

# YSC3248: Parallel, Concurrent and Distributed Programming

Byzantine Fault Tolerance and Blockchains

# 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

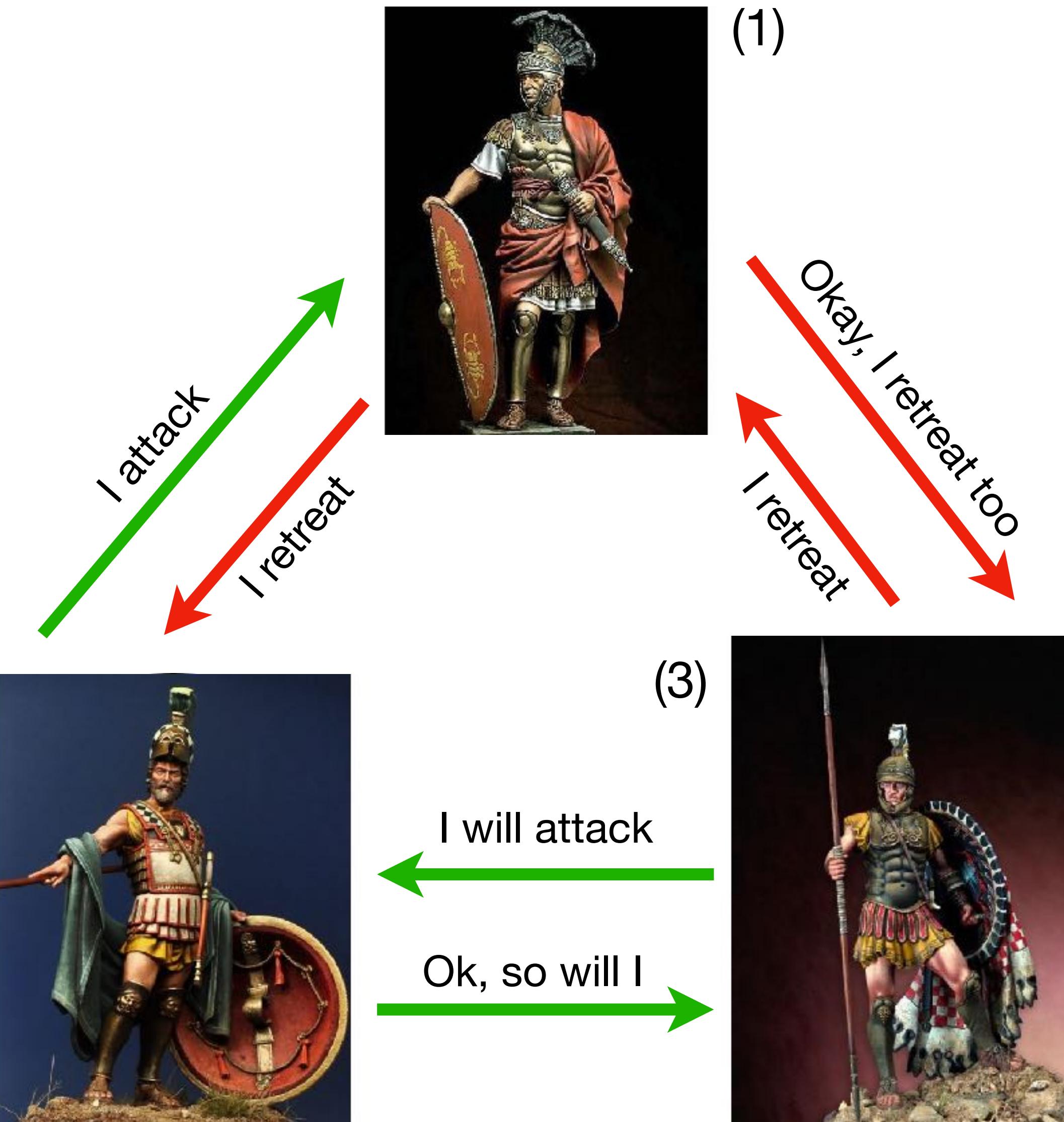


# Byzantine Consensus

- All **loyal generals** decide upon the *same* plan of action.
- A *small number of traitors* ( $f \ll N$ ) *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 a 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 *malfuncting replicas or adversaries*
- *Byzantine army* is a *deterministic replicate 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
  - 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

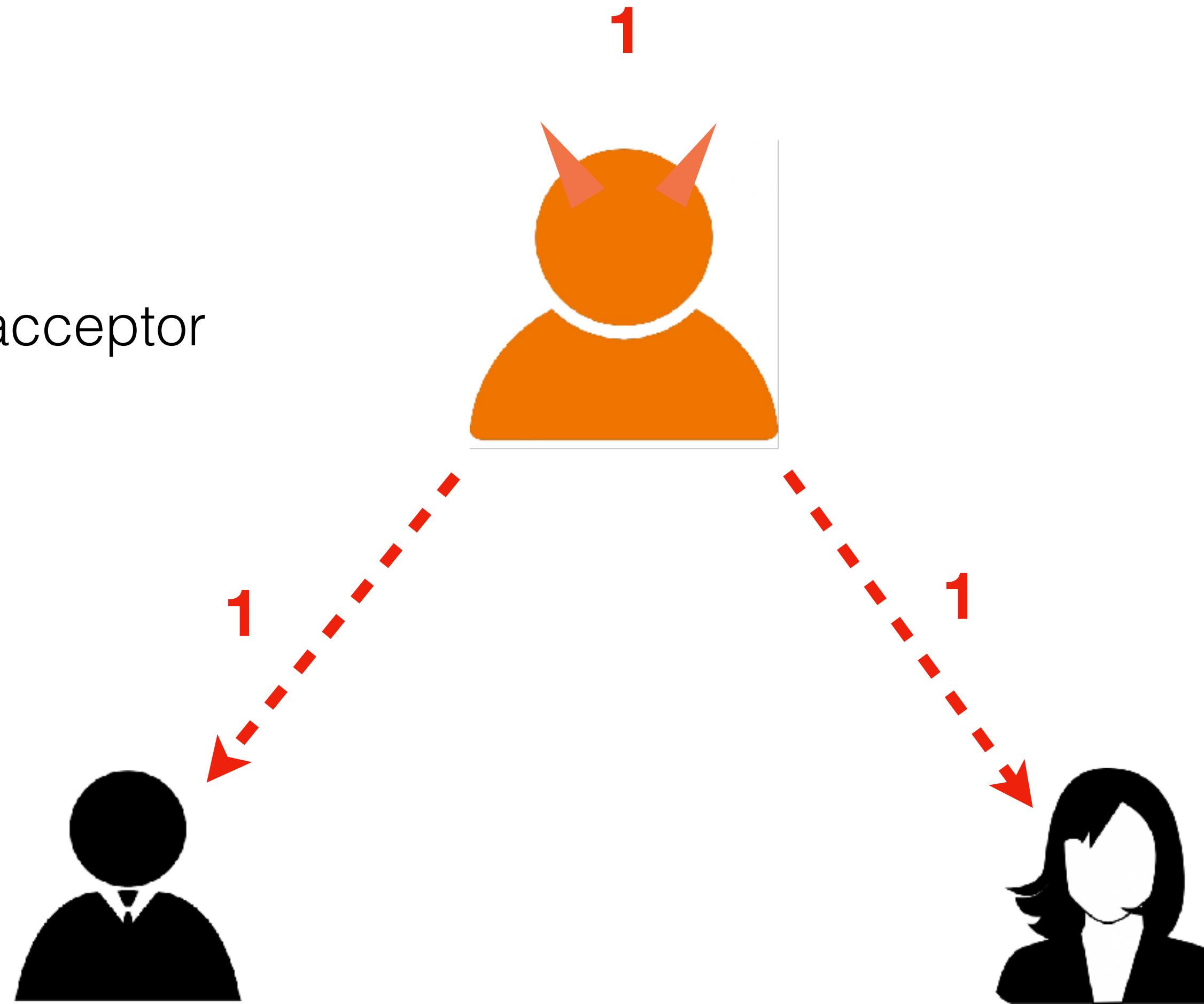
# Paxos and Byzantine Faults

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



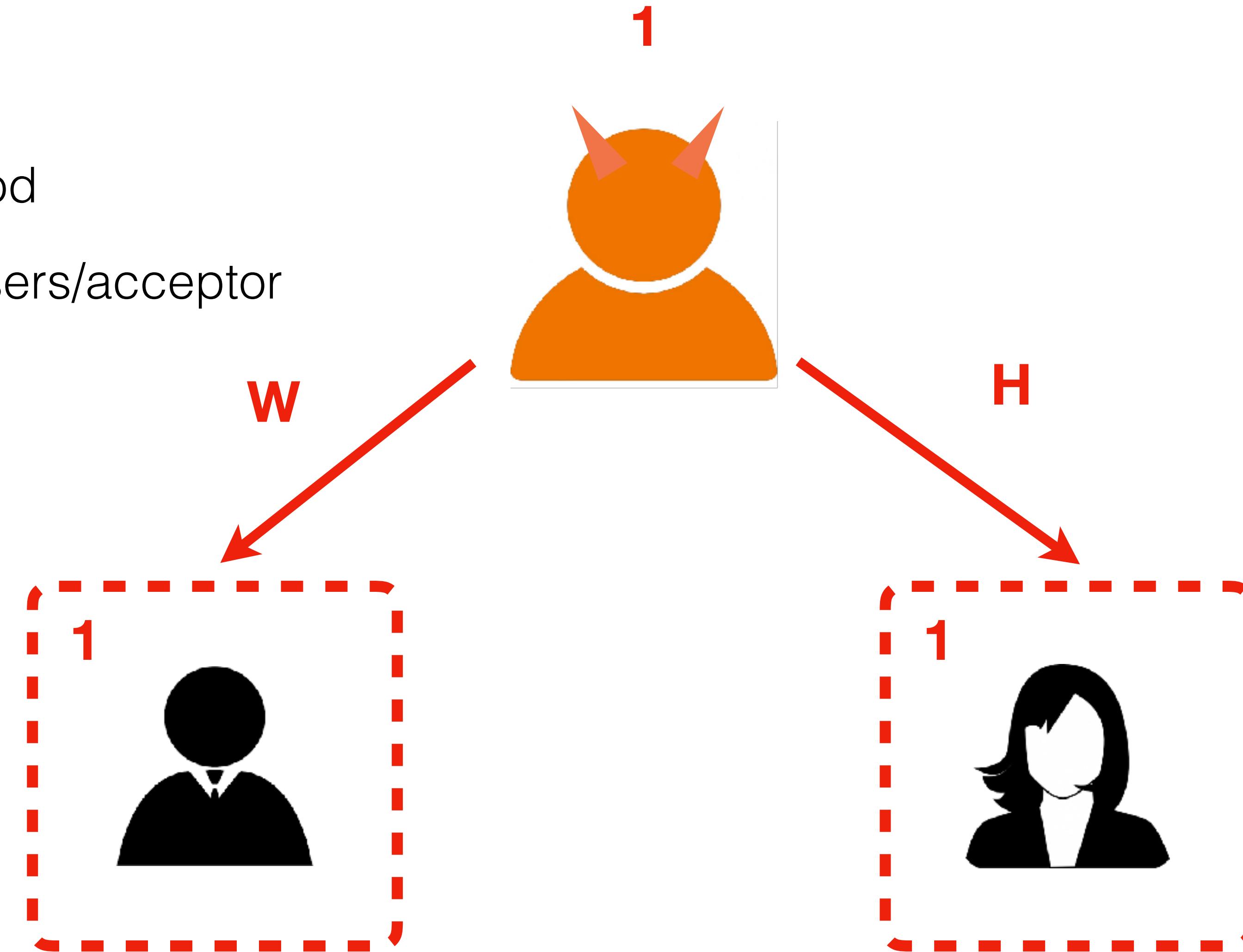
# Paxos and Byzantine Faults

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



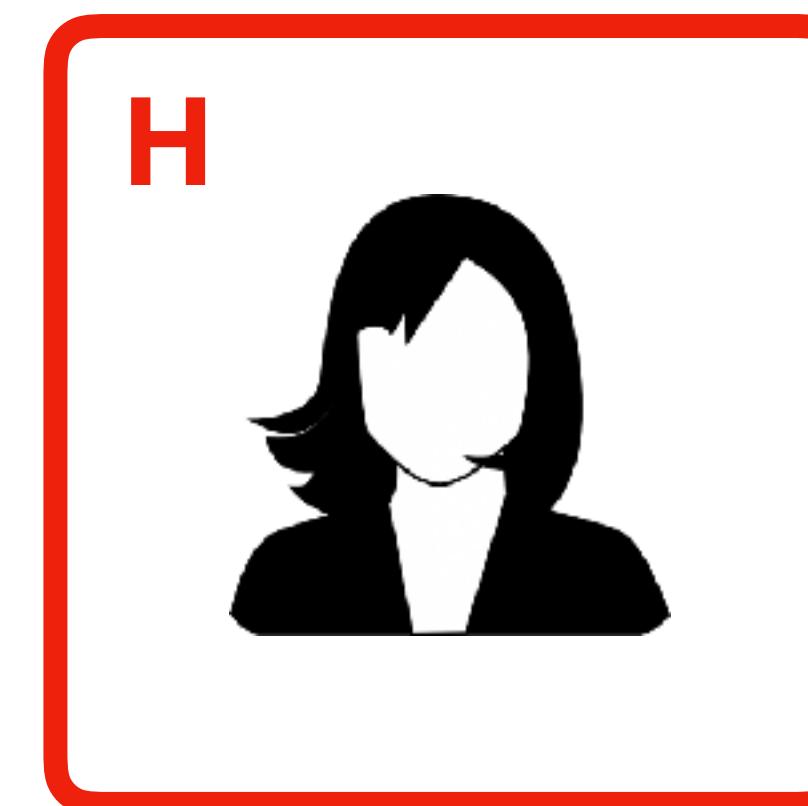
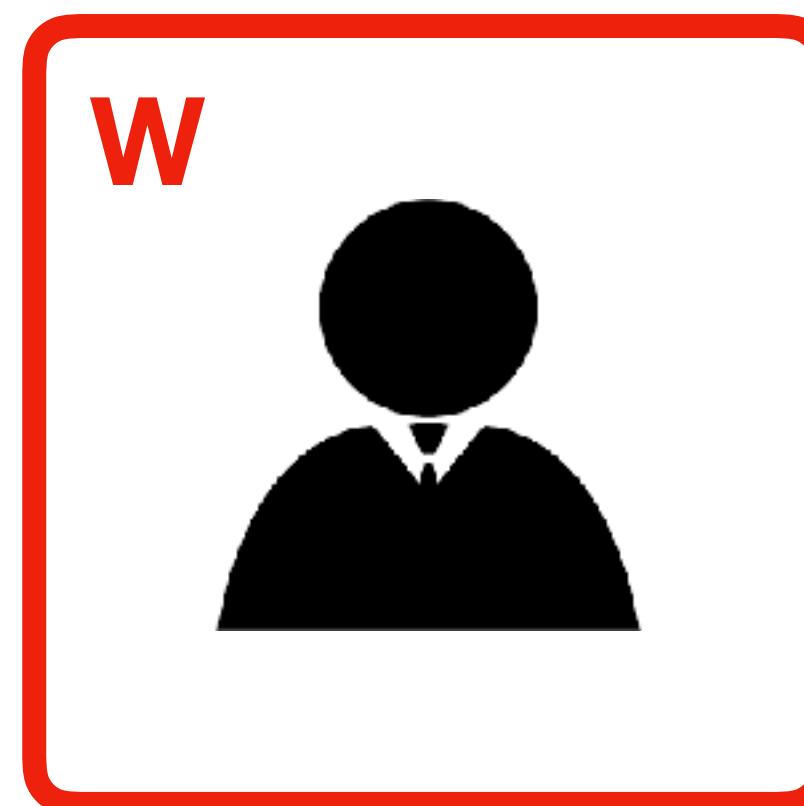
# Paxos and Byzantine Faults

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



# Paxos and Byzantine Faults

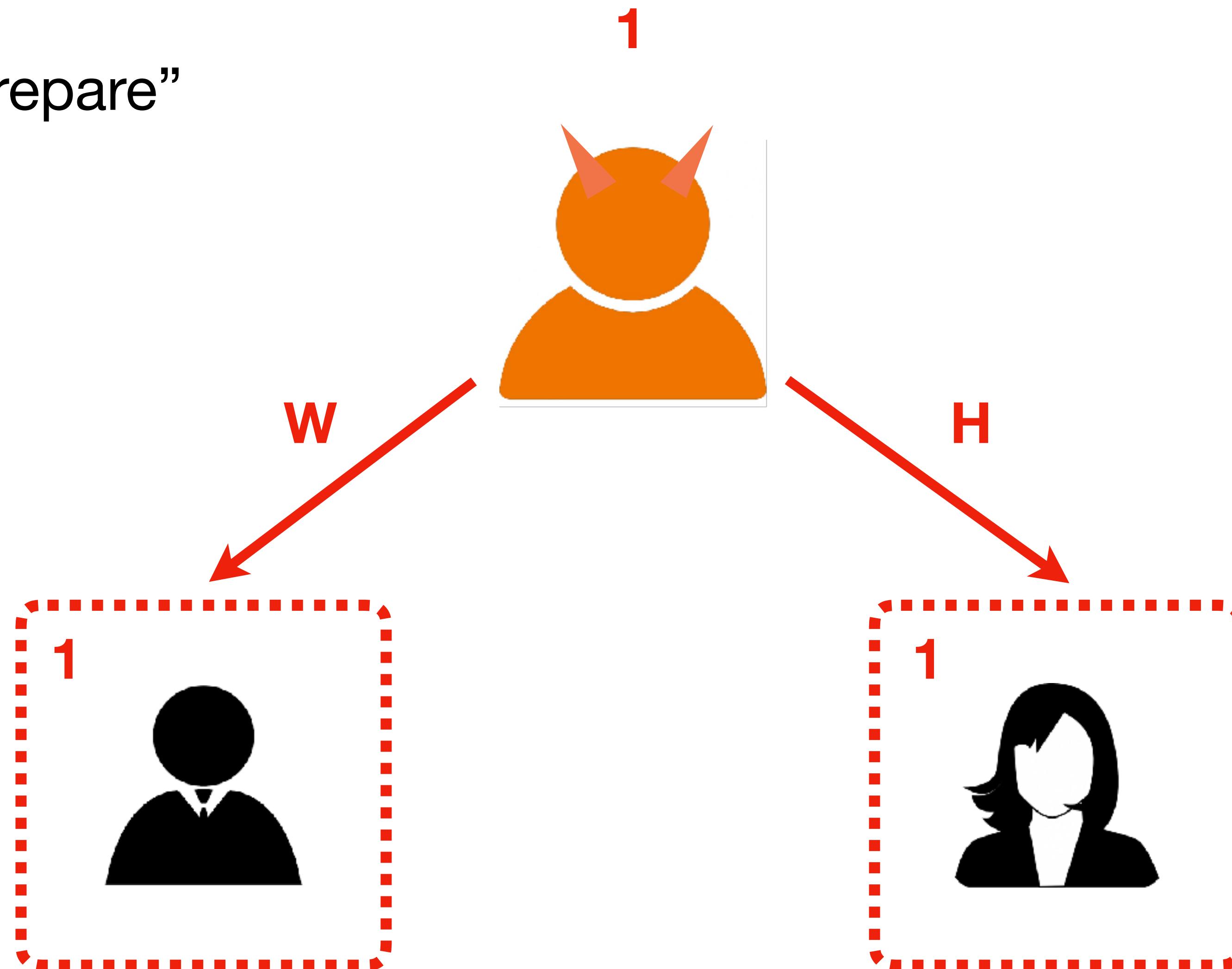
- $N = 3, f = 1$
- $N/2 + 1 = 2$  are good
- everyone is proposers/acceptor



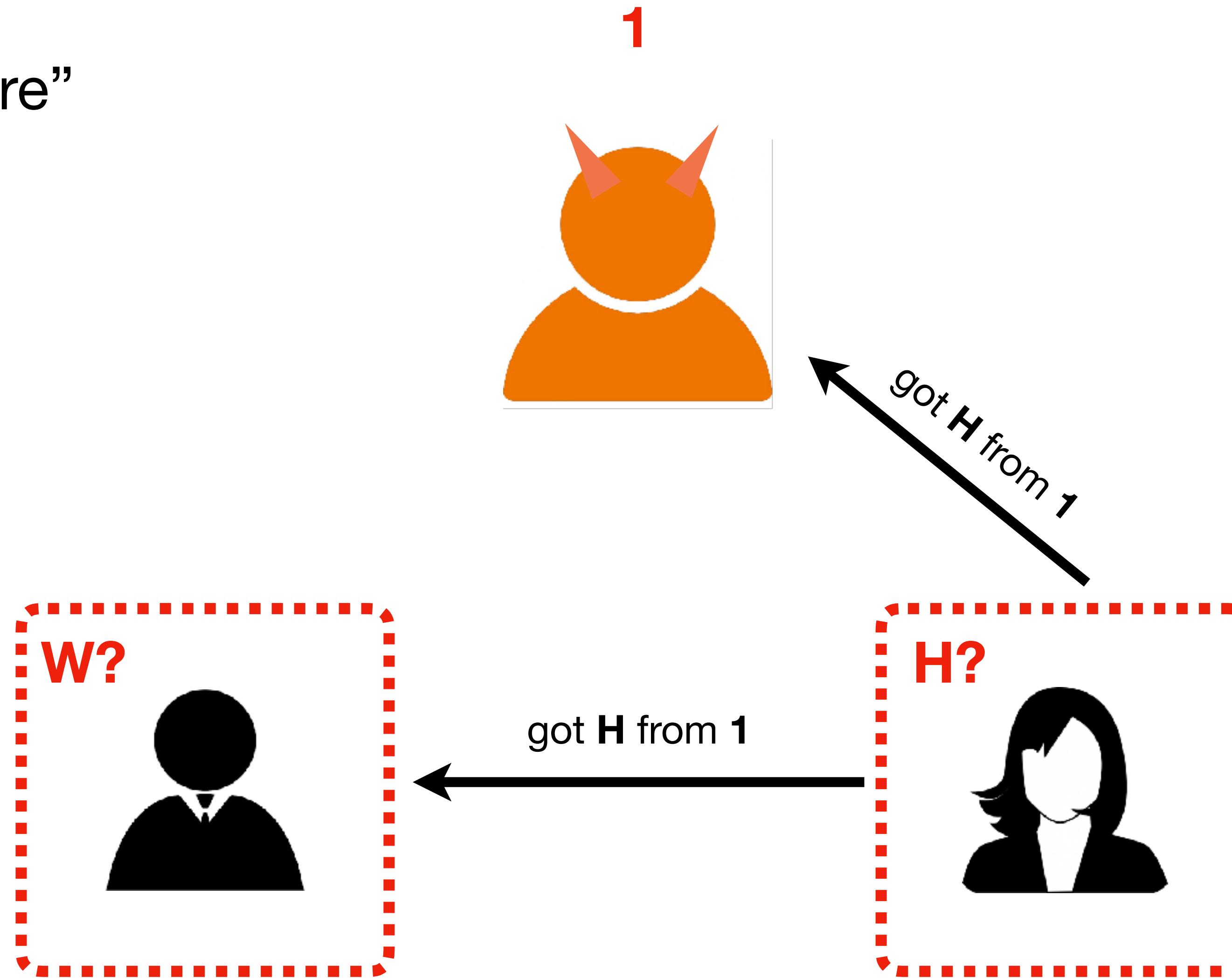
# What went wrong?

- 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).

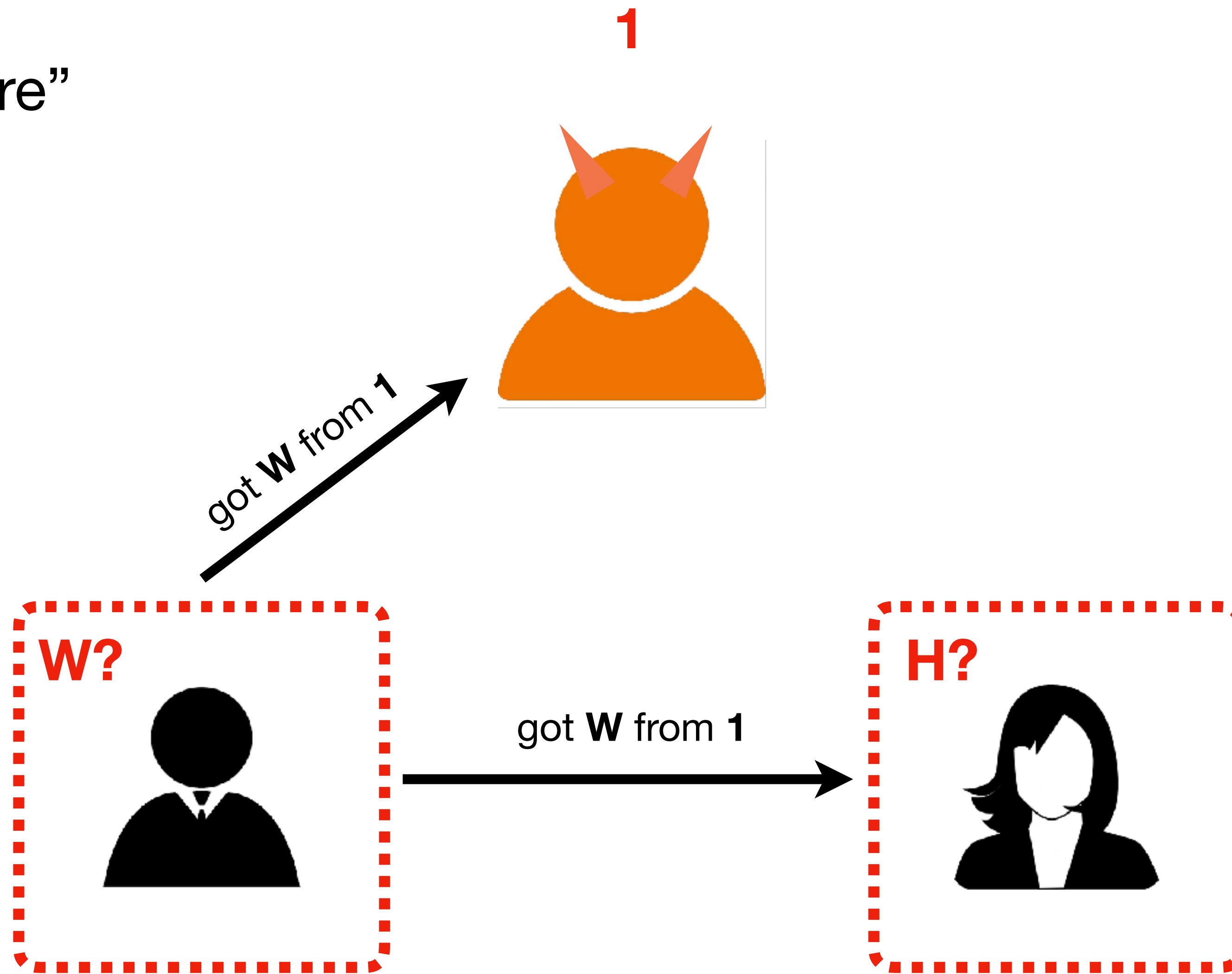
## Phase 1: “Pre-prepare”



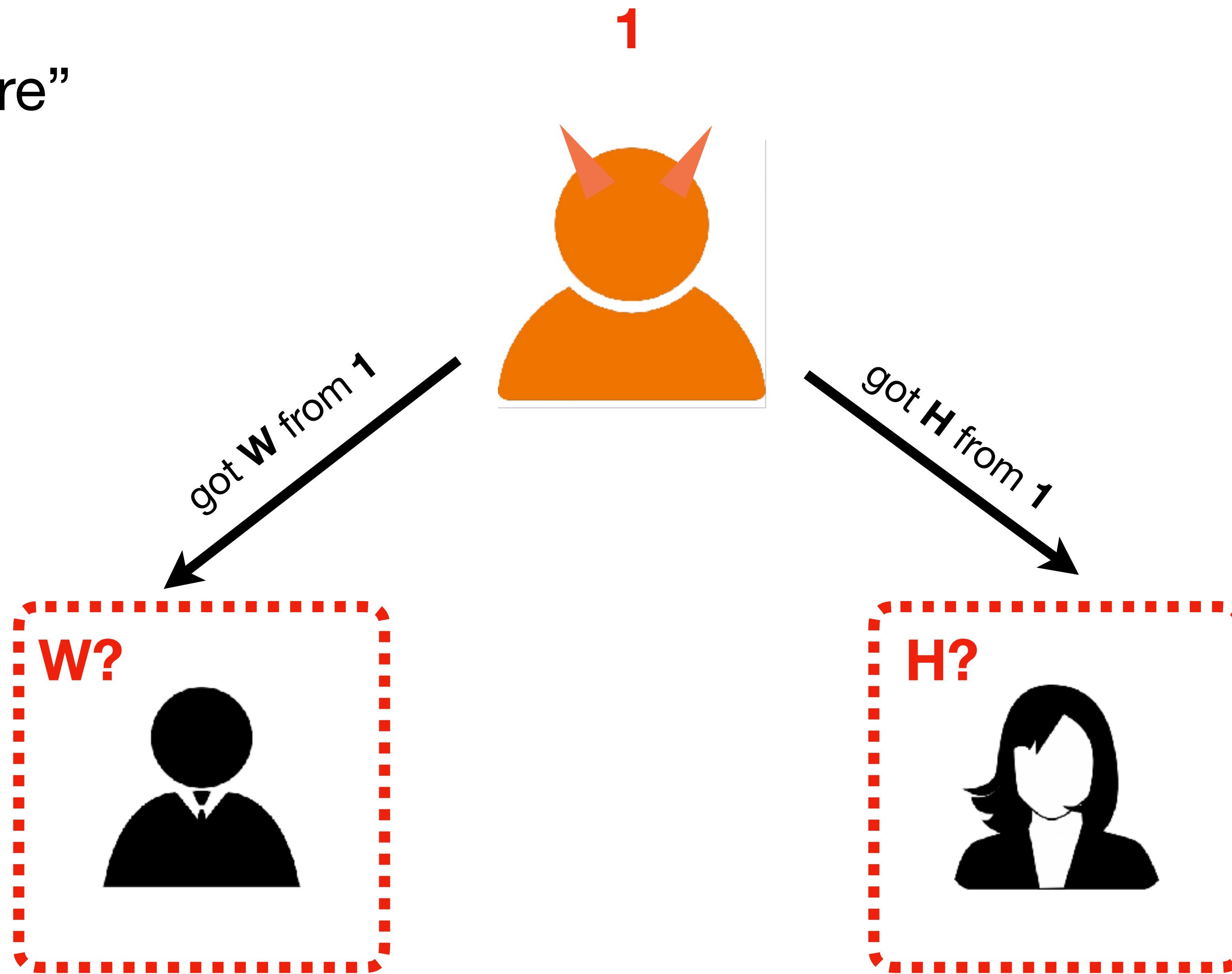
## Phase 2: “Prepare”



## Phase 2: “Prepare”

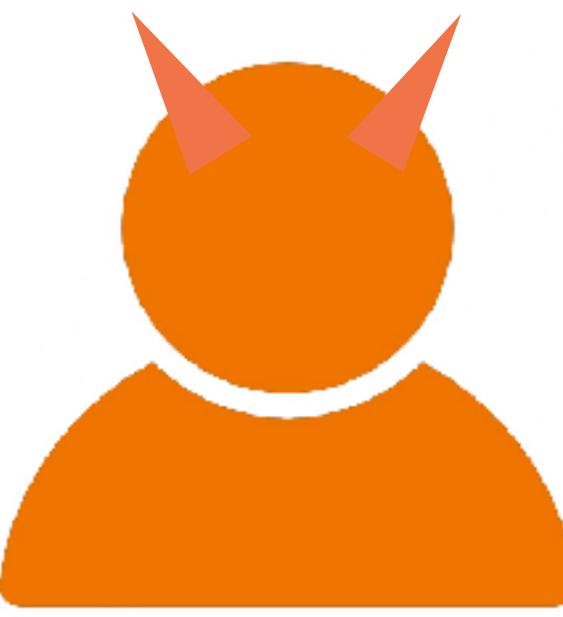
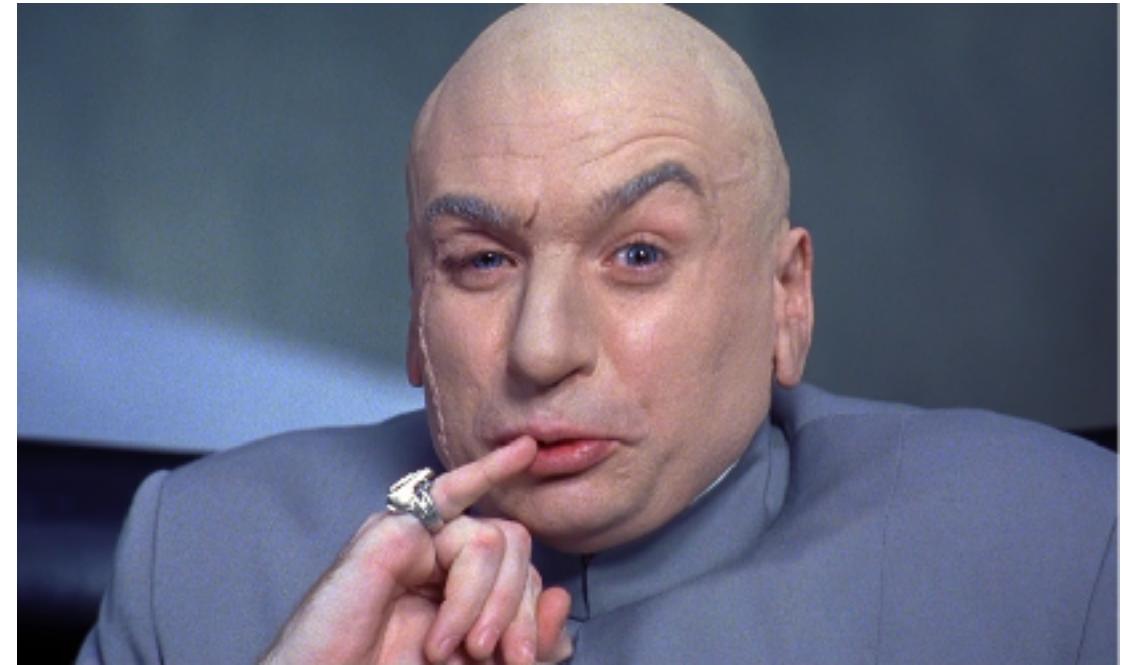


## Phase 2: “Prepare”

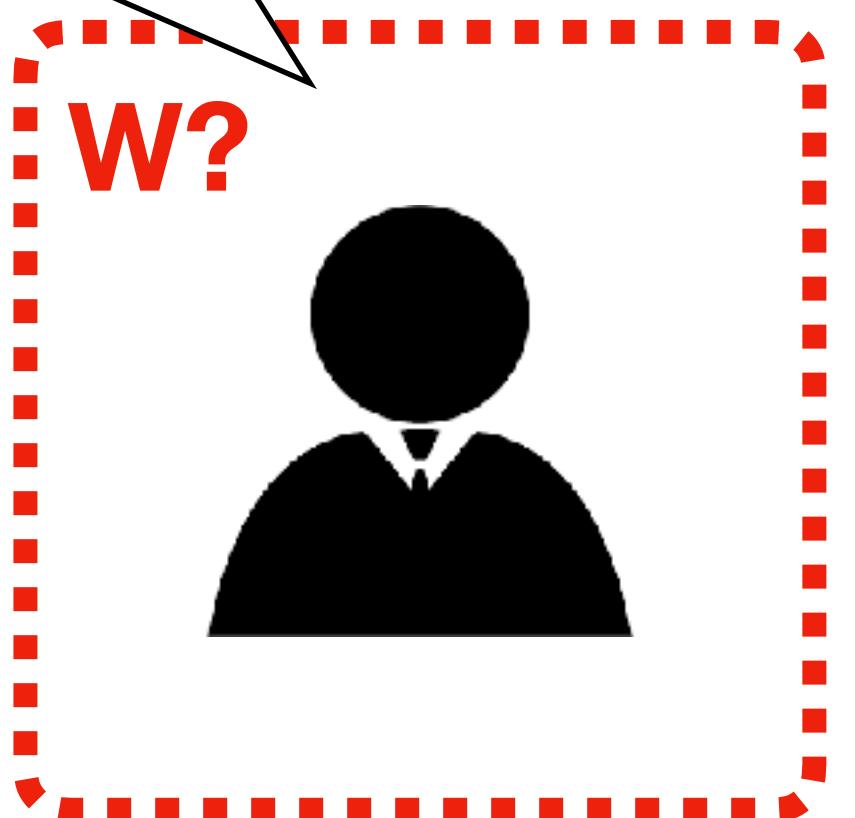


## Phase 2: “Prepare”

1



**Two** out of **three**  
want to commit **W**  
It's a **quorum** for **W!**

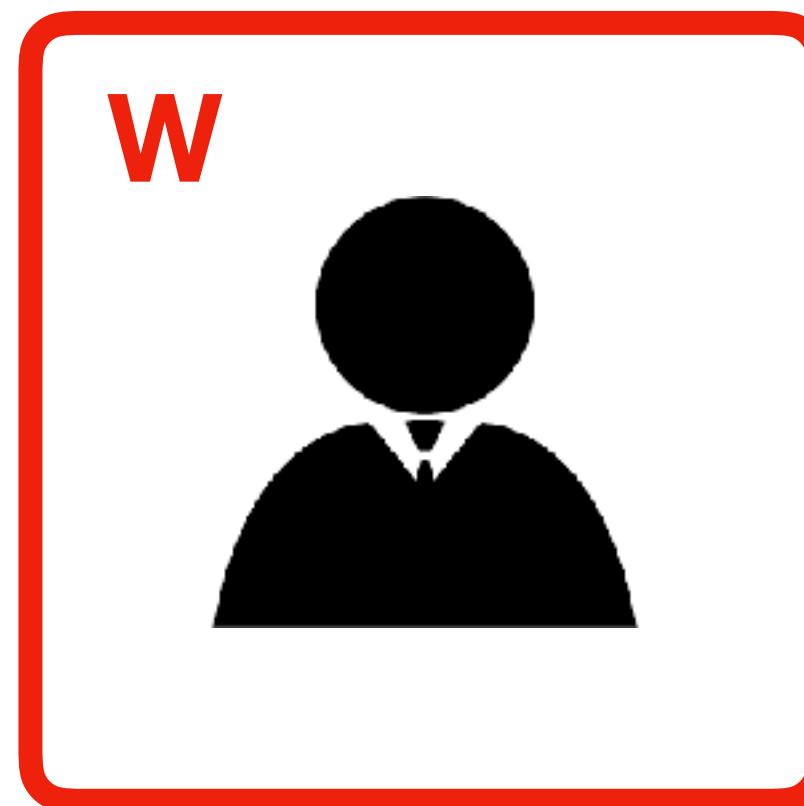


**Two** out of **three**  
want to commit **H**  
It's a **quorum** for **H!**



1

## Phase 3: “Commit”

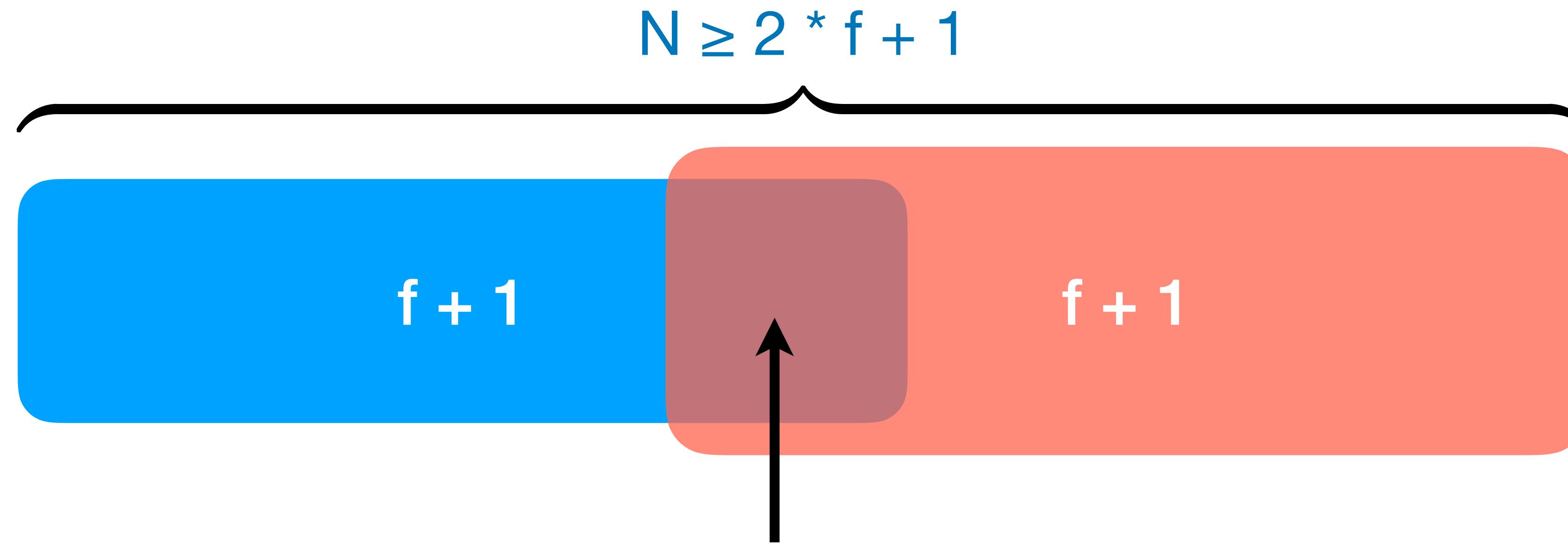


# What went wrong now?

- 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



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

# Choosing the Quorum Size

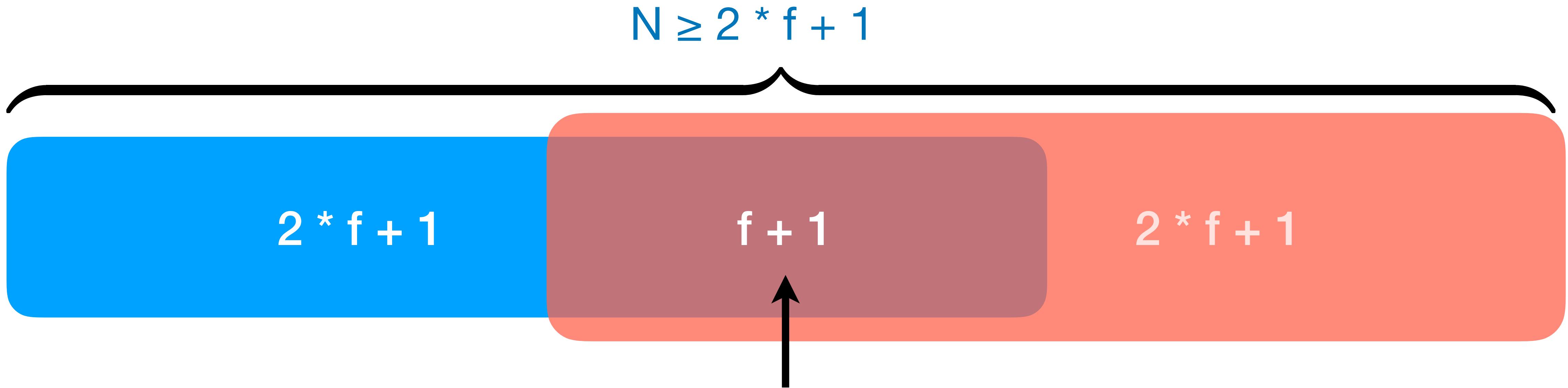


An adversarial node *in the intersection* can “lie” about the value:

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  $\geq \frac{2}{3} * N + 1$   
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

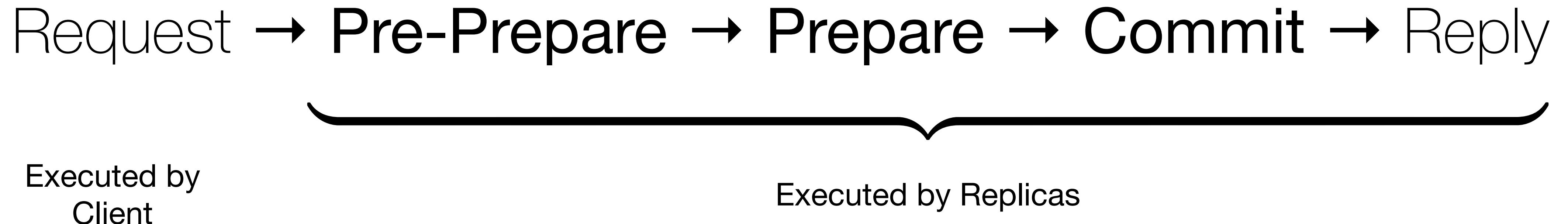
# Practical Byzantine Fault Tolerance (PBFT)

- 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

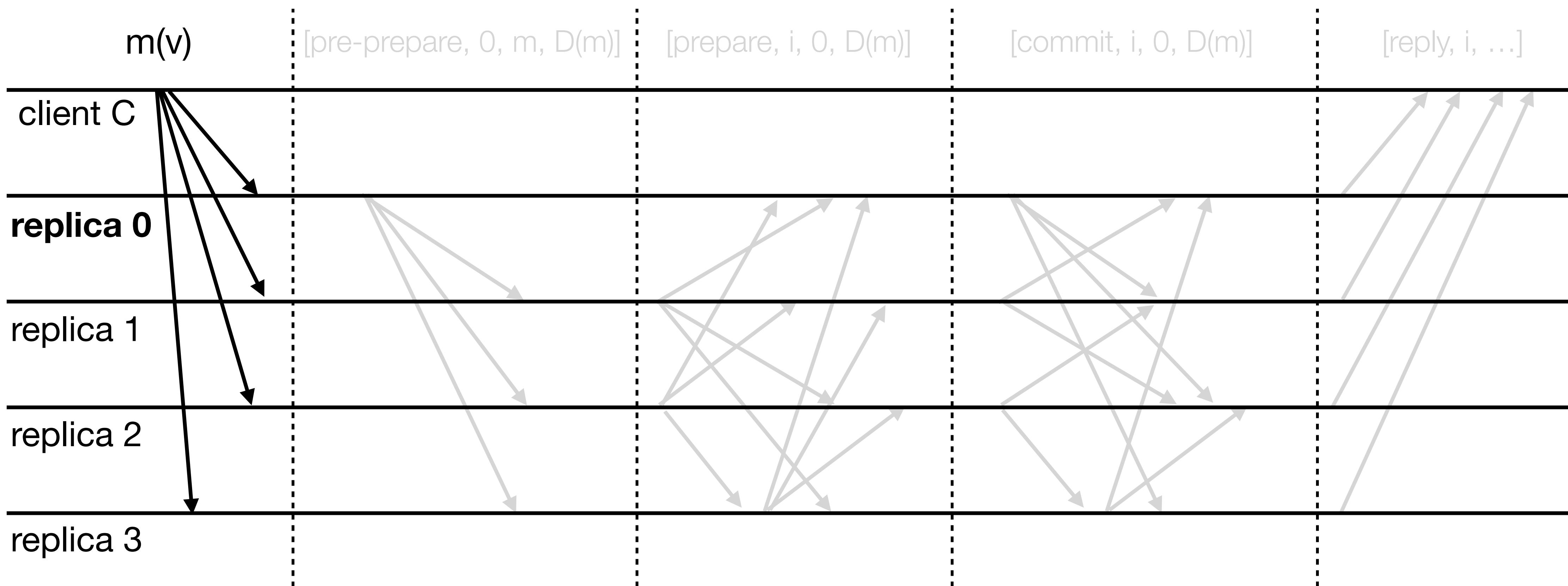
- **Replicas** — nodes participating in a consensus  
(no more *acceptor/proposer* dichotomy)
- A *dedicated replica* (primary) acts as a proposer/leader
  - A primary can be re-elected if suspected to be compromised
  - **Backups** — other, non-primary replicas
- **Clients** — communicate directly with 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



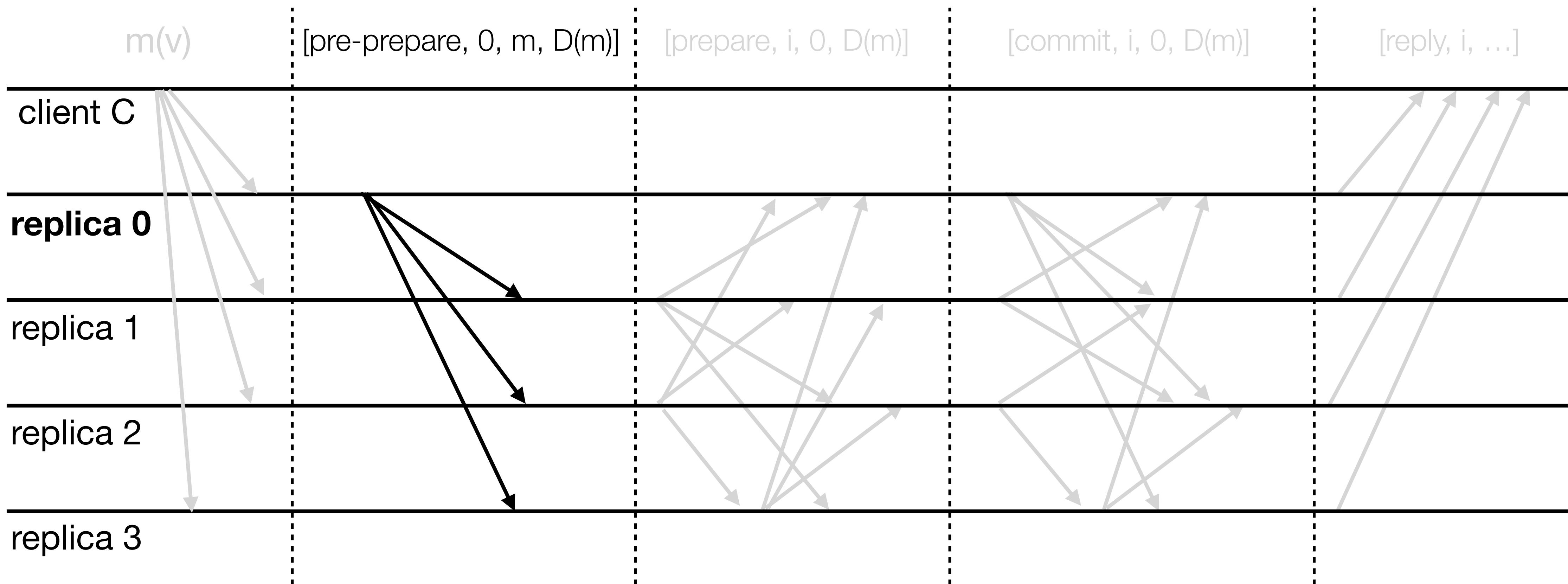
# Request

Client C sends a message to *all* replicas



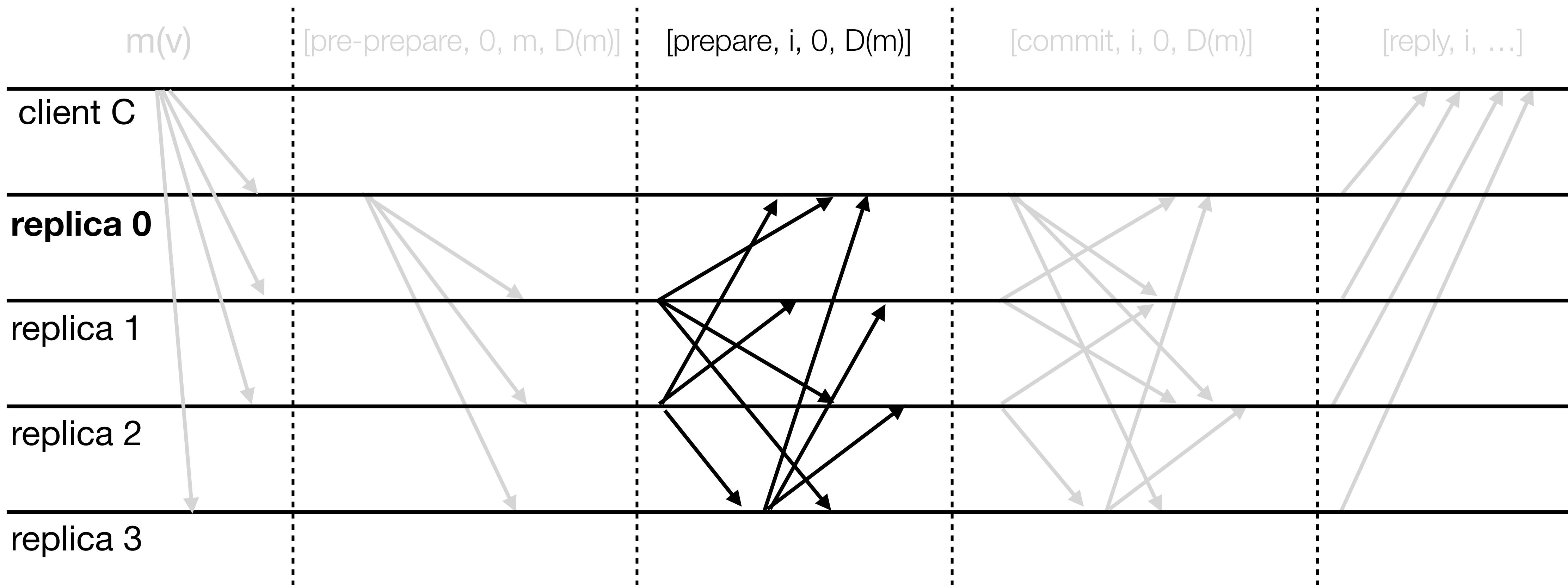
# Pre-prepare

- 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



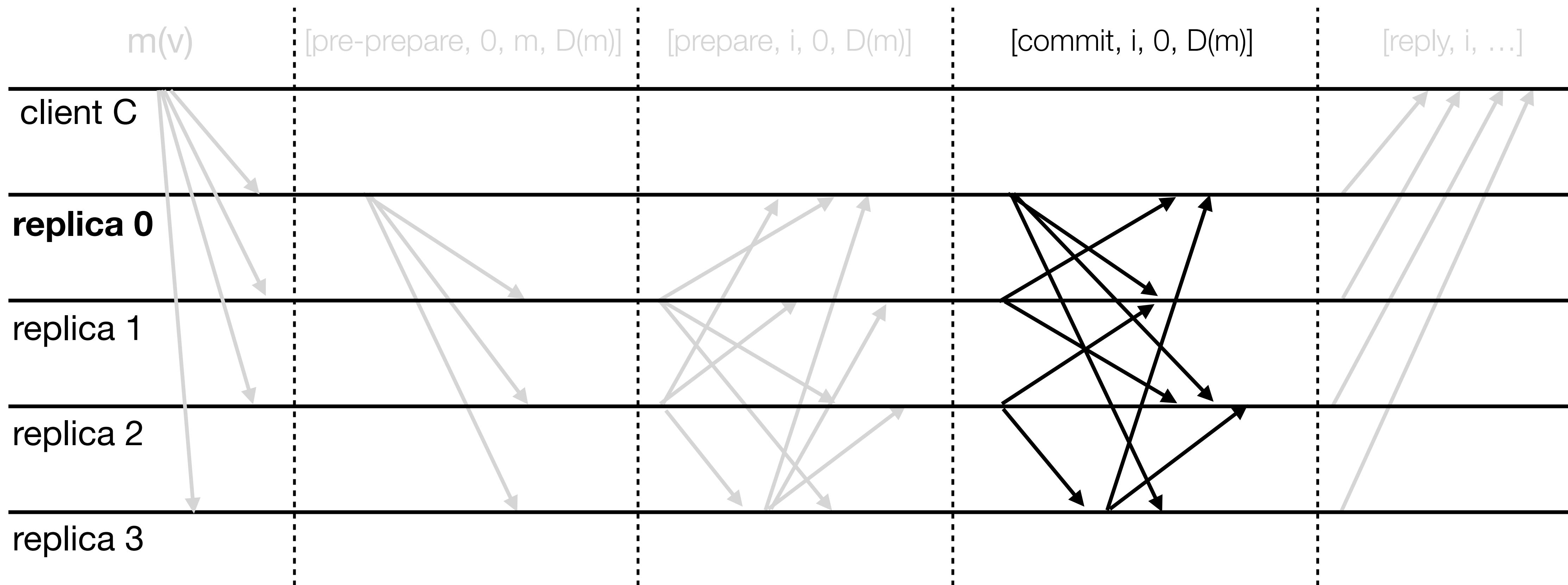
# Prepare

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



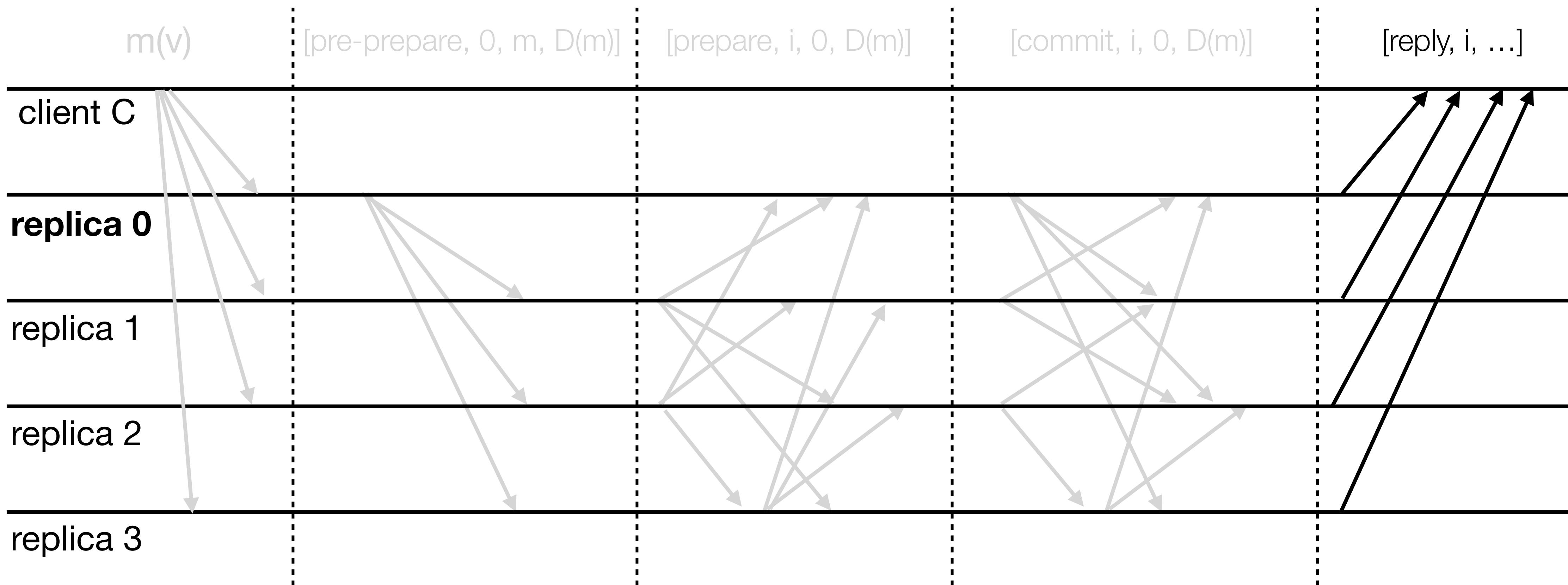
# Commit

- 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**
  - **Elastico/Zilliqa**
  - **Chainspace**
- Used for implementing *to speed-up* blockchain-based consensus
- Many blockchain solutions build on similar ideas
  - **Stellar Consensus Protocol**

# 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

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

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

blockchain  
consensus protocol

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

transactions  
can be *anything*

$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$  $[tx_5, tx_3] \rightarrow [tx_4] \rightarrow [tx_1, tx_2]$  $tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$

$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$  $[tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$  $tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$

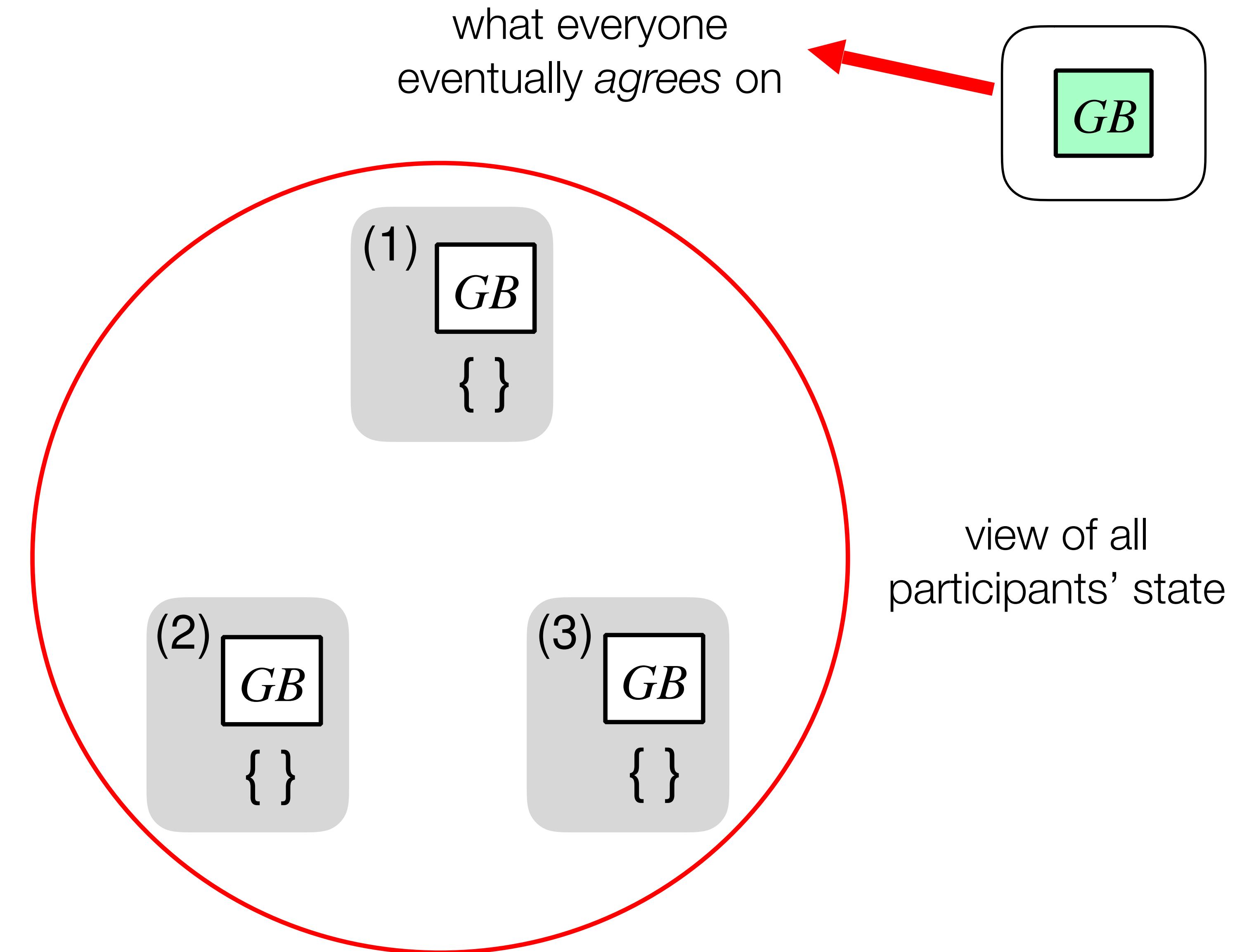
$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$  $[] \leftarrow [tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$ 

**GB** = genesis block

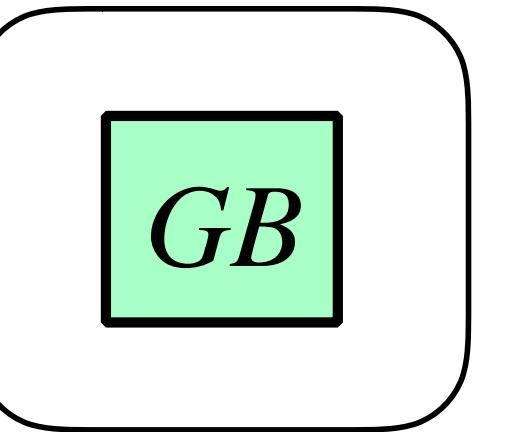
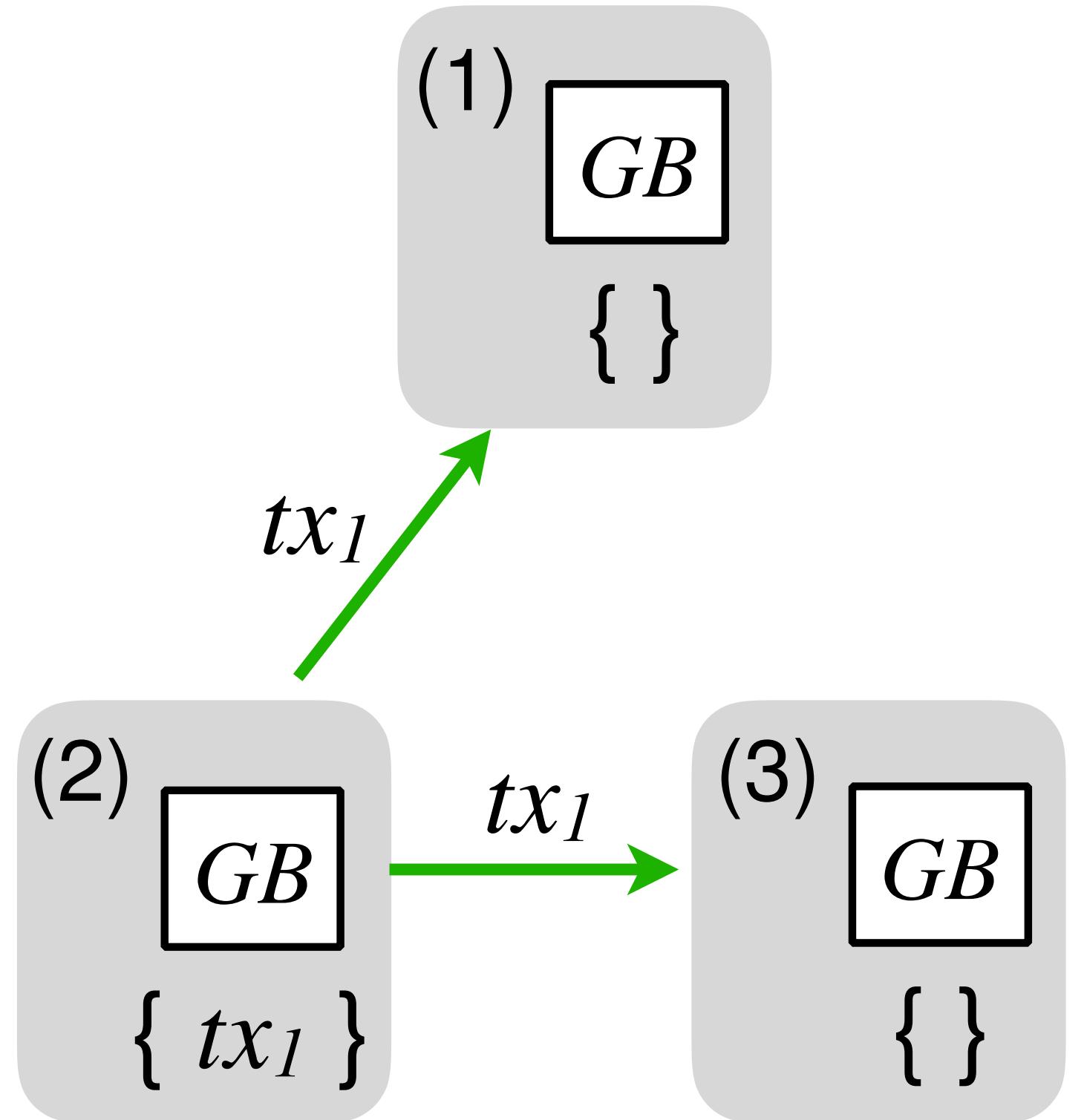
 $tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$

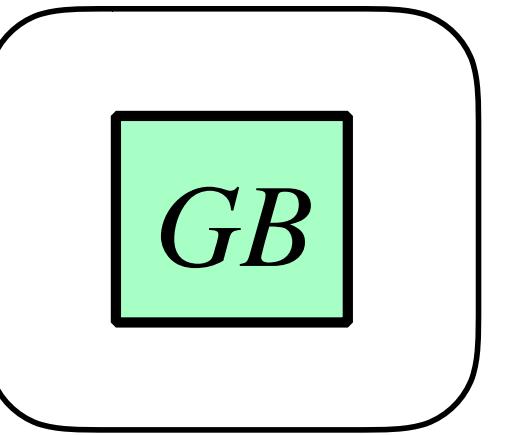
# How blockchain protocols work

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

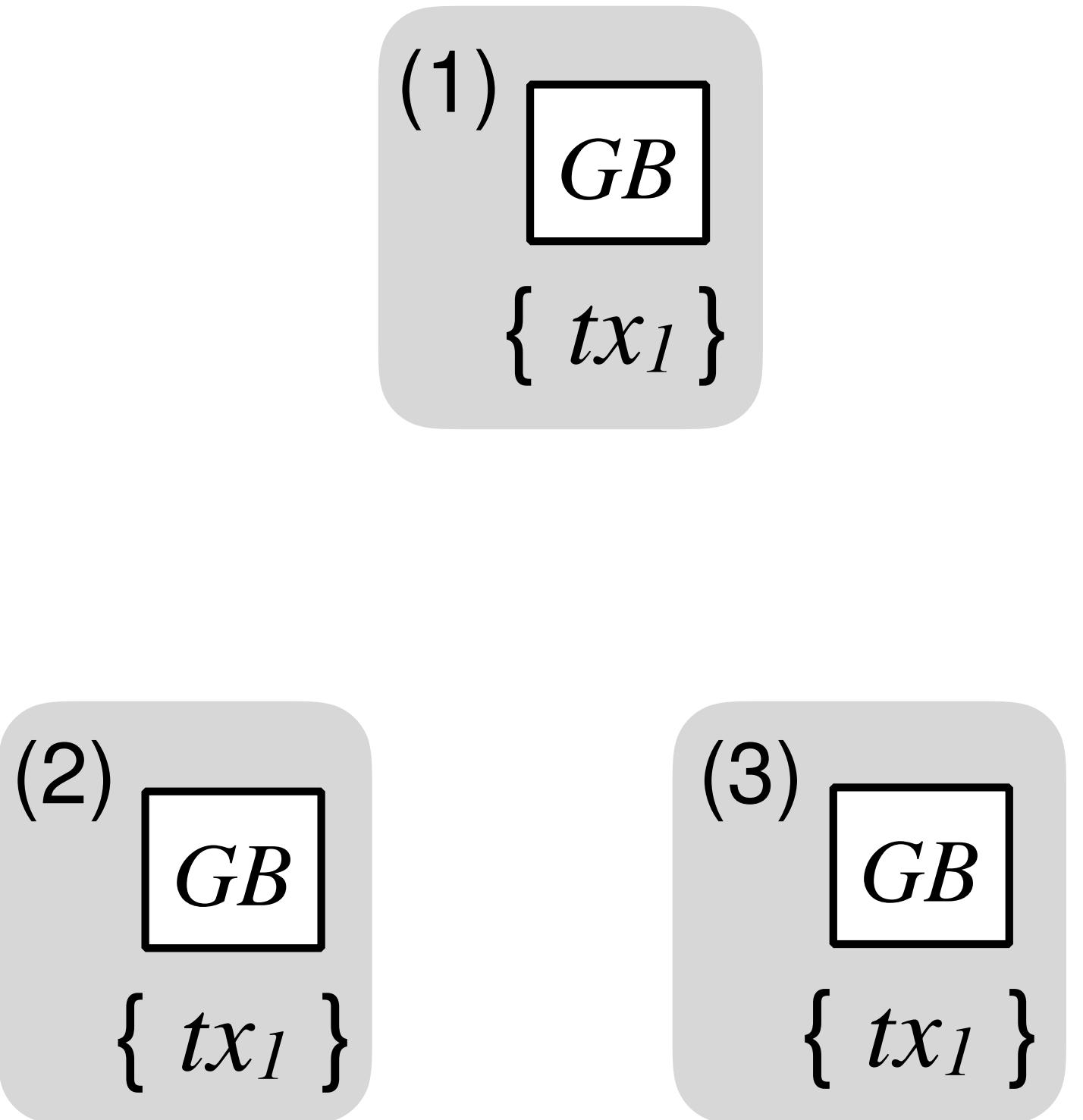


- **distributed**
  - multiple nodes
  - *message-passing* over a network
- all start with same GB

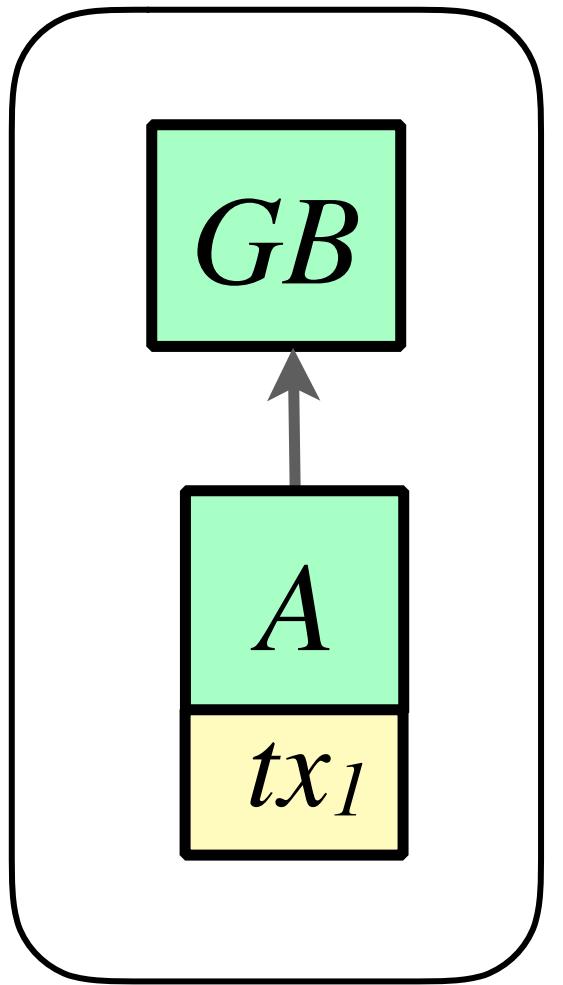
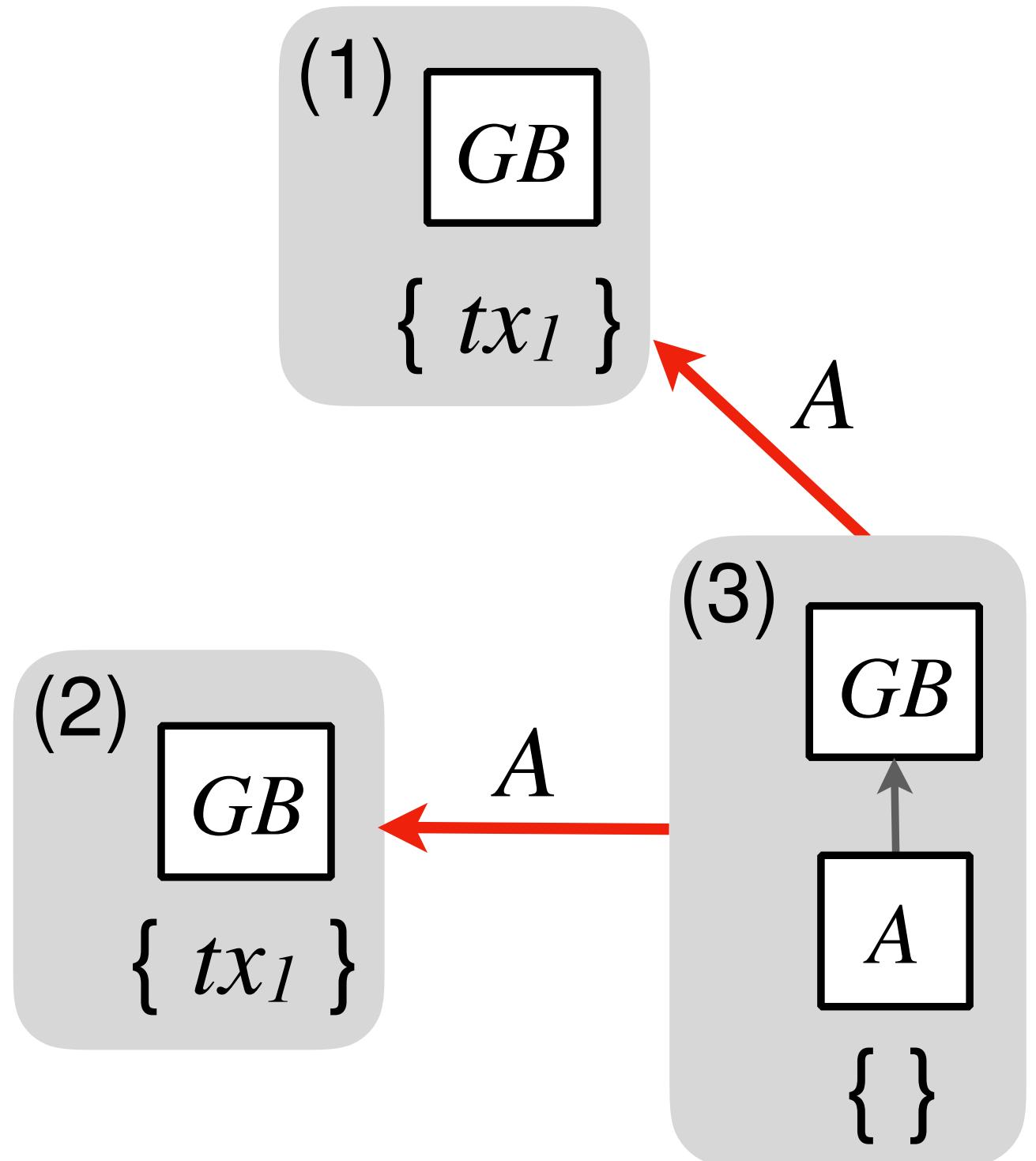




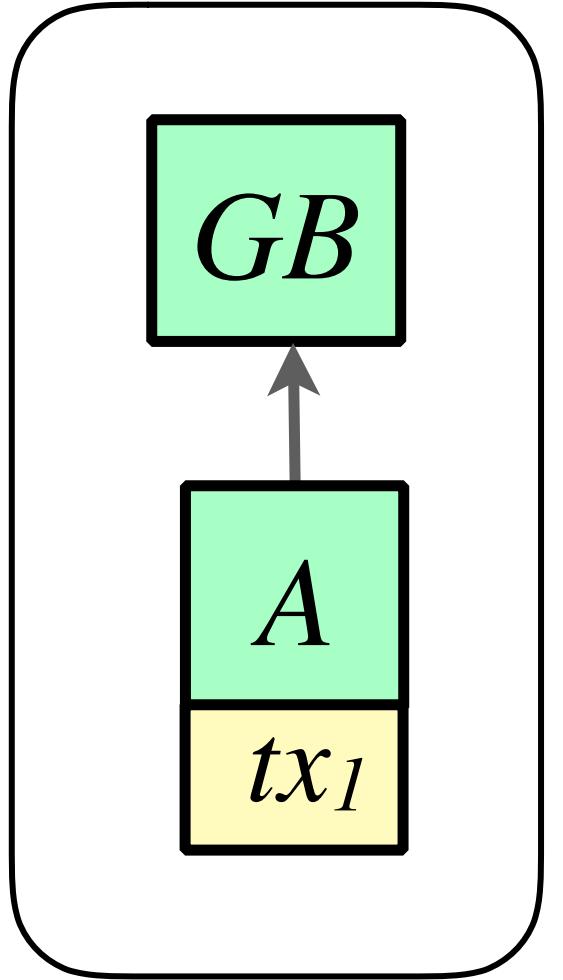
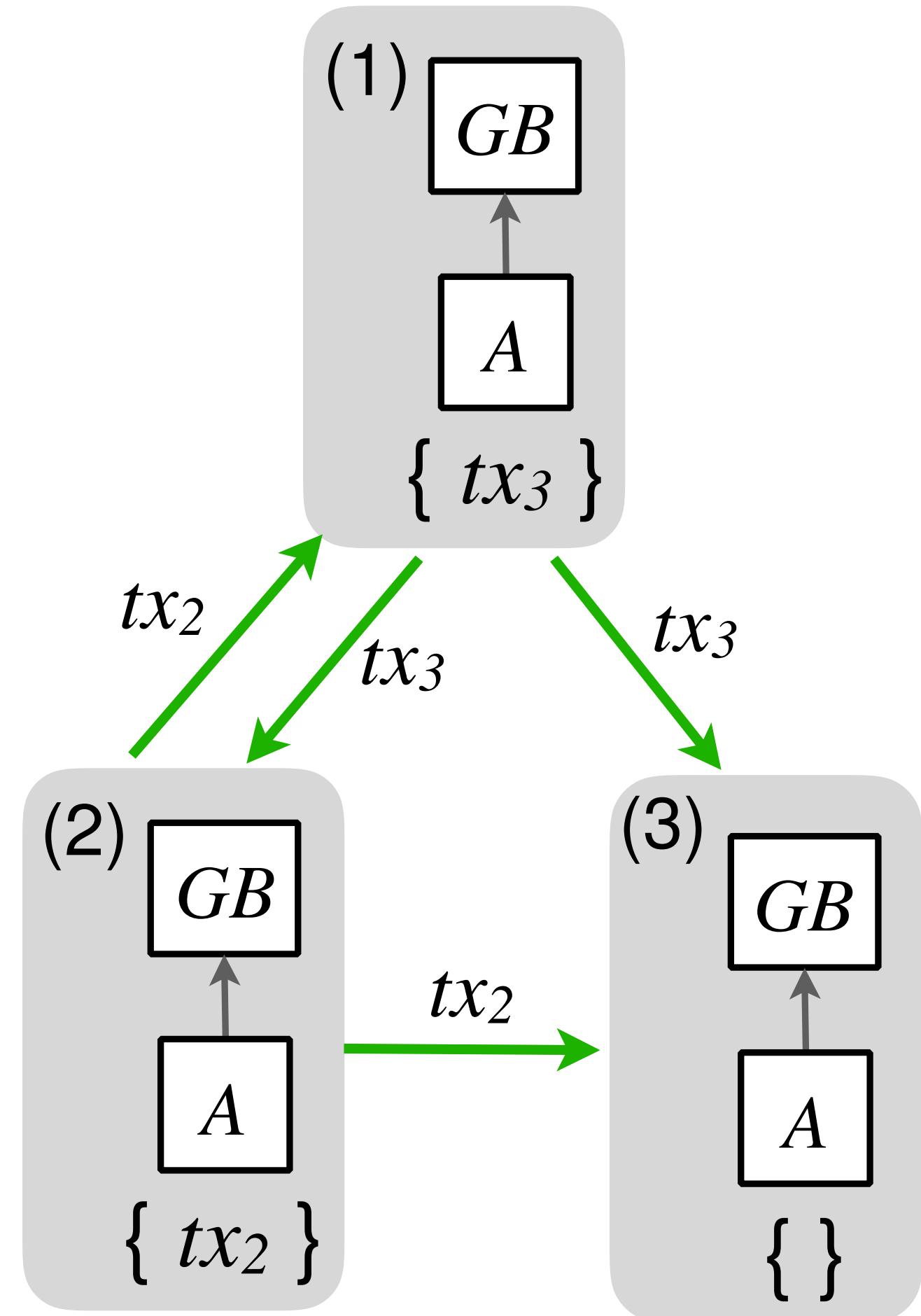
- **distributed**
  - multiple nodes
  - message-passing over a network
- all start with same GB
- have a transaction pool



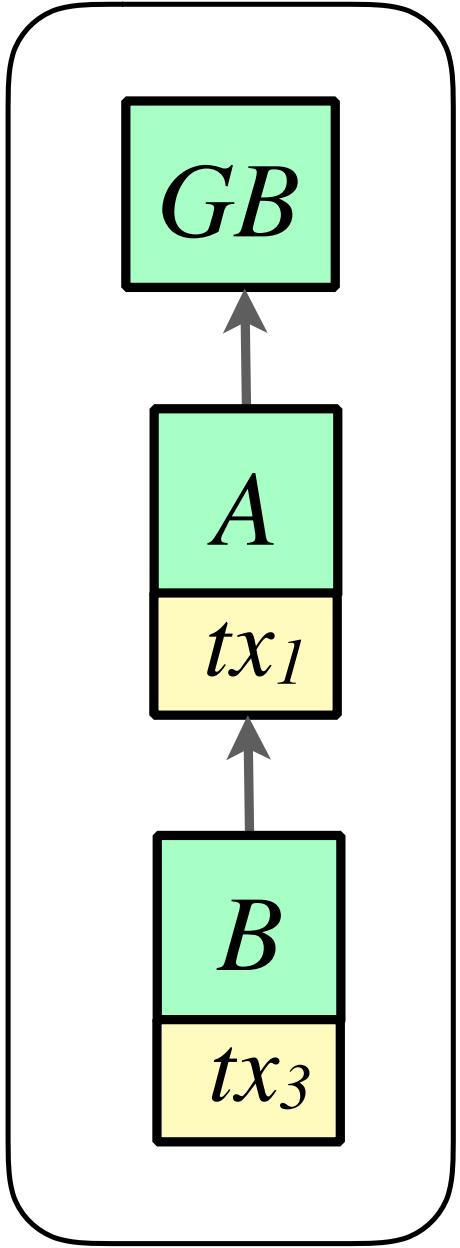
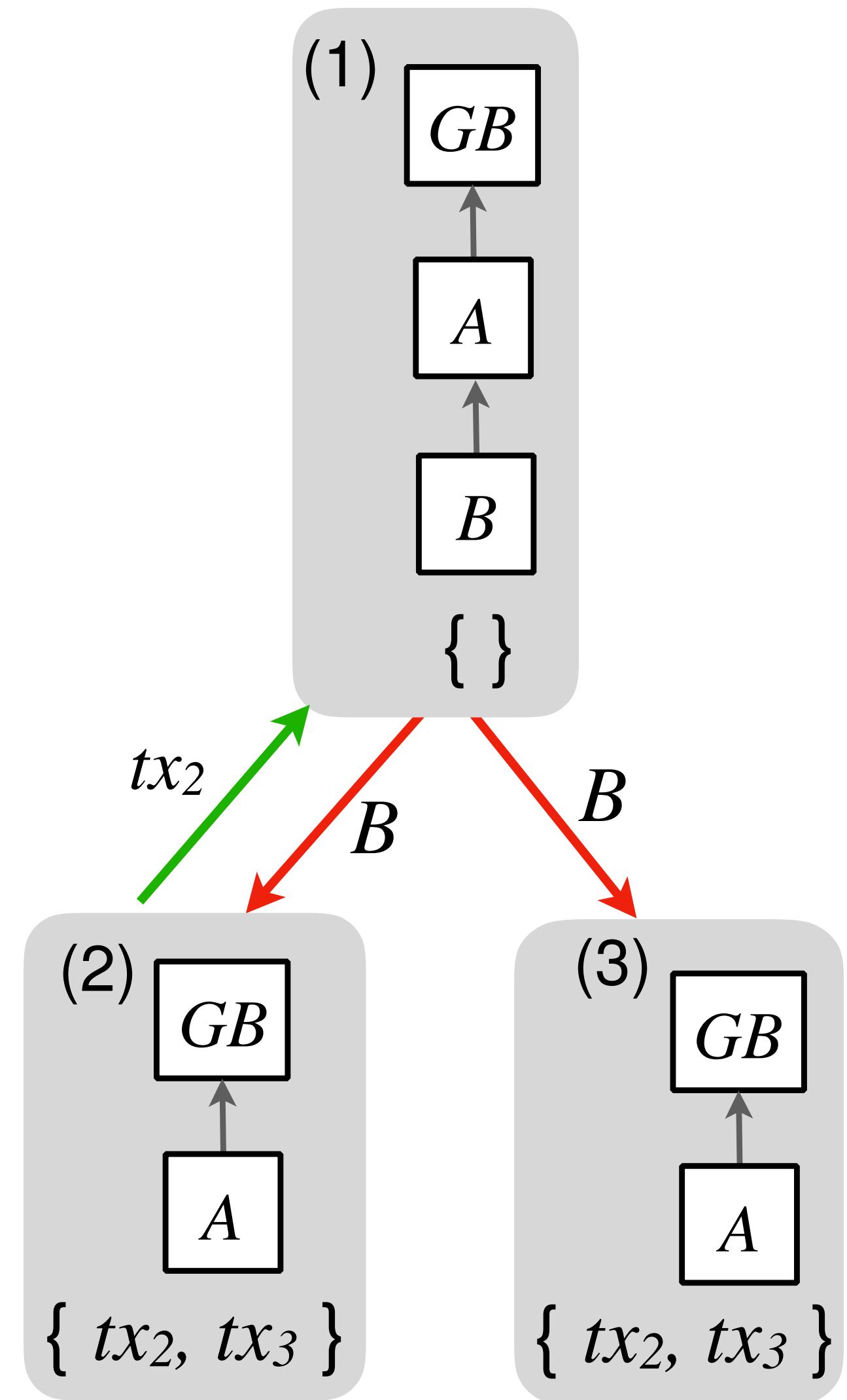
- **distributed**
  - multiple nodes
  - message-passing over a network
- all start with same GB
- have a transaction pool
- can *create (mint) blocks*



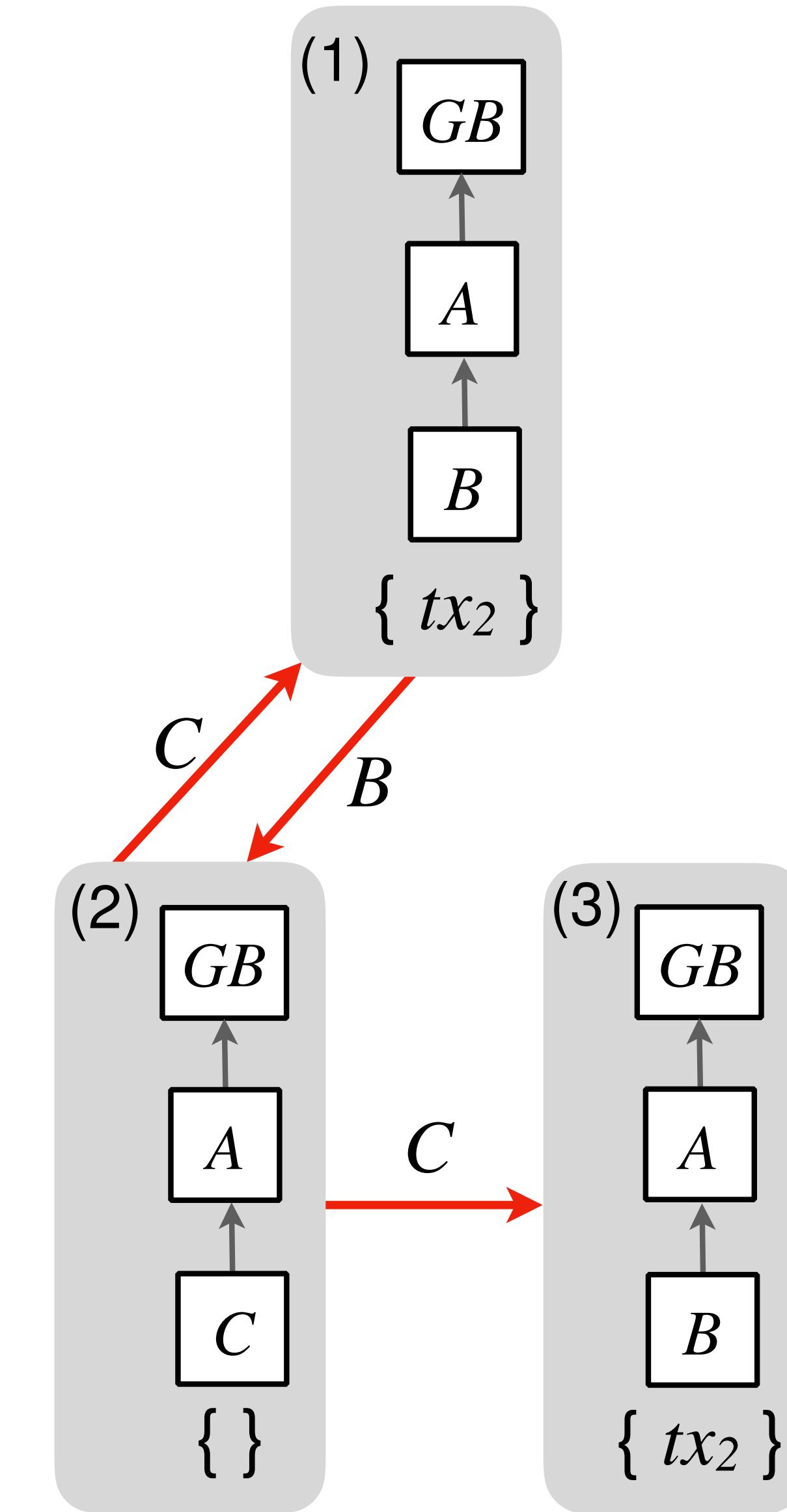
- **distributed**  $\Rightarrow$  concurrent
  - multiple nodes
  - message-passing over a network
- multiple transactions can be issued and propagated *concurrently*



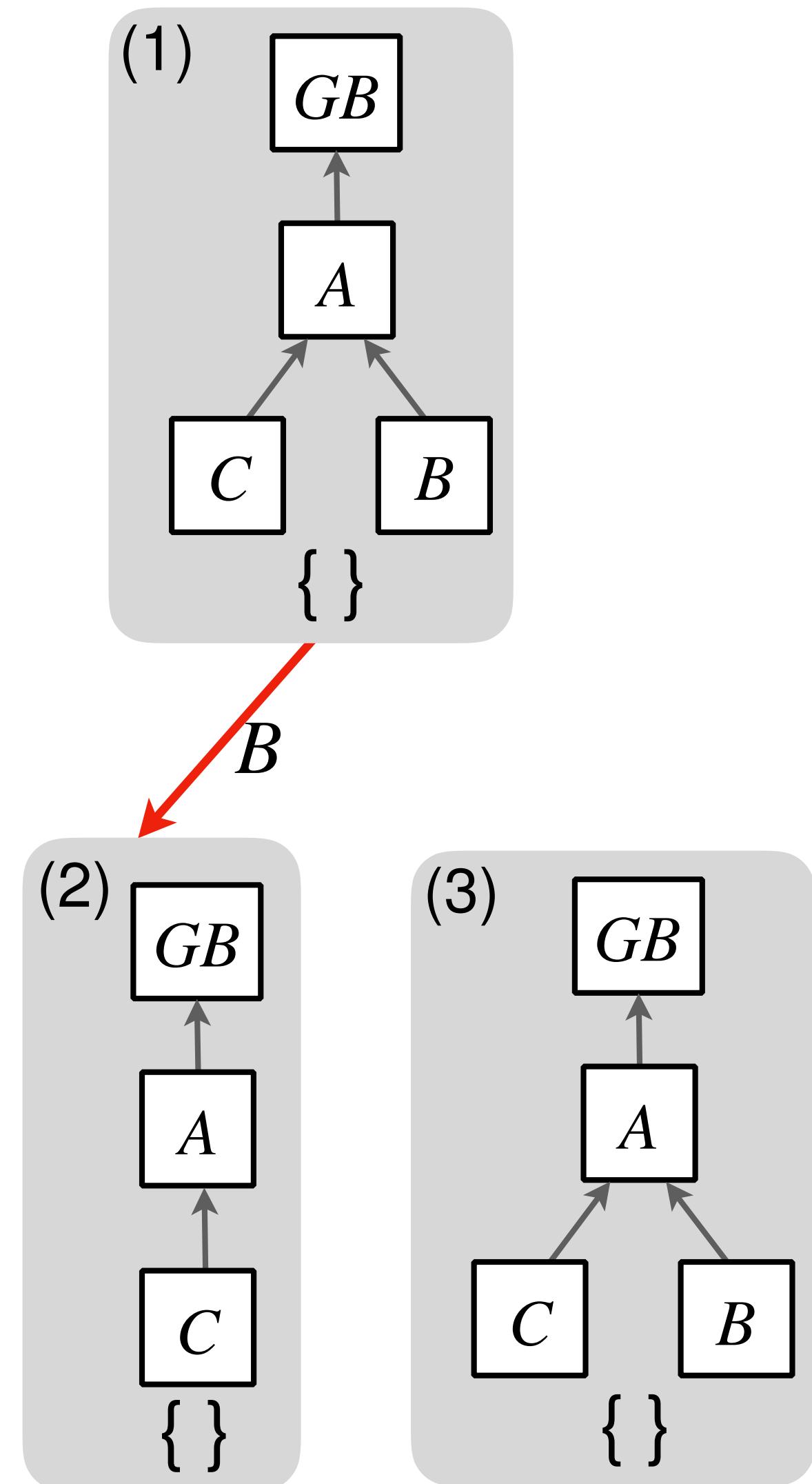
- **distributed**  $\Rightarrow$  concurrent
  - multiple nodes
  - message-passing over a network
- blocks can be created *without full knowledge* of all transactions



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

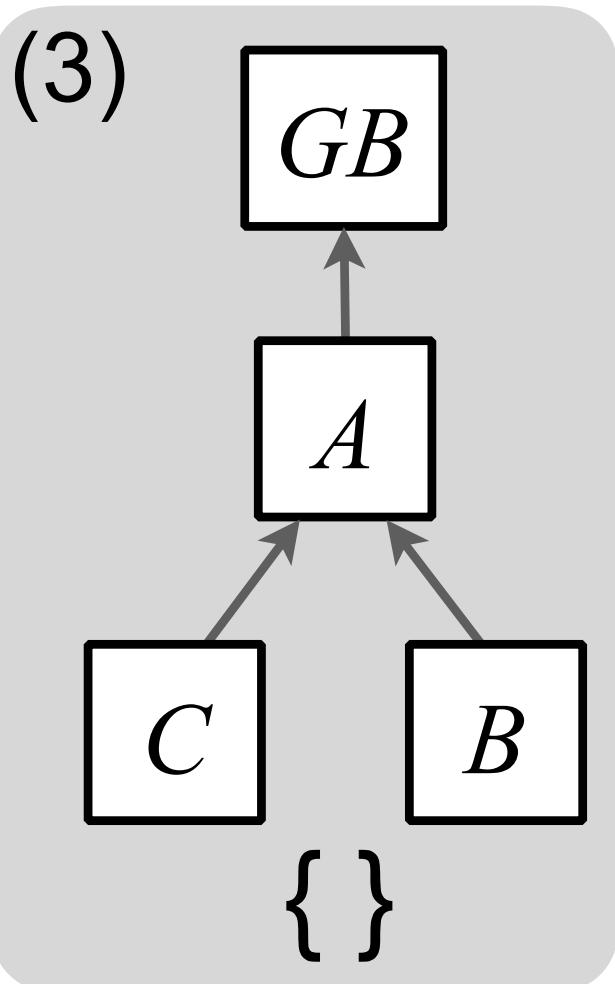
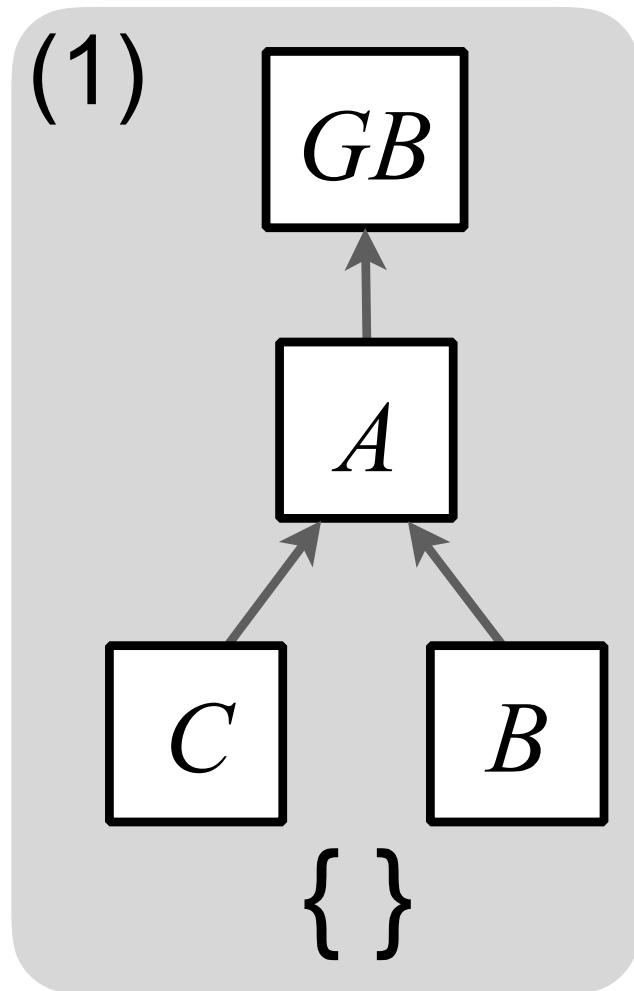


- as block messages propagate, nodes become aware of the *fork*



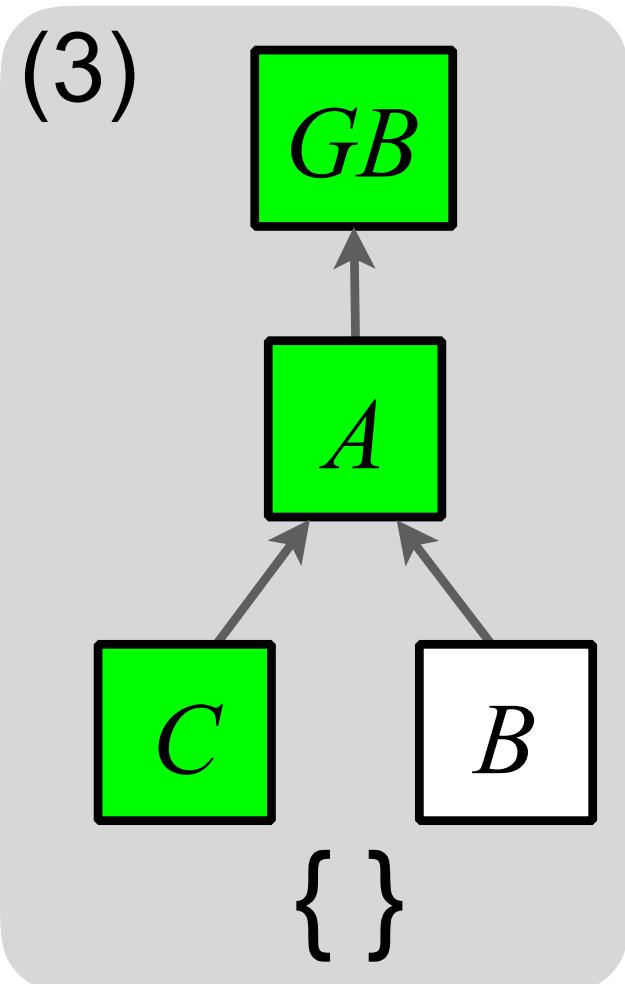
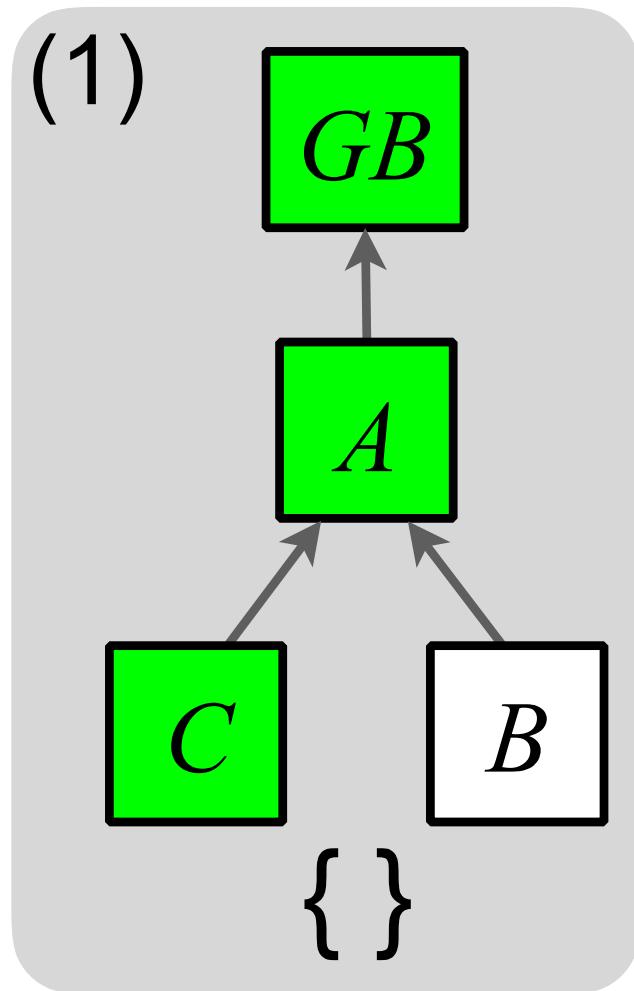
# Problem: need to choose

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



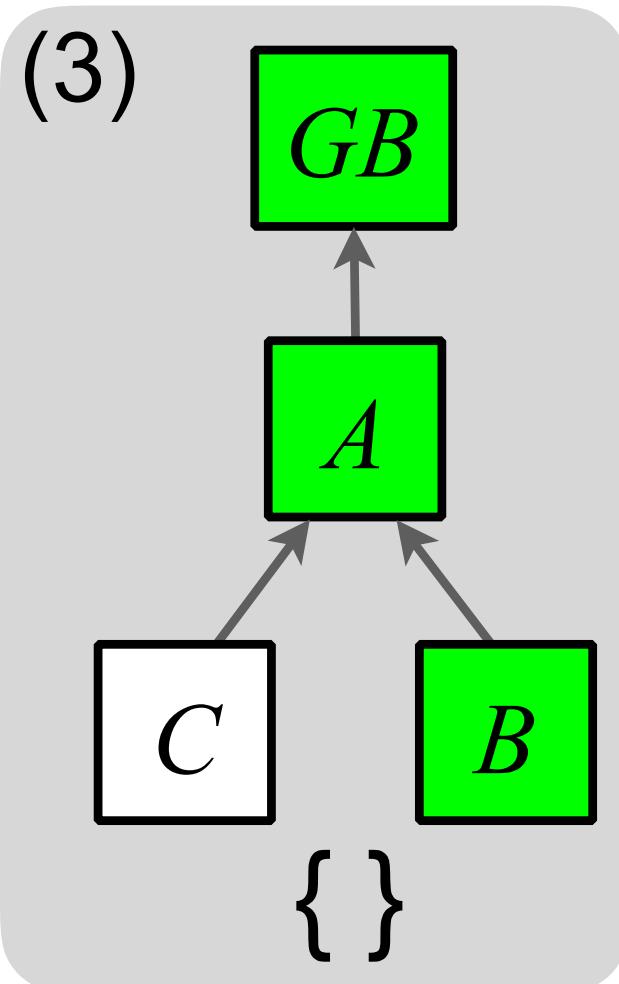
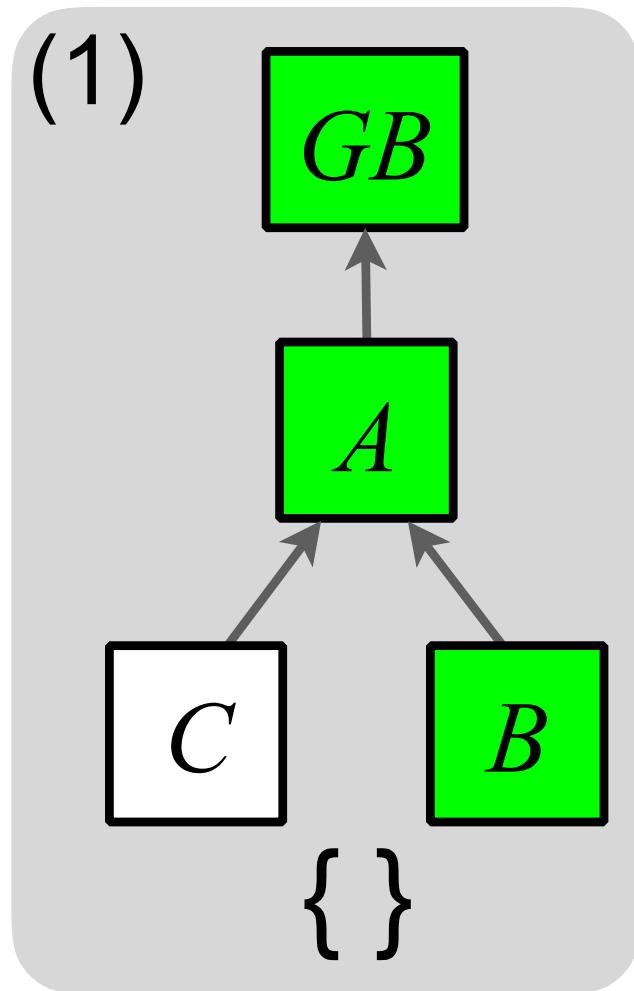
# Problem: need to choose

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



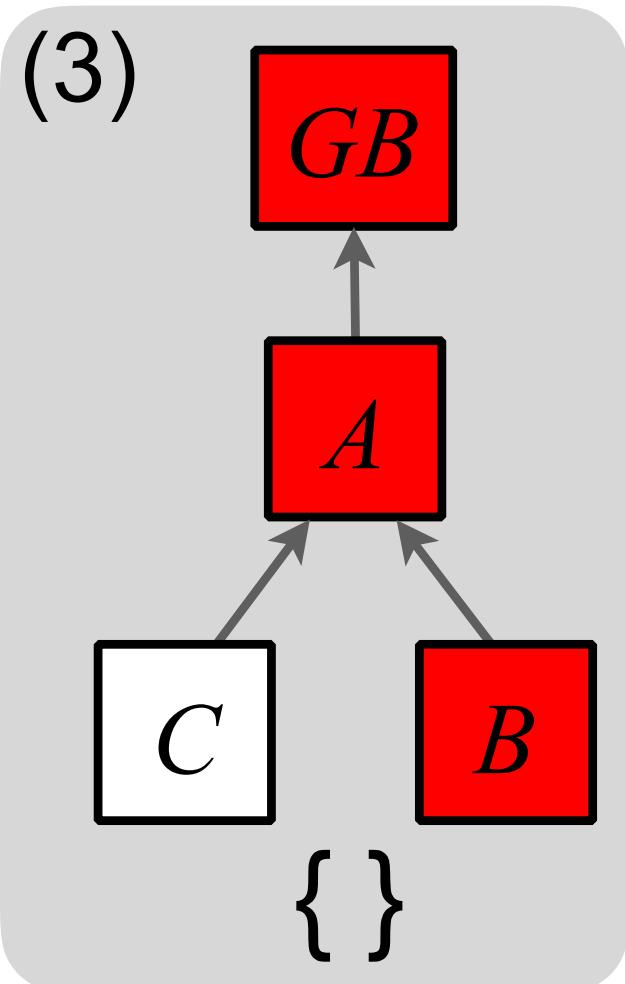
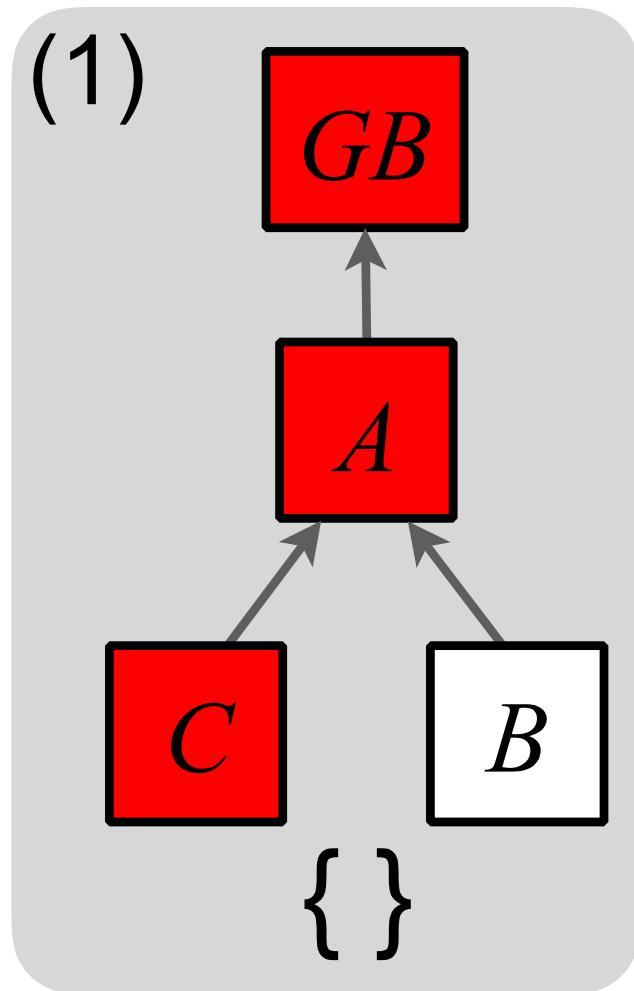
# Problem: need to choose

- 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 (>)

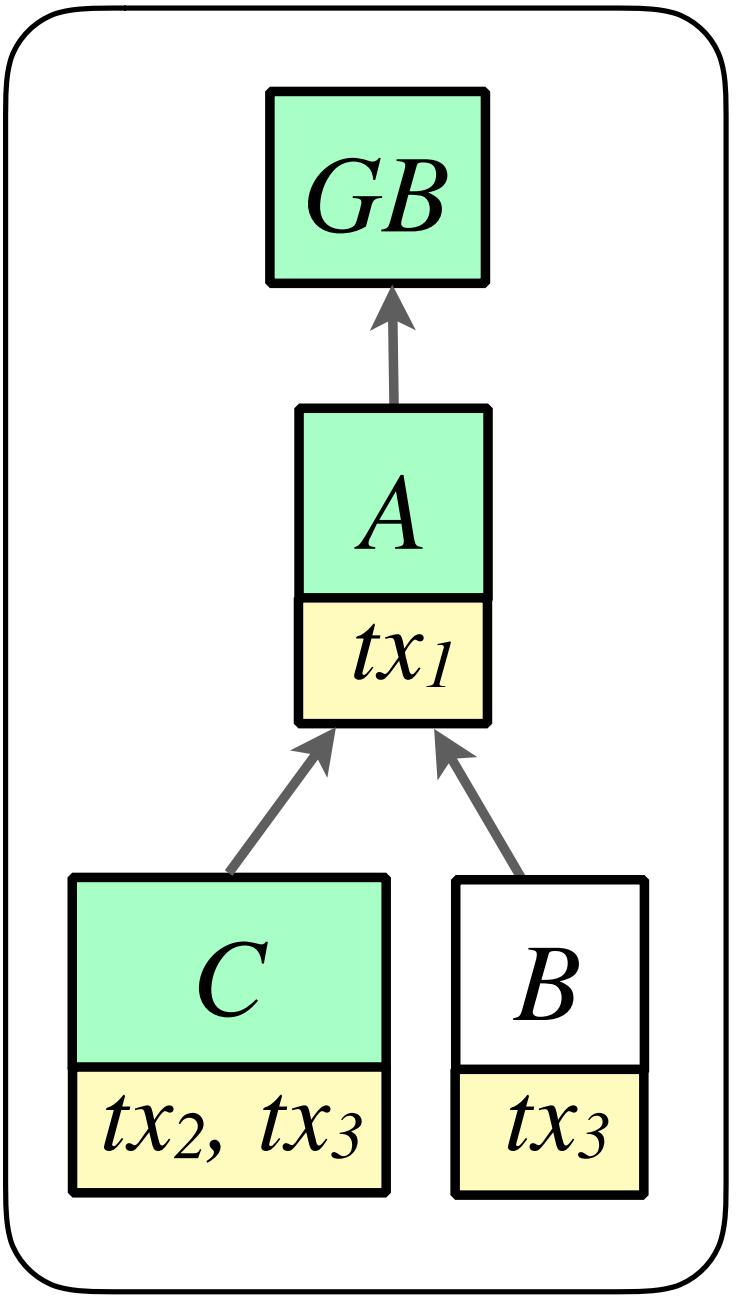
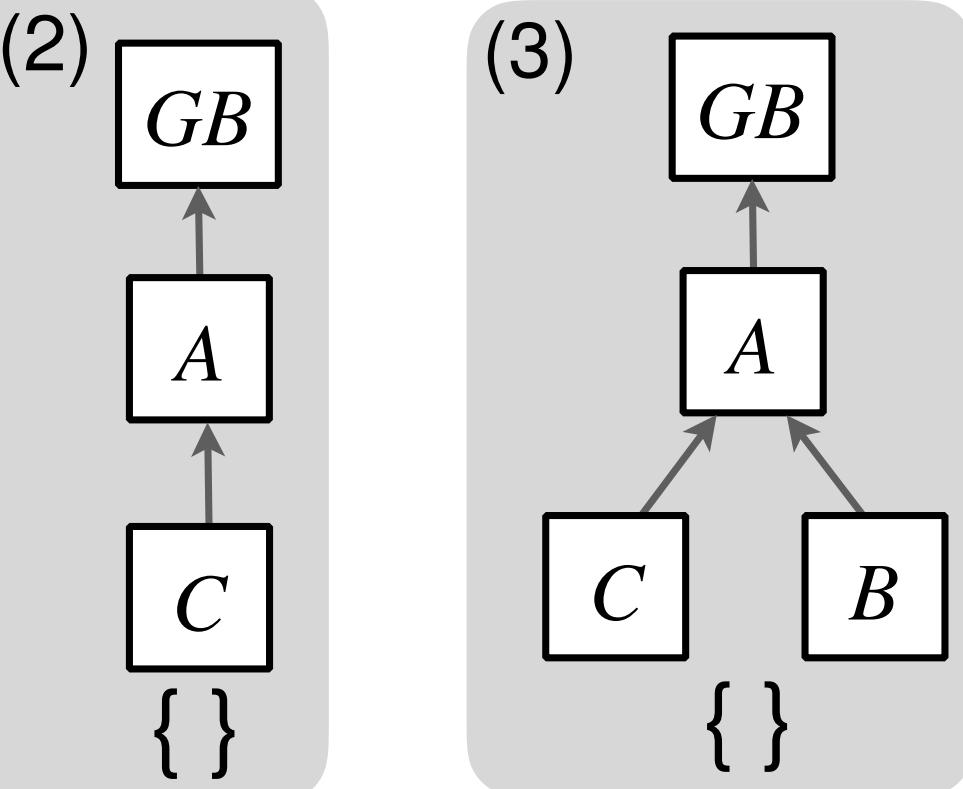
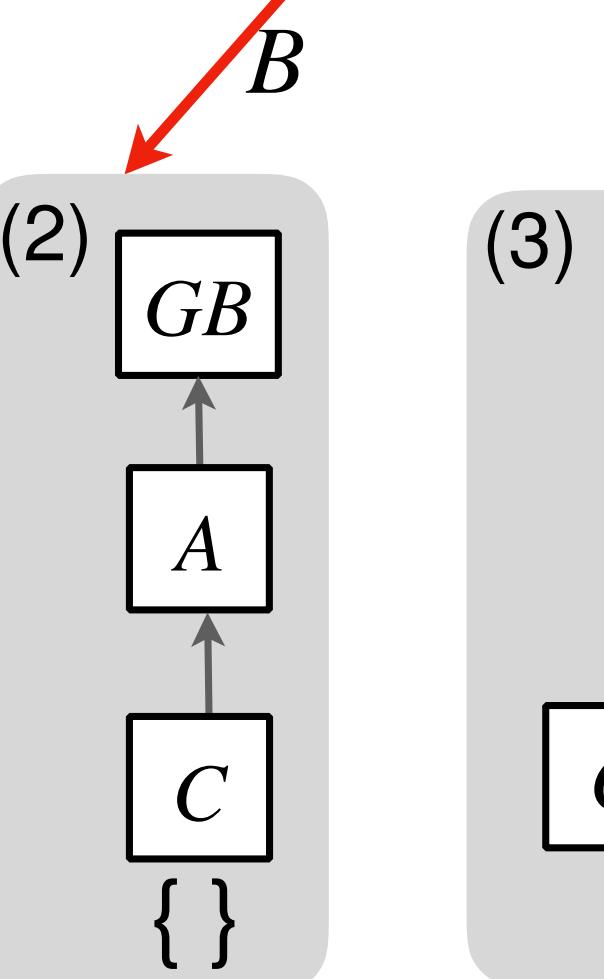
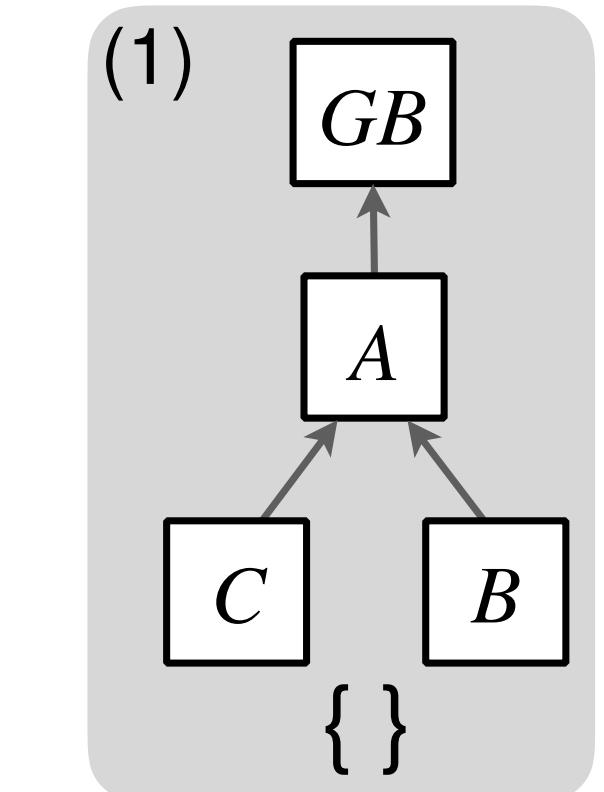
... > [GB, A, C] > ... > [GB, A, B] > ... > [GB, A] > ... > [GB] > ...

**Bitcoin:** FCR based on “most cumulative work”.  
New blocks take *a lot of time and CPU* to create.

# Quiescent consistency

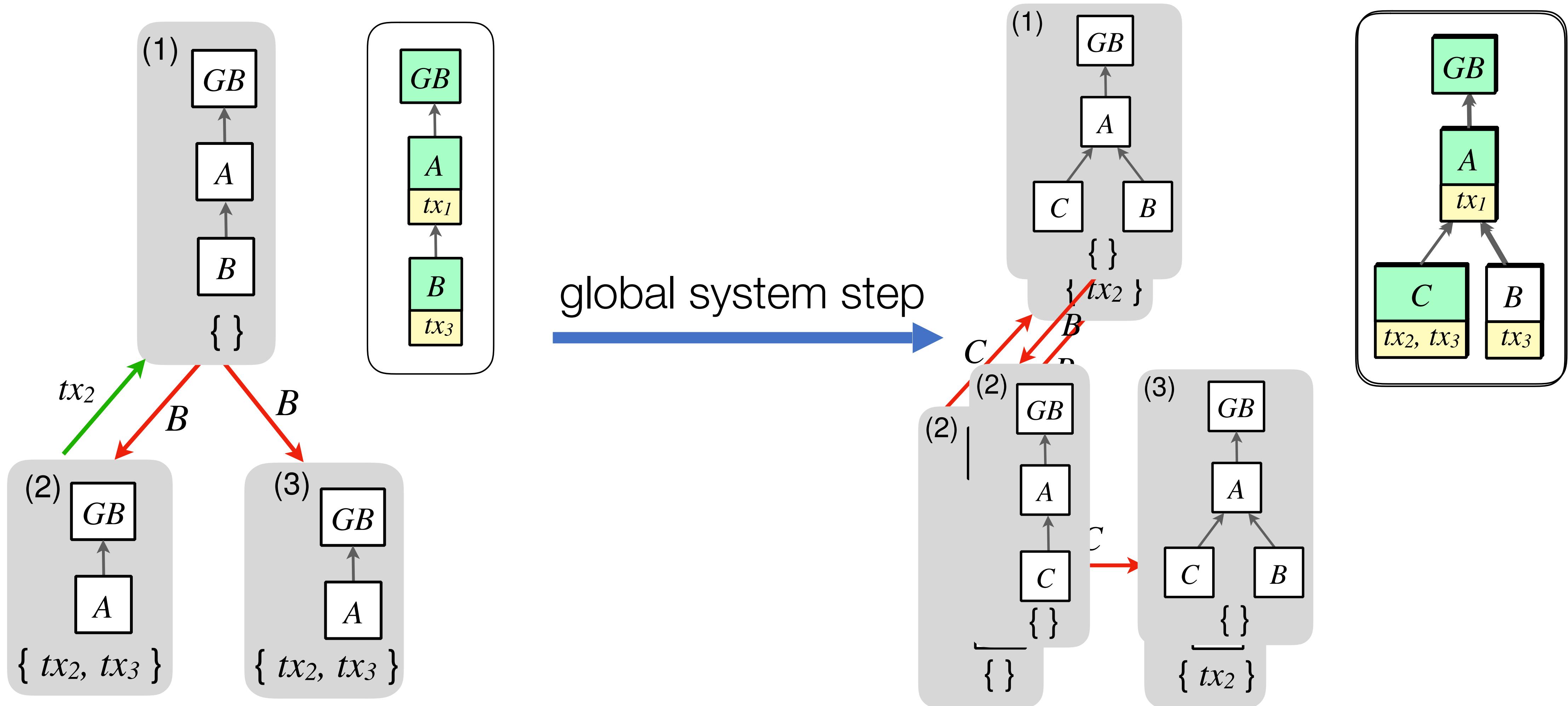
- **distributed**
  - multiple nodes
  - all start with GB
  - message-passing over a network
  - *equipped with same FCR*
- **Quiescent Consistency:**

when *all* block messages have been delivered, *everyone (good)* agrees

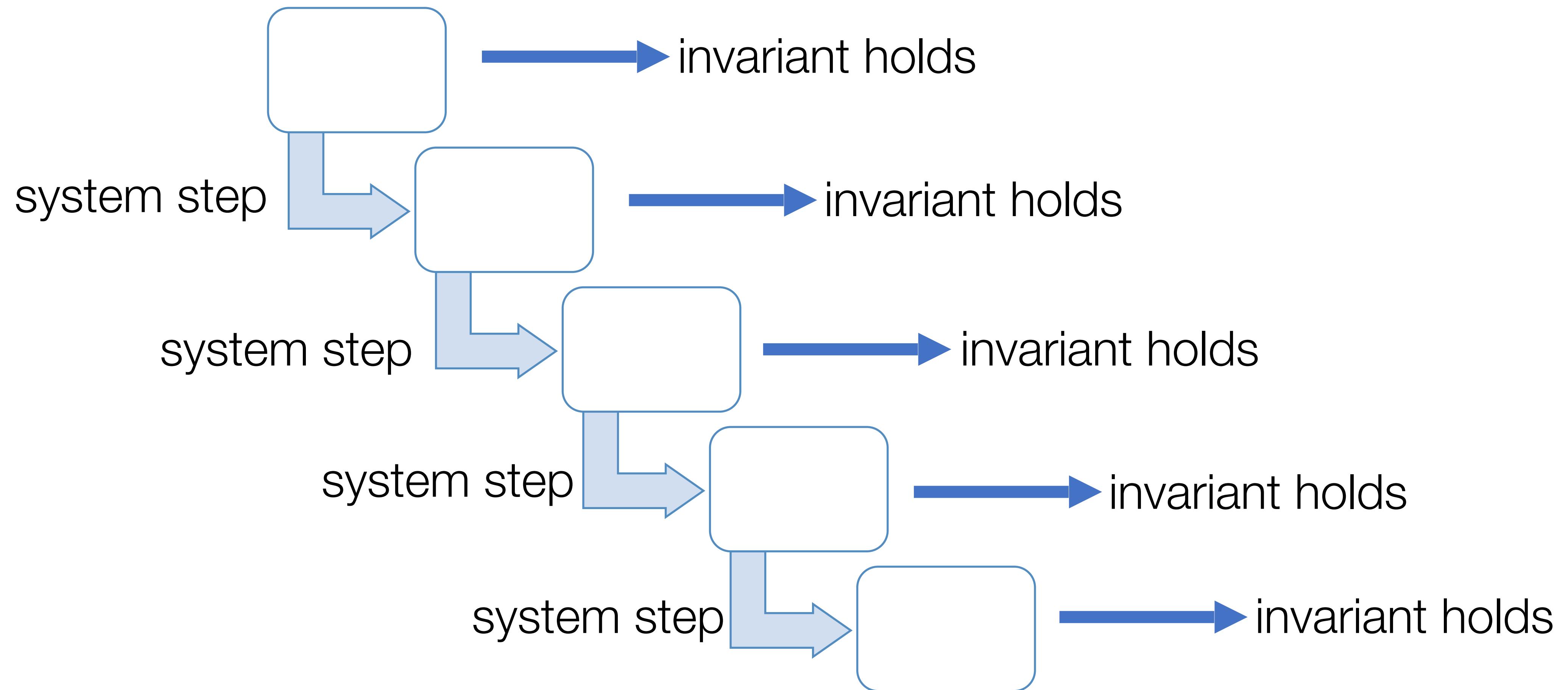


# Why it works

# Invariant: local state + “in-flight” = global



# 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

# 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

# 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

# YSC3248: Parallel, Concurrent and Distributed Programming

Conclusion

# Concurrency is Tricky



# Concurrency is Tricky

- It can be *very confusing*



# 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*
- ... and we want our systems to be *reliable*

# We learned to *understand* concurrency and implement it to *safely* and *efficiently*

- Amdahl's Law
- Safety, Liveness
- Programming with Threads
- Event Orderings and Mutual Exclusion
- Linearizability and Sequential Consistency
- RMW operations, consensus numbers, CAS
- Spin-locks and contention
- Monitors: waiting and signalling
- Design of concurrent objects
- Fine-grained, lazy, and optimistic locking
- Concurrent Elimination, ABA problem
- Skip lists
- Thread pools
- Data race detection
- Futures and Promises
- Data parallelism, splitters and combiners
- Actors and message-passing
- Distributed consensus protocols

# Now you know Kung Fu



# Show me



The final project assignment will be posted right after this lecture