Confiabilidade de Sistemas Distribuídos
Dependable Distributed Systems

DI-FCT-UNL, Henrique Domingos, Nuno Preguiça

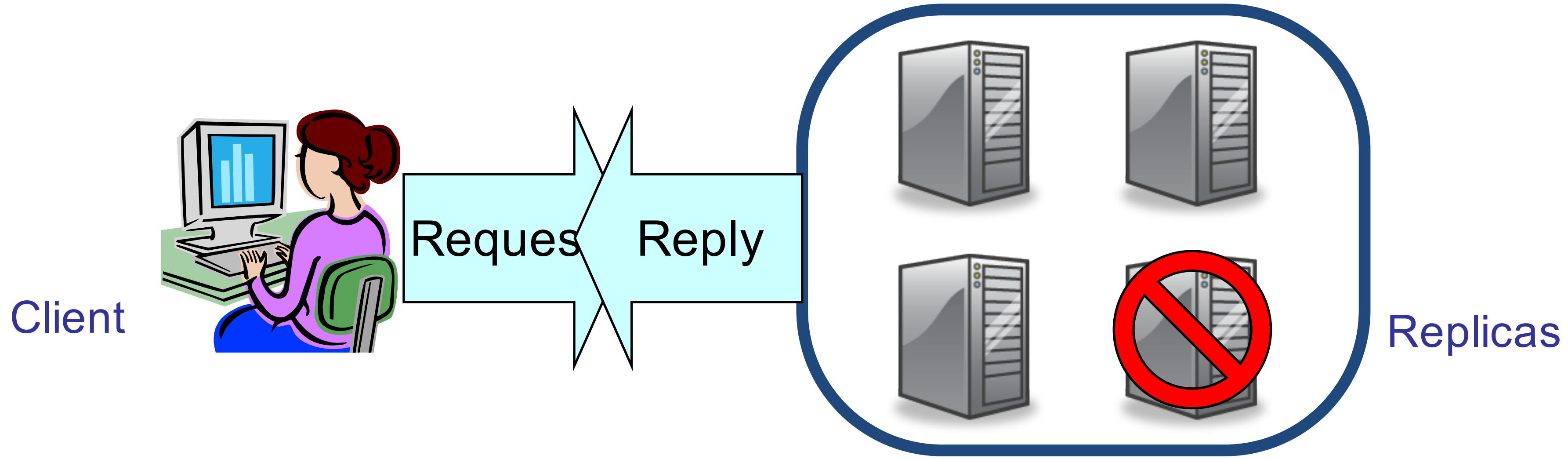
Lect. 3
Byzantine Fault-tolerance

2015/2016, 2nd SEM

MIEI
Mestrado Integrado em Engenharia Informática

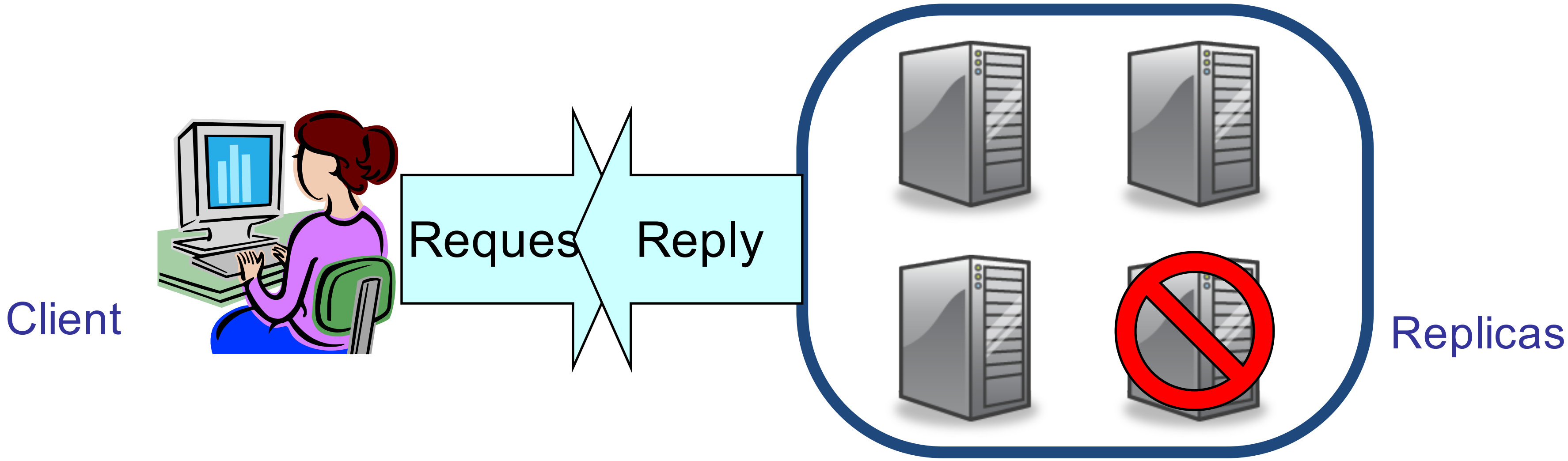
Last lecture: Read/write register replication

1. Service is replicated
2. Operations execute in a quorum of replicas and provide the illusion of a single replica (atomicity)

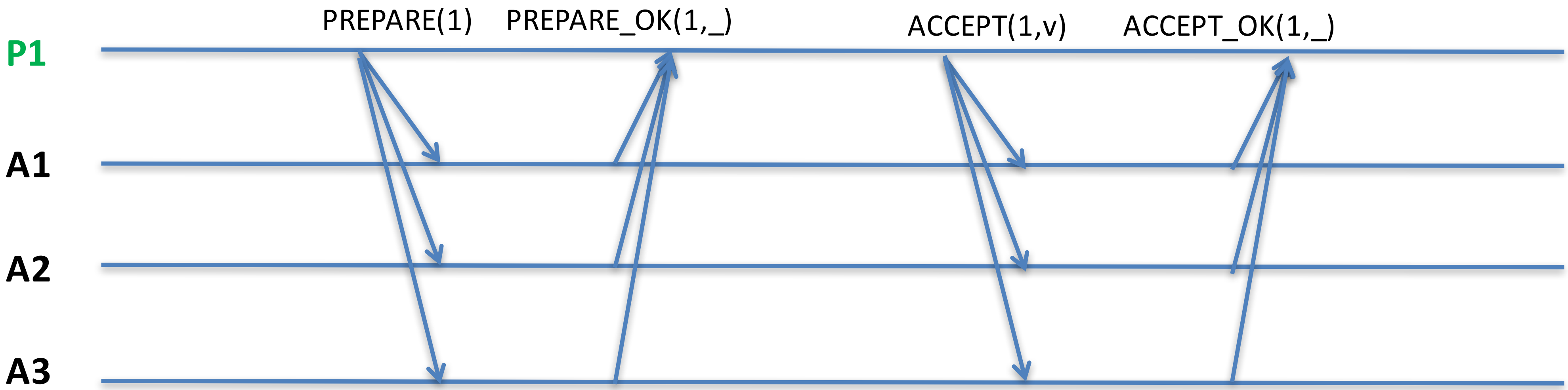


Last lecture: State machine replication (SMR)

1. Service is deterministic (i.e., all operation are deterministic)
2. Service is replicated
3. All correct replicas execute the same sequence of operations



Paxos



Today

- Byzantine fault model
  - Byzantine consensus
- Byzantine fault-tolerant read/write register
- Byzantine fault-tolerant state-machine replication

Byzantine fault model

- Processes that fail can exhibit arbitrary behavior
  - Return wrong replies
  - Take too long to execute a computation step
  - Do not follow the communication protocol
  - Collude with other processes

Why is the model interesting?

- Model addresses behavior due to:
  - Software bugs
  - Memory/disk corruption
  - Overloaded machine
- Additionally addresses malicious behavior of machines controlled by an attacker

Common assumption when dealing with Byzantine faults

- Only a subset of the machines exhibits arbitrary behavior
- It is impossible to break cryptographic primitices
  - Cannot lead to hash collisions
  - Cannot forge digital signatures nor authenticators
- Cannot directly change the state of other processes
- Can replay old authenticated messages

Minimum number of processes for consensus

- It is impossible to solve consensus with n processes and f Byzantine faults if n 3f

Byzantine Consensus

- Inputs: each process has its initial proposal in variable v_i
- Outputs: each process has an output variable $decision_i$, initially *null*
- C1 [Validity] If all correct processes have $v_i = v$, then v is the only allowed output
- C2 [Agreement] Two correct processes cannot decide different values
- C3 [Termination] All correct processes eventually decide
- C4[integrity] If a correct process decides v, then v was the initial proposal of some process

Today

- Byzantine fault model
  - Byzantine consensus
- Byzantine fault-tolerant read/write register
- Byzantine fault-tolerant state-machine replication

ABD: State and write algorithm

- State
  - val_i → value of the variable, initially v0
  - tag_i → pair <number of sequence,id> initially <0,0>
    - <s1,i1> > <s2,i2> iff s1 > s2 || (s1 == s2

Problem 1
If process can fake their identity, how to know that we have received a quorum of replies?

- Client c : Write(v)
  - Step 1:

  Send(<read-tag>) to all processes (or to a quroum)

  Wait for a quorum Q of replies

  Let seqmax = max{sn : <sn,id>   Q }
  - Step 2:

  Send(<write(<seqmax+1,c>,v)>) to all pr

  Wait for a quorum of acks

Solution
Use authenticated channels

ABD: State and write algorithm

- State
  - val_i → value of the variable, initially v0
  - tag_i → pair <number of sequence,id> initially <0,0>
    - <s1,i1> > <s2,i2> iff s1 > s2 || (s1 == s2

Problem 2
Replica in the intersection of two quorums can be Byzantine

- Client c : Write(v)
  - Step 1:
  Send(<read-tag>) to all processes (or to a quroum)
  Wait for a quorum Q of replies
  Let seqmax = max{sn : <sn,id> Q }
  - Step 2:
  Send(<write(<seqmax+1,c>,v)>) to all processes (or to a quroum)
  Wait for a quorum of acks

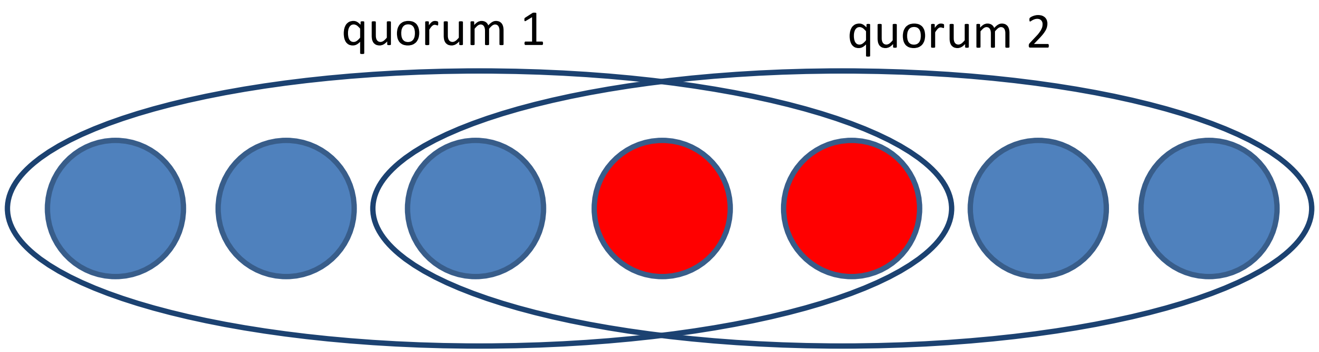
Solution
Need to have larger quorums

Byzantine quorums

- What is the size of quorums and the number of replicas?

- (i) Quorums cannot have more than n-f replicas. Why?
- Otherwise it could be impossible to get a quorum: Byzantine replicas may never reply
- (ii) Every two quorums must intersect in at least one correct replica
- (i) Q <= N-f
- (ii) N – (N-Q) – (N-Q) >= f+1

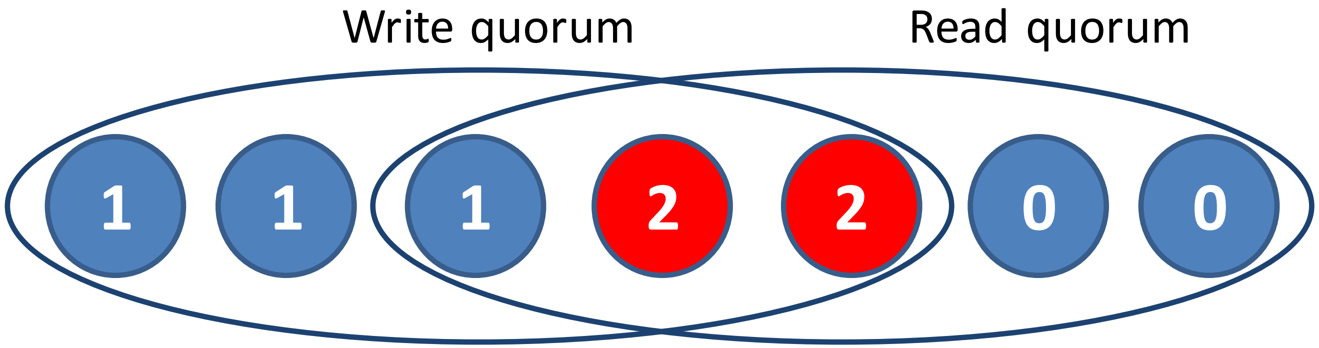
Optimal solution: N = 3f+1, Q=2f+1



Is this enough for read/write registers?

- Consider there are no writes executing
- Which (type of) values can be returned in a read quroum?
  - Correct and actual value (at least how many?)
  - Correct but old values
  - Incorrect values (returned by Byzantine replicas)

Example



Solution: clients sign writes

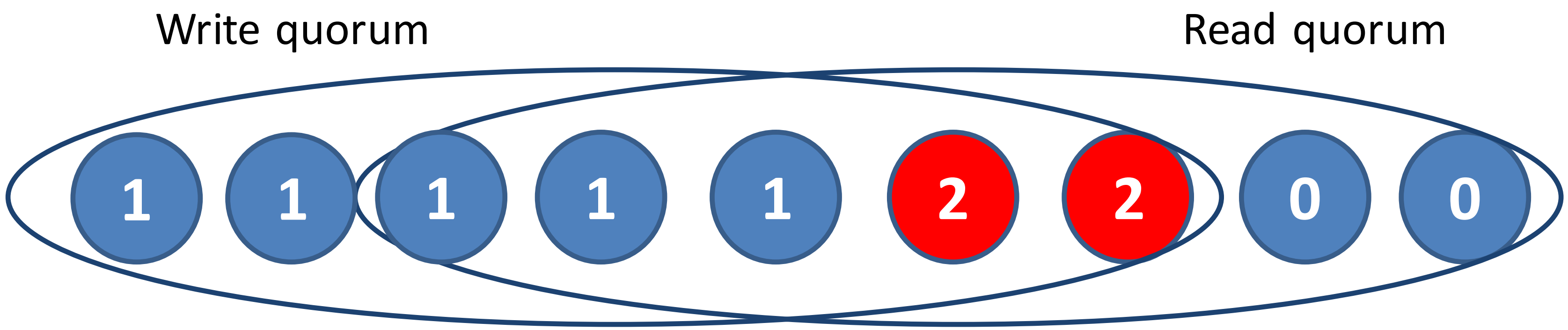
- In the write, the client signs the pair <tag,valor>
- Replicas store and return the signature
- On read, the client discards replies with invalid signatures
- Need to send nonce with each request/reply to avoid “replay attacks”

Alternative solution: larger quorums

- Guarantee that correct actual values have larger votes that incorrect votes
- Quorums must intersect in 2f+1 replicas
  - Intersection has, in the worst case, f+1 correct replicas and f Byzantine
- Requires n=4f+1, Q=3f+1
- Read result is the largest value returned by >= f+1 replicas

- Problem: it might be impossible to find f+1 equal values. In which case?

Example



ABD: State and write algorithm

- State
  - val_i → value of the variable, initially v0
  - tag_i → pair <number of sequence,id> initially <0,0>
  - sig_i → signature of <val_i, tag_i >

Why is te nonce needed?

- Client c : Write(v)

  Generate nonce

  - Step 1:

  Send(<read-tag(nonce)>) to all processes (or to a quroum)

  Wait for a quorum Q of valid replies (with nonce and authenticated)

  Let seqmax = max{sn : <sn,id,sig> Q }

  - Step 2:

  Send(<write(<seqmax+1,c>,v,sig,nonce)>) to all processes (or to a quroum) ,
  with sig = sign(<<seqmax+1,c>,v>)

  Wait for a quorum of valid acks with the given nonce

ABD: Algorithm for replica i

- on_recv(<read_tag(nonce)>)
  - Return <tag_i ,val_i, sig_i , nonce>

- on_recv(<write(new-tag,new-val,new-sig,nonce)>)
  - If valid(newsig,<new-tag,new-val>) new-tag > tag_i then
    - tag_i = new tag
    - val_i = new-val
    - sig_i = new-sig
  - Return ack

- on_recv(<read(nonce)>)
  - Return <tag_i,val_i,sig_i,nonce>

ABD: Algorithm for read

- Client c : Read()

  Generate nonce

  – Step 1:

  Send(<read(nonce)>) to all processes (or to a quroum)

  Wait for a quorum Q of valid replies (with nonce and authenticated)

  Let <tagmax,valmax,sigmax>   Q be the reply with largest tagmax

  – Step 2:

  Send(<write(tagmax,valmax,sigmax,nonce)>) to all processes (or to a quroum)

  Wait for a quorum of valid acks

  Return valmax

Today

- Byzantine fault model
  - Byzantine consensus
- Byzantine fault-tolerant read/write register
- Byzantine fault-tolerant state-machine replication

Pratical Byzantine Fault-Tolerance (BFT)

- Replication algorithm that tolerates Byzantine faults
  - State-machine replication
    - The same sequence of operations is executed in all replicas
    - Guarantees that all coorrect replicas will converge to the same state
  - Can be used as a basis for repplicating any service (e.g. NFS, DB)
    - Operations can be generic, assuming that they are deterministic

- First algorithm to show that Byzantine fault-tolerance can be practical
  - i.e., that it can be implemented without prohibitive overhead

- System requires 3f+1 nodes to tolerate f failures

System model

- Asynchronous distributed system
  - Network may fail to deliver messages, delay them, duplicate them, or deliver them out of order
  - If messages are retransmitted, they will be eventually delivered to the destination

- Byzantine fault model
  - Nodes may behave arbitrarily
  - Faulty nodes may collude for attacking the system

- Uses public-key cryptography: all messages are signed
  - Nodes know each other's public key
  - Attacker cannot subvert cryptographic techniques used

Protocol basis

- Protocol proceeds in a sequence of views
  - All views have the same nodes

- For a given view, a particular node is designated as the primary node; other nodes are backup nodes
  - Primary = v mod n
    - N is number of nodes
    - V is the view number

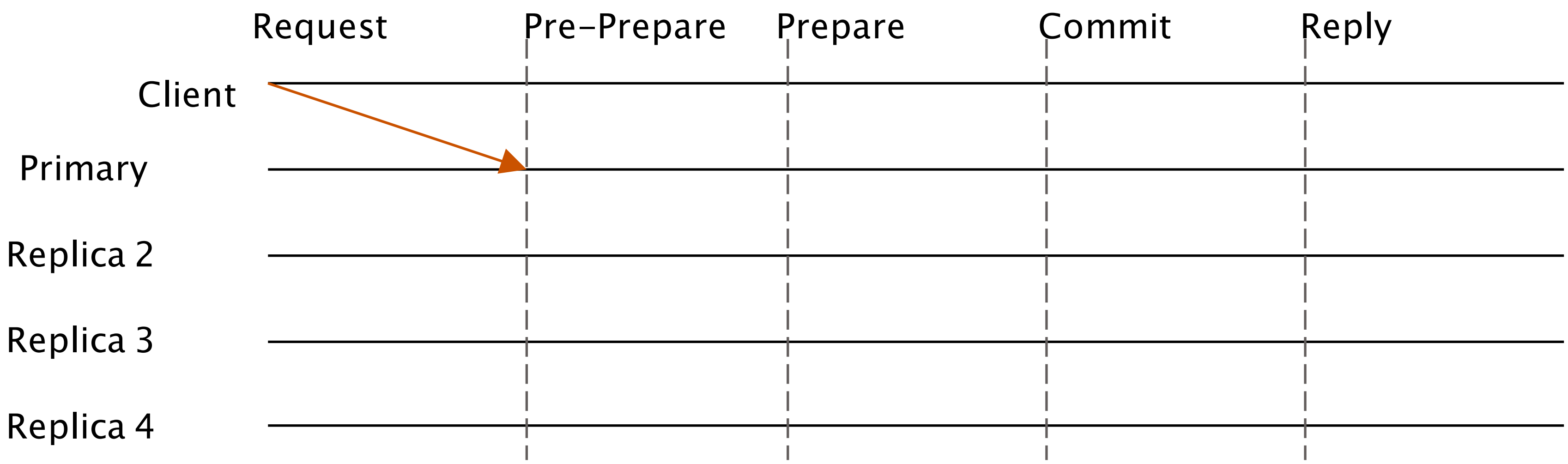
- Each node maintains the following state
  - Log
  - View number
  - Service state

Protocol basis (cont.ed)

- Protocol strategy
  - Primary runs the protocol in the normal case
  - Replicas *watch* the primary and do a view change if it fails

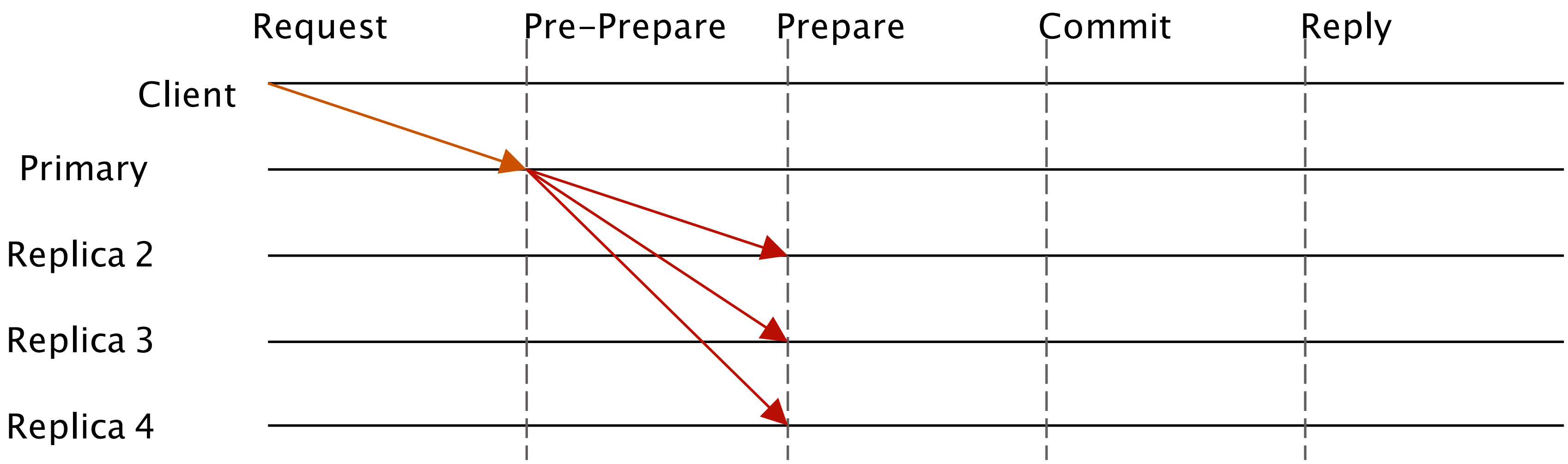
- Protocol in three phases
  - Client sends message to primary
  - **Pre-prepare**: Primary proposes an order
  - **Prepare**: Backup copies agree on #
  - **Commit**: agree to commit
  - Replicas reply directly to the client

Protocol: normal case



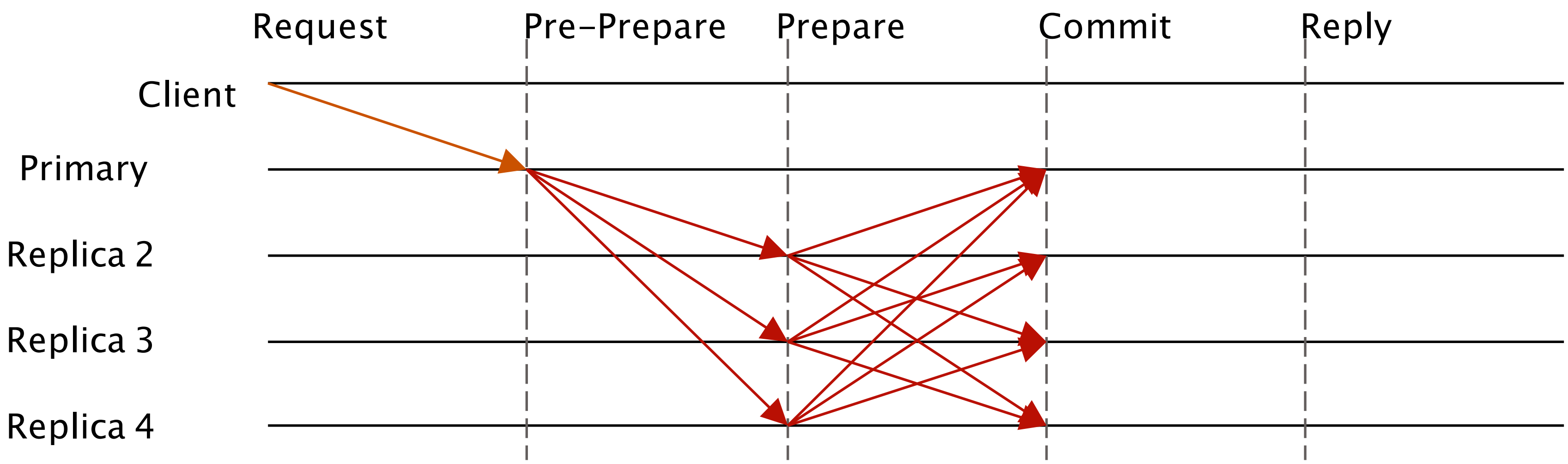
- Client starts by sending the request to the expected primary
- Primary check if the request is valid

Protocol: normal case



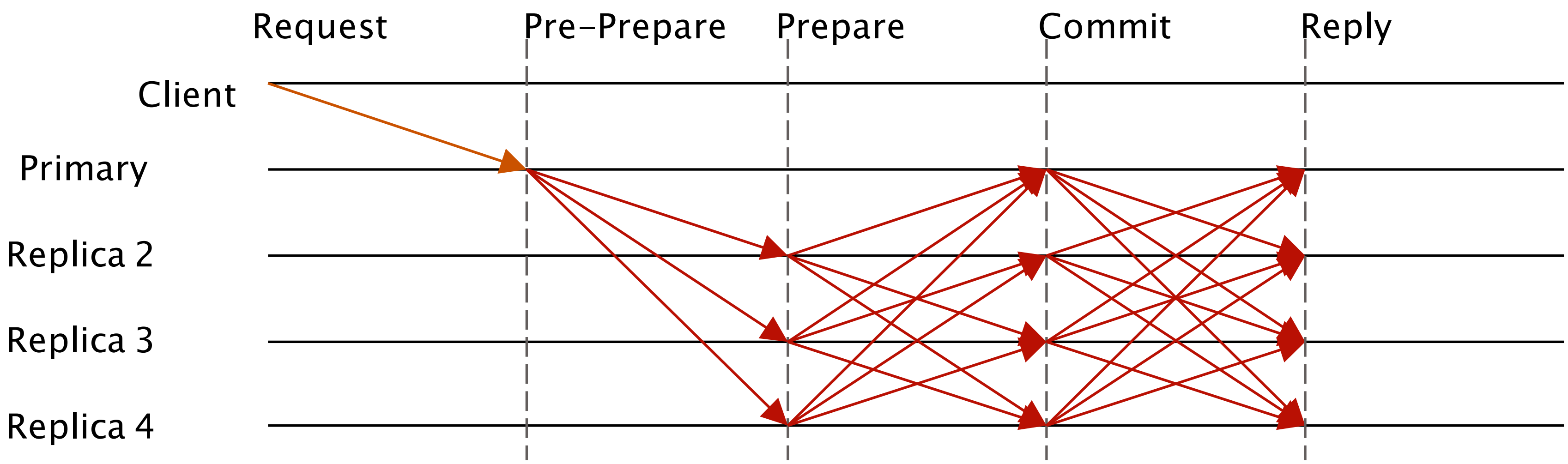
- Primary sends **pre-prepare** message to all
- **Pre-prepare** contains <view#,seq#,op>
  - Primary records operation in log as pre-prepared

Protocol: normal case



- Replicas check the pre-prepare and if it is ok (signed, no previous pre-prepare with the same seq #):
  - Record operation in log as pre-prepared
  - Send **prepare** messages to all
  - **Prepare** from replica i contains <i,view#,seq#,op>

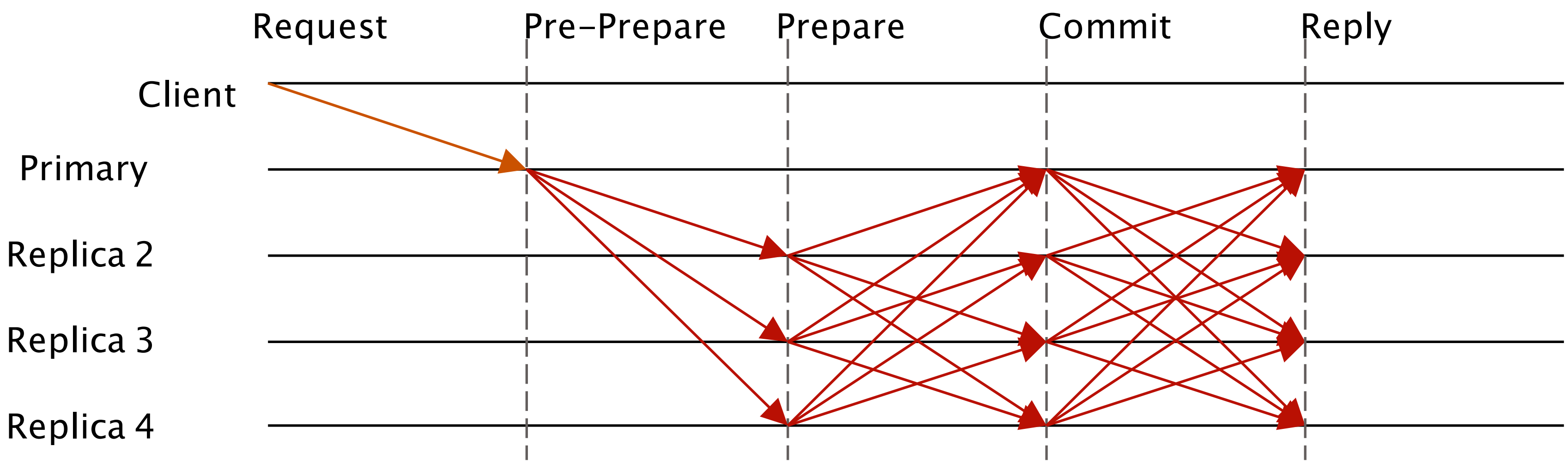
Protocol: normal case



- Replicas wait for 2f+1 matching prepares
  - Record operation in log as prepared
  - Send **commit** message to all
  - **Commit** contains <i,view#,seq#,op>

What does a replica know when it has received 2f+1 matching prepares?
It knows that f+1 correct replicas agreed on ordering the operation with the given seq#

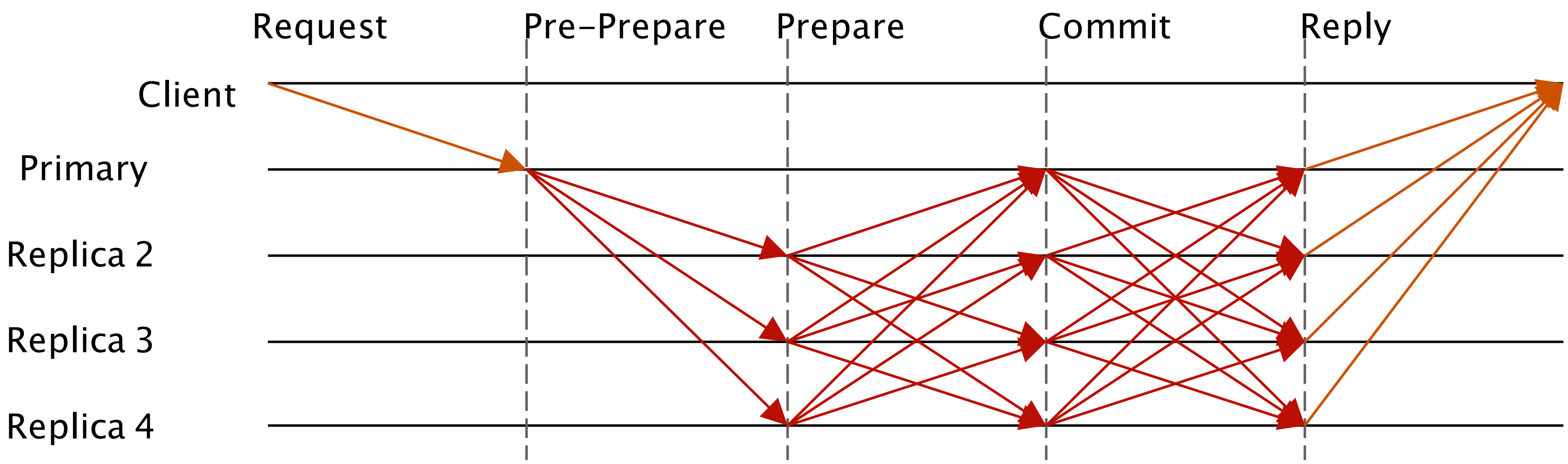
Protocol: normal case



- Replicas wait for 2f+1 matching prepares
  - Record operation in log as prepared
  - Send **commit** message to all
  - **Commit** contains <i,view#,seq#,op>

Why cannot execute operation immediately?
In a view change, the information that an order has been agreed might be lost.

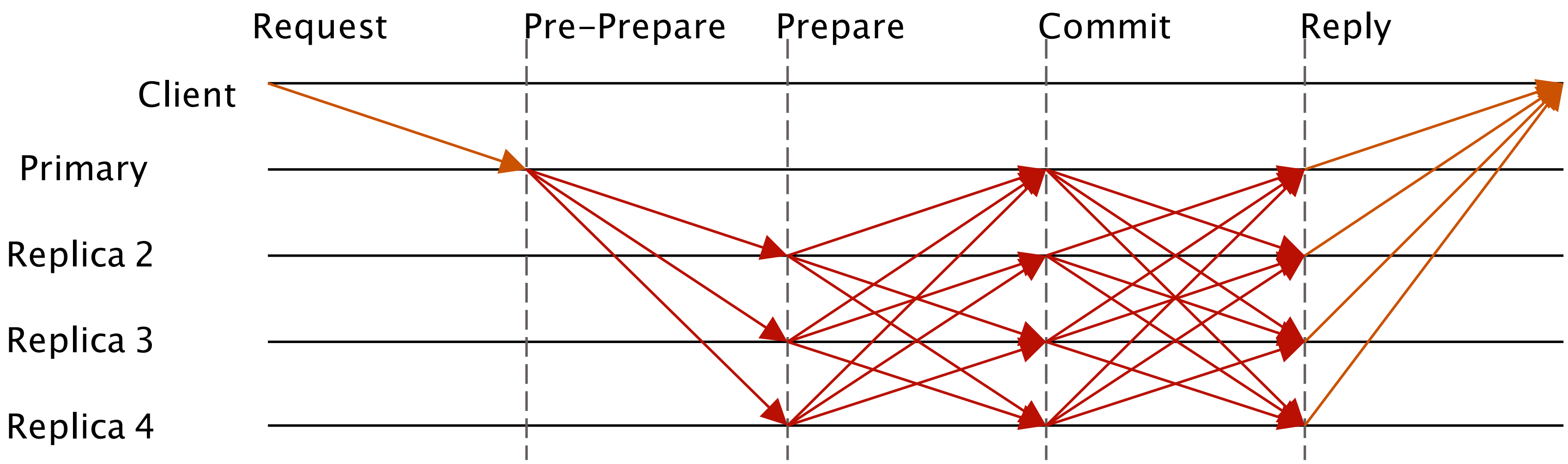
Protocol: normal case



- Replicas wait for 2f+1 matching commits
  - Record operation in log as committed
  - Execute the operation
  - Send result to the client

What does a replica know when it has received 2f+1 matching commits?
It knows that f+1 correct replicas prepared to execute the operation

Protocol: normal case



- Client waits for f+1 matching replies

What does the client know when it has received f+1 matching replies?
It knows that: f+1 correct replicas prepared to execute the operation with some seq# and that the returned result is correct (as it has been returned by at least one correct replica)

Correctness

- Safety:
  - Correct replicas cannot execute a wrong step (influenced by faulty ones)? Why?

- Liveness:
  - It is guaranteed that the system makes progress? Why?

Protocol: view change

- Backups watch the primary
- If some backup suspects the Primary, it calls for a view change
  - When a backup receives a valid view change request it starts a timer (if it is not running)
  - When the timer expires, the Primary must be faulty. Decide to change view.

  - If backups receive requests from the primary, when receiving no request, how will it be suspected?
    - Clients that do not receive a reply send the request to all servers

Protocol: view change

- A backup sends a view-change message
  - Request includes check-pointing information + messages prepared
- When the primary of the new view receives 2f view-change messages from other replicas
  - Declares the new view
  - Send a new-view message, including a proof that 2f+1 nodes decided to change the view
  - The new-view message includes also messages that were not completed in the previous view

Practical aspect

- Operation only sent in the pre-prepare message
  - Other messages carry an hash of the operation
- Cryptography
  - Instead of signing every message with public key crypto, it is possible to use na array of authenticators (hash signed with symettric key)

Improved Performance

- Fast reads (one round trip)
  - Client sends to all; they respond immediately
  - Client waits for 2f+1 matching responses