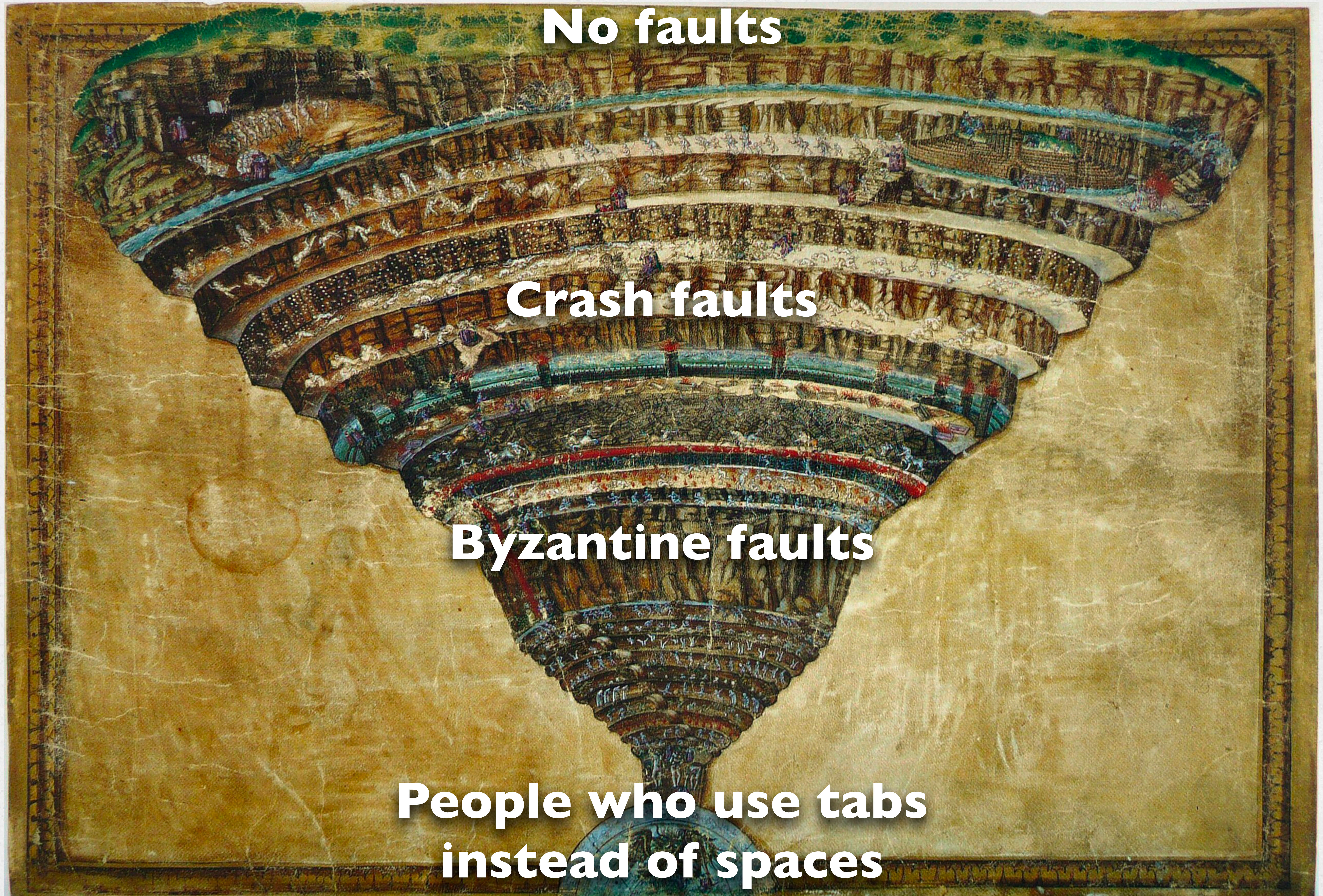
Byzantine Fault Tolerance

Ellis Michael

A Hierarchy of Fault Models



Byzantine Faults

- Also called "general" or "arbitrary" faults.
- Faulty nodes can take any actions. They can send any messages, collude with each other, etc. in an attempt to "trick" the non-faulty nodes and subvert the protocol.
- Why this model?

Strange Things Happen at Scale

- Hardware failures are real and can cause both crashes and aberrant behavior.
- Cosmic rays from outer space can and will randomly flip bits in memory.
- Software bugs are all too common.
- Security vulnerabilities can let attackers into distributed systems.

We'll come back to these at the end of the lecture.

What About Paxos?

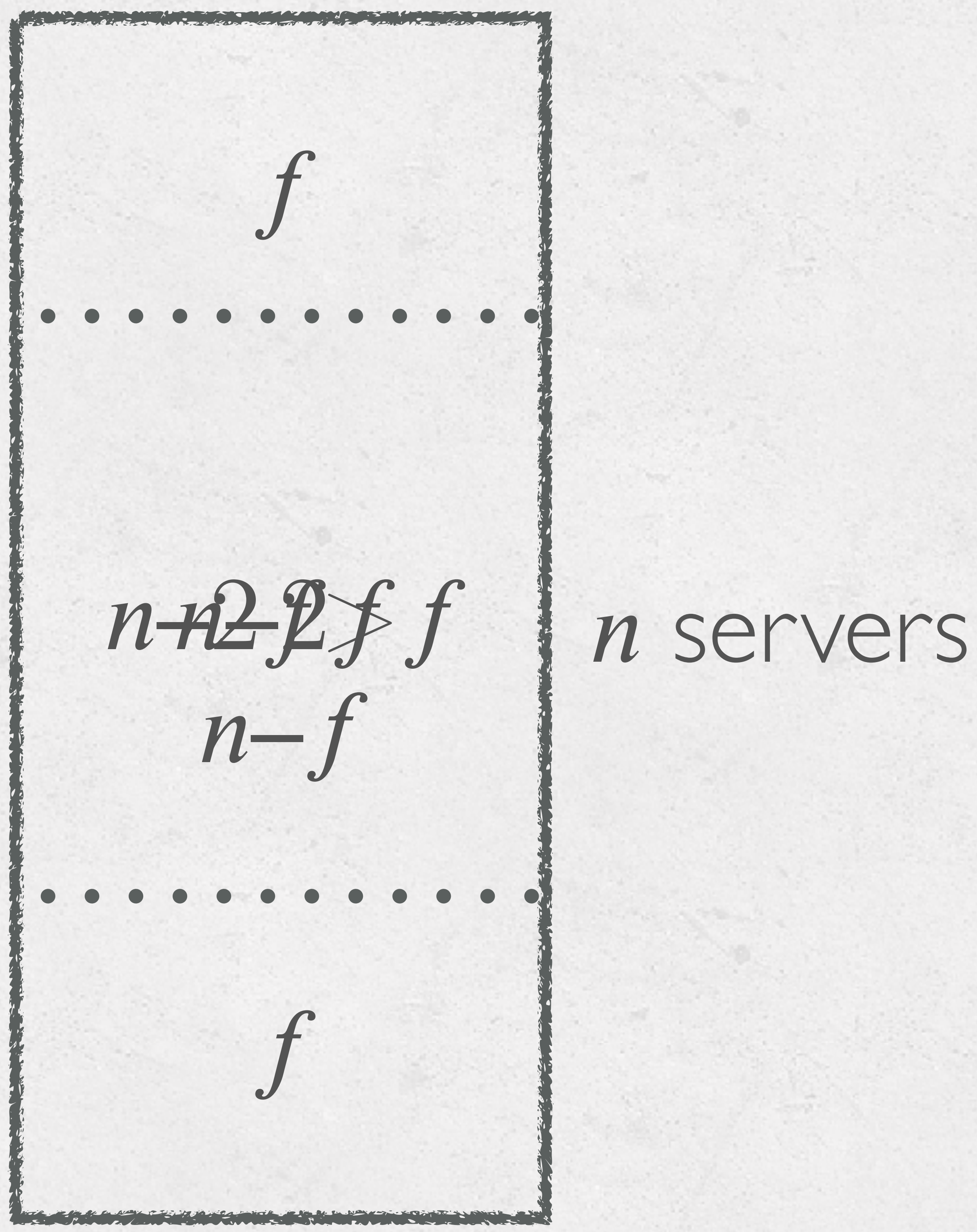
- Paxos tolerates a minority of processing failing by **crashing**.
- What could a malicious replica do to a Paxos deployment?
  - Stop processing requests.
  - A leader could report incorrect results to a client.
  - A follower could acknowledge a proposal and then discard it.
  - A follower could respond to prepare messages without all previously acknowledged commands.
  - A server could continually start new leader elections.
  - ...

Byzantine Quorums

Obviously, if all servers are Byzantine, we can't guarantee anything. How many servers do we need to tolerate f faults?

- In order to make progress ... $n-f$ servers.
- What if two different serv... ms? If they intersect at f or fewe... ...ood.
- Therefore, we need at least $3f+1$ servers. Any two quorums of $2f+1 = n-f$ will intersect at at least one non-faulty server.

Provable lower bound.



Setup

- $n=3f+1$ servers, f of which can be faulty. Unlimited clients.
- We assume public-key infrastructure. Servers and clients can sign messages and verify signatures. Signatures aren't forgeable.
  - We denote message m with $\langle m \rangle$, and message m signed by p as $\langle m \rangle_p$.
- Servers also have access to a digest function (cryptographic hash) on messages, $D(m)$, which we assume is collision-resistant.
- The attacker controls f faulty servers and knows the protocol the other servers are running. The attacker also has control over the network and can delay and reorder messages to all nodes.

Goal

The goal, as in Paxos, is state-machine replication.

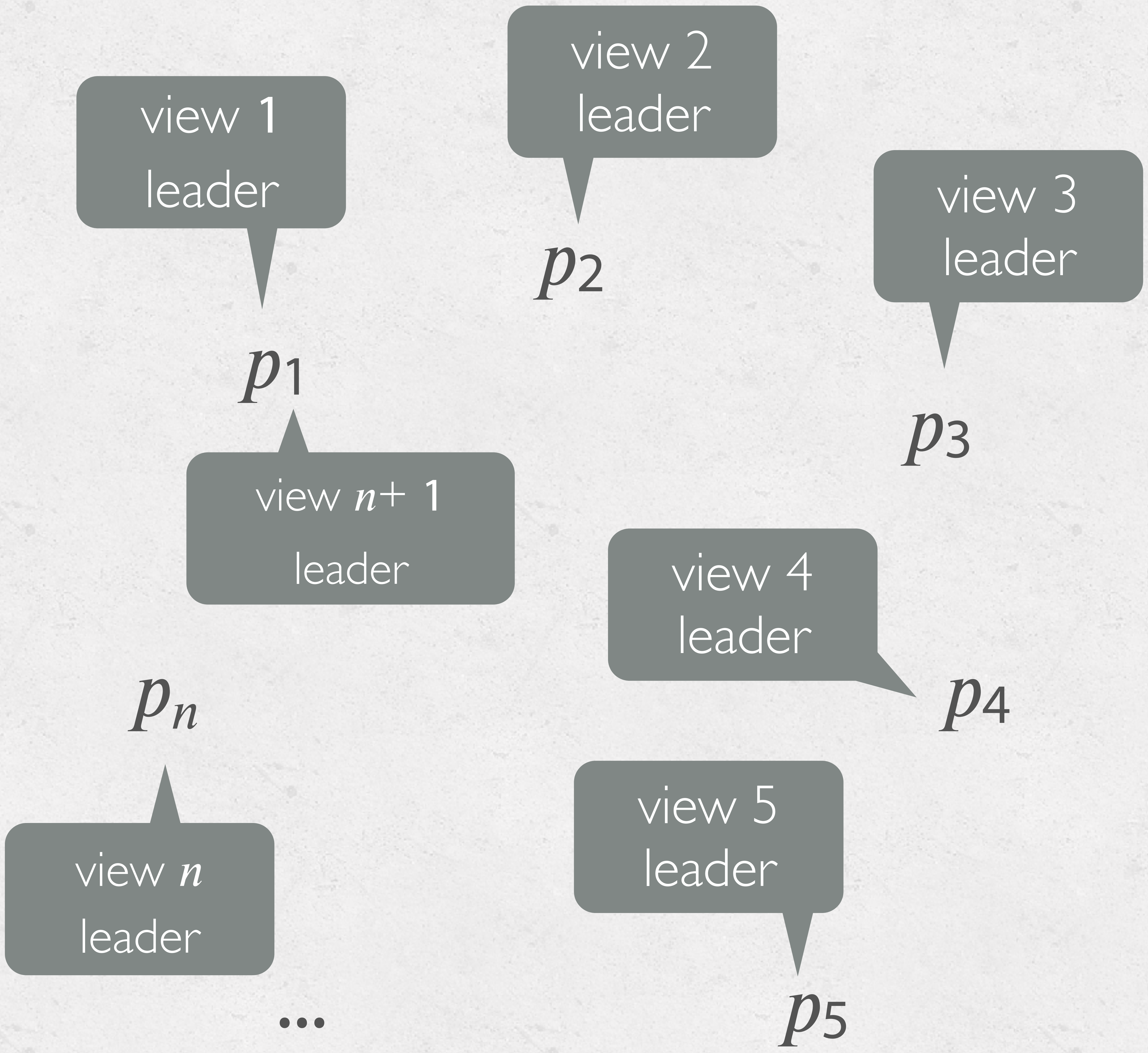
We want to guarantee safety when there are f or fewer failures (or an unlimited number of crash failures) and liveness during periods of synchrony.

Easy, right?

PBFT: The Basic Idea

Practical Byzantine Fault Tolerance (PBFT) is leader-based, just like Paxos. But it more closely resembles Viewstamped Replication [Oki and Liskov '88].

- The system progresses through a series of numbered **views**. There is a single leader associated with each view.
- The clients will send their commands to the leader.
- The leader assigns the command a sequence number (slot number) and forwards to the followers.
- The protocol ensures that this decision is permanently fixed; then they respond to the client.



What's The Worst That Could Happen?

- The leader could be faulty.
  - It could assign c the same sequence numb
  - It could try to send the wrong result to the client.
  - It could ignore the clients altogether.
- The followers co about the commands they

Clients wait for $f+1$ matching replies.

Followers can replace a misbehaving leader with a **view change**.

What About Faulty Clients?

- We assume that there is some existing way for clients to authenticate themselves with the system.
- Access controls can be used to restrict what each client is allowed to do.
- System administrators (or the system itself) can revoke access for faulty clients.

Papers, Please

- Servers don't take each others' word for anything. They require proof.
- In order to verify that a client's command is legitimate, they need the signed message from the client (or proof thereof).
- All other steps in the system are taken only after receiving signed messages from a **quorum of $2f+1$** servers. Servers can also collect these messages into **certificates** they can use to prove to each other the legitimacy of certain steps.

PROTOCOL OVERVIEW

Three sub-protocols:

1. **Normal operations**

   Phase 1: Pre-prepare

   Phase 2: Prepare

   Phase 3: Commit

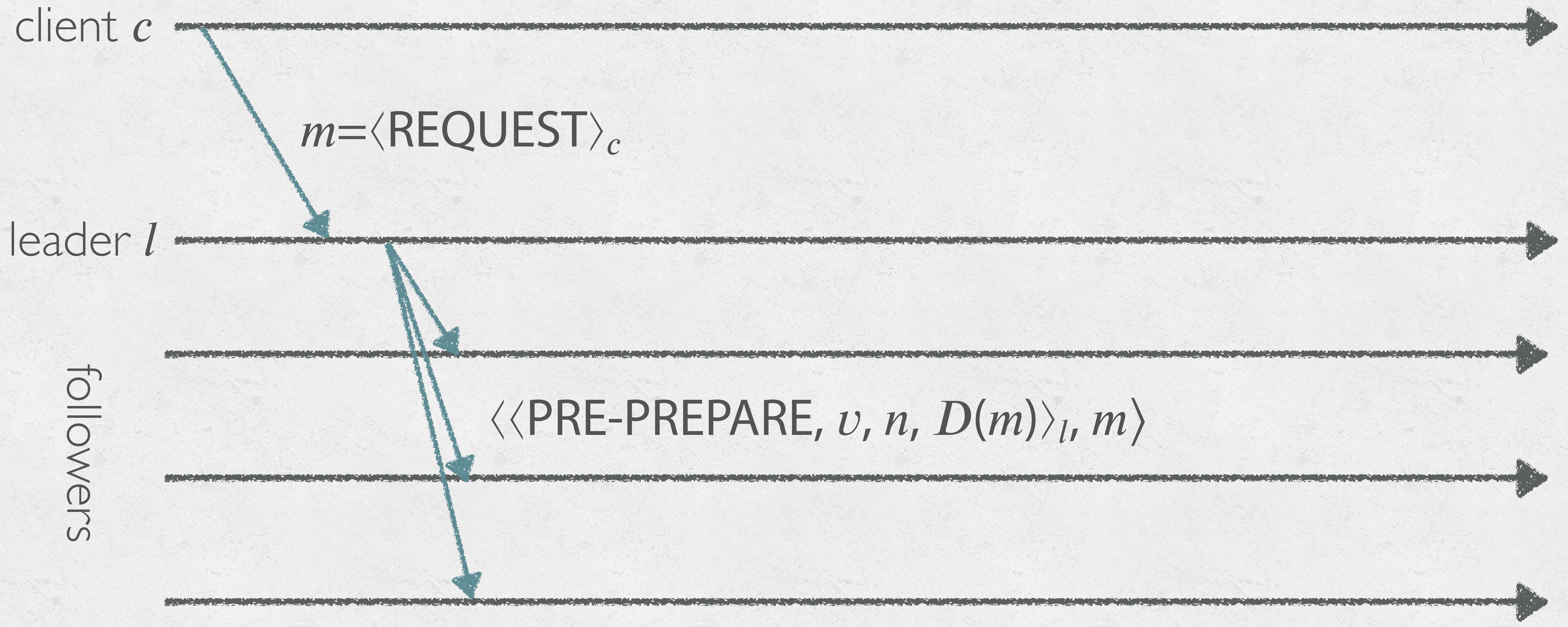
2. **View change**

3. **Garbage collection**

Server state:

- Current view
- State machine checkpoint
- Current state machine state
- Log of all not garbage collected messages

Normal Operations (I)



Accepting Pre-Prepares

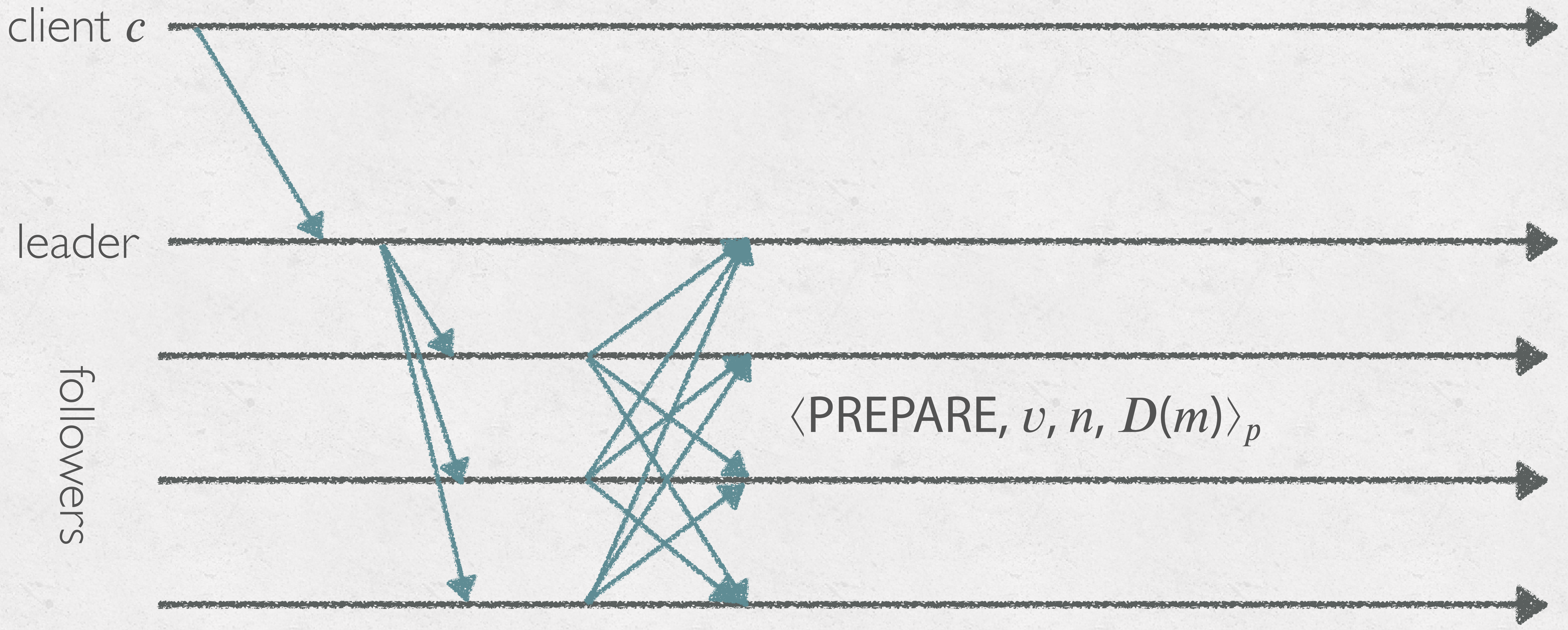
The leader sends $\langle\langle \text{PRE-PREPARE}, v, n, D(m)\rangle_l, m\rangle$ to the followers.

- v is the view number.
- n is the sequence number assigned by the leader.
- $D(m)$ is a digest of the message (to reduce amount of public key crypto).

A follower accepts the PRE-PREPARE if:

- The client request is valid.
- The follower is in view v.
- The follower hasn't accepted a different PRE-PREPARE for the same sequence number in the same view.
- The sequence number isn't too far ahead (to prevent sequence numbers from getting unnecessarily large).

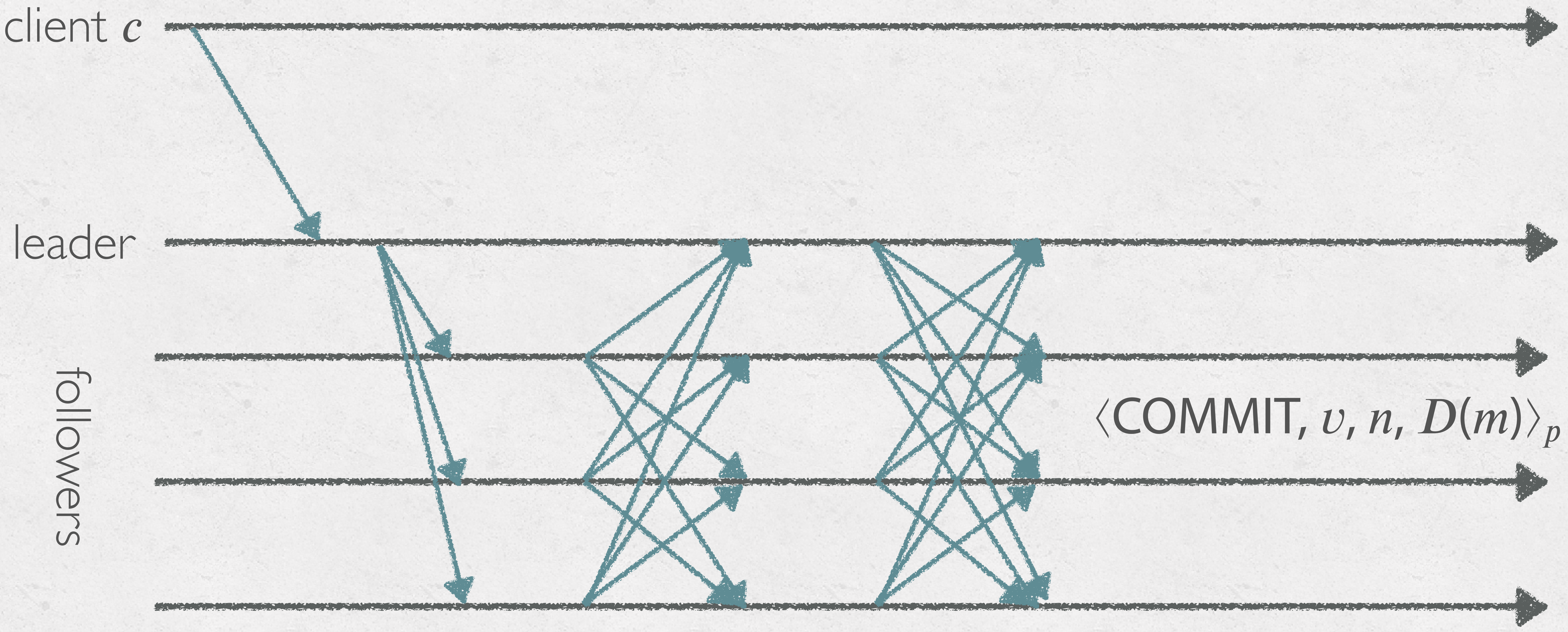
Normal Operations (II)



Prepare Certificates

- Once followers accept the PRE-PREPARE, they broadcast (signed) PREPARE messages.
- Once a server has received $2f$ matching PREPAREs and the associated PRE-PREPARE, it has a **Prepare Certificate**.
- Because quorums intersect at at least one honest server, and honest servers don't prepare different commands in the same slot, *no two prepare certificates ever exist for the same view and same sequence number and different commands*.
- However, a single server having a prepare certificate is not enough. What about view changes? The new leader might not get the Prepare Certificate, might not have enough information to pick the correct command in the new view.

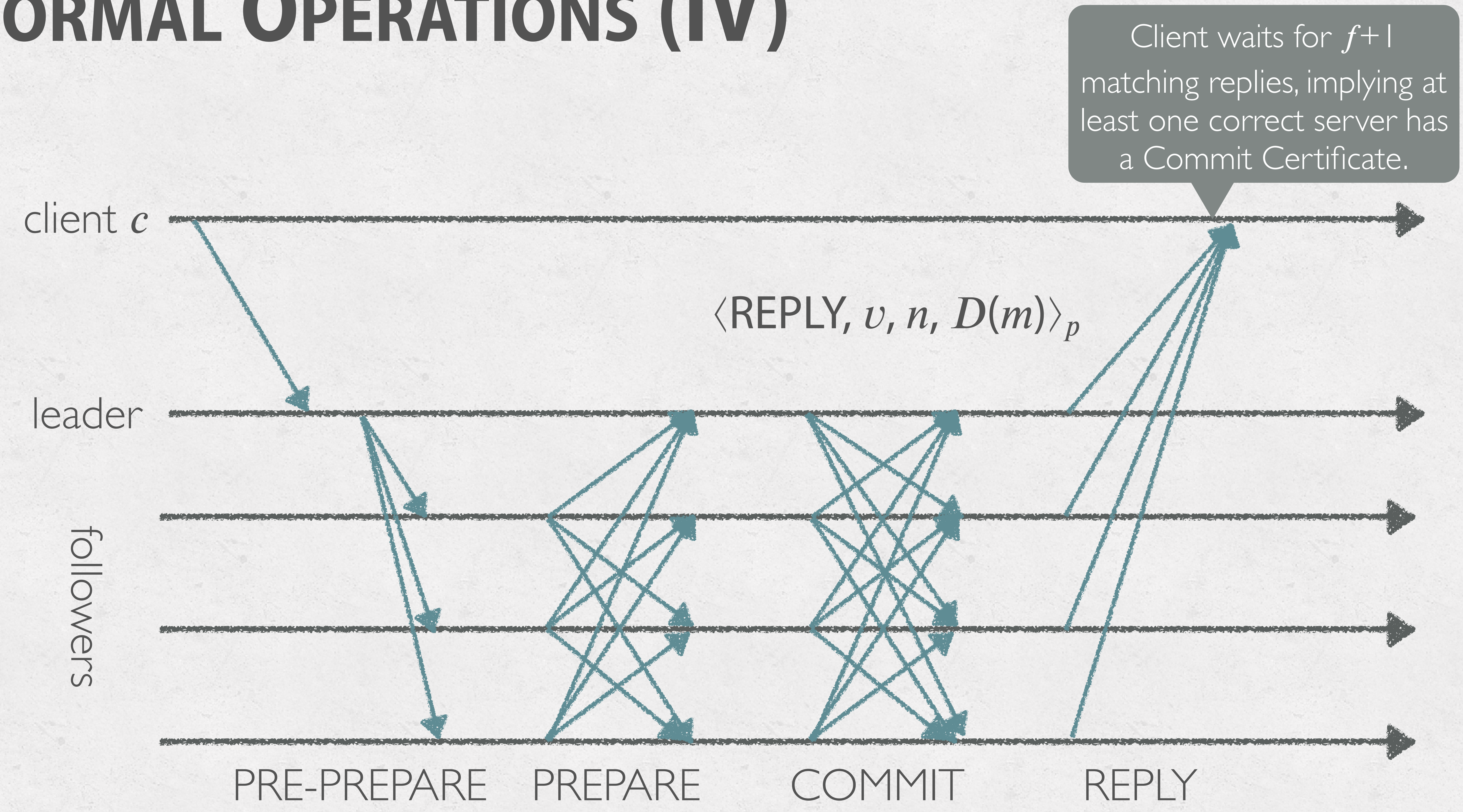
NORMAL OPERATIONS (III)



Commit Certificates

- Once a server has a Prepare Certificate, it broadcasts a COMMIT message.
- Once a server has $2f$+1 matching COMMITs (and the associated client message), it has a **Commit Certificate**.
- A commit certificate proves that every quorum of $2f$+1 servers has at least one non-faulty node with a Prepare Certificate. This command is now stable and will be fixed in the same slot future view changes.
- The server can then execute the command (provided it executed all previous commands) and reply to the client.

Normal Operations (IV)



VIEW CHANGE

- Followers monitor the leader. If the leader stops responding to pings or does anything shady, they start a view change.
- First, the follower sends $\langle \text{VIEW-CHANGE}, v{+}1, \mathscr{P} \rangle_p$ to the leader of view v+1 and $\langle \text{VIEW-CHANGE}, v{+}1 \rangle_p$ to the other followers. The follower stops accepting messages for the old view.
  - $\mathscr{P}$ is the set of all Prepare Certificates (or Commit Certificates) the follower has received.
- Other followers join in the view change when they receive f+1 VIEW-CHANGE messages.

Starting a New View

Once the new leader receives $2f$ VIEW-CHANGE messages from the other servers, it broadcasts $\langle$NEW-VIEW, v+1, $\mathscr{V}$, $\mathscr{O}\rangle_p$

- $\mathscr{V}$ is the set of VIEW-CHANGE messages it received.
- $\mathscr{O}$ is a set of PRE-PREPARES in the new view, one for every sequence number less than or equal to the largest sequence number seen in a Prepare Certificate in a VIEW-CHANGE message. If there is a Prepare Certificate for that sequence number, the PRE-PREPARE is for that command. Otherwise, the leader pre-prepares a no-op.

Followers can independently verify that the view was started correctly from the set $\mathscr{V}$.

If everything checks out, they start the new view and process the PRE-PREPARES in $\mathscr{O}$ as normal.


=committed
=prepared
⊥=no-op
Status in previous view
c_1 c_2 c_3 c_4 c_5 c_6
1 2 3 4 5 6 7 8 9 10
Possible new leader's log
c_1 ⊥ c_2 ⊥ ⊥ c_4 ⊥ c_5
1 2 3 4 5 6 7 8 9 10

Garbage Collection

- In the normal case, servers save their log of commands and all of the messages they receive.
- In the non-Byzantine case, servers can periodically compact their logs. They can bring out-of-date servers back up-to-date with a **state transfer**.
- In the Byzantine case, a server can't just accept a state transfer from another node. It needs proof.

Garbage collection (II)

- Servers periodically decide to take a **checkpoint**.
- Each server hashes the state of its state machine and broadcasts $\langle \text{CHECKPOINT}, n, D(S) \rangle_p$, where n is the sequence number of the last executed command and $D(S)$ is a hash of the state.
- Once a server has f+1 CHECKPOINT messages, it can compact its log and discard old protocol messages. These messages serve as a Checkpoint Certificate, proving the validity of the state.

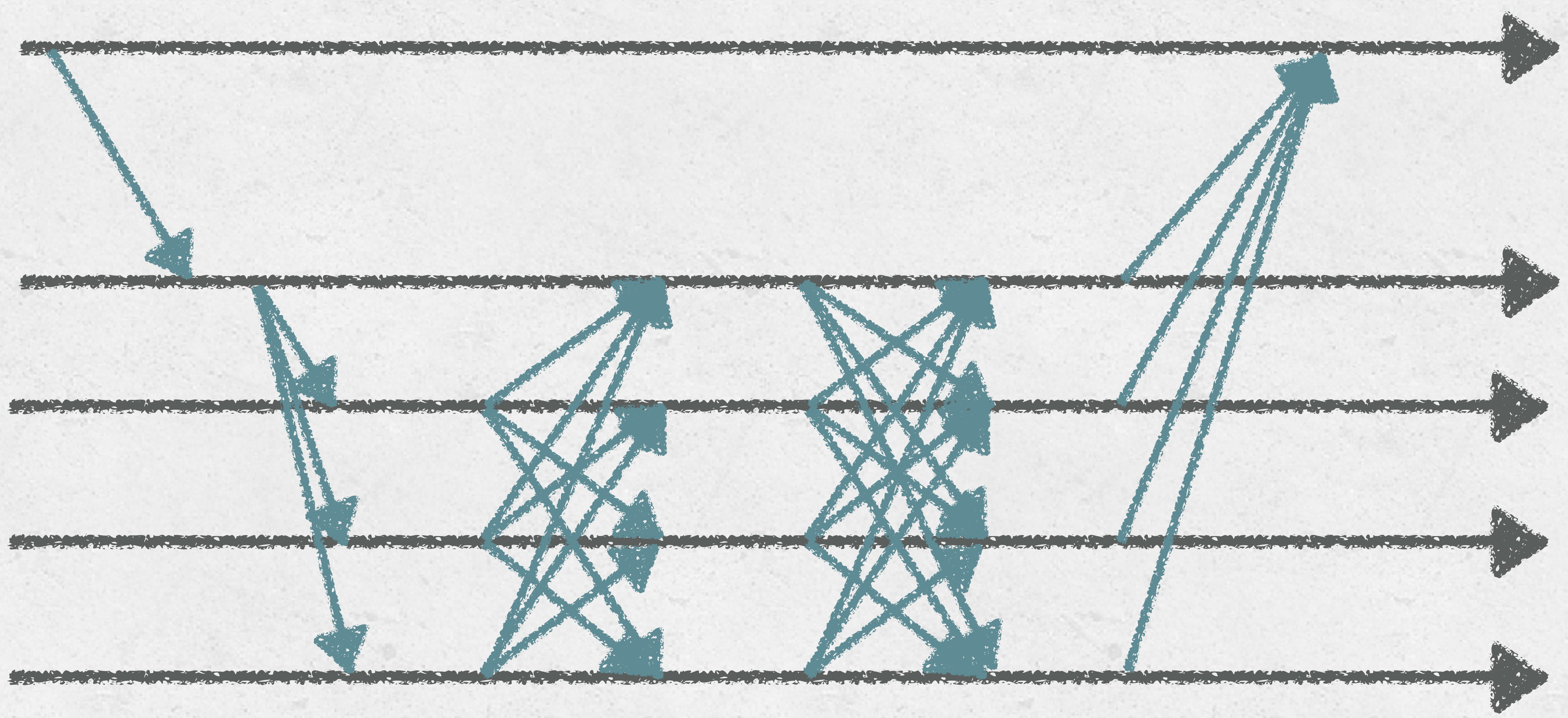
But What Did That Buy Us?

But What Did That Buy Us?

- Before, we could only tolerate crash failures.
- PBFT tolerates *any* failures, as long as only less than a third of the servers are faulty. (What happens if more are faulty?)
- However, as far as I know, PBFT and friends haven't seen wide adoption.

Performance

- Extra round of communication adds latency. (Can be avoided with speculative execution.)
- Committing a single operation requires $O(n^2)$ messages. (This can be improved, though at the cost of added latency.)
- Cryptography operations are slow! (Though the paper describes some strategies to speed them up using MACs.)



Paxos

PBFT

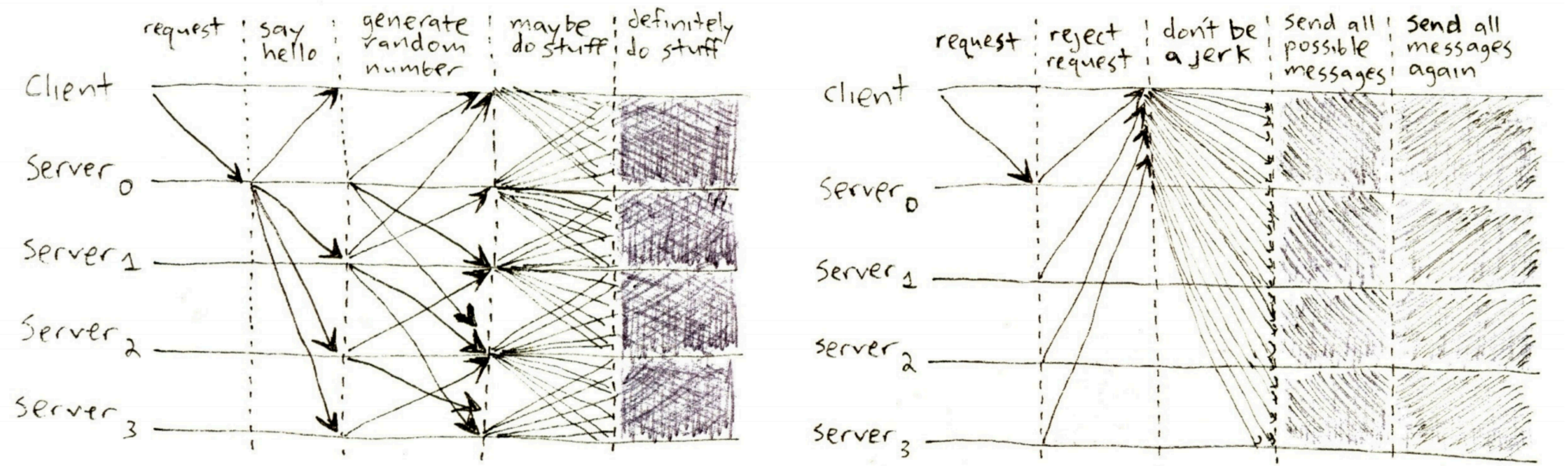


Figure 2: Our new protocol is clearly better.

[Mickens '13, *The Saddest Moment*]

How to Use BFT?

In order to use BFT, we need to have some reason to believe that the number of Byzantine failures is going to be limited, or at least that the failures will be independent and separated in time.

This probably holds true for hardware failures.

What about security flaws and software bugs?

One *possible* solution: n-version programming