CS 582: Distributed Systems

PBFT and ZooKeeper

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgments: Slides use some material from Ryan Stutsman (University of Utah)

Agenda

- Wrap-up discussion of PBFT
- ZooKeeper

Specific learning outcomes

By the end of today's lecture, you should be able to:

- ❑ Describe the core components and workflow of the PBFT algorithm
- ❑ Analyze the effectiveness of PBFT in maintaining consensus under Byzantine conditions
- ❑ Evaluate the performance of PBFT
- ❑ Explain the core design of ZooKeeper
- ❑ Analyze how ZooKeeper is able to scale system throughput with increasing replicas
- ❑ Analyze the consistency guarantees that ZooKeeper is providing

Byzantine Fault Tolerance

⬇

Design services that continue to function <u>correctly</u> despite Byzantine faults

⬇

Solve the replicated state machine problem with Byzantine faults

⬇

Solving <u>consensus</u> with Byzantine faults

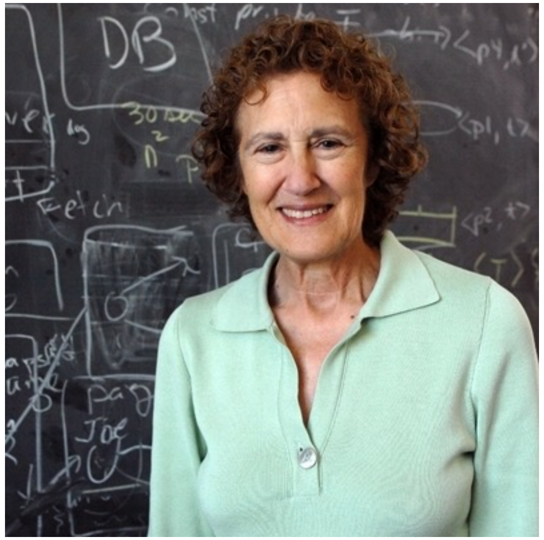
Recap: Can't use Paxos/Raft with Byzantine Faults

Bare majority quorums may not be enough in the presence of byzantine faults

A leader might be a byzantine node so we can't trust it

Byzantine Fault Tolerance

- Practical Byzantine Fault Tolerance (PBFT) Algorithm
  - [Liskov and Castro, 1999]



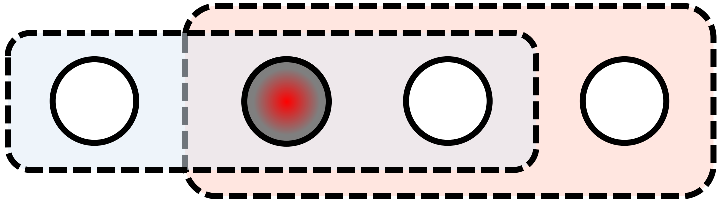
Barbara Liskov

PBFT: Impact

- Seminal work on byzantine fault tolerance
  - Byzantine fault tolerance was for long considered an exotic topic
- PBFT showed Byzantine fault tolerance is possible under certain assumptions
  - Has inspired a large body of work in the last two decades
- Used by multiple blockchains like Zilliqa, Hyperledger fabric, etc.

PBFT Overview

- $3f + 1$ replicas to survive f byzantine failures
  - Shown to be minimal



- Quorum of $2f + 1$ replicas
- Three phases (instead of two)
- Primary-backup protocol
  - Since primary can be byzantine node, replicas observe, can trigger change
  - Change through the idea of <u>views</u>; primary = view mod |R|
- Clients: need $f + 1$ matching responses from different replicas

Key assumptions

- No more than f out of $3f + 1$ replicas can be faulty
  - The other $2f + 1$ replicas operate correctly – follow the PBFT protocol
- No client failure – clients can never do anything bad
- Worst case
  - A single attacker controlling f faulty replicas to break the system
- Note: faulty → could be experiencing byzantine faults

What are the attacker’s powers?

What are the attacker's powers?

- Supplies the code that faulty replicas run
- Knows the code the non-faulty replicas are running
- Knows the faulty replicas' crypto keys
- Can read network messages
- However, can't forge messages of non-faulty nodes
  - Messages are encrypted -- no guessing of crypto keys or breaking of cryptography

PBFT is a primary-backup protocol

- What can go wrong if we have a Byzantine primary?
  - It can send a wrong result to the client
  - Different updates to different replicas
  - Ignore a client request

- How do clients interact with the system?
  - If they just contact the primary, and the primary is byzantine, then the system is in trouble

Views

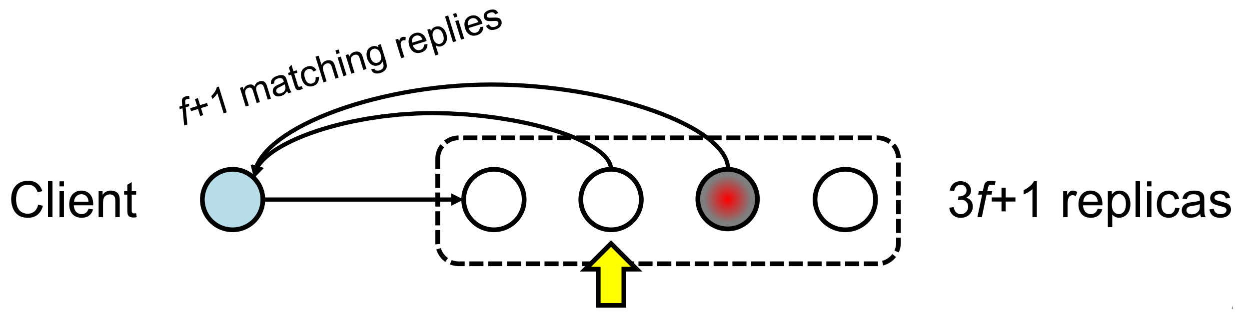
Views

- Identify each replica by an integer
  - For a set of R replicas *{0,…, |R-1|}*
- The replicas move through succession of configurations called views
- In a view, one replica is the primary and the rest are backups
- The primary of a view is a replica p such that *p = v mod |R|* where v is the view number

How Clients determine success?

How Clients determine success?

- Clients wait for $f + 1$ identical replies from different replicas
- But ≥ one reply is from a non-faulty replica



What Clients exactly do?

- Send requests to the primary replica
  - The primary multicasts the request to the backups
  - Replicas execute the request and send a reply to the client
- If the client does not receive replies soon enough, it broadcasts the request to all replicas
  - If the request has already been processed, the replicas simply resend reply
    - Replicas remember the last reply message they sent to each client
  - Otherwise, if the replica is not the primary it relays the request to the primary
    - If the primary does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change

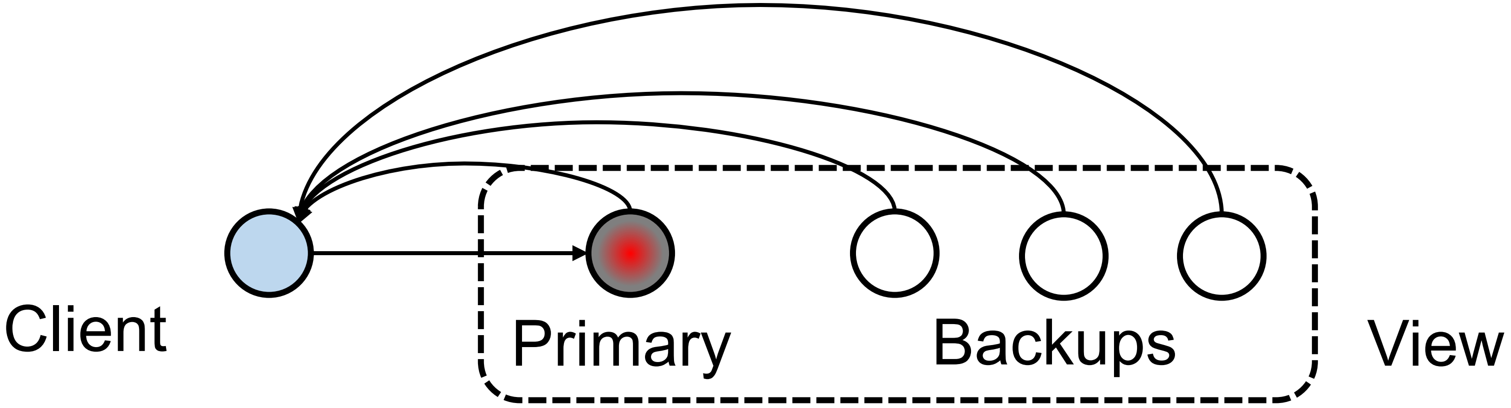
What Replicas do?

- Carry out a protocol that ensures that
  - Replies from honest (non-faulty) replicas are correct
  - Enough replicas process each request to ensure that
    - The non-faulty replicas process the same requests and
    - In the same order
- Non-faulty replicas obey the protocol

Ordering Requests

Ordering Requests

- Primary picks the ordering of requests
    - But the primary might be a byzantine node

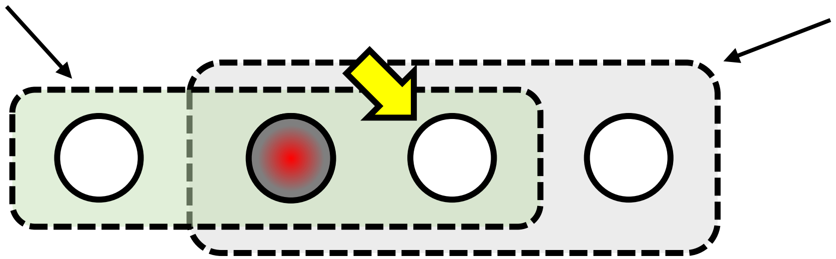


- Backups ensure primary behaves correctly
    - Check and certify ordering
    - Trigger view changes to replace faulty primary

Byzantine Quorums

Byzantine Quorums

- A Byzantine quorum contains ≥ 2f+1 replicas (given 3f+1 total nodes)



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

Message Authentication?

Message Authentication

- Public-key cryptography for signatures
- Each client and server has a private and public key
- All hosts know all public keys
- Signed messages are signed with private key
- Public key can verify that message came from host with the private key

How many phases of interactions?

Three Phases

- Pre-prepare: pick order of request and inform replicas
- Prepare: ensures order within views
- Commit: ensures order across views

Normal Operation (primary is not faulty)

Normal Operation (primary is not faulty)

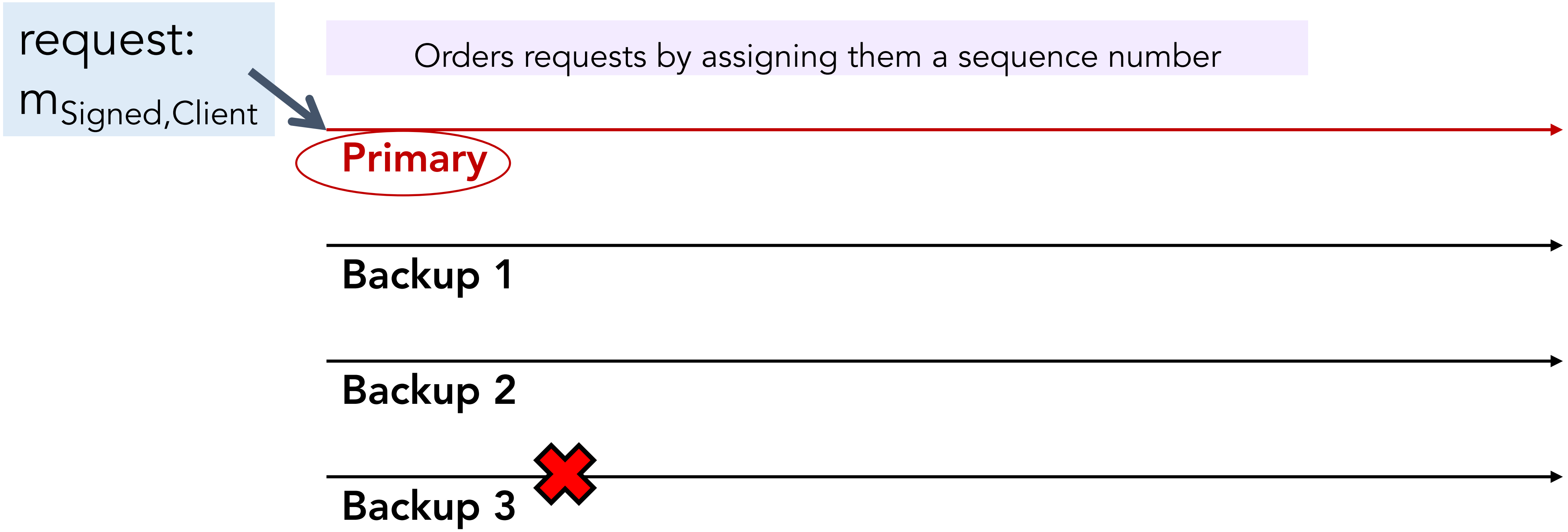
Normal Operation (primary is not faulty)

request:
$m_{Signed,Client}$

Message includes *operation*, *timestamp*, client id

Assumption
A client sends operations one by one

Normal Operation (primary is not faulty)

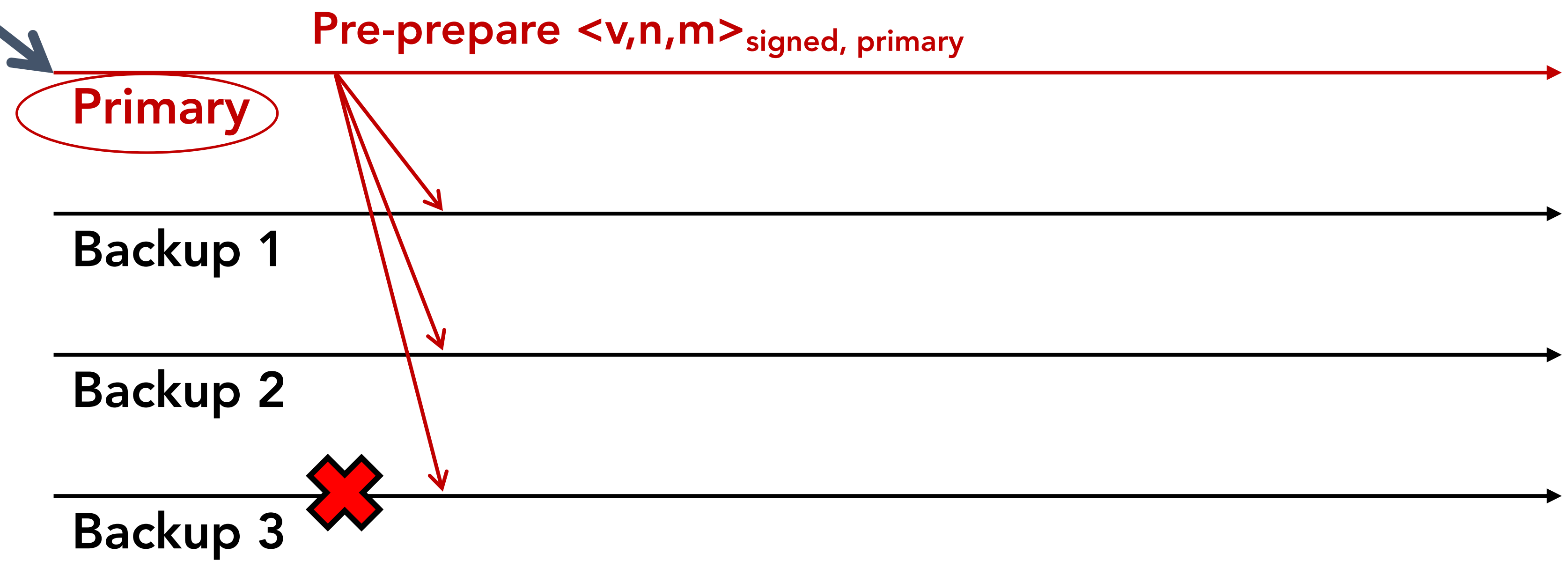


- Primary chooses the request's sequence number
  - Sequence number determines order of execution

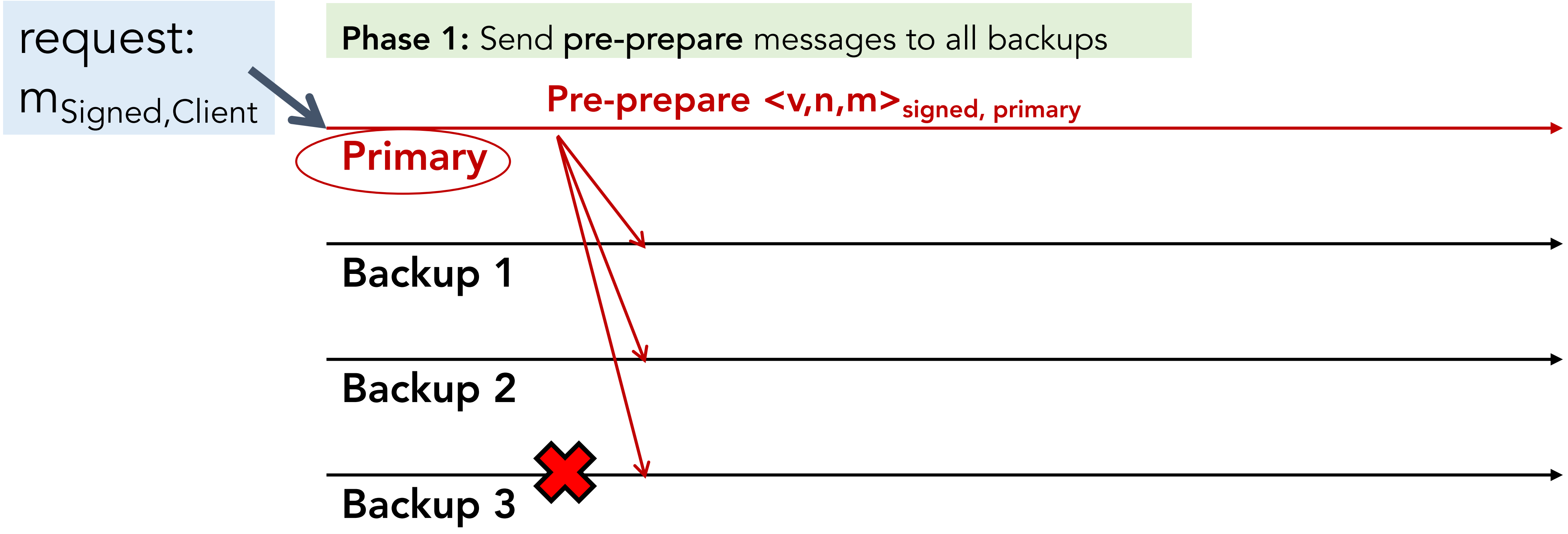
Normal Operation (primary is not faulty)

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

Phase 1: Send **pre-prepare** messages to all backups



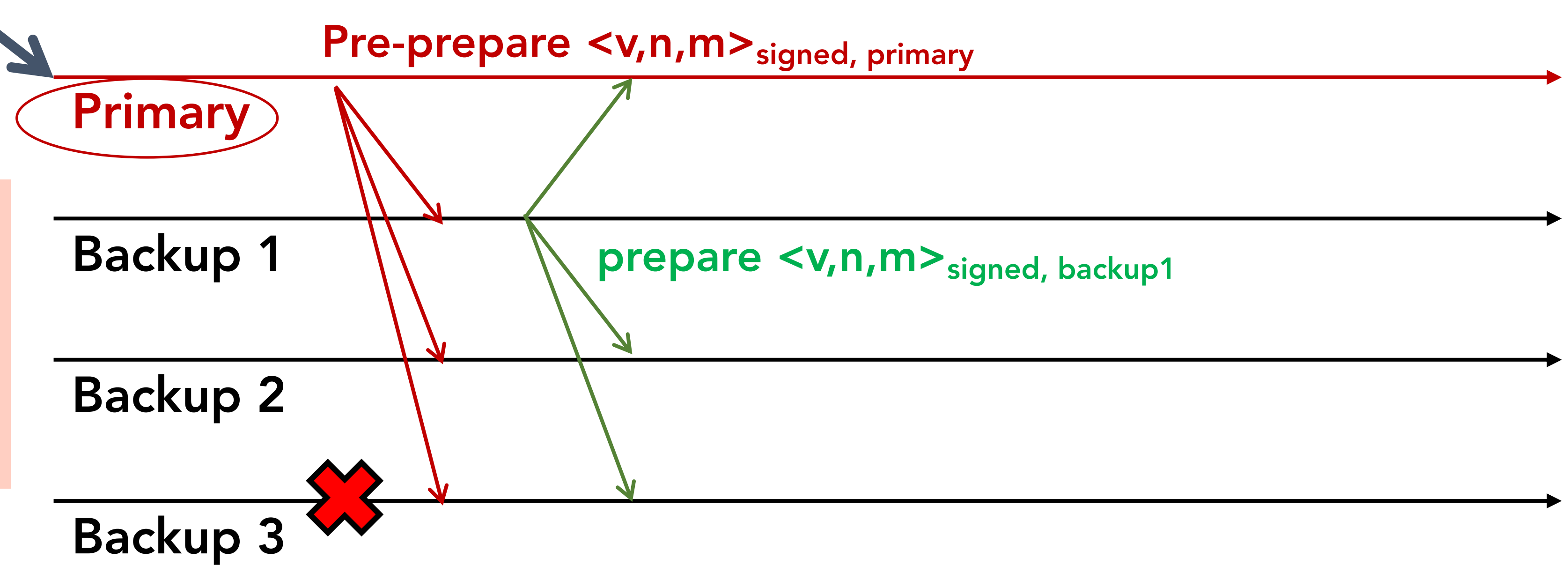
Normal Operation (primary is not faulty)



- Backups do the following (and some other checks):
  - They check they are in view v
  - It has not seen another message with view v and sequence number n

Normal Operation (primary is not faulty)

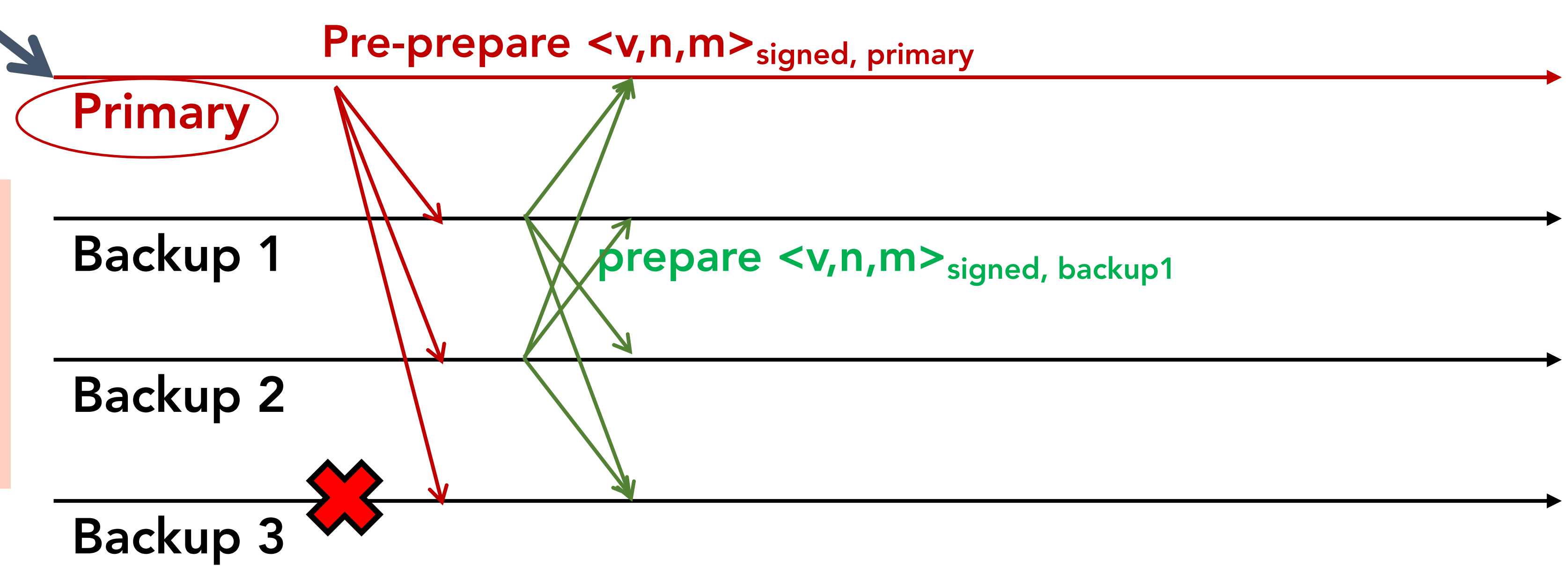
request:
$m_{Signed,Client}$



Phase 2: Each backup which accepts **pre-prepare** msgs broadcasts a **prepare** message

Normal Operation (primary is not faulty)

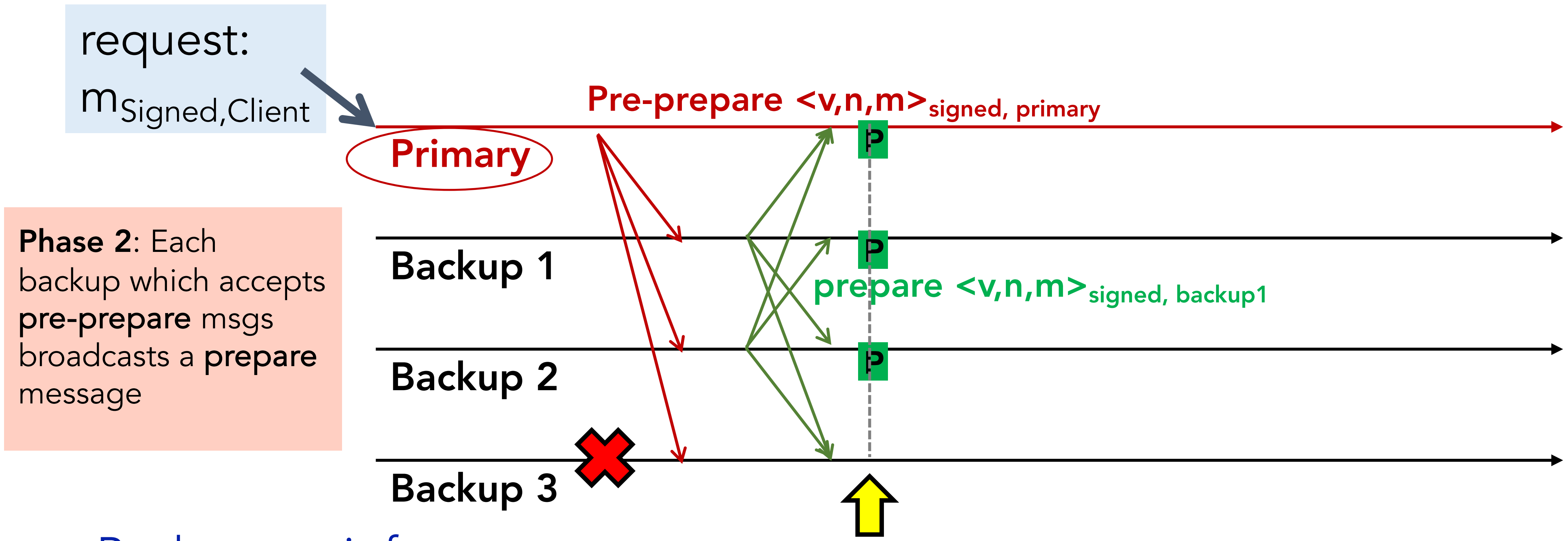
request:
$m_{Signed,Client}$



Phase 2: Each backup which accepts **pre-prepare** msgs broadcasts a **prepare** message

- Backups wait for:
  - <u>Prepared certificate:</u> a collection of 2f matching prepare messages from replicas (including itself) + 1 matching pre-prepare message from the primary

Normal Operation (primary is not faulty)

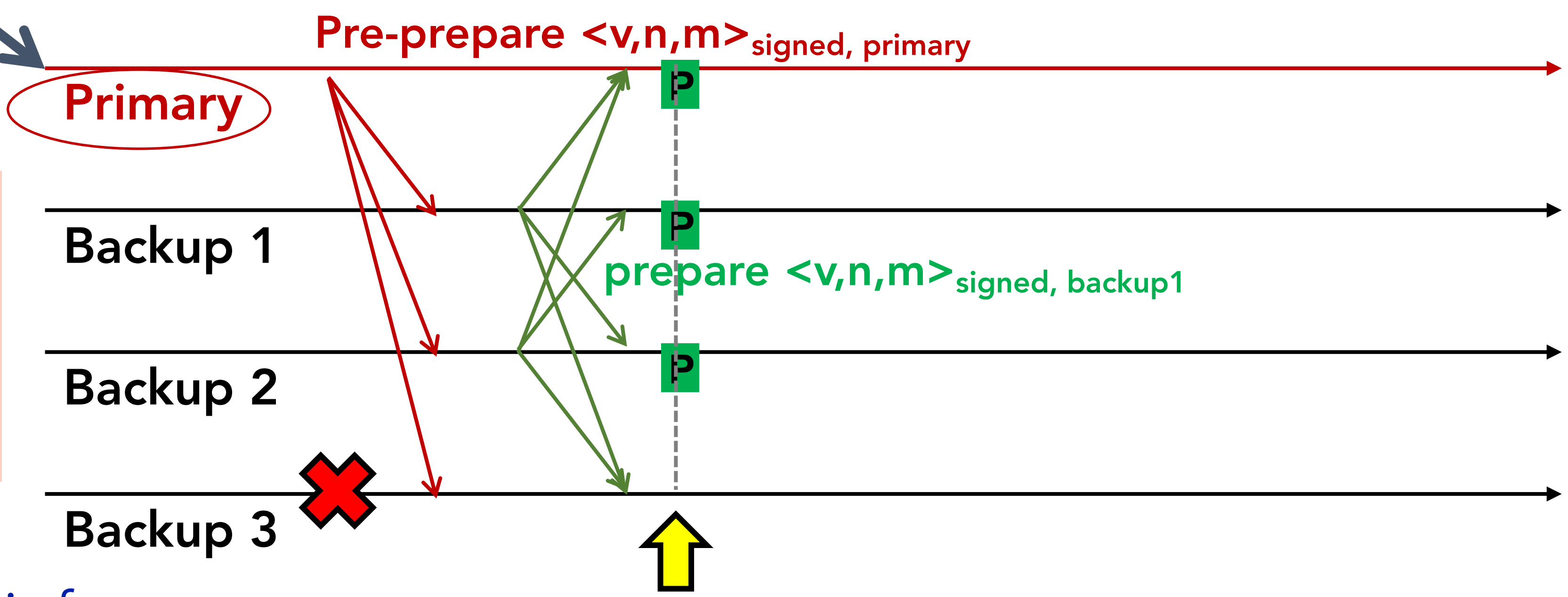


- Backups wait for:
  - <u>Prepared certificate:</u> a collection of 2f matching prepare messages from replicas (including itself) + 1 matching pre-prepare message from the primary

Normal Operation (primary is not faulty)

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

Phase 2: Each backup which accepts **pre-prepare** msgs broadcasts a **prepare** message



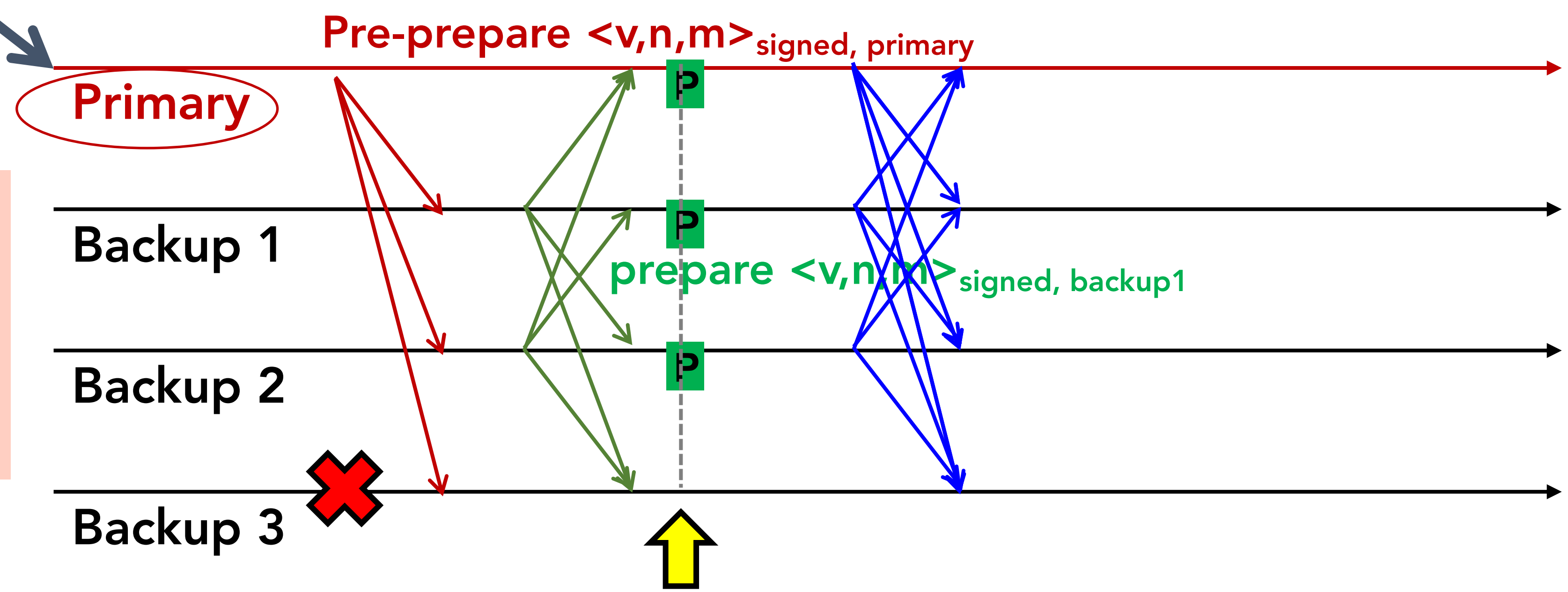
- Backups wait for:
  - Prepared certificate: a collection of 2f matching prepare msgs from replicas (including itself) + 1 matching pre-prepare message from the primary

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!

Normal Operation (primary is not faulty)

request:
$m_{Signed,Client}$

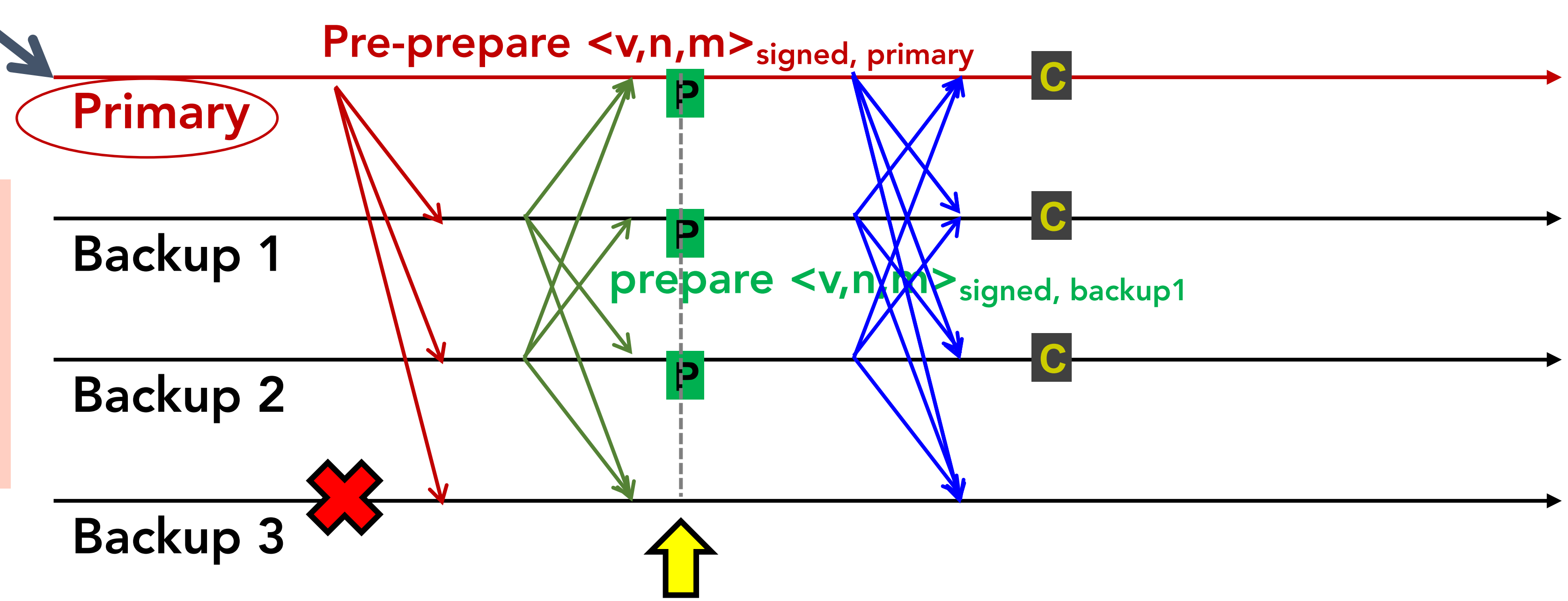
Phase 3: Each backup which has a **prepared** certificate broadcasts a **commit** message



Normal Operation (primary is not faulty)

request:
$m_{Signed,Client}$

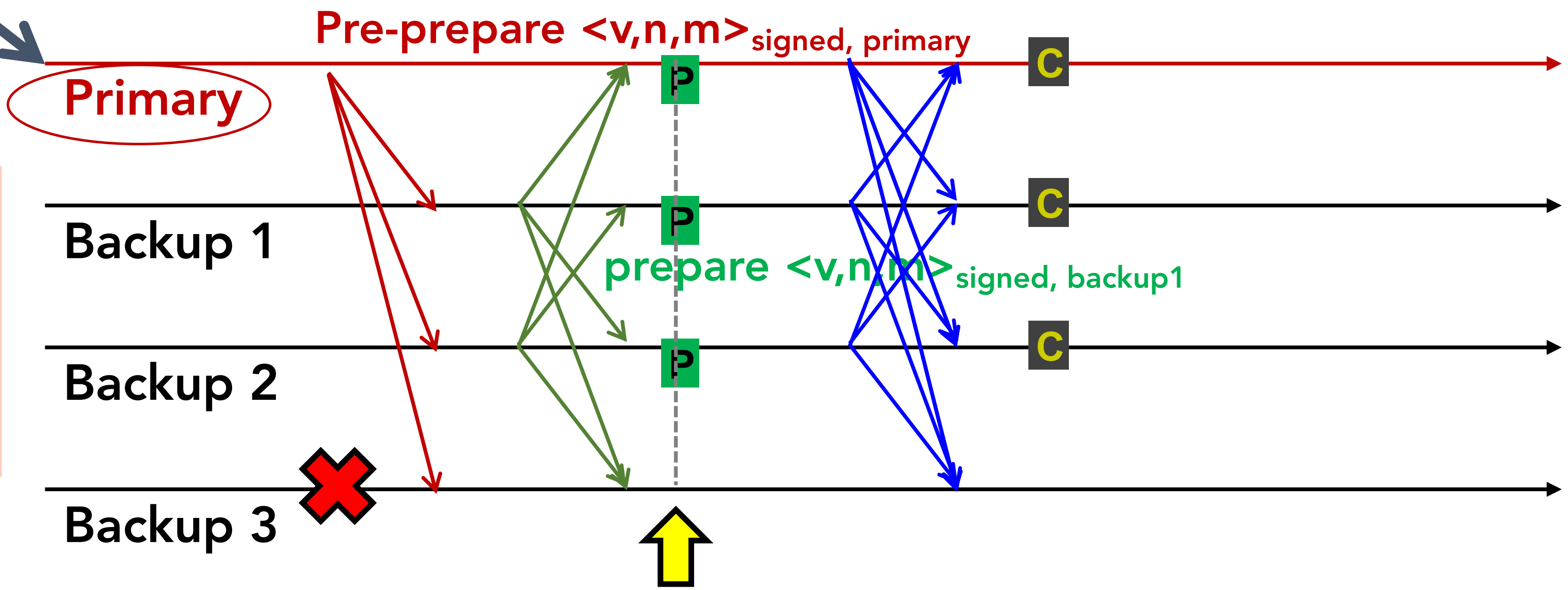
Phase 3: Each backup which has a **prepared certificate** broadcasts a **commit** message



- Backups wait for:
  - **Commit certificate**: a collection of 2f+1 matching commit messages
    - Same view, sequence number and message

Normal Operation (primary is not faulty)

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



Phase 3: Each backup which has a **prepared certificate** broadcasts a **commit** message

- Backups wait for:
  - Commit certificate: a collection of 2f+1 matching commit messages
    - Same view, sequence number and message

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

Byzantine Primary

Primary

Backup 1

Backup 2

Backup 3

Byzantine Primary

Primary

Pre-prepare $\langle v,n,m \rangle_{\text{signed, primary}}$

Backup 1

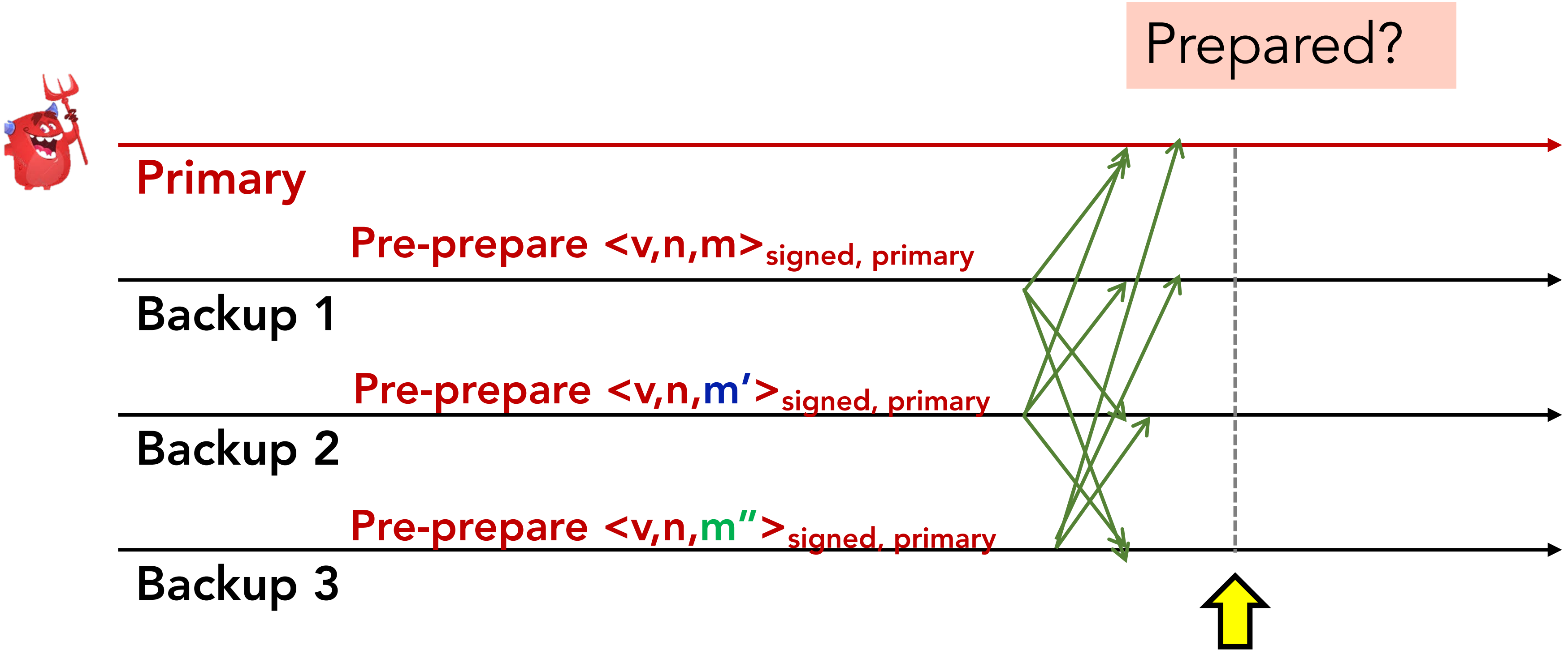
Pre-prepare $\langle v,n,m' \rangle_{\text{signed, primary}}$

Backup 2

Pre-prepare $\langle v,n,m'' \rangle_{\text{signed, primary}}$

Backup 3

Byzantine Primary



Byzantine primary

- In general, backups **won't prepare** 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 $2f+1$) would **intersect** at an **honest** replica
  - So that honest replica would have sent a prepare 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 *view change* request to all replicas
    - Everyone acks the view change request
- New primary collects a quorum of ($2f$+1) responses
  - Sends a ***new-view*** message with this certificate
- Need committed operations to survive into next view
  - Client may have gotten answer
  - View change request contain checkpoints + newer prepare certificates

Other Bits

- Garbage collection
  - Can’t let log grow without bound
  - So shrink log when its gets too big
- Proactive recovery
  - Recover the replica to a known good state whether faulty or not

PBFT Performance

- How much time does it take to commit requests?

- At least 3 Round Trip Network Delays + Disk Writes

Failures can be correlated

- You have to be careful about the assumption of independent failures

Non-deterministic state machines

- Many services may have some form of non-determinism, e.g.,
  - Timers
  - Random numbers

- Paxos/Raft/PBFT by default designed for deterministic state machines

Summary of key ideas

- 2f+1 Quorum
  - We need 2f+1 replicas in a quorum to deal with byzantine faults
  - Assuming a total of 3f+1 replicas with atmost f byzantine faults
- Primary backup with view changes
- Three Rounds
  - Pre-prepare: pick order of request and inform clients
  - Prepare: ensures order within views
  - Commit: ensures order across views
- Replicas directly contact the clients
  - Clients wait for f+1 matching responses

PBFT Conclusion

- Byzantine fault tolerence was for long considered an exotic topic
  - 1980s focused primarily on fail-stop failures
- PBFT showed Byzantine fault tolerance is possible under certain assumptions
- But there were still challenges around performance
  - Challenges: lots of coordination with three phases
  - The paper and a lot of the followup work is about making byzantine fault tolerance more performant

ZooKeeper

ZooKeeper

- General-purpose distributed coordination service
- Many use cases
  - Configuration management, leader election, naming service, data synchronization, etc.
- Used by many companies
  - Yahoo, Yelp, Reddit, Facebook, Twitter, eBay, Rackspace, etc.

Performance Question

- In a replicated system with N replica servers, can we get N times higher performance?

ZooKeeper Setup

ZooKeeper Performance

Servers	100% Reads	0% Reads
13	460k	8k
9	296k	12k
7	257k	14k
5	165k	18k
3	87k	21k

Table 1: The throughput performance of the extremes of a saturated system.