Byzantine Fault Tolerance

CS 475: Concurrent & Distributed Systems (Fall 2021)
Lecture 10

Yue Cheng

Some material taken/derived from:
- Princeton COS-418 materials created by Kyle Jamieson.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.

So far: Fail-stop failures

- Traditional state machine replication tolerates **fail-stop failures:**
  - Node crashes
  - Network breaks or partitions

- State machine replication with $N = 2f+1$ replicas can tolerate f simultaneous fail-stop failures
  - **Two algorithms:** Paxos, Raft

Byzantine faults

- **Byzantine fault:** Node/component fails arbitrarily
  - Might perform incorrect computation
  - Might give conflicting information to different parts of the system
  - Might collude with other failed nodes

Byzantine faults

- **Byzantine fault:** Node/component **fails arbitrarily**
  - Might perform incorrect computation
  - Might give conflicting information to different parts of the system
  - Might collude with other failed nodes

- Why might nodes or components fail arbitrarily?
  - Software bug present in code
  - Hardware failure occurs
  - Hack attack on system

Today: Byzantine fault tolerance

- Can we provide state machine replication for a service in the presence of Byzantine faults?
- Such a service is called a Byzantine Fault Tolerant (BFT) service
- *Why might we care about this level of reliability?*

Motivation for BFT

- The ideas surrounding Byzantine fault tolerance have found numerous applications:
  - Commercial airliner flight control computer systems
  - Digital currency systems

- Some limitations, but…
  - Inspired much follow-on research to address these limitations

Mini-case-study: Boeing 777 fly-by-wire primary flight control system

- Triple-redundant, dissimilar processor hardware:
  1. Intel 80486
  2. Motorola
  3. AMD
- Each processor runs code from a different compiler

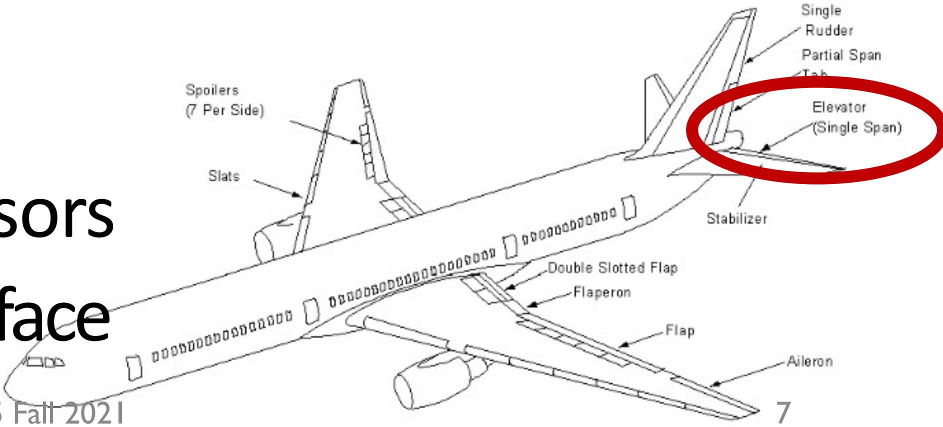
Key techniques:

Hardware and software diversity,

Voting between components

Simplified design:

- Pilot inputs → three processors
- Processors vote → control surface



Today

1. Traditional state-machine replication for BFT?
2. Practical BFT replication algorithm

Review: Tolerating one fail-stop failure

- Traditional state machine replication (Paxos) requires, *e.g.*, $2f + 1$ = **three** replicas, if $f = 1$

- Operations are totally ordered → correctness
  - A two-phase protocol

- Each operation uses $\geq f + 1 = 2$ of them
  - Overlapping quorums
    - So at least one replica "remembers"

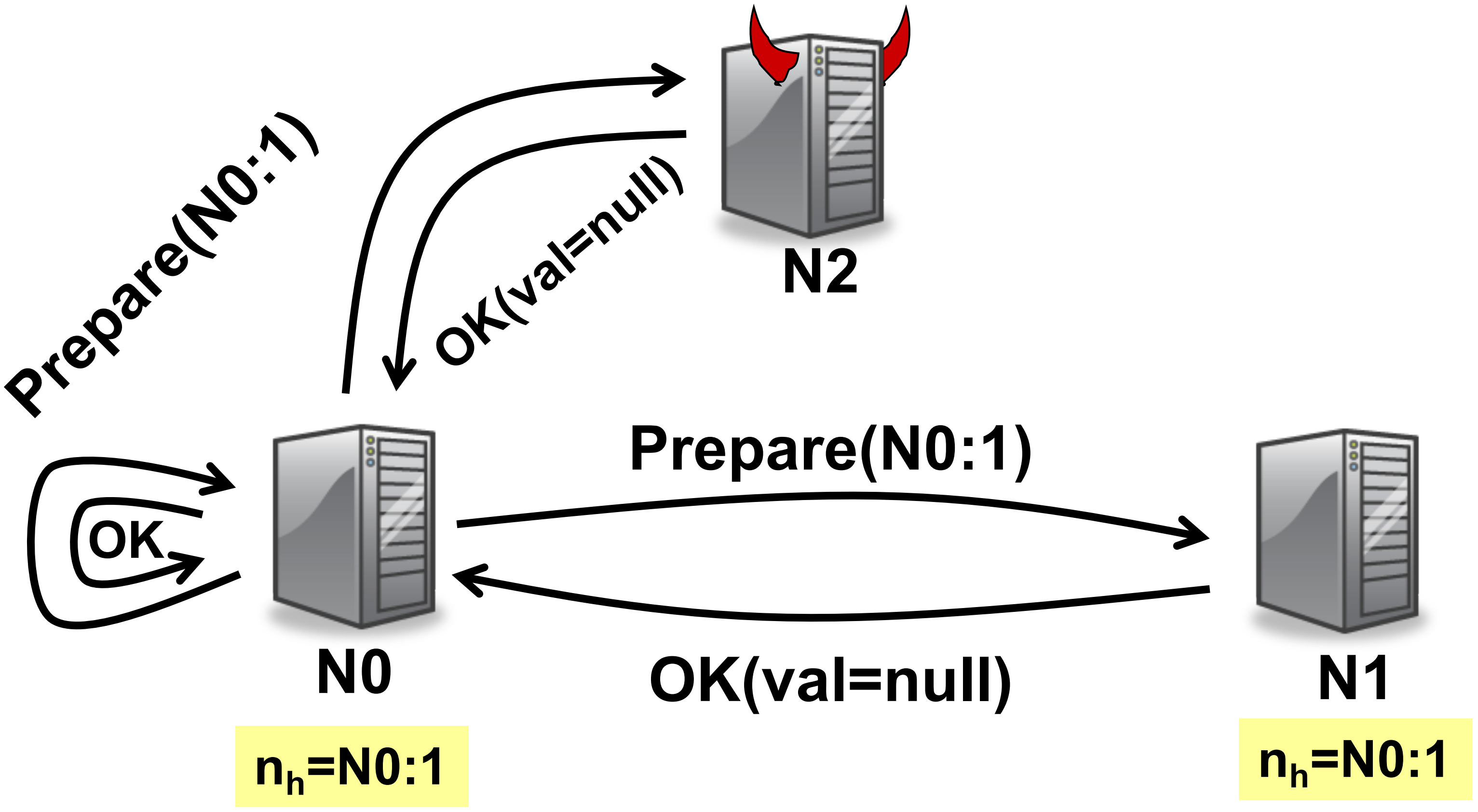
Use Paxos for BFT?

1. Can't rely on the primary to assign seqno
    - Could assign same seqno to different requests

2. **Can't use Paxos** for view change
    - Under Byzantine faults, the intersection of two majority (*f* + 1 node) quorums may be bad node
    - Bad node tells **different** quorums different things!
        - *e.g.* tells N0 accept val1, but N1 accept val2

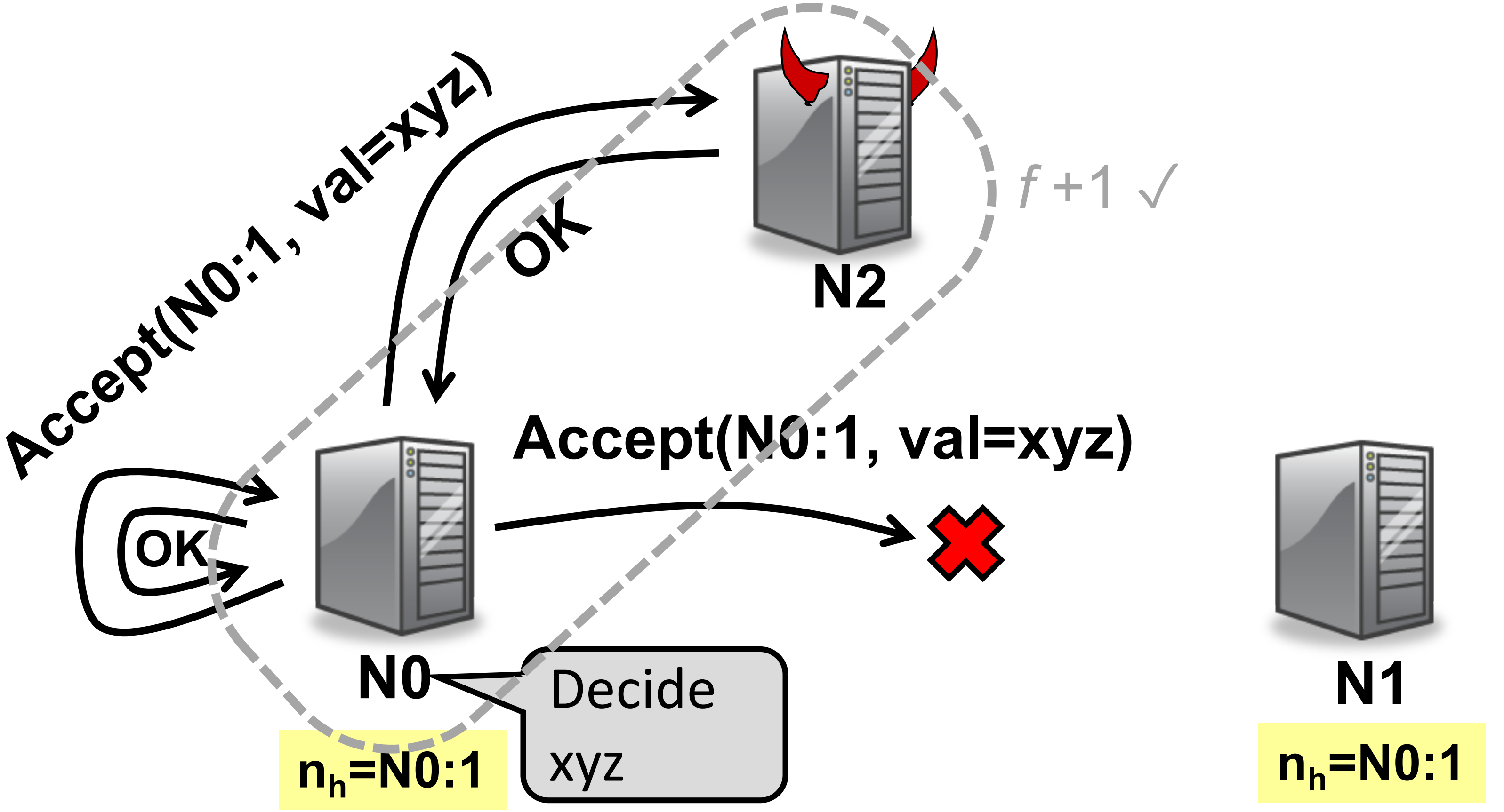
Paxos under Byzantine faults

($f = 1$)



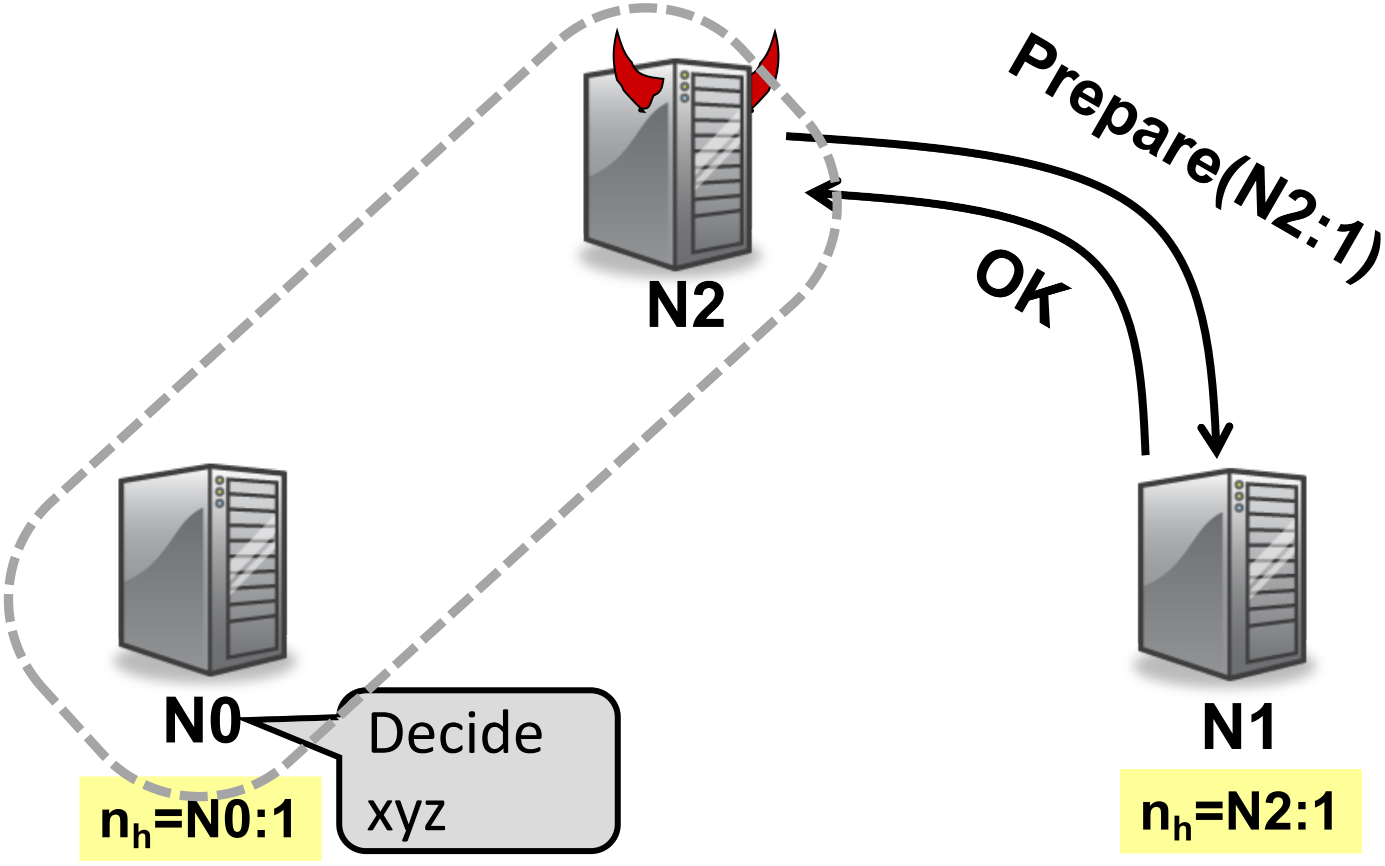
Paxos under Byzantine faults

(f = 1)



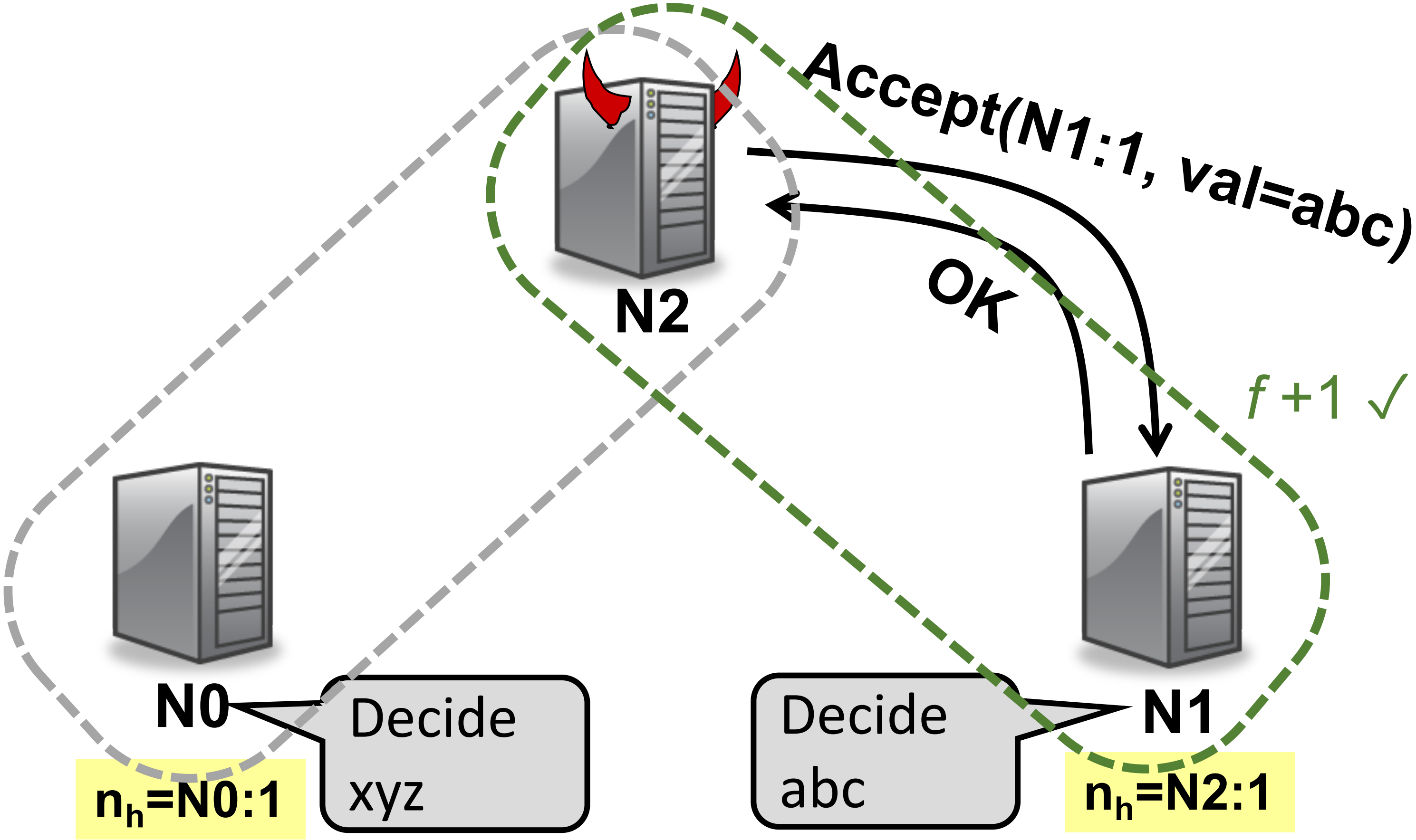
Paxos under Byzantine faults

($f = 1$)



Paxos under Byzantine faults

(f = 1)



Conflicting decisions!

Theoretical fundamentals: Byzantine Generals

General #2

Unreliable messenger

General #1

General #3

Theoretical fundamentals: Byzantine Generals

General #2

Unreliable messenger

General #1

General #3

Result: Using messengers, problem **solvable iff** > ⅔ of the generals are loyal

Put burden on client instead?

- Clients **sign** input data before storing it, then verify signatures on data retrieved from service

- **Example:** Store signed file f1="aaa" with server
    - Verify that returned f1 is correctly signed

But a Byzantine node can **replay stale,** signed **data** in its response

Inefficient: Clients have to perform computations and sign data

Today

1. Traditional state-machine replication for BFT?

2. Practical BFT replication algorithm
   [Castro & Liskov, 1999]

Practical BFT: Overview

- Uses $3f+1$ **replicas** to survive f **failures**
  - Shown to be minimal **(Lamport)**

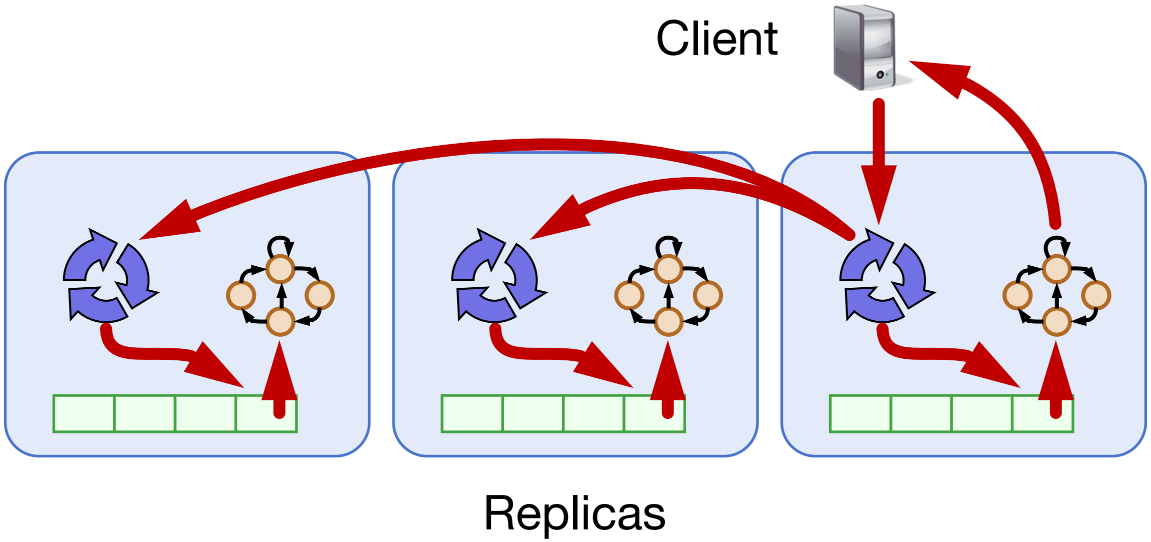
- Requires **three phases (not two)**

- Provides state machine replication
  - Arbitrary service accessed by operations, *e.g.*,
    - File system ops read and write files and directories
  - Tolerates Byzantine-faulty clients

Correctness argument

- Assume
  - Operations are deterministic
  - Replicas **start in same state**

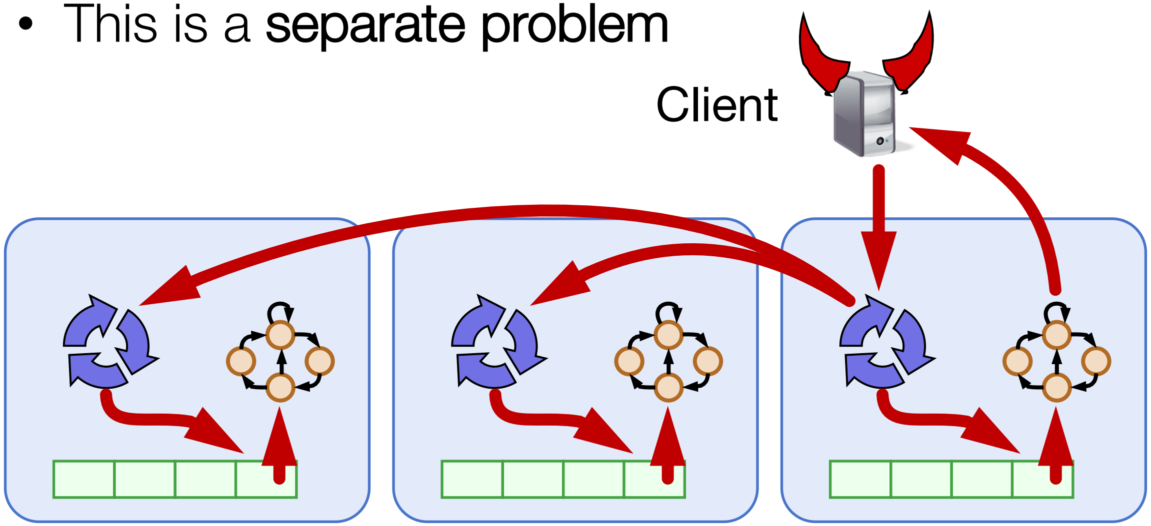
- Then if replicas execute the **same requests** in the **same order:**
  - Correct replicas will produce identical results



Non-problem: Client failures

- Clients can’t cause internal inconsistencies of the data in servers
  - State machine replication property

- Clients can write bogus data to the system
  - **Sol’n:** Authenticate clients and separate their data
    - This is a **separate problem**



Client

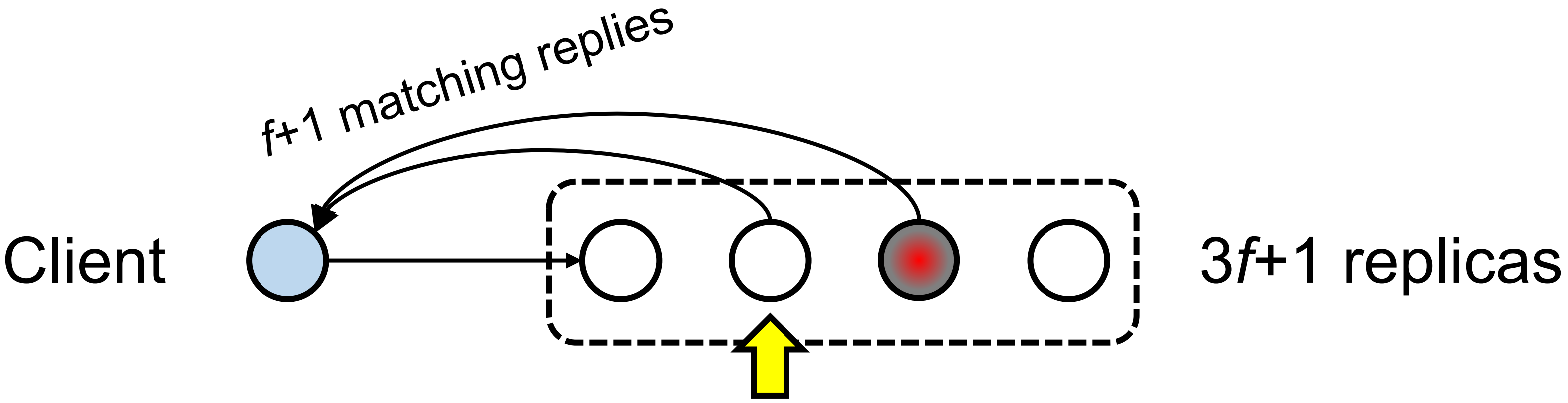
Replicas

What clients do

1. Send requests to the primary replica

2. Wait for *f*+1 **identical** replies
   - **Note:** The replies may be deceptive
     - *i.e.,* replica returns "correct" answer, but locally does otherwise!

- But at least one reply is from a non-faulty replica



What replicas do

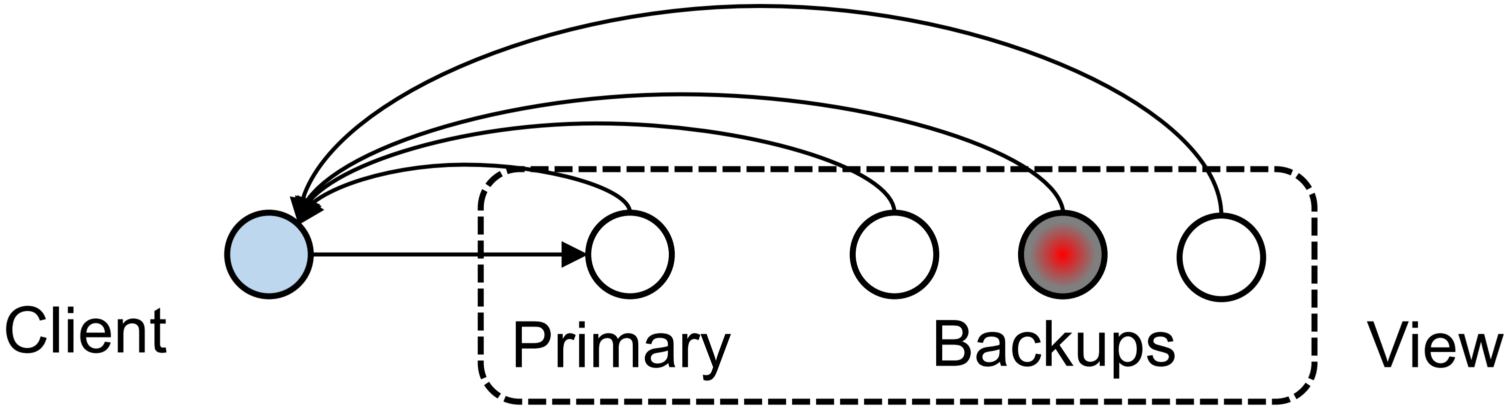
- Carry out a protocol that ensures that
  - Replies from honest replicas are correct

  - Enough replicas process each request to ensure that
    - The **non-faulty** replicas process the **same requests**
    - In the **same order**

- Non-faulty replicas obey the protocol

Primary-Backup protocol

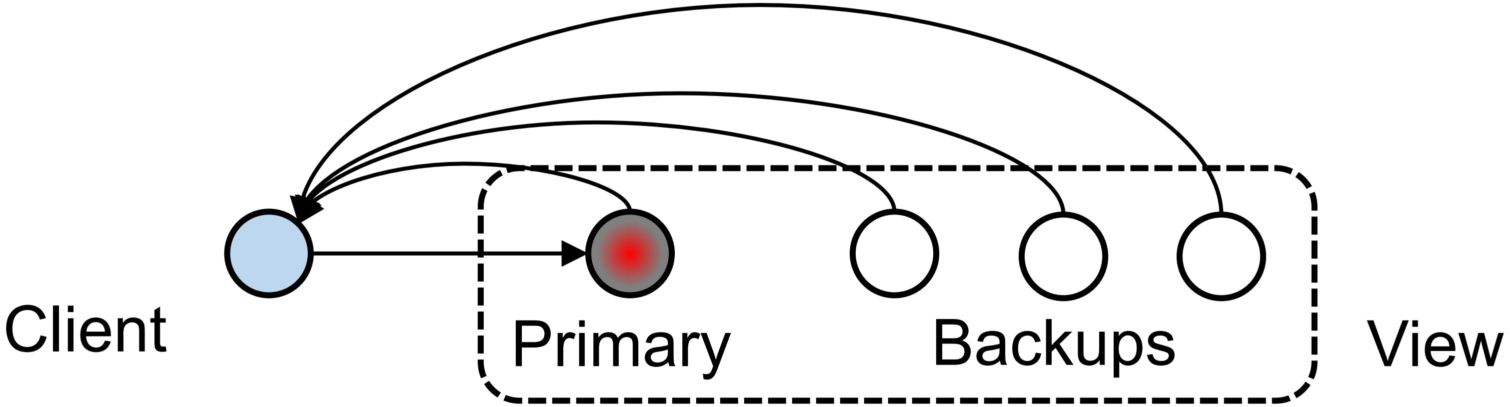
- Primary-Backup protocol: Group runs in a **view**
  - View **number** designates the **primary** replica



- Primary is the node whose `id == view# (modulo N)`

Ordering requests

- Primary picks the ordering of requests
  - But the **primary** might be a liar!

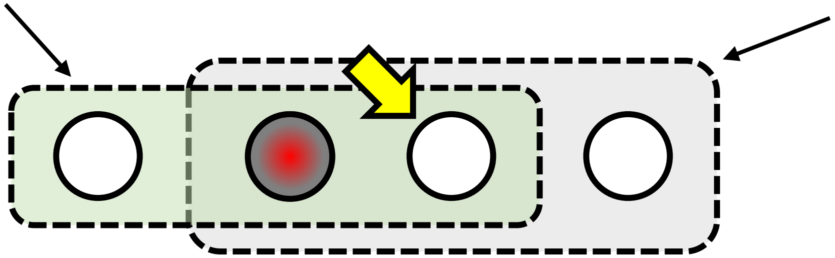


- Backups ensure primary behaves correctly
  - Check and certify correct ordering
  - Trigger **view changes** to replace faulty primary

Byzantine quorums

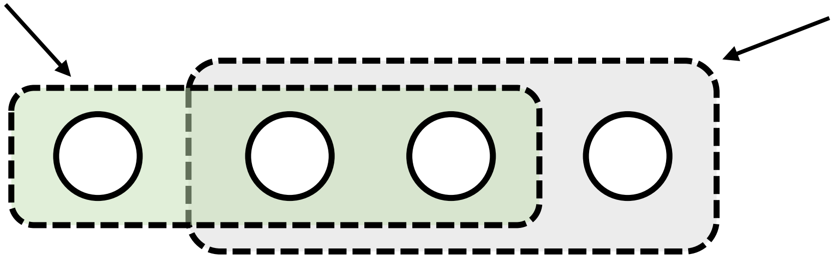
($f = 1$)

A ***Byzantine quorum*** contains ≥ $2f+1$ replicas



- One op's quorum overlaps with next op's quorum
  - There are $3f+1$ replicas, in total
    - So overlap is ≥ $f+1$ replicas
- $f+1$ replicas must contain ≥ 1 non-faulty replica

Quorum certificates



A ***Byzantine quorum*** contains ≥ 2*f*+1 replicas

- ***Quorum certificate:*** a collection of $2f + 1$ signed, identical messages from a Byzantine quorum
  - All messages agree on the **same statement**

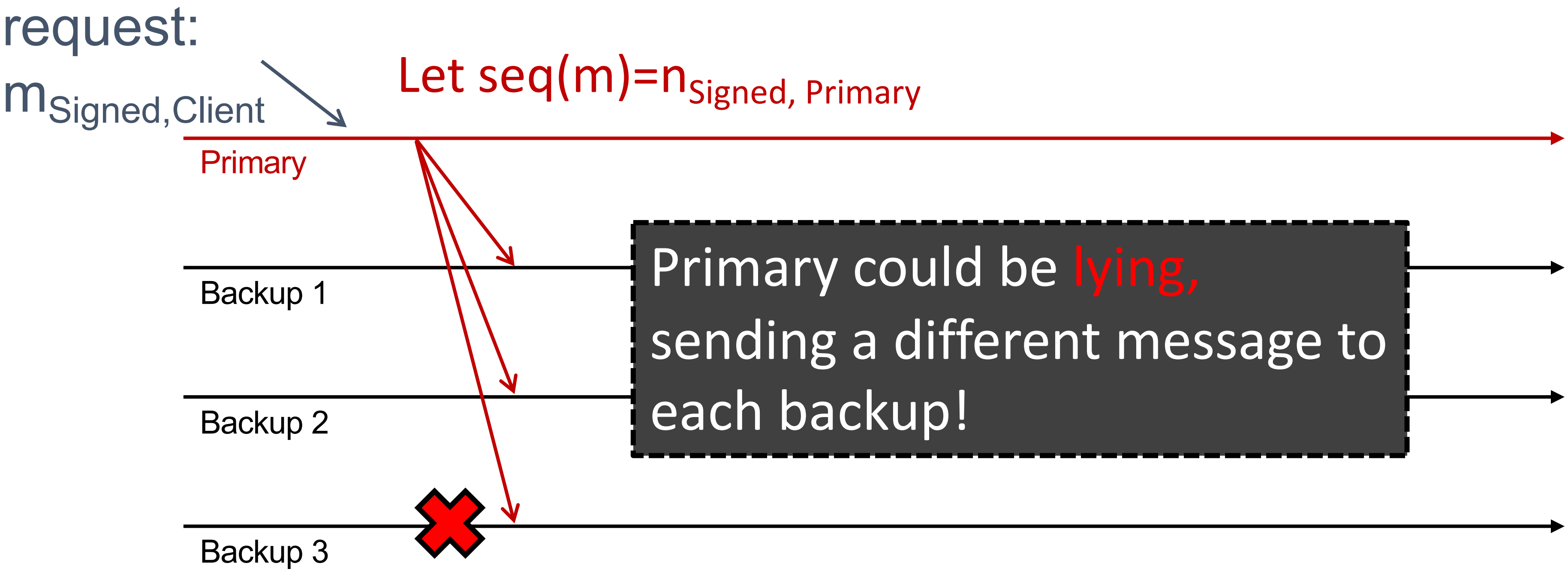
Keys

- Each client and replica has a **private-public keypair**

- **Secret keys:** symmetric cryptography
  - Key is known only to the two communicating parties
  - Bootstrapped using the public keys

- **Each client, replica** has the following secret keys:
  - One key per replica for sending messages
  - One key per replica for receiving messages

Ordering requests



request: $m_{\text{Signed,Client}}$

Let $\text{seq}(m)=n_{\text{Signed, Primary}}$

Primary

Backup 1

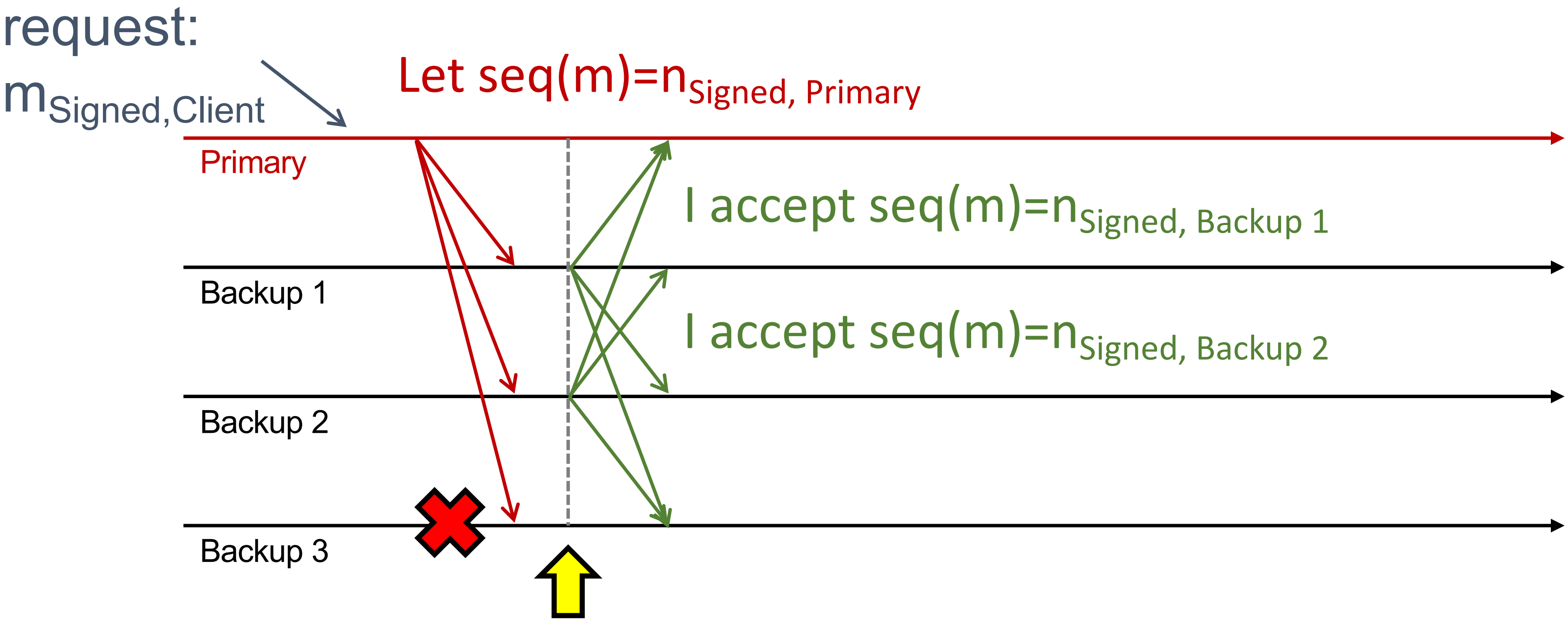
Backup 2

Backup 3

Primary could be lying, sending a different message to each backup!

- Primary chooses the request's *sequence number* (*n*)
  - Sequence number determines order of execution

Checking the primary's message



request: $m_{\text{Signed,Client}}$

Let $seq(m)=n_{\text{Signed, Primary}}$

Primary

I accept $seq(m)=n_{\text{Signed, Backup 1}}$

Backup 1

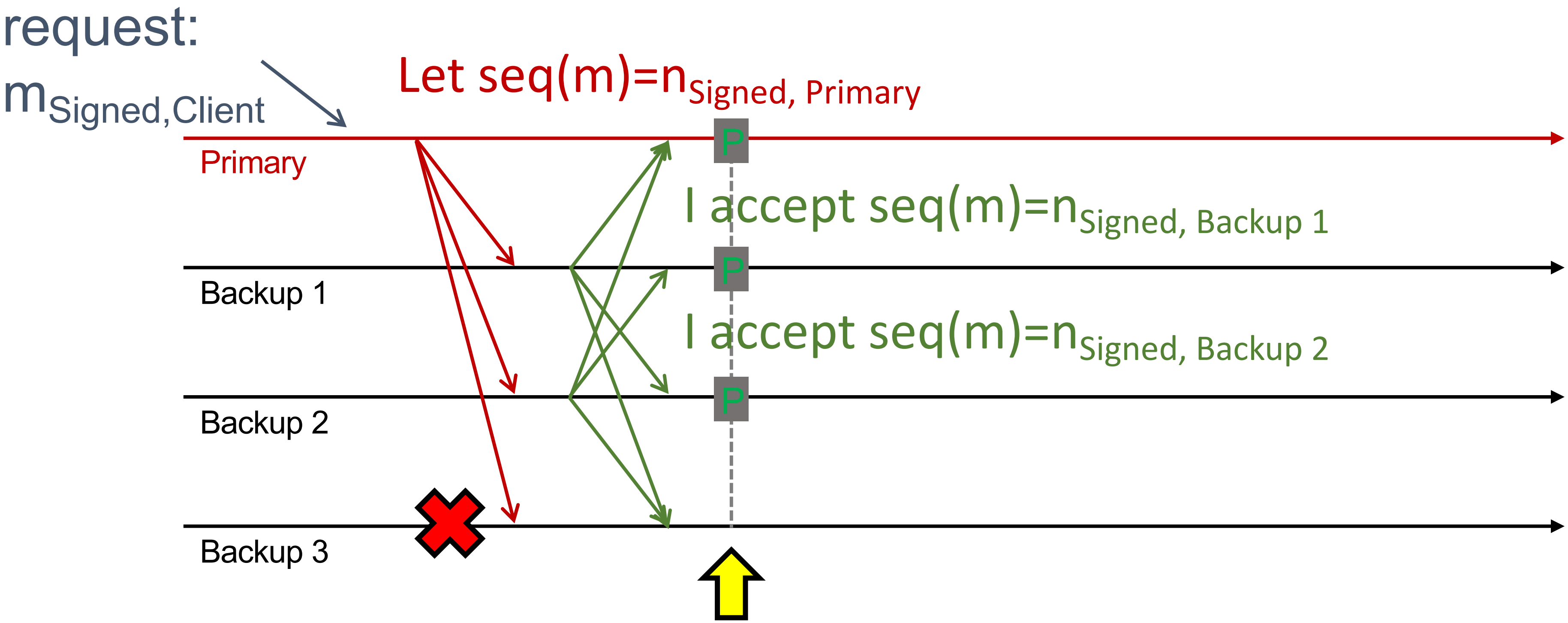
I accept $seq(m)=n_{\text{Signed, Backup 2}}$

Backup 2

Backup 3

- Backups **locally** verify they've seen ≤ **one** client request for sequence number *n*
  - If local check passes, replica broadcasts *accept* message
    - Each replica makes this decision **independently**

Collecting a *prepared certificate* ($f = 1$)



request: $m_{\text{Signed,Client}}$

Let $\text{seq}(m)=n_{\text{Signed, Primary}}$

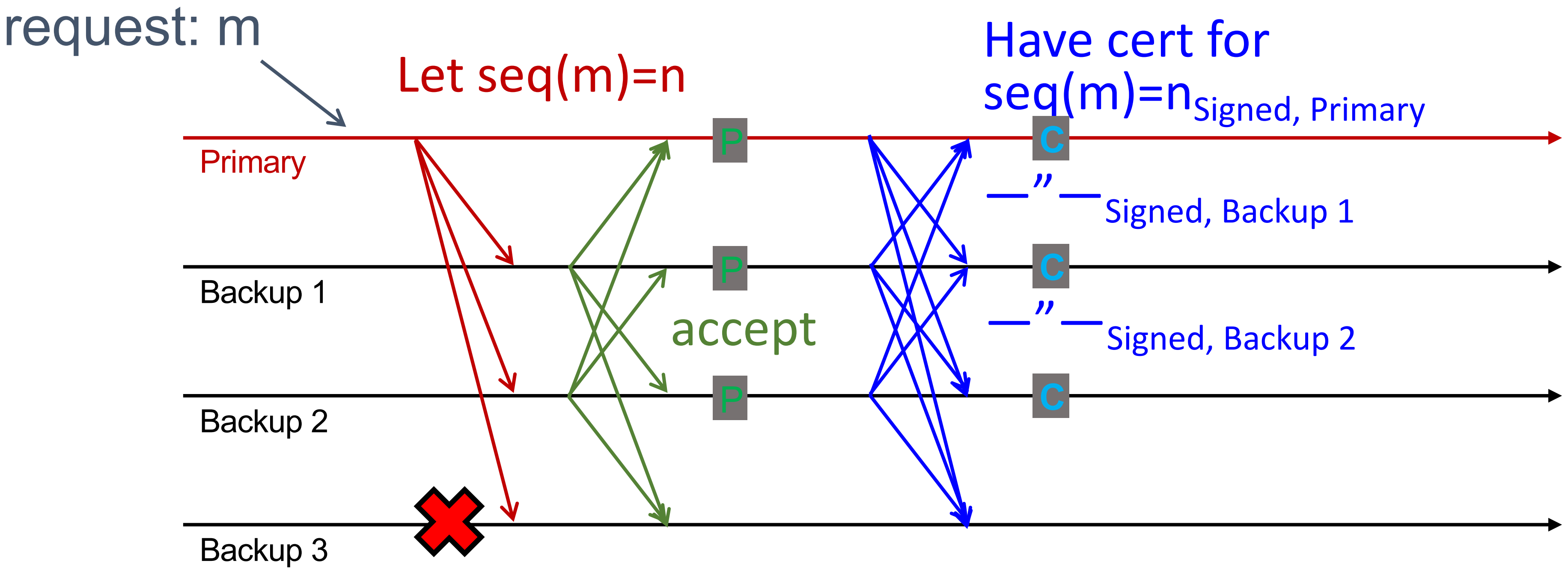
I accept $\text{seq}(m)=n_{\text{Signed, Backup 1}}$

I accept $\text{seq}(m)=n_{\text{Signed, Backup 2}}$

- Backups wait to collect a *prepared* quorum certificate
- Message is *prepared* (P) at a replica when it has:
  - A message from the primary proposing the seqno
  - $2f$ messages from itself and others accepting the seqno

Each correct node has a prepared certificate locally, but does not know whether the other correct nodes do too! So, we can't commit yet!

Collecting a *committed certificate* ($f = 1$)



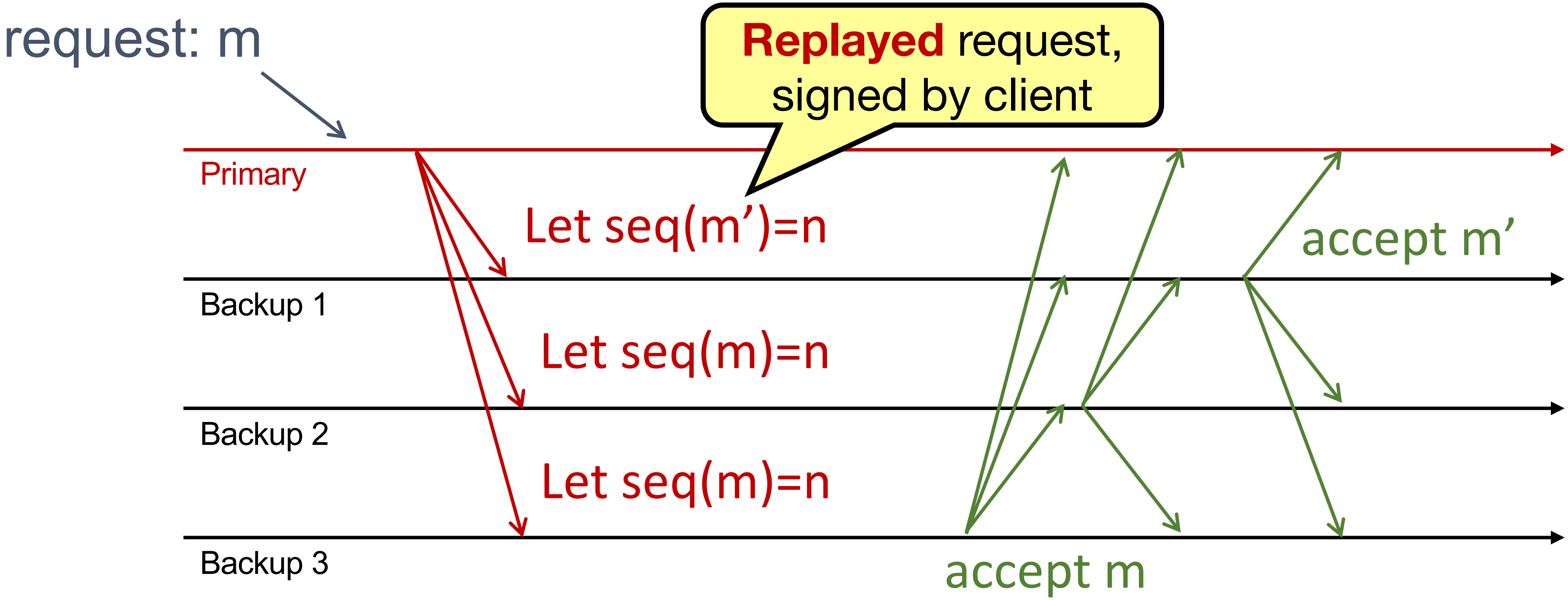
Once the request is **committed**, replicas execute the operation and send a reply directly back to the client.

Byzantine primary: replaying old requests

- The client assigns each request a unique, monotonically increasing *timestamp t*
- Servers track greatest *t* executed for each client *c*, T(c), and their corresponding reply
  - On receiving request to execute with timestamp *t*:
    - If t < T(c), skip the request execution
    - If t = T(c), resend the reply but skip execution.
    - If t > T(c), execute request, set T(c) ← t, remember reply

Malicious primary can invoke t = T(c) case but **cannot compromise safety**

Byzantine primary: Splitting replicas (*f* = 1)



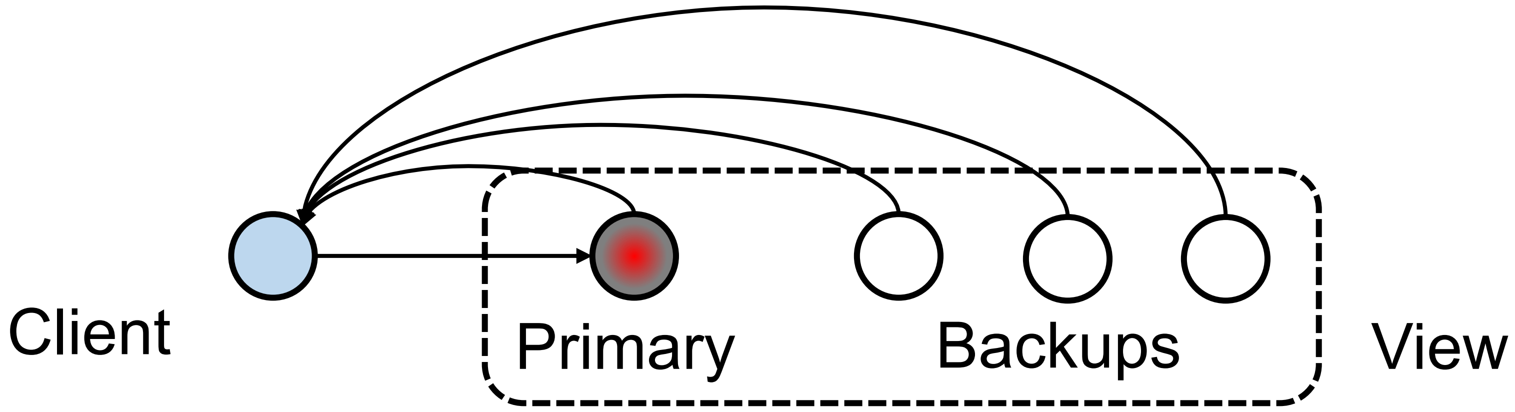
- **Recall:** To prepare, need primary message and 2*f* accepts
  - Backup 1: Won’t prepare m’
  - Backups 2, 3: Will prepare m

Byzantine primary: Splitting replicas

- In general, backups won't prepare two different requests with the same seqno if primary lies

- **Suppose they did:** two distinct requests **m** and **m′** for the same sequence number n
  - Then prepared quorum certificates (each of size 2*f*+1) would **intersect** at an **honest** replica
  - So that honest replica would have sent an accept message for both m and m′
    - So m = m′

View change



- If a replica suspects the primary is faulty, it requests a *view change*
  - Sends a *viewchange* request to all replicas
    - Everyone acks the view change request

- New primary collects a quorum (2f+1) of responses
  - Sends a *new-view* message with this certificate

Considerations for view change

- Need committed operations to survive into next view
    - Client may have gotten answer

- Need to preserve liveness
    - If replicas are too fast to do view change, but really primary is okay – then performance problem

    - Or malicious replica tries to subvert the system by proposing a bogus view change

Garbage collection

- Storing all messages and certificates into a **log**
  - Can't let log grow without bound

- Protocol to **shrink the log** when it gets too big
  - Discard messages, certificates on commit?
    - No! Need them for view change
  - Replicas have to agree to shrink the log