



DD245 I  
Parallel and Distributed Computing

---

FDD3008  
Distributed Algorithms

Lecture 9  
Fault Tolerance

Mads Dam  
Autumn/Winter 2011

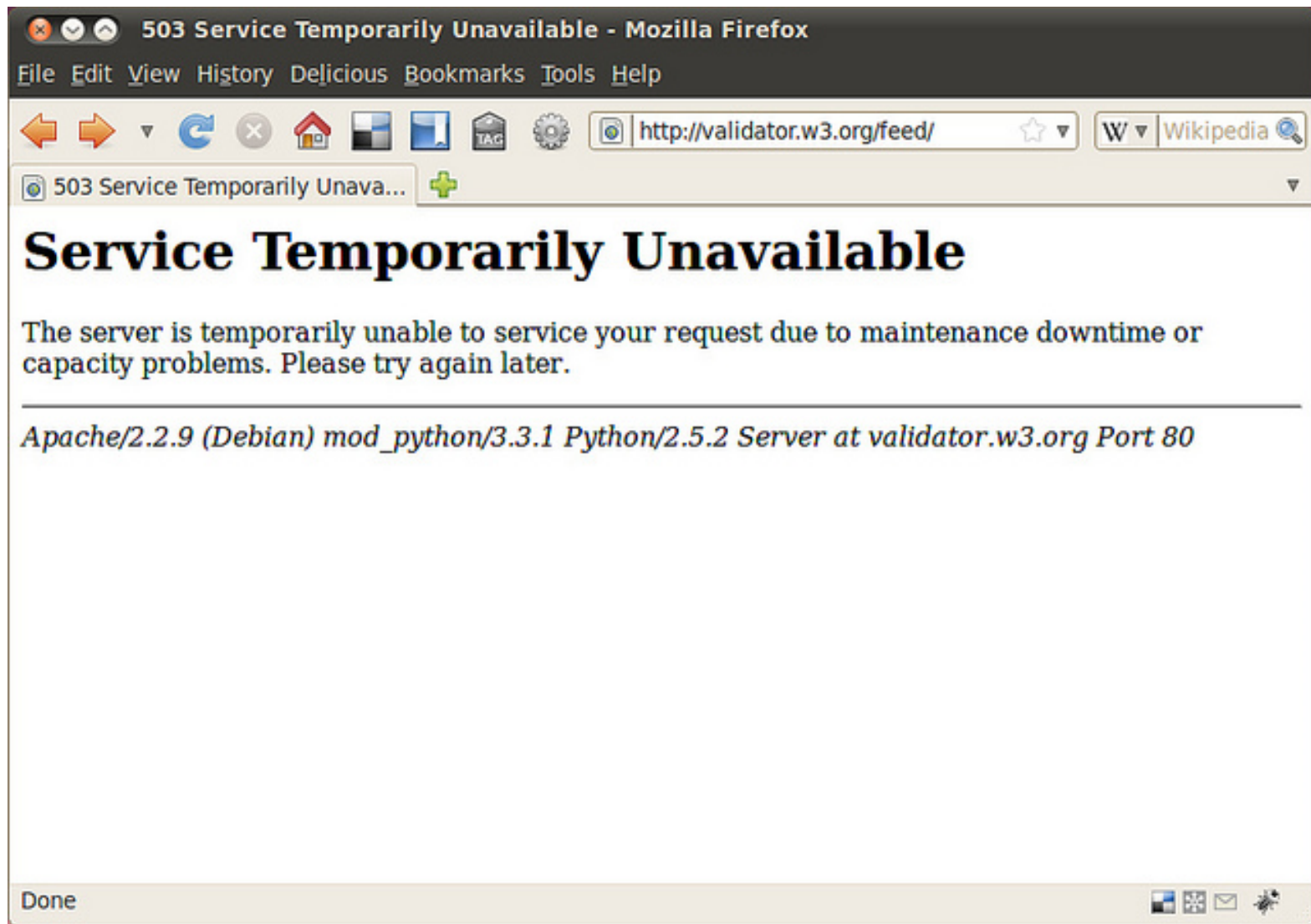
Much material due to R. Wattenhofer

# Overview

- Introduction
- Strong Consistency
  - Crash Failures: Primary Copy, Commit Protocols
  - Crash-Recovery Failures: Paxos
- Crash recovery, practice:
  - Chubby
- Byzantine failures:
  - PBFT, Zyzzyva

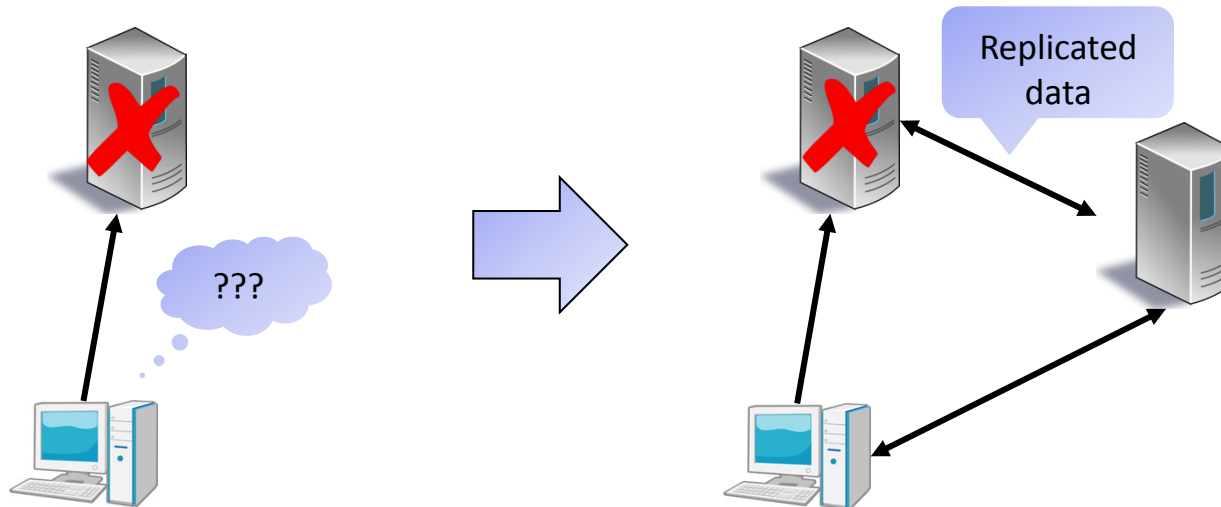
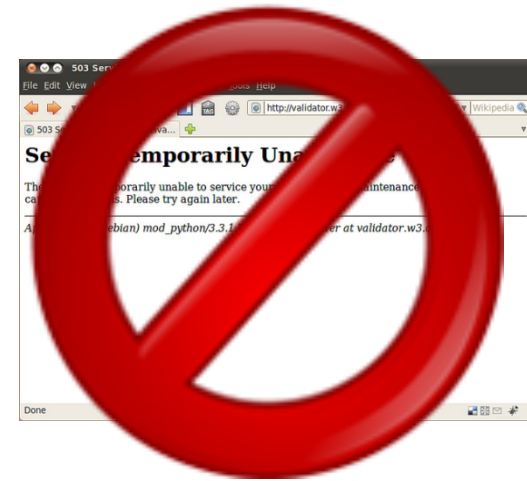
# Computability vs. Efficiency

- In the last part, we studied computability
  - When is it possible to guarantee consensus?
  - What kind of failures can be tolerated?
  - How many failures can be tolerated?
- What about practice?
  - Simple approaches that work well in practice
  - Often delay bounds are reasonable
  - Sometimes full consistency is too much
  - Need to produce practical, cost-effective systems



# Fault-Tolerance in Practice

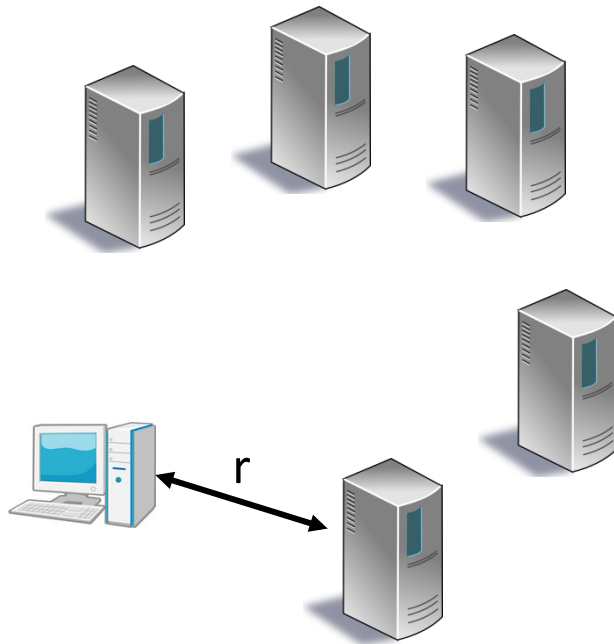
- Fault tolerance is achieved through replication



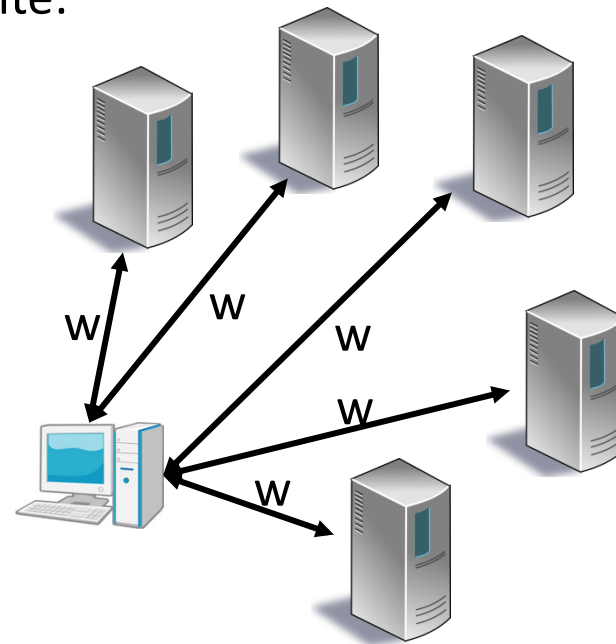
# Replication Is Expensive

- Reading a value is simple -just query any server
- Writing is more work - inform all servers about the update
- What if some servers are not available?

Read:



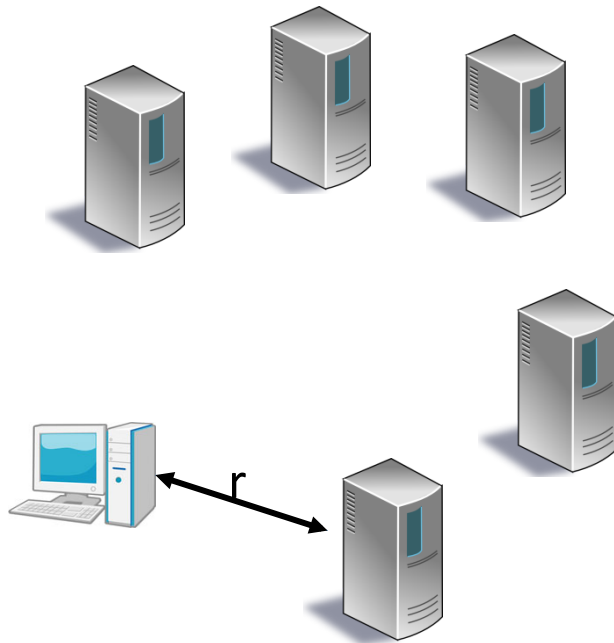
Write:



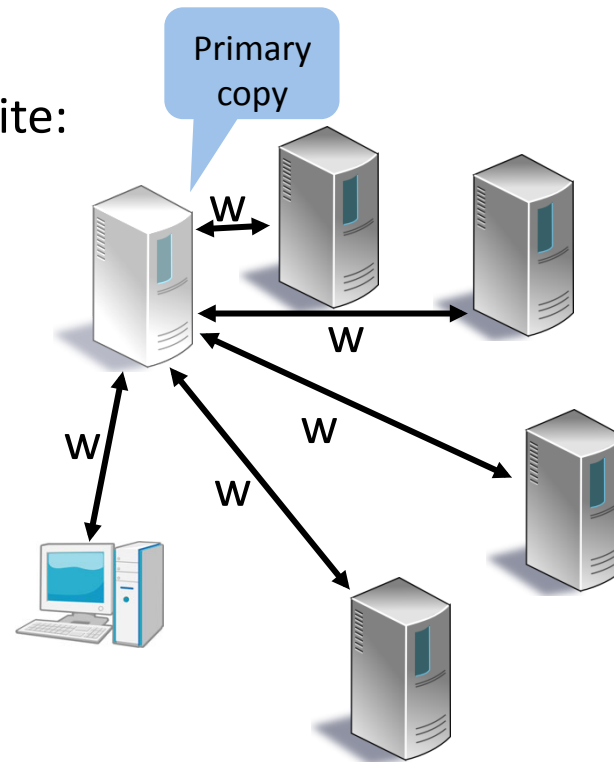
# Primary Copy

- Can we reduce the load on the clients?
- Yes! Write only to one server (the primary copy), and let primary copy distribute the update

Read:

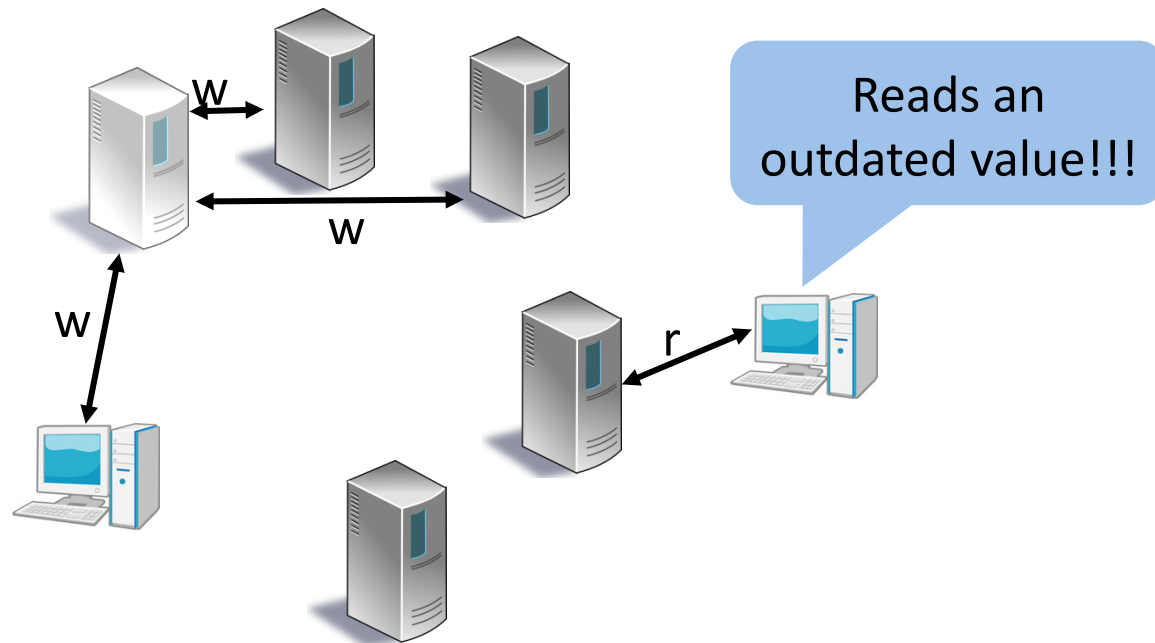


Write:



# Problem with Primary Copy

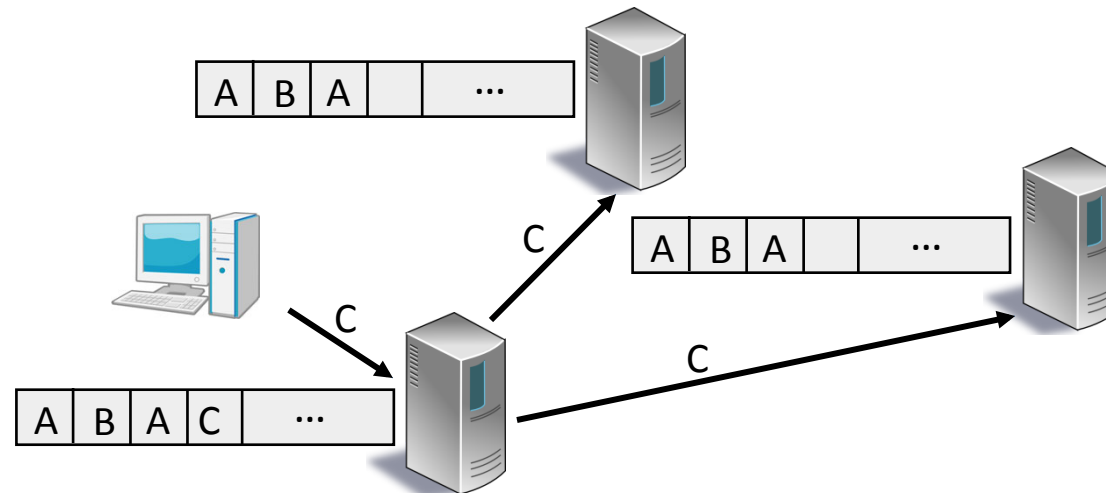
- If the clients can only send read requests to the primary copy, the system stalls if the primary copy fails
- However, if the clients can also send read requests to the other servers, the clients may not have a consistent view





# State Machine Replication?

- The state of each server has to be updated in the same way
- This ensures that all servers are in the same state whenever all updates have been carried out!



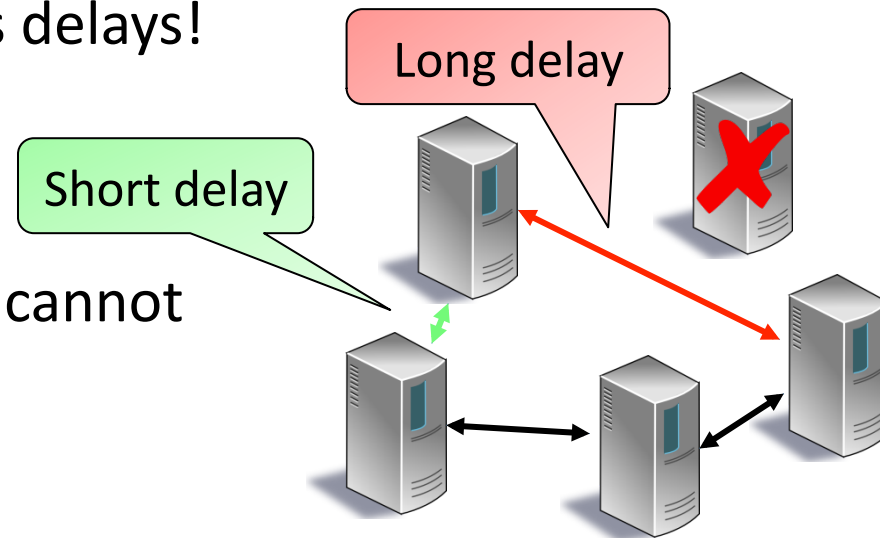
- The servers have to agree on each update
- But agreement in asynchronous systems with just one failure is impossible!

# From Theory to Practice

- Communication is often not synchronous, but not completely asynchronous either:
  - There may be reasonable bounds on the message delays
  - Practical systems often use message passing. The machines wait for the response from another machine and abort/retry after time-out
  - Failures: It depends on the application/system what kind of failures have to be handled...
- That is...
  - Real-world protocols also make assumptions about the system
  - These assumptions allow us to circumvent the lower bounds!

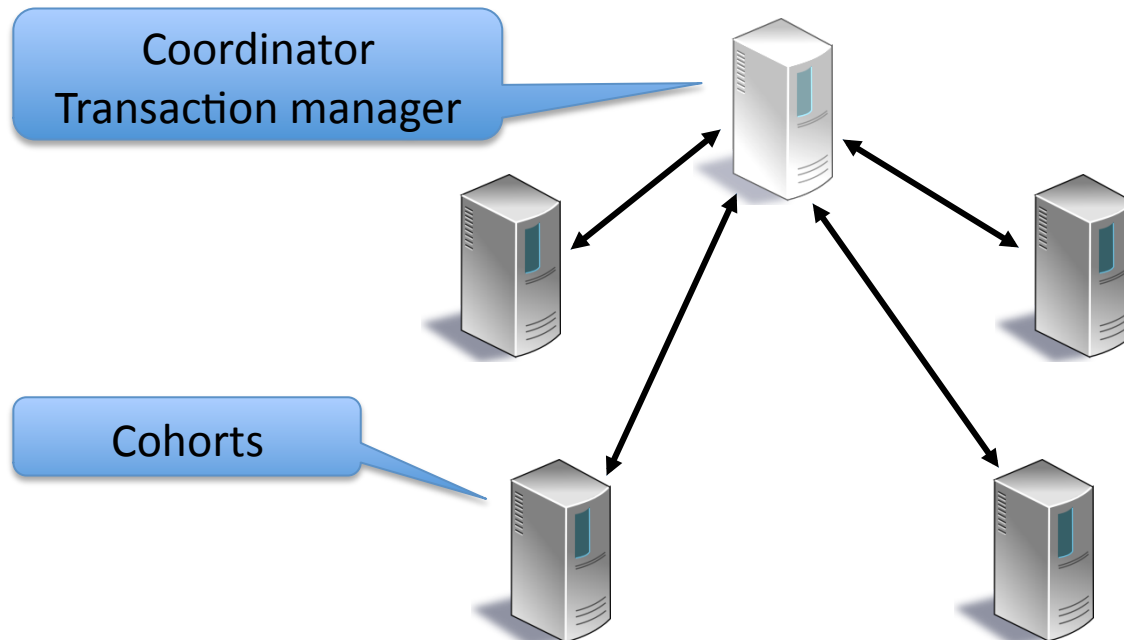
# Transactions

- In order to achieve consistency, updates have to be **atomic**
- A write has to be an atomic transaction
  - Updates are synchronized
- Either all nodes (servers) **commit** a transaction or all **abort**
- How do we handle transactions in asynchronous systems?
  - Unpredictable messages delays!
- Moreover, any node may fail...
  - Recall that this problem cannot be solved in theory!



# Two-Phase Commit

- Idea: Assume a coordinator
  - All other nodes communicate only with the coordinator
  - The coordinator communicates the final decision

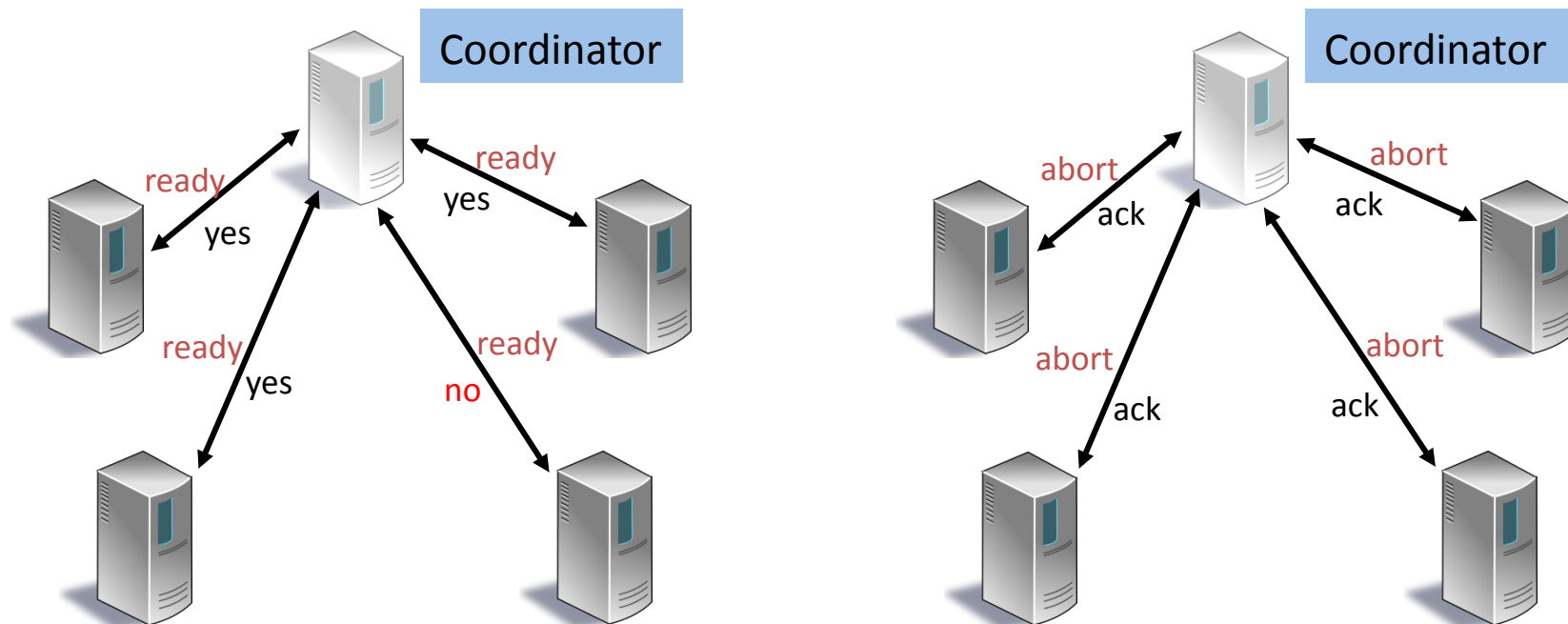


# Two-Phase Commit: Failures

- Fail-stop model: We assume that a failed node does not re-emerge
- In practice failed nodes may recover and we have to take action
- Failures are detected (instantly)
  - E.g. time-outs are used in practical systems to detect failures
- If the coordinator fails, a new coordinator takes over (instantly)
  - How can this be accomplished reliably?
- Interesting cases:  $\geq 1$  cohort failure, 1 TM failure, 1 TM failure +  $\geq 1$  cohort failure

# Two-Phase Commit: Protocol

- In the first phase, the coordinator asks if all nodes are ready to commit
- In the second phase, the coordinator sends the decision (commit/abort)
  - The coordinator aborts if at least one node said no



# What Happens in Practice

- First phase:
  - Cohorts take locks, prepare undo and redo logs, write updated state to persistent memory
- Second phase:
  - If cohorts receive commit, locks and reserved resources are released
  - When the coordinator receives ack she completes the transaction log and cleans up after herself

# Two-Phase Commit: Protocol

Phase 1:

Coordinator sends *ready* to all nodes

If a node receives *ready* from the coordinator:

If it is ready to commit

    Send *yes* to coordinator

else

    Send *no* to coordinator



# Two-Phase Commit: Protocol

Phase 2:

If the coordinator receives only *yes* messages:

    Send *commit* to all nodes

else

    Send *abort* to all nodes

If a node receives *commit* from the coordinator:

*Commit* and complete the transaction

else       (*abort* received)

*Abort* the transaction

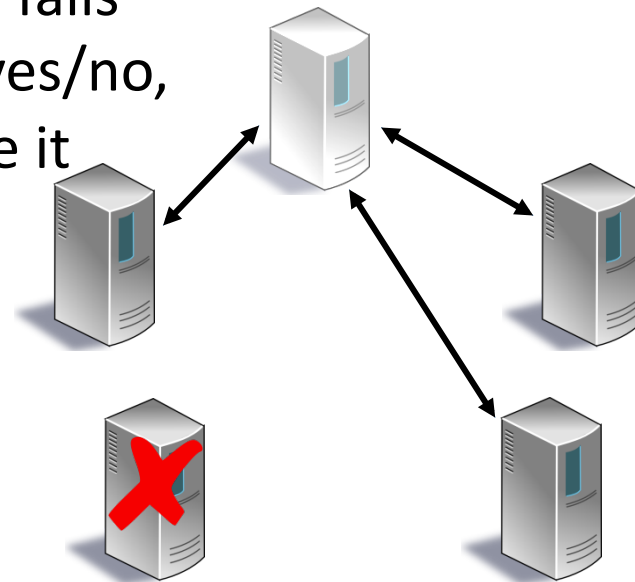
Send *ack* to coordinator

Once the coordinator received all *ack* messages:

It completes the transaction by *committing* or *aborting* itself

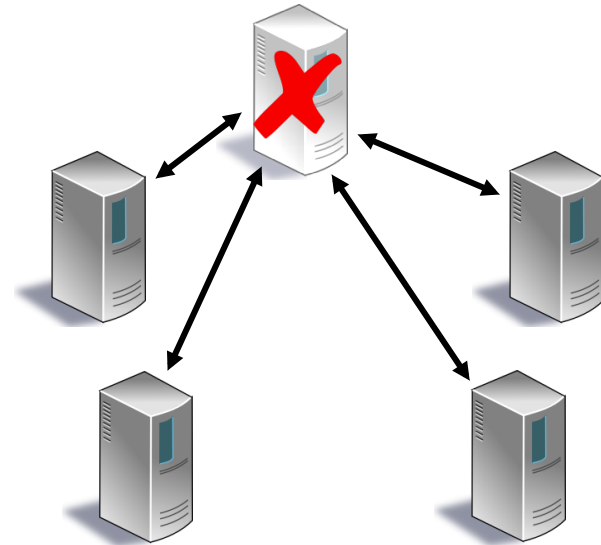
# Two-Phase Commit: Analysis

- 2PC obviously works if there are no failures
- 2PC also works if there is one failure
- *Weak termination* – termination is guaranteed only if there are no failures
- If a node that is not the coordinator fails
  - If the node fails before sending yes/no, the coordinator can either ignore it or safely abort the transaction
  - After sending commit or abort cohorts must acknowledge
  - Coordinator can recover by adding a failure transition to commit or abort state



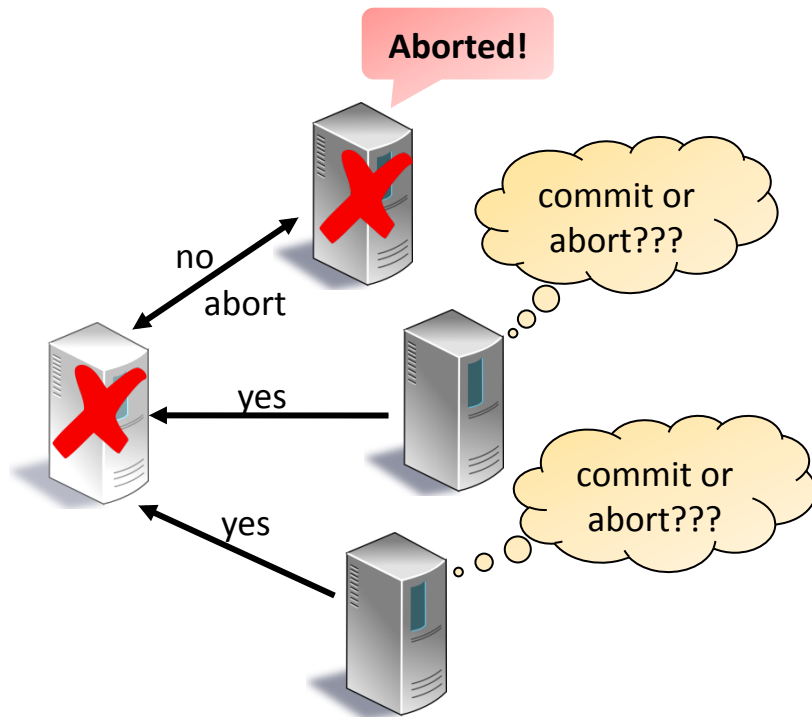
# What If the Coordinator Fails?

- This is (somehow) detected and a new coordinator takes over
- New coordinator:
  - Ask other nodes if a node has already received a commit
  - If a node has received commit:  
(All nodes answered “yes” and are ready)  
Reply “yes” to coordinator
  - Otherwise reply “no” and refuse to accept a commit that may arrive from the old coordinator
  - If some node replied “yes”, the new coordinator broadcasts commit
  - Otherwise can complete or abort
- This works if there is only one failure
- Does 2PC still work with multiple failures...?

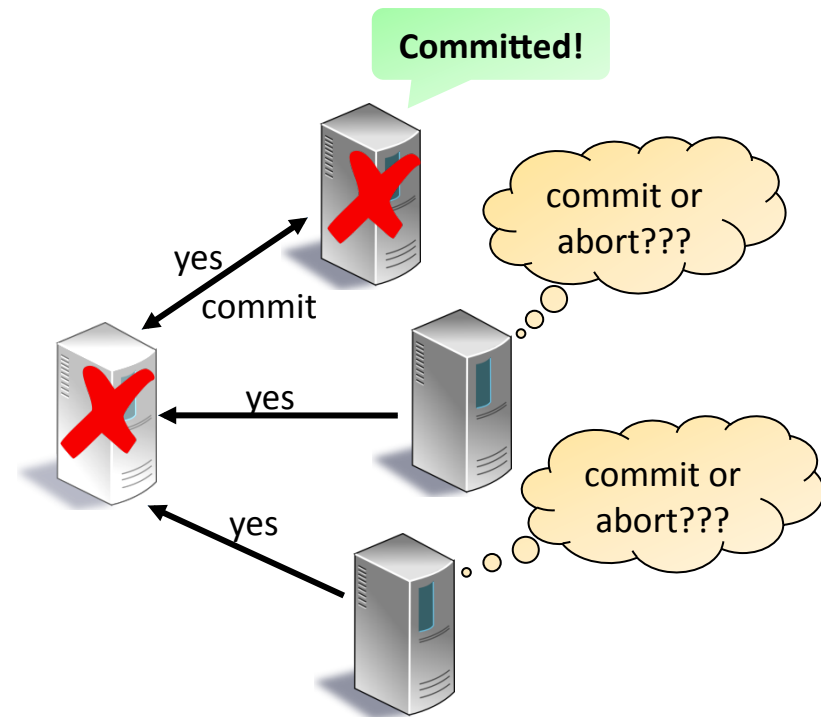


# Two-Phase Commit: Multiple Failures

- As long as the coordinator is alive, multiple failures are no problem
  - The same arguments as for one failure apply
- What if the coordinator and another node crashes?



The nodes cannot commit!



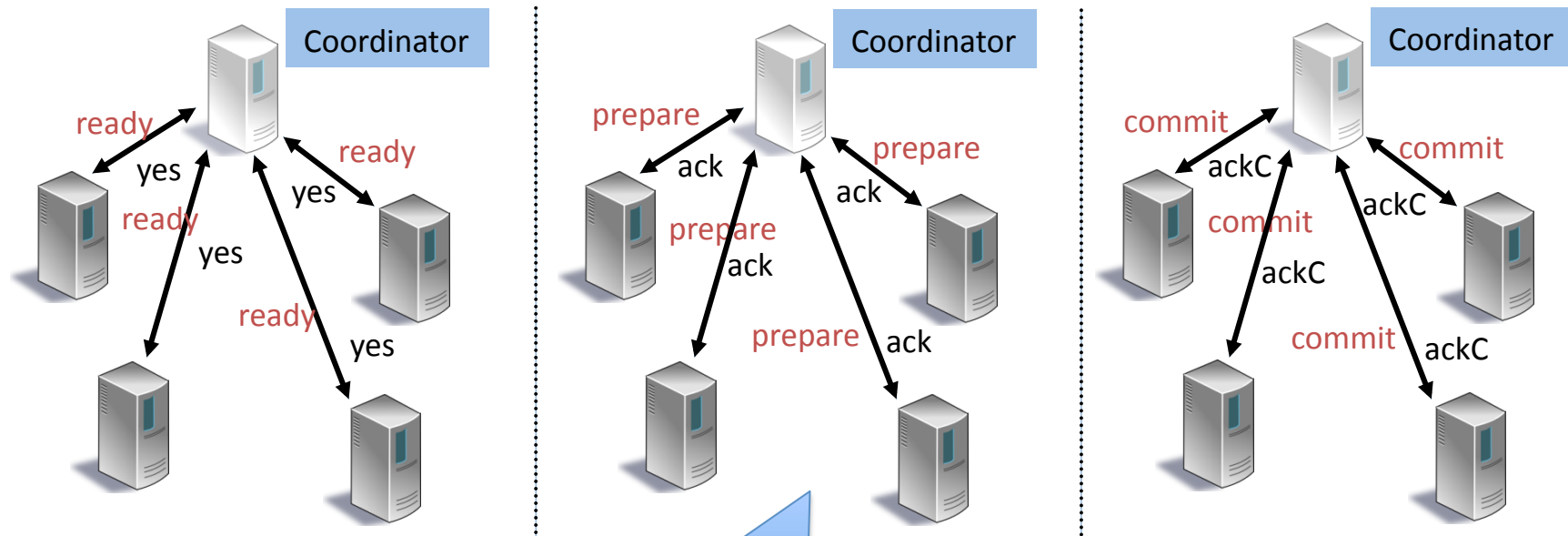
The nodes cannot abort!

# Three-Phase Commit

- What to do in case of multiple failures incl. coordinator?
- Add another phase to the protocol!
  - The new phase precedes the commit phase
  - The goal is to inform all nodes that all are ready to commit (or not)
  - At the end of this phase, every node knows whether or not all nodes want to commit *before* any node has actually committed or aborted!
- This protocol is called the three-phase commit (3PC) protocol

# Three-Phase Commit: Protocol

- In the new (second) phase, the coordinator sends prepare (to commit) messages to all nodes



When cohorts have received "prepare" they know that all responded ready in phase 1

# Three-Phase Commit: Protocol

Phase 1:

Coordinator sends *ready* to all nodes

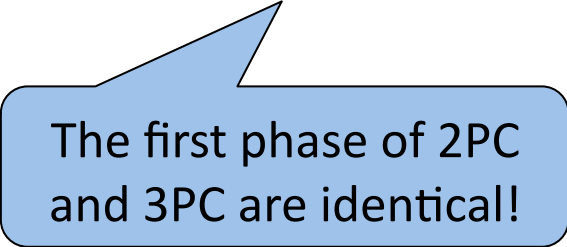
If a node receives *ready* from the coordinator:

If it is ready to commit

    Send *yes* to coordinator

else

    Send *no* to coordinator



The first phase of 2PC  
and 3PC are identical!

# Three-Phase Commit: Protocol

Phase 2:

If the coordinator receives only *yes* messages:

    Send *prepare* to all nodes

else

    Send *abort* to all nodes

If a node receives *prepare* from the coordinator:

    Prepare to commit the transaction

else           (*abort* received)

**Abort** the transaction

Send *ack* to coordinator



This is the new phase



# Three-Phase Commit: Protocol

Phase 3:

Once the coordinator received all *ack* messages:

If the coordinator sent *abort* in Phase 2

    The coordinator **aborts** the transaction  
    as well

else       (it sent *prepare*)

    Send *commit* to all nodes

If a node receives *commit* from the coordinator:

**Commit** the transaction

    Send *ackCommit* to coordinator

Once the coordinator received all *ackCommit*  
messages:

    It completes the transaction by **committing**  
    itself

# Three-Phase Commit: Analysis

- 3PC is correct + non-blocking
- Case 1: Coordinator is correct
  - Coordinator gets all correct nodes to either all commit or all abort
  - Termination: If some node fails before sending *yes/no*, coordinator can abort. If some node fails after the coordinator sent *prepare*, the coordinator can still enforce a commit because all nodes must have sent *yes*
- Case 2: Coordinator fails
  - No-one else fails: New coordinator can restart the protocol
- Coordinator and cohort fails and some node committed.
  - Coordinator must have received *ack* messages from all nodes
  - All nodes must have received a *prepare* message. The new coordinator can enforce a commit.
- Coordinator and cohort fails and some node aborted
  - No node can have received a *prepare* message. Thus, the new coordinator can safely abort the transaction

# Three-Phase Commit: Analysis

- + 3PC is correct and non-blocking
- + 3PC can handle more than one random crash failure
- 3PC depends on a single coordinator
  - Fundamentally non-scalable
- Relies on instantaneous failure detector – not very realistic
  - What if nodes disagree about who failed?
  - Such as: The coordinator
- What are the implications of transient failures?
- Still, 3PC and 2PC are used successfully in practice
- However, it would be nice to have a practical protocol that does not depend on a single coordinator
  - and that can handle temporary failures!

# Paxos

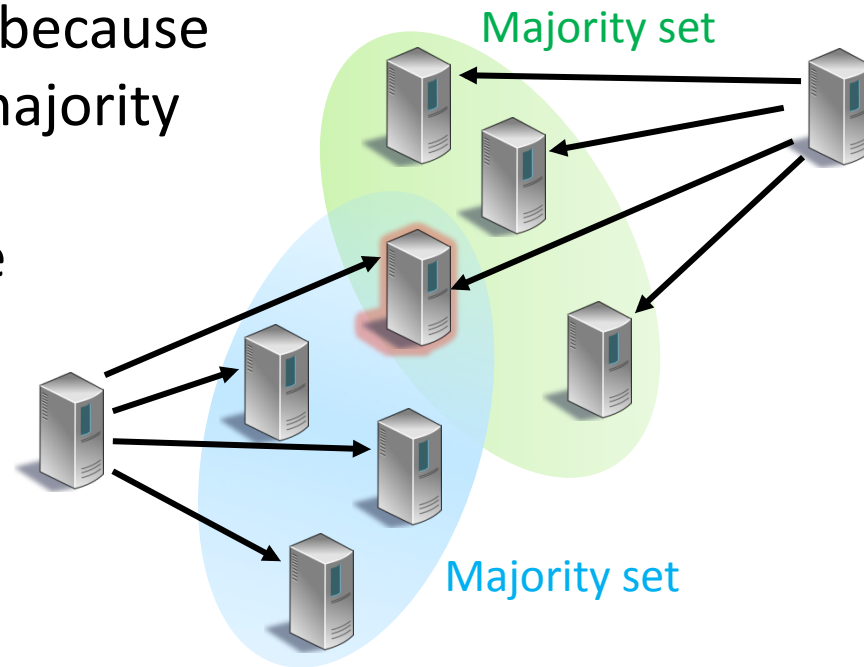
- Historical note
  - In the 1980s, a fault-tolerant distributed file system called “Echo” was built
  - According to the developers, it achieves “consensus” despite any number of failures as long as a majority of nodes is alive
  - The steps of the algorithm are simple if there are no failures and quite complicated if there are failures
  - Leslie Lamport thought that it is impossible to provide guarantees in this model and tried to prove it
  - Instead of finding a proof, he found a much simpler algorithm that works: The Paxos algorithm
- Paxos is an algorithm that does not rely on a coordinator
  - Communication is still asynchronous
  - All nodes may crash at any time and they may also recover

L. Lamport: The Part-Time Parliament, ACM ToCS 16, 2, 1998  
L. Lamport: Paxos Made Simple, ACM SIGACT News 32, 4, 2001

fail-recover model

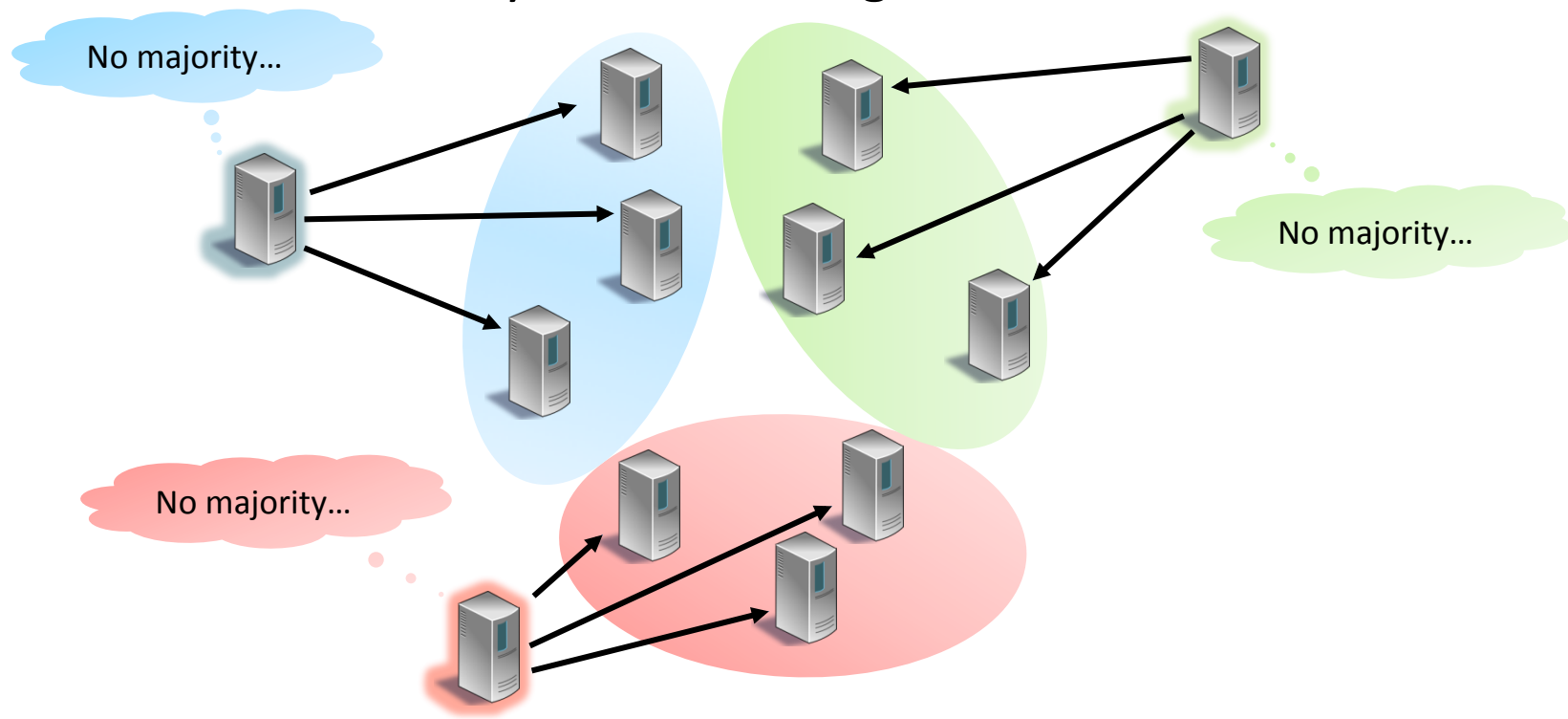
# Paxos: Majority Sets

- Paxos is a two-phase protocol, but more resilient than 2PC
- Why is it more resilient?
  - There is no coordinator. A majority of the nodes is asked if a certain value can be accepted
  - A majority set is enough because the intersection of two majority sets is not empty
  - If a majority chooses one value, no majority can choose another value!



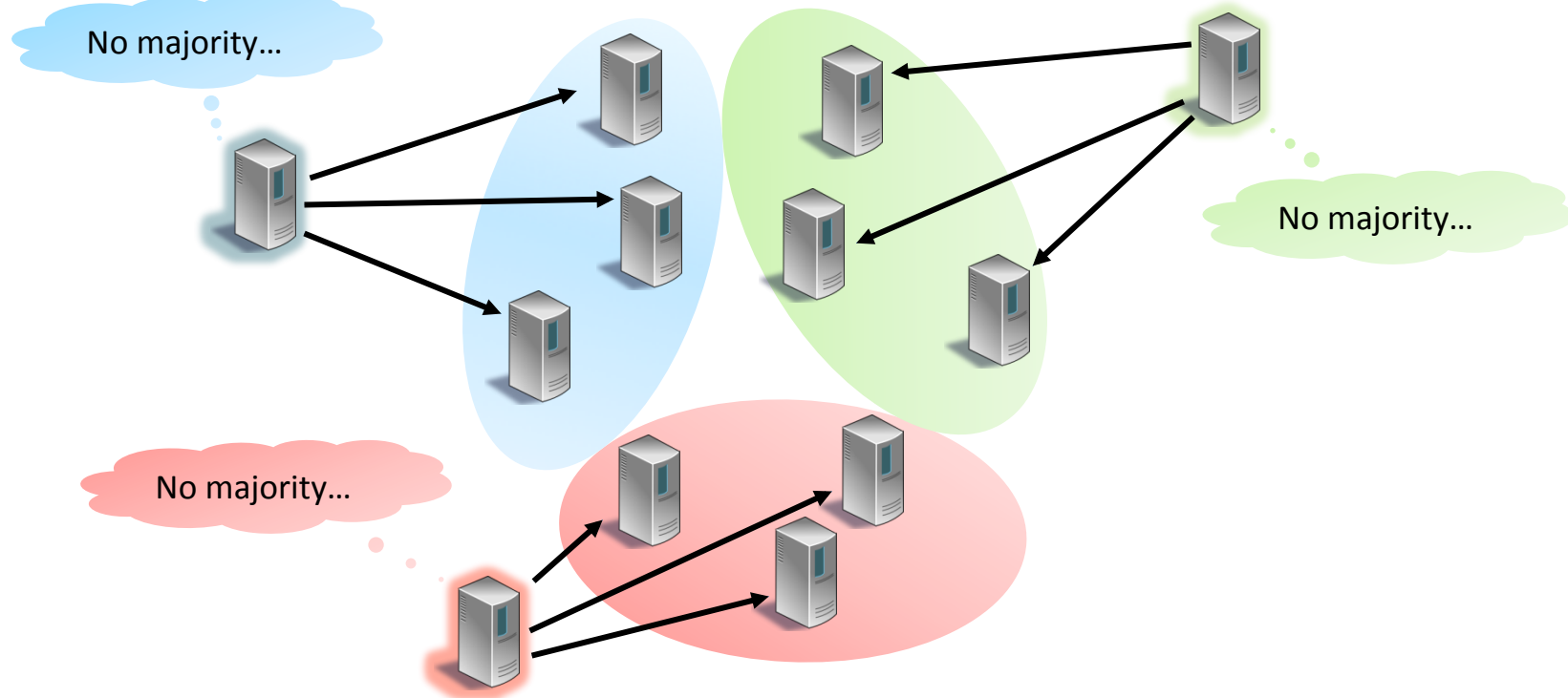
# Paxos: Majority Sets

- Majority sets are a good idea
- But, what happens if several nodes compete for a majority?
  - Conflicts have to be resolved
  - Some nodes may have to change their decision



# Paxos: Majority Sets

- Majority sets are a good idea
- But, what happens if several nodes compete for a majority?
  - Conflicts have to be resolved
  - Some nodes may have to change their decision



# Paxos: Roles

- Each node has one or more roles:
- **Proposer**
  - A proposer is a node that proposes a certain value for acceptance
  - There can be any number of proposers at the same time
- **Acceptor**
  - An acceptor is a node that receives a proposal from a proposer
  - An acceptor can either accept or reject a proposal
- **Learner**
  - A learner is a node that is not involved in the decision process
  - The learners must learn the final result from the proposers/acceptors



# Paxos: Proposal

- Proposal  $(x,n)$ :  $x$  is proposed value,  $n$  is proposal number
- New proposal uses a larger (unique) proposal number
- An acceptor *accepts* a proposal  $(x,n)$  if  $n$  is larger than any proposal number it has ever heard
- An acceptor can *accept* any number of proposals
  - An accepted proposal may not necessarily be *chosen*
  - The value of a *chosen proposal* is the *chosen value*
- An acceptor can even *choose* any number of proposals
  - However, if two proposals  $(x,n)$  and  $(y,m)$  are chosen, then  $x = y$



Consensus

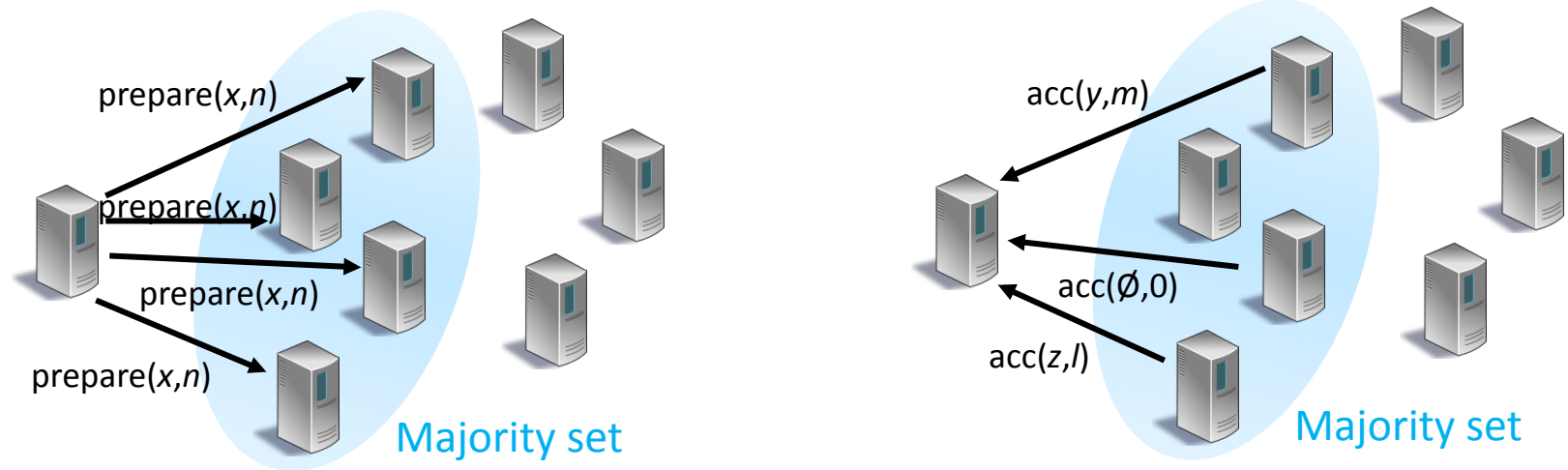
# Paxos: Prepare Phase

- Before a node sends  $\text{propose}(x,n)$ , it sends  $\text{prepare}(x,n)$ 
  - This message is used to indicate that the node wants to propose  $(x,n)$
- If  $n$  is larger than all received request numbers, an acceptor returns the *accepted* proposal  $(y,m)$  with the largest request number  $m$ 
  - If it never accepted a proposal, the acceptor returns  $(\emptyset,0)$
  - The proposer learns about accepted proposals!



$m < n$

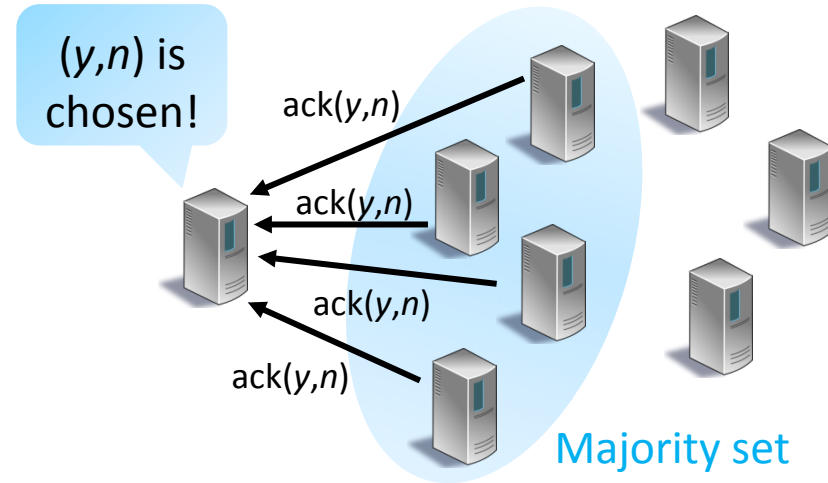
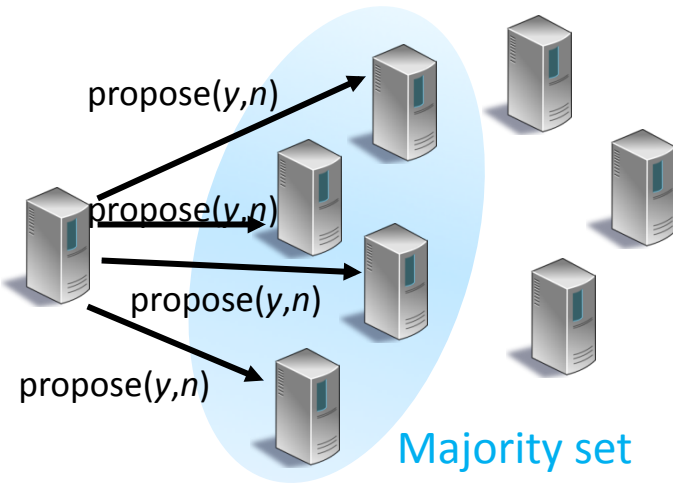
# Paxos: Prepare Phase



# Paxos: Propose Phase

- If the proposer receives replies from a majority, it sends a proposal
- Proposer proposes own value if only reply is  $\text{acc}(\emptyset, 0)$
- Otherwise the proposer adopts the value  $y$  in the proposal with the largest request number  $m$ 
  - The proposal still contains its sequence number  $n$ , i.e.,  $(y, n)$  is proposed
- If the proposer receives acknowledgements  $\text{ack}(y, n)$  from a majority the proposal is *chosen*

# Paxos: Propose Phase



# Paxos: Algorithm of Proposer

Proposer wants to propose  $(x, n)$ :

Send  $\text{prepare}(x, n)$  to a majority of the nodes  
if a majority of the nodes replies then

    let  $(y, m)$  be the received proposal with the  
        largest request number

    if  $m = 0$  then (No acceptor ever accepted  
        another proposal)

        Send  $\text{propose}(x, n)$  to the same set of  
            acceptors

    else

        Send  $\text{propose}(y, n)$  to the same set of acceptors

if a majority of the nodes replies with  $\text{ack}(x, n)$   
respectively  $\text{ack}(y, n)$

The proposal is chosen!

After a time-out, the proposer gives up and may send a new proposal

# Paxos: Algorithm of Acceptor

Why persistently?

Initialize and store persistently:

$n_{\max} := 0$

Largest request number ever received

$(x_{\text{last}}, n_{\text{last}}) := (\emptyset, 0)$

Last accepted proposal

Acceptor receives  $\text{prepare}(x, n)$ :

if  $n > n_{\max}$  then

$n_{\max} := n$

Send  $\text{acc}(x_{\text{last}}, n_{\text{last}})$  to the proposer

Acceptor receives  $\text{propose}(x, n)$ :

if  $n = n_{\max}$  then

$x_{\text{last}} := x$

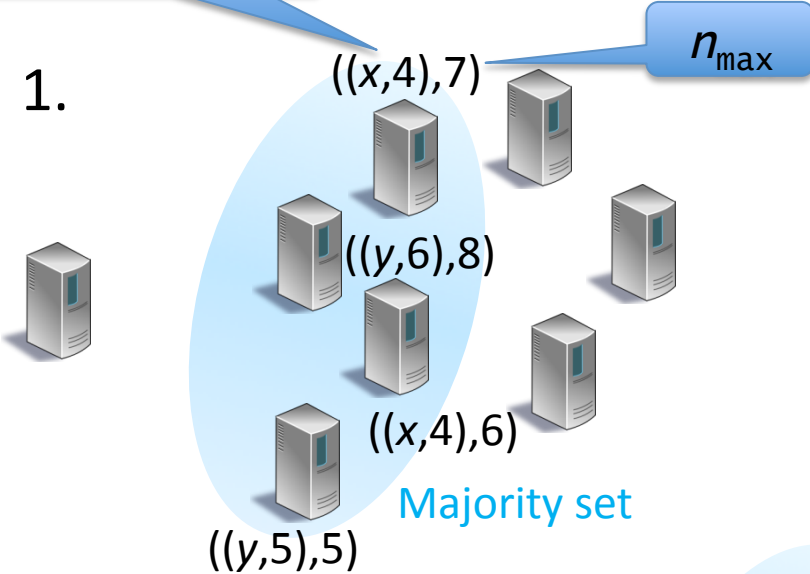
$n_{\text{last}} := n$

Send  $\text{ack}(x, n)$  to the proposer

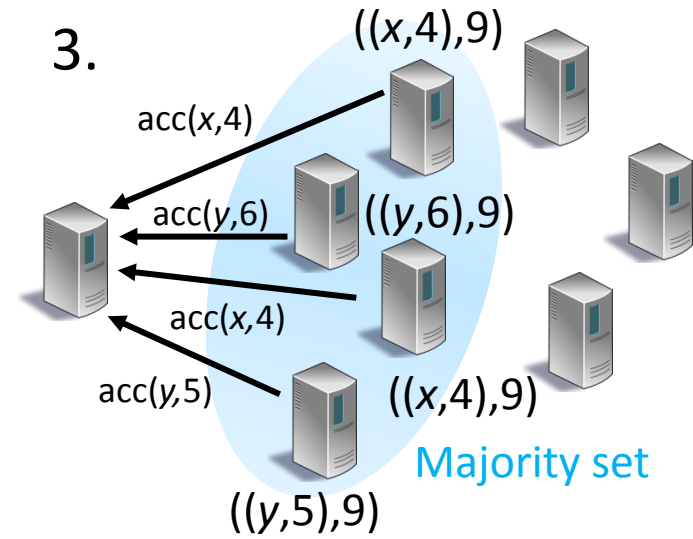
# Paxos: Prepare Phase

$(x_{last}, n_{last})$

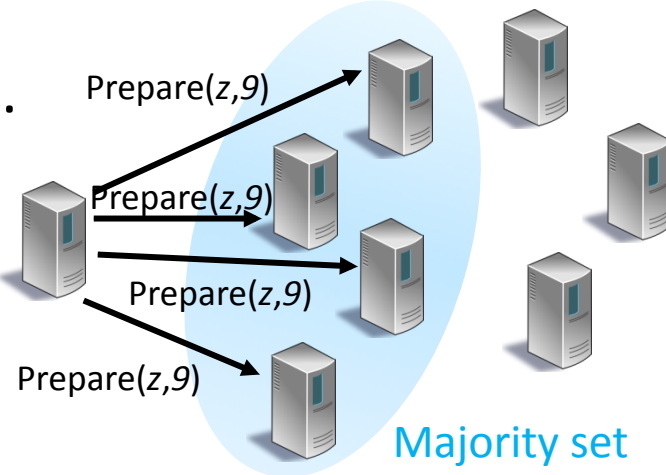
1.



3.

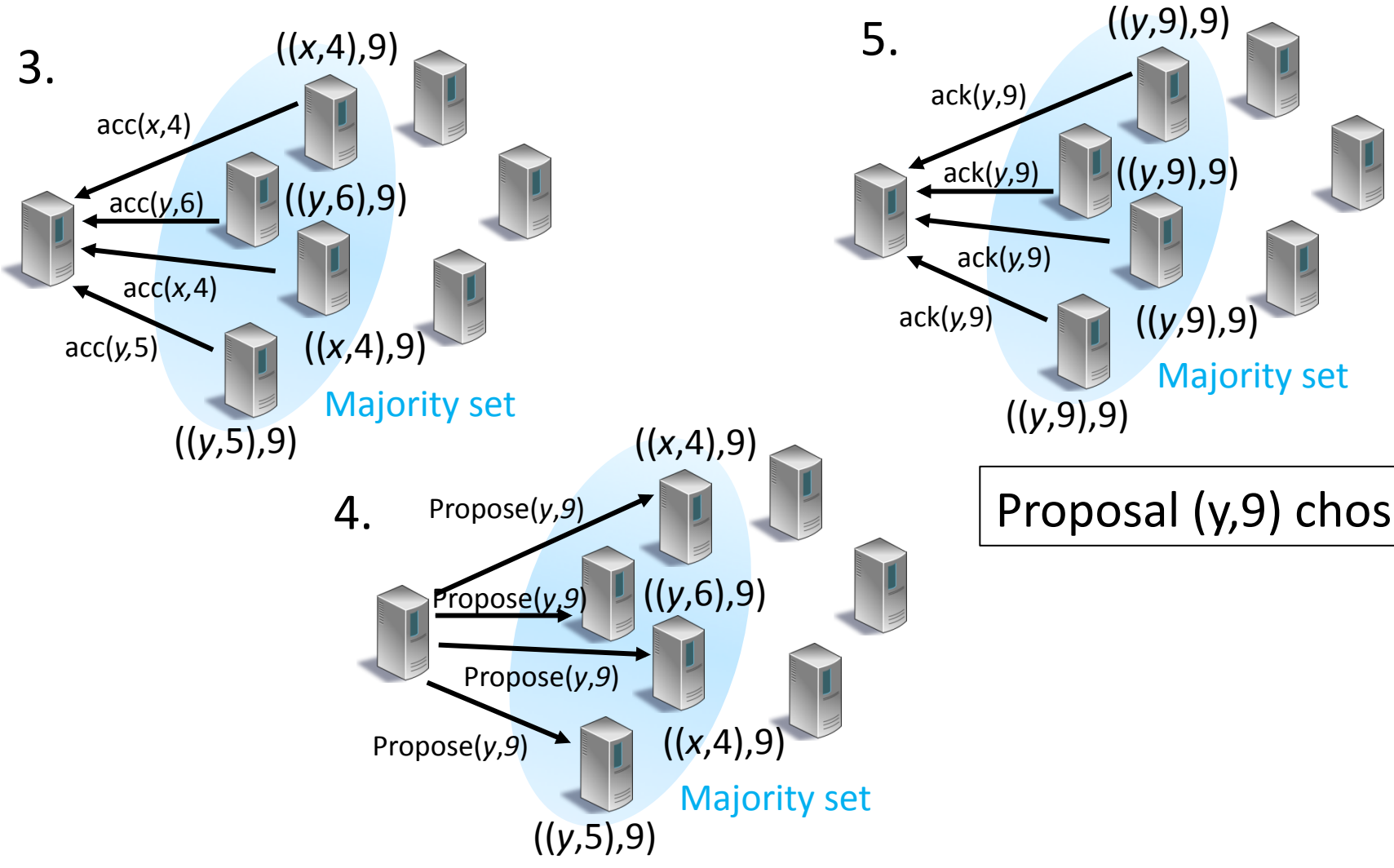


2.



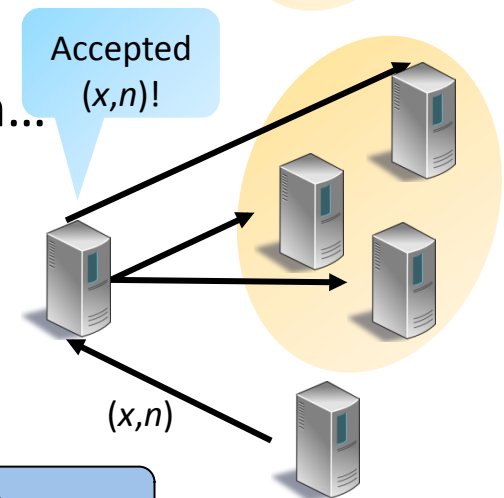
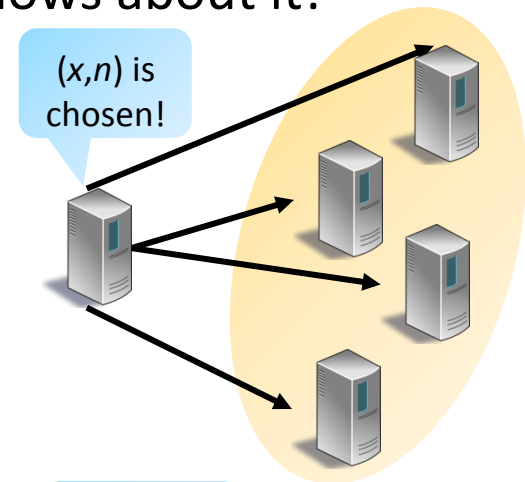


# Paxos: Propose Phase



# Paxos: Spreading the Decision

- After a proposal is chosen, only the proposer knows about it!
- How do the others (learners) get informed?
- The proposer could inform all learners directly
  - Only  $n-1$  messages are required
  - If the proposer fails, the learners are not informed (directly)...
- The acceptors could broadcast every time they accept a proposal
  - Much more fault-tolerant
  - Many accepted proposals may not be chosen...
  - Moreover, choosing a value costs  $O(n^2)$  messages without failures!
- Something in the middle?
  - The proposer informs  $b$  nodes and lets them broadcast the decision



Trade-off: fault-tolerance vs. message complexity

# Paxos: Agreement

**Lemma** If a proposal  $(x,n)$  is *chosen*, then for every issued proposal  $(y,n')$  for which  $n' > n$  it holds that  $x = y$

**Proof:**

- Assume that there are proposals  $(y,n')$  for which  $n' > n$  and  $x \neq y$ . Consider the proposal with the smallest proposal number  $n'$
- Consider the non-empty intersection  $S$  of the two sets of nodes that function as the acceptors for the two proposals
- Proposal  $(x,n)$  has been accepted  $\rightarrow$  Since  $n' > n$ , the nodes in  $S$  must have received  $\text{prepare}(y,n')$  after  $(x,n)$  has been accepted
- This implies that the proposer of  $(y,n')$  would also propose the value  $x$  unless another acceptor has accepted a proposal  $(z,n^*)$ ,  $z \neq x$  and  $n < n^* < n'$ . However, this means that some node must have proposed  $(z,n^*)$ , a contradiction because  $n^* < n'$  and we said that  $n'$  is the smallest proposal number!

# Paxos: Consensus

**Theorem** If  $x$  is chosen, all nodes choose  $x$

**Proof:**

- Once a proposal  $(x, n)$  is chosen, each proposal  $(y, n')$  that is sent afterwards has the same proposal value, i.e.,  $x = y$  according to the lemma on the previous slide
- Since every subsequent proposal has the same value  $x$ , every proposal that is accepted after  $(x, n)$  has been chosen has the same value  $x$
- Since no other value than  $x$  is accepted, no other value can be chosen!

# Paxos: Wait a Minute...

- Paxos is great!
- It is a simple, **deterministic** algorithm that works in **asynchronous** systems and tolerates  $f < n/2$  failures
- Is this really possible...?



Theorem

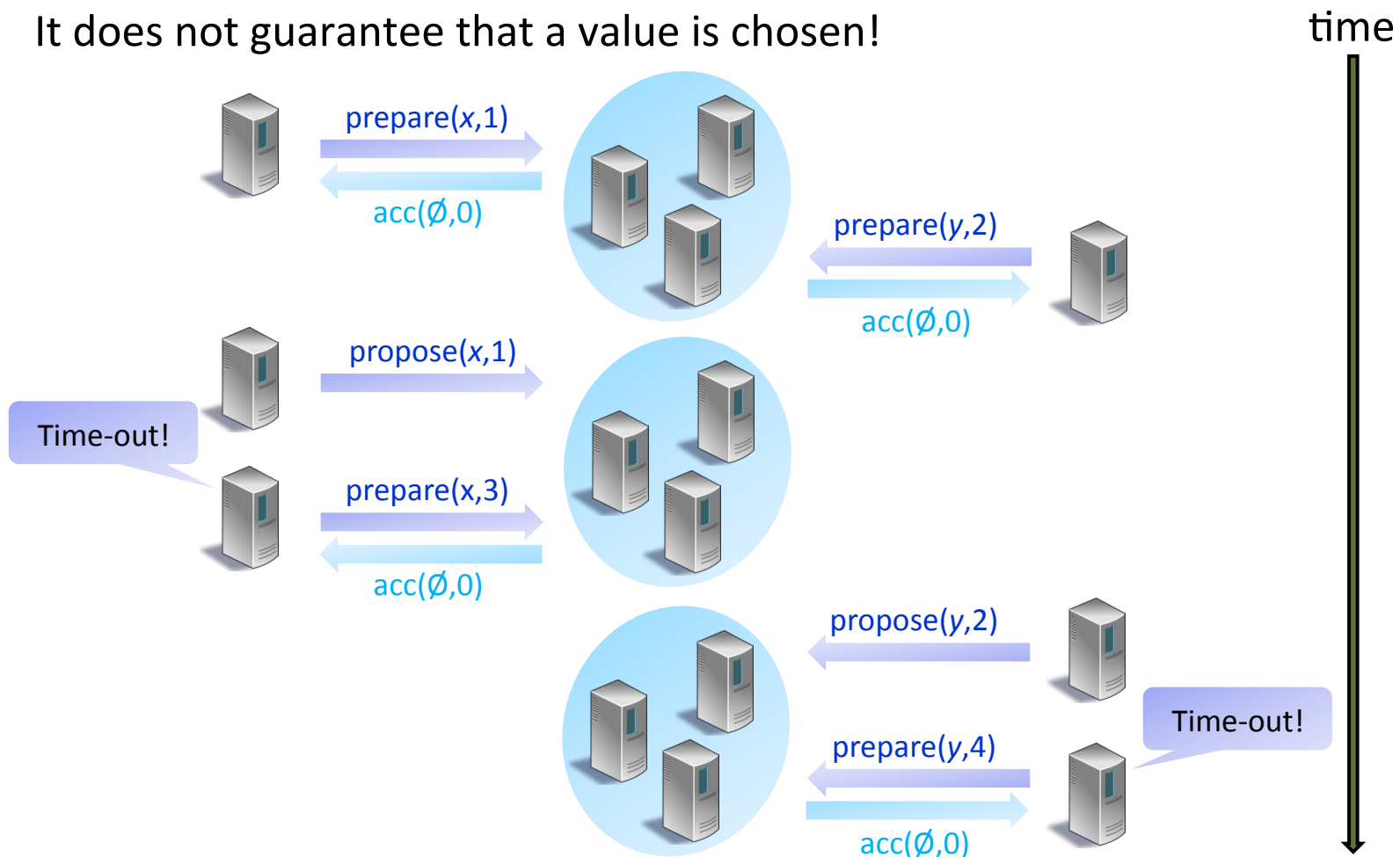
A deterministic algorithm cannot guarantee consensus in asynchronous systems even if there is just one faulty node



- Does Paxos contradict this lower bound...?

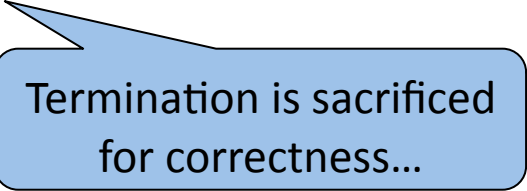
# Paxos: No Liveness Guarantee

- The answer is no! Paxos only guarantees that if a value is chosen, the other nodes can only choose the same value
- It does not guarantee that a value is chosen!



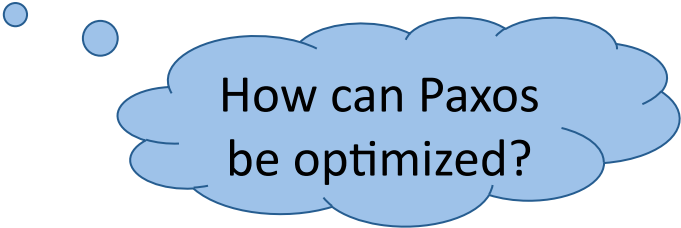
# Paxos: Agreement vs. Termination

- In asynchronous systems, a deterministic consensus algorithm cannot have both, guaranteed **termination** and **correctness**
- Paxos is always correct. Consequently, it cannot guarantee that the protocol terminates in a certain number of rounds



Termination is sacrificed  
for correctness...

- Although Paxos may not terminate in theory, it is quite efficient in practice using a few optimizations .



How can Paxos  
be optimized?

# Paxos in Practice

- There are ways to optimize Paxos by dealing with some practical issues
  - For example, the nodes may wait for a long time until they decide to try to submit a new proposal
  - A simple solution: The acceptors send NAK if they do not accept a prepare message or a proposal. A node can then abort immediately
  - Note that this optimization increases the message complexity...
- Paxos is indeed used in practical systems!
  - Yahoo!'s *ZooKeeper*: A management service for large distributed systems uses a variation of Paxos to achieve consensus
  - Google's *Chubby*: A distributed lock service library. Chubby stores lock information in a replicated database to achieve high availability. The database is implemented on top of a fault-tolerant log layer based on Paxos



# Paxos: Fun Facts

- Why is the algorithm called Paxos?
- Leslie Lamport described the algorithm as the solution to a problem of the parliament on a fictitious Greek island called Paxos
- Many readers were so distracted by the description of the activities of the legislators, they did not understand the meaning and purpose of the algorithm. The paper was rejected
- Leslie Lamport refused to rewrite the paper. He later wrote that he *“was quite annoyed at how humorless everyone working in the field seemed to be”*
- After a few years, some people started to understand the importance of the algorithm
- After eight years, Leslie Lamport submitted the paper again, basically unaltered. It got accepted!



# Chubby

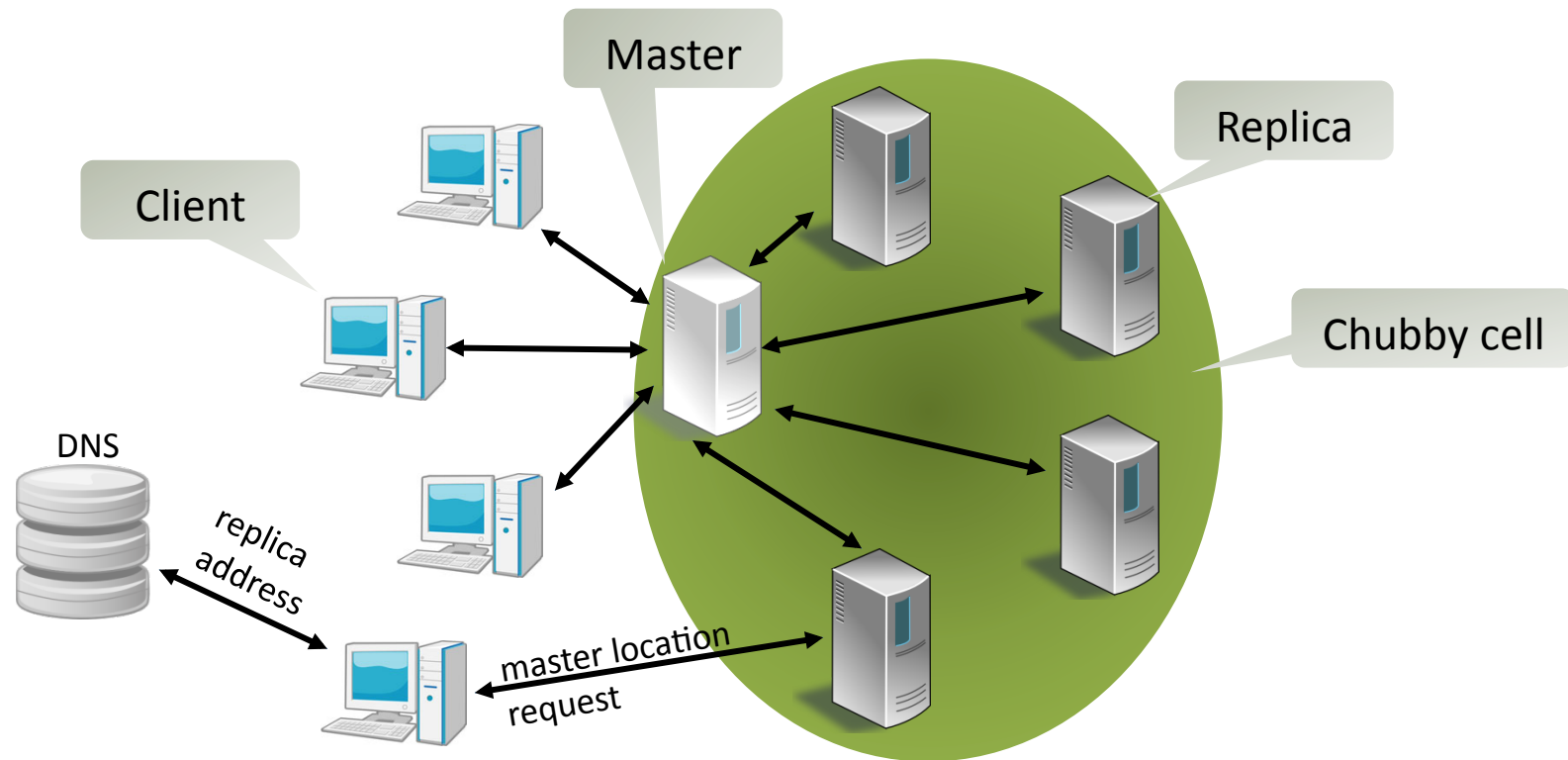


- Chubby is a coarse-grained distributed lock service
  - Coarse-grained: Locks are held for hours or even days
- Chubby allows clients to synchronize activities
  - E.g., synchronize access through a leader in a distributed system
  - The leader is elected using Chubby: The node that gets the lock for this service becomes the leader!
- Design goals are high availability and reliability
  - High performance is not a major issue
- Chubby is used in many tools, services etc. at Google
  - Google File System (GFS)
  - BigTable (distributed database)

# Chubby: System Structure

- A Chubby cell typically consists of 5 servers
  - One server is the master, the others are replicas
  - The clients only communicate with the master
  - Clients find the master by sending master location requests to some replicas listed in the DNS

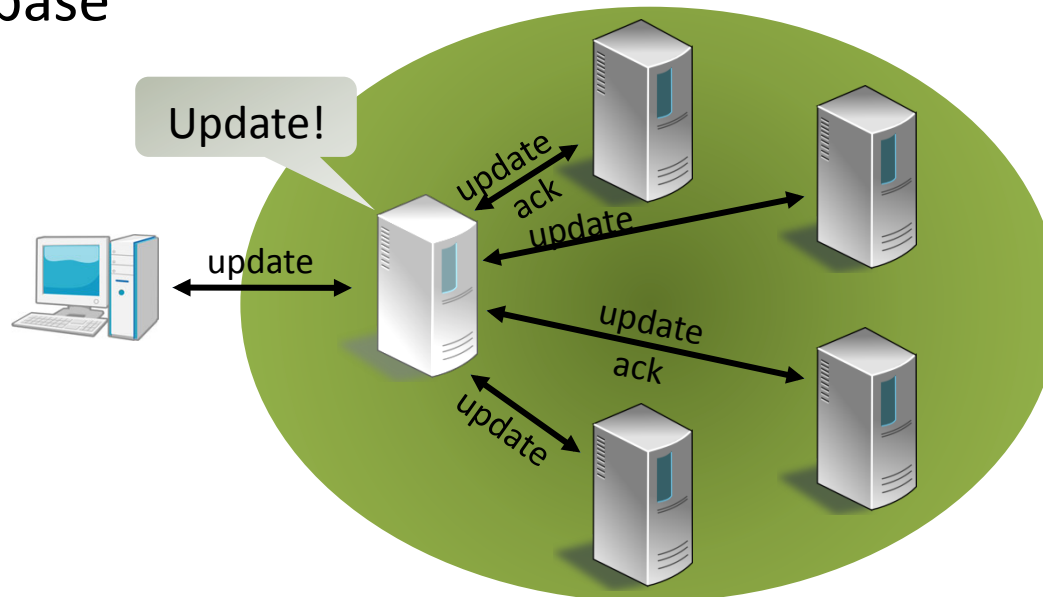
# Chubby: System Structure



- Replicas located to reduce likelihood of correlated failure
- Often close, but can be far apart

# Chubby: System Structure

- The master handles all read accesses
- The master also handles writes
  - Copies of the updates are sent to the replicas
  - Majority of replicas must acknowledge receipt of update before master writes its own value and updates the official database



# Chubby: Master Election

- The master remains the master for the duration of the master lease
  - Before the lease expires, the master can renew it (and remain the master)
  - No new master is elected before the lease expires
  - A new master is elected as soon as the lease expires
  - Uses a leader election protocol
  - This ensures that the system does not freeze (for a long time) if the master crashed
- How do the servers in the Chubby cell agree on a master?
- How are the replicated servers kept consistent?
- They run (a variant of) the Paxos algorithm!

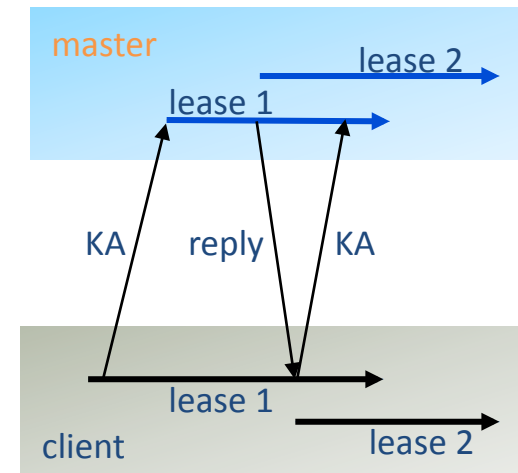
# Chubby: Locks

- Locks are advisory (not mandatory)
  - As we are used to from shared memory (by now)
  - However, data can be *read* without the lock!
  - Why? Most accesses are reads!
  - Mandatory locks: If the lock holder crashes, then all reads are stalled until the situation is resolved



# Chubby: Sessions

- Client initially contacts master to establish a session
  - Session: Relationship between Chubby cell and Chubby client
- Each session has an associated lease
  - Master can extend the lease
  - Master may not revoke the lease
  - Longer lease times if the load is high
- Periodic KeepAlive (KA) handshake to maintain lease
  - The master does not respond until the client's previous lease is close to expiring
  - Then it responds with the duration of the new lease
  - The client reacts immediately and issues the next KA
- Ending a session
  - Either the client terminates session explicitly
  - or the lease expires



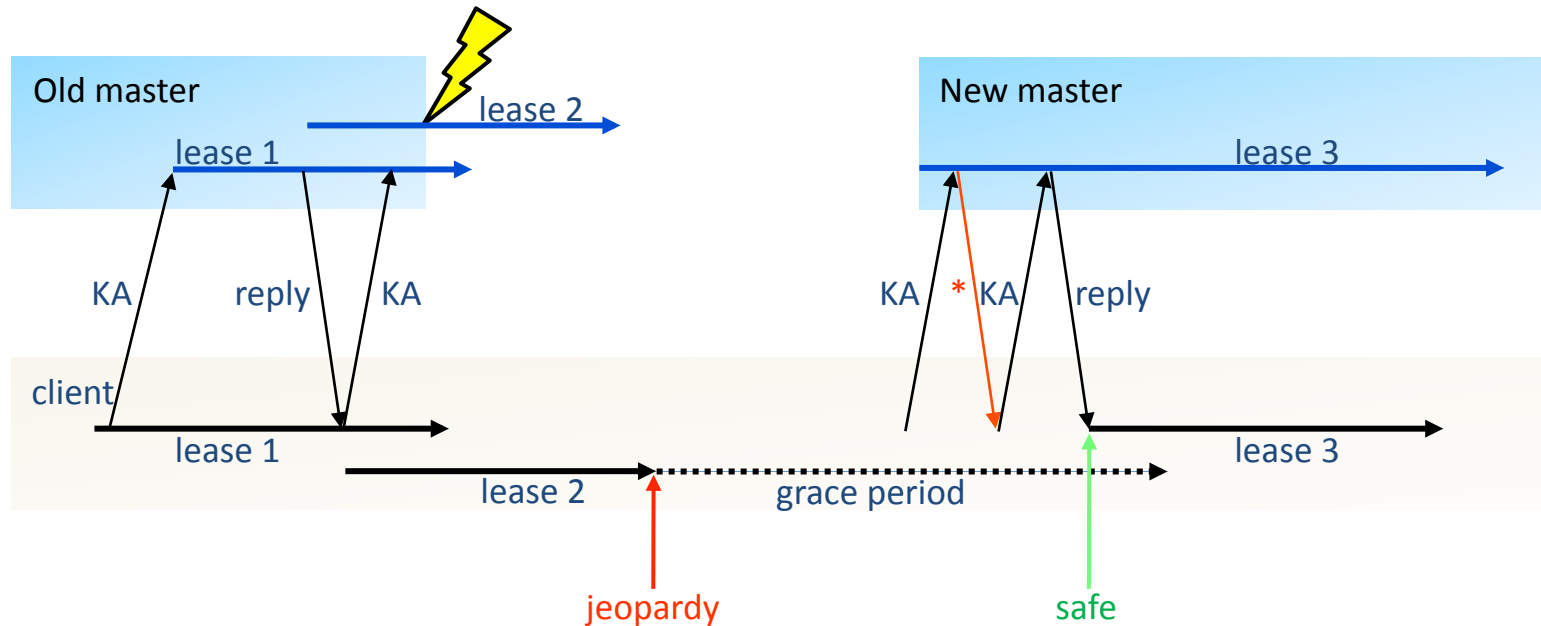


# Chubby: Lease Timeout

- The client maintains a local lease timeout
  - The client knows (roughly) when it has to hear from the master again
- If the local lease expires, the session is in jeopardy
- As soon as a session is in jeopardy, the grace period (45s by default) starts
  - If there is a successful KeepAlive exchange before the end of the grace period, the session is saved!
  - Otherwise, the session expired
- This might happen if the master crashed...

# Chubby: Master Failure

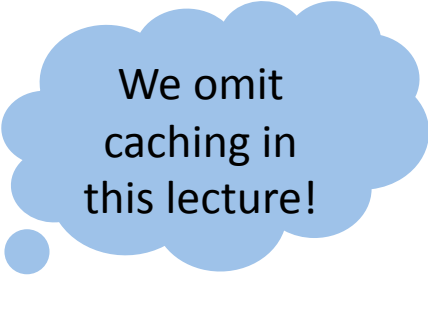
- The grace period can save sessions



- The client finds the new master using a master location request
- Its first KA to the new master is denied (\*) because the new master has a new epoch number (sometimes called view number)
- The next KA succeeds with the new number

# Chubby: Master Failure

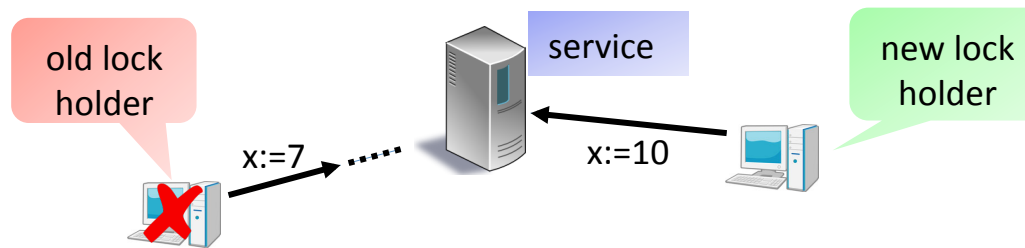
- A master failure is detected once the master lease expires
- A new master is elected, which tries to resume exactly where the old master left off
  - Read data that the former master wrote to disk (this data is also replicated)
  - Obtain state from clients
- Actions of the new master
  1. It picks a new epoch number
    - It only replies to master location requests
  2. It rebuilds the data structures of the old master
    - Now it also accepts KeepAlives
  3. It informs all clients about failure → Clients flush cache
    - All operations can proceed



We omit caching in this lecture!

# Chubby: Locks and Crashes

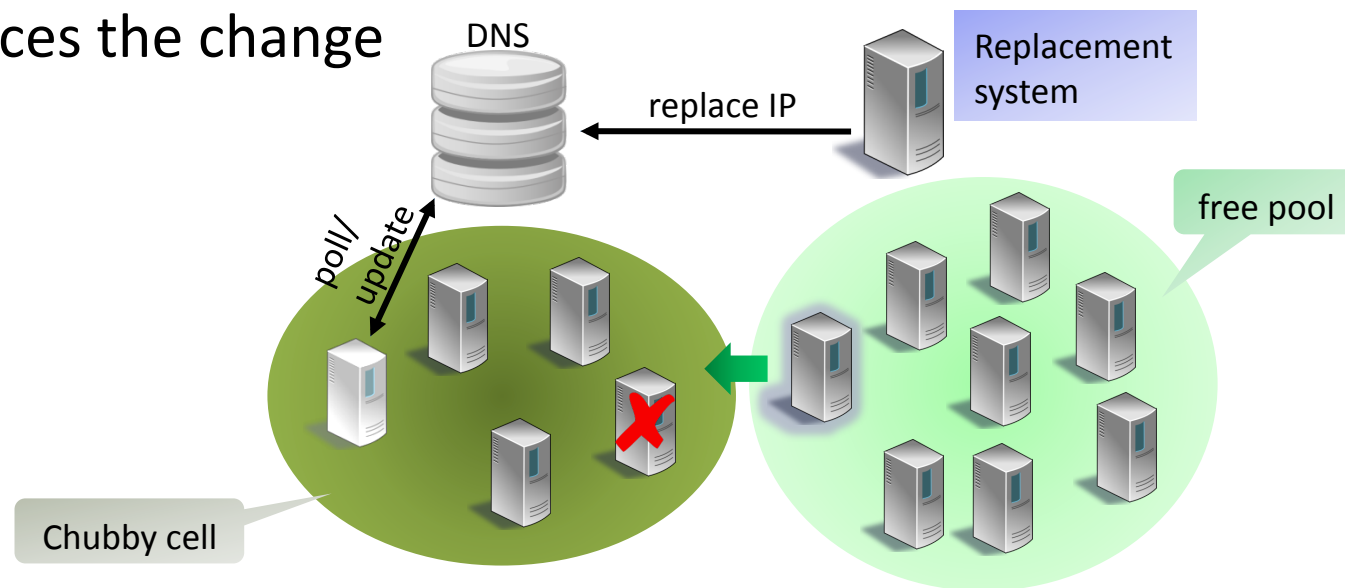
- What if a lock holder crashes and its (write) request is still in transit?
  - This write may undo an operation of the next lock holder!



- Heuristic I: Sequencer
  - Add a sequencer (which describes the state of the lock) to the access requests
  - The sequencer is a bit string that contains the name of lock, the mode (exclusive/shared), and the lock generation number
  - The client passes the sequencer to server. The server is expected to check if the sequencer is still valid and has the appropriate mode
- Heuristic II: Delay access
  - If a lock holder crashed, Chubby blocks the lock for a period called the lock delay

# Chubby: Replica Replacement

- What happens when a replica crashes?
  - If it does not recover for a few hours, a replacement system selects a fresh machine from a pool of machines
  - Subsequently, the DNS tables are updated by replacing the IP address of the failed replica with the new one
  - The master polls the DNS periodically and eventually notices the change

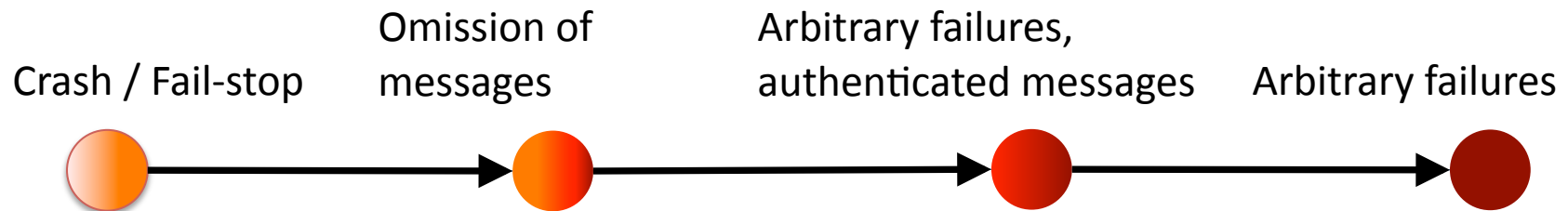


# Chubby: Performance

- According to Chubby...
  - Chubby performs quite well
- 90K+ clients can communicate with a single Chubby master (2 CPUs)
- System increases lease times from 12s up to 60s under heavy load
- Clients cache virtually everything
- Only little state has to be stored
  - All data is held in RAM (but also persistently stored on disk)

# Practical Byzantine Fault-Tolerance

- So far, we have only looked at systems that deal with simple (crash) failures
- We know that there are other kind of failures:



- Worst case: Malicious nodes
  - Nodes can lie, vote inconsistently, collude with each other

# PBFT

- The PraByzantine Fault-Tolerant system
- The system consists of clients that read/write data stored at  $n$  servers

## Goal

- The system can be used to implement any deterministic replicated service with a *state* and some *operations*
- Linearizability, reliability, and availability

## Model

- Communication is asynchronous, but message delays are bounded
- Messages may be lost, duplicated or may arrive out of order
- Messages can be authenticated using digital signatures (in order to prevent spoofing, replay, impersonation)
- At most  $f < n/3$  of the servers are Byzantine



# Why Does Paxos Not Work?

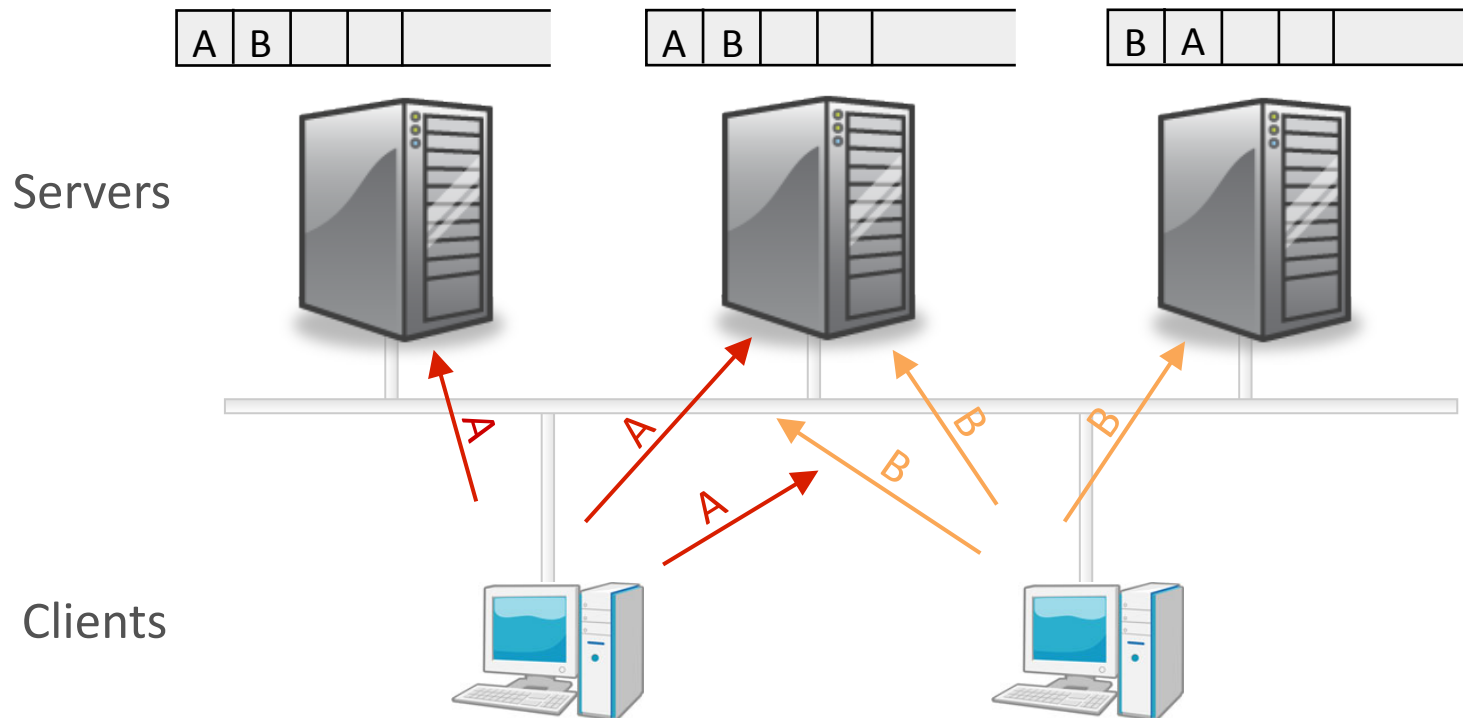
- Cannot rely on proposers or acceptors to respects sequence numbers
- Paxos uses majority accept quorum to allow  $f < n/2$  faults
- Bad node can vote “yes” in one majority vote (quorum), “no” in another
- So Paxos does not work for Byzantine faults

# PBFT: Order of Operations

- Replicated state machines:
- If:
  - All servers start in the same state
  - All operations are deterministic
  - All operations are executed in the same order
- Then:
  - All servers remain in the same state!

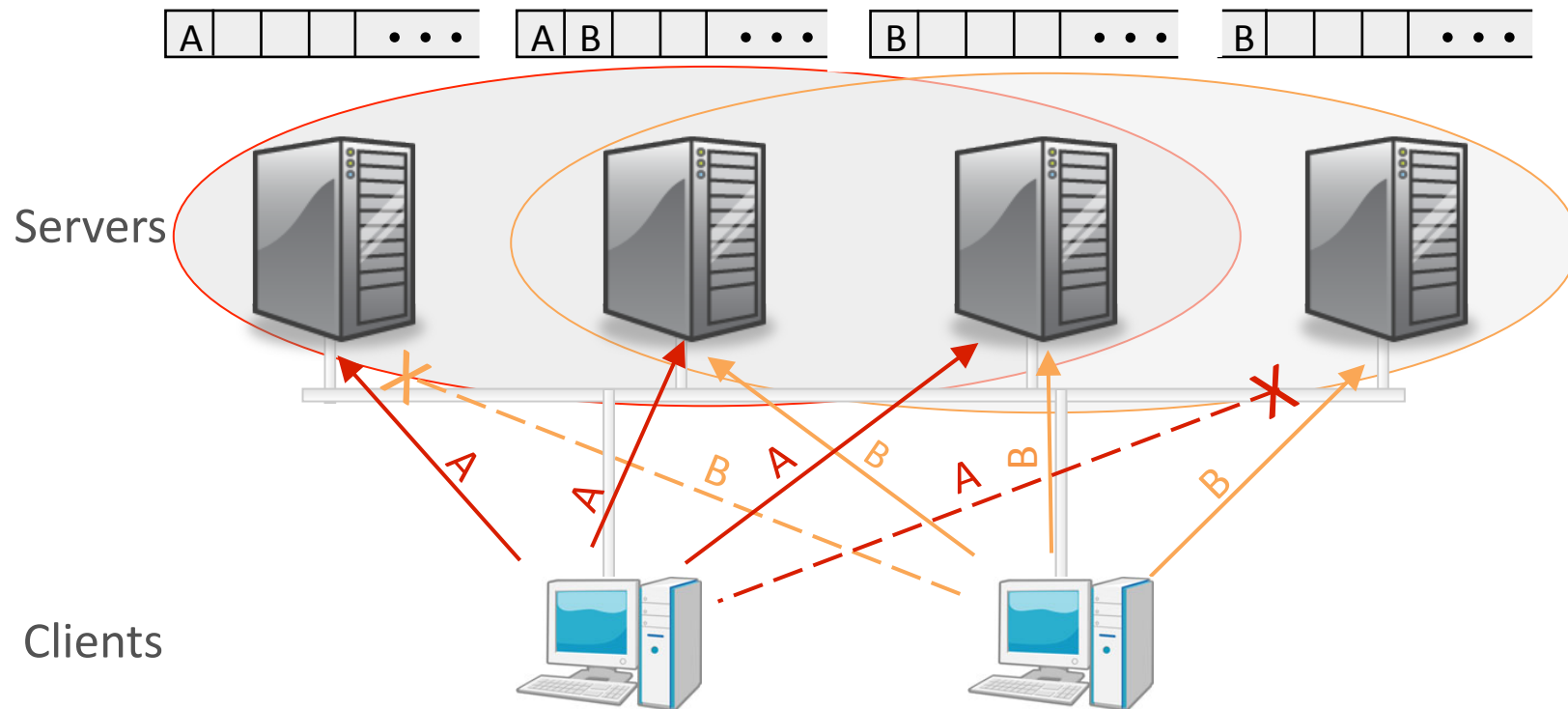
# PBFT: Order of Operations

- Variable message delays may be a problem:



# PBFT: Order of Operations

- If messages are lost, some servers may not receive all updates...

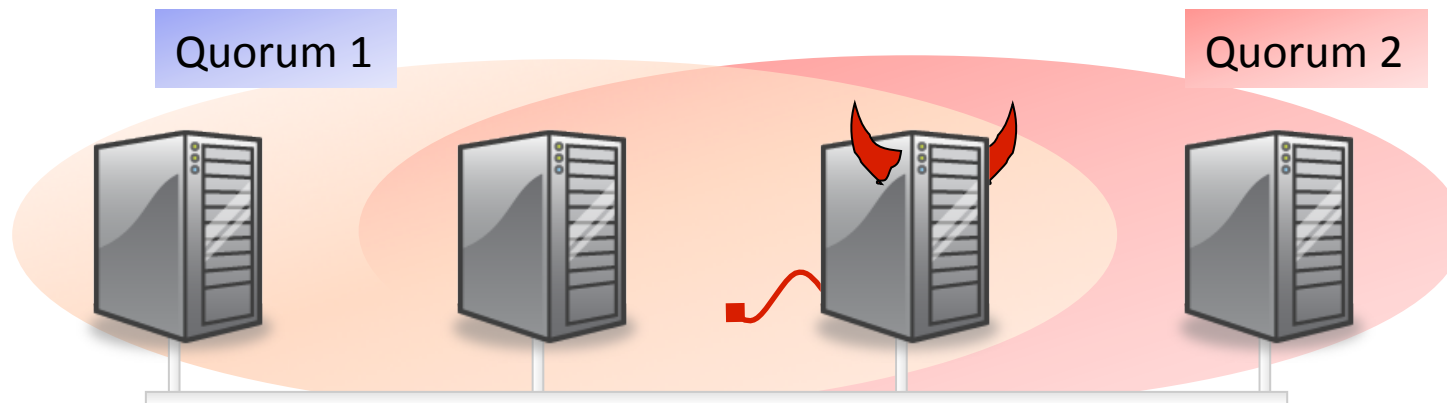


# PBFT: Basic Idea

- Such problems can be solved by using a *coordinator*
- One server is the *primary*
  - The clients send *signed* commands to the primary
  - The primary assigns sequence numbers to the commands
  - These sequence numbers impose an order on the commands
- The other servers are *backups*
  - The primary forwards commands to the other servers
  - Information about commands is replicated at a *quorum* of backups
  - Clients can access backups directly if she suspects problems

# Byzantine Quorums

- For liveness, the quorum size can be *at most*  $n - f$
- For correctness, any two quorums must intersect at least one correct node:
- So:  $(n - f) + (n - f) - n > f \Rightarrow n > 3f$
- Fix  $n = 3f + 1$ , quorum size  $2f + 1$

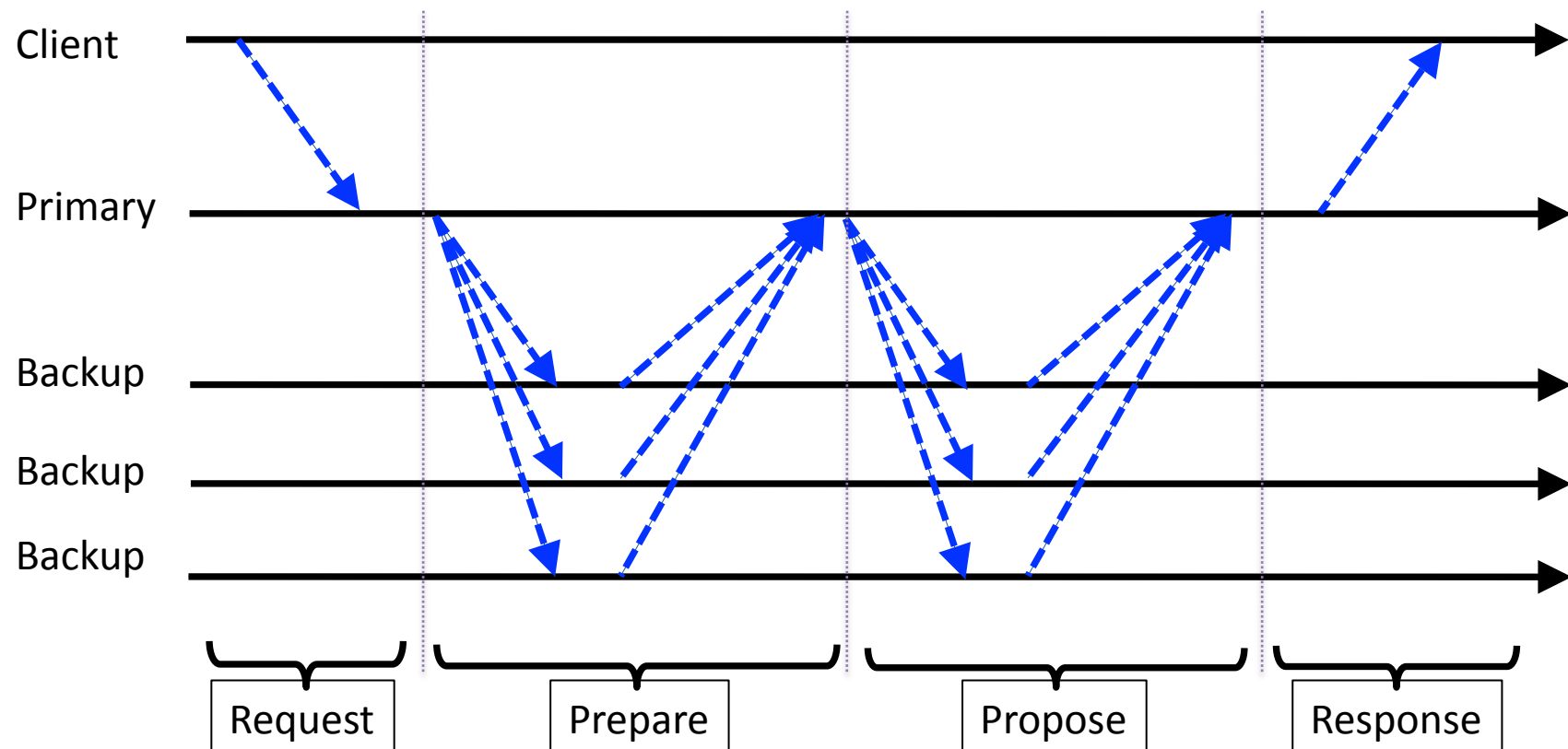


# PBFT: Main Algorithm

- 3 rounds in total . . . (unless . . . )
- In the first round, the client sends the command  $op$  to the primary
- The following three rounds are
  - Pre-prepare
  - Prepare
  - Propose
- In the fifth round, the client receives replies from the servers
  - If  $f+1$  (authenticated) replies are the same, the result is accepted
  - Since there are only  $f$  Byzantine servers, at least one correct server supports the result
- The algorithm is somewhat similar to Paxos

# PBFT: Paxos

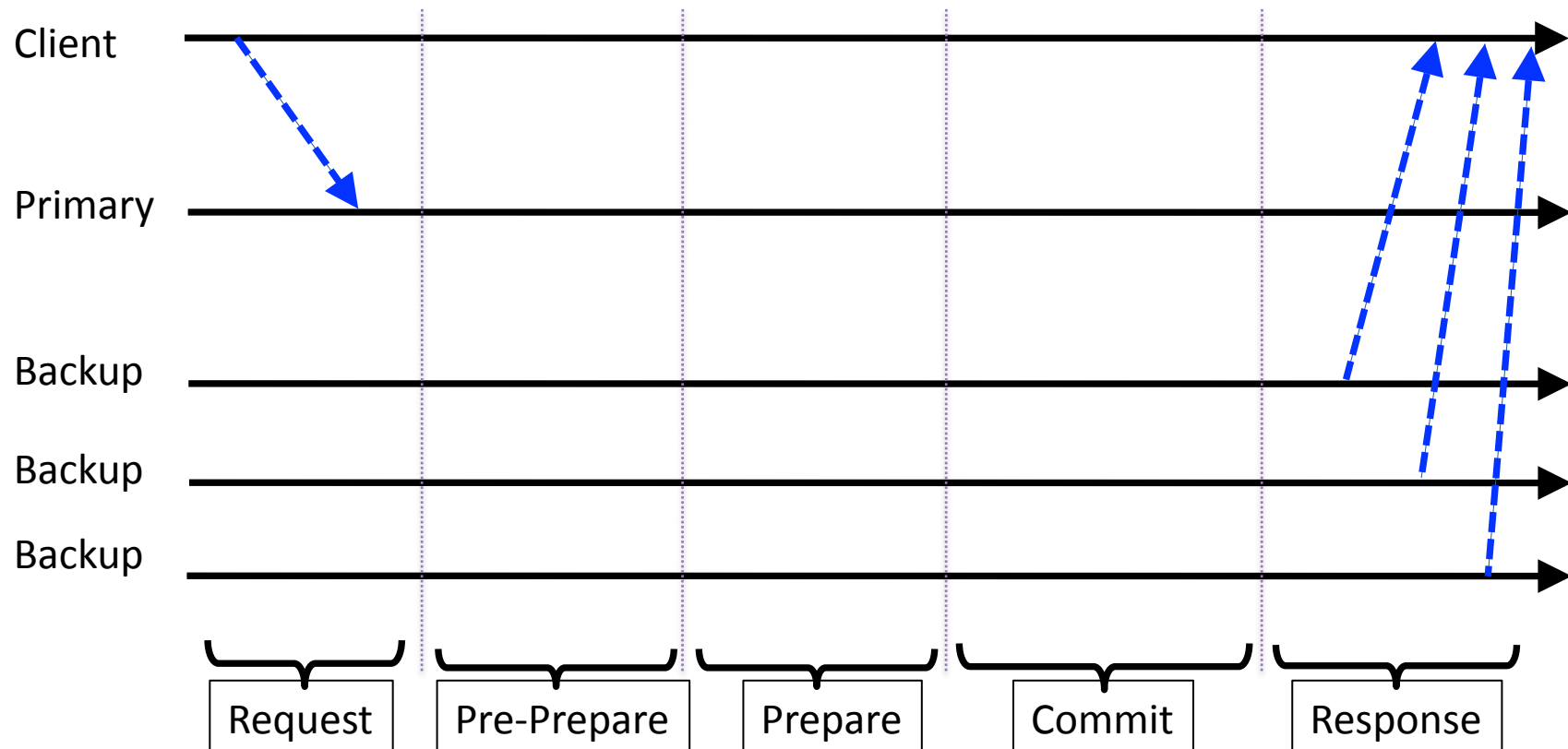
- In Paxos, there is only a prepare and a propose phase
- The primary is the node issuing the proposal
- In the response phase, the clients learn the final result





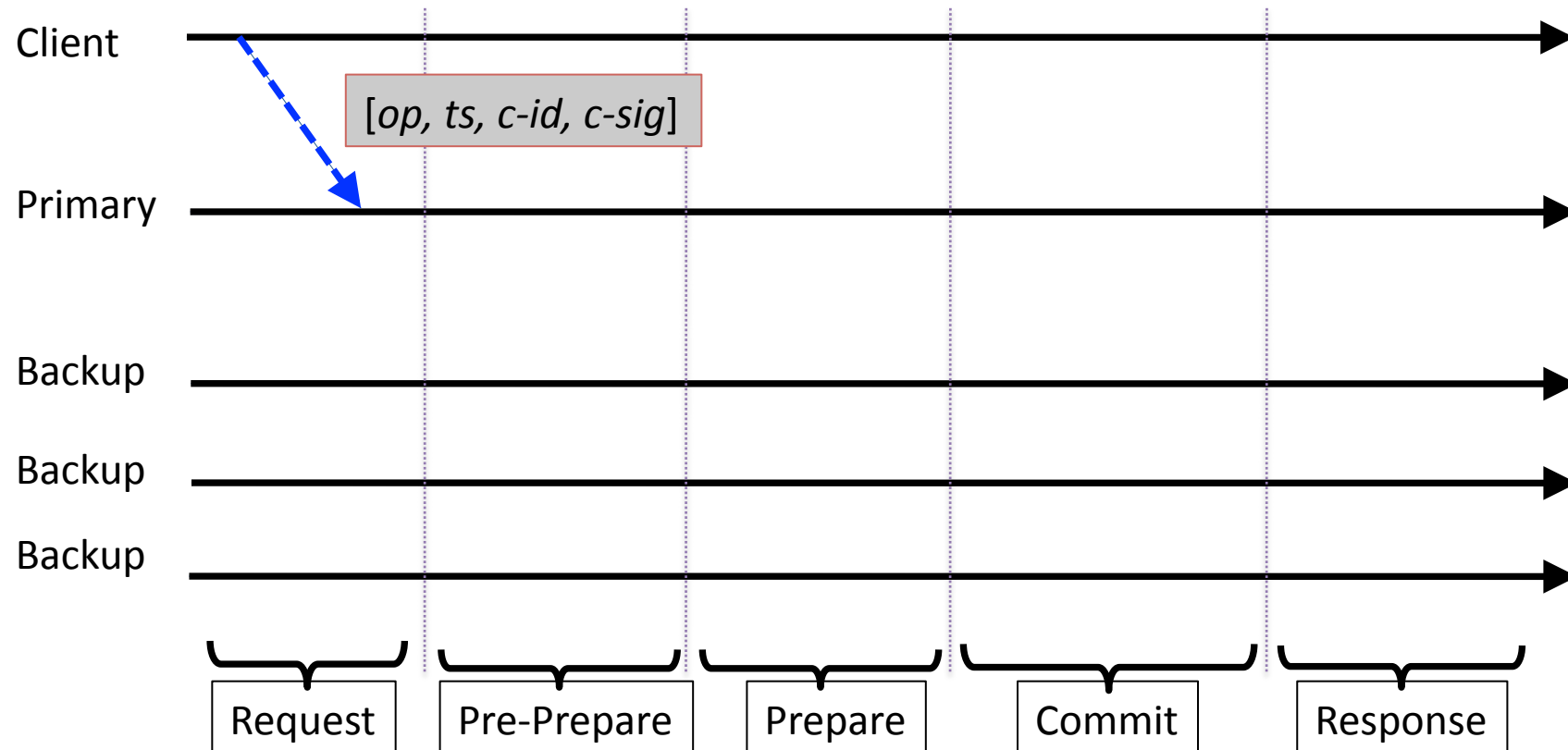
# PBFT: Algorithm

- PBFT takes 3 rounds of communication:
  - Pre-prepare, Prepare, Commit
  - + Request + Response



# PBFT: Request Phase

- Client sends the command  $op$  to the primary
- Along with timestamp  $ts$ , a client identifier  $c-id$  and a signature  $c-sig$

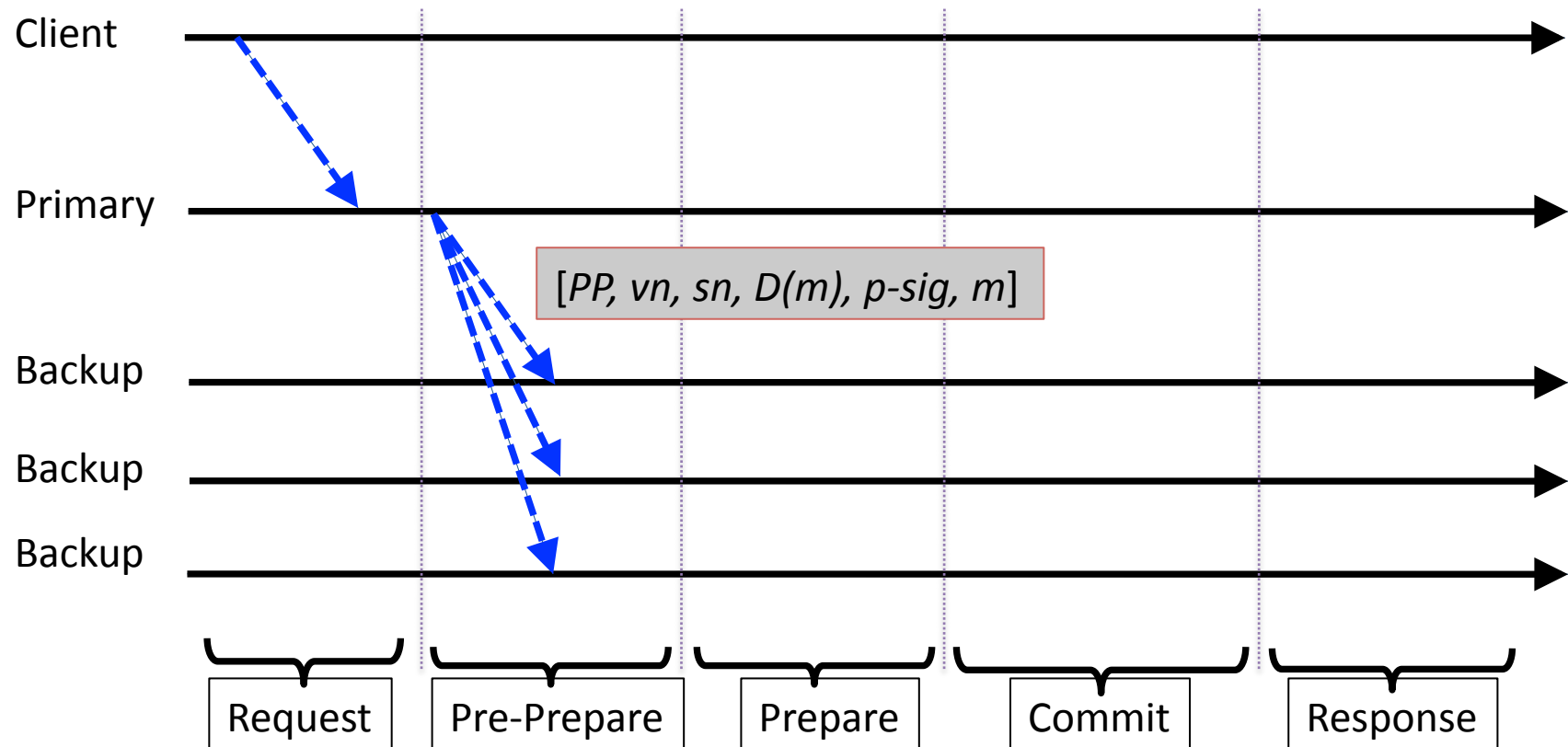


# PBFT: Request Phase

- Why adding a timestamp?
  - The timestamp ensures that a command is recorded/ executed exactly once
- Why adding a signature?
  - It is not possible for another client (or a Byzantine server) to issue commands that are accepted as commands from client  $c$
  - The system also performs access control: If a client  $c$  is allowed to write a variable  $x$  but  $c'$  is not,  $c'$  cannot issue a write command by pretending to be client  $c$ !

# PBFT: Pre-Prepare Phase

- Primary multicasts  $m = [op, ts, c-id, c-sig]$  to the backups, including a view number  $vn$ , an assigned sequence number  $sn$ , a message digest  $D(m)$ , and its own signature  $p-sig$



# PBFT: Pre-Prepare Phase

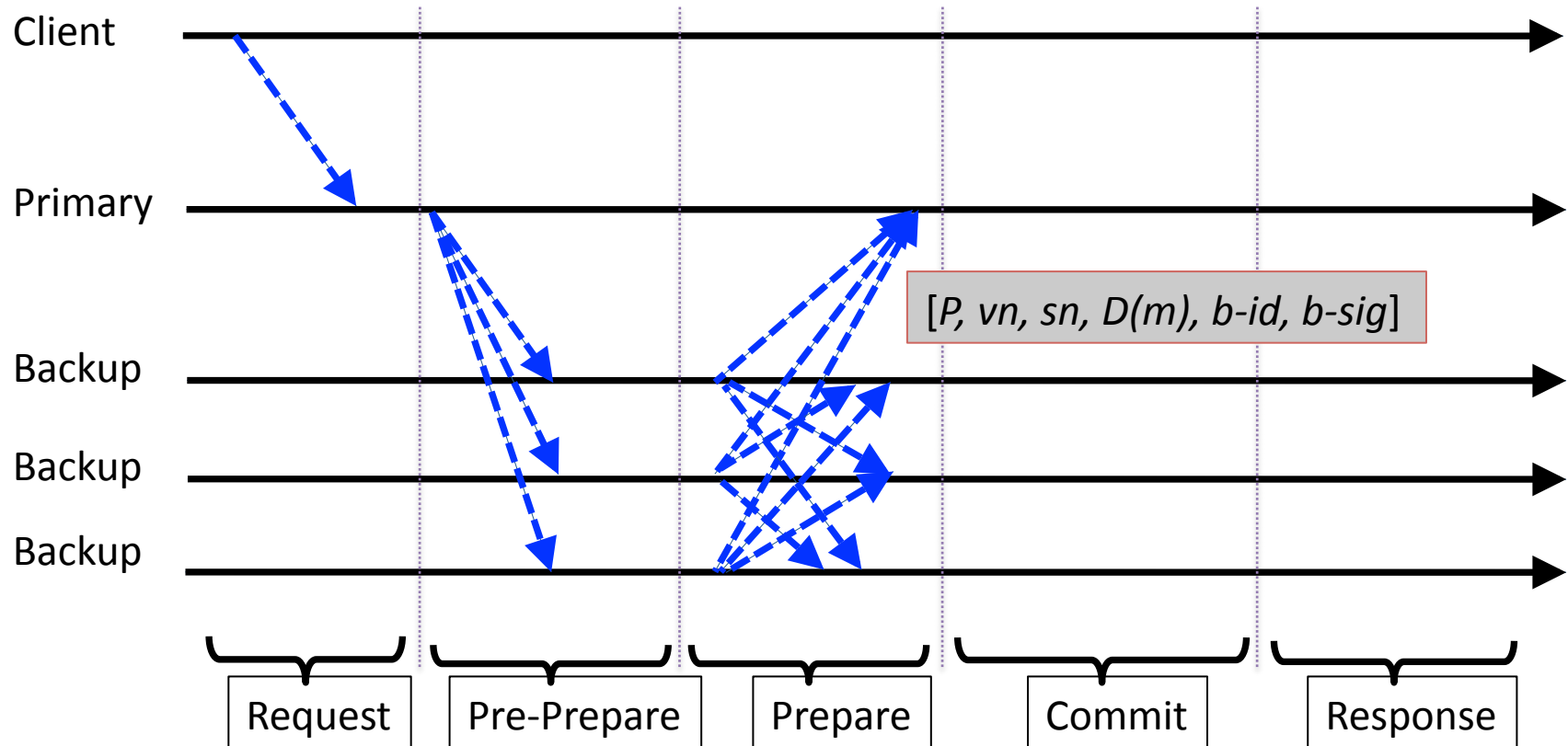
- Sequence number orders the commands
- Signature verifies message authenticity
- Why adding the message digest of the client's message?
  - The primary signs only  $[PP, vn, sn, D(m)]$ . This is more efficient!
- View?
  - A view is a configuration of the system. Here we assume that the system comprises the same set of servers, one of which is the primary
  - I.e., the primary determines the view: Two views are different if a different server is the primary
  - A view number identifies a view
  - The primary in view  $vn$  is the server whose identifier is  $vn \bmod n$
  - Ideally, all servers are (always) in the same view
  - A view change occurs if a different primary is elected

# PBFT: Pre-Prepare Phase

- A backup accepts a pre-prepare message if
  - the signatures are correct
  - $D(m)$  is the digest of  $m = [op, ts, cid, c-sig]$
  - it is in view  $vn$
  - It has not accepted a pre-prepare message for view number  $vn$  and sequence number  $sn$  containing a different digest
  - the sequence number is between a low water mark  $h$  and a high water mark  $H$
  - This prevents a faulty primary from exhausting the space of sequence numbers
- Each accepted pre-prepare message is stored in the local log

# PBFT: Prepare Phase

- If a backup  $b$  accepts the pre-prepare message, it enters the prepare phase and multicasts  $[P, vn, sn, D(m), b-id, b-sig]$  to all other replicas and stores this prepare message in its log



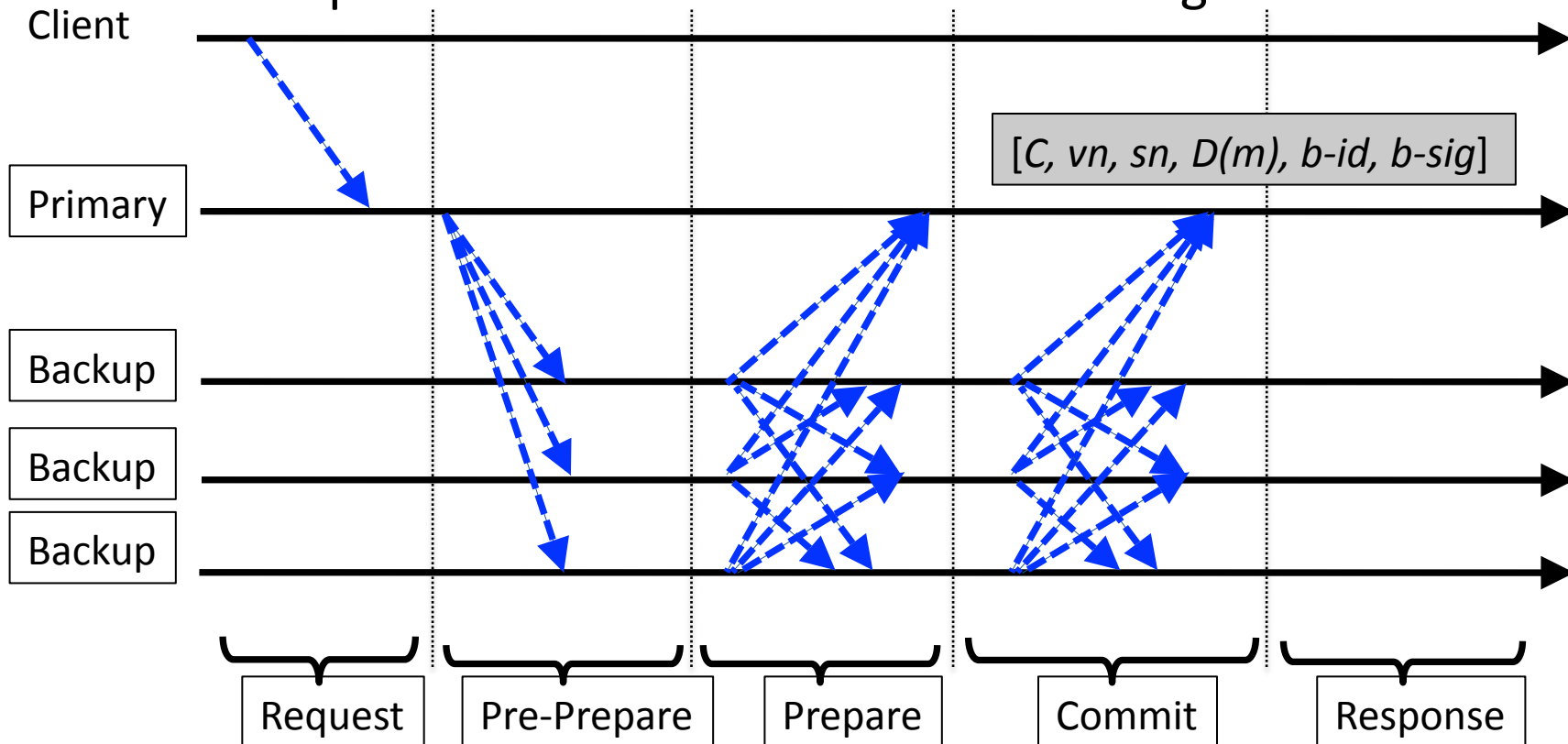
# PBFT: Prepare Phase

- A replica (including the primary) accepts a prepare message if
  - the signatures are correct
  - it is in view  $vn$
  - the sequence number is between a low water mark  $h$  and a high water mark  $H$
- Each accepted prepare message is also stored in the local log



# PBFT: Commit Phase

- If backup  $b$  has accepted pre-prepare message  $m$ , and  $2f + 1$  corresponding accepted prepare messages from different replicas in its log, it multicasts  $[C, vn, sn, D(m), b-id, b-sig]$  to all other replicas and stores this commit message

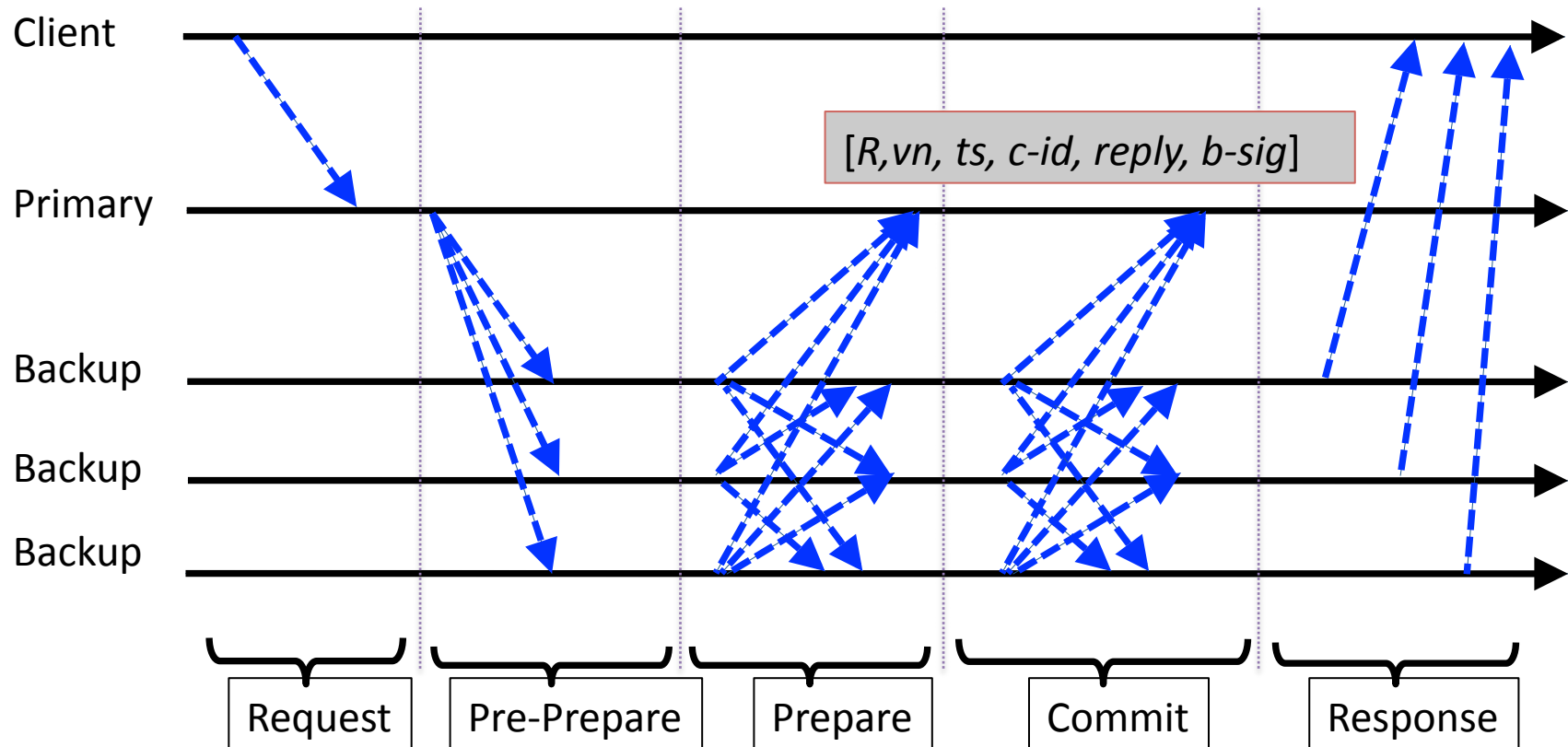


# PBFT: Commit Phase

- A replica (including the primary) accepts a commit message if
  - the signatures are correct
  - it is in view  $vn$
  - the sequence number is between a low water mark  $h$  and a high water mark  $H$
- Each accepted commit message is also stored in the local log

# PBFT: Response Phase

- If a backup  $b$  has accepted  $2f+1$  matching commit messages, it performs  $op$  (“commits”), record operation in log, and sends a reply to the client



# PBFT: Garbage Collection

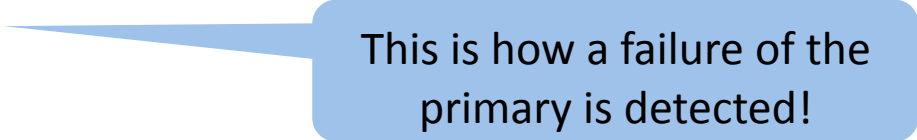
- The servers store all messages in their log
- In order to discard messages in the log, the servers create *checkpoints* (snapshots of the state) every once in a while
- A checkpoint contains the  $2f+1$  signed commit messages for the committed commands in the log
- The checkpoint is multicast to all other servers
- If a server receives  $2f+1$  matching checkpoint messages, the checkpoint becomes stable and any command that preceded the commands in the checkpoint are discarded
- Note that the checkpoints are also used to set the low water mark  $h$ 
  - to the sequence number of the last stable checkpointand the high water mark  $H$ 
  - to a “sufficiently large” value

# PBFT: Correct Primary

- If the primary is correct, the algorithm works
  - All  $2f+1$  correct nodes receive pre-prepare messages and send prepare messages
  - All  $2f+1$  correct nodes receive  $2f+1$  prepare messages and send commit messages
  - All  $2f+1$  correct nodes receive  $2f+1$  commit messages, commit, and send a reply to the client
  - The client accepts the result

# PBFT: No Replies

- What happens if the client does not receive replies?
  - Because the command message has been lost
  - Because the primary is Byzantine and did not forward it
- After a time-out, the client multicasts the command to all servers
  - A server that has already committed the result sends it again
  - A server that is still processing it ignores it
  - A server that has not received the pre-prepare message forwards the command to the primary
  - If the server does not receive the pre-prepare message in return after a certain time, it concludes that the primary is faulty/Byzantine



This is how a failure of the primary is detected!

# PBFT: View Change

- If a server suspects that the primary is faulty
  - it stops accepting messages except checkpoint, view change and new view messages
  - it sends a view change message containing the identifier  $i = vn+1 \bmod n$  of the next primary and also a certificate for each command for which it accepted  $2f+1$  prepare messages
  - The reason for view change must be verifiable!
  - A certificate simply contains the  $2f+1$  accepted signatures
- When server  $i$  receives  $2f$  view change messages from other servers, it broadcasts a new view message containing the signed view change
- The servers verify the signature and accept the view change!
- The new primary issues pre-prepare messages with the new view number for all commands with a correct certificate

# PBFT: Ordered Commands

- Commands are totally ordered using the view numbers and the sequence numbers
- We must ensure that a certain  $(vn, sn)$  pair is always associated with a unique command  $m$ !
- If a correct server committed  $[m, vn, sn]$ , then no other correct server can commit  $[m', vn, sn]$  for any  $m \neq m'$  s.t.  $D(m) \neq D(m')$ 
  - If a correct server committed, it accepted a set of  $2f+1$  authenticated commit messages
  - The intersection between two such sets contains at least  $f+1$  authenticated commit messages
  - There is at least one correct server in the intersection
  - A correct server does not issue (pre-)prepare messages with the same  $vn$  and  $sn$  for different  $m$ !



# PBFT: Correctness

**Theorem:** If a client accepts a result, no correct server commits a different result

**Proof:**

- A client only accepts a result if it receives  $f+1$  authenticated messages with the same result
- At least one correct server must have committed this result
- As we argued on the previous slide, no other correct server can commit a different result

# PBFT: Liveness

**Theorem:** PBFT terminates eventually

**Proof:**

- Case 1: The primary is correct
  - As we argued before, the algorithm terminates after  $3 + 2$  rounds if no messages are lost
  - Message loss is handled by retransmitting after certain time-outs
  - Assuming that messages arrive eventually, the algorithm also terminates eventually

# PBFT: Liveness, II

**Theorem:** PBFT terminates eventually

**Proof:**

- Case 2: The primary is Byzantine
  - If the client does not accept an answer in a certain period of time, it sends its command to all servers
  - In this case, the system behaves as if the primary is correct and the algorithm terminates eventually!
- Thus, the Byzantine primary cannot delay the command indefinitely. As we saw before, if the algorithm terminates, the result is correct!
  - i.e., at least one correct server committed this result

# PBFT: Evaluation

The Andrew benchmark\* emulates a software development workload  
It has 5 phases:

1. Create subdirectories recursively
2. Copy a source tree
3. Examine the status of all the files in the tree without examining the data
4. Examine every byte in all the files
5. Compile and link the files

It is used to compare 3 systems

- BFS (PBFT) and 4 replicas and BFS-nr (PBFT without replication)
- BFS (PBFT) and NFS-std (network file system)

Measured normal-case behavior (i.e. no view changes) in an isolated network

\*: J. H. Howard et al, "Scale and performance in a distributed file system," Trans. Comp. Sys., vol. 6, 1988

# PBFT: Evaluation

- Most operations in NFS V2 are not read-only (r/o)
  - E.g., *read* and *lookup* modify the time-last-accessed attribute
- A second version of PBFT has been tested in which lookups are read-only
- Normal (strict) PBFT is only 26% slower than PBFT without replication
  - Replication does not cost too much!
- Normal (strict) PBFT is only 3% slower than NFS-std, and PBFT with read-only lookups is even 2% faster!

phase	BFS		BFS-nr
	strict	r/o lookup	
1	0.55 (57%)	0.47 (34%)	0.35
2	9.24 (82%)	7.91 (56%)	5.08
3	7.24 (18%)	6.45 (6%)	6.11
4	8.77 (18%)	7.87 (6%)	7.41
5	38.68 (20%)	38.38 (19%)	32.12
total	64.48 (26%)	61.07 (20%)	51.07

phase	BFS		NFS-std
	strict	r/o lookup	
1	0.55 (-69%)	0.47 (-73%)	1.75
2	9.24 (-2%)	7.91 (-16%)	9.46
3	7.24 (35%)	6.45 (20%)	5.36
4	8.77 (32%)	7.87 (19%)	6.60
5	38.68 (-2%)	38.38 (-2%)	39.35
total	64.48 (3%)	61.07 (-2%)	62.52

Times are in seconds  
“Phases” are phases in  
the Andrew benchmark!

# PBFT: Discussion

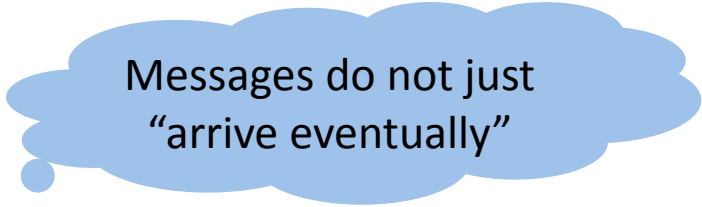
- PBFT guarantees that the commands are totally ordered
- If a client accepts a result, it knows that at least one correct server supports this result
- Disadvantages:
- Commit not at all correct servers
  - It is possible that only one correct server commits the command
  - We know that  $f$  other correct servers have sent commit, but they may only receive  $f+1$  commits and therefore do not commit themselves...
- Byzantine primary can slow down the system
  - Ignore the initial command
  - Send pre-prepare always after the other servers forwarded the command
  - No correct server will force a view change!

Recall that quorum size is  $2f+1$

# Beating the Lower Bounds...

- We know several crucial impossibility results and lower bounds
  - No deterministic algorithm can achieve consensus in asynchronous systems even if only one node may crash
  - Any deterministic algorithm for synchronous systems that tolerates  $f$  crash failures takes at least  $f+1$  rounds
- Yet we have just seen a deterministic algorithm/system that
  - achieves consensus in asynchronous systems and that tolerates  $f < n/3$  Byzantine failures
  - The algorithm only takes five rounds...?
- So, why does the algorithm work...?

# Beating the Lower Bounds...

- So, why does the algorithm work...?
- It is not really an asynchronous system
  - There are bounds on the message delays
  - This is almost a synchronous system...
- We used authenticated messages
  - It can be verified if a server really sent a certain message
- The algorithm takes more than 5 rounds in the worst case
  - It takes more than  $f$  rounds!
  - How is that?

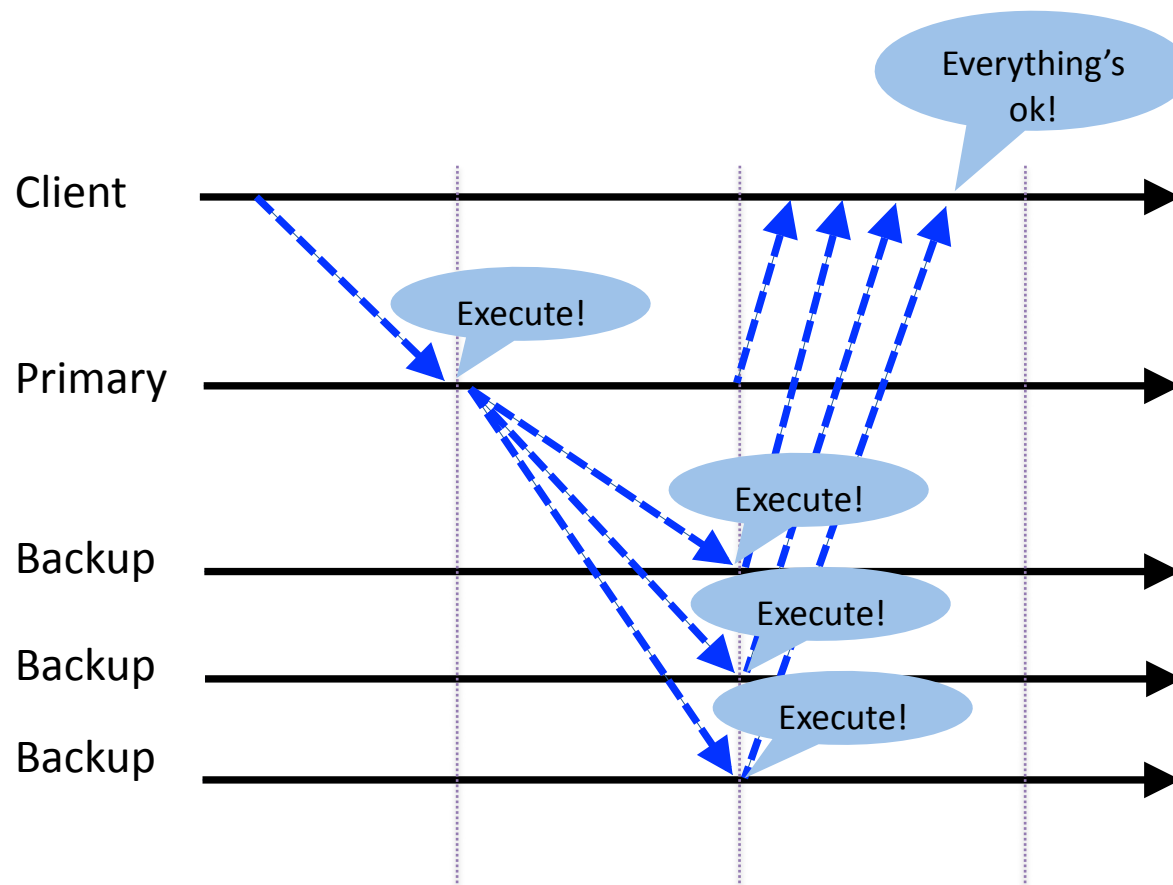


# Zyzyva

- Zyzyva is another BFT protocol
- **Idea**
  - The protocol should be very efficient if there are no failures
  - The clients *speculatively execute* the command without going through an agreement protocol!
- **Problem**
  - States of correct servers may diverge
  - Clients may receive diverging/conflicting responses
- **Solution**
  - Clients detect inconsistencies in the replies and help the correct servers to converge to a single total ordering of requests

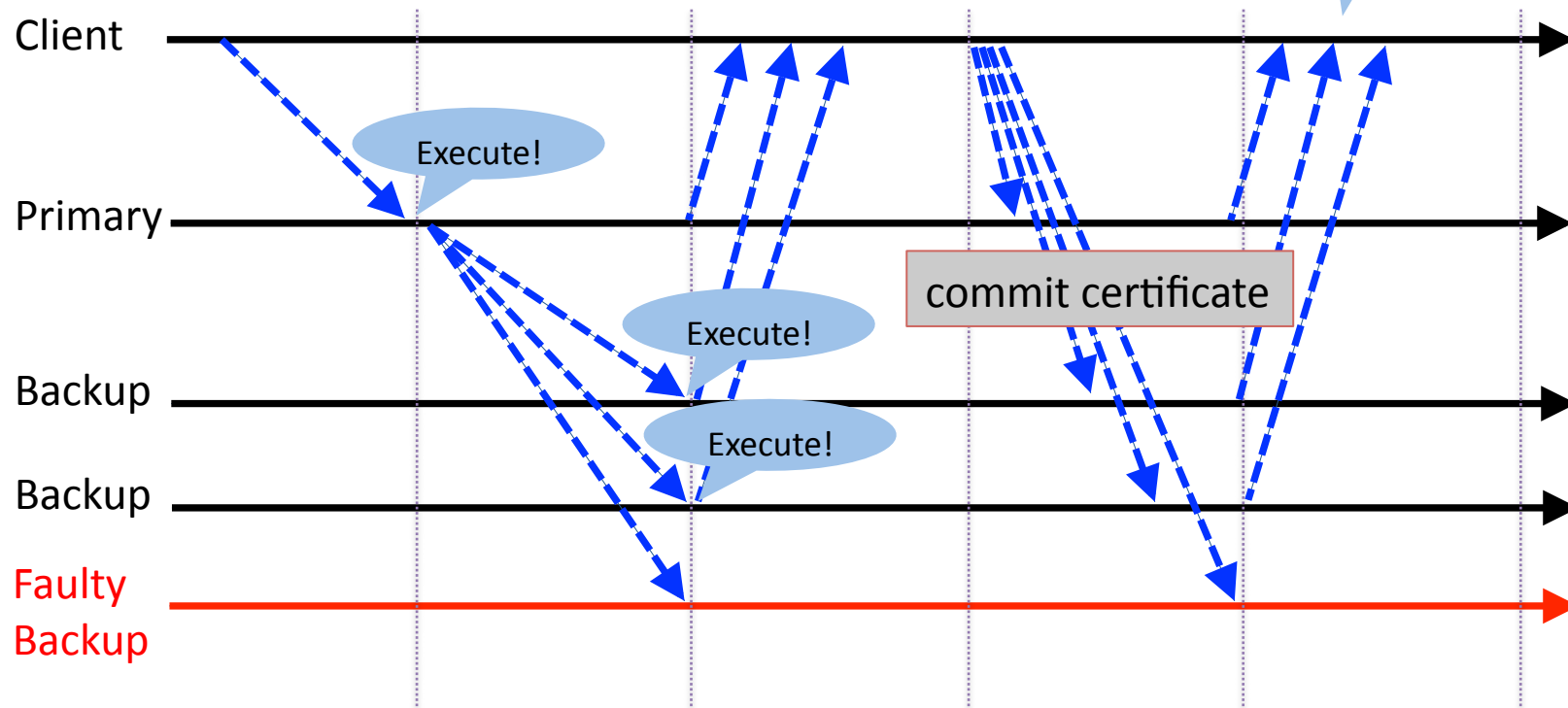
# Zyzyva

- Normal operation: Speculative execution!
- Case 1: All  $3f+1$  report the same result



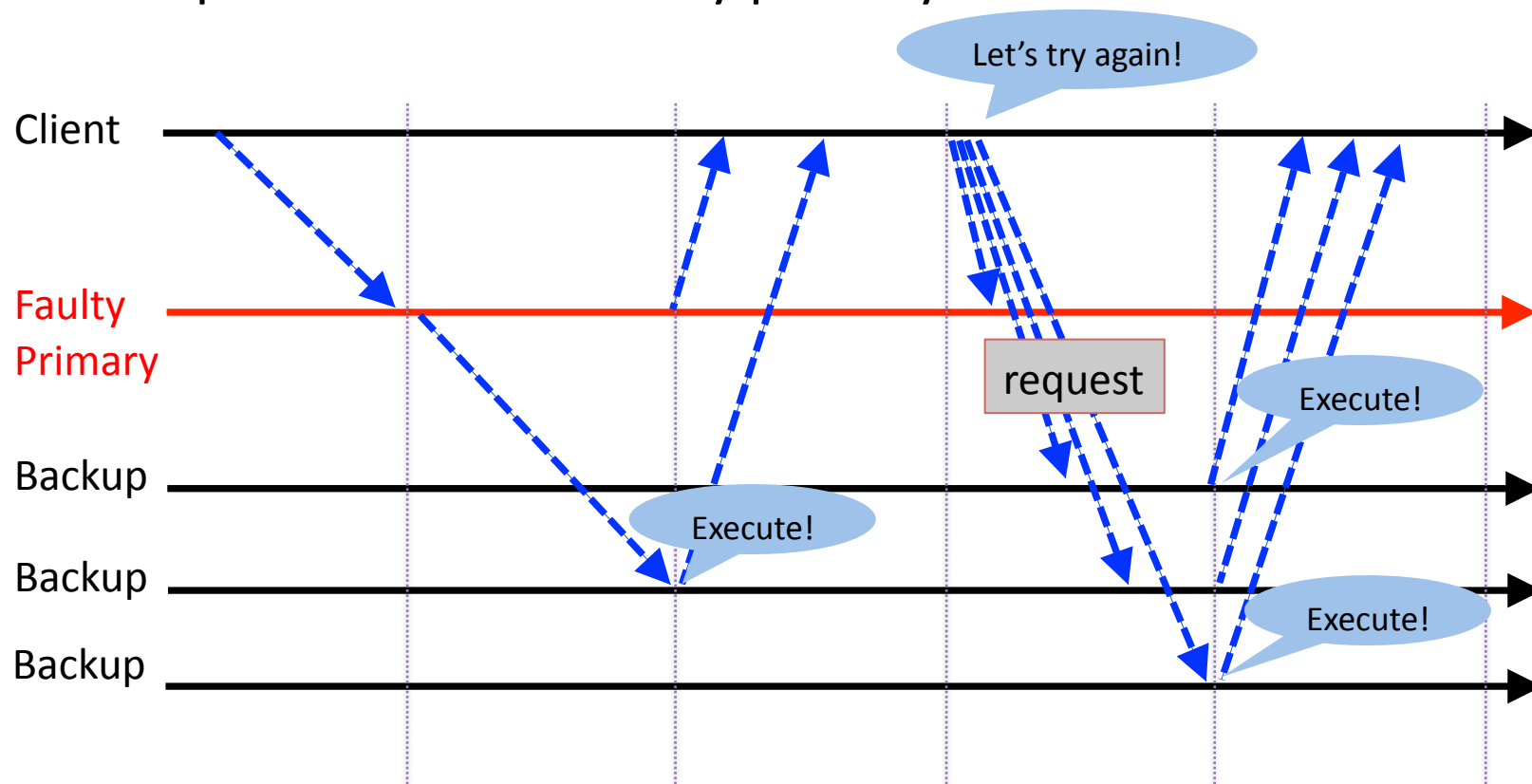
# Zyzyva

- Case 2: Between  $2f+1$  and  $3f$  results are the same
- The client broadcasts a commit certificate containing the  $2f+1$  results
- The client commits upon receiving  $2f+1$  replies



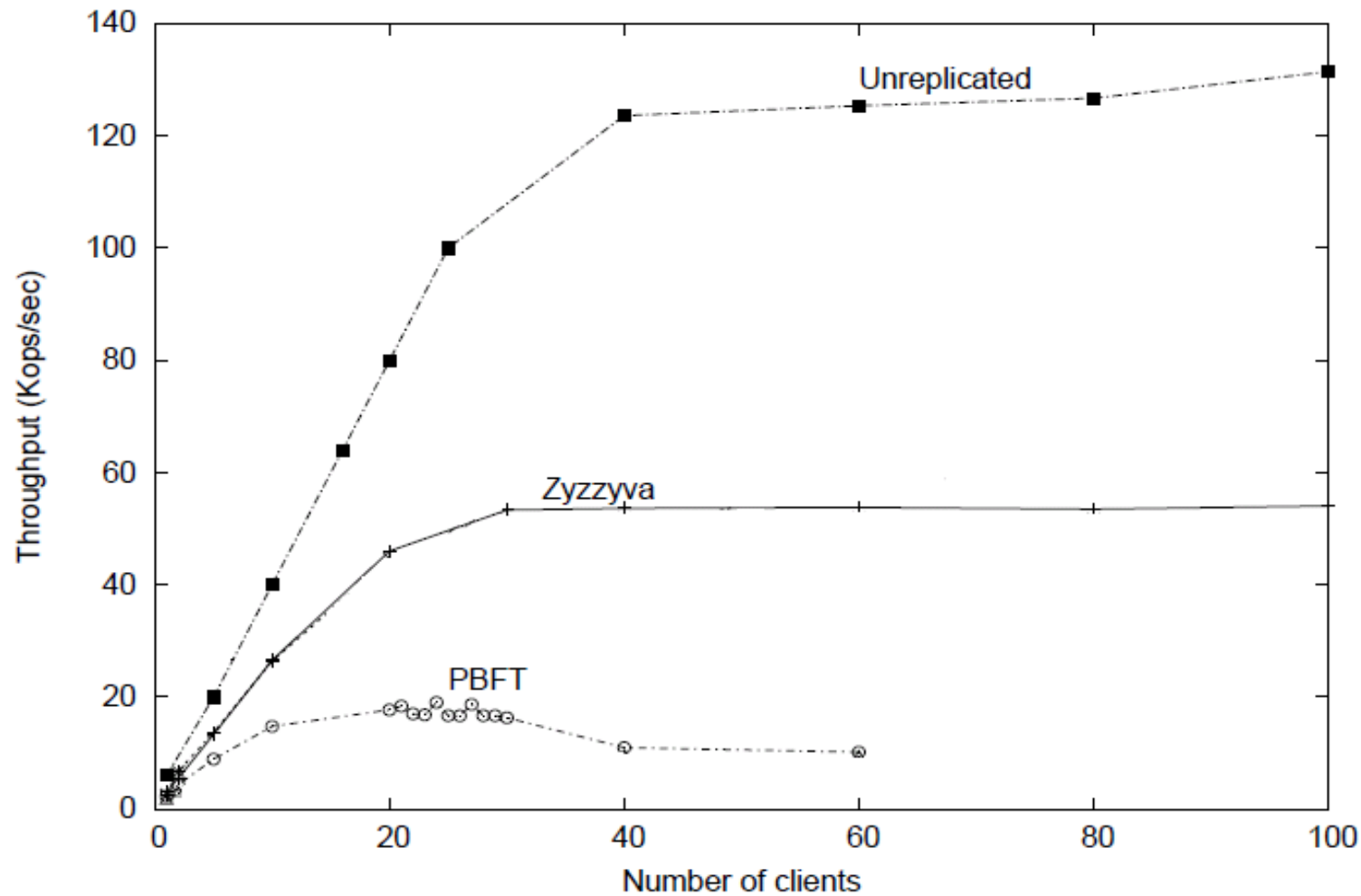
# Zyzyva

- Case 3: Less than  $2f+1$  replies are the same
- The client broadcasts its request to all servers
- This step circumvents a faulty primary



# Zyzyva: Evaluation

- Zyzyva outperforms PBFT - it normally takes only 3 rounds!



# What Else Is Going On?

- Lots . . .
- Techniques to tell if node reports a correct value
- Sybil attacks:
  - One node can pretend to be many
  - Difficult to guard against
- Use hash chaining to create auditable logs
  - Only partial solutions
- Use remote attestation
  - Secure coprocessors (“sim cards”, TPM’s)
  - Can implement a secure log
  - Can allow smaller quorum size

# Some References

- PeerReview:  
Haeberlen, Kouznetsov, Druschel: PeerReview: practical accountability for distributed systems, SOSP'07  
Secure nodes, hash chaining, audit logs
- Chun, Maniatis, Shenker, Kubiawicz: Attested append-only memory: Making adversaries stick to their words, SOSP'07  
Secure log using trusted computing – improves PBFT to  $n \geq 2f$
- Singh et al: Zeno: Eventually consistent Byzantine-fault tolerance, NSDI'09  
BFT for high availability instead of strong consistency