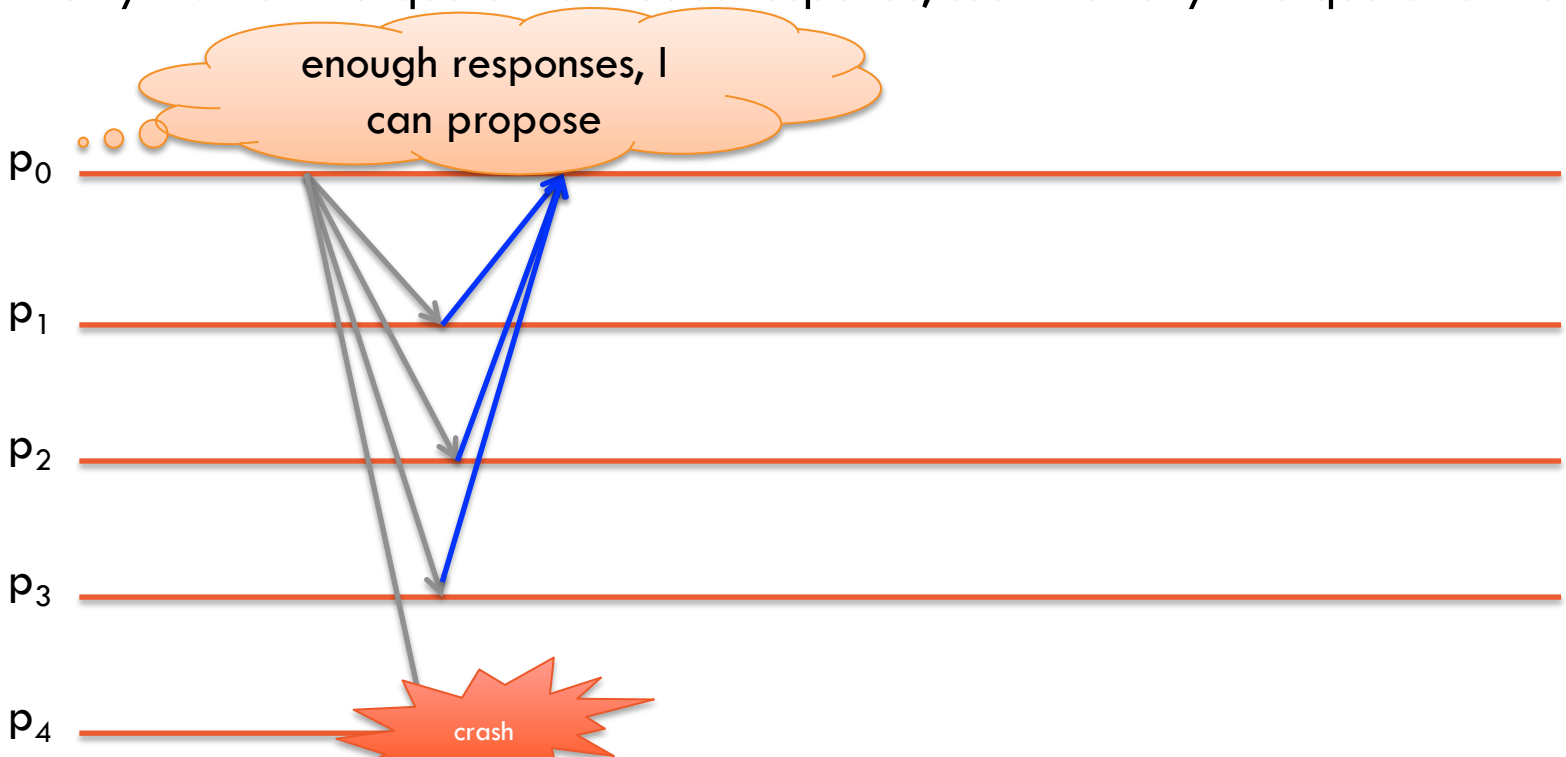
- What if a response is missing?



Paxos

Prepare phase in detail (con't)

- What if a response is missing?
  - only wait until a quorum of nodes responds, such that any two quorums intersect

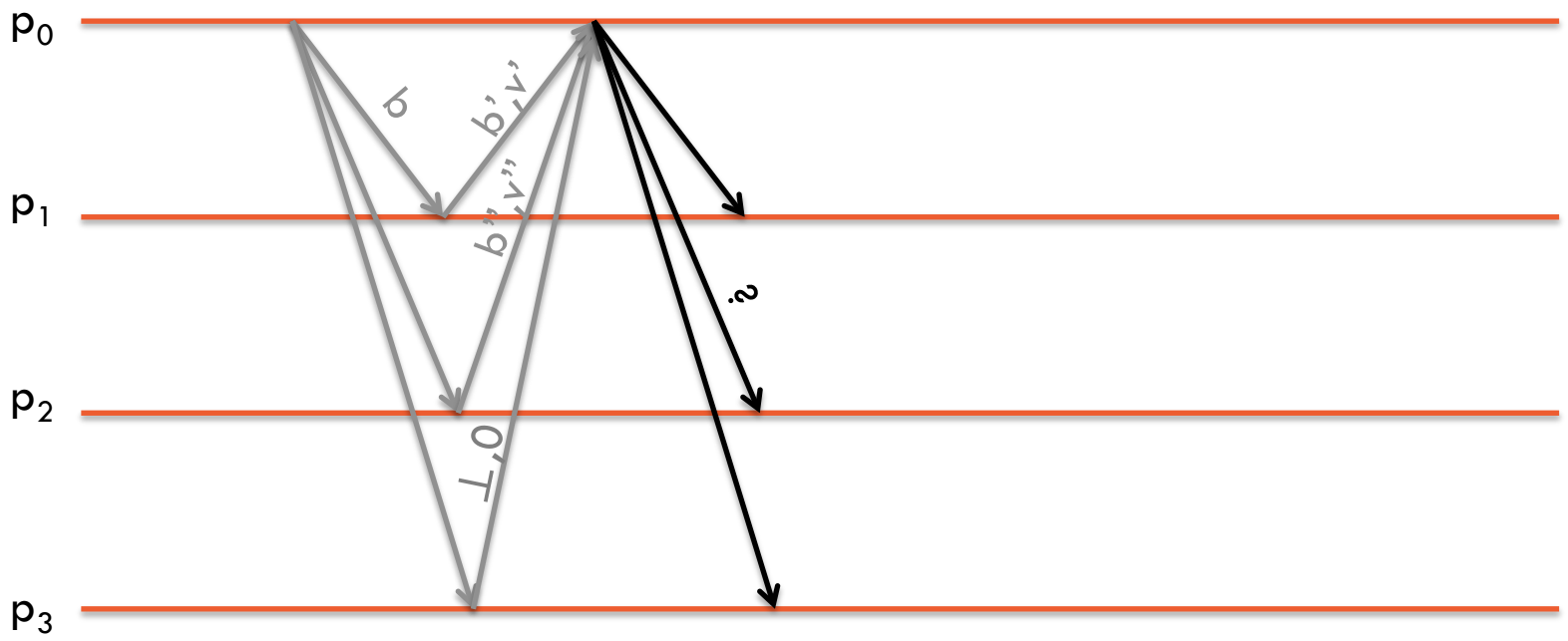


  - *Quorums* are mutually intersecting sets (e.g., {p0,p1,p2} ∩ {p1,p2,p3} ≠ ∅)

Paxos

Propose phase in detail

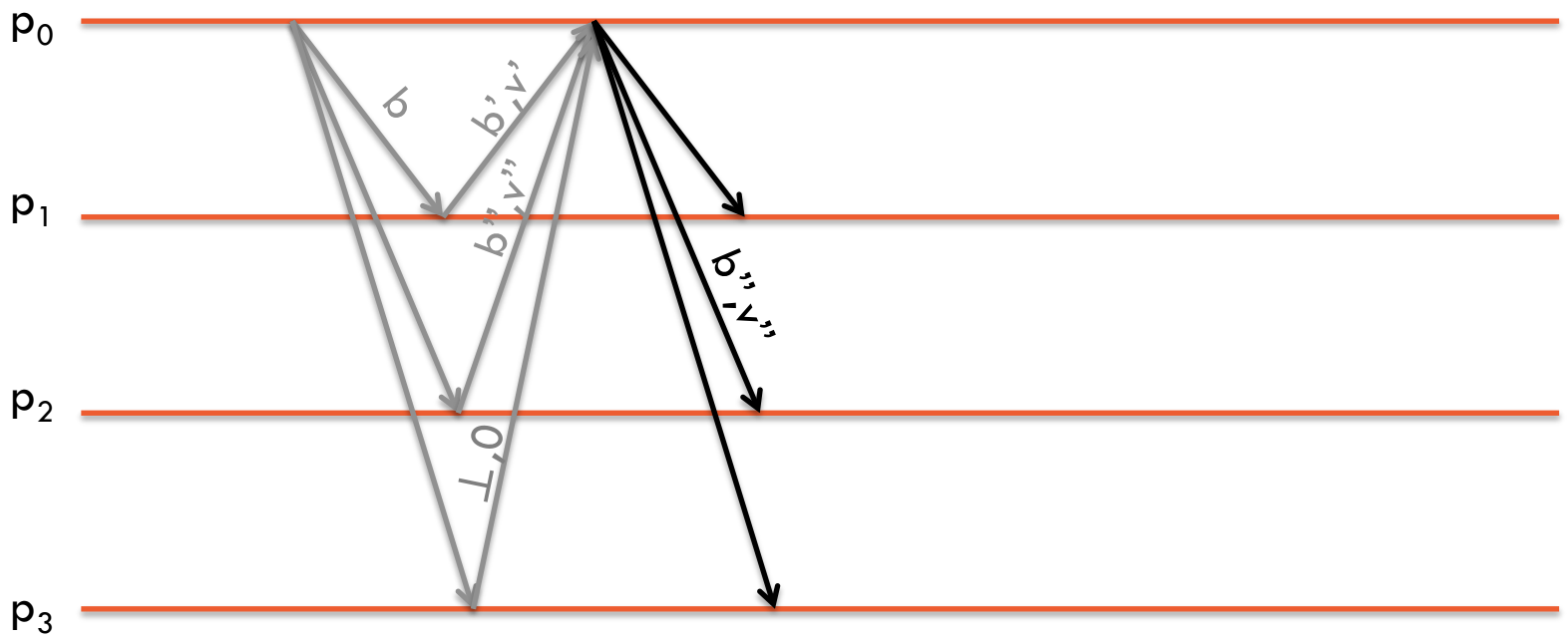
- How to choose a value to propose?



Paxos

Propose phase in detail (con't)

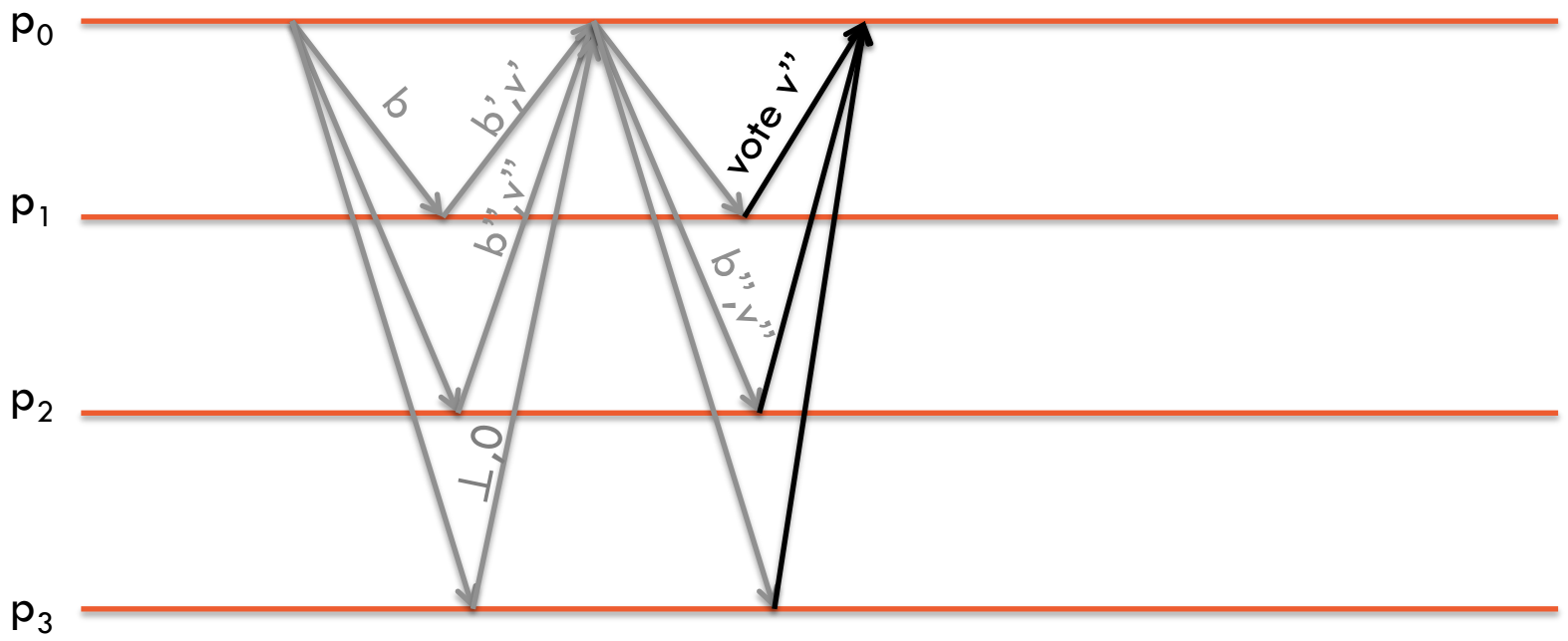
- How to choose a value to propose?
  - the value associated with the highest ballot number (if it exists) or a new one



Paxos

Propose phase in detail (con't)

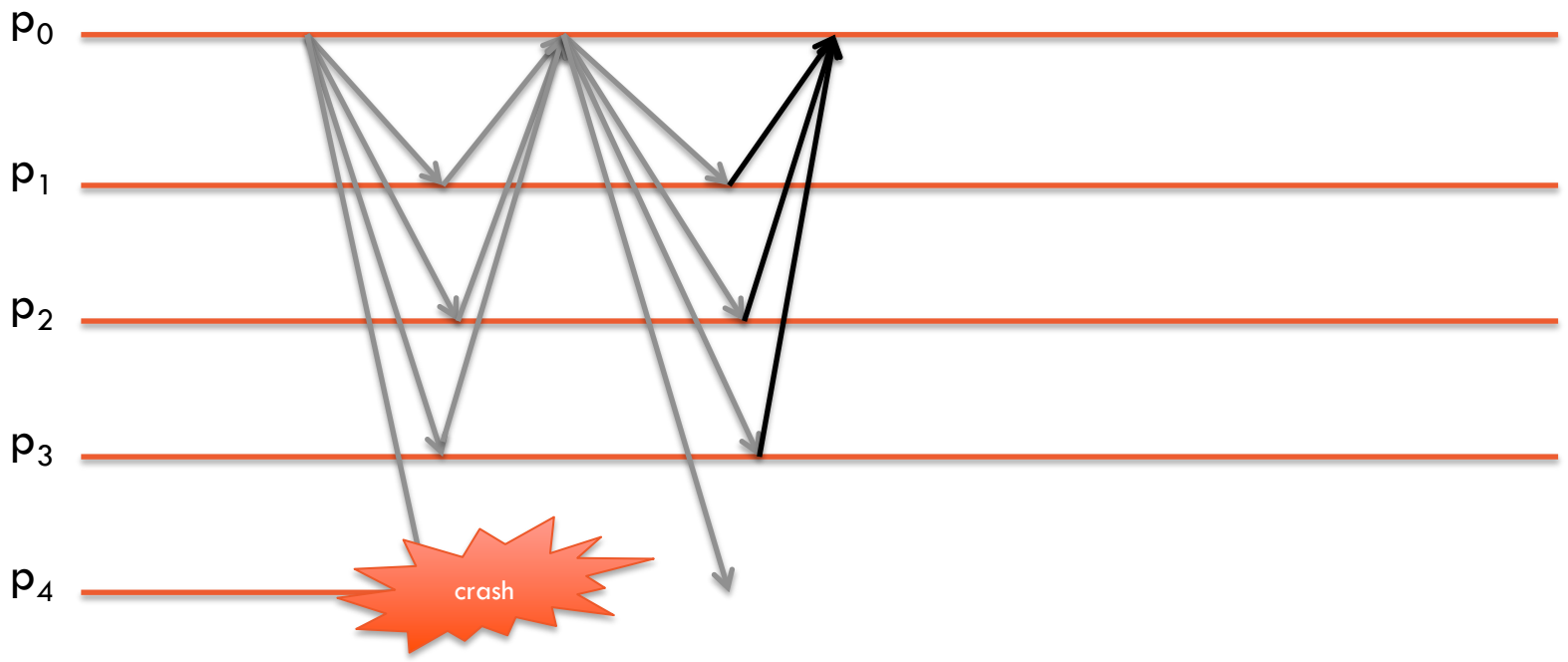
- How to vote?
  - if I have not seen any ballot with larger number, I can vote for this one



Paxos

Propose phase in detail (con't)

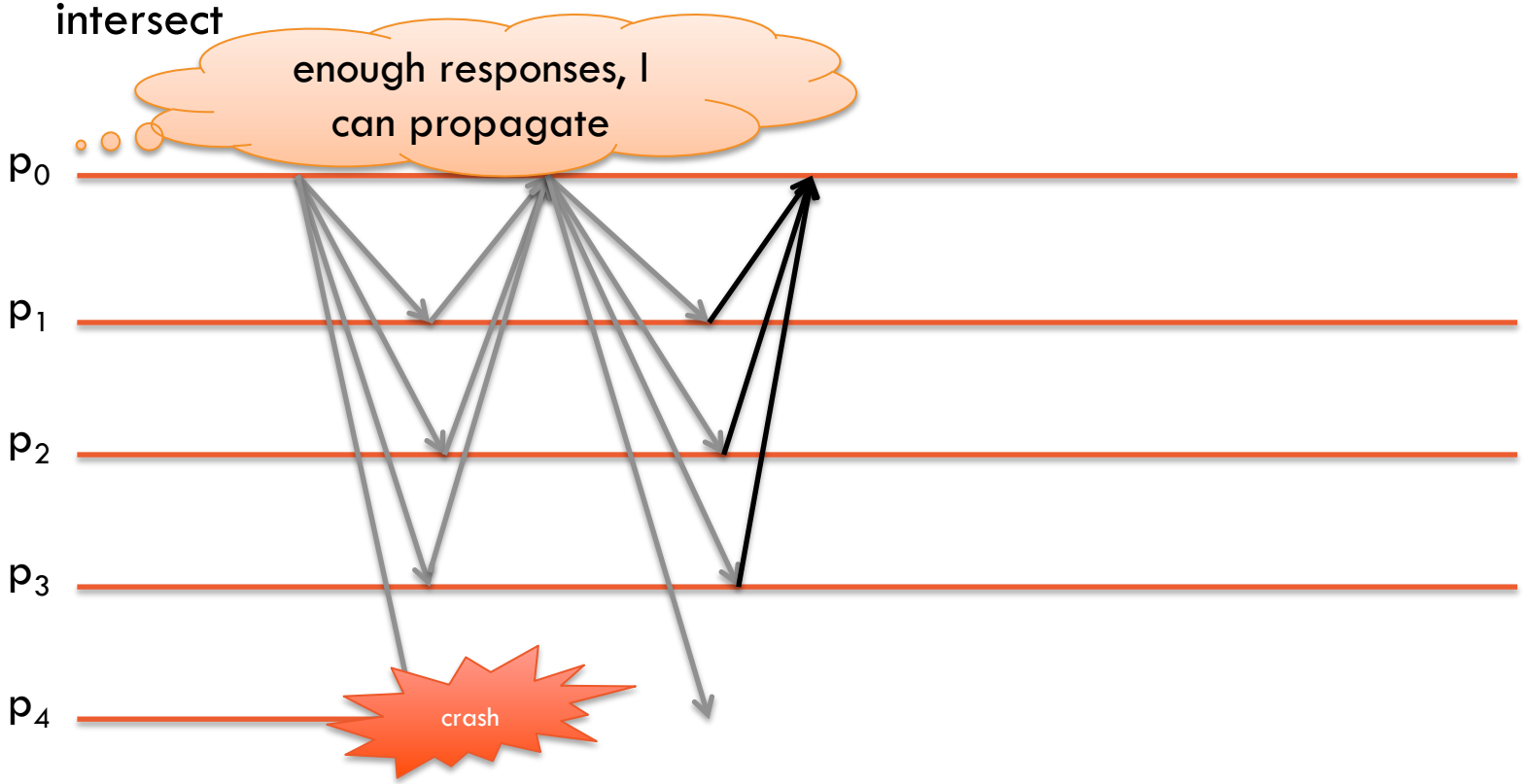
- What if a response is missing?



Paxos

Propose phase in detail (con't)

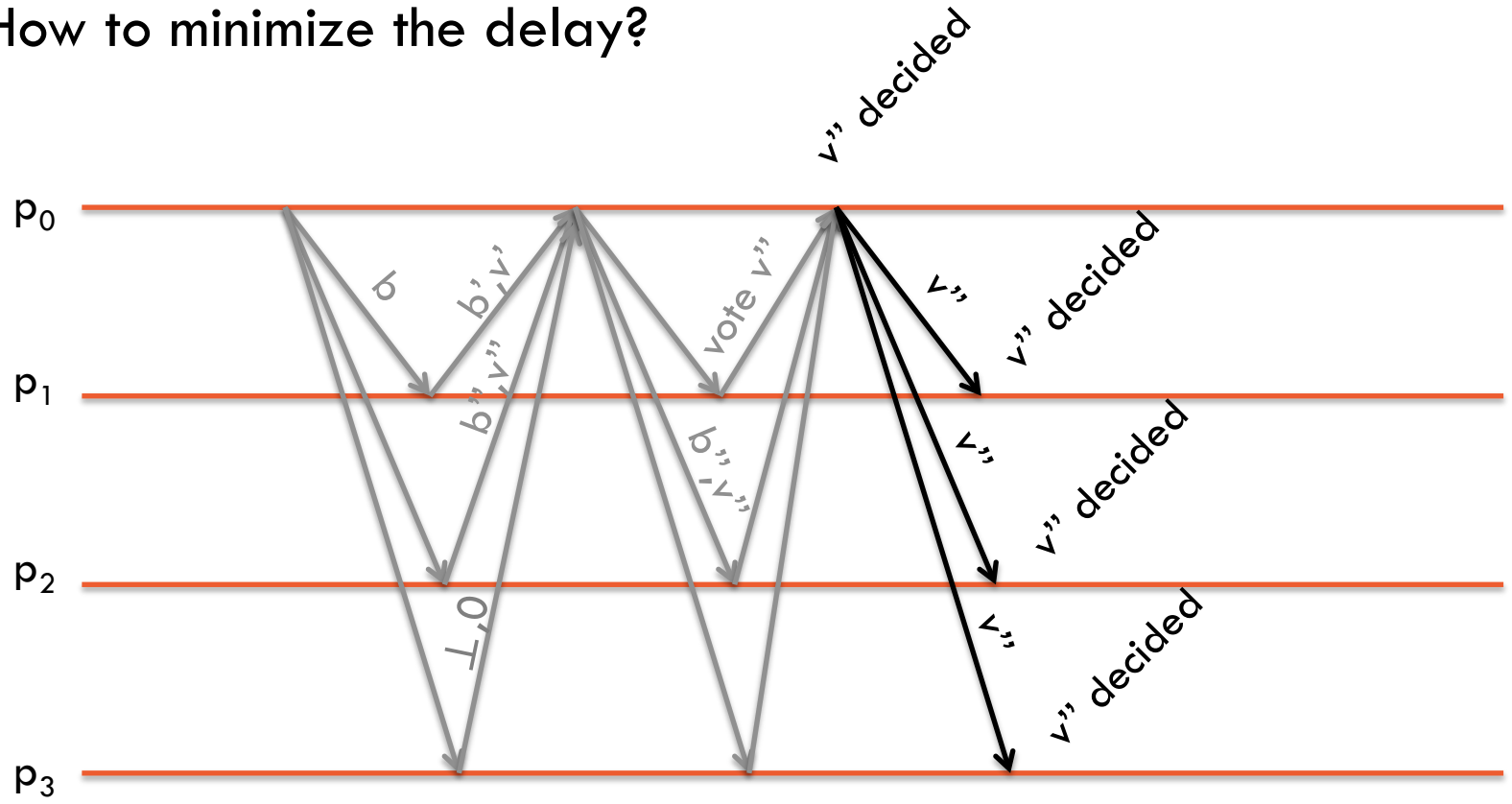
- What if a response is missing?
  - only wait until a quorum of nodes responds, such that any two quorums intersect



Paxos

Propagate phase in detail

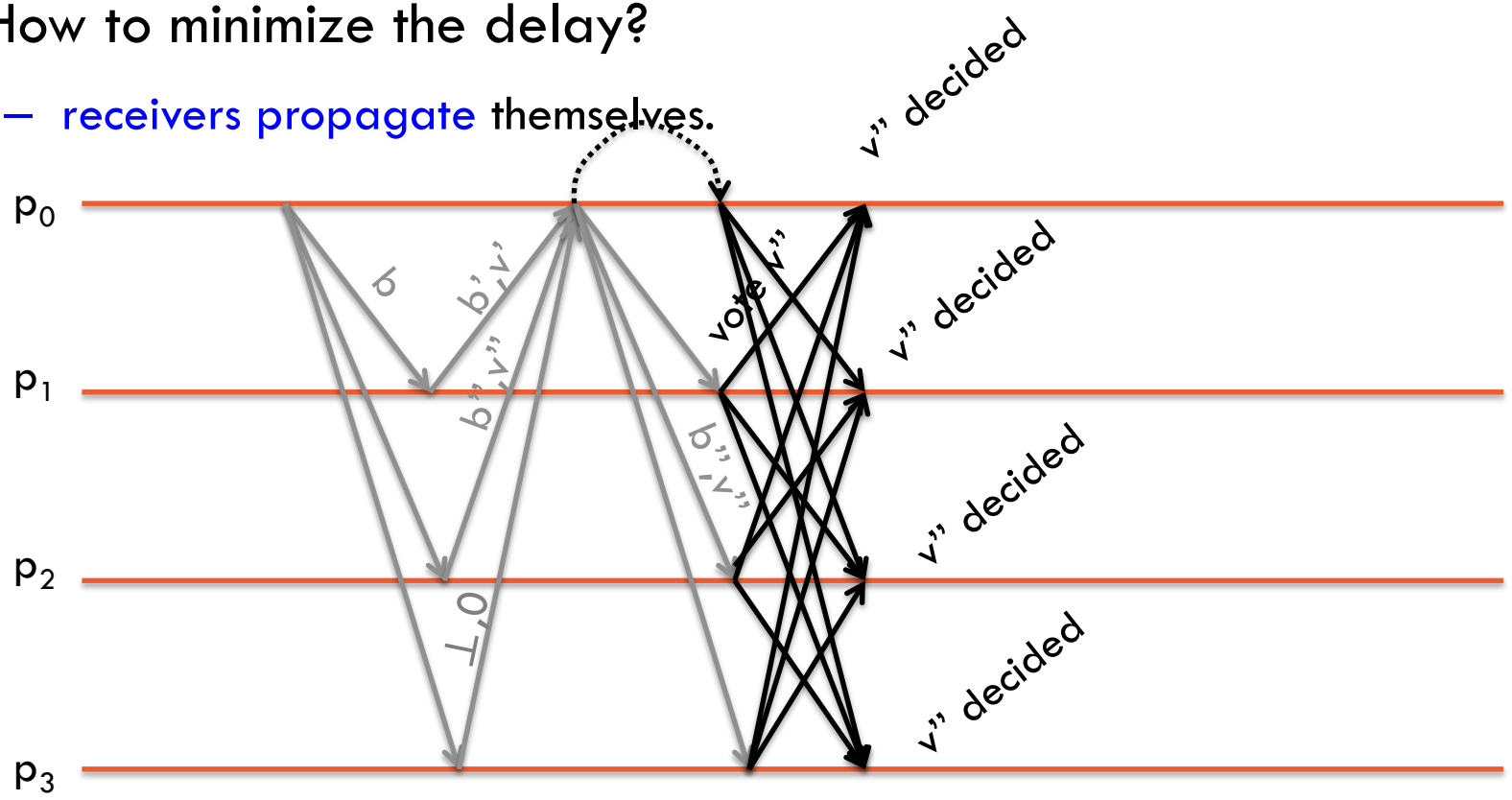
- How to minimize the delay?



Paxos

Propagate phase in detail (con't)

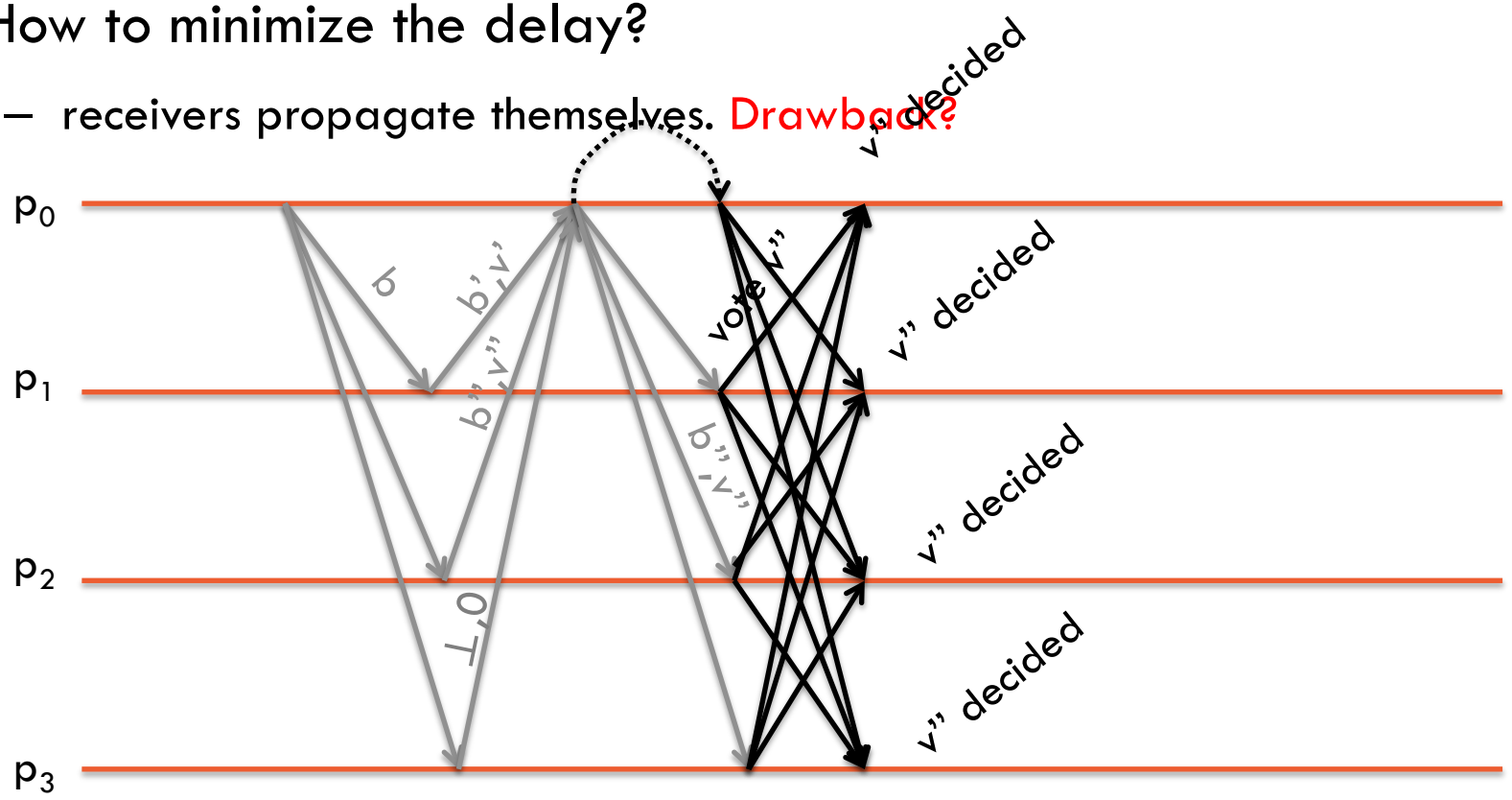
- How to minimize the delay?
  - receivers propagate themselves.



Paxos

Propagate phase in detail (con't)

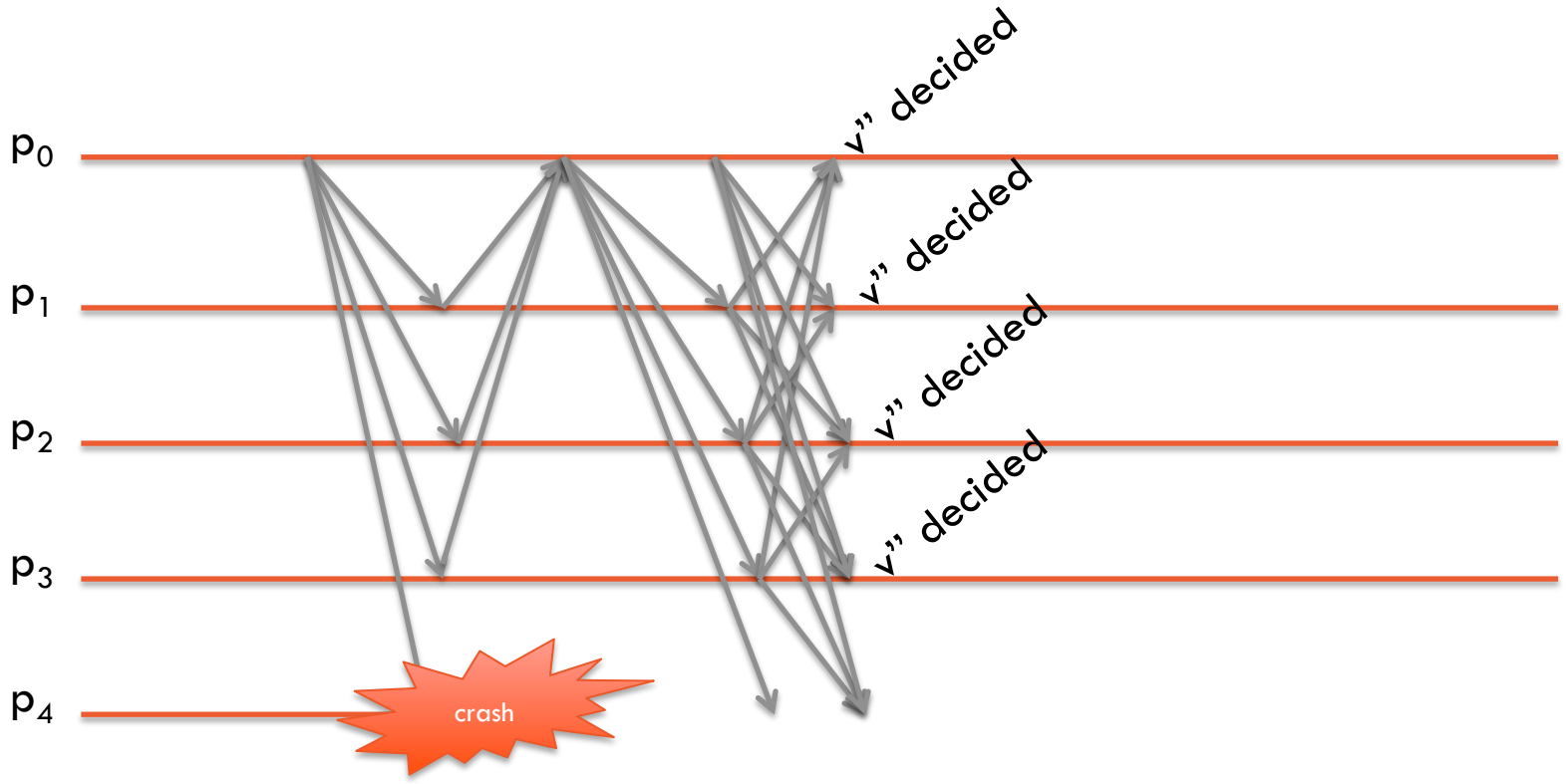
- How to minimize the delay?
  - receivers propagate themselves. Drawback?



Paxos

Propagate phase in detail (con't)

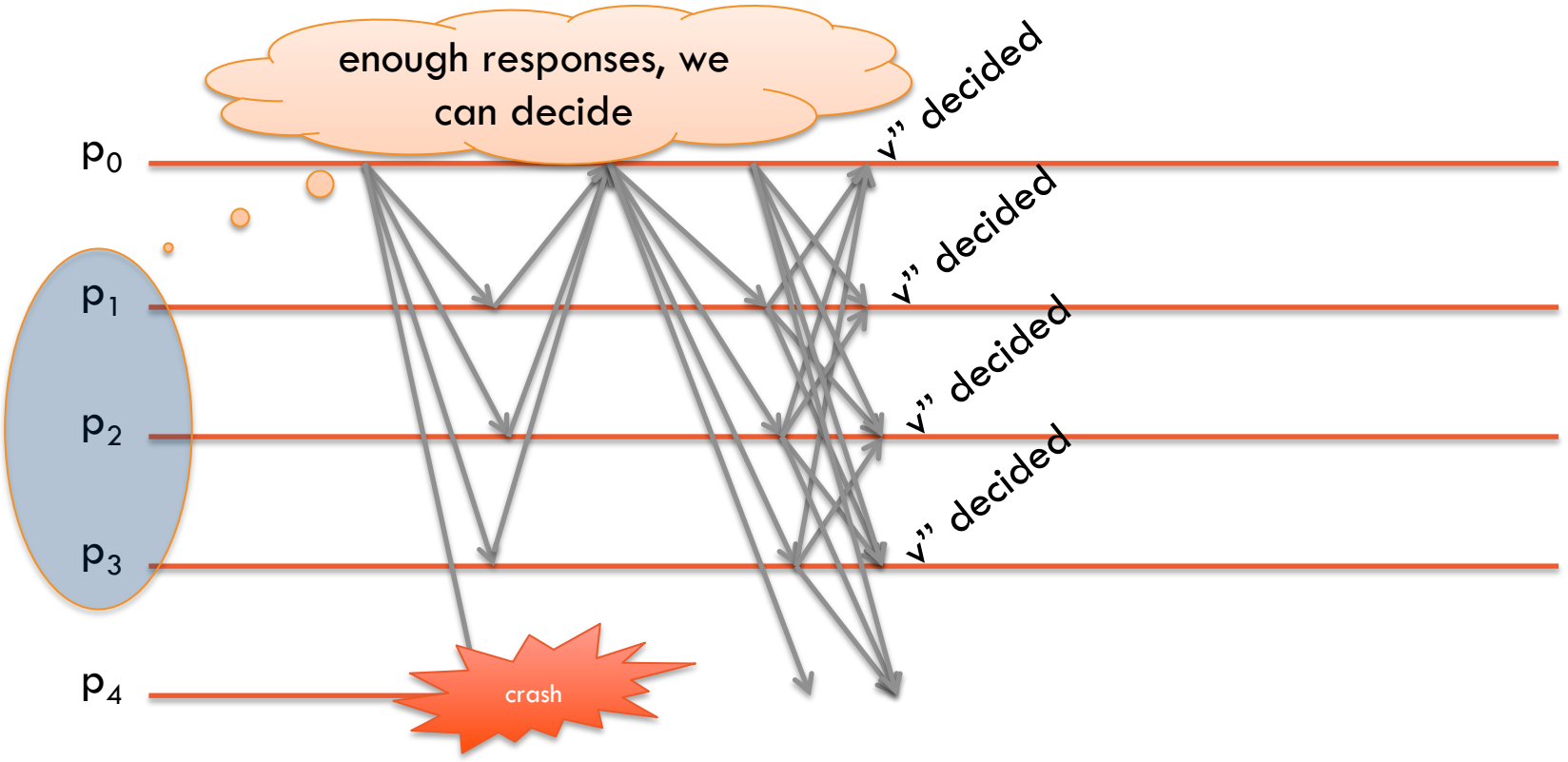
- What if the responses of one node are missing?



Paxos

Propagate phase in detail (con't)

- What if the responses of one node are missing?
  - Only wait until a quorum of nodes responds, such that any two quorums intersect



Paxos

Summary

- Paxos solves consensus in asynchronous systems despite crash failures only after network becomes reliable and messages get delivered
  - Termination is not guaranteed otherwise (validity/agreement never jeopardized)

Optimizations

- Sends negative acknowledgment during prepare/propose phases if a higher ballot number was seen (reduce waiting time)
- Multi-Paxos: A stable leader can skip the prepare phase (2 msg delays, w/o clt msg)
- Fast-Paxos: The client can contact directly the acceptors (2 msg delays, clt -> learners)
- 1st (resp. 2nd) phase contacts a quorum of proposers (resp. acceptors,)
  - $P \cap A \neq \varnothing$, however, no need for $P \cap P' \neq \varnothing$
  - Learners (3rd phase) = set of correct nodes

Applications:

Google: Chubby distributed lock service, Spanner [OSDI 2012] and Megastore RDBMS [CIDR 2011]

MSR Autopilot [OSR 2007]: Software Provisioning and Deployment