YSC4231: Parallel, Concurrent and Distributed Programming

Byzantine Fault Tolerance and Blockchains

Wrap-Up

Why Distributed Consensus is difficult?

- Arbitrary message delays (asynchronous network)
- Independent parties (nodes) can go offline (and also back online)
- Network partitions
- Message reorderings
- Malicious (Byzantine) parties

Why Distributed Consensus is difficult?

- Arbitrary message delays (asynchronous network)
- Independent parties (nodes) can go offline (and also back online)
- Network partitions
- Message reorderings
- Malicious (Byzantine) parties

Byzantine Generals Problem

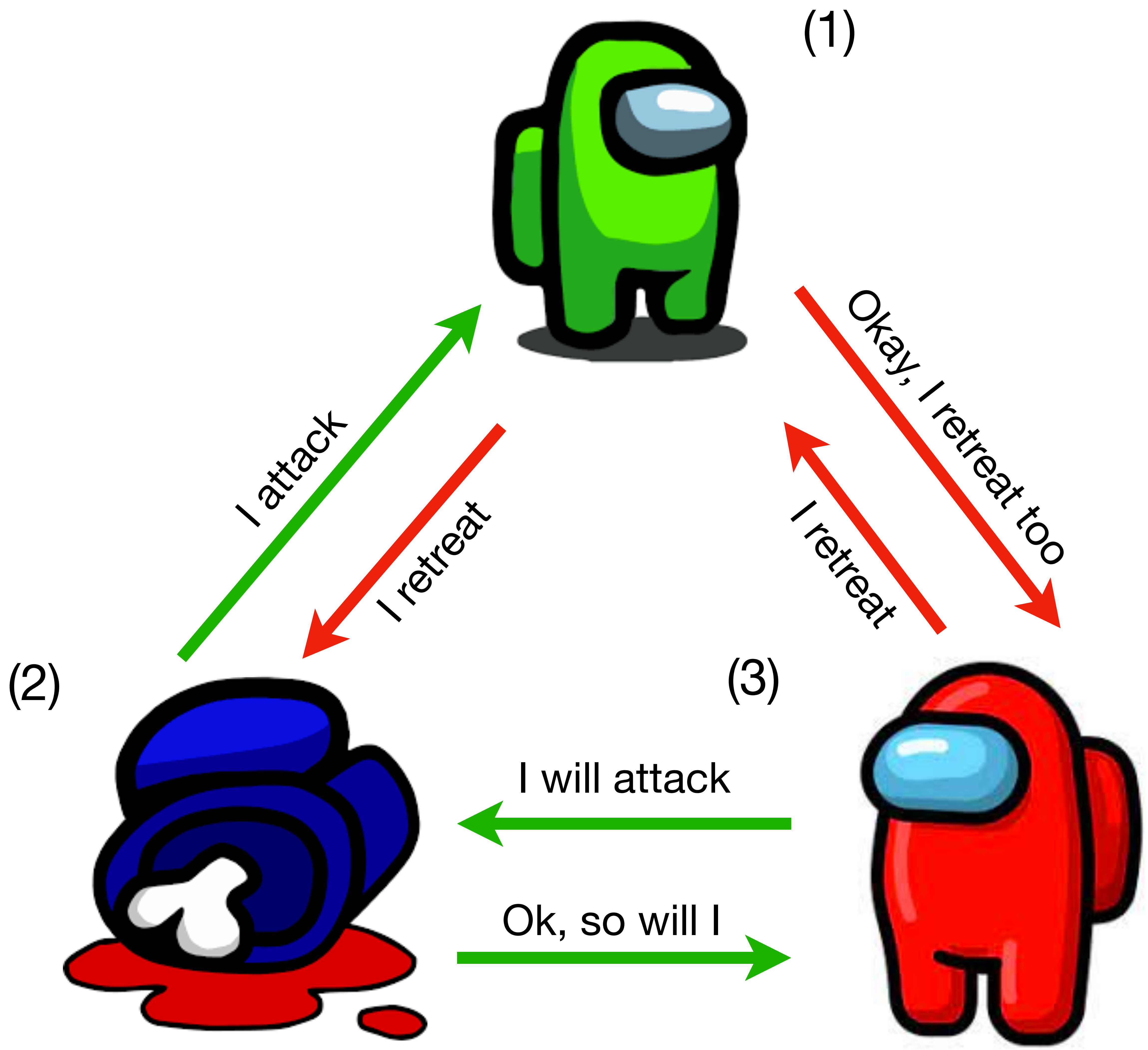
- A Byzantine army decides to attack/retreat
- N generals, f of them are *traitors* (can *collude*)
- Generals camp outside the battle field:
  decide individually based on their field information
- Exchange their plans by unreliable *messengers*
  - Messengers can be *killed*, can be *late*, *etc*.
  - Messengers *cannot* ***forge*** a general's seal on a message

Byzantine Consensus

- All loyal generals decide upon the *same* plan of action.
- A *small* number of traitors (**f << N**) *cannot* cause the loyal generals to adopt a bad plan or *disagree* on the course of actions.
- All the usual consensus properties:
  uniformity (amongst the loyal generals), *non-triviality*, and *irrevocability*.

Why is Byzantine Agreement Hard?

- Simple scenario
  - 3 generals, general (3) is an ~~imposter~~ traitor
  - Traitor (3) sends different plans to (1) and (2)
  - If decision is based on majority
    - (1) and (2) decide differently
    - (2) attacks and gets defeated
- More complicated scenarios
  - Messengers get killed, spoofed
  - Traitors confuse others:
    (3) tells (1) that (2) retreats, etc



Byzantine Consensus in Computer Science

- A *general* is a **program component/replica/node**
  - *Replicas* communicate via ***messages/remote procedure calls***
  - *Traitors* are *malfunctioning replicas or adversaries*

- *Byzantine army* is a ***deterministic replicated service***
  - All (good) replicas should act similarly and execute the *same logic*
  - The service should cope with failures, keeping its state *consistent* across the replicas

- Seen in *many applications*:
  - replicated file systems, backups, distributed servers
  - shared ledgers between banks, decentralised *blockchain protocols*

Byzantine Fault Tolerance Problem

- Consider a system of similar distributed *replicas* (*nodes*)
  - N replicas in total
  - f of them might be faulty (crashed or compromised)
  - All replicas initially start from the *same state*

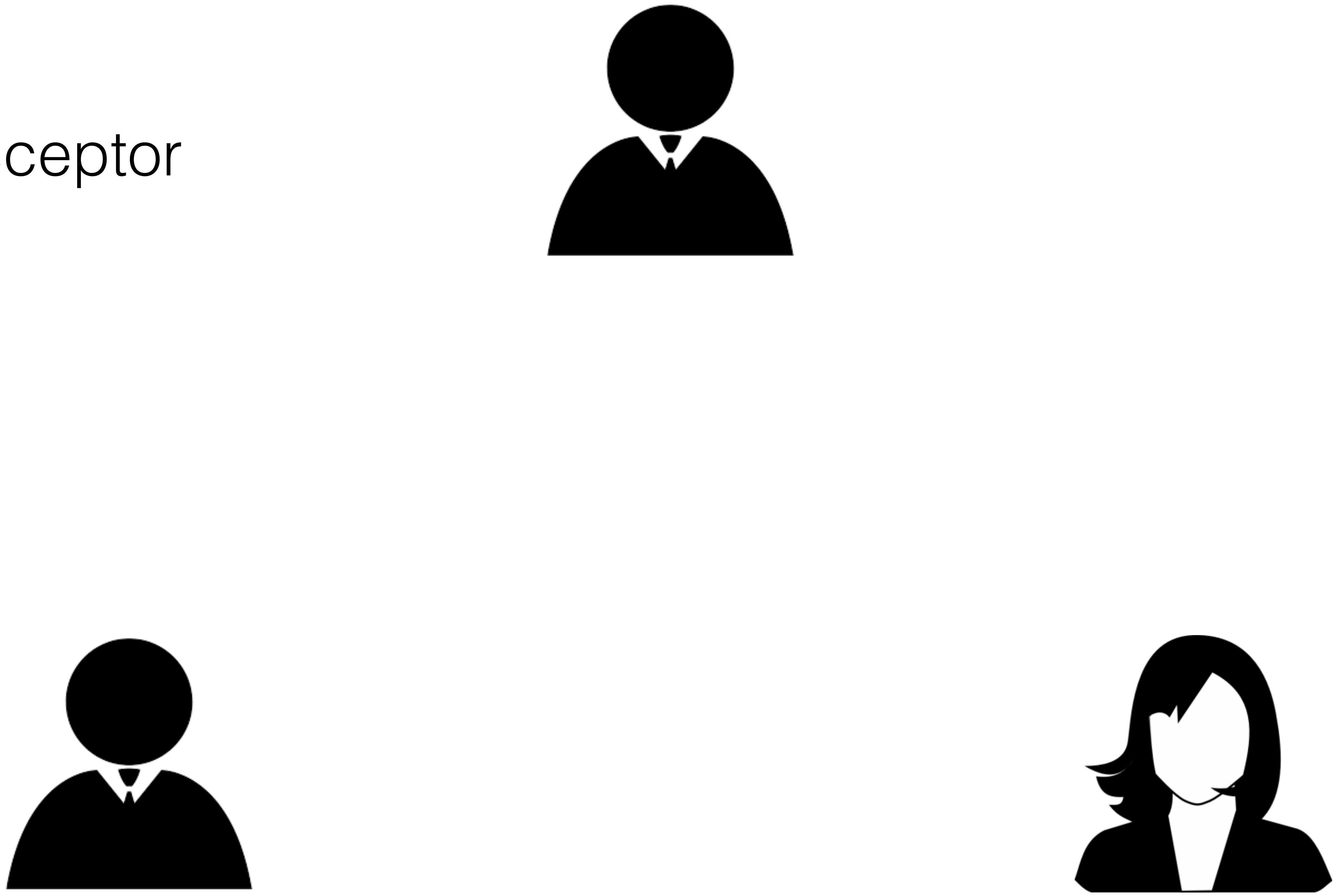
- Given a *request/operation* (e.g., a *transaction*), the goal is to
  - guarantee that all non-faulty replicas *agree* on the next state
  - provide system *consistency* even when some replicas may be inconsistent

Previous lecture: Paxos

- Communication model
  - Network is *asynchronous*: messages are *delayed arbitrarily*, but eventually delivered; they *are not deceiving*.
  - Protocol tolerates (benign) crash-failure
- Key design points
  - Works in *two phases* — secure quorum, then commit
  - Require at least $2f + 1$ replicas to tolerate f faulty replicas

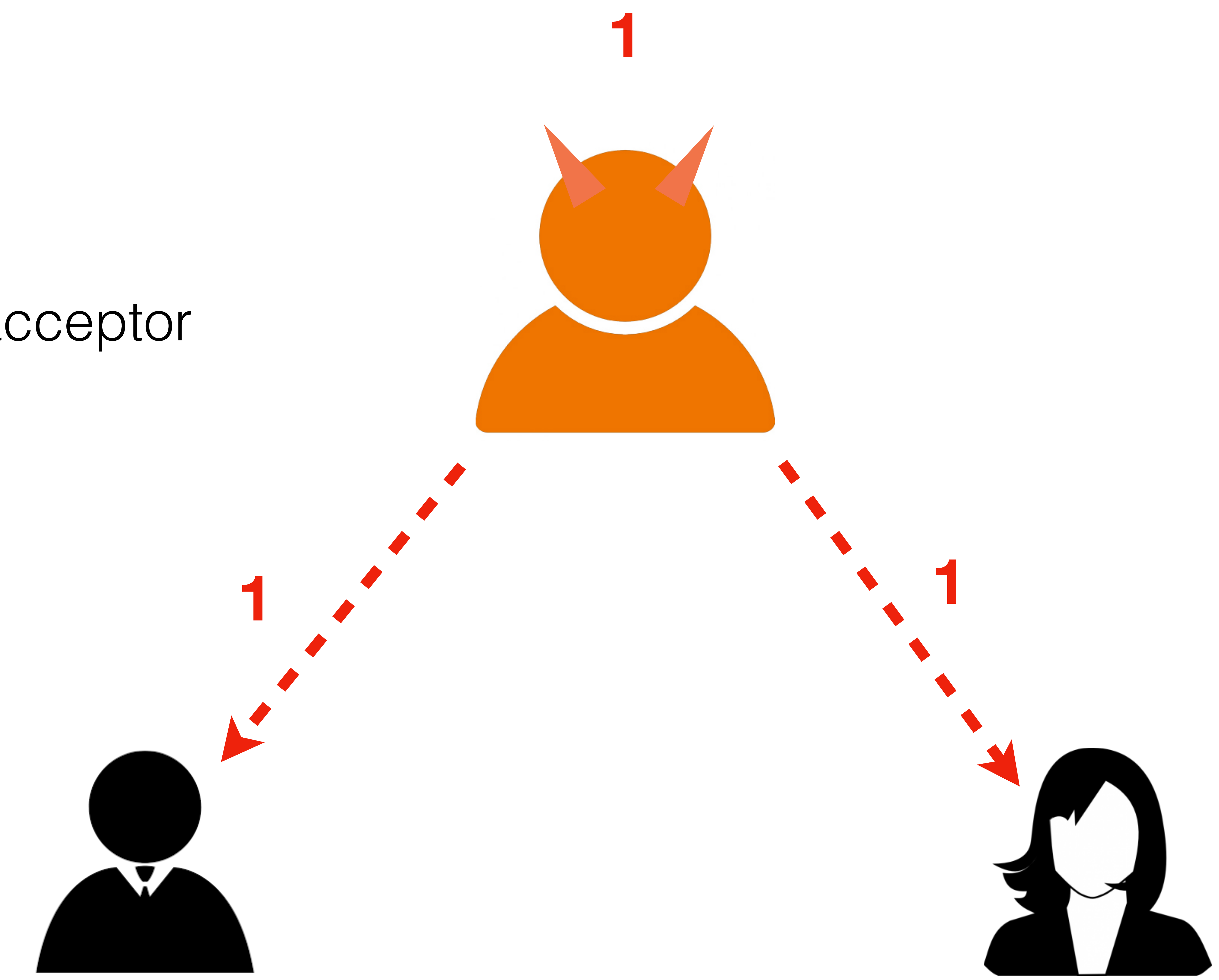
Paxos and Byzantine Faults

- N = 3, f = 1
- N/2 + 1 = 2 are good
- everyone is proposer/acceptor



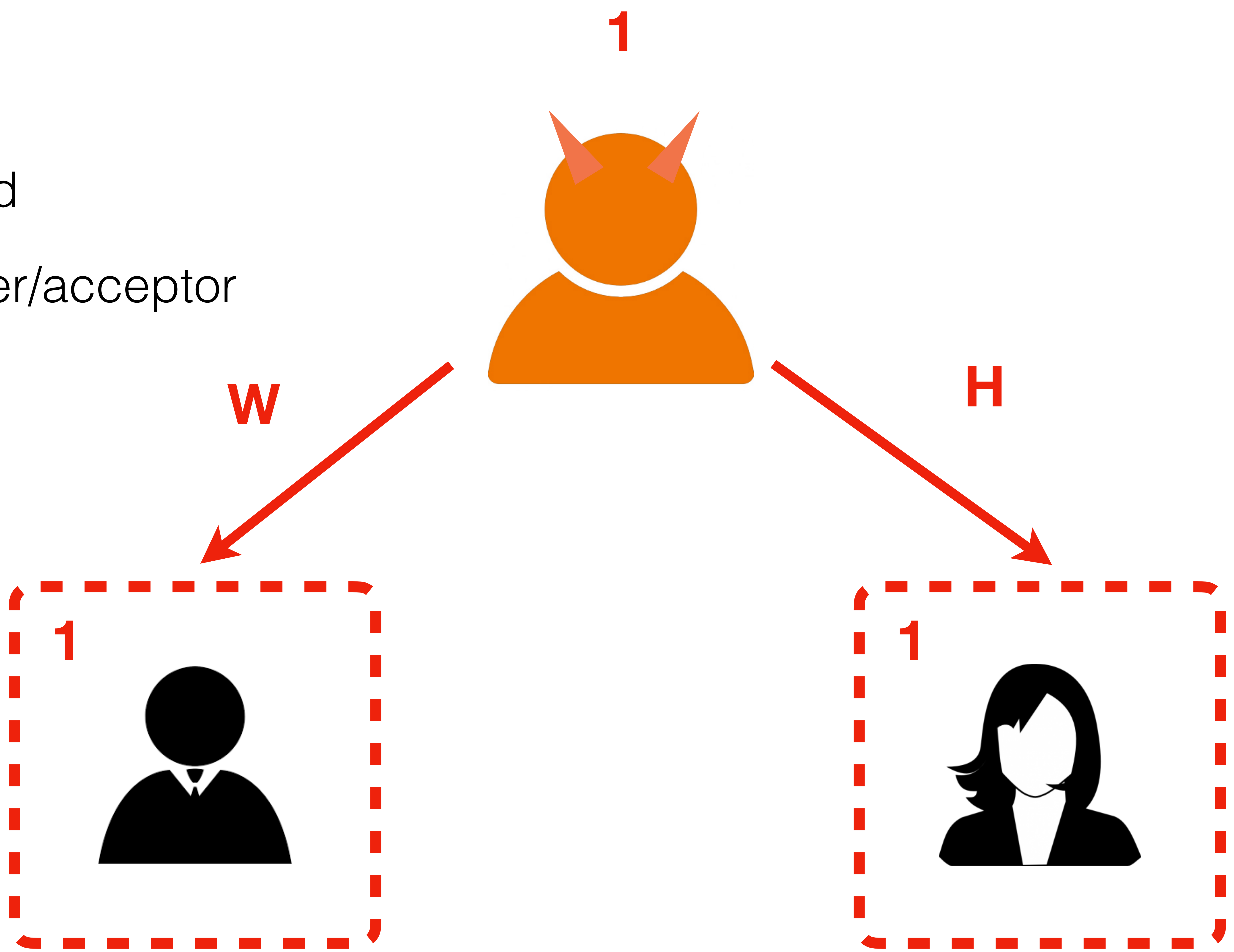
Paxos and Byzantine Faults

- N = 3, f = 1
- N/2 + 1 = 2 are good
- everyone is proposer/acceptor



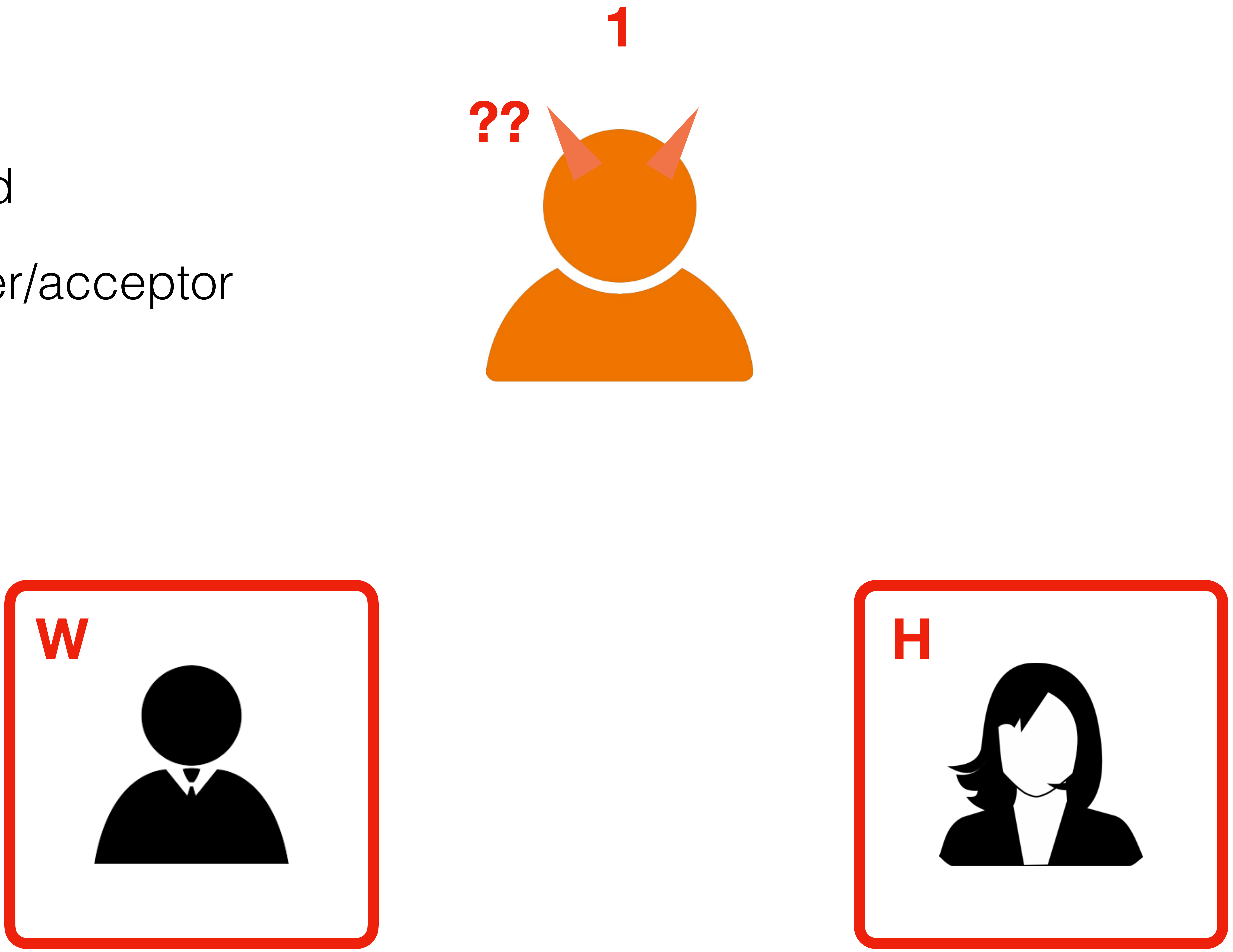
Paxos and Byzantine Faults

- N = 3, f = 1
- N/2 + 1 = 2 are good
- everyone is proposer/acceptor



Paxos and Byzantine Faults

- N = 3, f = 1
- N/2 + 1 = 2 are good
- everyone is proposer/acceptor

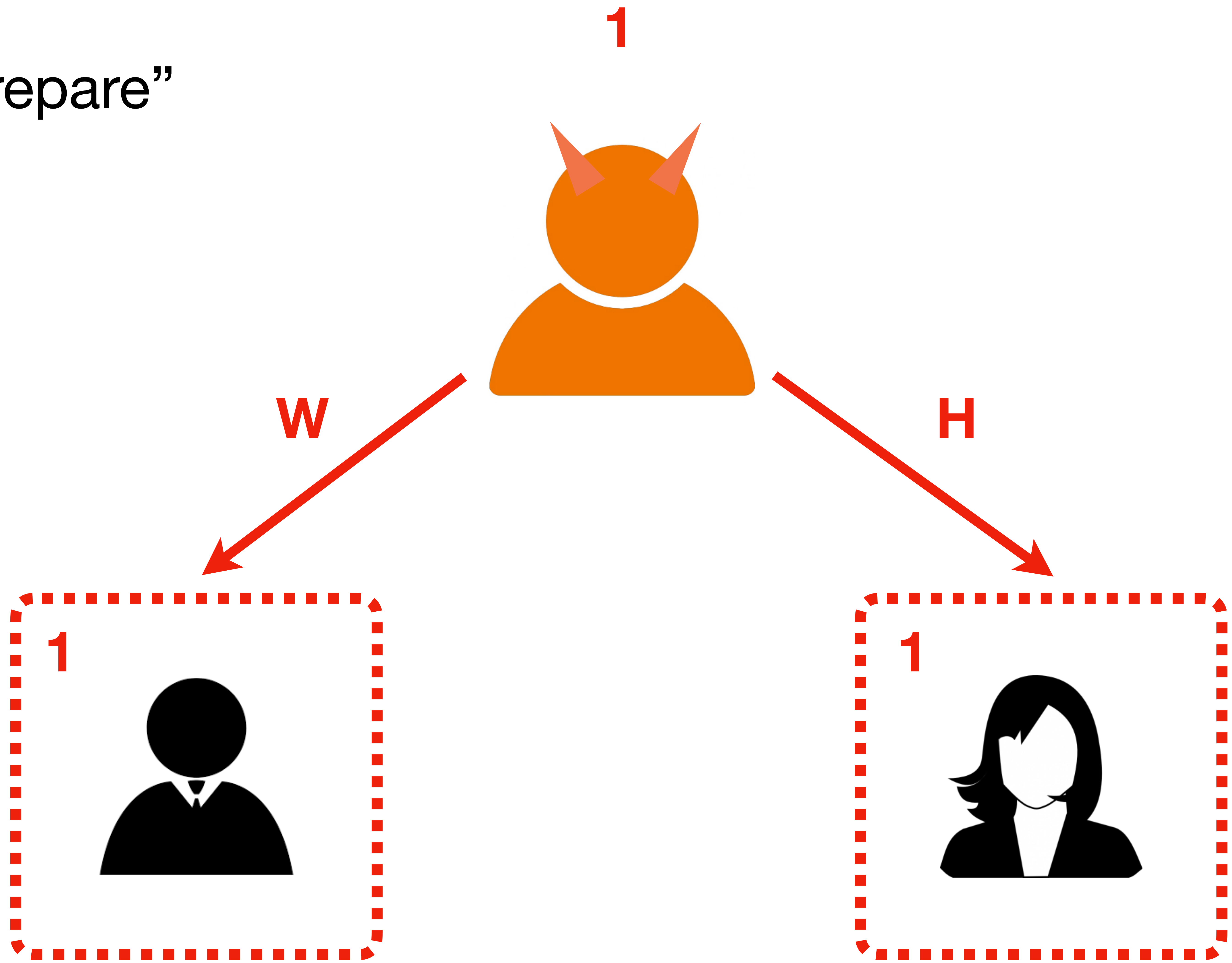


What went wrong?

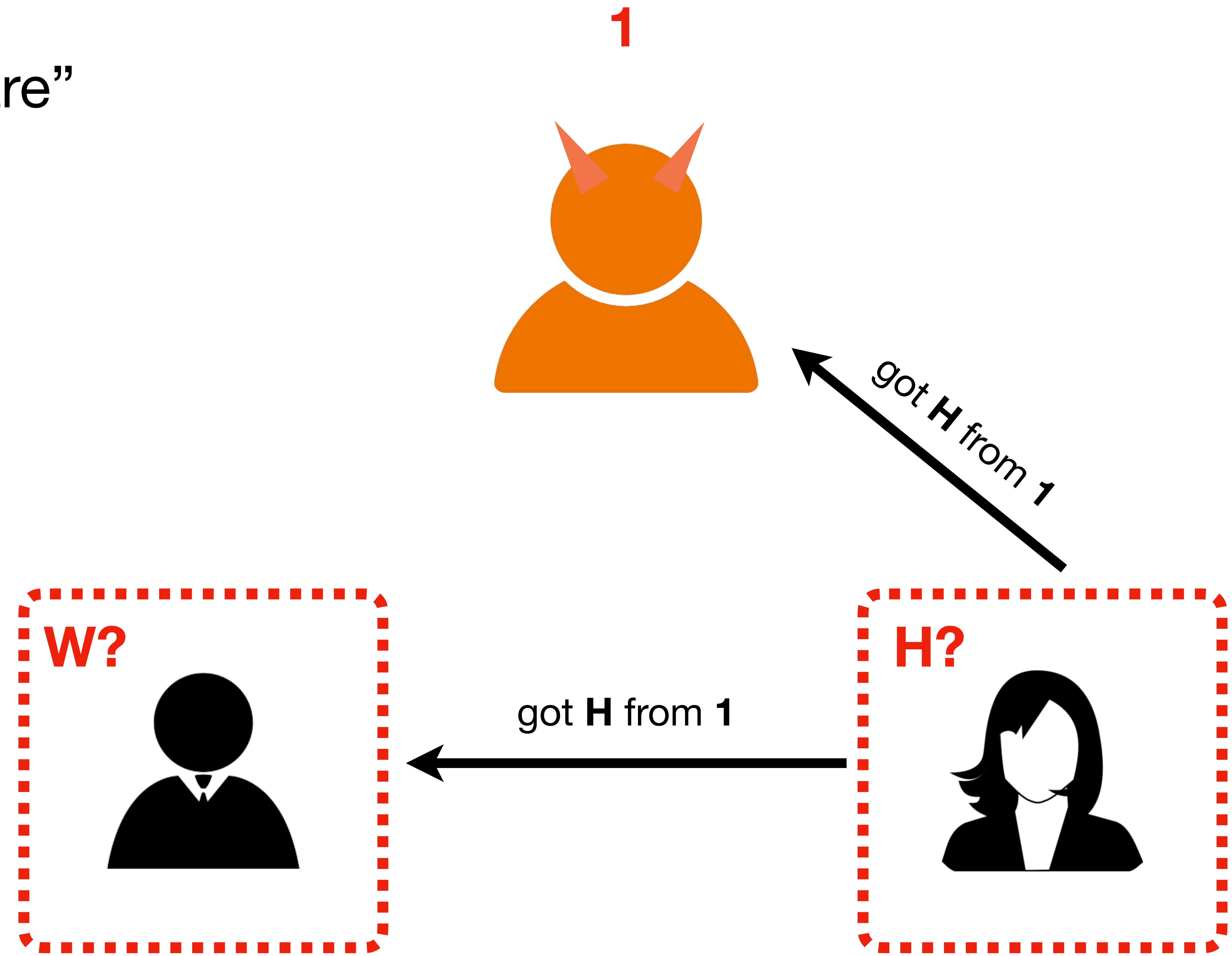
- Problem 1:
  Acceptors did not communicate with each other to check the consistency of the values proposed to everyone.

- Let us try to fix it with an additional Phase 2 (Prepare), executed *before* everyone commits in Phase 3 (Commit).

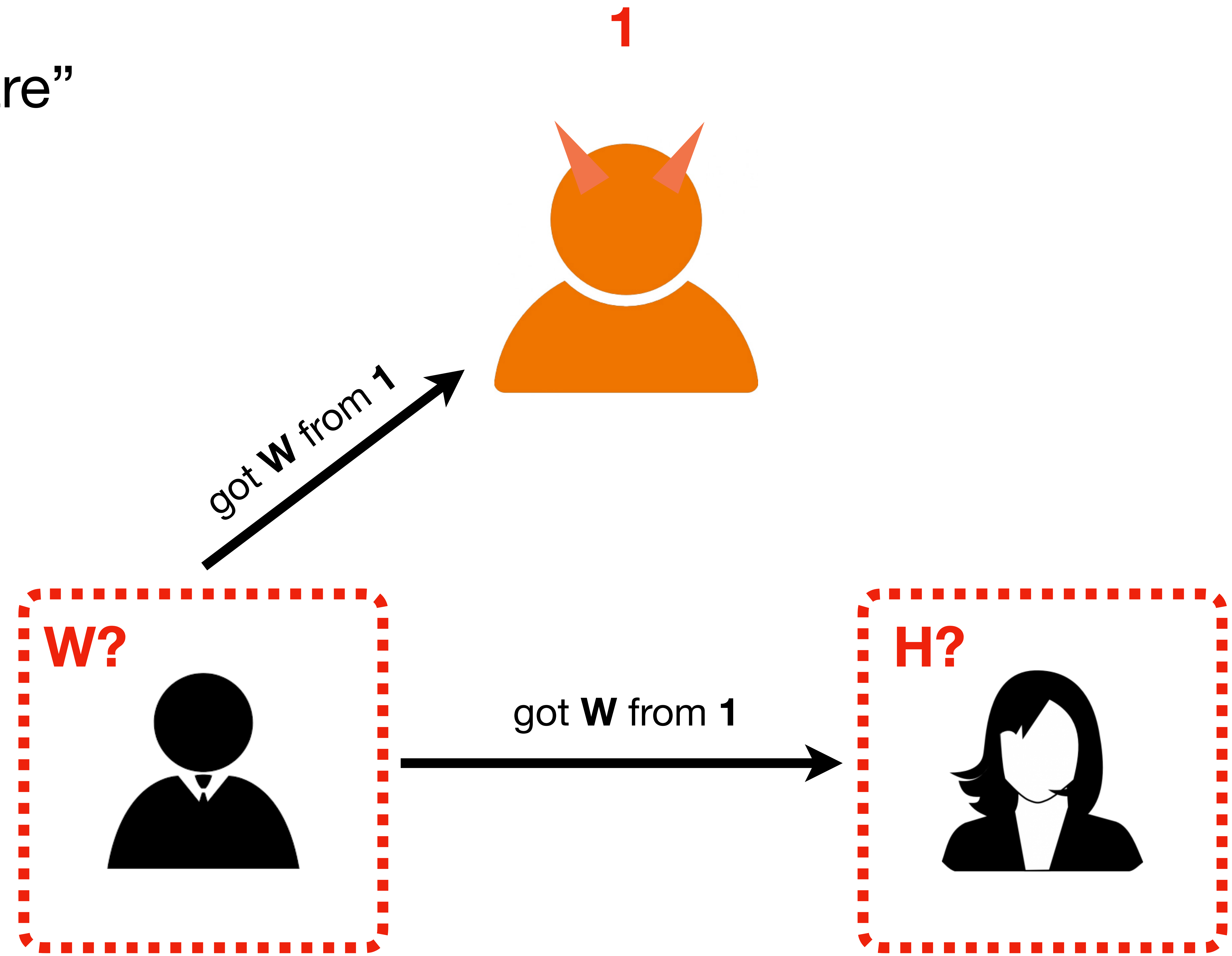
Phase 1: “Pre-prepare”



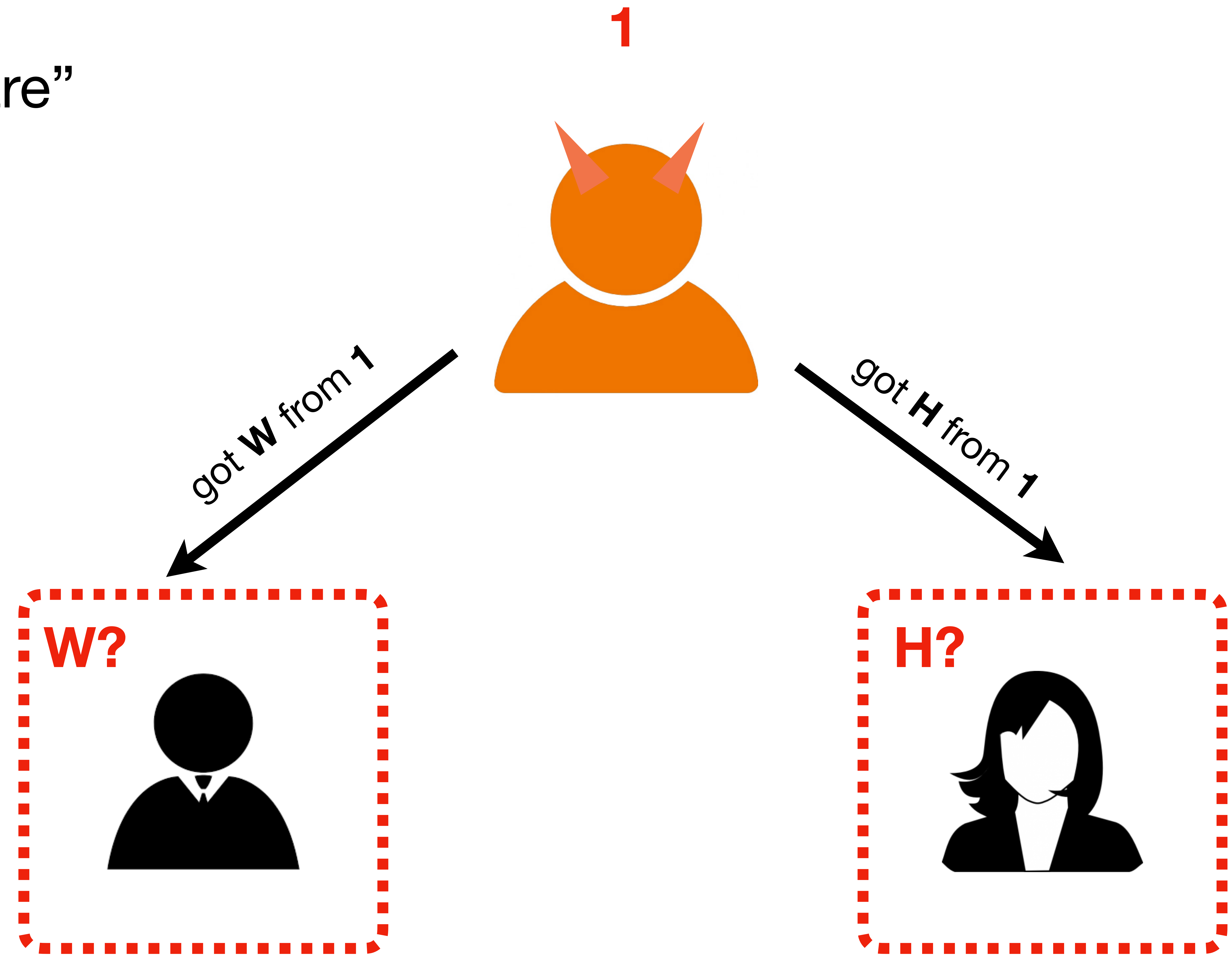
Phase 2: “Prepare”



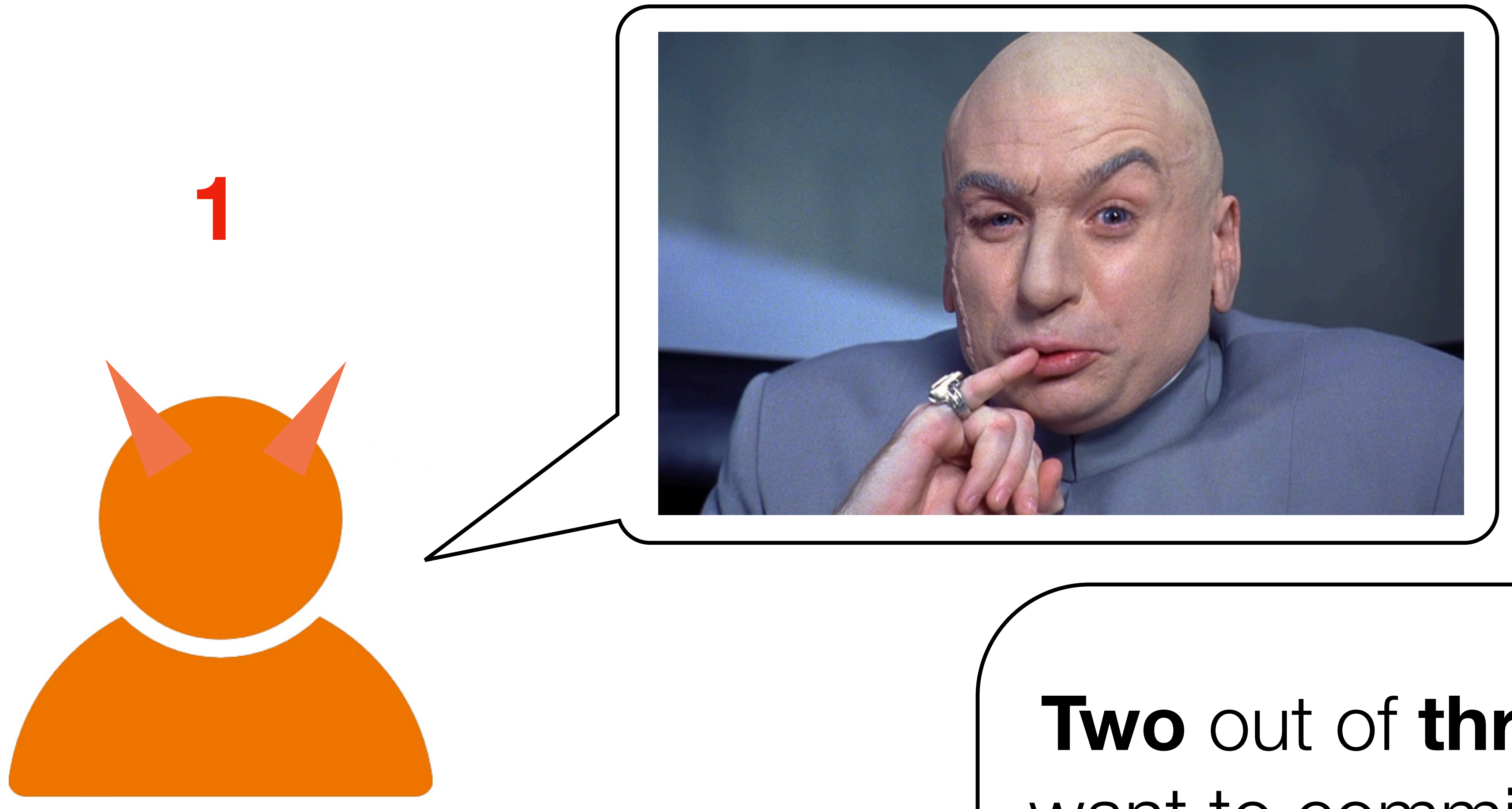
Phase 2: “Prepare”

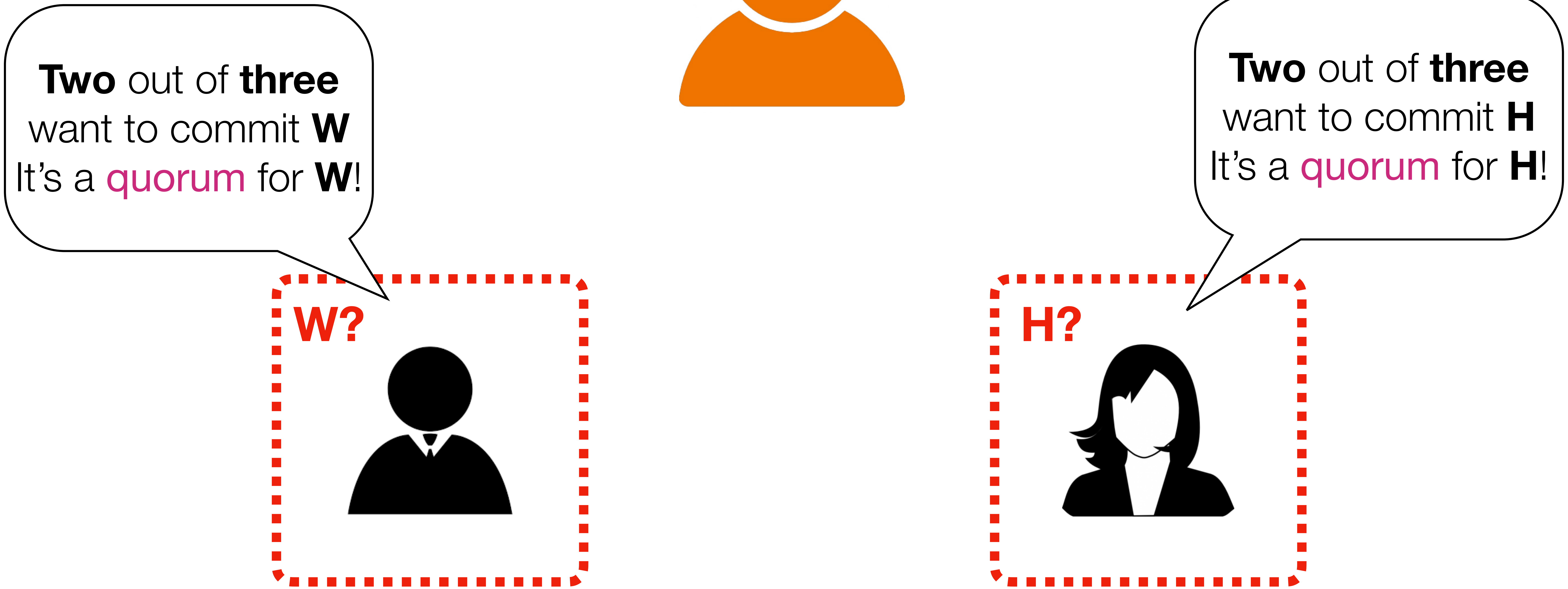


Phase 2: “Prepare”

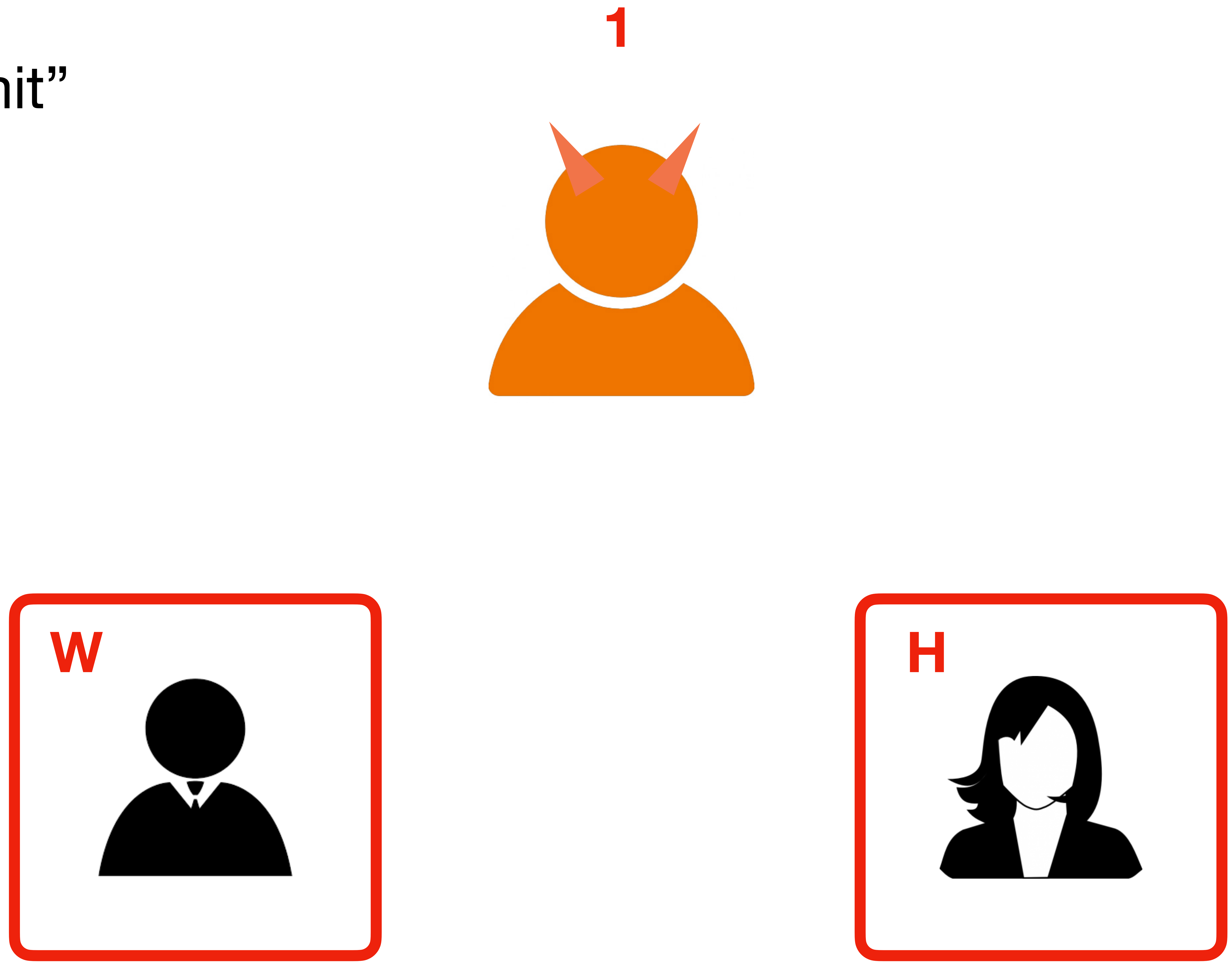


Phase 2: “Prepare”





Phase 3: “Commit”



What went wrong now?

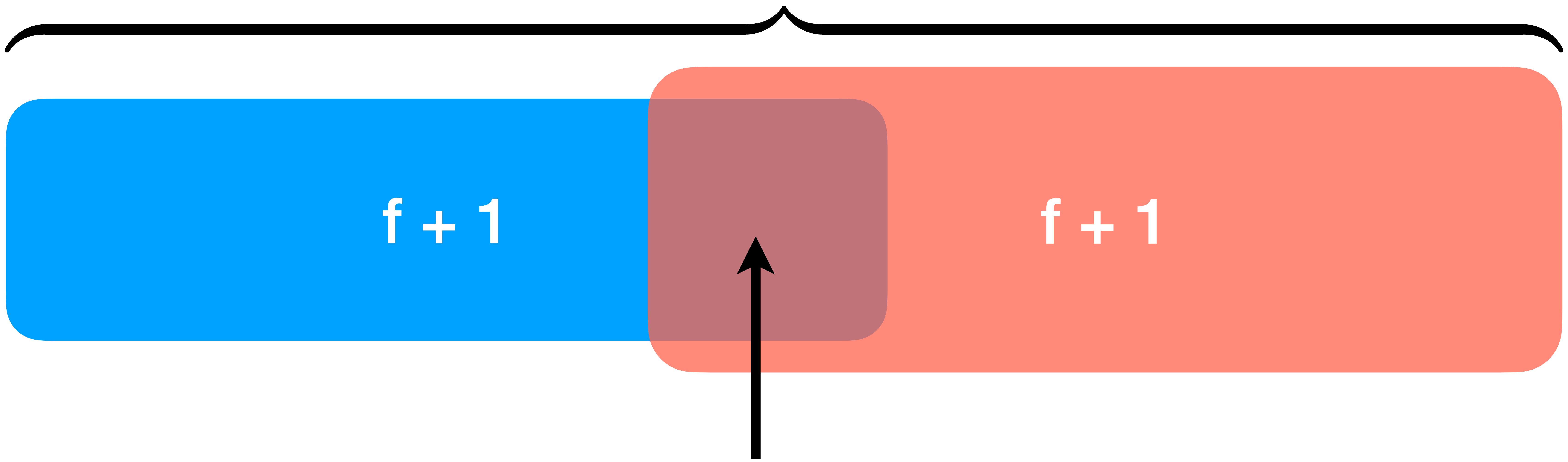
- **Problem 2**:
  Even though the acceptors communicated, the *quorum size* was *too small* to avoid “contamination” by an adversary.

- We can fix it by **increasing** the quorum size relative to the *total number of nodes*.

Choosing the Quorum Size

- *Paxos:* any two quorums must have non-empty intersection

$N \geq 2 * f + 1$



Sharing *at least one* node: must agree on the value

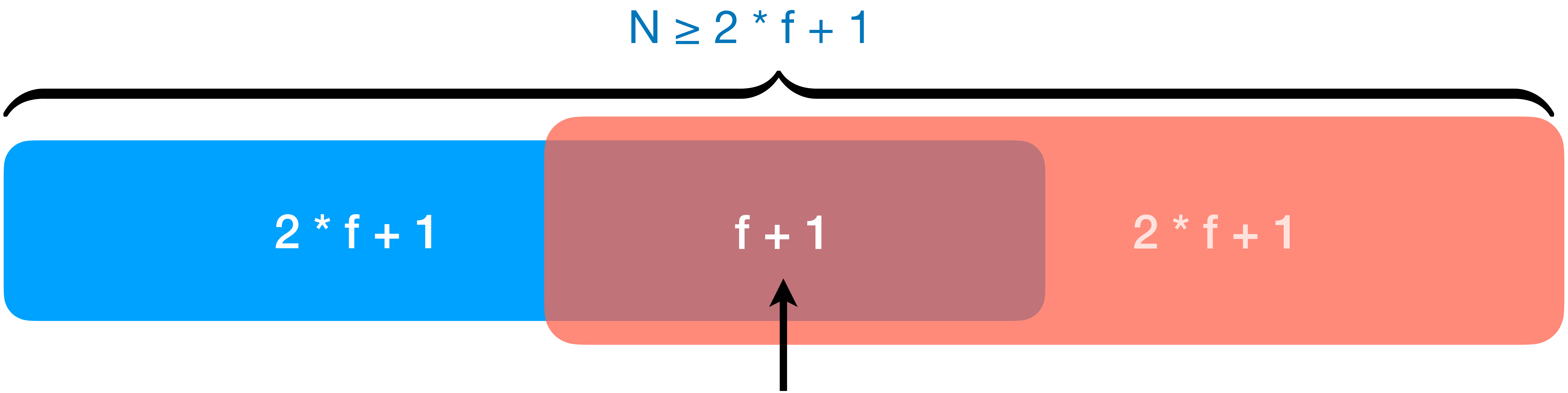
Choosing the Quorum Size

An adversarial node *in the intersection* can “lie” about the value:

to honest parties it might look like *there is not split, but in fact, there is*!

Choosing the Quorum Size

- *Byzantine consensus:* let's make a quorum to be ≥ $2/3 * N + 1$
  any two quorums must have **at least one non-faulty node** in their intersection.



Up to f adversarial nodes *will not manage* to deceive the others.

Two Key Ideas of Byzantine Fault Tolerance

- **3-Phase protocol**: *Pre-prepare, Prepare, Commit*
  - Cross-validating each other's intentions amongst replicas
- **Larger quorum size**: 2/3*N + 1 (instead of N/2 + 1)
  - Allows for up to 1/3 * N adversarial nodes
  - Honest nodes still reach an agreement

Practical Byzantine Fault Tolerance (PBFT)

- Introduced by **Miguel Castro & Barbara Liskov** in 1999
  - almost 10 years after Paxos

- Addresses real-life constraints on Byzantine systems:
  - *Asynchronous* network
  - *Byzantine* failure
  - Message senders *cannot be forged* (via public-key crypto)

PBFT Terminology and Layout

- Replicas — nodes participating in a consensus (no more *acceptor*/*proposer* dichotomy)
- A *dedicated replica* (**primary**) acts as a proposer/leader
  - A primary can be re-elected if suspected to be compromised
  - Backups — other, non-primary replicas
- Clients — communicate directly with primary/replicas
- The protocol uses *time-outs* (partial synchrony) to *detect faults*
  - *E.g.*, a primary not responding *for too long is considered compromised*

Overview of the Core PBFT Algorithm

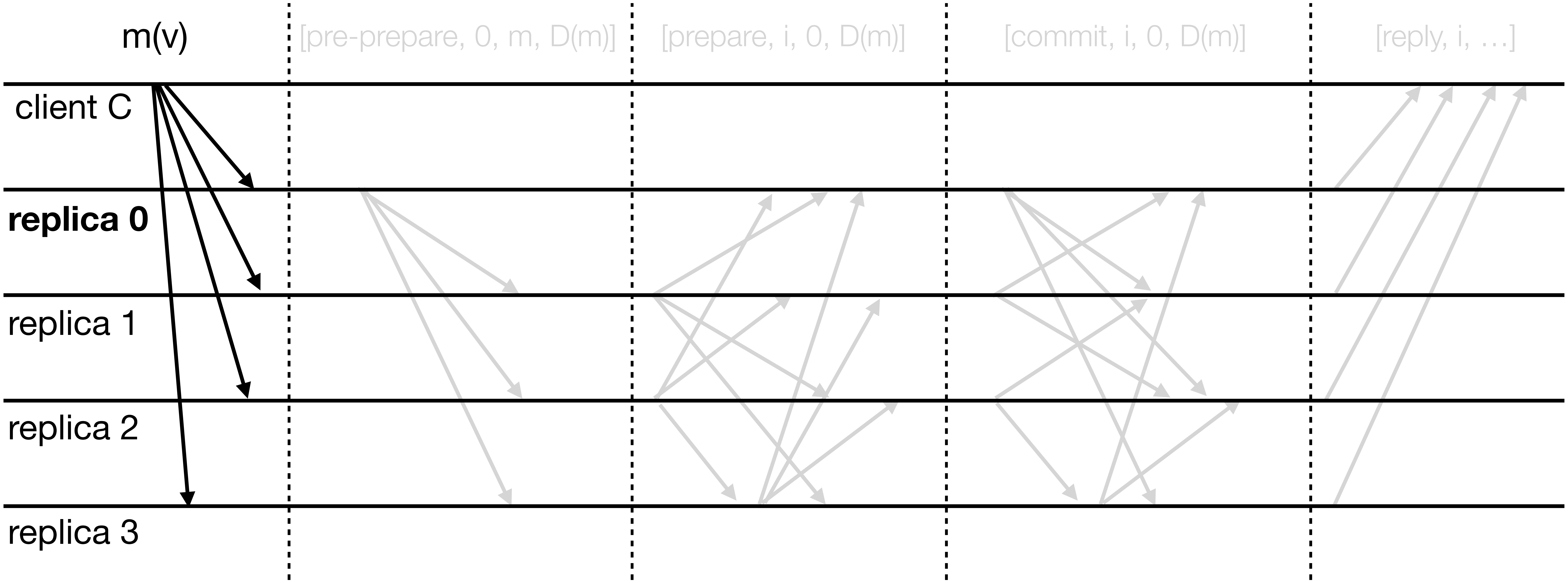
Request → **Pre-Prepare → Prepare → Commit** → Reply

Executed by Client

Executed by Replicas

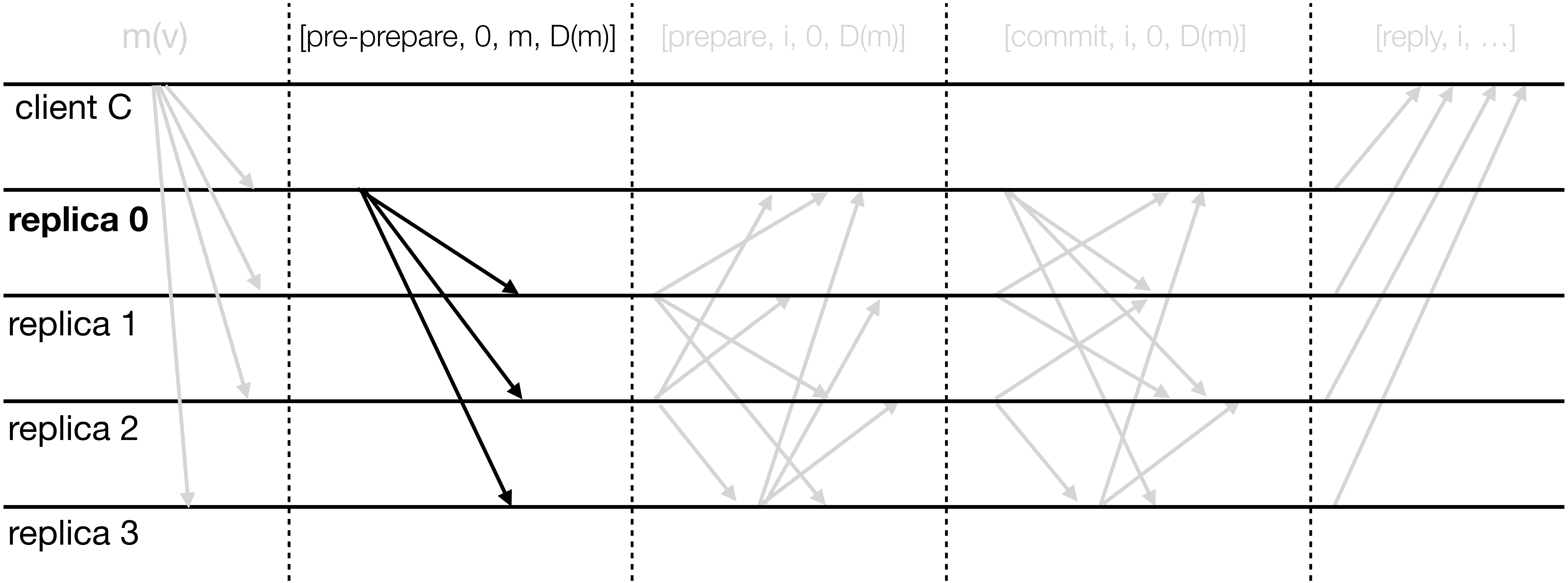
Request

Client C sends a message to *all* replicas



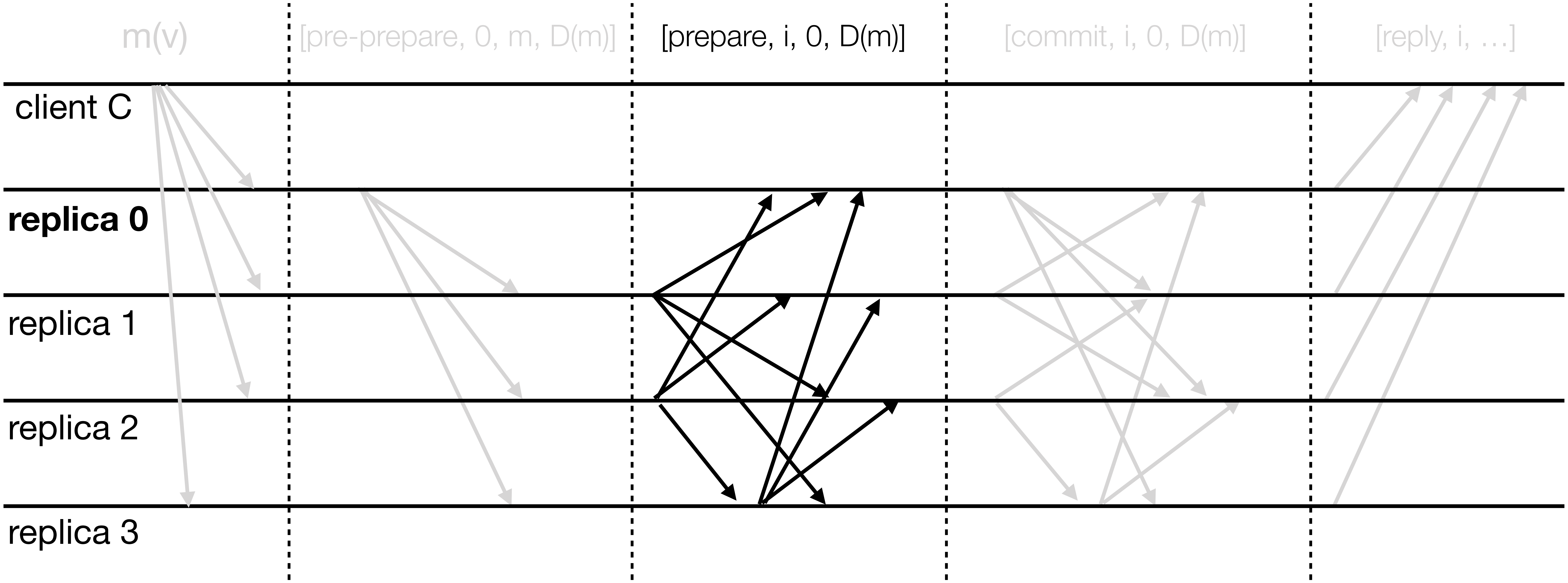
Pre-prepare

- Primary (0) sends a signed pre-prepare message with the to *all backups*
  - It also includes the *digest (hash)* D(m) of the original message



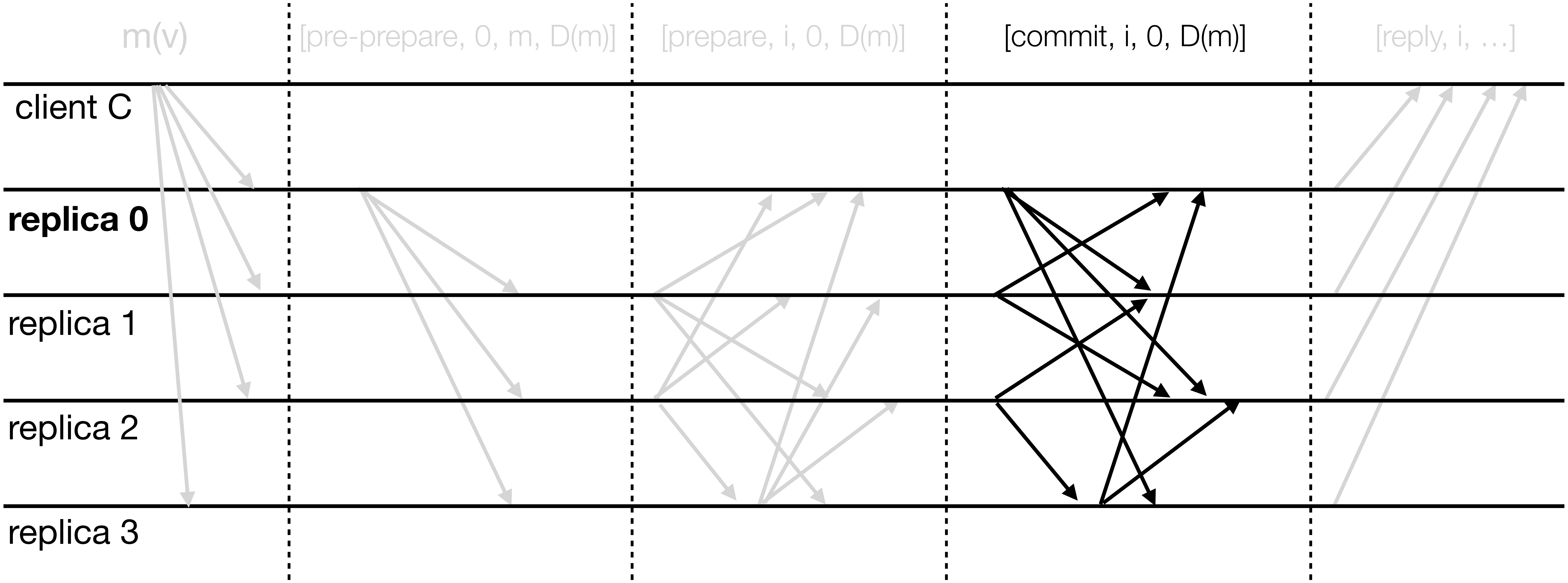
Prepare

- Each replica sends a prepare-message to all other replicas
- It proceeds if it receives 2/3*N + 1 prepare-messages *consistent* with its own



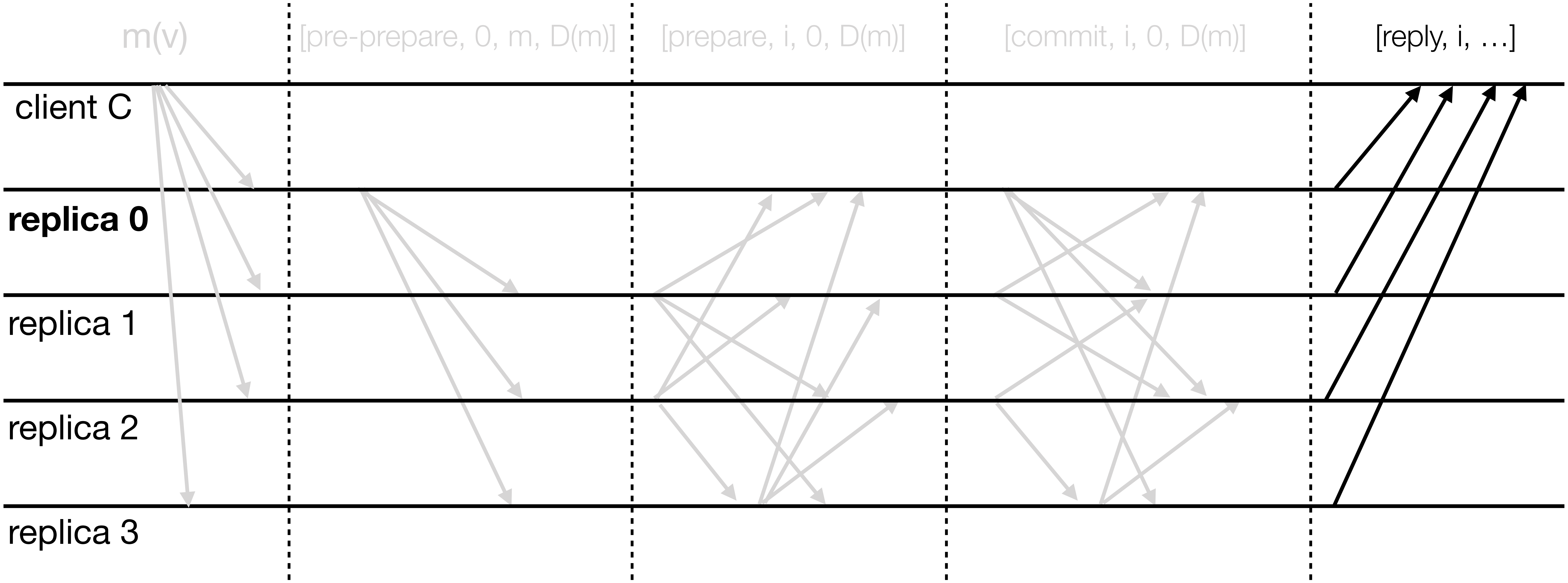
Commit

- Each replica sends a signed commit-message to all other replicas
- It commits if it receives 2/3*N+1 commit-messages *consistent* with its own



Reply

- Each replica sends a signed response to the initial client
- The client trusts the response once she receives N/3 + 1 matching ones



What if Primary is compromised?

- Thanks to large quorums, it *won't break integrity* of the good replicas
- Eventually, replicas and the clients will detect it *via time-outs*
  - Primary sending inconsistent messages would cause the system to *"get stuck"* between the phases, without reaching the end of **commit**
- Once a faulty primary is detected, backups-will launch a ***view-change***, *re-electing a new primary*
- View-change is *similar to reaching a consensus* but gets tricky in the presence of partially committed values
  - See the *Castro & Liskov '99 PBFT* paper for the details…

PBFT in Industry

- Widely adopted in practical developments:
  - Tendermint
  - IBM's Openchain
  - Zilliqa
  - Libra/Novi
  - Solana
- Used for implementing *to speed-up* blockchain-based consensus
- Many blockchain solutions build on similar ideas
  - Stellar Consensus Protocol, HotStuff

PBFT Shortcomings

- Can be used only for a *fixed* set of replicas
  - Agreement is based on *fixed-size quorums*
- *Open* systems (used in Blockchain Protocols) rely on alternative mechanisms of **Proof-of-X** (e.g., Proof-of-Work, Proof-of-Stake)

Blockchain Consensus Protocols

What blockchain does

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

blockchain consensus protocol

transactions can be *anything*

- transforms a **set** of transactions into a *globally-agreed* **sequence**
- "distributed timestamp server" (Nakamoto 2008)

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$\downarrow$$

$$[tx_5, tx_3] \to [tx_4] \to [tx_1, tx_2]$$

$$\downarrow$$

$$tx_5 \to tx_3 \to tx_4 \to tx_1 \to tx_2$$

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$[tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$$

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

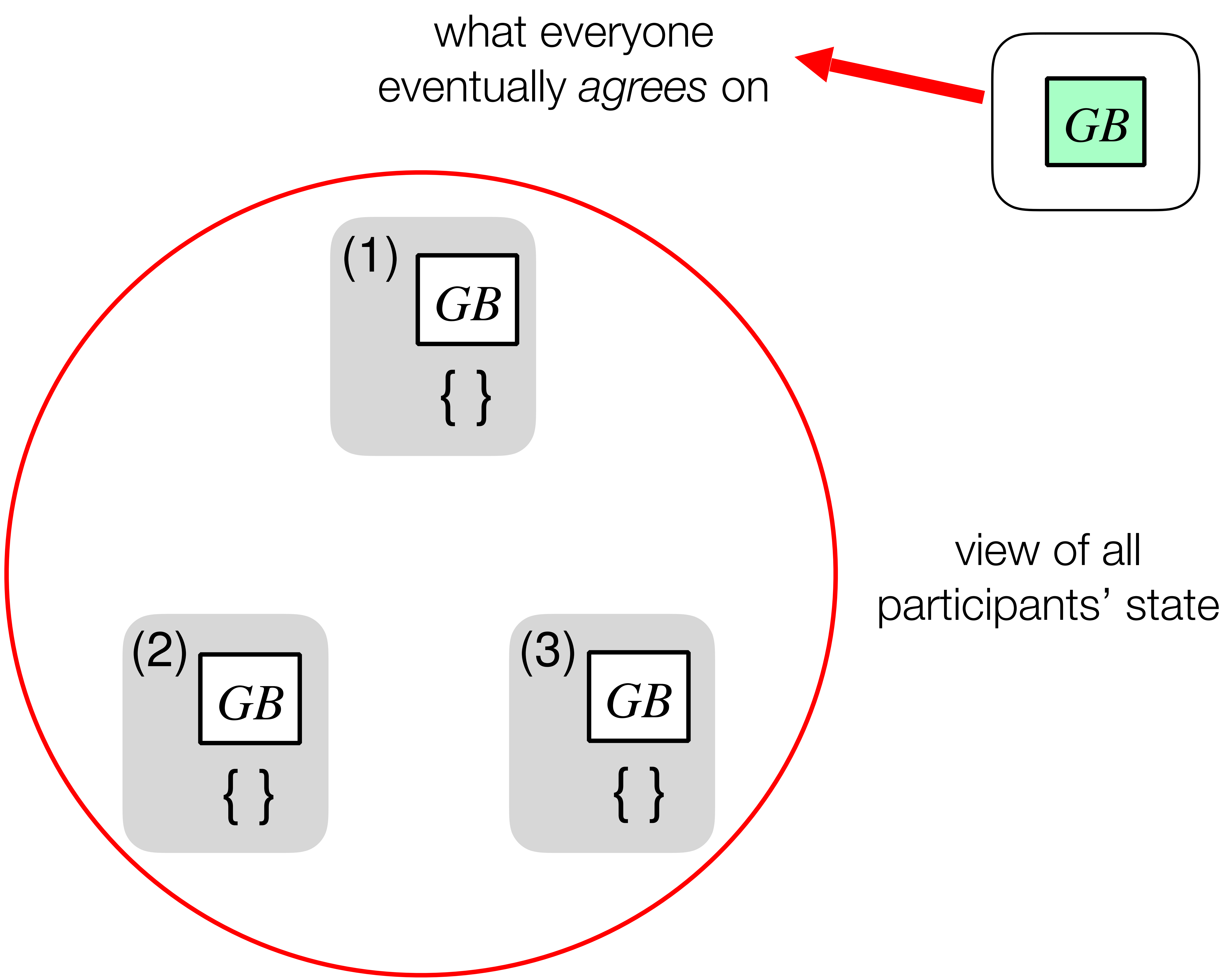
$$[] \leftarrow [tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$$

GB = genesis block

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

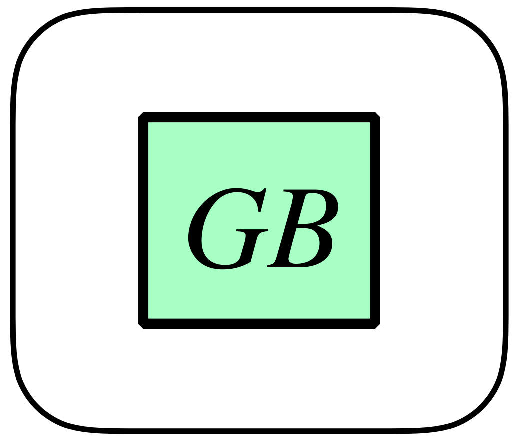
How blockchain protocols work

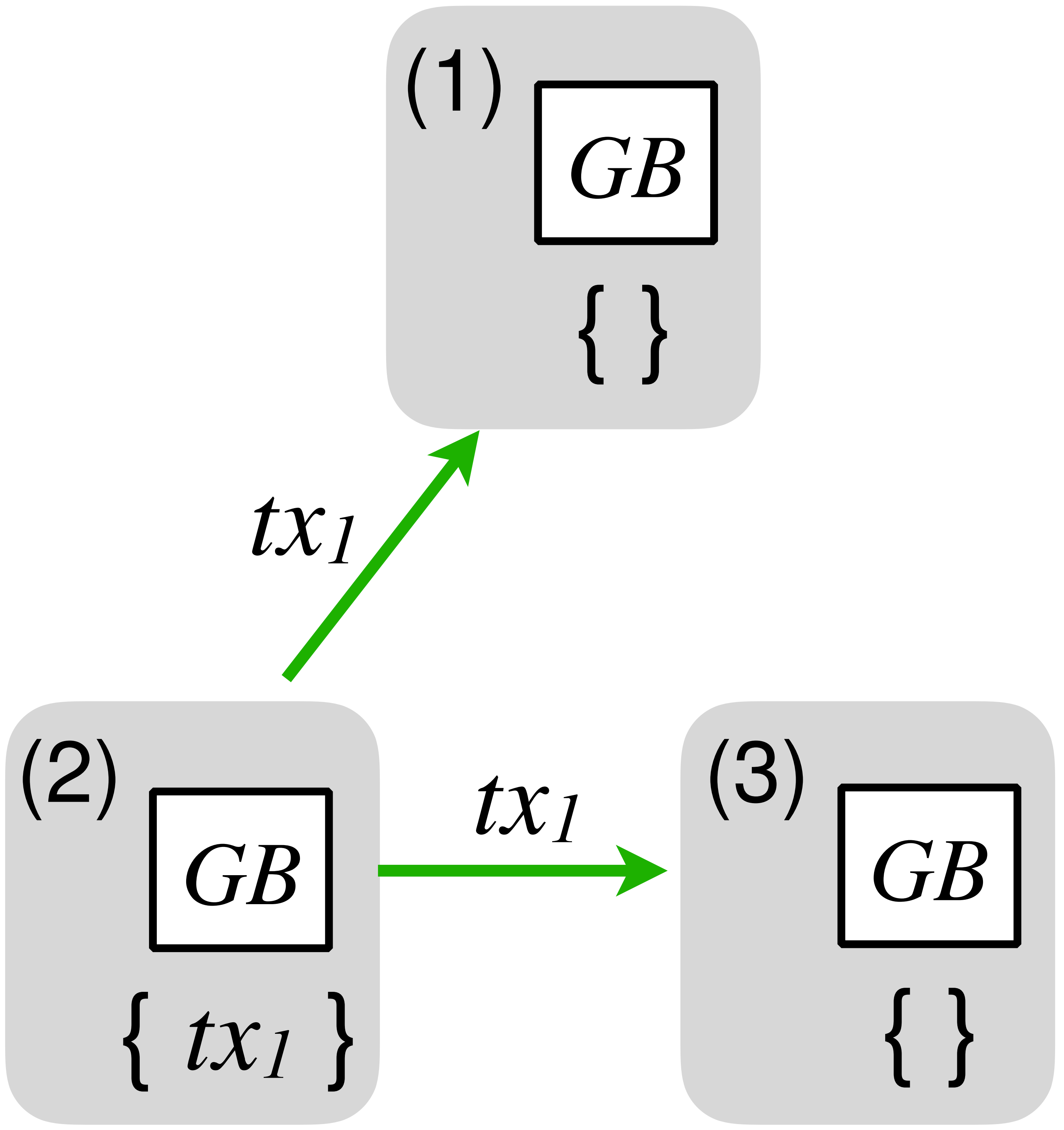
- **distributed**
  - multiple *nodes*
- all start with same GB



- **distributed**
  - multiple nodes
  - *message-passing* over a network

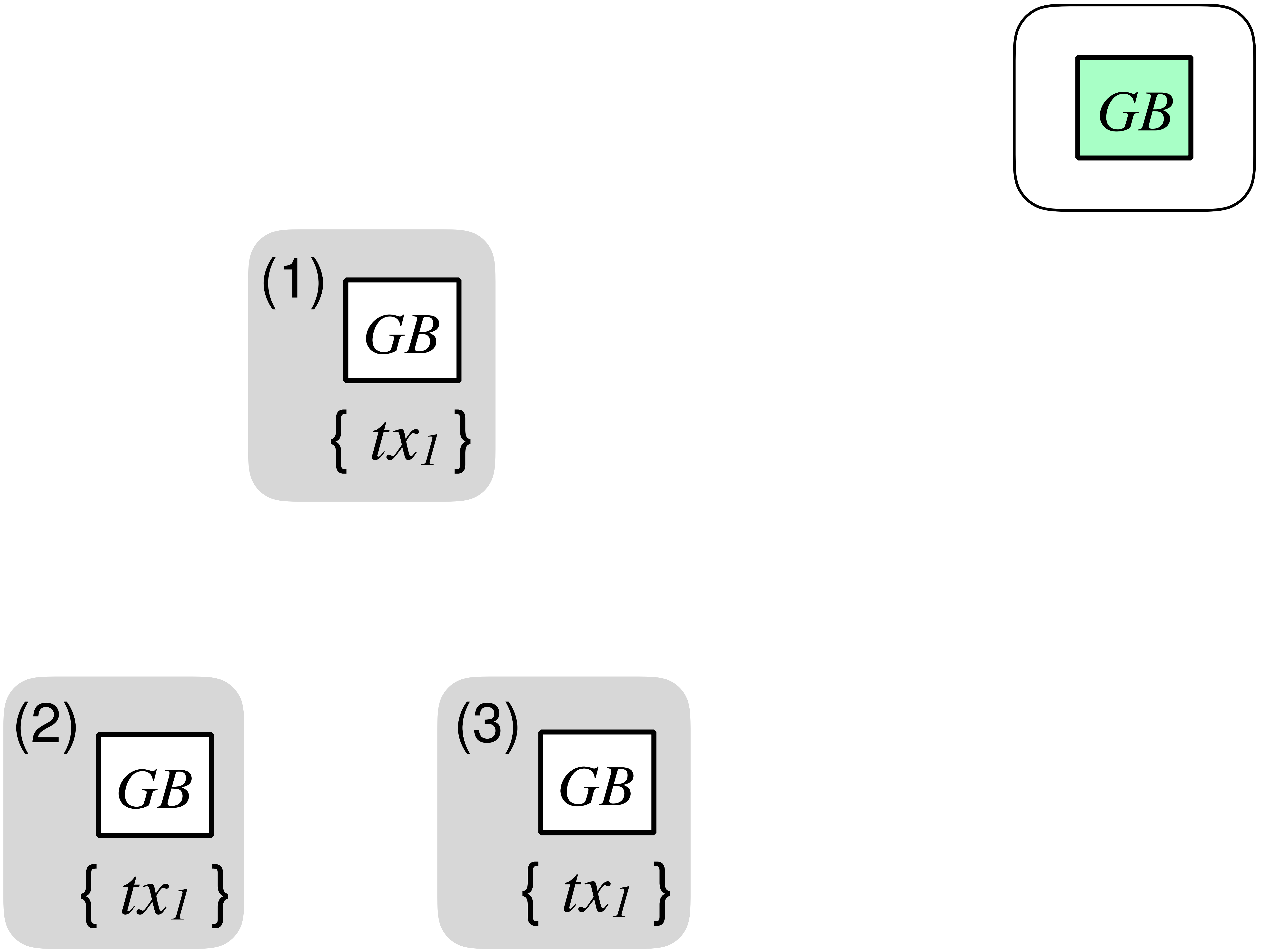
- all start with same GB



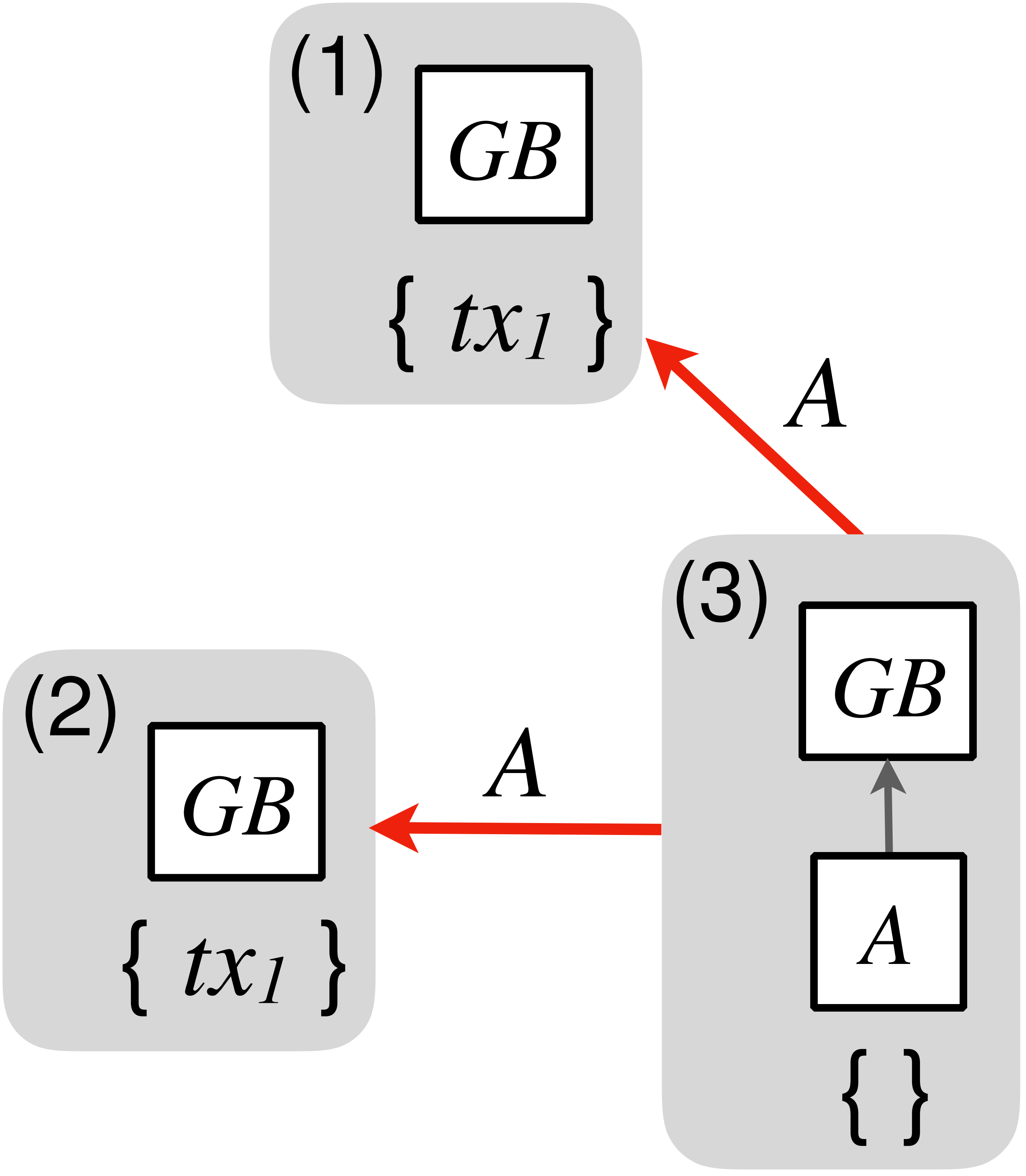


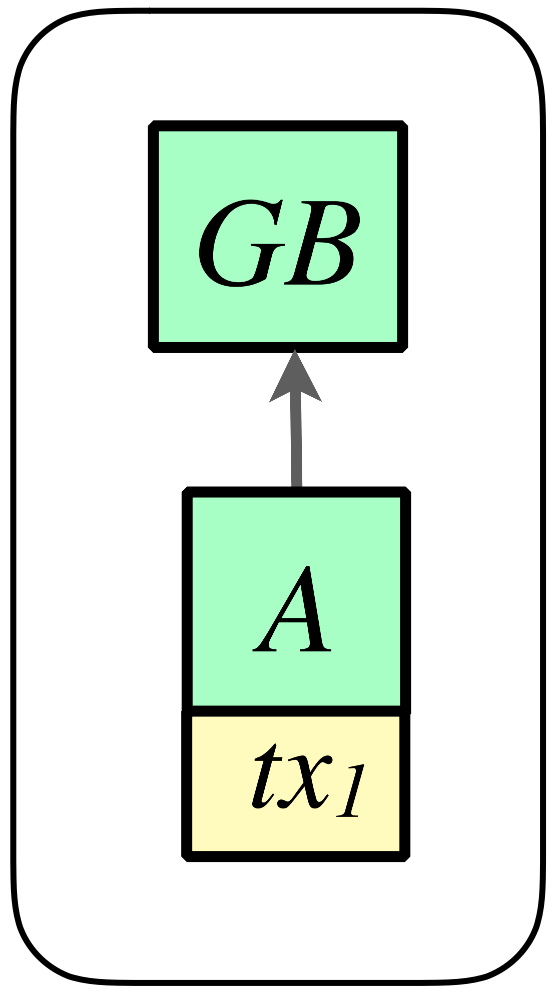
- **distributed**
  - multiple nodes
  - message-passing over a network

- all start with same GB
- have a transaction pool



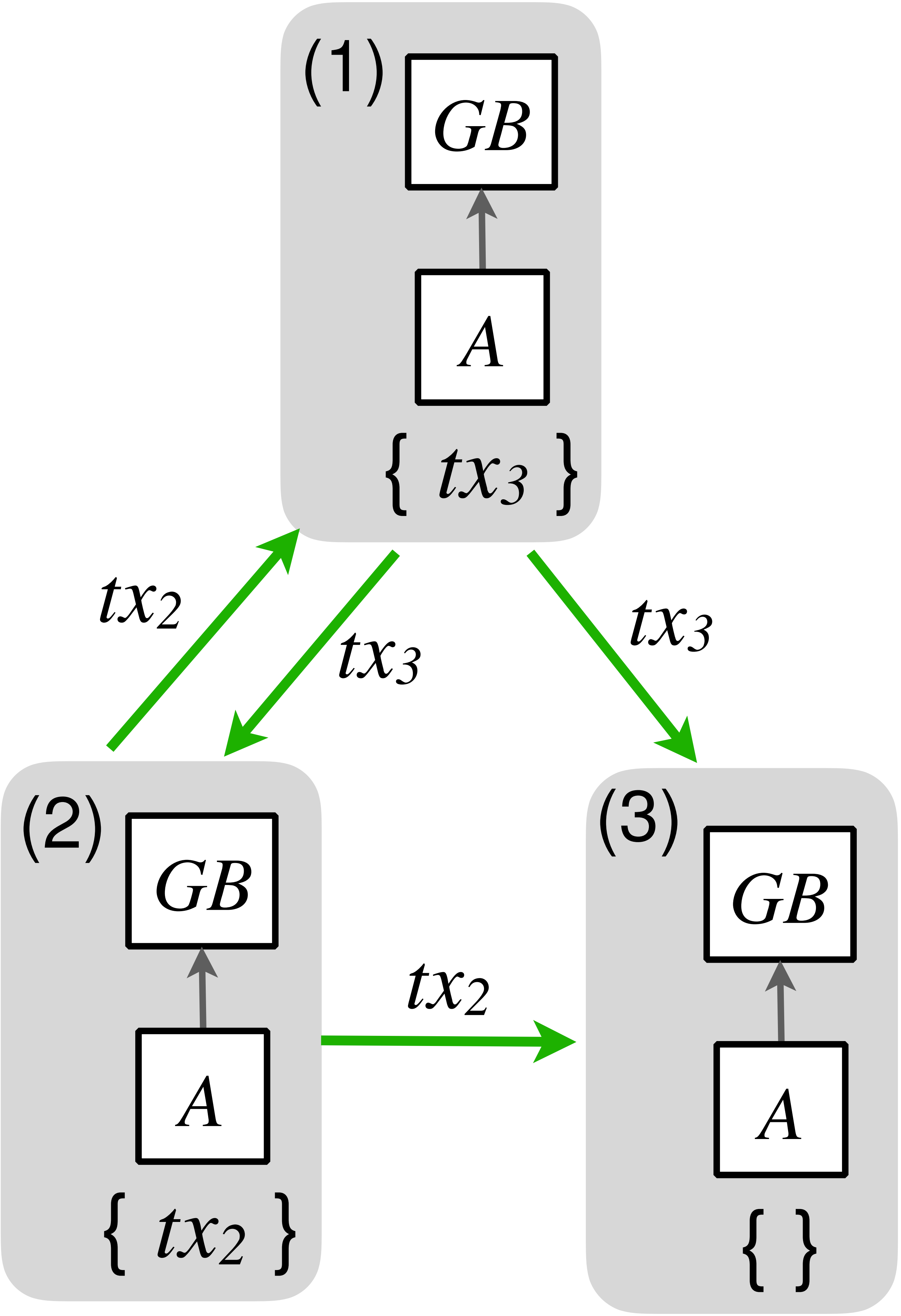
- **distributed**
  - multiple nodes
  - message-passing over a network
- all start with same GB
- have a transaction pool
- can *create (mint) blocks*

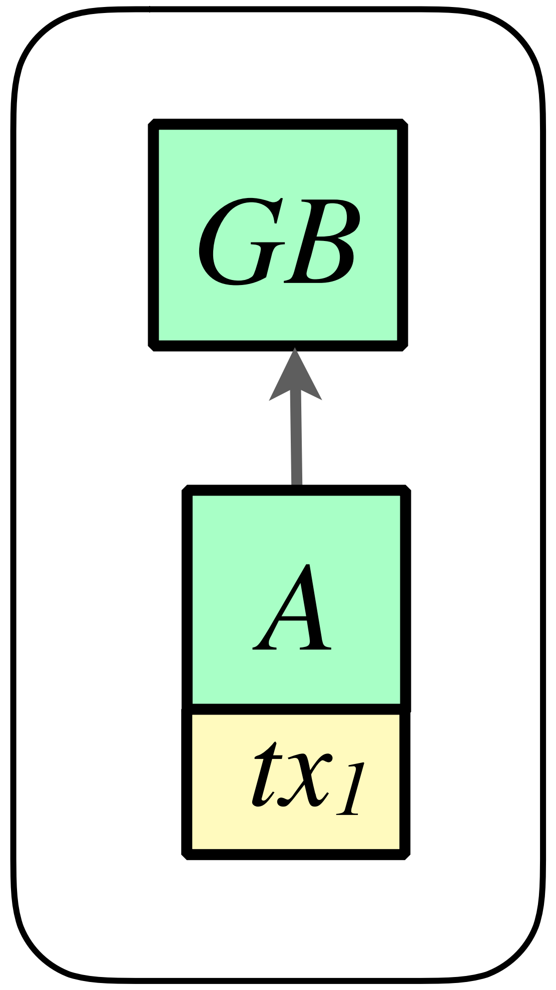




- **distributed** ⇒ *concurrent*
  - multiple nodes
  - message-passing over a network

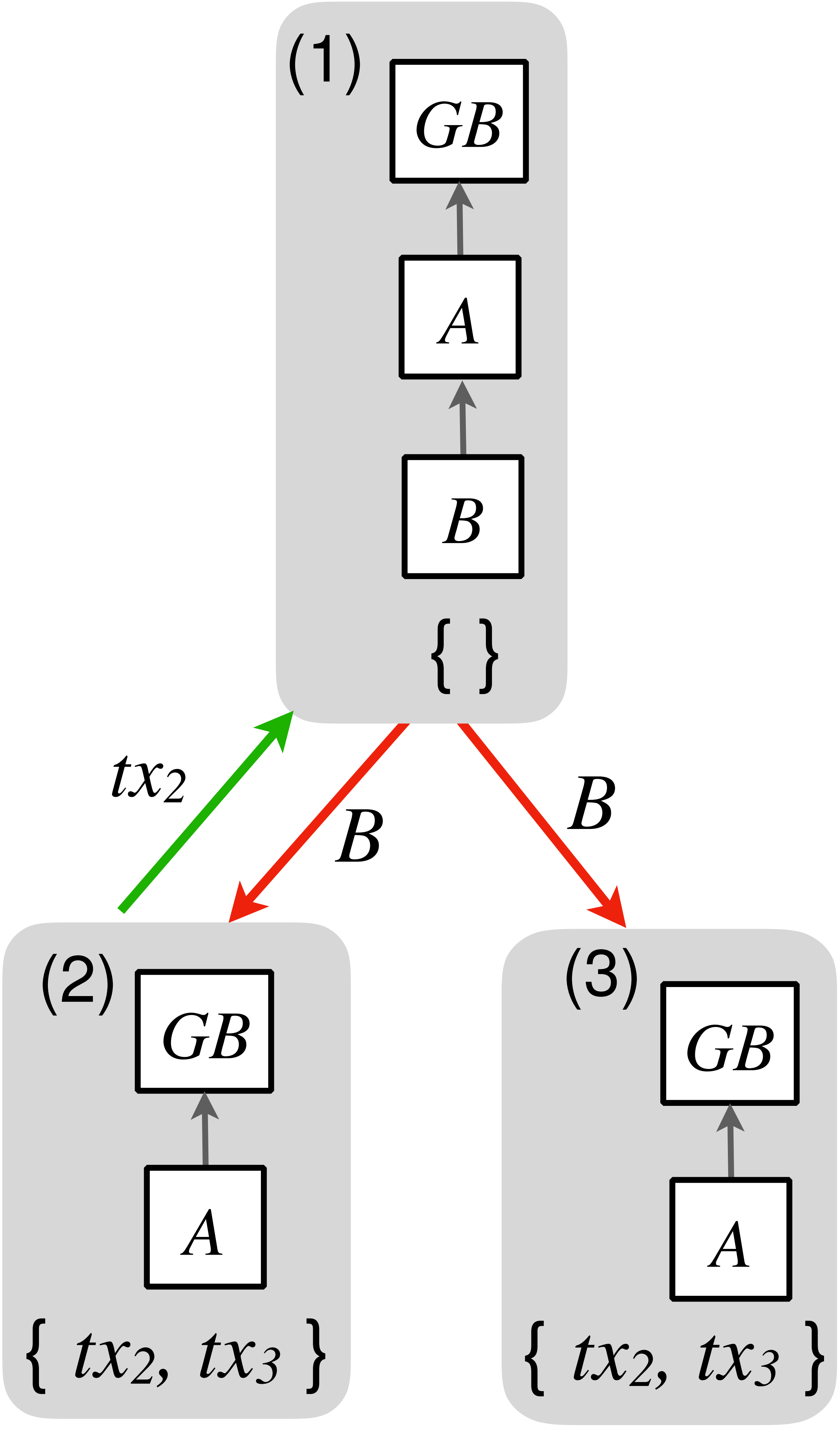
- multiple transactions can be issued and propagated *concurrently*

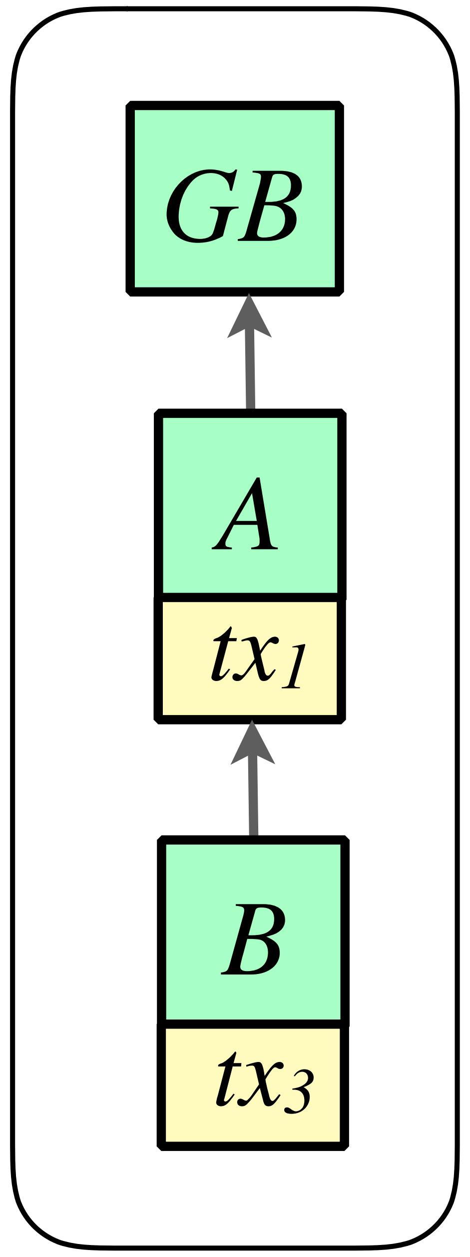




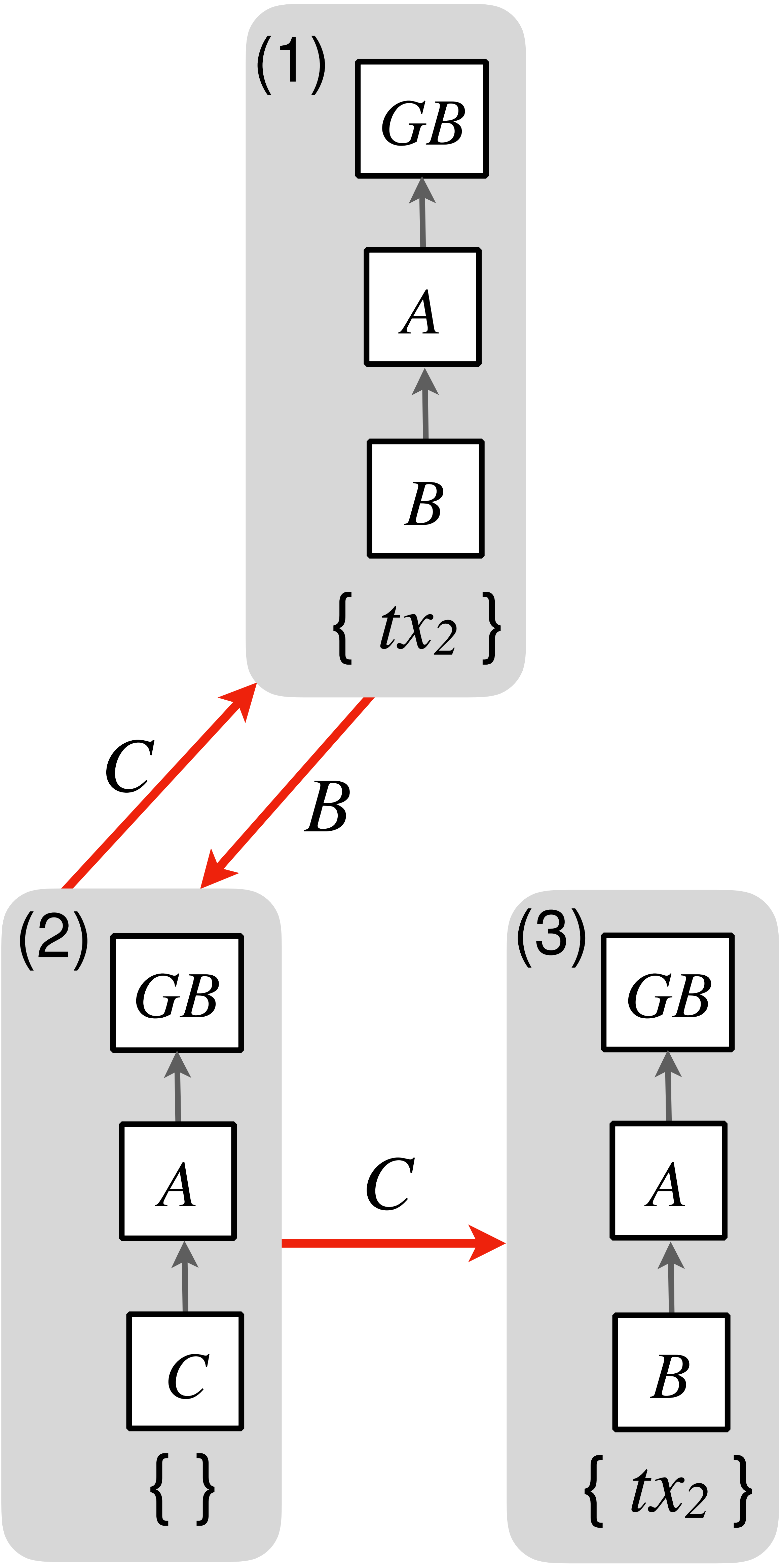
- **distributed** ⇒ *concurrent*
  - multiple nodes
  - message-passing over a network

- blocks can be created *without full knowledge* of all transactions

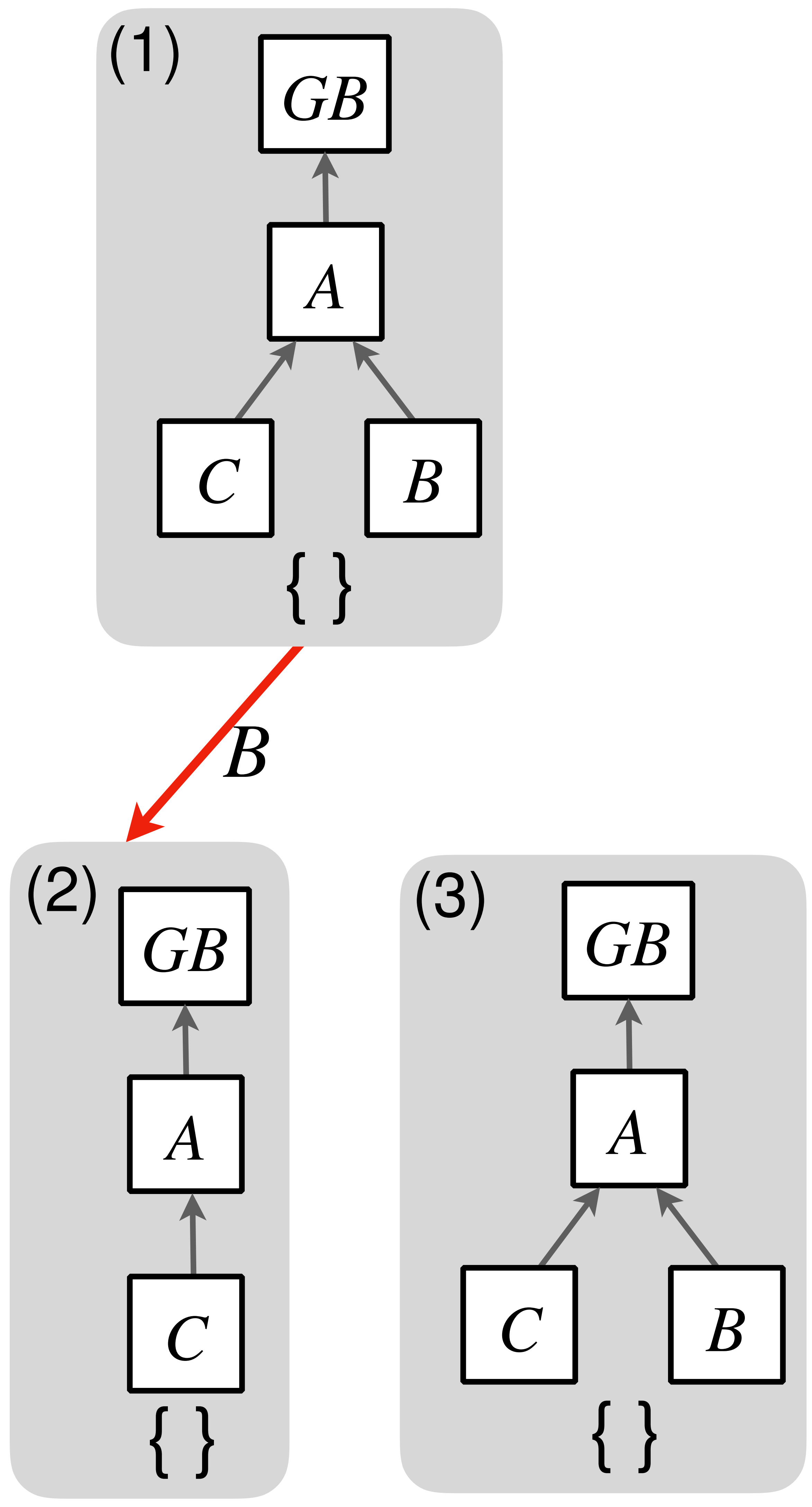




- *chain fork* has happened, but nodes don't know about it

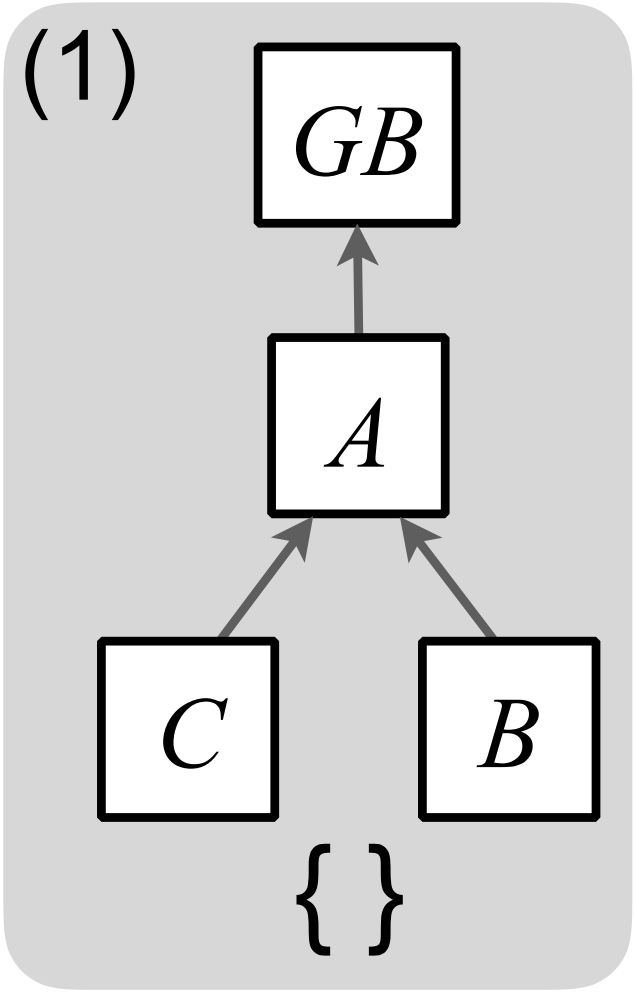


- as block messages propagate, nodes become aware of the *fork*

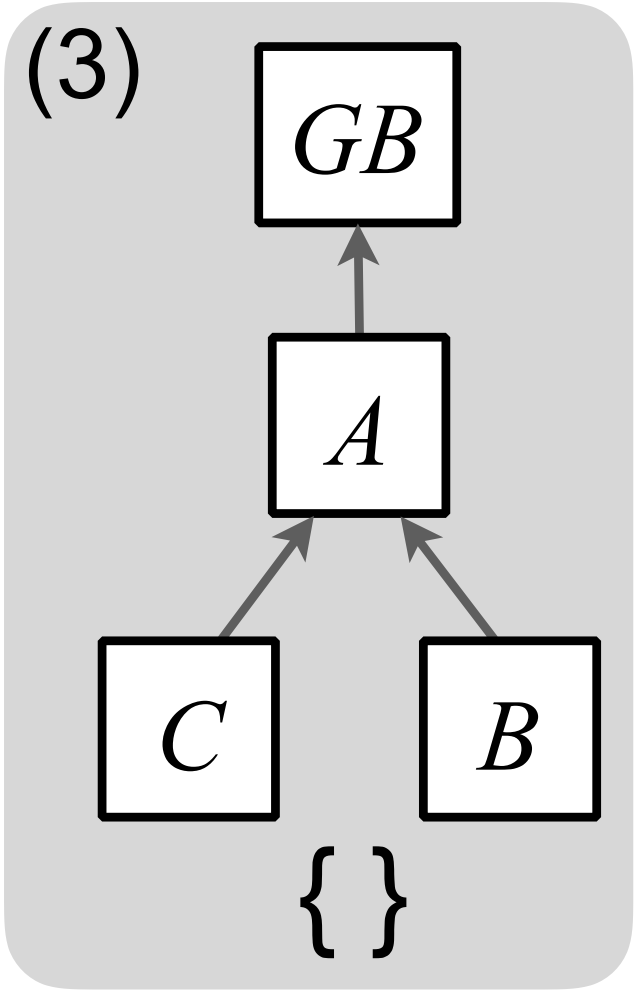


Problem: need to choose

- blockchain "promise" = *one globally-agreed chain*

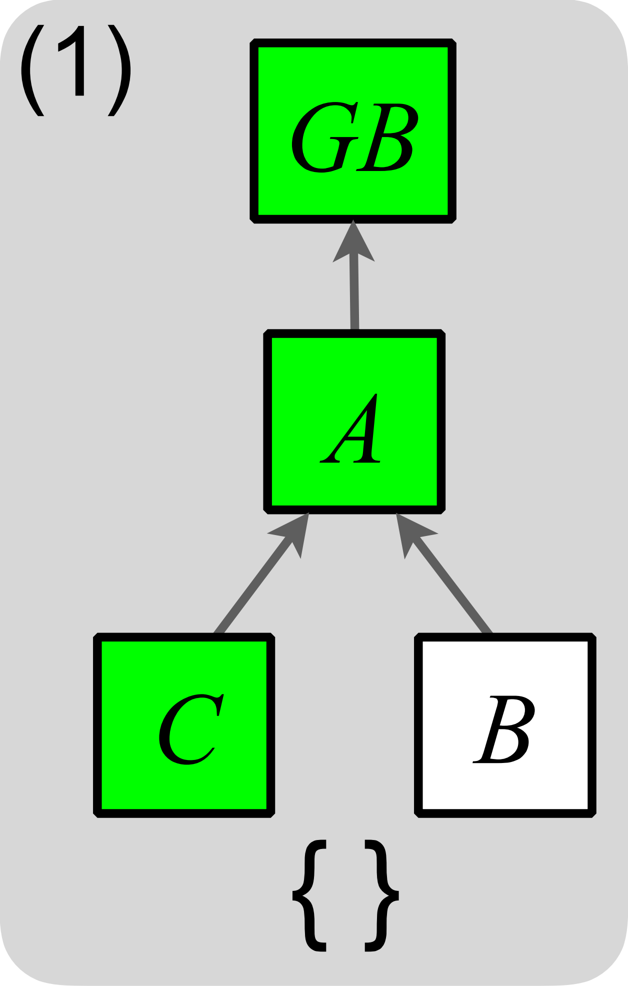


  - each node must choose *one* chain
  - nodes with the same information must choose *the same* chain

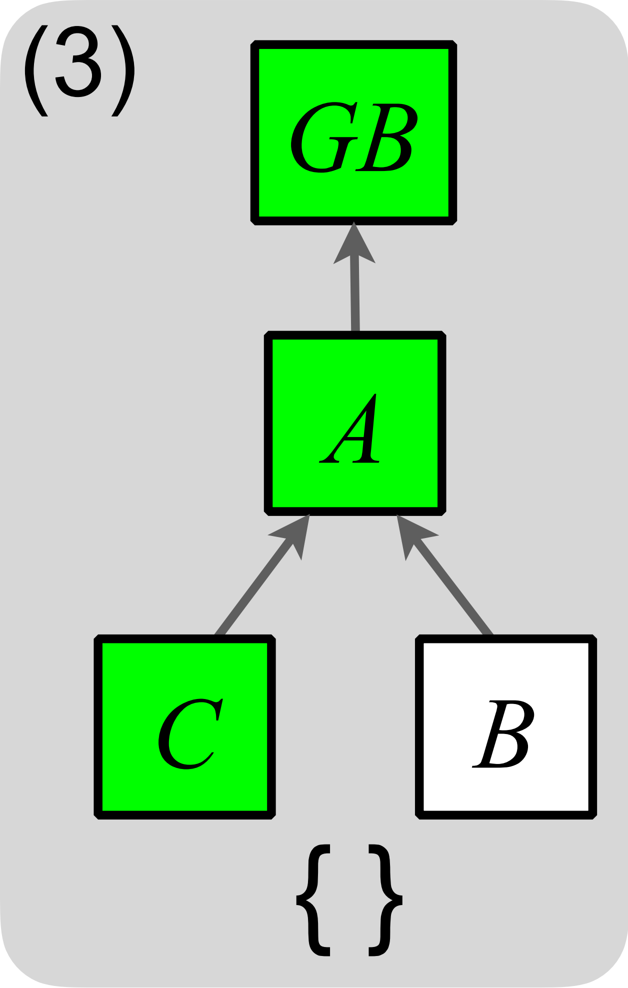


Problem: need to choose

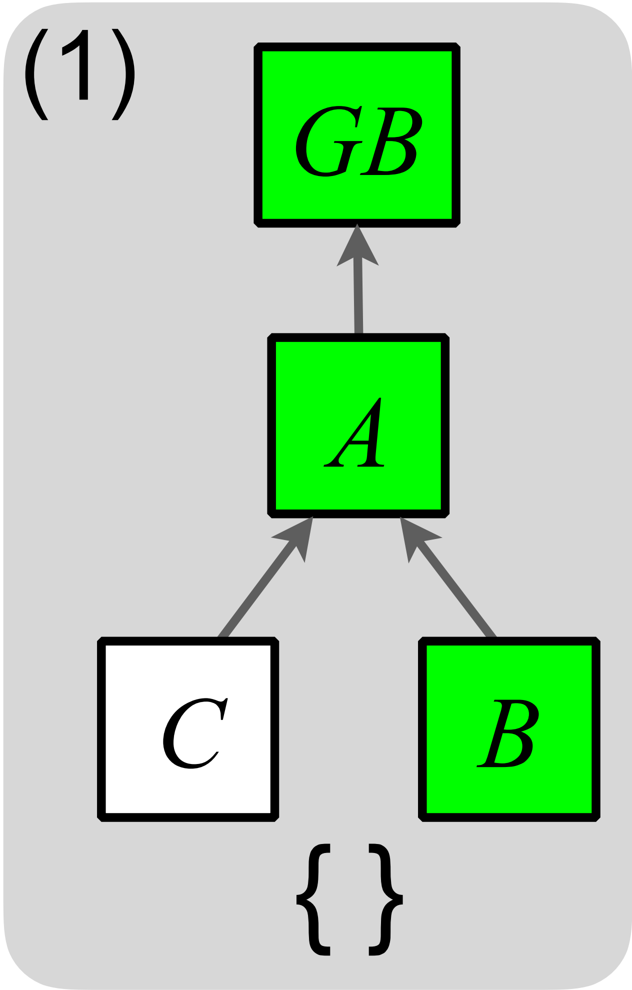
- blockchain “promise” = *one globally-agreed chain*



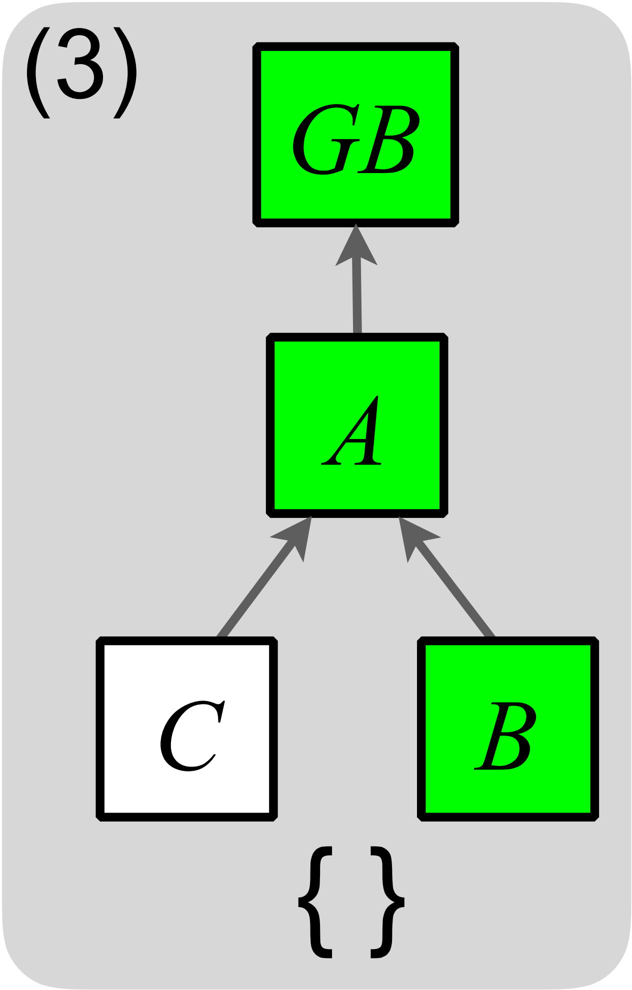
  - each node must choose *one* chain
  - nodes with the same information must choose *the same* chain



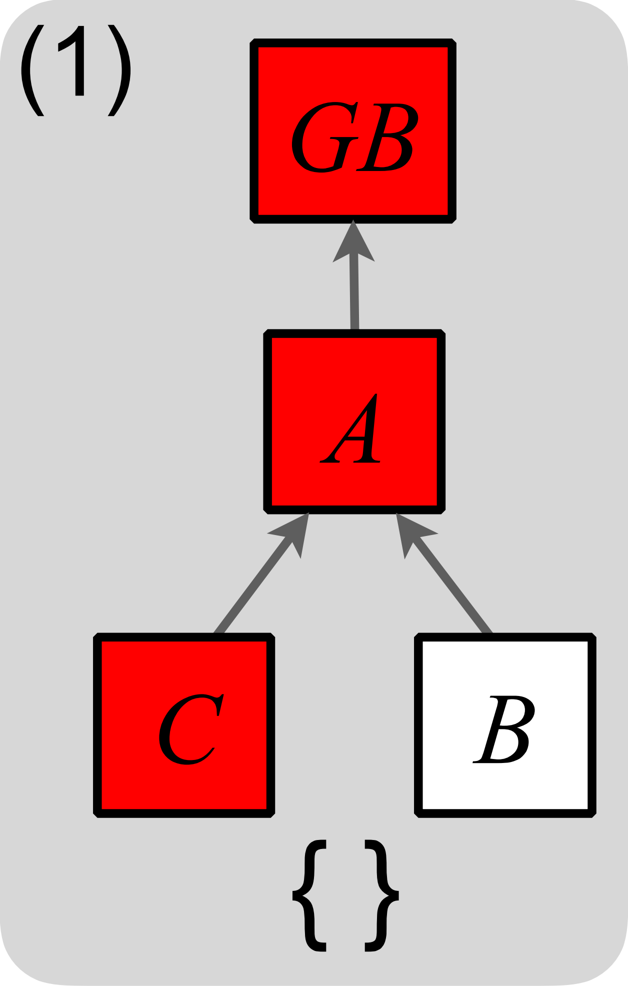
Problem: need to choose



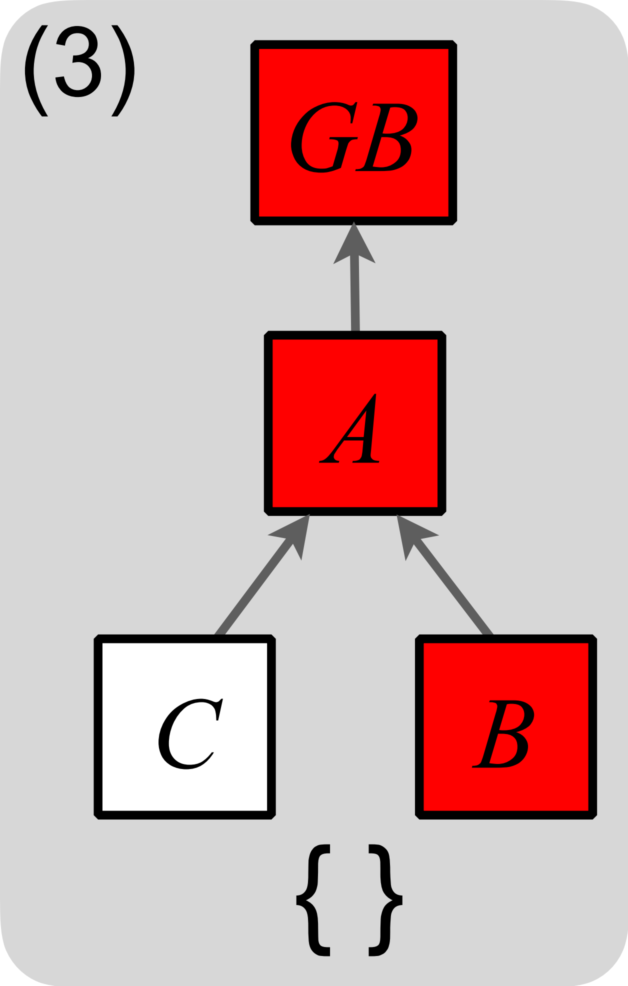
- blockchain "promise" = *one globally-agreed chain*
  - each node must choose *one* chain
  - nodes with the same information must choose *the same* chain



Problem: need to choose



- blockchain "promise" = *one globally-agreed chain*
  - each node must choose *one* chain
  - nodes with the same information must choose *the same* chain



Solution: *fork choice rule*

- **Fork choice rule** (FCR, >):
  - given two blockchains, says which one is "*heavier*"
  - imposes a *strict total order* on all possible blockchains
  - same FCR *shared* by all nodes

- Nodes adopt "heaviest" chain they know

- "Lying" to different nodes is computationally *very expensive* and *cannot* be done for *multiple subsequent blocks*

FCR (>)

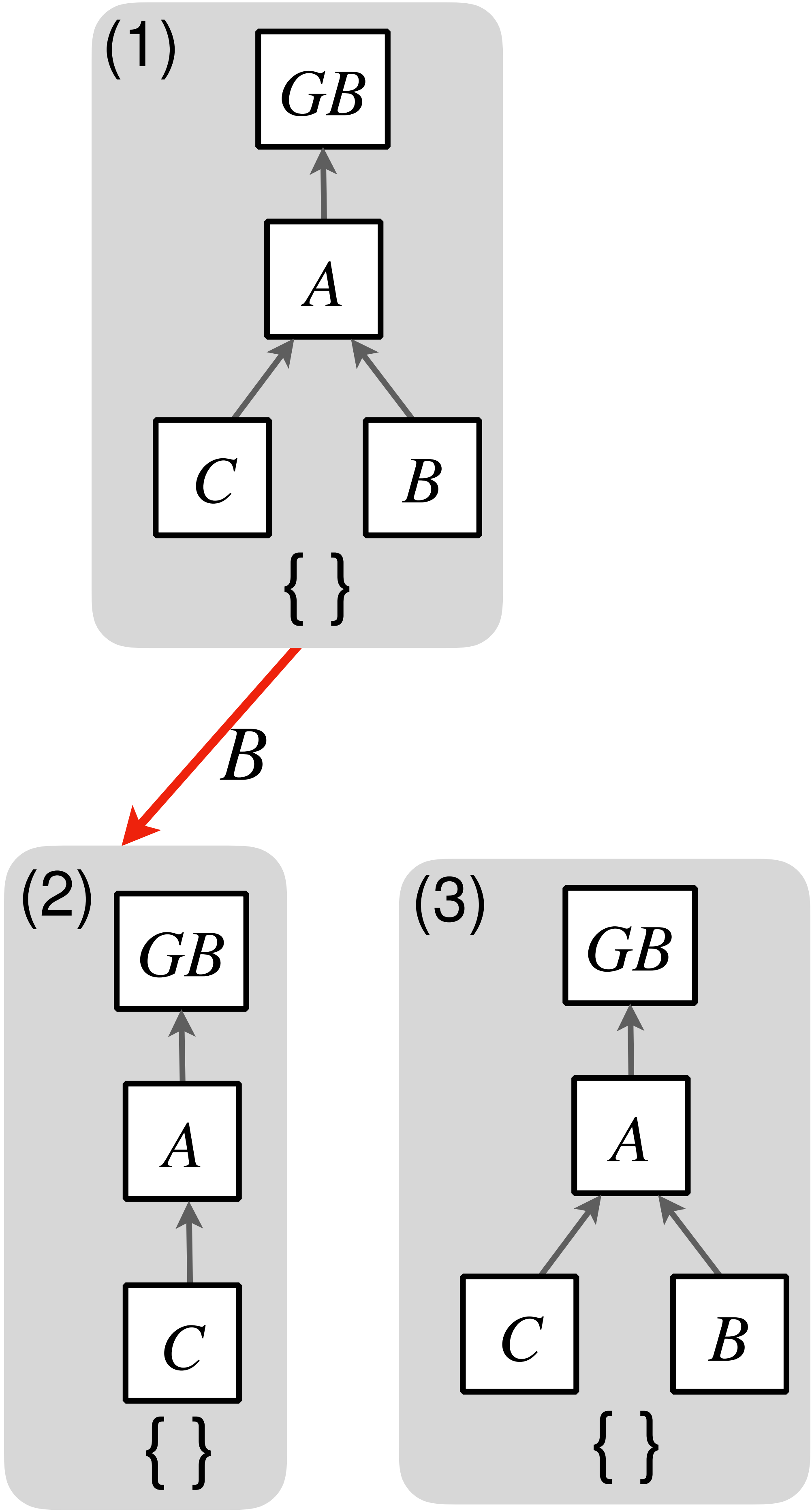
… > [GB, A, C] > … > [GB, A, B] > … > [GB, A] > … > [GB] > …

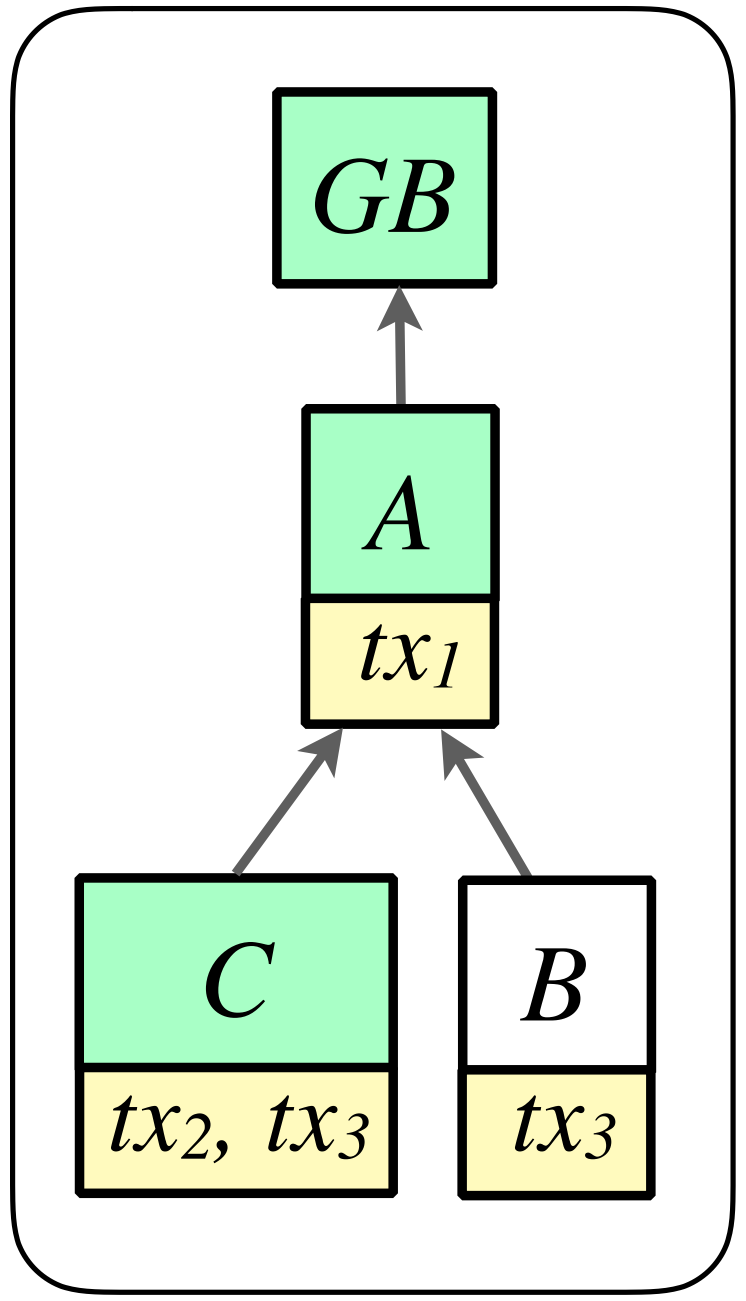
Bitcoin: FCR based on “most cumulative work”.

New blocks take *a lot of time and CPU* to create.

Quiescent consistency

- **distributed**
  - multiple nodes
  - all start with GB
  - message-passing over a network
  - *equipped with same FCR*

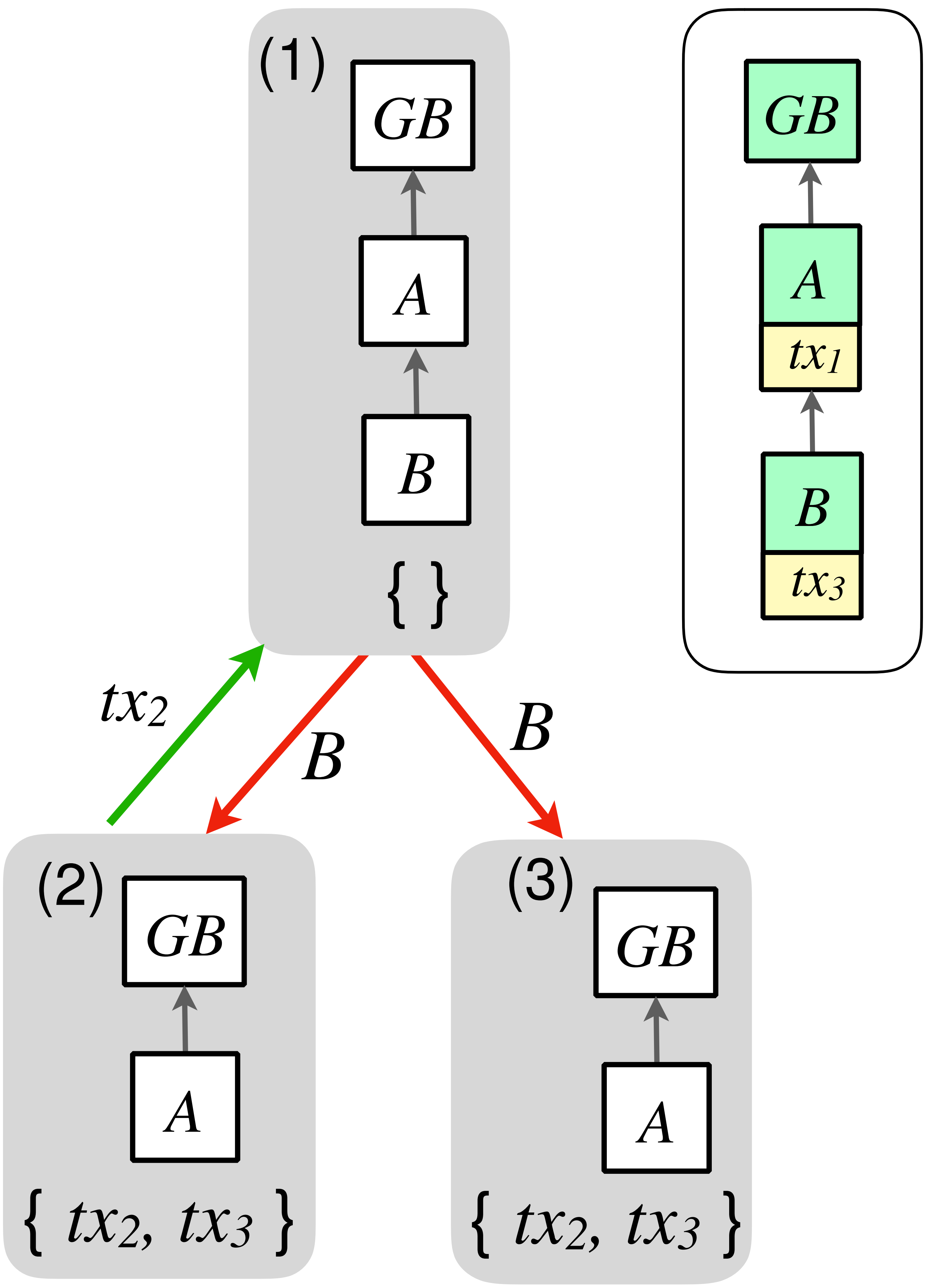




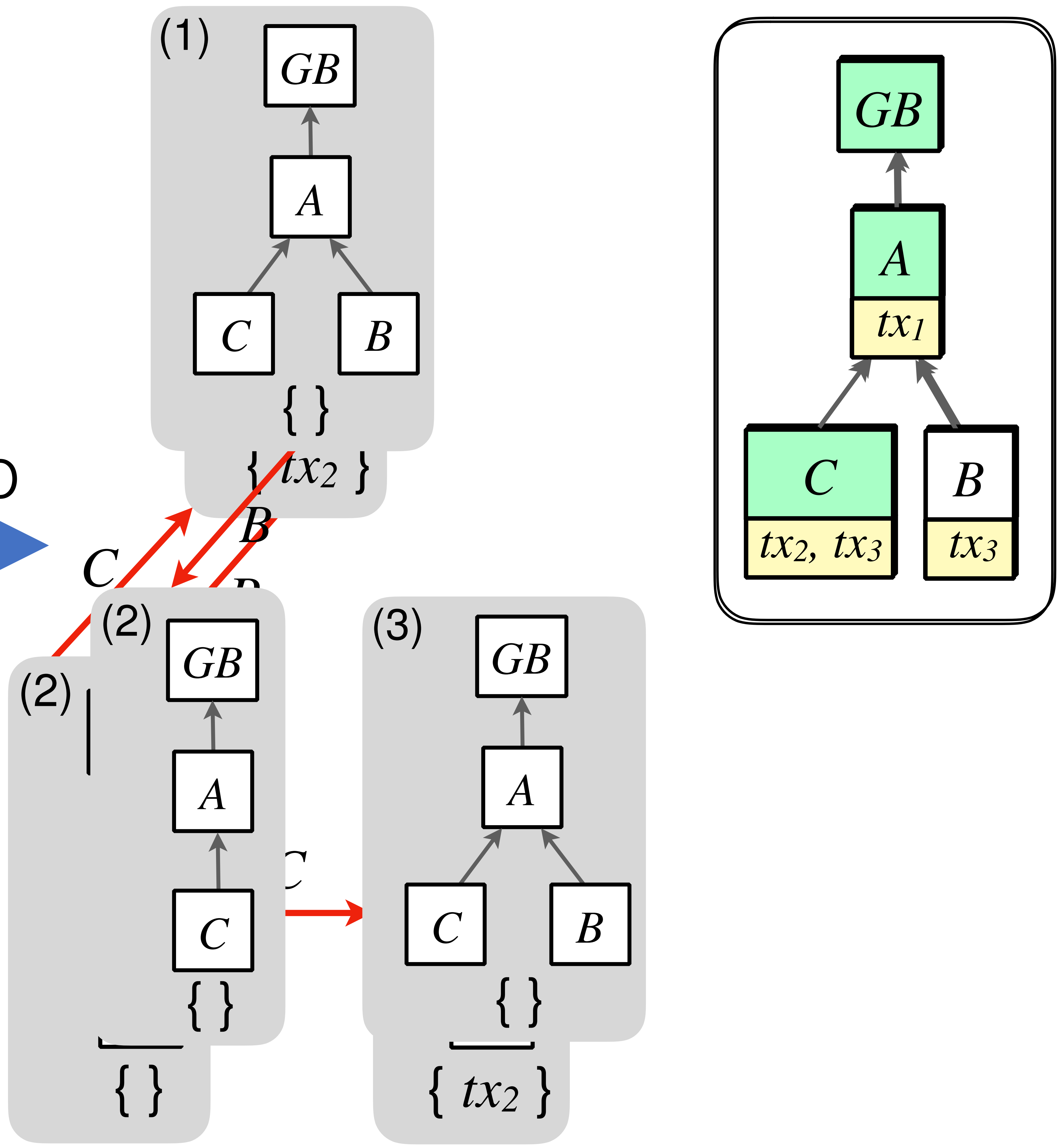
- *Quiescent Consistency*:
  when *all* block messages have been delivered, *everyone (good) agrees*

Why it works

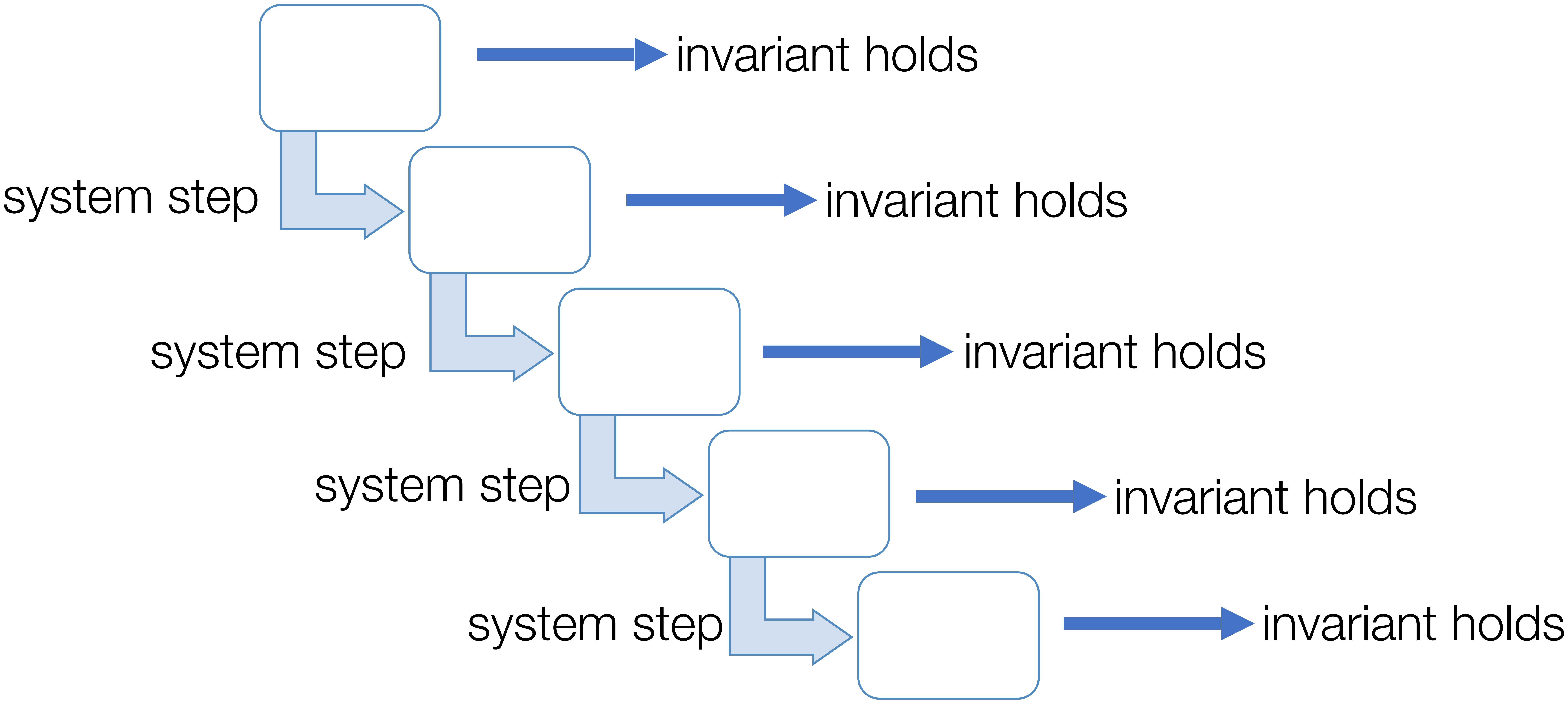
Invariant: local state + "in-flight" = global



global system step



Invariant is inductive



Invariant implies Quiescent Consistency

- QC: when all blocks *delivered*, everyone *agrees*

How:

- local state + ~~“in-flight”~~ = global
- use FCR to extract “heaviest” chain out of local state
- since everyone has *same initial state* & *same FCR*
  - ➢consensus

Mechanising Blockchain Consensus

George Pîrlea
University College London, UK
george.pirlea.15@ucl.ac.uk

Ilya Sergey
University College London, UK
i.sergey@ucl.ac.uk

Abstract

We present the first formalisation of a blockchain-based distributed consensus protocol with a proof of its consistency mechanised in an interactive proof assistant.

Our development includes a reference mechanisation of the *block forest* data structure, necessary for implementing provably correct per-node protocol logic. We also define a model of a network, implementing the protocol in the form of a replicated state-transition system. The protocol's executions are modeled via a small-step operational semantics for asynchronous message passing, in which packages can be rearranged or duplicated.

In this work, we focus on the notion of global system safety, proving a form of eventual consistency. To do so, we provide a library of theorems about a pure functional implementation of block forests, define an inductive system invariant, and show that, in a quiescent system state, it implies a global agreement on the state of per-node transaction ledgers. Our development is parametric *wrt.* implementations of several security primitives, such as *hash*-functions, a notion of a *proof object*, a *Validator Acceptance Function*, and a *Fork Choice Rule*. We precisely characterise the assumptions, made about these components for proving the global system consensus, and discuss their adequacy. All results described in this paper are formalised in Coq.

1 Introduction

The notion of decentralised blockchain-based consensus is a tremendous success of the modern science of distributed computing, made possible by the use of basic cryptography, and enabling many applications, including but not limited to cryptocurrencies, smart contracts, application-specific arbitration, voting, *etc.*

In a nutshell, the idea of a distributed consensus protocol based on *blockchains*, or *transaction ledgers*,[1] is rather simple. In all such protocols, a number of stateful nodes (participants) are communicating with each other in an asynchronous message-passing style. In a message, a node (a) can announce a *transaction*, which typically represents a certain event in the system, depending on the previous state of the node or the entire network (we intentionally leave out the details of what can go into a transaction, as they are application-specific); a node can also (b) create and broadcast a *block* that contains the encoding of a certain vector of transactions, created locally or received via messages of type (a) from other nodes. Each recipient of a block message should then *validate* the block (*i.e.*, check the consistency of the transaction sequence included in it), and, in some cases, append it to its local ledger, thus, extending its subjective view of the global sequence of transactions that have taken place in the system to date. The process continues as more

Blockchain Transactions

$$[] \leftarrow [tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$$

- Executed by each node *locally*, alter the *replicated* state.
- Simplest variant: *transferring funds* from ***A*** to ***B***,
  consensus: *no* double spending.
- More interesting: deploying and executing *replicated computations*

  Smart Contracts

Smart Contracts

- *Stateful mutable* objects replicated via a consensus protocol
- State typically involves a stored amount of *funds/currency*
- One or more entry points: invoked *reactively* by a client *message*
- Main usages:
  - crowdfunding and ICO
  - multi-party accounting
  - voting and arbitration
  - puzzle-solving games with distribution of rewards
- Supporting platforms: Ethereum, Solana, Sui, Avalanche, Cardano,…

Smart Contracts are Like Concurrent Objects

```
contract Accounting {
  /* Define contract fields */
  address owner;
  mapping (address => uint) assets;

  /* This runs when the contract is executed */
  function Accounting(address _owner) {
    owner = _owner;
  }

  /* Sending funds to a contract */
  function invest() returns (string) {
    if (assets[msg.sender].initialized()) { throw; }
    assets[msg.sender] = msg.value;
    return "You have given us your money";
  }
}
```

Mutable fields

Constructor

Entry point

- `msg` argument is implicit
- funds accepted implicitly
- can be called as a function from another contract

Smart Contracts are Like Concurrent Objects

A Concurrent Perspective on Smart Contracts

Ilya Sergey[1] and Aquinas Hobor[2]

[1] University College London, United Kingdom
i.sergey@ucl.ac.uk
[2] Yale-NUS College and School of Computing, National University of Singapore
hobor@comp.nus.edu.sg

Abstract. In this paper, we explore remarkable similarities between multi-transactional behaviors of smart contracts in cryptocurrencies such as Ethereum and classical problems of shared-memory concurrency. We examine two real-world examples from the Ethereum blockchain and analyzing how they are vulnerable to bugs that are closely reminiscent to those that often occur in traditional concurrent programs. We then elaborate on the relation between observable contract behaviors and well-studied concurrency topics, such as atomicity, interference, synchronization, and resource ownership. The described *contracts-as-concurrent-objects* analogy provides deeper understanding of potential threats for smart contracts, indicate better engineering practices, and enable applications of existing state-of-the-art formal verification techniques.

Accounts using ***smart contracts*** in a blockchain
are like
threads using ***concurrent objects*** in shared memory.

Accounts using ***smart contracts*** in a blockchain
are like
threads using ***concurrent objects*** in shared memory.

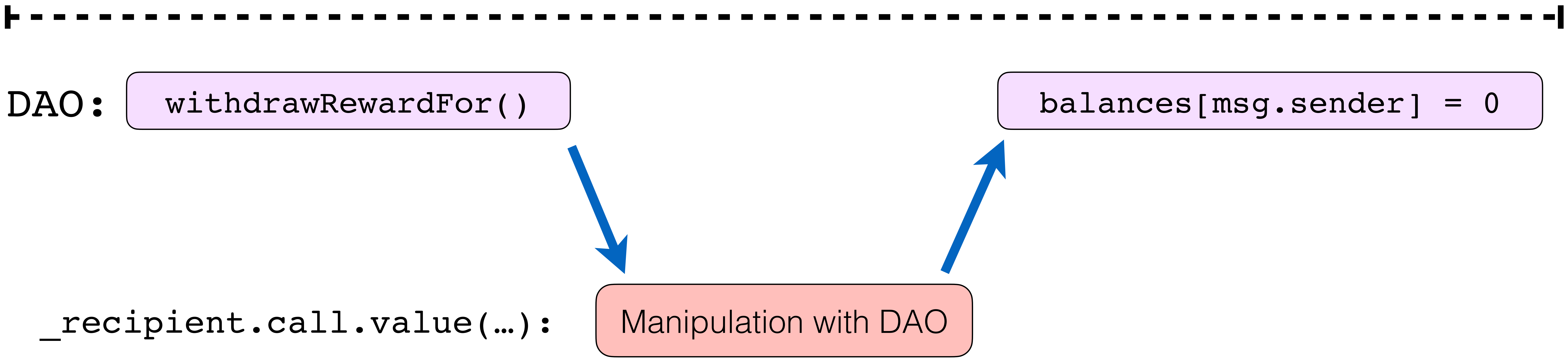
contract state	—	object state
call/send	—	context switching

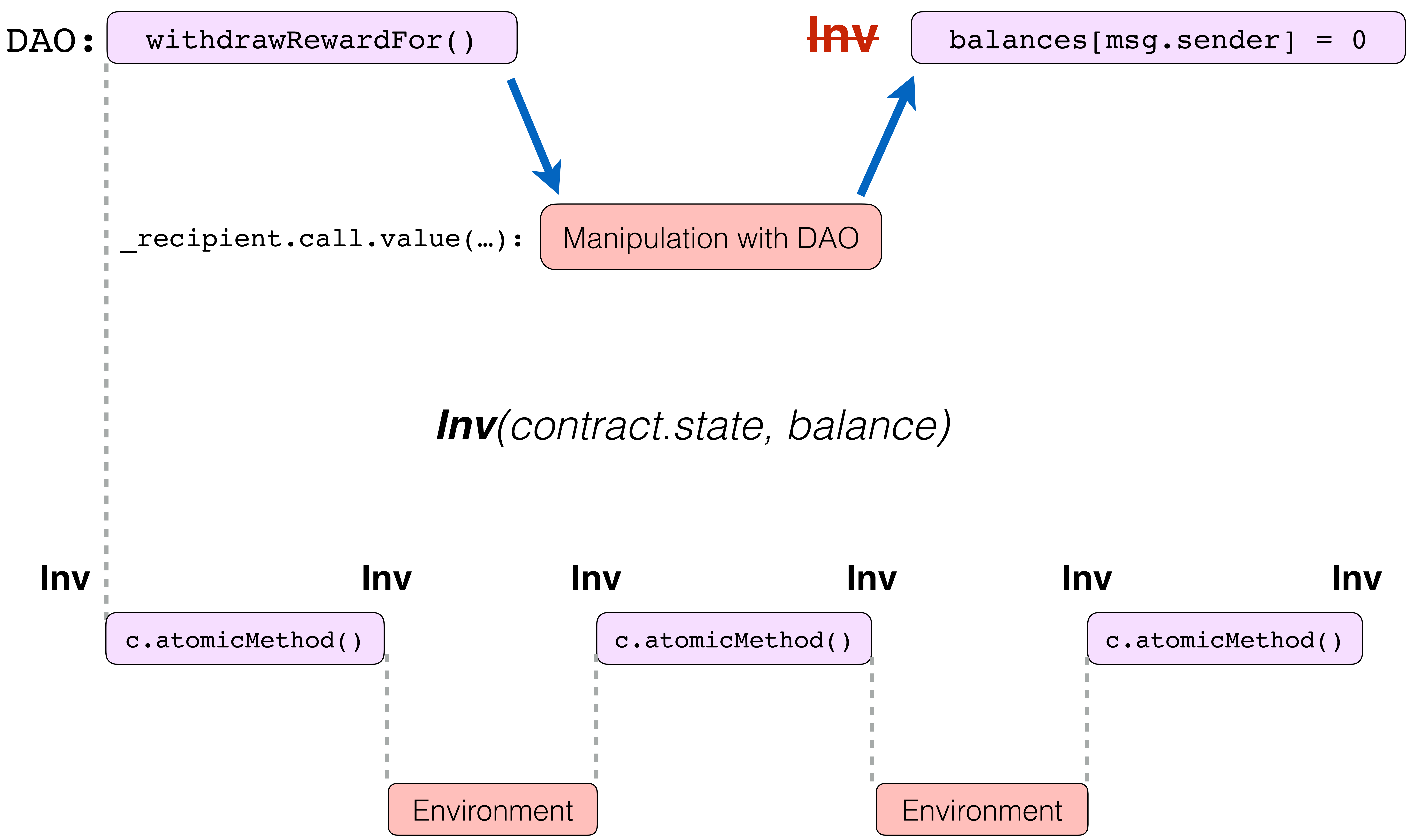
Reentrancy and multitasking

```
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  totalSupply -= balances[msg.sender];
  balances[msg.sender] = 0;
  paidOut[msg.sender] = 0;
  return true;
}
```

Reentrancy and multitasking

```
    // Burn DAO Tokens
    Transfer(msg.sender, 0, balances[msg.sender]);
    withdrawRewardFor(msg.sender); // be nice, and get his rewards
    totalSupply -= balances[msg.sender];
    balances[msg.sender] = 0;
    paidOut[msg.sender] = 0;
    return true;
}
```

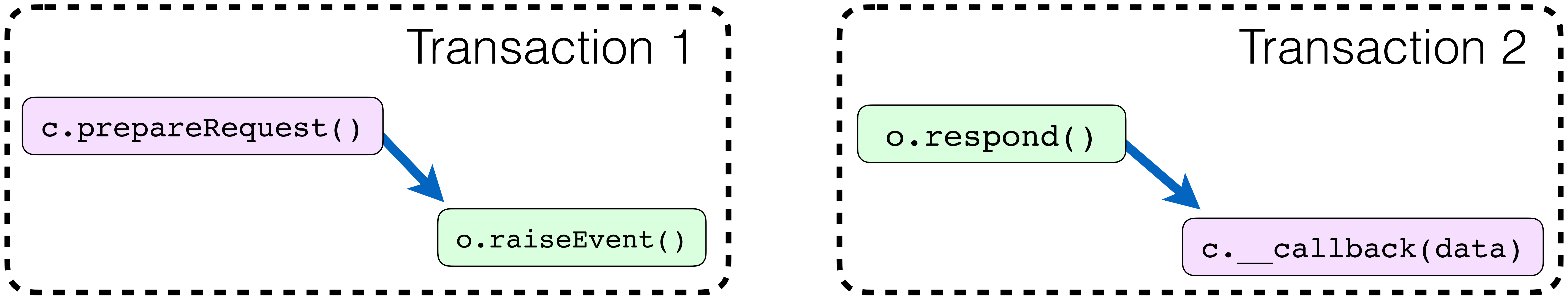



DAO:
withdrawRewardFor()
Inv
balances[msg.sender] = 0
_recipient.call.value(…):
Manipulation with DAO
Inv(contract.state, balance)
Inv
Inv
Inv
Inv
Inv
Inv
c.atomicMethod()
c.atomicMethod()
c.atomicMethod()
Environment
Environment

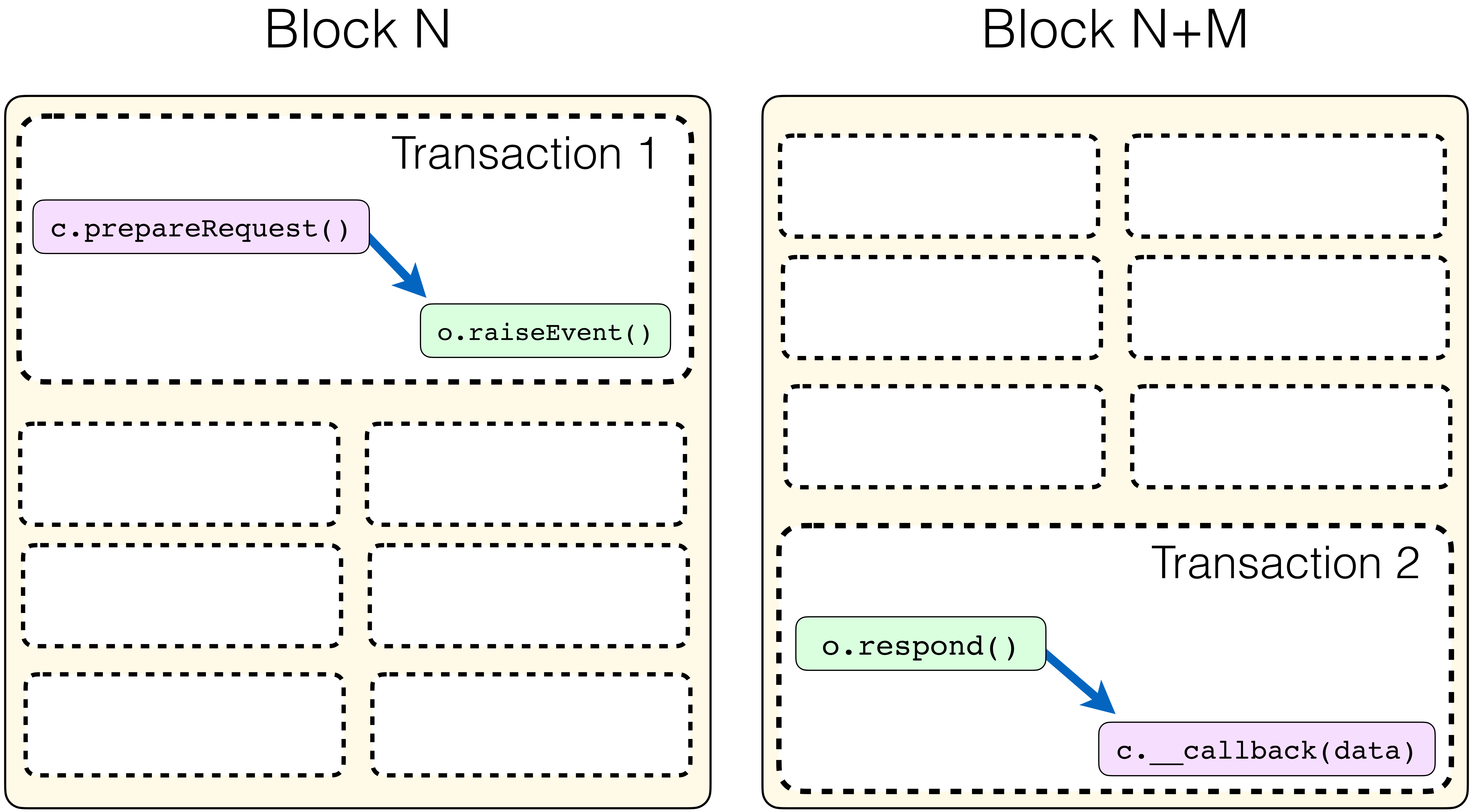
Accounts using ***smart contracts*** in a blockchain
are like
threads using ***concurrent objects*** in shared memory.

contract state	—	object state
call/send	—	context switching
Reentrancy	—	(Un)cooperative multitasking
Invariants	—	Atomicity

Querying an Oracle



Call/Return in Two Transactions



BlockKing via Oraclize

```
function enter() {
  if (msg.value < 50 finney) {
     msg.sender.send(msg.value);
     return;
  }
  warrior = msg.sender;
  warriorGold = msg.value;
  warriorBlock = block.number;
  bytes32 myid =
      oraclize_query(0,”WolframAlpha","random number between 1 and 9");
}
```

```
function __callback(bytes32 myid, string result) {
  if (msg.sender != oraclize_cbAddress()) throw;
  randomNumber = uint(bytes(result)[0]) - 48;
  process_payment();
}
```

More on that in the paper.

Accounts using ***smart contracts*** in a blockchain
are like
threads using ***concurrent objects*** in shared memory.

contract state	—	object state
call/send	—	context switching
Reentrancy	—	(Un)cooperative multitasking
Invariants	—	Atomicity
Non-determinism	—	data races

To Take Away

- *Byzantine Fault-Tolerant Consensus* is a common issue addressed in distributed systems, where participants *do not trust each other*.
- For a *fixed set* of nodes, a Byzantine consensus can be reached via
  - (a) making an agreement to proceed in *three phases*
  - (b) increasing the *quorum size*
  - These ideas are implemented in **PBFT**, which relies on *cryptographically signed* messages and *partial synchrony*.
- In *open* systems (blockchains), consensus can be reached via a universally accepted ***Fork-Chain-Rule***:
  - It measures the *amount of work*, while comparing two “conflicting” proposals

YSC4231: Parallel, Concurrent and Distributed Programming

Wrapping Up

Concurrency is Tricky

Concurrency is Tricky

- It can be ***very confusing***
- It takes ***a lot of*** time to get right
- … but we simply ***cannot get away without it***
- … because we want our programs to be *fast*
- … we want our interfaces to be *responsive*
- … and we want our systems to be *reliable*

We learned to *understand* concurrency and to implement it *correctly* and *efficiently*

- Amdahl's Law
- Safety, Liveness
- Dining Philosophers Problem
- Programming with Threads
- Event Orderings and Mutual Exclusion
- Linearizability and Sequential Consistency
- Spin-locks and contention
- Monitors: waiting and signalling
- Design of concurrent objects
- Fine-grained, lazy, and optimistic locking
- Concurrent Stacks, Queues, Skiplists
- Concurrent Elimination, ABA problem
- Thread pools
- Data race detection
- Asynchronous Computations via Futures
- Data parallelism, Splitters and Combiners
- Actors and message-passing
- Distributed consensus, Paxos, PBFT, Blockchains

We learned to *understand* concurrency and to implement it *correctly* and *efficiently*

- Amdahl's Law
- Safety, Liveness
- Dining Philosophers Problem
- Programming with Threads
- Event Orderings and Mutual
- Linearizability and Sequential Consistency
- Spin-locks and contention
- Monitors: waiting and signalling
- Design of concurrent objects
- Fine-grained, lazy, and optimistic locking
- Concurrent Stacks, Queues, Skiplists
- mination, ABA problem
- ction
- Asynchronous Computations via Futures
- Data parallelism, Splitters and Combiners
- Actors and message-passing
- Distributed consensus, Paxos, PBFT, Blockchains

Stuff we Didn't Discuss

- More Concurrent Algorithms
  - Concurrent Hashing, Counting Networks, Priority Queues
- Compilers and Concurrency
  - Automated Parallelisation, Memory Models for C/C++11 and Java
- Concurrent Garbage Collection
- Software Transactional Memory
- Web Services, Distributed File Systems, Gossip Protocols, Apache ZooKeeper
- Verification of Concurrent Algorithms
  - Linearisability proofs, Program Logics, embedding into Coq
- Formalisation and Verification of Distributed Protocols
  - I/O Automata, TLA+, Proof Automation, Composition, Invariant Inference

Where to go From Here

- Programming Languages for Concurrency
  - Erlang (everything is an actor)
  - Go (lightweight threads)
  - Kotlin (Coroutines for asynchronous programming)
  - Rust (really cool type system prevents data races)

Research in Concurrency

- Conferences (proceedings available on the web):
  - Principles of Distributed Computing (PODC)
  - International Symposium on DIStributed Computing (DISC)
  - Principles and Practice of Parallel Programming (PPoPP)
  - Symposium on Operating Systems Principles (SOSP)
  - Operating Systems Design and Implementation(OSDI)
  - Programming Language Design and Implementation (PLDI)

- Researchers to check out
  - Edsger Dijkstra, Leslie Lamport, Barbara Liskov, Nancy Lynch, Maurice Herlihy, Faith Ellen, James Aspnes, Nir Shavit

Papers On Distributed Consensus

- L. Lamport. *The part-time parliament*. ACM Trans. Comput. Syst., 16(2):133–169, 1998.
- L. Lamport. *Paxos made simple*. SIGACT News, 32, 2001.
- T.D. Chandra et al. *Paxos made live: an engineering perspective*. PODC 2007
- B. W. Lampson, *The ABCD's of Paxos*. PODC 2001
- P. Kellomäki. *An Annotated Specification of the Consensus Protocol of Paxos Using Superposition in PVS*. 2004
- C. Dragoi et al. *PSync: a partially synchronous language for fault-tolerant distributed algorithms*. In POPL, 2016.
- M. Jaskelioff and S. Merz. *Proving the correctness of disk Paxos*. Archive of Formal Proofs, 2005.
- C. Hawblitzel et al. *IronFleet: proving practical distributed systems correct*. In SOSP 2015.
- D. Ongaro and J. K. Ousterhout. *In search of an understandable consensus algorithm*. USENIX Annual Technical Conference, 2014
- B.M. Oki and B. Liskov, *Viewstamped Replication: A General Primary Copy*. PODC 1988
- O. Padon, et al. *Paxos made EPR: decidable reasoning about distributed protocols.* PACMPL, 1(OOPSLA):108:1–108:31, 2017.
- V. Rahli, et al. *Formal specification, verification, and implementation of fault-tolerant systems using EventML*. In AVOCS. EASST, 2015.
- A. Pillai, *Mechanised Verification of Paxos-like Consensus Protocols*, BSc Thesis, 2018
- R. van Renesse and D. Altinbuken. Paxos Made Moderately Complex. ACM Comput. Surv., 47(3):42:1–42:36, 2015.
- J.R. Wilcox et al., *Verdi: a framework for implementing and formally verifying distributed systems*, PLDI 2015
- Á. García-Pérez et al., *Paxos Consensus, Deconstructed and Abstracted*, ESOP 2018

Papers On Distributed Consensus

- L. Lamport. *The part-time parliament*. ACM Trans. Comput. Syst., 16(2):133–169, 1998.
- **L. Lamport. *Paxos made simple*. SIGACT News, 32, 2001.**
- T.D. Chandra et al. *Paxos made live: an engineering perspective*. PODC 2007
- B. W. Lampson, *The ABCD's of Paxos*. PODC 2001
- P. Kellomäki. *An Annotated Specification of the Consensus Protocol of Paxos Using Superposition in PVS*. 2004
- C. Dragoi et al. *PSync: a partially synchronous language for fault-tolerant distributed algorithms*. In POPL, 2016.
- M. Jaskelioff and S. Merz. *Proving the correctness of disk Paxos*. Archive of Formal Proofs, 2005.
- C. Hawblitzel et al. *IronFleet: proving practical distributed systems correct*. In SOSP 2015.
- D. Ongaro and J. K. Ousterhout. *In search of an understandable consensus algorithm*. USENIX Annual Technical Conference, 2014
- B.M. Oki and B. Liskov, *Viewstamped Replication: A General Primary Copy*. PODC 1988
- O. Padon, et al. *Paxos made EPR: decidable reasoning about distributed protocols.* PACMPL, 1(OOPSLA):108:1–108:31, 2017.
- V. Rahli, et al. *Formal specification, verification, and implementation of fault-tolerant systems using EventML*. In AVOCS. EASST, 2015.
- A. Pillai, *Mechanised Verification of Paxos-like Consensus Protocols*, BSc Thesis, 2018
- **R. van Renesse and D. Altinbuken. Paxos Made Moderately Complex. ACM Comput. Surv., 47(3):42:1–42:36, 2015.**
- **J.R. Wilcox et al., *Verdi: a framework for implementing and formally verifying distributed systems*, PLDI 2015**
- **Á. García-Pérez et al., *Paxos Consensus, Deconstructed and Abstracted*, ESOP 2018**

Papers on BFT, Blockchains, Smart Contracts

- L. Lamport et al. *The Byzantine Generals Problem*. ACM Trans. Program. Lang. Syst. 4(3): 382-401, 1982
- M. Castro and B. Liskov. *Practical Byzantine Fault Tolerance*. In OSDI, 1999
- R. Guerraoui et al. *The next 700 BFT protocols*. In EuroSys 2010
- L. Lamport. *Byzantizing Paxos by Refinement*. In DISC, 2011
- C. Cachin et al. *Introduction to Reliable and Secure Distributed Programming* (2. ed.). Springer, 2011
- L. Lamport. *Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm* (2013)
- M. Castro. *Practical Byzantine Fault Tolerance*. Technical Report MIT-LCS-TR-817. Ph.D. MIT, Jan. 2001.
- V. Rahli et al. *Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq*. ESOP, 2018
- L. Luu et al. *A Secure Sharding Protocol For Open Blockchains*. ACM CCS, 2016
- M. Al-Bassam et al. *Chainspace: A Sharded Smart Contracts Platform*. NDSS 2018
- E. Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*, MSc Thesis, 2016
- D. Maziéres. *The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus*, 2016.
- G. Pîrlea, I. Sergey. *Mechanising blockchain consensus*. In CPP, 2018.
- I. Sergey, A. Hobor. *A Concurrent Perspective on Smart Contracts.* In WTSC 2017

Papers on BFT, Blockchains, Smart Contracts

- **L. Lamport et al. *The Byzantine Generals Problem*. ACM Trans. Program. Lang. Syst. 4(3): 382-401, 1982**
- **M. Castro and B. Liskov. *Practical Byzantine Fault Tolerance*. In OSDI, 1999**
- R. Guerraoui et al. *The next 700 BFT protocols*. In EuroSys 2010
- L. Lamport. *Byzantizing Paxos by Refinement*. In DISC, 2011
- C. Cachin et al. *Introduction to Reliable and Secure Distributed Programming* (2. ed.). Springer, 2011
- L. Lamport. *Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm* (2013)
- M. Castro. *Practical Byzantine Fault Tolerance*. Technical Report MIT-LCS-TR-817. Ph.D. MIT, Jan. 2001.
- V. Rahli et al. *Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq*. ESOP, 2018
- L. Luu et al. *A Secure Sharding Protocol For Open Blockchains*. ACM CCS, 2016
- M. Al-Bassam et al. *Chainspace: A Sharded Smart Contracts Platform*. NDSS 2018
- E. Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*, MSc Thesis, 2016
- D. Maziéres. *The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus*, 2016.
- **G. Pîrlea, I. Sergey. *Mechanising blockchain consensus*. In CPP, 2018.**
- **I. Sergey, A. Hobor. *A Concurrent Perspective on Smart Contracts.* In WTSC 2017**

Related Classes at NUS School of Computing

- CS4231: Parallel and Distributed Algorithms
  - Parallel programming: mutual exclusion, semaphores, consistency, wait-free synchronization. Distributed computing: time, global state, snapshots, message ordering. Relationships: consensus, fault-tolerance, transactions, self-stabilization.

- CS5223: Distributed Systems
  - Process Management Communication in Distributed Systems; Distributed Synchronisation; Distributed Real-time Systems; File Systems; Naming Security; Fault Tolerant Distributed Systems; Distributed Simulation; WWW
  - Autumn term: *theory-oriented* (Prof. Haifeng Yu),
    Spring term: *practice-oriented* (Prof. Jialin Li)


Davidlohr Bueso
@davidlohr
A programmer had a problem. He thought to himself, "I know, I'll solve it with threads!". has Now problems. two he
7:16 AM · Jan 9, 2013 · Twitter Web Client
4.5K Retweets 1.4K Likes

The End

blue.nus.edu.sg