Consensus Algorithms - III

(Practical Byzantine fault tolerance Algorithm

&

Three phase commit Protocol)

Distributed Consensus - Properties

- **Termination**: Every correct individual decides some value at the end of the consensus protocol
- **Validity:** If all the individuals proposes the same value, then all correct individuals decide on that value
- **Integrity:** Every correct individual decides at most one value, and the decided value must be proposed by some individuals
- **Agreement:** Every correct individual must agree on the same value

Synchronous vs Asynchronous Systems

- **Synchronous Message Passing System:** The message must be received within a predefined time interval
  - Strong guarantee on message transmission delay

- **Asynchronous Message Passing System:** There is no upper bound on the message transmission delay or the message reception time
  - No timing constraint, message can be delayed for arbitrary period of times

Asynchronous Consensus

- **FLP85 (Impossibility Result):** In a **purely asynchronous distributed system**, the consensus problem is **impossible** (**with a deterministic solution**) to solve if in the presence of a **single crash failure**.
  - Results by Fischer, Lynch and Patterson (most influential paper awarded in ACM PODC 2001)
  - Randomized algorithms may exist

Correctness of Distributed Consensus

- **Safety:**

  Correct individuals must not agree on incorrect value

  - Nothing bad happened
- **Liveliness (or Liveness):**

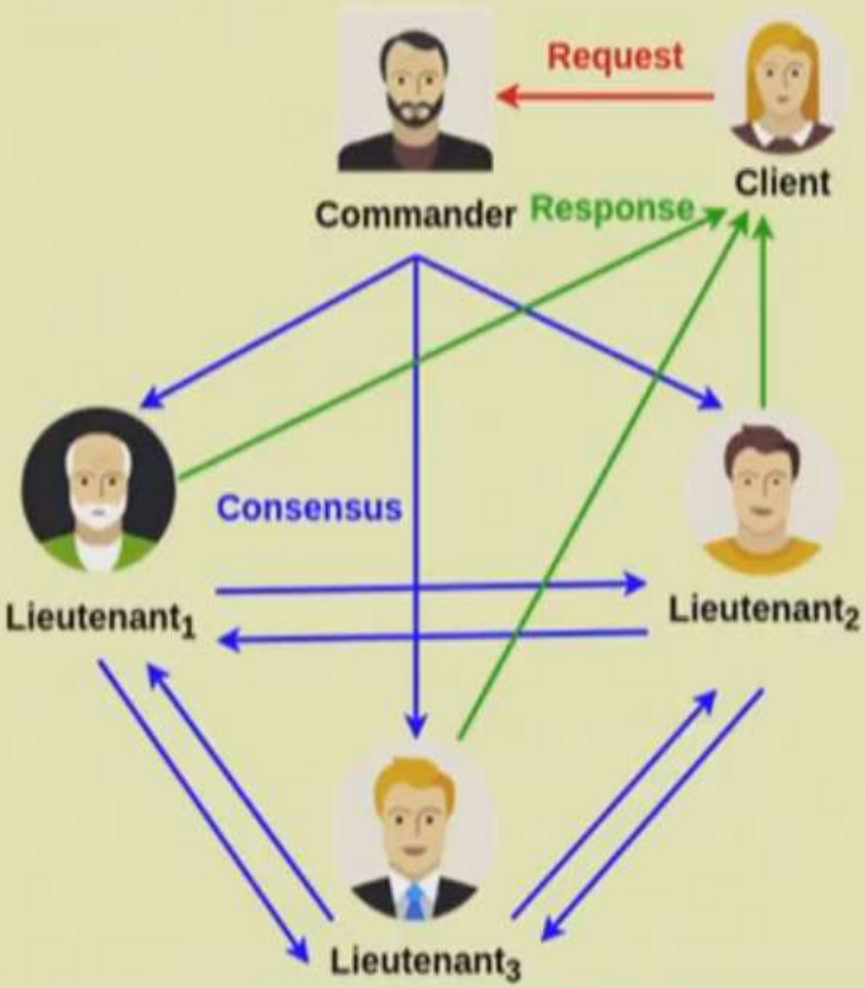
  Every correct value must be accepted eventually

  - Something good eventually happens

Practical Byzantine Fault Tolerant

- Why **Practical**?
  - Ensures safety over an asynchronous network (not liveness!)
  - Byzantine Failure
  - Low overhead
- **Real Applications**
  - Tendermint
  - IBM's Openchain
  - ErisDB
  - Hyperledger

Practical Byzantine Fault Tolerant Model



- Asynchronous distributed system
  - delay, out of order message
- Byzantine failure handling
  - arbitrary node behavior
- Privacy
  - tamper-proof message, authentication

Hashing

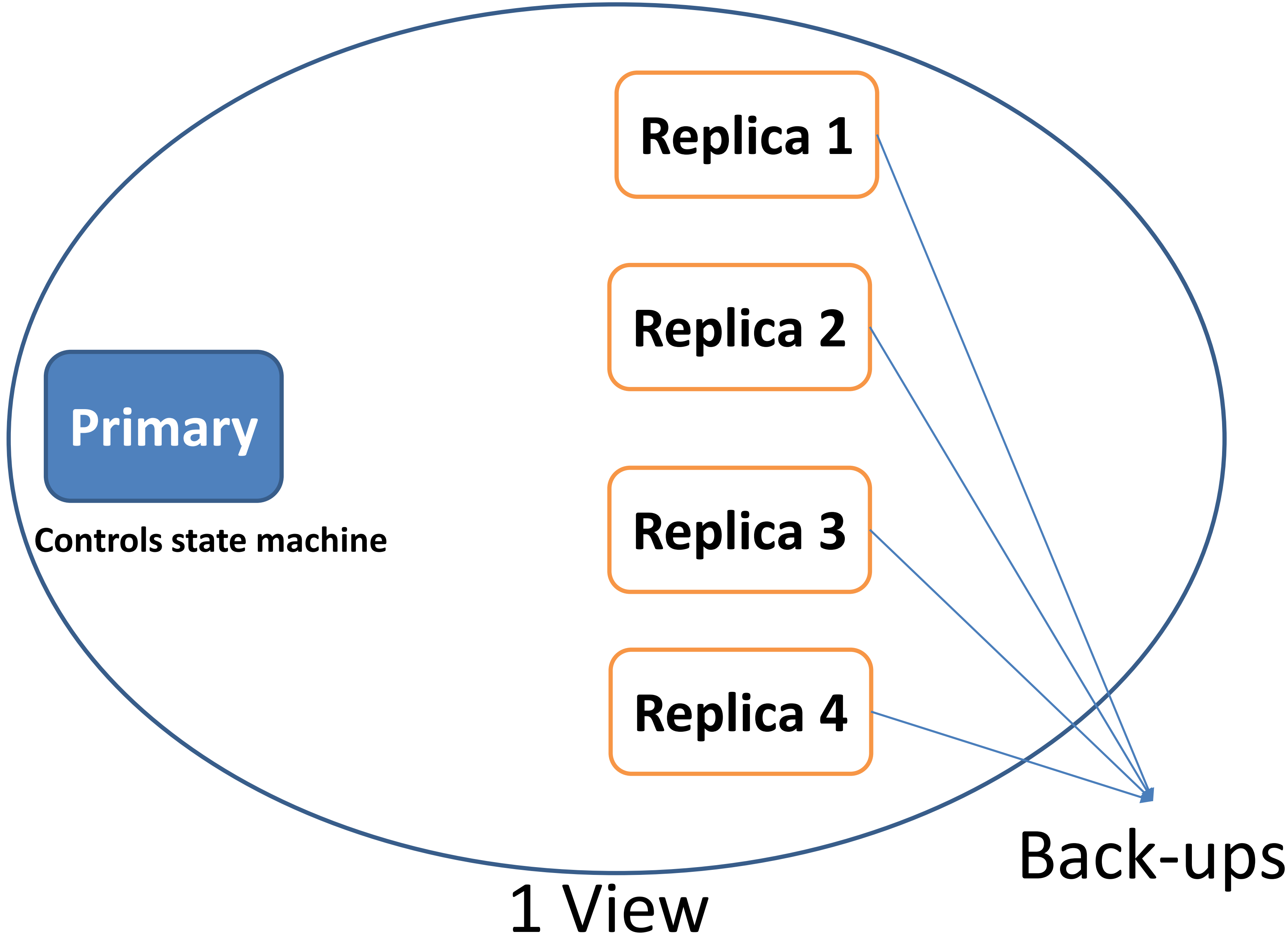
Digital Signature

Practical Byzantine Fault Tolerant Model

- A state machine is replicated across different nodes
- $3f + 1$ replicas are there where f is the number of faulty replicas
- The replicas move through a successions of configurations, known as *views*

States of commanders and lieutenants

- One replica in a *view* is *primary* and others are *backups*

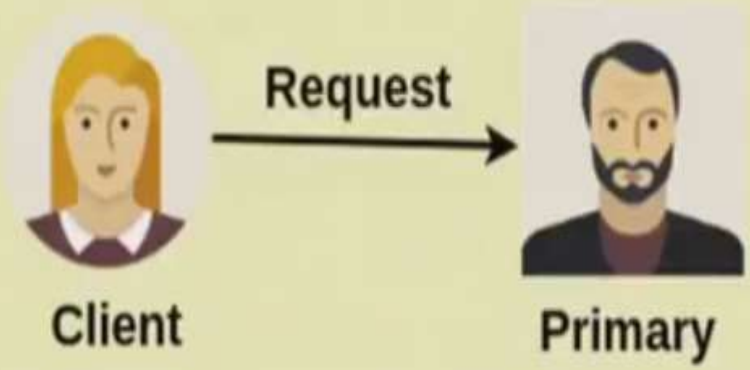


** In another ***View***, primary may become replica and one of the replica may become primary

Practical Byzantine Fault Tolerant Model

- Views are changed when a *primary* is detected as faulty
- Every view is identified by a unique integer number v
- Only the messages from the current views are accepted

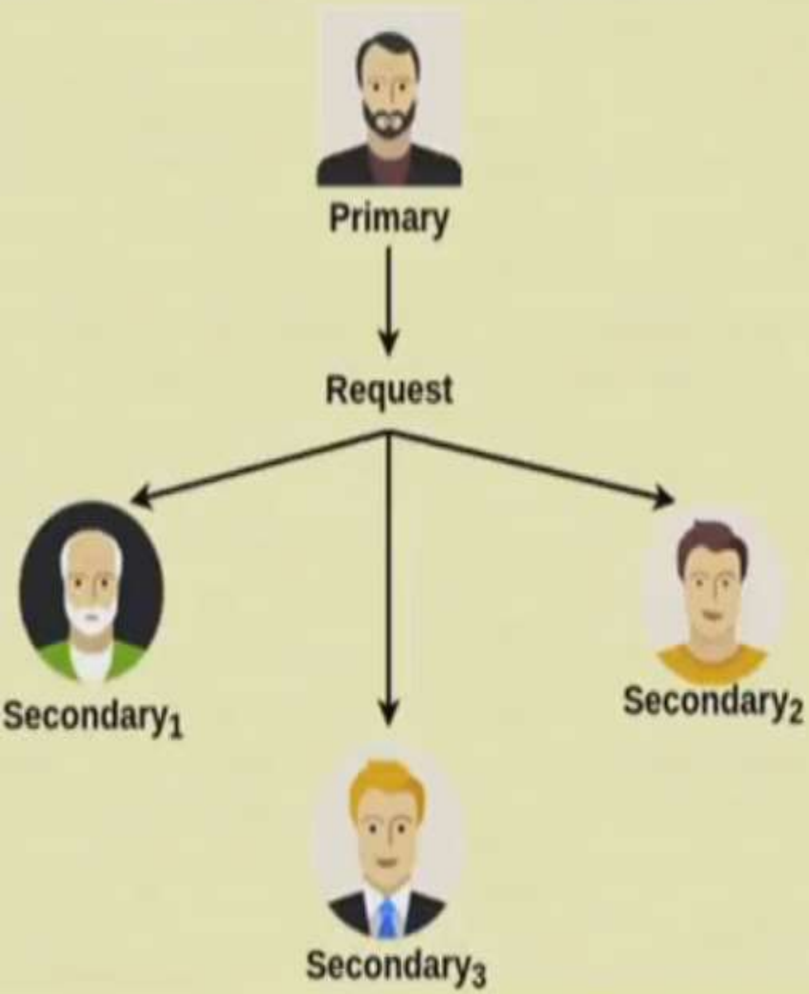
Practical Byzantine Fault Tolerant Algorithm



- A client sends a request to invoke a service operation to the primary

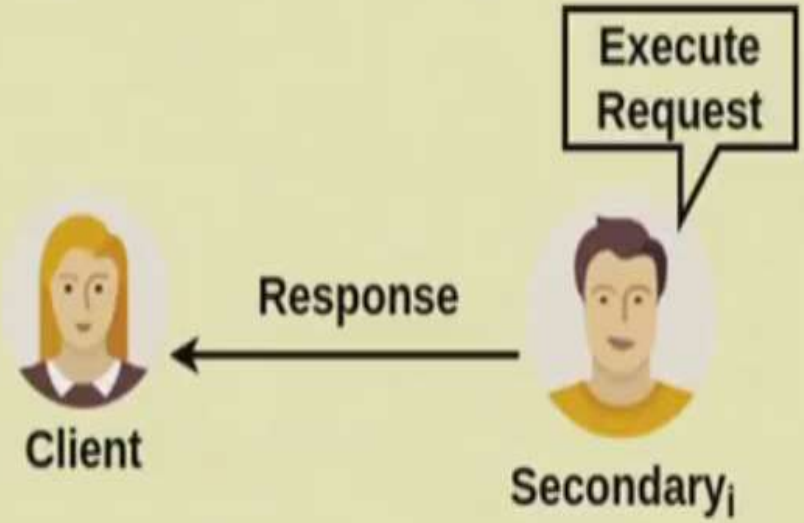
(Transaction request)

Practical Byzantine Fault Tolerant Algorithm



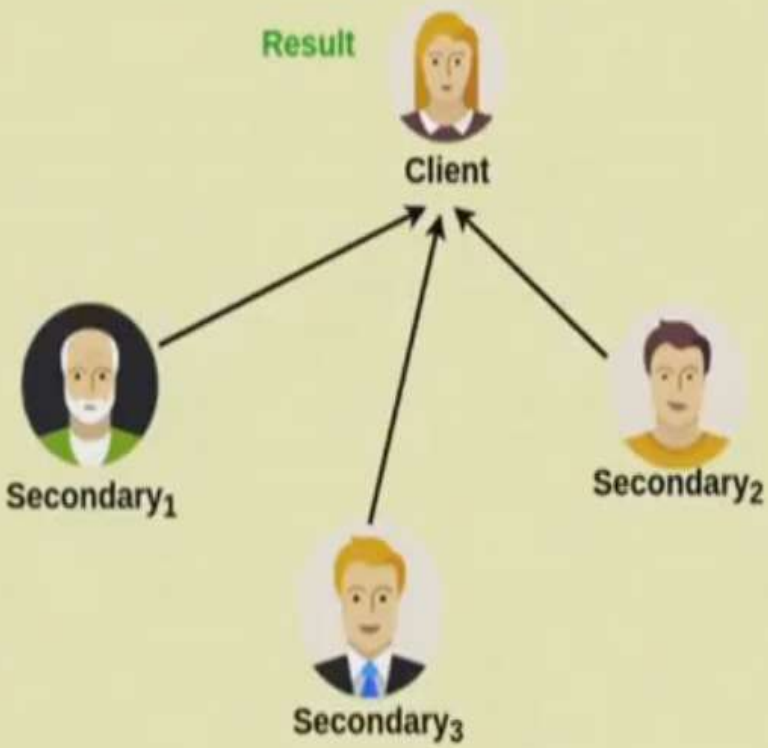
- The primary multicasts the request to the backups

Practical Byzantine Fault Tolerant Algorithm



- Backups execute the request and send a reply to the client

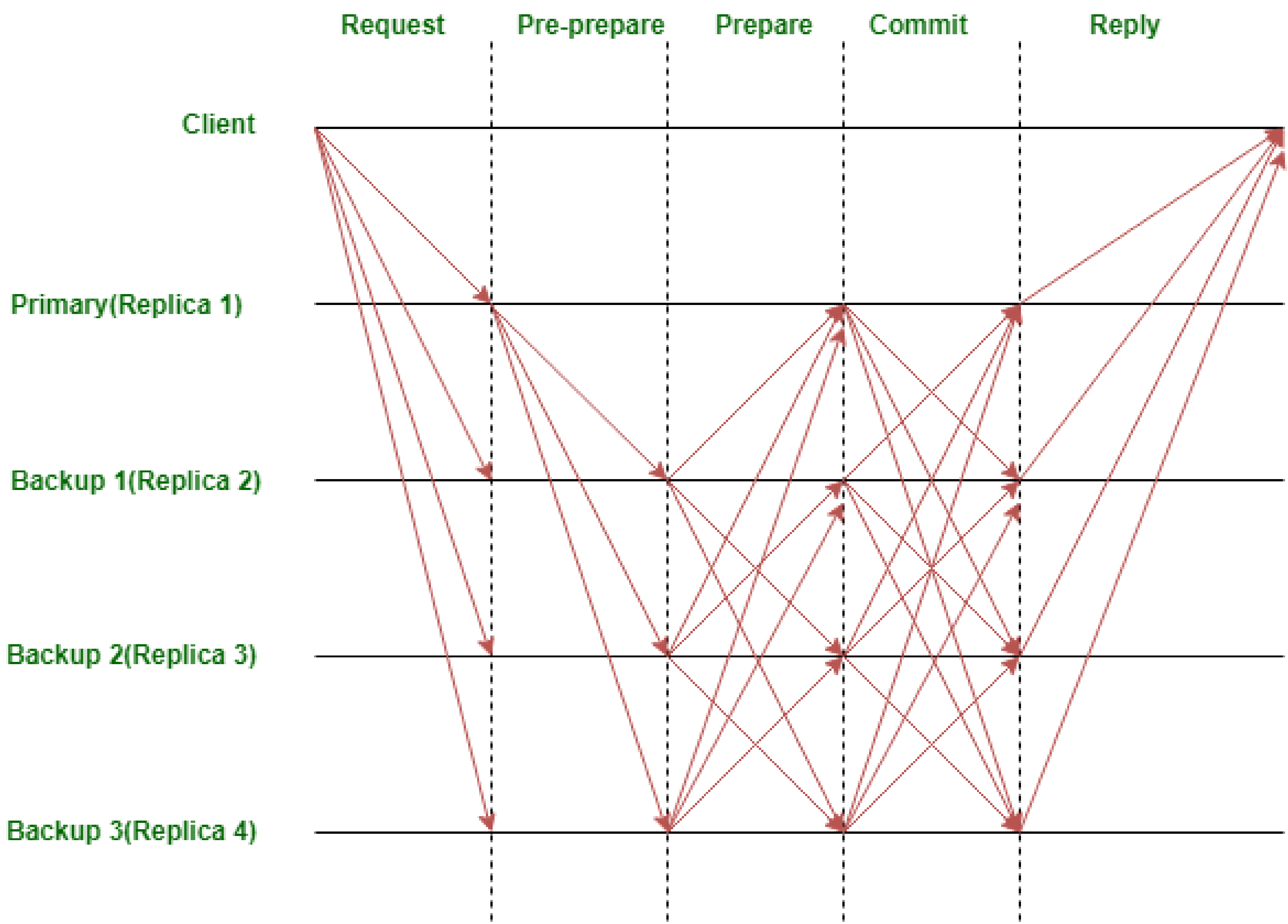
Practical Byzantine Fault Tolerant Algorithm



- The client waits for $f + 1$ replies from different backups with the same result
  - f is the maximum number of faulty replicas that can be tolerated

Total: 3f + 1
Faulty: f
Non-Faulty: 3f + 1 – f => 2f + 1
Majority: f + 1

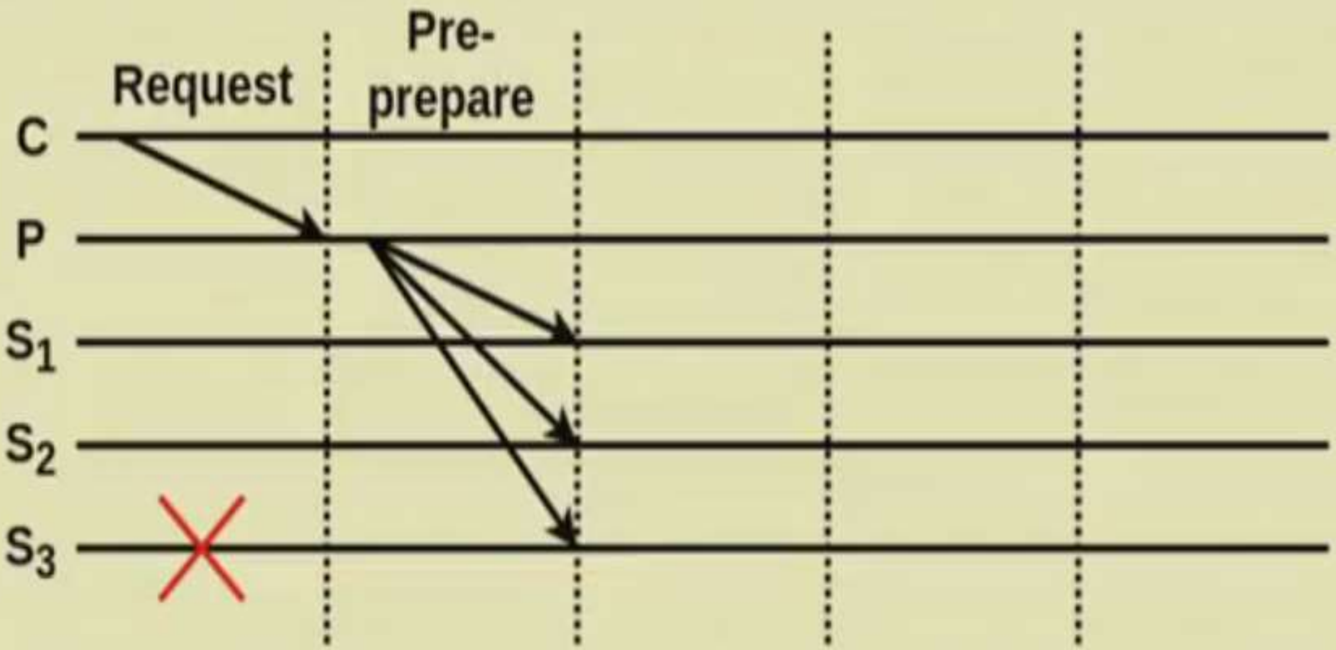
PBFT works in three phases: Pre-prepare, Prepare & Commit


Request
Pre-prepare
Prepare
Commit
Reply
Client
Primary(Replica 1)
Backup 1(Replica 2)
Backup 2(Replica 3)
Backup 3(Replica 4)

Three Phase Commit Protocol - Pre-Prepare

- **Pre-prepare:** Primary assigns a sequence number n to the request and multicast a message $<< PRE-PREPARE, v, n, d >_{\sigma\_p}, m >$ to all the backups
  - v is the current view number
  - n is the message sequence number
  - d is the message digest
  - σ_p is the private key of primary - works as a digital signature
  - m is the message to transmit

Three Phase Protocol

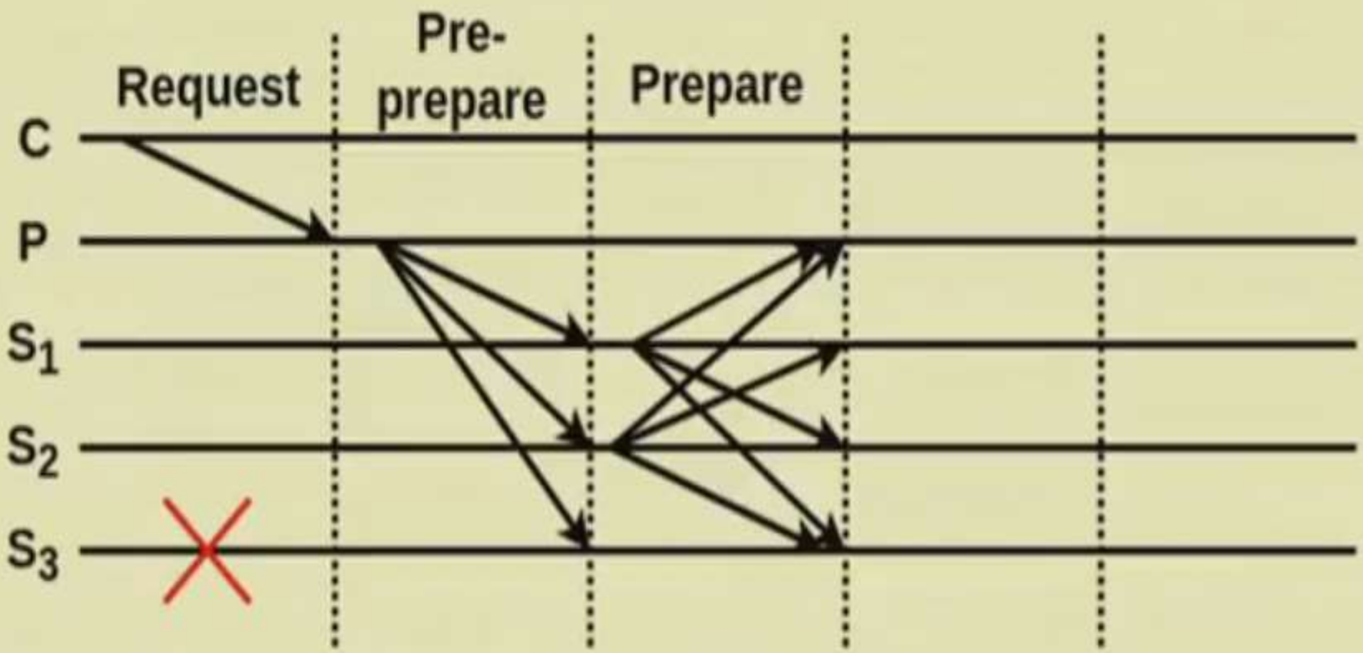


- Pre-prepare:
  - Acknowledge the request by a unique sequence number

Three Phase Commit Protocol - Pre-Prepare

- Pre-prepare messages are used as a proof that request was assigned sequence number n is the view v
- A backup accepts a pre-prepare message if
  - The signature is correct and d is the digest for m
  - The backup is in view v
  - It has not received a different PRE-PREPARE message with sequence n and view v with a different digest
  - The sequence number is within a threshold

Three Phase Protocol



- Prepare:
  - Replicas agree on the assigned sequence number

Three Phase Commit Protocol - Prepare

- If the backup accepts the PRE-PREPARE message, it enters prepare phase by multicasting a message $< PREPARE, v, n, d, i >_{\sigma\_i}$ to all other replicas

- A replica (both primary and backups) accepts prepare messages if
  - Signatures are correct
  - View number equals to the current view
  - Sequence number is within a threshold

Three Phase Commit Protocol

- Pre-prepare and prepare ensure that non-faulty replicas guarantee on a total order for the requests within a view

- Commit a message if
  - $2f$ prepares from different backups matches with the corresponding pre-prepare
  - You have total $2f + 1$ votes (one from primary that you already have!) from the non-faulty replicas

Three Phase Commit Protocol

Why do you require $3f + 1$ replicas to ensure safety in an asynchronous system when there are f faulty nodes?

- If you have $2f + 1$ replicas, you need all the votes to decide the majority - boils down to a synchronous system
- You may not receive votes from certain replicas due to delay, in case of an asynchronous system
- $f + 1$ votes do not ensure majority, may be you have received f votes from Byzantine nodes, and just one vote from a non-faulty node (note Byzantine nodes can vote for or against - You do not know that a priori!)

Three Phase Commit Protocol

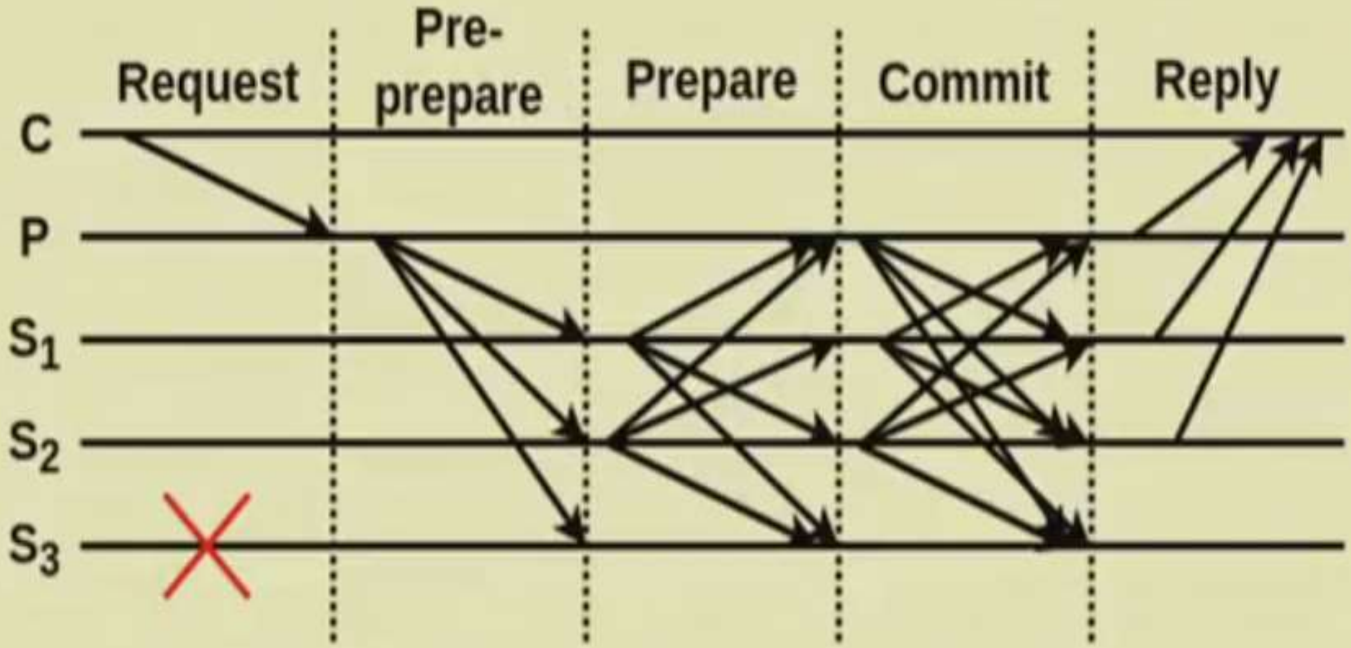
- **Why do you require $3f + 1$ replicas to ensure safety in an asynchronous system when there are f faulty nodes?**
  - If you do not receive a vote
    - The node is faulty and not forwarded a vote at all
    - The node is non-faulty, forwarded a vote, but the vote got delayed
  - Majority can be decided once $2f + 1$ votes have arrived - even if f are faulty, you know $f + 1$ are from correct nodes, do not care about the remaining f votes

Three Phase Commit Protocol - Commit

- Multicast $< COMMIT, v, n, d, i >_{\sigma\_i}$ message to all the replicas including primary

- Commit a message when a replica
  - Has sent a commit message itself
  - Has received $2f + 1$ commits (including its own)

Three Phase Protocol

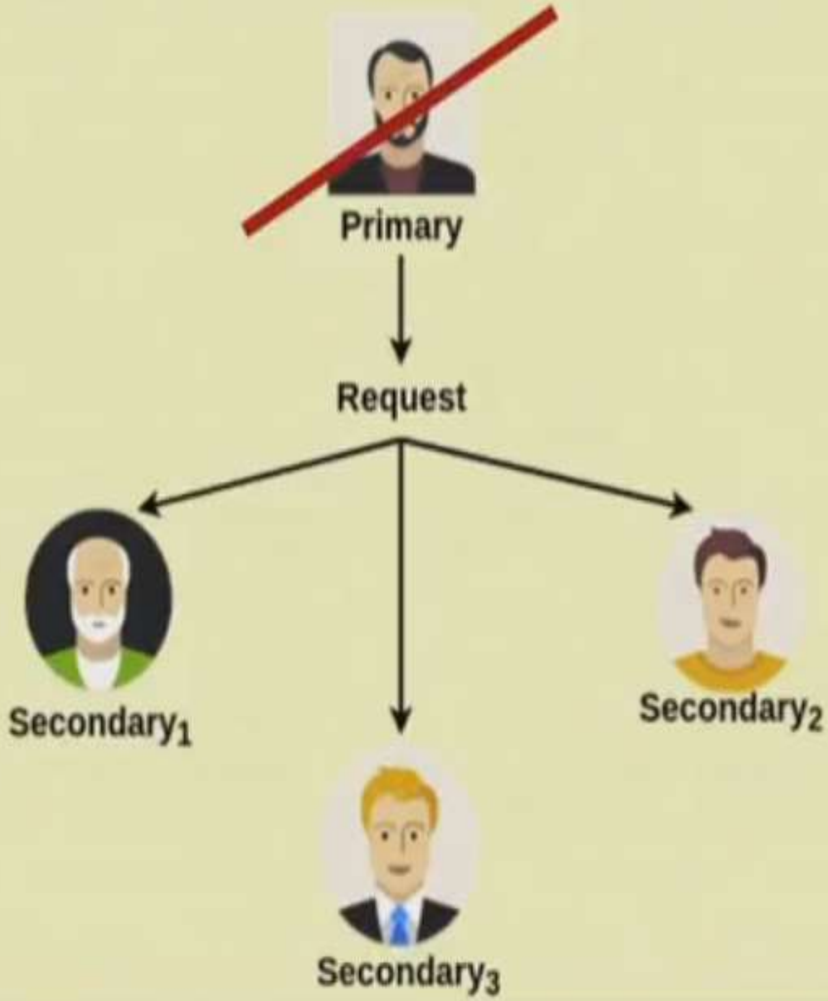


- Commit:
  - Establish consensus throughout the views

View Change

What if the **primary** is **faulty**??

- non-faulty replicas detect the fault
- replicas together start view change operation



Primary Faulty conditions

- **Problem (Case 1)**
- Sequence number 1: INSERT (APPLE) INTO FRUIT

- Sequence number 4: INSERT (PEAR) INTO FRUIT
- Sequence number 5: SELECT * FROM FRUIT The

- replica will be stuck waiting for request with sequence number 2...

- **Problem (Case 2)**
  – Client sends request to primary
  – Primary doesn’t forward the request to the replicas...

View Changes

- View-change protocol provides **liveness**
  - Allow the system to make progress when primary fails
- If the primary fails, backups will not receive any message (such as PRE_PREPARE or COMMIT) from the primary
- View changes are triggered by timeouts
  - Prevent backups from waiting indefinitely for requests to execute

Correctness of Distributed Consensus

- **Safety:**

  Correct individuals must not agree on incorrect value

  – Nothing bad happened

- **Liveliness (or Liveness):**

  Every correct value must be accepted eventually

  – Something good eventually happens

View Change

- Backup starts a timer when it receives a request, and the timer is not already running
  - The timer is stopped when the request is executed
  - Restarts when some new request comes

- If the timer expires at view v
  - Backup starts a view change to move the system to view $v + 1$

View Change

- On timer expiry, a backup stops accepting messages except
  - Checkpoint
  - View-change
  - New-View

View Change

Multicasts a $< VIEW\_CHANGE, v+1, n, \mathcal{C}, \mathcal{P}, i >_{,\sigma\_i}$ message to all replicas

- n is the sequence number of the last stable checkpoint s known to i
- $\mathcal{C}$ is a set of $2f+1$ valid checkpoint messages proving the correctness of s
- $\mathcal{P}$ is a set containing a set $\mathcal{P}_m$ for each request m that prepared at i with a sequence number higher than n
  - Each set $\mathcal{P}_m$ contains a valid pre-prepare message and $2f$ matching

Correctness

- **Safety:** The algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally

Correctness

Liveness: To provide liveness, replicas must move to a new view if they are unable to execute a request

- A replica waits for $2f + 1$ view change messages and then starts a timer to initiate a new view (*avoid starting a view change too soon*)
- If a replica receives a set of $f + 1$ valid view change messages for views greater than its current view, it sends view change message (*prevents starting the next view change too late*)
- Faulty replicas are unable to impede progress by forcing frequent view change

Consensus in Permissioned Model

- PBFT has well adopted in consensus for permissioned blockchain environments
  - Hyperledger
  - Tendermint Core

- Several scalability issues are still there, we'll discuss those in details in the later part of the course!

THANK YOU