Strong Consistency

Part 2, Chapter 2

Roger Wattenhofer

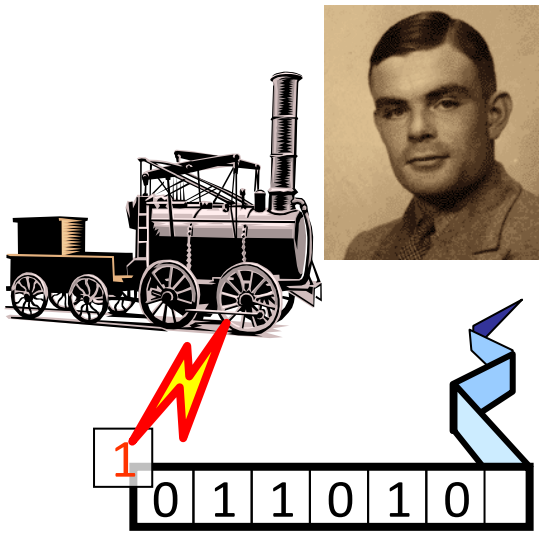
Overview

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

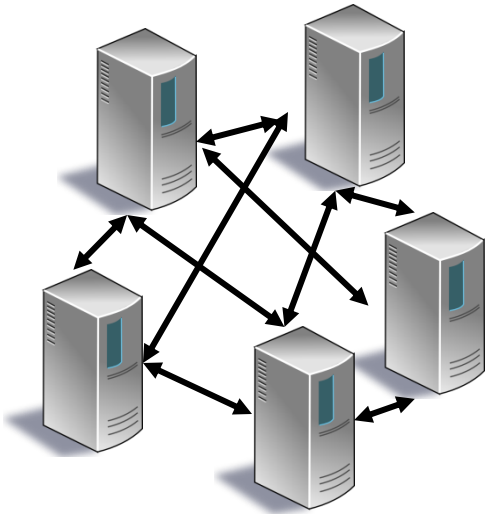
Computability vs. Efficiency

- 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?

Worst-case scenarios!

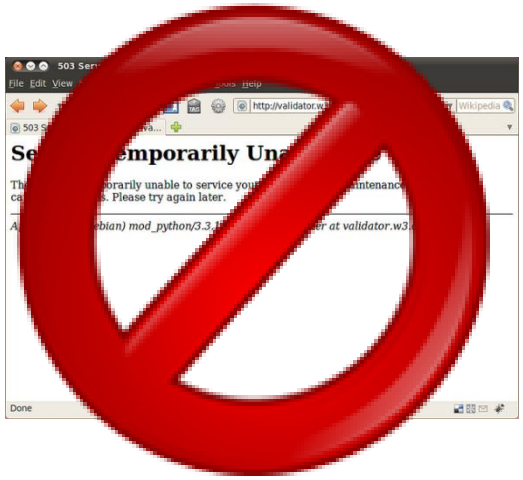


- In this part, we consider practical solutions
  - Simple approaches that work well in practice
  - Focus on efficiency

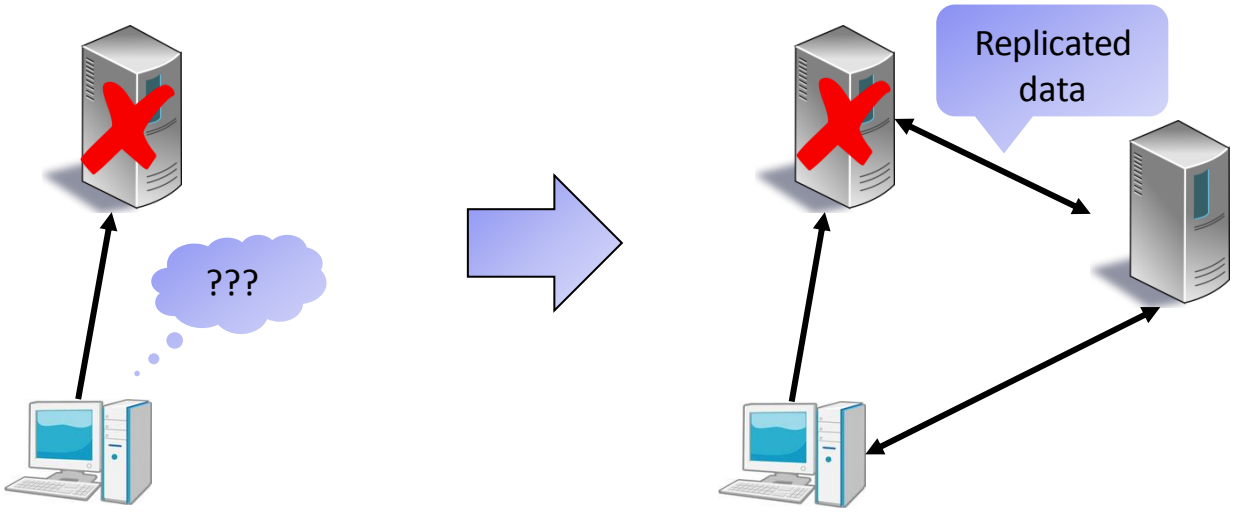



503 Service Temporarily Unavailable - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://validator.w3.org/feed/
Wikipedia
503 Service Temporarily Unava...
Service Temporarily Unavailable
The server is temporarily unable to service your request due to maintenance downtime or capacity problems. Please try again later.
Apache/2.2.9 (Debian) mod_python/3.3.1 Python/2.5.2 Server at validator.w3.org Port 80
Done

Fault-Tolerance in Practice



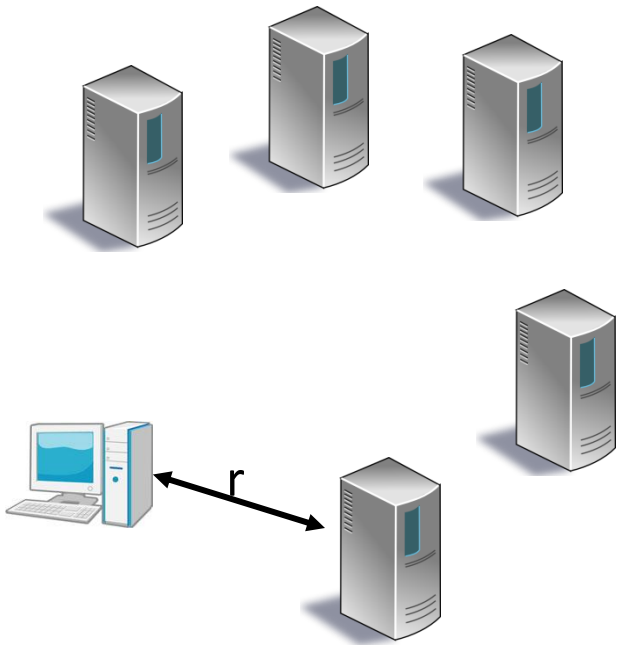
- Fault-Tolerance is achieved through *replication*



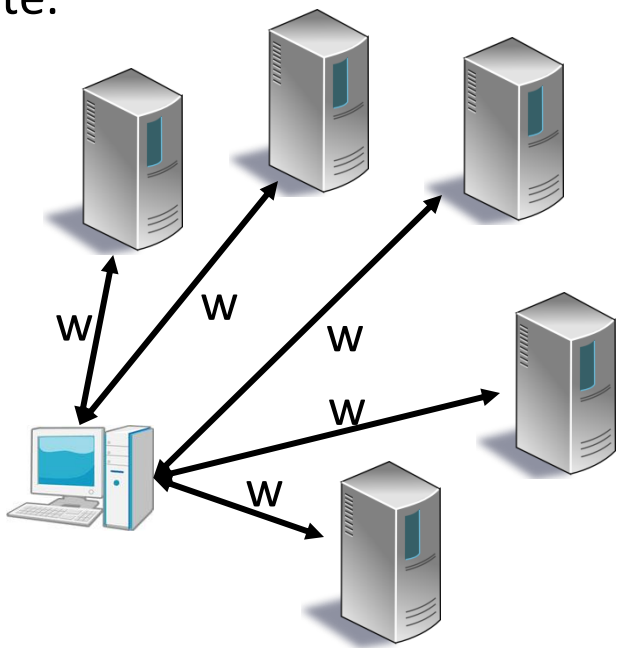
Replication is Expensive

- 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:



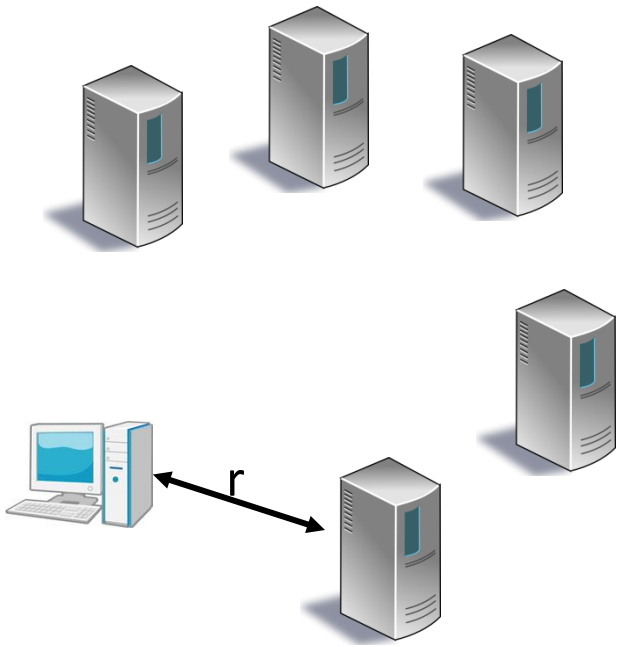
Write:



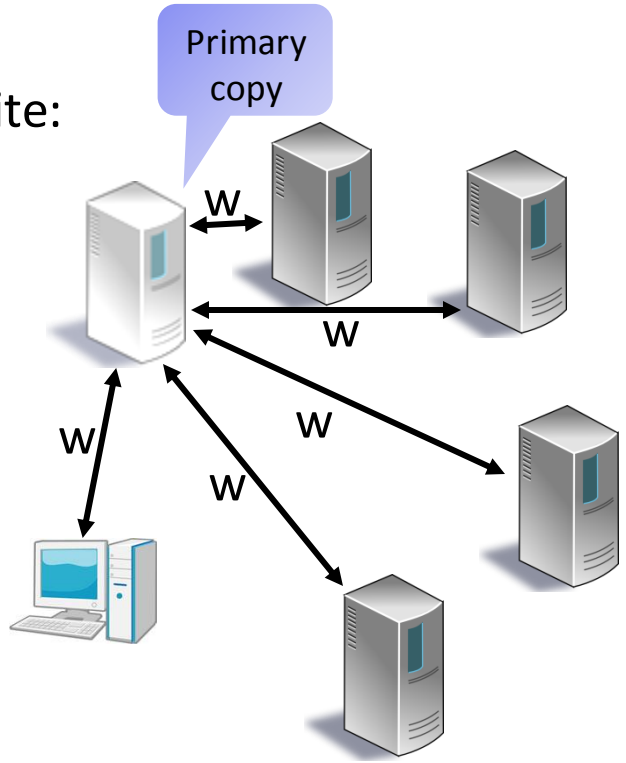
Primary Copy

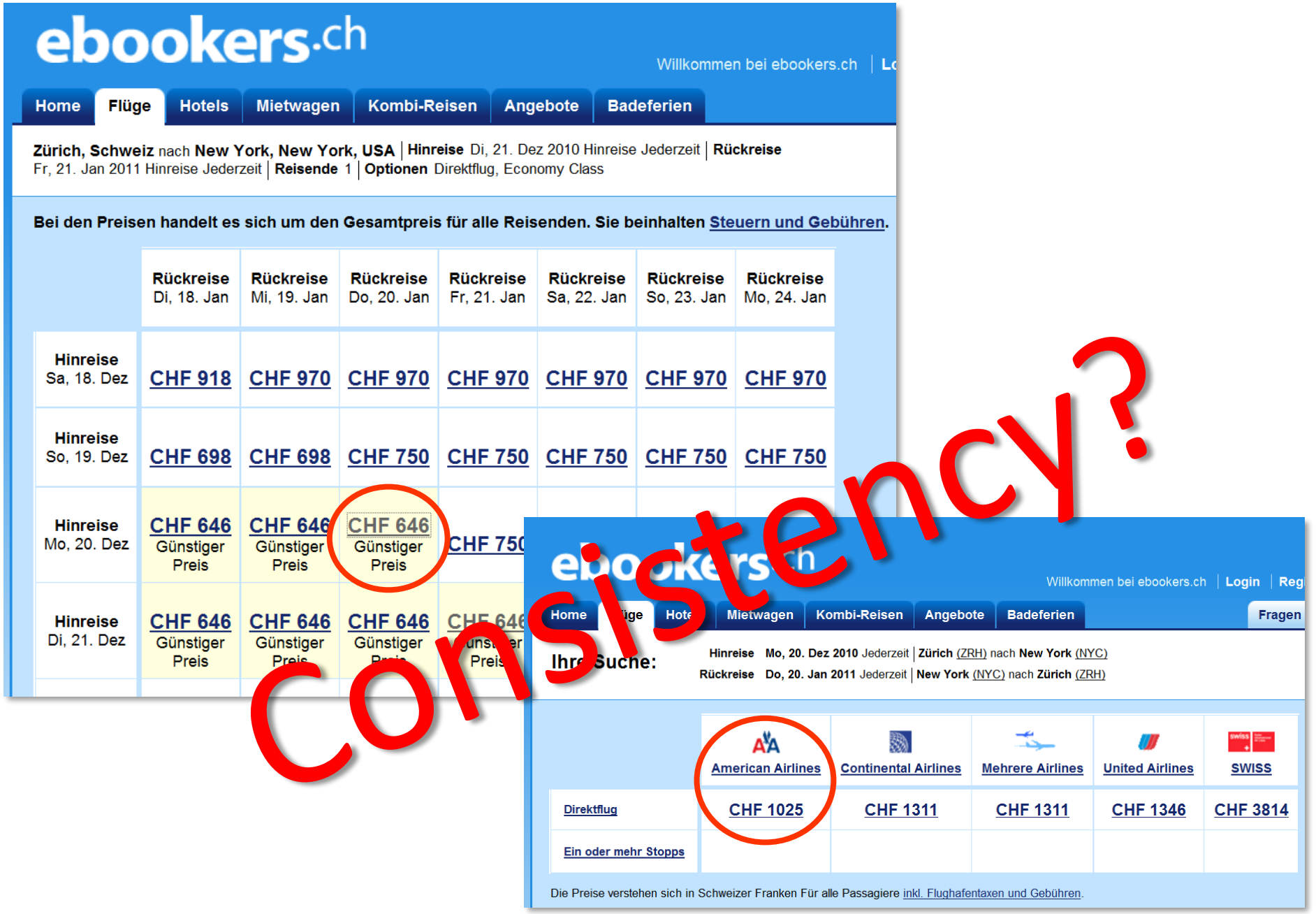
- Can we reduce the load on the clients?
- Yes! Write only to one server (the primary copy), and let primary copy distribute the update
  - This way, the client only sends one message in order to read and write

Read:



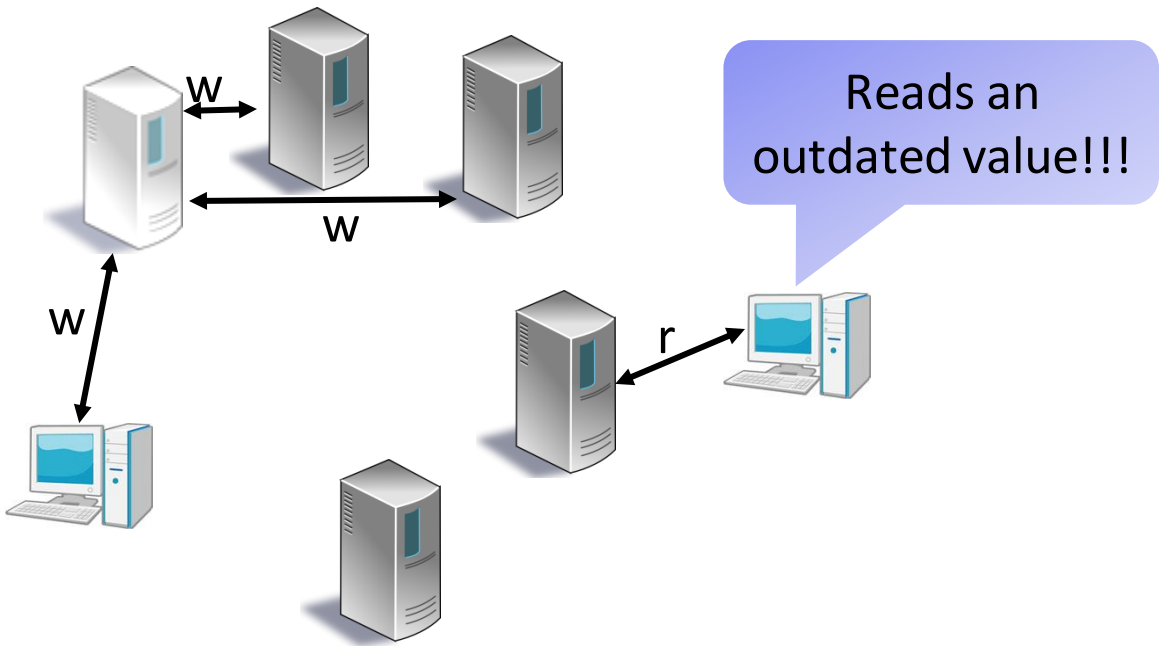
Write:




ebookers.ch
Willkommen bei ebookers.ch
Home
Flüge
Hotels
Mietwagen
Kombi-Reisen
Angebote
Badeferien
Zürich, Schweiz nach New York, New York, USA | Hinreise Di, 21. Dez 2010 Hinreise Jederzeit | Rückreise Fr, 21. Jan 2011 Hinreise Jederzeit | Reisende 1 | Optionen Direktflug, Economy Class
Bei den Preisen handelt es sich um den Gesamtpreis für alle Reisenden. Sie beinhalten Steuern und Gebühren.
Rückreise Di, 18. Jan
Rückreise Mi, 19. Jan
Rückreise Do, 20. Jan
Rückreise Fr, 21. Jan
Rückreise Sa, 22. Jan
Rückreise So, 23. Jan
Rückreise Mo, 24. Jan
Hinreise Sa, 18. Dez
CHF 918
CHF 970
CHF 970
CHF 970
CHF 970
CHF 970
CHF 970
Hinreise So, 19. Dez
CHF 698
CHF 698
CHF 750
CHF 750
CHF 750
CHF 750
CHF 750
Hinreise Mo, 20. Dez
CHF 646
Günstiger Preis
CHF 646
Günstiger Preis
CHF 646
Günstiger Preis
Hinreise Di, 21. Dez
CHF 646
Günstiger Preis
CHF 646
Günstiger Preis
CHF 646
Günstiger Preis
Consistency?
ebookers.ch
Willkommen bei ebookers.ch | Login | Reg
Home
Mietwagen
Kombi-Reisen
Angebote
Badeferien
Fragen
Ihre Suche:
Hinreise Mo, 20. Dez 2010 Jederzeit | Zürich (ZRH) nach New York (NYC)
Rückreise Do, 20. Jan 2011 Jederzeit | New York (NYC) nach Zürich (ZRH)
American Airlines
Continental Airlines
Mehrere Airlines
United Airlines
SWISS
Direktflug
CHF 1025
CHF 1311
CHF 1311
CHF 1346
CHF 3814
Ein oder mehr Stopps
Die Preise verstehen sich in Schweizer Franken Für alle Passagiere inkl. Flughafentaxen und Gebühren.

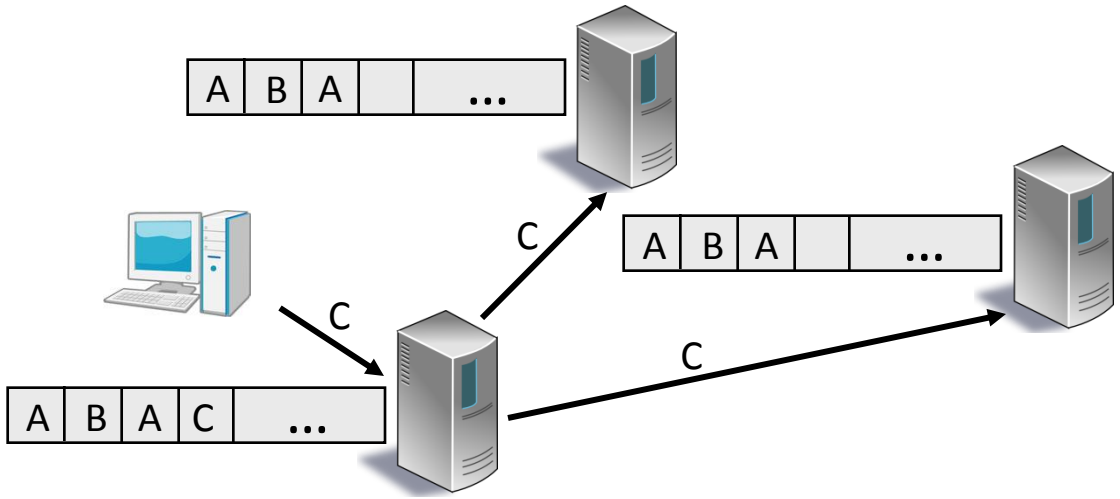
Problem with Primary Copy

- 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?

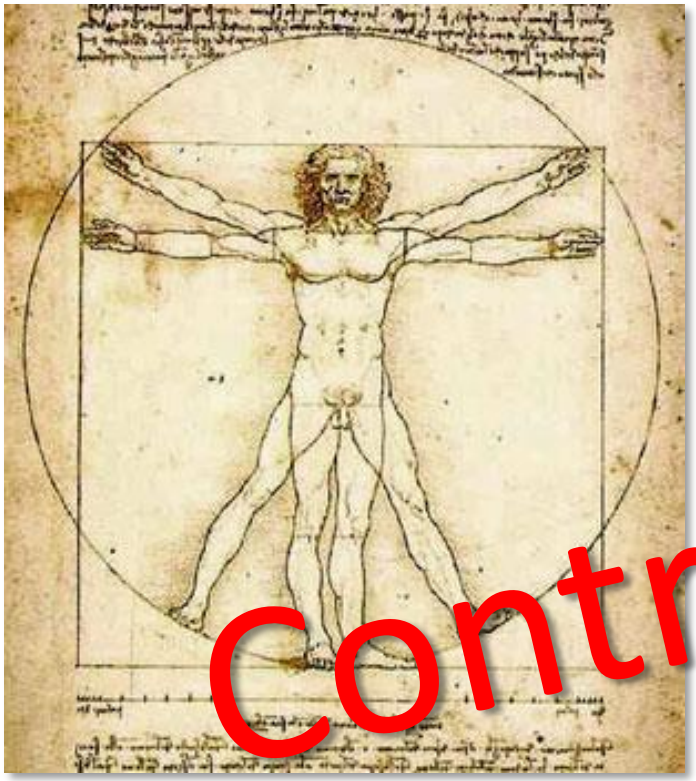
- The state of each server has to be updated in the same way
- This ensures that all servers are in the same state whenever all updates have been carried out!



- The servers have to agree on each update

  → Consensus has to be reached for each update!

Theory



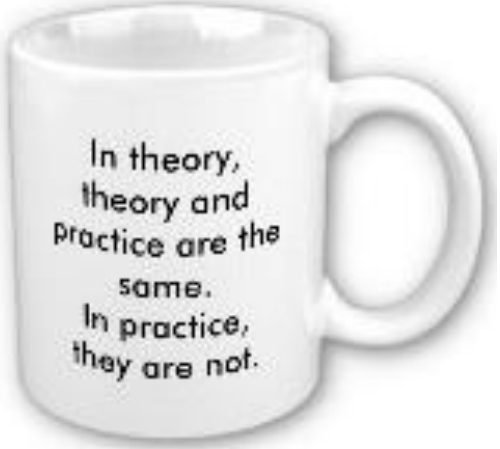
Impossible to guarantee consensus using a deterministic algorithm in asynchronous systems even if only one node is faulty

Practice

Consensus is required to guarantee consistency among different replicas

Contradiction?

From Theory to Practice



- So, how do we go from theory to practice…?

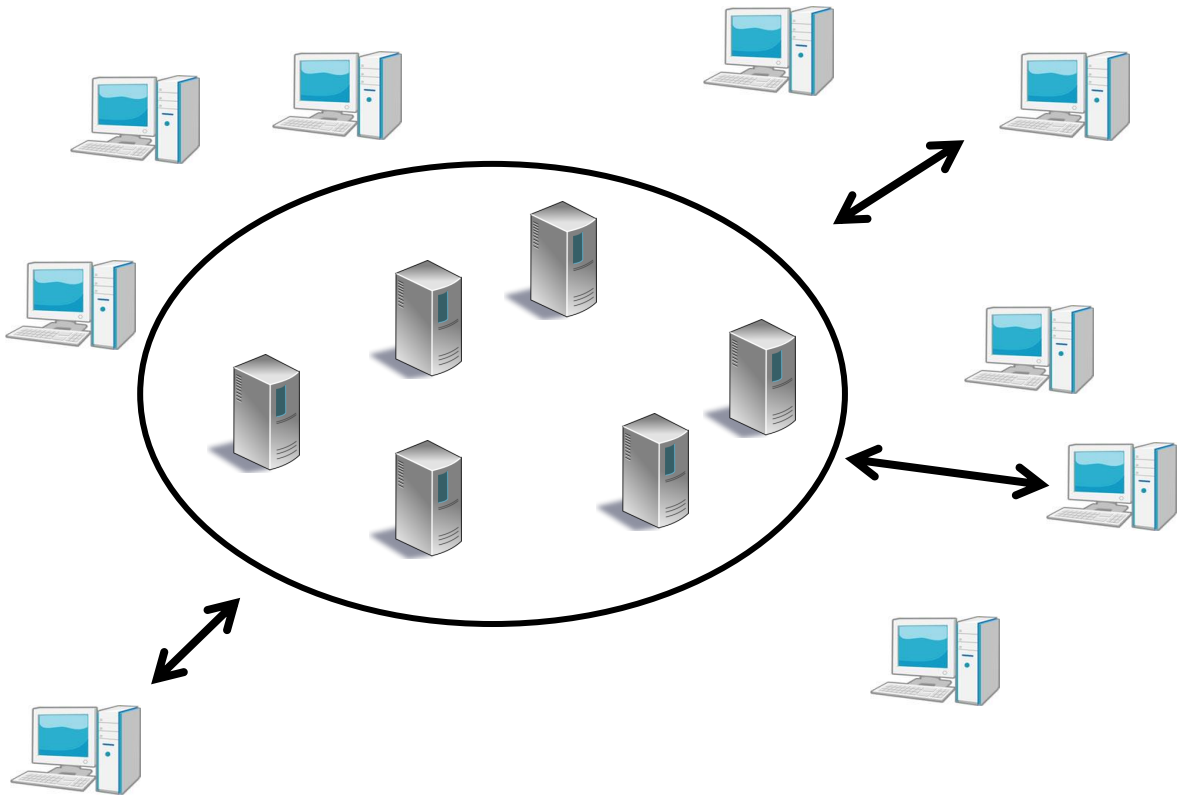
- Communication is often not synchronous, but not completely asynchronous either
  - There may be reasonable bounds on the message delays
  - Practical systems often use message passing. The machines wait for the response from another machine and abort/retry after time-out
  - Failures: It depends on the application/system what kind of failures have to be handled…

Depends on the bounds on the message delays!

- That is...
  - Real-world protocols also make assumptions about the system
  - These assumptions allow us to circumvent the lower bounds!

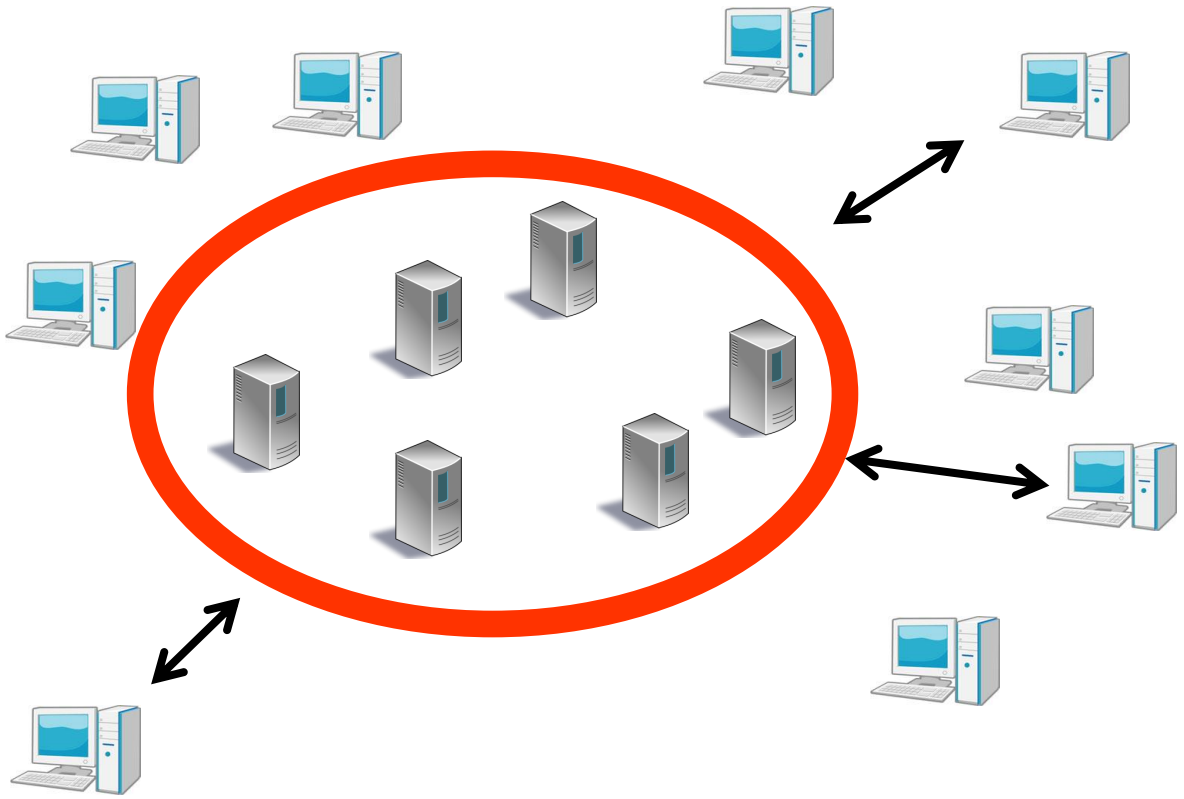
System

- Storage System
  - Servers: 2...Millions
  - Store data and react to client request
- Processes
  - Clients, often millions
  - Read and write/modify data



Consistency Models (Client View)

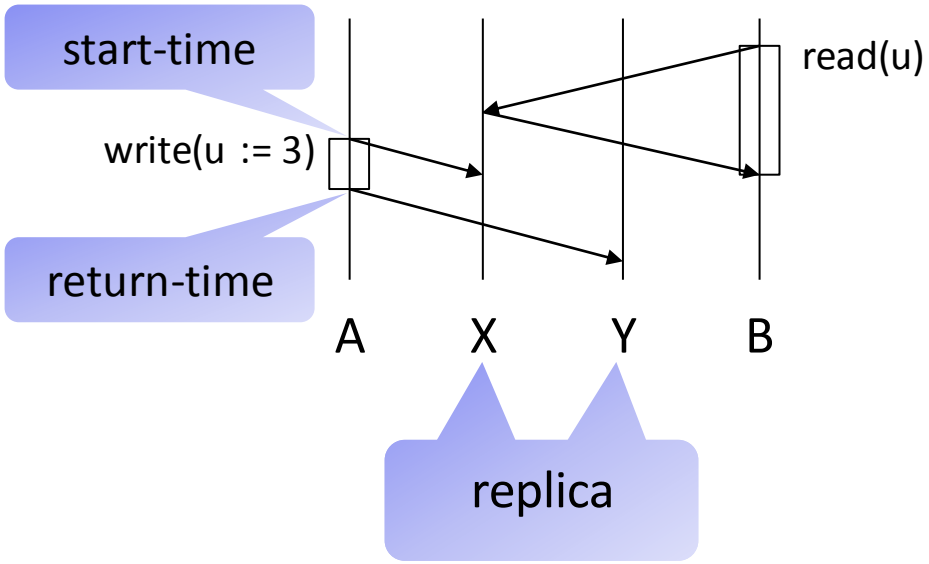
- Interface that describes the system behavior (abstract away implementation details)
- If clients read/write data, they expect the behavior to be the same as for a single storage cell.



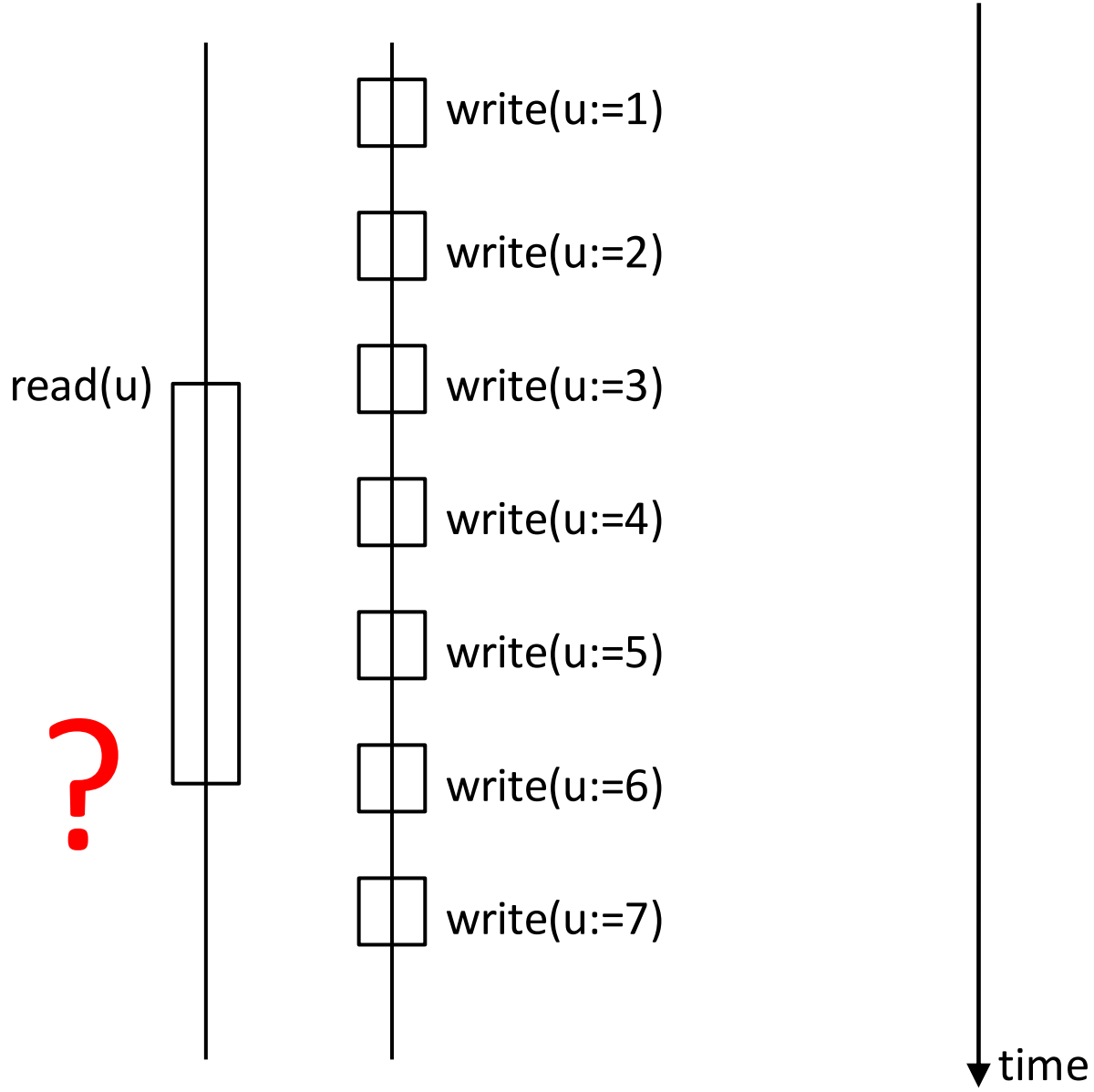
Let‘s Formalize these Ideas

- We have memory that supports 3 types of operations:
  - write(*u* := *v*): write value *v* to the memory location at address *u*
  - read(*u*): Read value stored at address *u* and return it
  - snapshot(): return a map that contains all address-value pairs

- Each operation has a start-time T_S and return-time T_R (time it returns to the invoking client). The duration is given by $T_R - T_S$.

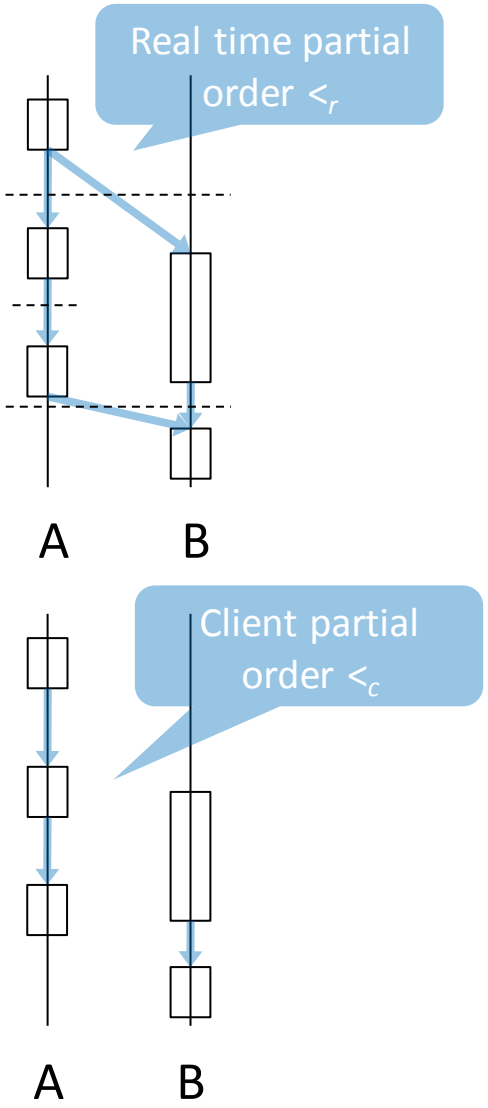


Motivation


write(u:=1)
write(u:=2)
write(u:=3)
write(u:=4)
write(u:=5)
write(u:=6)
write(u:=7)
read(u)
?
time

Executions

- We look at executions E that define the (partial) order in which processes invoke operations.
- Real-time partial order of an execution $<_r$:
  - $p <_r q$ means that duration of operation p occurs entirely before duration of q (i.e., p returns before the invocation of q in real time).
- Client partial order $<_c$:
  - $p <_c q$ means p and q occur at the same client, and that p returns before q is invoked.



Strong Consistency: Linearizability

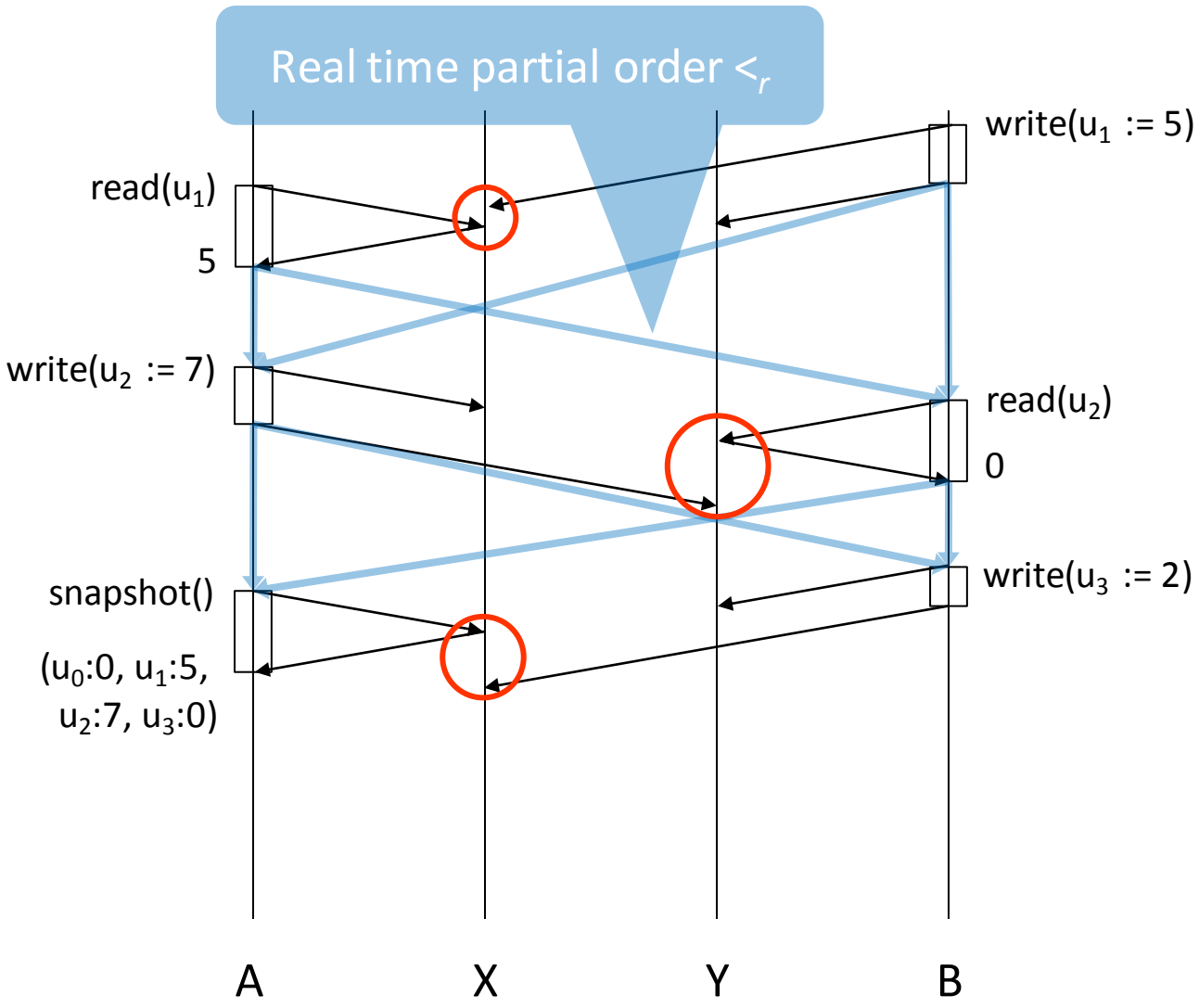
- A replicated system is called linearizable if it behaves exactly as a single-site (unreplicated) system.

Definition

Execution E is linearizable if there exists a sequence H such that:

1) H contains exactly the same operations as E, each paired with the return value received in E
2) The total order of operations in H is compatible with the real-time partial order $<_r$
3) H is a legal history of the data type that is replicated

Example: Linearizable Execution



Valid sequence H:

1.) write(u_1 := 5)
2.) read(u_1) → 5
3.) read(u_2) → 0
4.) write(u_2 := 7)
5.) snapshot() →
(u_0: 0, u_1: 5, u_2:7, u_3:0)
6.) write(u_3 := 2)

For this example, this is the only valid H. In general there might be several sequences H that fullfil all required properties.

Strong Consistency: Sequential Consistency

- Orders at different locations are disregarded if it cannot be determined by any observer within the system.

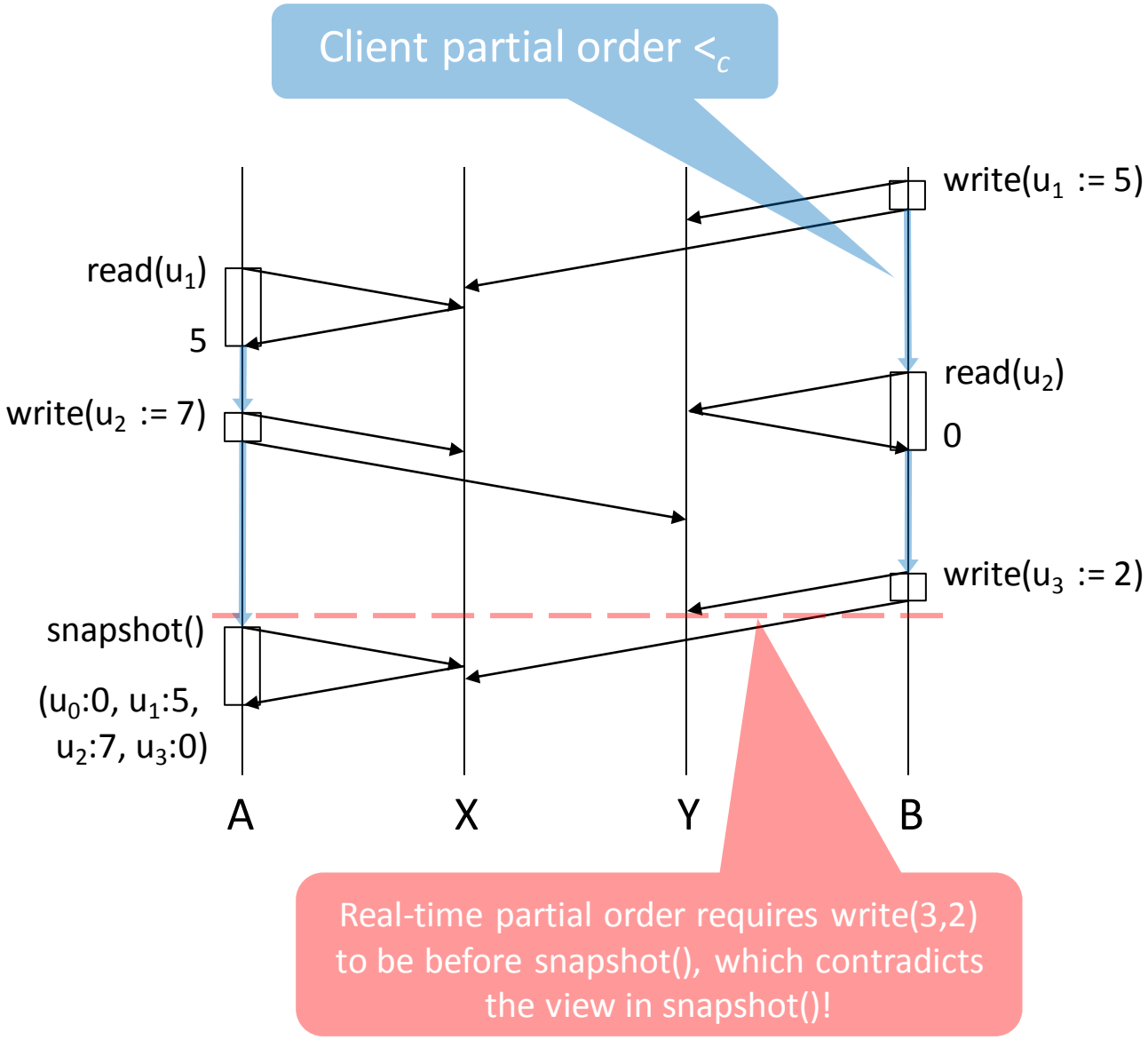
- I.e., a system provides sequential consistency if every node of the system sees the (write) operations on the same memory address in the same order, although the order may be different from the order as defined by real time (as seen by a hypothetical external observer or global clock).

Definition

Execution E is sequentially consistent if there exists a sequence H such that:

1) H contains exactly the same operations as E, each paired with the return value received in E
2) The total order of operations in H is compatible with the client partial order $<_c$
3) H is a legal history of the data type that is replicated

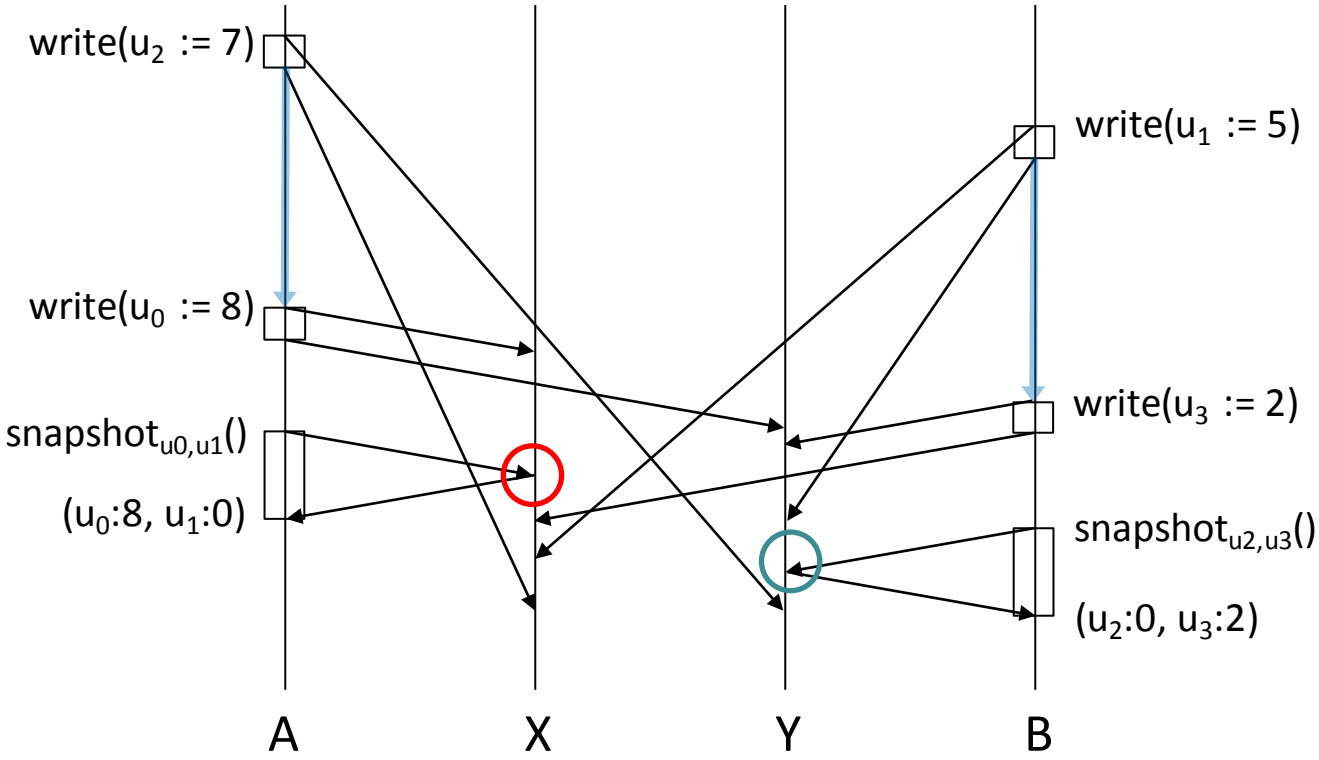
Example: Sequentially Consistent

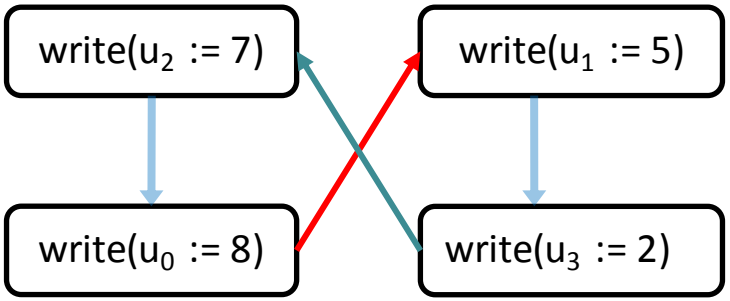


Valid sequence H:

1.) write(u_1 := 5)
2.) read(u_1) → 5
3.) read(u_2) → 0
4.) write(u_2 := 7)
5.) snapshot() →
(u_0:0, u_1:5, u_2:7, u_3:0)
6.) write(u_3 := 2)

Is Every Execution Sequentially Consistent?

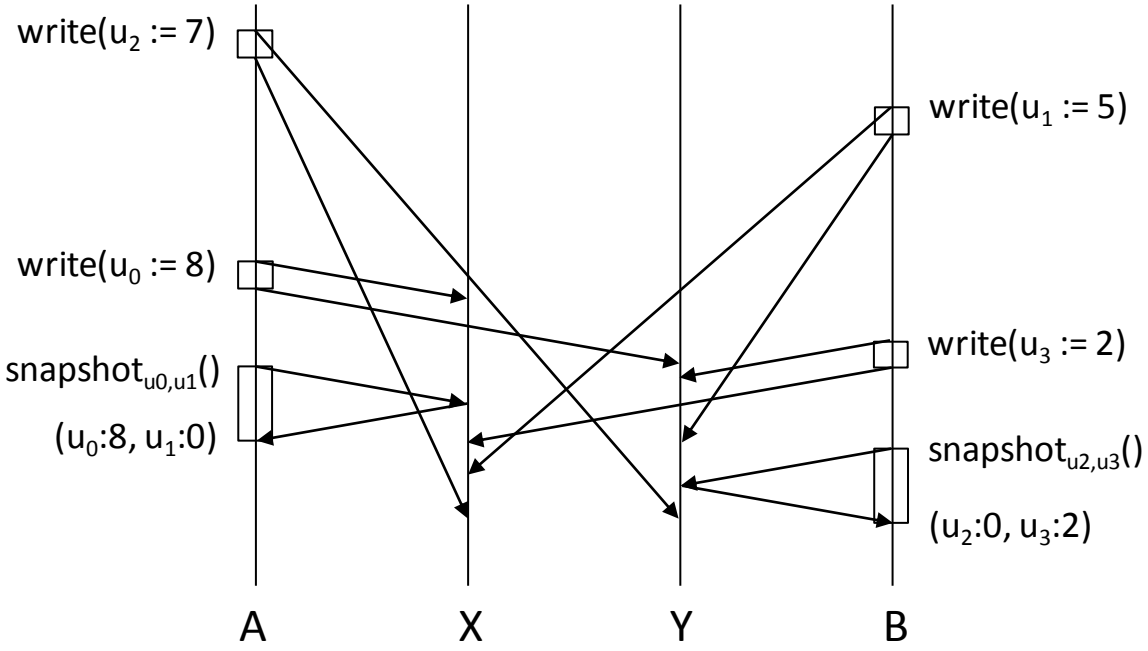




Circular dependencies!

I.e., there is no valid total order and thus above execution is not sequentially consistent

Sequential Consistency does not Compose



- If we only look at data items 0 and 1, operations are sequentially consistent
- If we only look at data items 2 and 3, operation are also sequentially consistent
- But, as we have seen before, the combination is not sequentially consistent

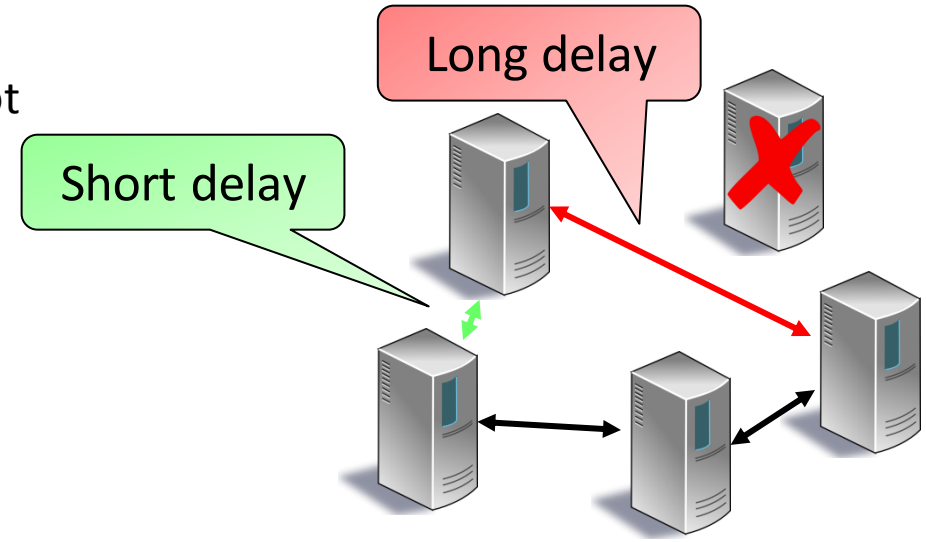
Sequential consistency does not compose!

(this is in contrast to linearizability)

Transactions

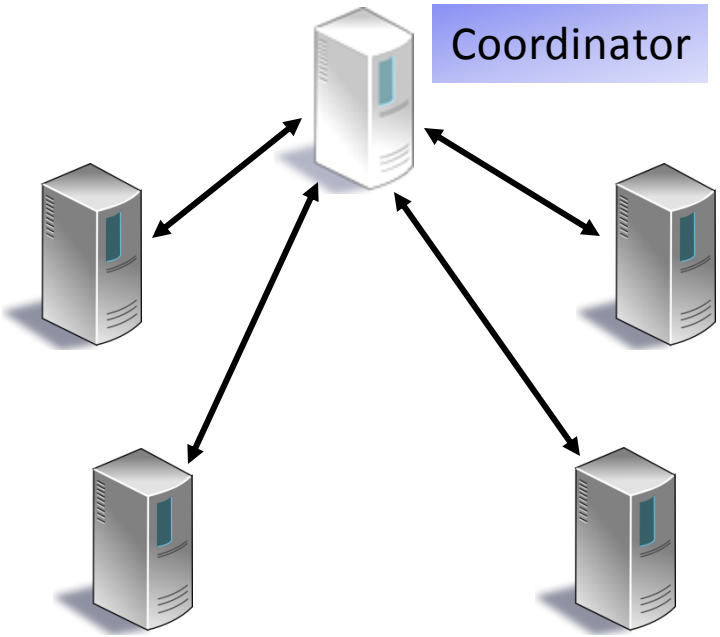
- In order to achieve consistency, updates have to be atomic
- A write has to be an atomic transaction
  - Updates are synchronized

- 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!



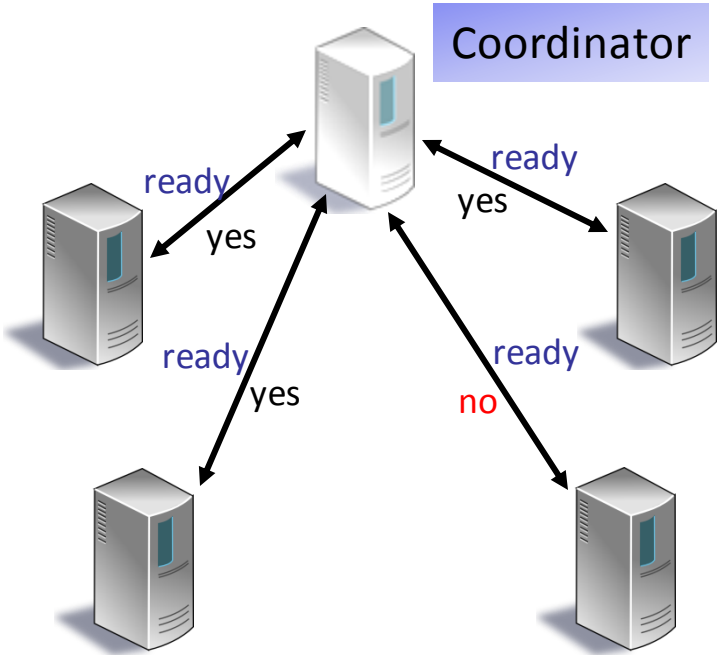
Two-Phase Commit (2PC)

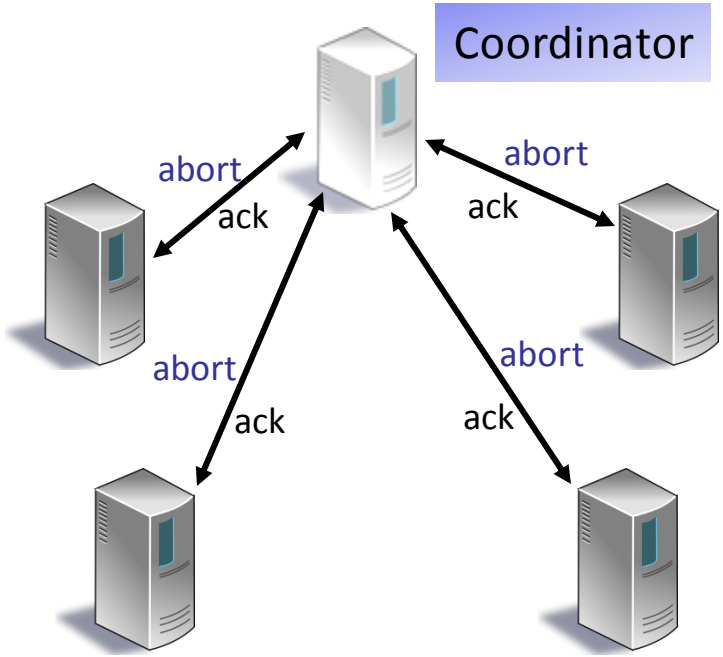
- A widely used protocol is the so-called two-phase commit protocol
- The idea is simple: There is a coordinator that coordinates the transaction
  - All other nodes communicate only with the coordinator
  - The coordinator communicates the final decision



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





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:

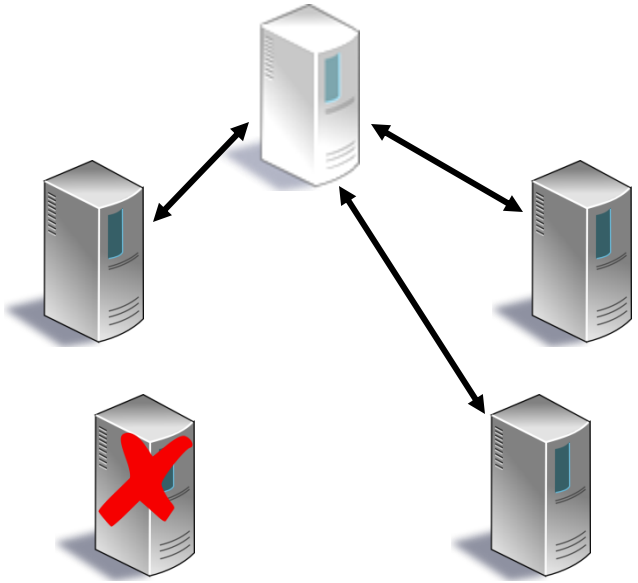
If the coordinator receives only *yes* messages:
    Send *commit* to all nodes
else
    Send *abort* to all nodes

If a node receives *commit* from the coordinator:
    Commit 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
- If a node that is not the coordinator fails, it still works
  - If the node fails before sending yes/no, the coordinator can either ignore it or safely abort the transaction
  - If the node fails before sending ack, the coordinator can still commit/abort depending on the vote in the first phase

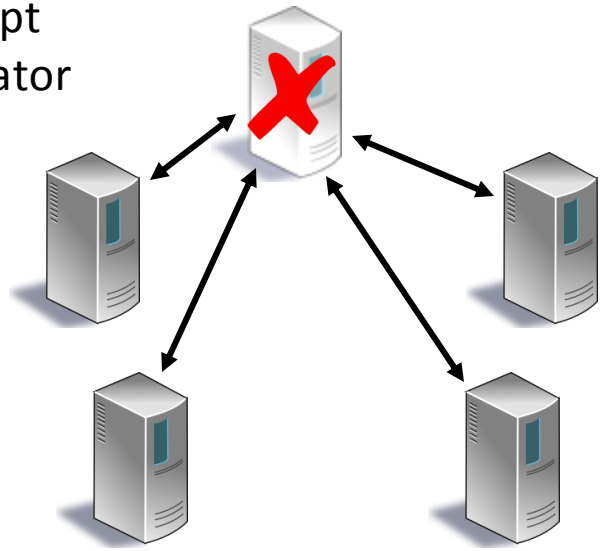


Two-Phase Commit: Analysis

- What happens if the coordinator fails?
- As we said before, this is (somehow) detected and a new coordinator takes over

This safety mechanism is not a part of 2PC…

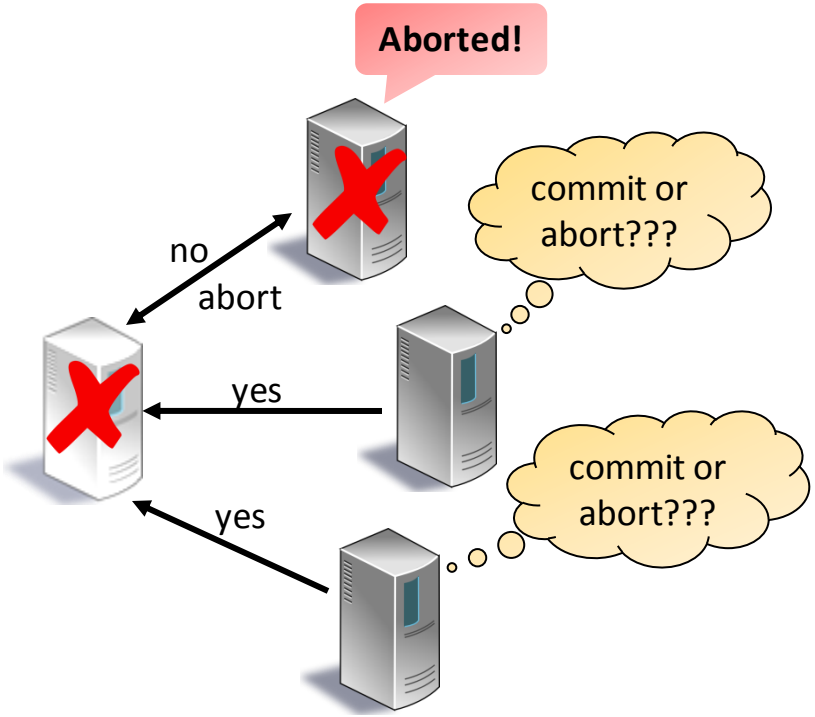
- How does the new coordinator proceed?
  - It must ask the other nodes if a node has already received a commit
  - A node that has received a commit replies yes, otherwise it sends no and promises not to accept a commit that may arrive from the old coordinator
  - If some node replied yes, the new coordinator broadcasts commit



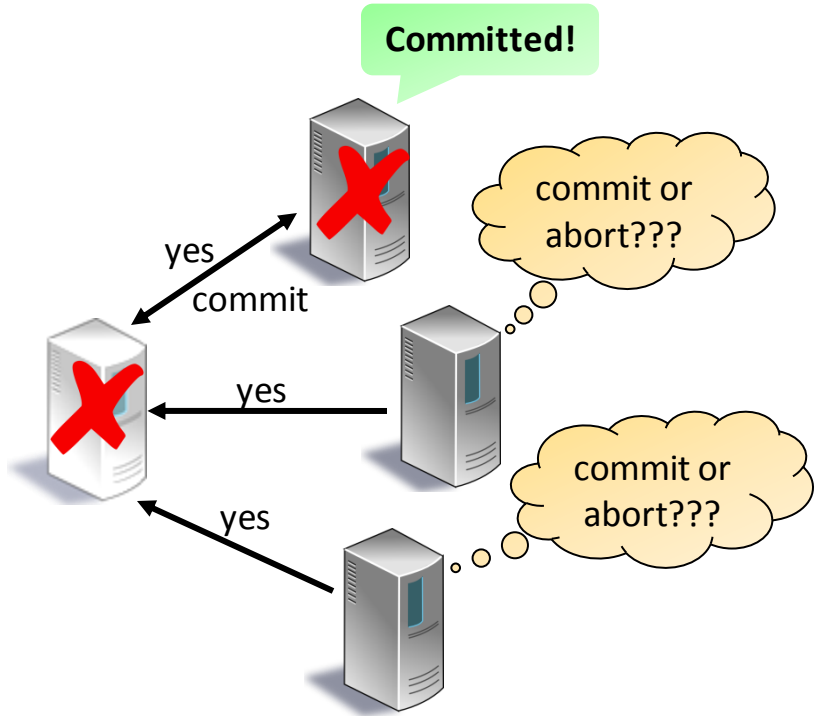
- This works if there is only one failure
- Does 2PC still work with multiple failures…?

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?



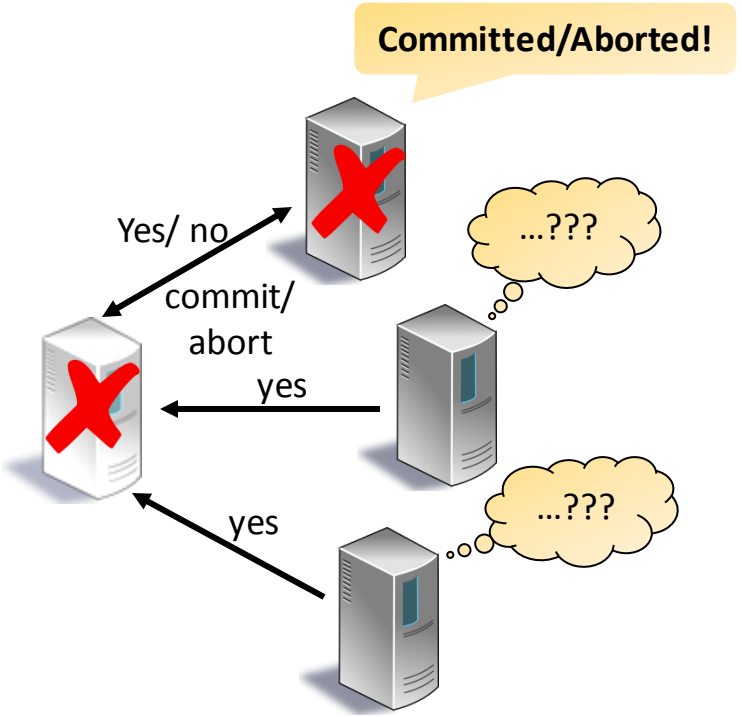
The nodes cannot commit!



The nodes cannot abort!

Two-Phase Commit: Multiple Failures

- What is the problem?
  - Some nodes may be ready to commit while others have already committed or aborted
  - If the coordinator crashes, the other nodes are not informed!
- How can we solve this problem?



The remaining nodes cannot make a decision!

Three-Phase Commit

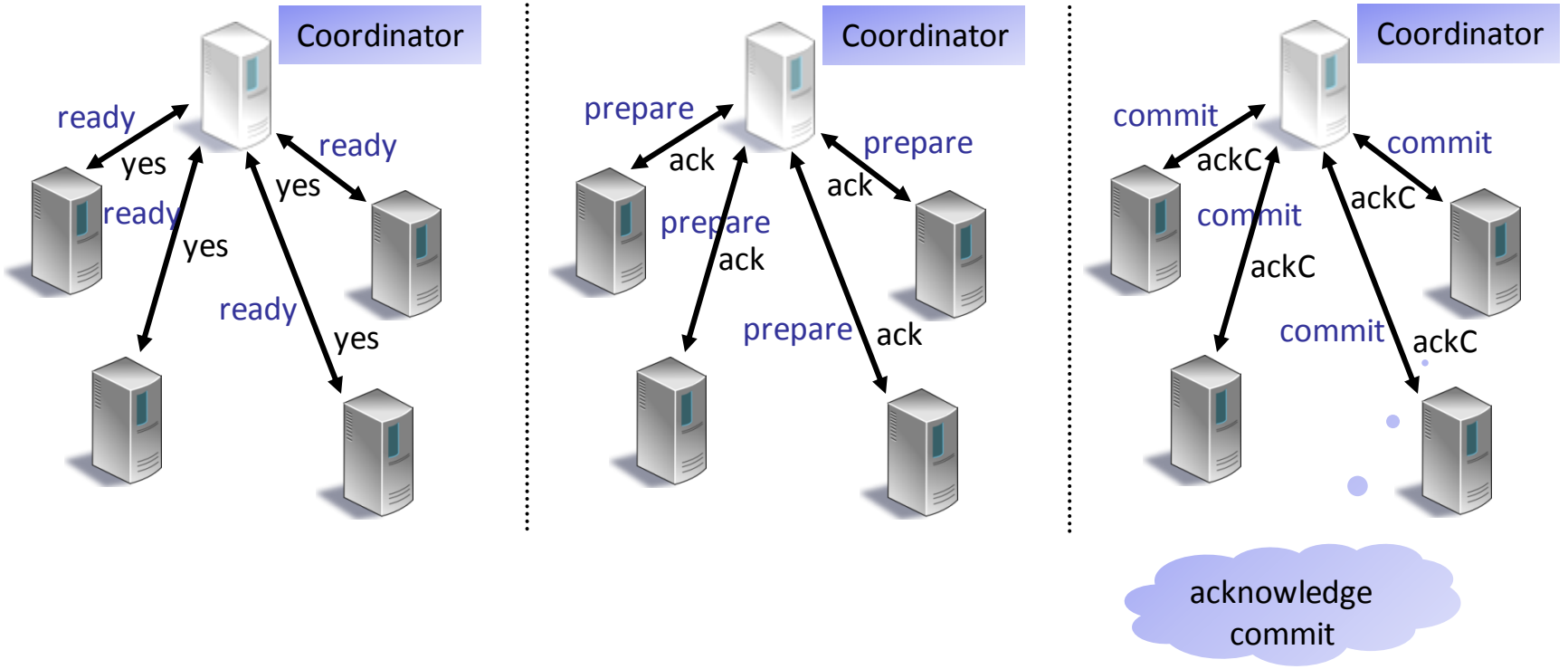
- Solution: Add another phase to the protocol!
  - The new phase precedes the commit phase
  - The goal is to inform all nodes that all are ready to commit (or not)
  - At the end of this phase, every node knows whether or not all nodes want to commit *before* any node has actually committed or aborted!

This solves the problem of 2PC!

- 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



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

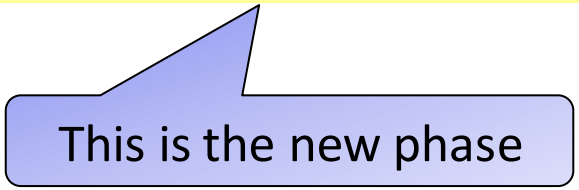
The first phase of 2PC and 3PC are identical!

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



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

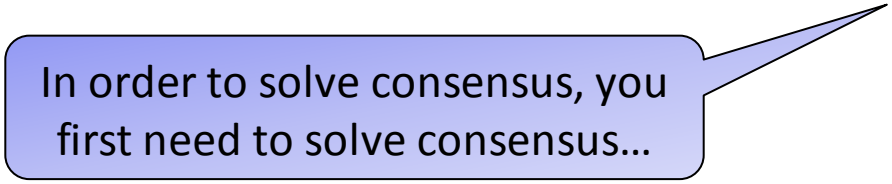
Once the coordinator received all *ackCommit* messages:
It completes the transaction by **committing** itself

Three-Phase Commit: Analysis

- All non-faulty nodes either commit or abort
  - If the coordinator doesn't fail, 3PC is correct because the coordinator lets all nodes either commit or abort
  - Termination can also be guaranteed: If some node fails before sending *yes*/*no*, the coordinator can safely abort. If some node fails after the coordinator sent *prepare*, the coordinator can still enforce a commit because all nodes must have sent *yes*
  - If only the coordinator fails, we again don't have a problem because the new coordinator can restart the protocol
  - Assume that the coordinator and some other nodes failed and that some node committed. The coordinator must have received *ack* messages from all nodes → All nodes must have received a *prepare* message. The new coordinator can thus enforce a commit. If a node aborted, no node can have received a *prepare* message. Thus, the new coordinator can safely abort the transaction

Three-Phase Commit: Analysis

- Although the 3PC protocol still works if multiple nodes fail, it still has severe shortcomings
  - 3PC still depends on a single coordinator. What if some but not all nodes assume that the coordinator failed?
    → The nodes first have to agree on whether the coordinator crashed or not!



  - Transient failures: What if a failed coordinator comes back to life? Suddenly, there is more than one coordinator!

- 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

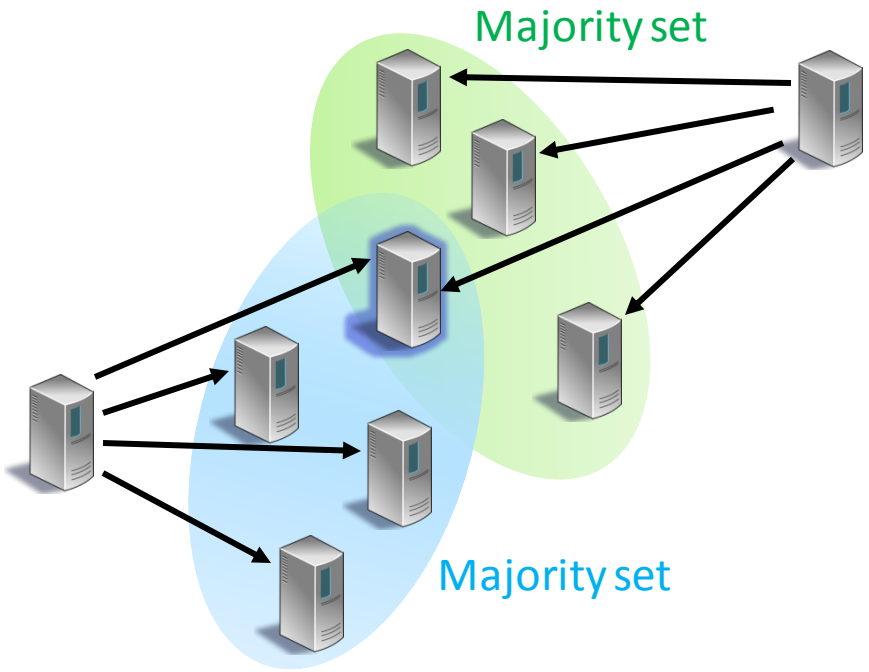
- Historical note
  - In the 1980s, a fault-tolerant distributed file system called “Echo” was built
  - According to the developers, it achieves “consensus” despite any number of failures as long as a majority of nodes is alive
  - The steps of the algorithm are simple if there are no failures and quite 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

fail-recover model

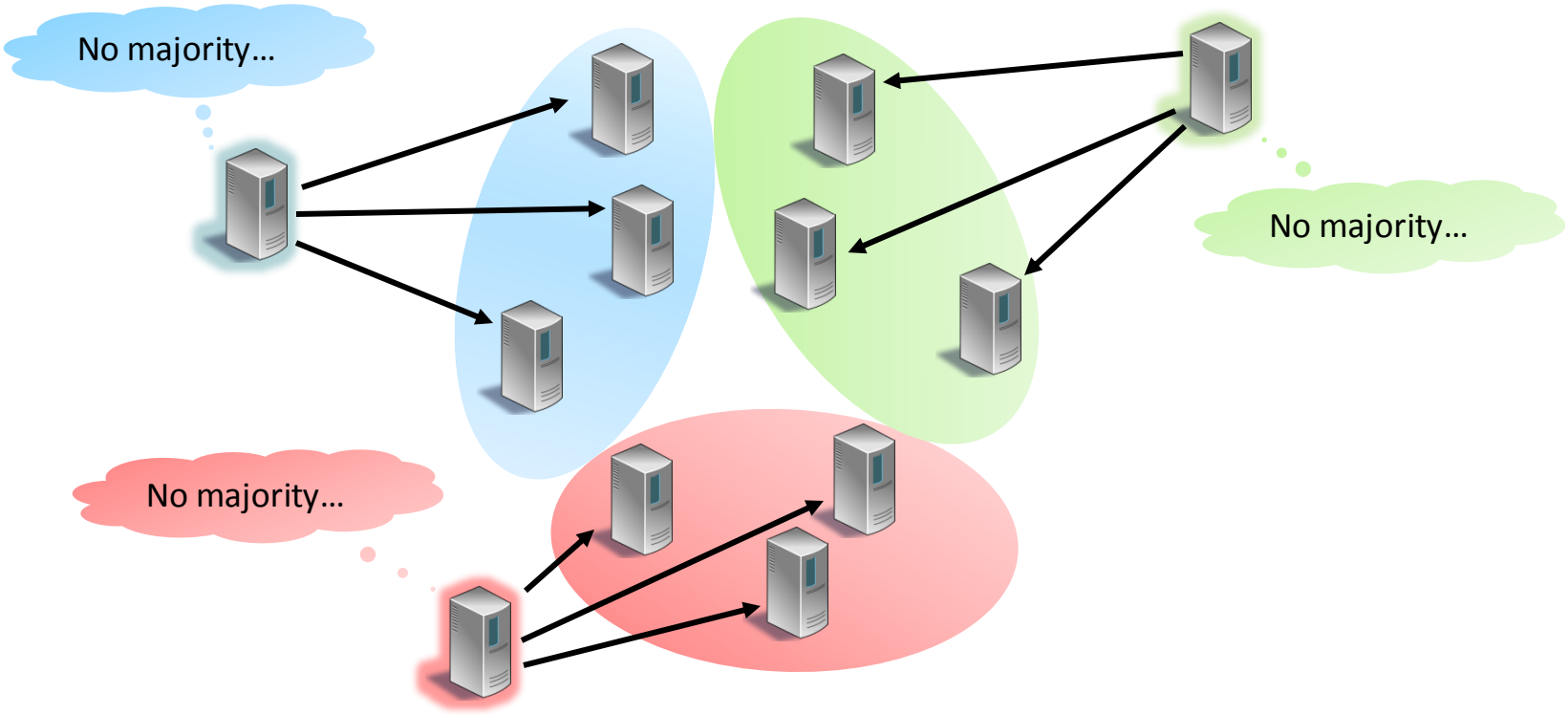
Paxos: Majority Sets

- 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: Roles

- Each node has one or more roles:

There are three roles

- Proposer
  - A proposer is a node that proposes a certain value for acceptance
  - Of course, 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

- A proposal (x,n) consists of the proposed value x and a proposal number n
- Whenever a proposer issues a new proposal, it chooses a larger (unique) proposal number
- An acceptor *accepts* a proposal (x,n) if n is larger than any proposal number it has ever heard

Give preference to larger proposal numbers!

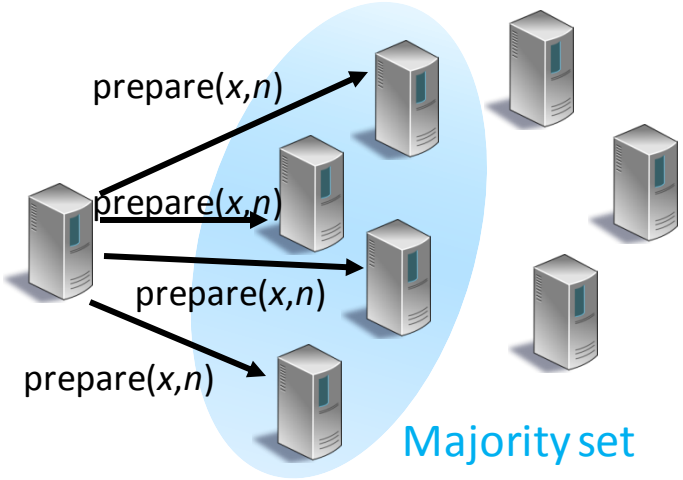
- An acceptor can *accept* any number of proposals
  - An accepted proposal may not necessarily be *chosen*
  - The value of a *chosen proposal* is the *chosen value*
- Any number of proposals can be *chosen*
  - However, if two proposals (x,n) and (y,m) are chosen, then $x = y$

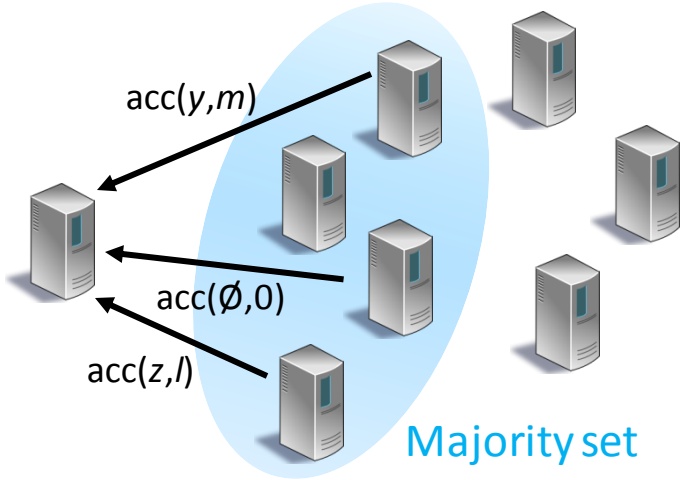
Consensus: Only one value can be chosen!

Paxos: Prepare

- Before a node sends propose(*x*,*n*), it sends 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 (Ø,0)
  - The proposer learns about accepted proposals!

Note that $m < n$!

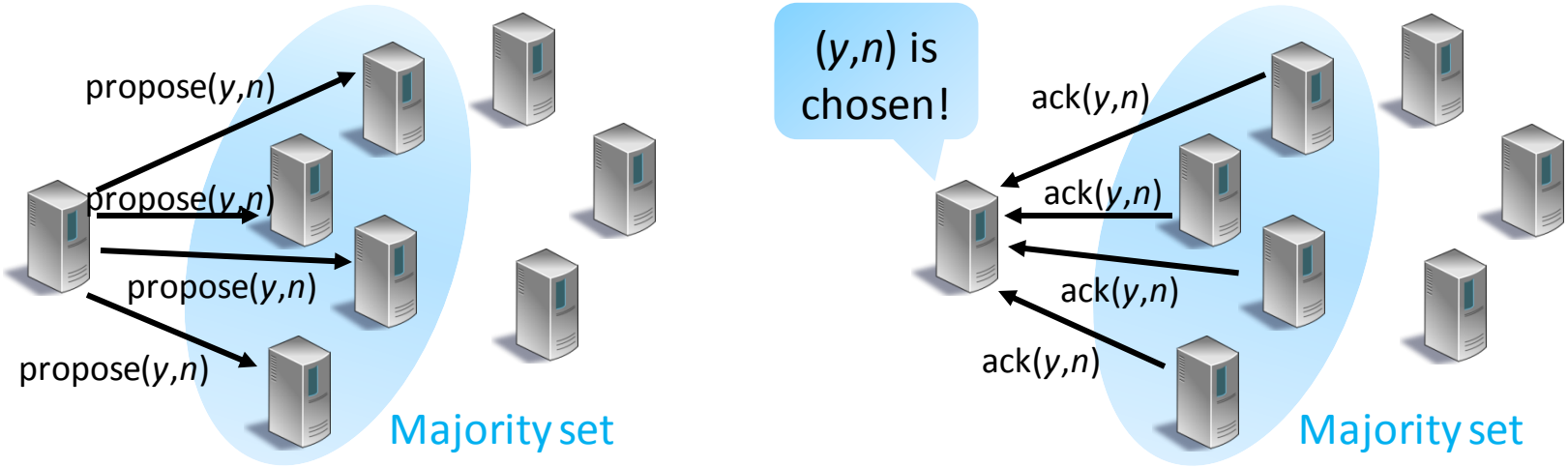




This is the first phase!

Paxos: Propose

- If the proposer receives all replies, it sends a proposal
- However, it only proposes its own value, if it only received acc(Ø,0), otherwise it 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 all acknowledgements ack(*y*,*n*), the proposal is *chosen*



This is the second phase!

Paxos: Algorithm of Proposer

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

Send 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 propose(*x*,*n*) to the same set of acceptors
    else
        Send propose(*y*,*n*) to the same set of acceptors

    if a majority of the nodes replies with ack(*x*,*n*) (or ack(*y*,*n*))
        The proposal is chosen!

The value of the proposal is also 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

$(x_{last}, n_{last}) := (\emptyset, 0)$ — Last accepted proposal

Acceptor receives prepare (x,n):

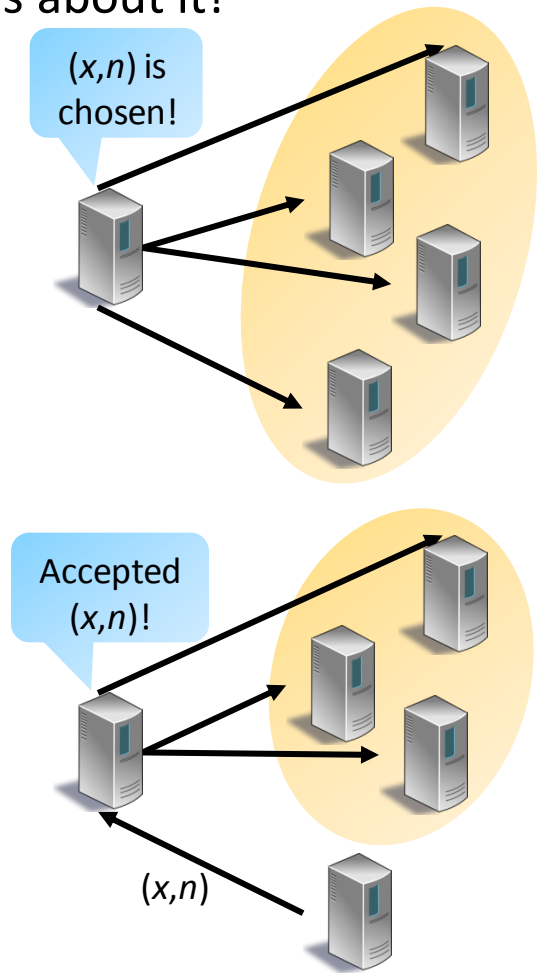
if $n > n_{max}$ then
    $n_{max} := n$
    Send acc(x_{last}, n_{last}) to the proposer

Acceptor receives proposal (x,n):

if $n = n_{max}$ then
    $x_{last} := x$
    $n_{last} := n$
    Send ack(x,n) to the proposer

Paxos: Spreading the Decision

- After a proposal is chosen, only the proposer knows about it!
- How do the other nodes get informed?
- The proposer could inform all nodes directly
  - Only *n*-1 messages are required
  - If the proposer fails, the others 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 → Since $n' > n$, the nodes in S must have received 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: Theorem

Theorem

If a value is chosen, all nodes choose this value

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…?

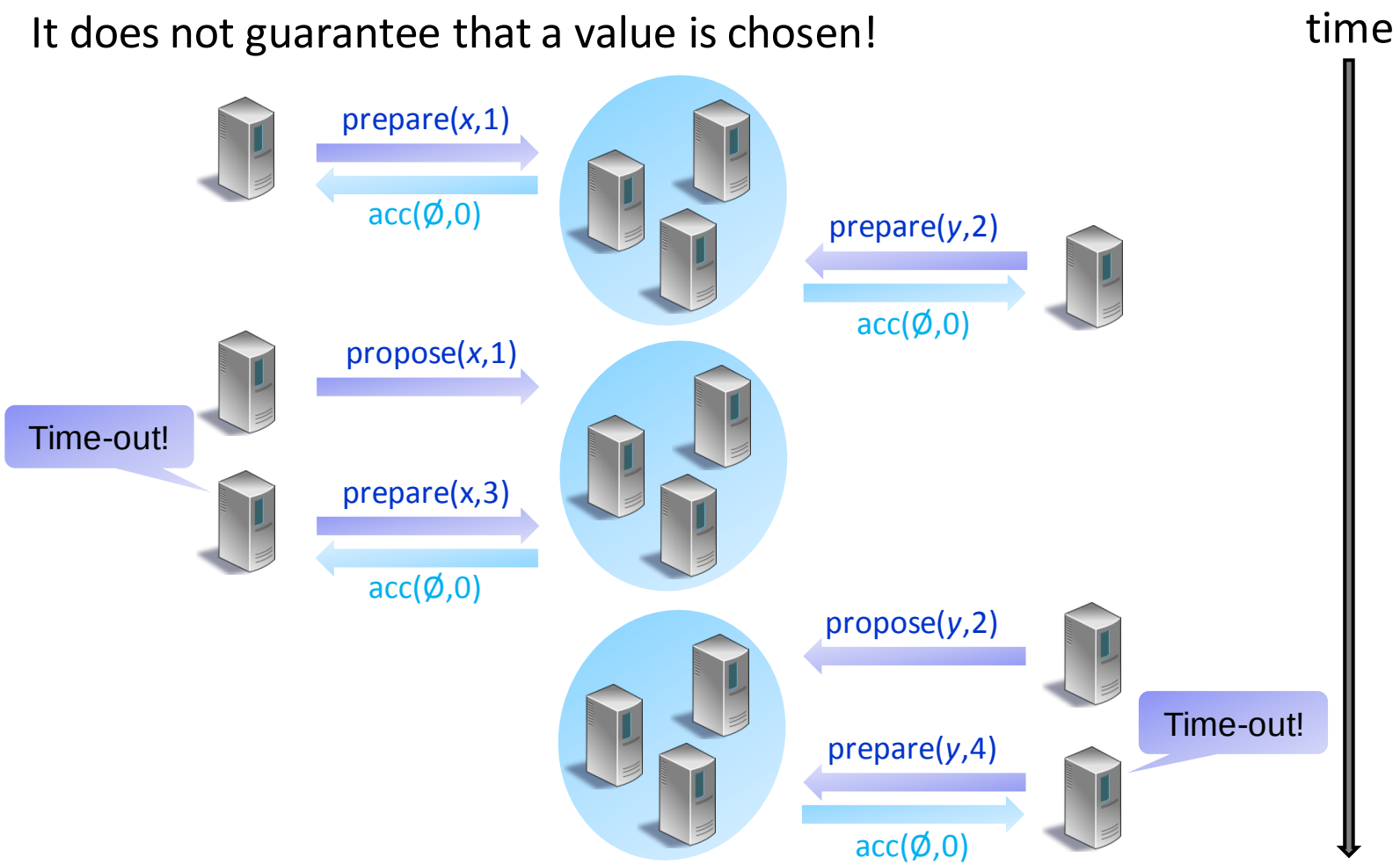
Theorem

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!

Quorum

Paxos used Majority sets: Can this be generalized?

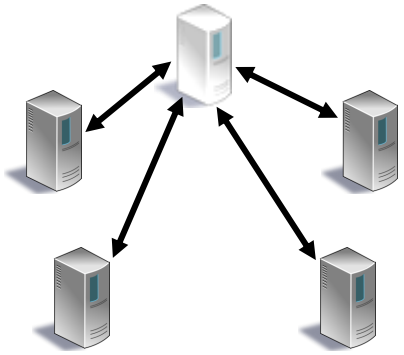
Yes: It's called Quorum

- In law, a quorum is the minimum number of members of a deliberative body necessary to conduct the business of the group.
- In our case: substitute "the minimum number of members of a deliberative body" with "any subset of servers of a distributed system"

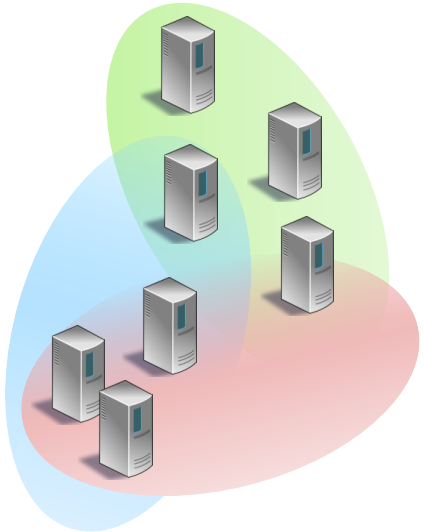
A Quorum does not automatically need to be a majority.

What else can you imagine? What are reasonable objectives?

Quorum: Primary Copy vs. Majority



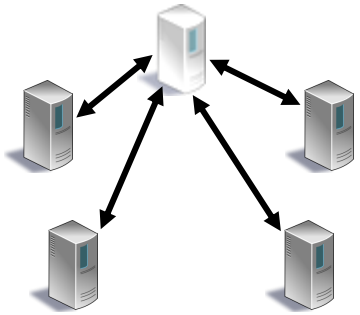
or

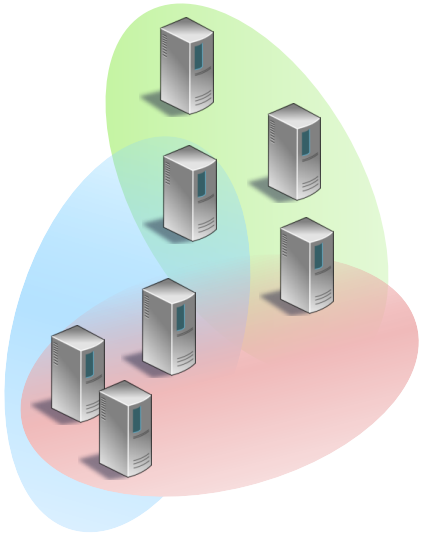


?

Quorum: Primary Copy vs. Majority

	Singleton	Majority
How many servers need to be contacted? (Work)	1	$> n/2$
What's the load of the busiest server? (Load)	100%	$\approx 50\%$
How many server failures can be tolerated? (Resilience)	0	$< n/2$





Definition: Quorum System

Definition

Quorum System

Let $P = \{P_1, \ldots, P_n\}$ be a set of servers.
A quorum system $\mathrm{Q} \subset 2^P$ is a set of subsets of P such that every two subsets intersect. Each $\mathrm{Q} \in \mathrm{Q}$ is called a quorum.

Definition

Minimal Quorum System

A quorum system Q is called minimal if $\forall\, Q, Q' \in \mathrm{Q}: Q \not\subset Q'$

Definition: Load

Definition

Access Strategy

An access strategy W is a random variable on a quorum system $\mathbb{Q}$, i.e. $\sum_{Q \in \mathbb{Q}} P_W(Q) = 1$

Definition

Load

The load induced by access strategy W on a server P_i is:

$$l_W(i) = \sum_{Q \in \mathbb{Q};\ P_i \in Q} P_W(Q)$$

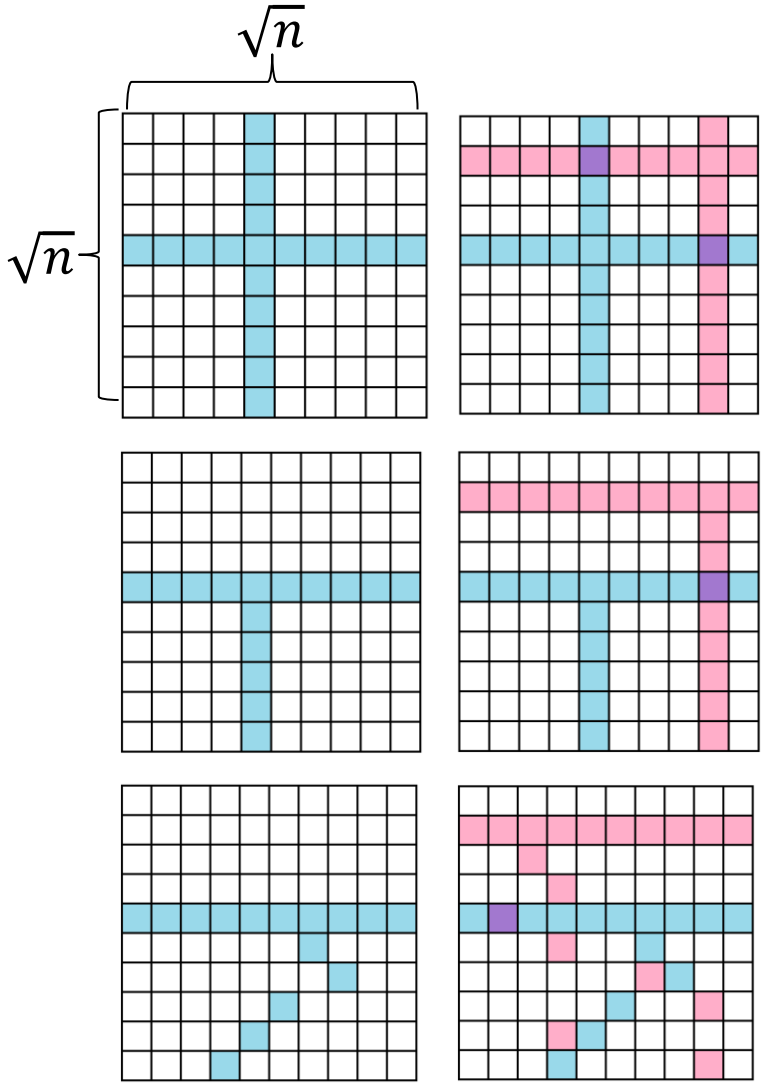
The load induced by W on a quorum system $\mathbb{Q}$ is the maximal load induced by W on any server in $\mathbb{Q}$.

$$L_W(\mathbb{Q}) = \max_{\forall P_i} l_W(i)$$

The system load of $\mathbb{Q}$ is

$$L(\mathbb{Q}) = \min_{\forall W} L_W(\mathbb{Q})$$

Quorum: Grid



- Work: $2\sqrt{n} - 1$
- Load: $\frac{2\sqrt{n}-1}{n}$

Definitions: Fault Tolerance

Definition

Resilience

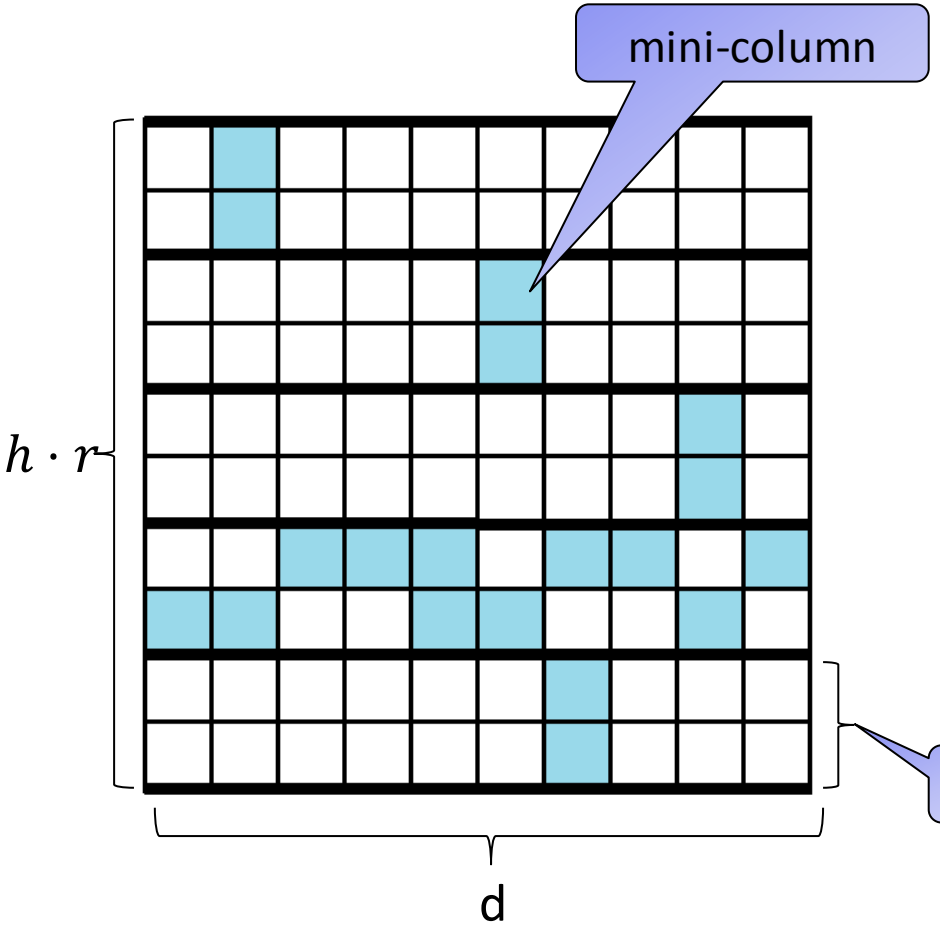
The resilience $R(\mathbb{Q})$ of a quorum system is the largest f such that for all sets $F \subset P, |F| = f$, there is at least one quorum $Q \in \mathbb{Q}$ with $F \cap Q = \emptyset$

Definition

Failure Probability

Assume that each server fails independently with probability p. The failure probability of a quorum system $\mathbb{Q}$ is the probability that no quorum $Q \in \mathbb{Q}$ is available.

Quorum: B-Grid



- Suppose $n = dhr$ and arrange the elements in a grid with d columns and $h \cdot r$ rows. Call every group of r rows a band and call r elements in a column restricted to a band a mini-column. A quorum consists of one mini-column in every band and one element from each mini-column of one band; thus, every quorum has $d + hr - 1$ elements
- Resilience?

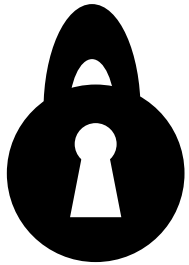
Quorum Systems: Overview

	Singleton	Majority	Grid	B-Grid**
Work	1	$> n/2$	$\theta(\sqrt{n})$	$\theta(\sqrt{n})$
Load	1	$1/2$	$\theta(1/\sqrt{n})$	$\theta(1/\sqrt{n})$
Resilience	0	$< n/2$	$\sqrt{n} - 1$	$\theta(\sqrt{n})$
Failure Prob.*	p	$\to 0$	$\to 1$	$\to 0$

*Assuming *p* constant but significantly less than ½.

**B-Grid: We set $d = \sqrt{n}, r = \log n$

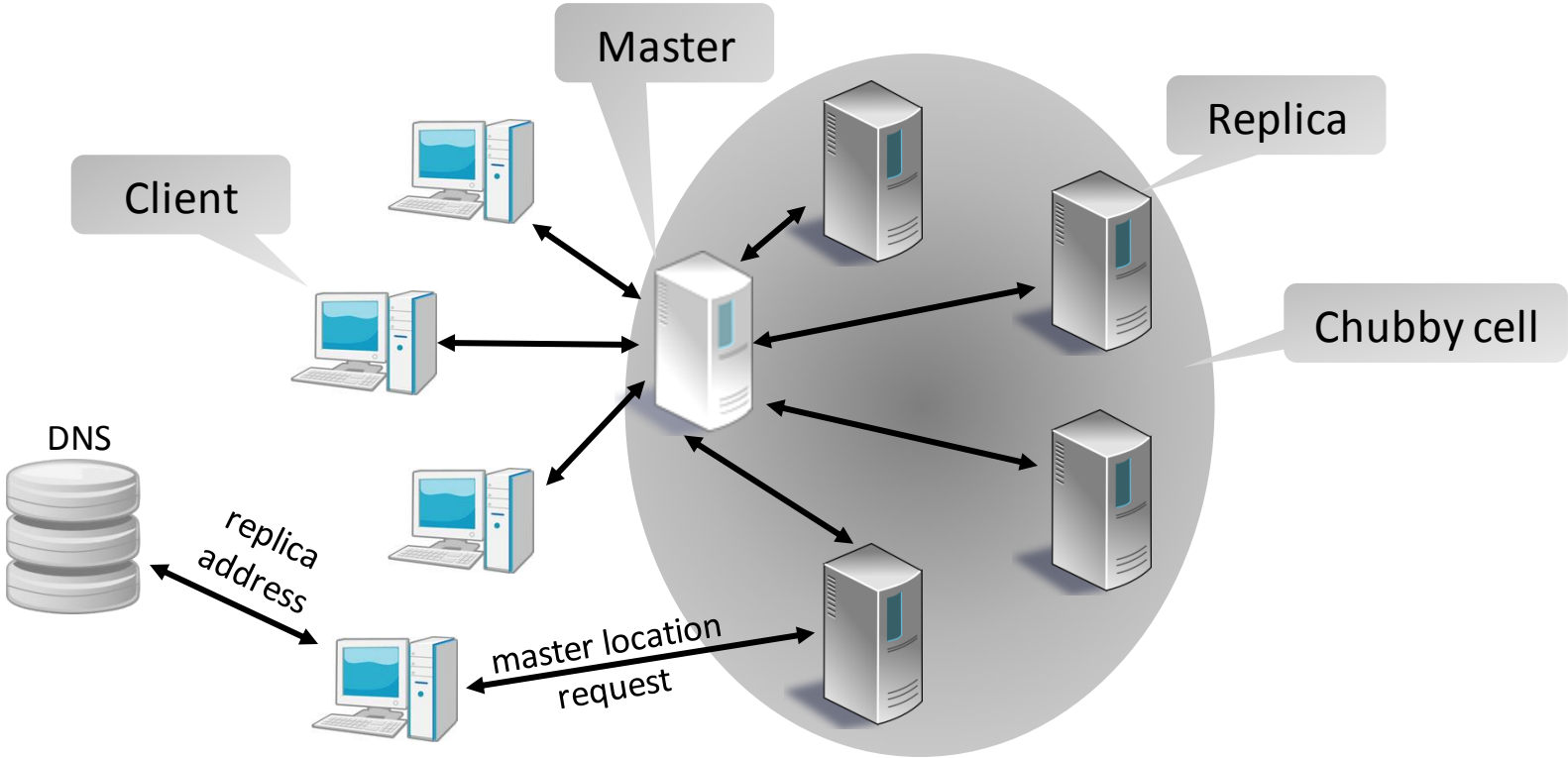
Chubby



- 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)

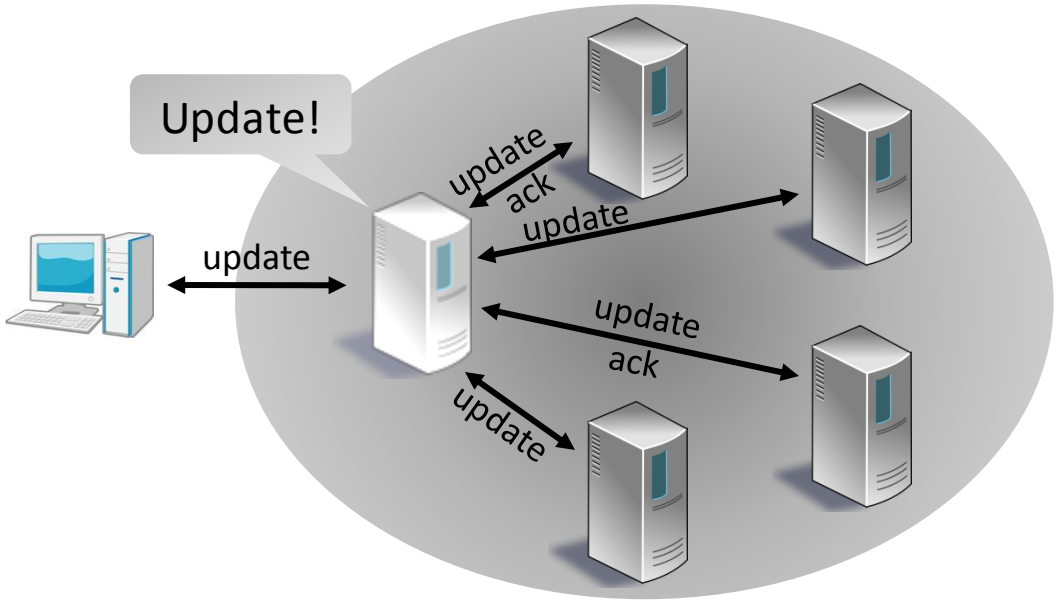
Chubby: System Structure

- 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

- 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

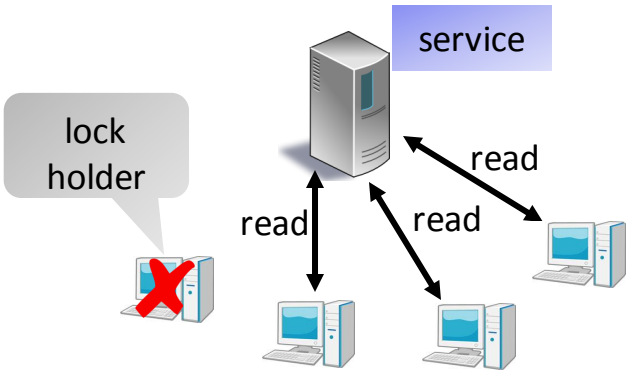
- The master remains the master for the duration of the master lease
  - Before the lease expires, the master can renew it (and remain the master)
  - It is guaranteed that no new master is elected before the lease expires
  - However, a new master is elected as soon as the lease expires
  - This ensures 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?
- They run (a variant of) the Paxos algorithm!

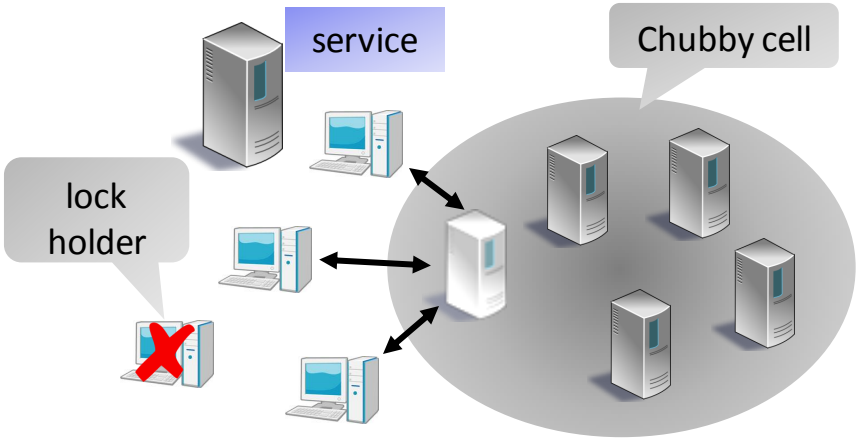
Chubby: Locks

- Locks are advisory (not mandatory)
  - As usual, locks are mutually exclusive
  - However, data can be *read* without the lock!
  - Advisory locks are more efficient than mandatory locks (where any access requires the lock): Most accesses are reads! If a mandatory lock is used and the lock holder crashes, then all reads are stalled until the situation is resolved
  - Write permission to a resource is required to obtain a lock

Advisory:

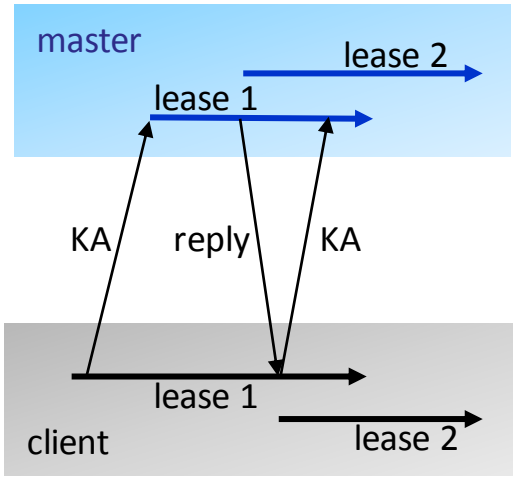


Mandatory:



Chubby: Sessions

- What happens if the lock holder crashes?
- Client initially contacts master to establish a session
  - Session: Relationship between Chubby cell and Chubby client
- Each session has an associated lease
  - The master can extend the lease, but it may not revoke the lease
  - Longer lease times if the load is high
- Periodic KeepAlive (KA) handshake to maintain relationship
  - 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
  - The client terminates the session explicitly
  - or the lease expires



Chubby: Lease Timeout

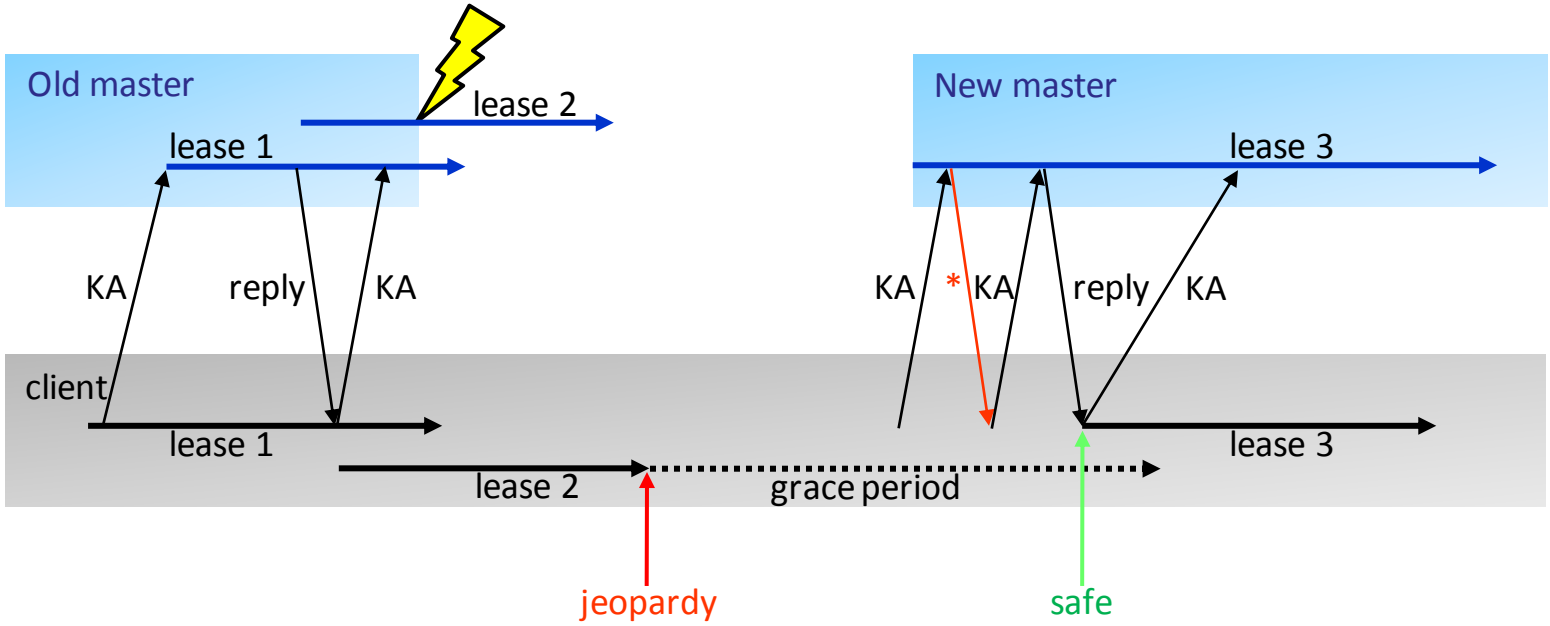
- The client maintains a local lease timeout
  - The client knows (roughly) when it has to hear from the master again
- If the local lease expires, the session is in jeopardy
- As soon as a session is in jeopardy, the grace period (45s by default) starts
  - If there is a successful KeepAlive exchange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

Time when lease expires

- This might happen if the master crashed…

Chubby: Master Failure

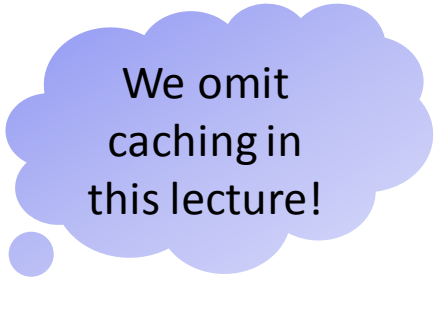
- The grace period can save sessions



- 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

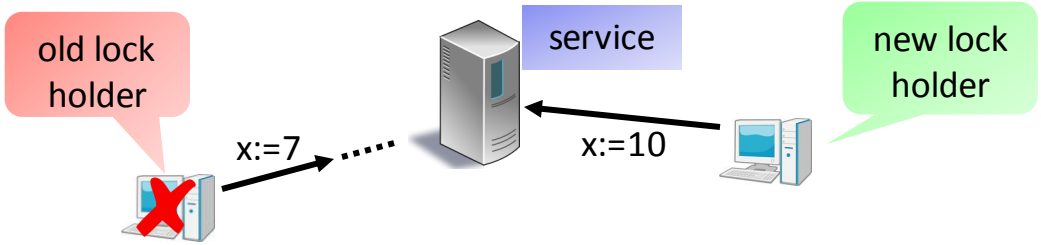
Chubby: Master Failure

- 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



Chubby: Locks Reloaded

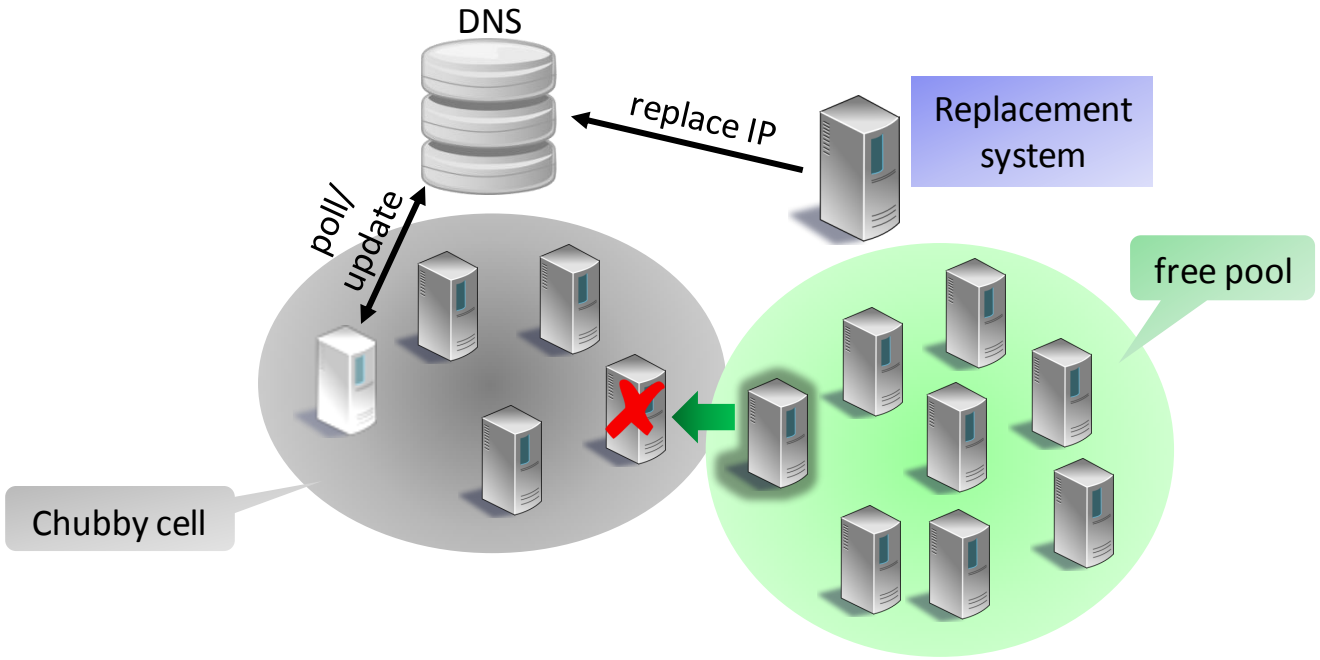
- What if a lock holder crashes and its (write) request is still in transit?
  - This write may undo an operation of the next lock holder!



- 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

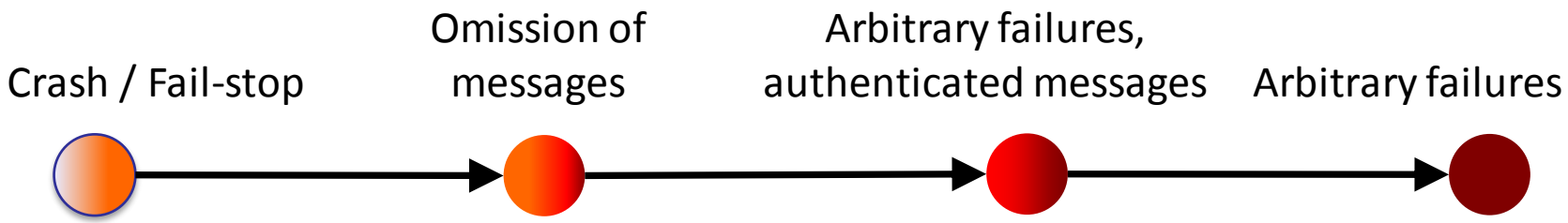


Chubby: Performance

- 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:



Practical Byzantine Fault-Tolerance

- Is it reasonable to consider Byzantine behavior in practical systems?
- There are several reasons why clients/servers may behave "arbitrarily"
  - Malfunctioning hardware
  - Buggy software
  - Malicious attacks
- Can we have a practical and efficient system that tolerates Byzantine behavior…?
  - We again need to solve consensus…

PBFT

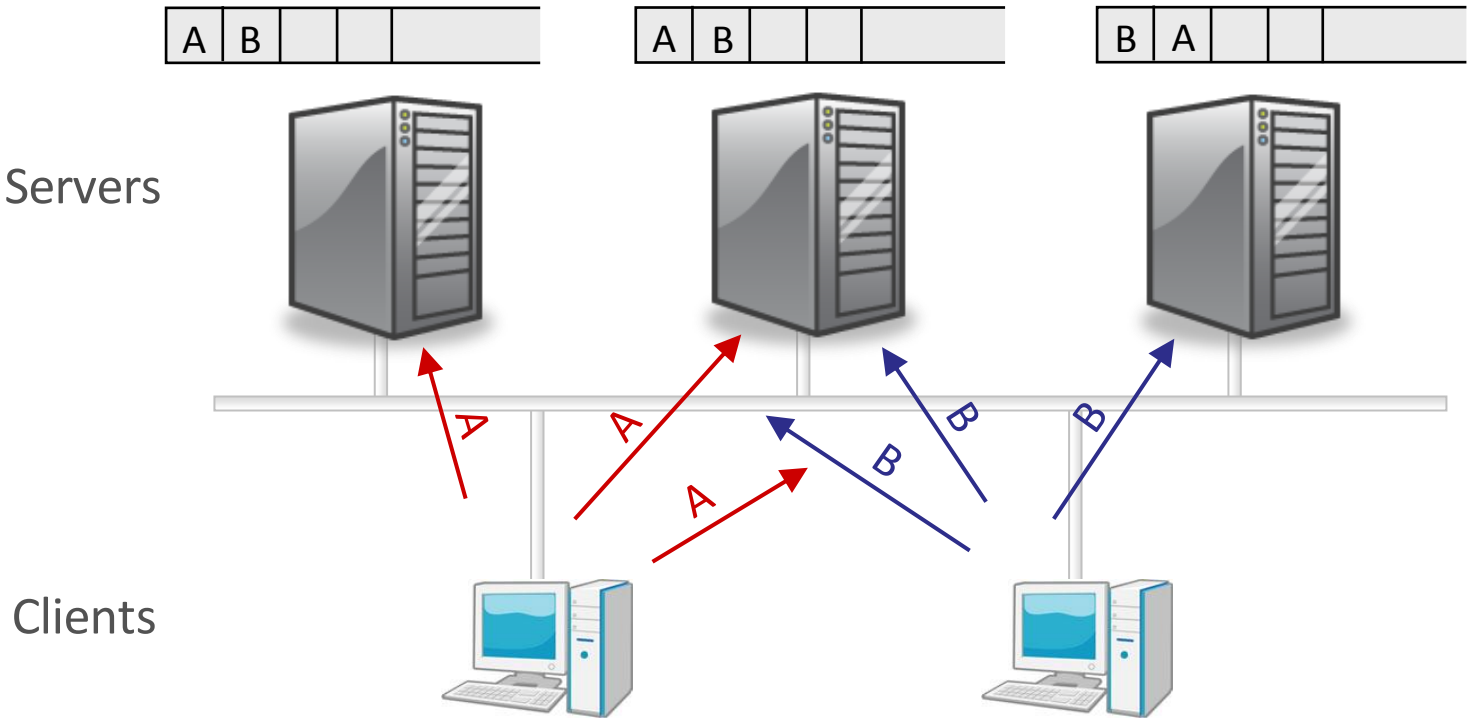
- We are now going to study the Practical Byzantine Fault-Tolerant (PBFT) system
- The system consists of clients that read/write data stored at *n* servers

- Goal
  - The system can be used to implement any deterministic replicated service with a *state* and some *operations*
  - Provide 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

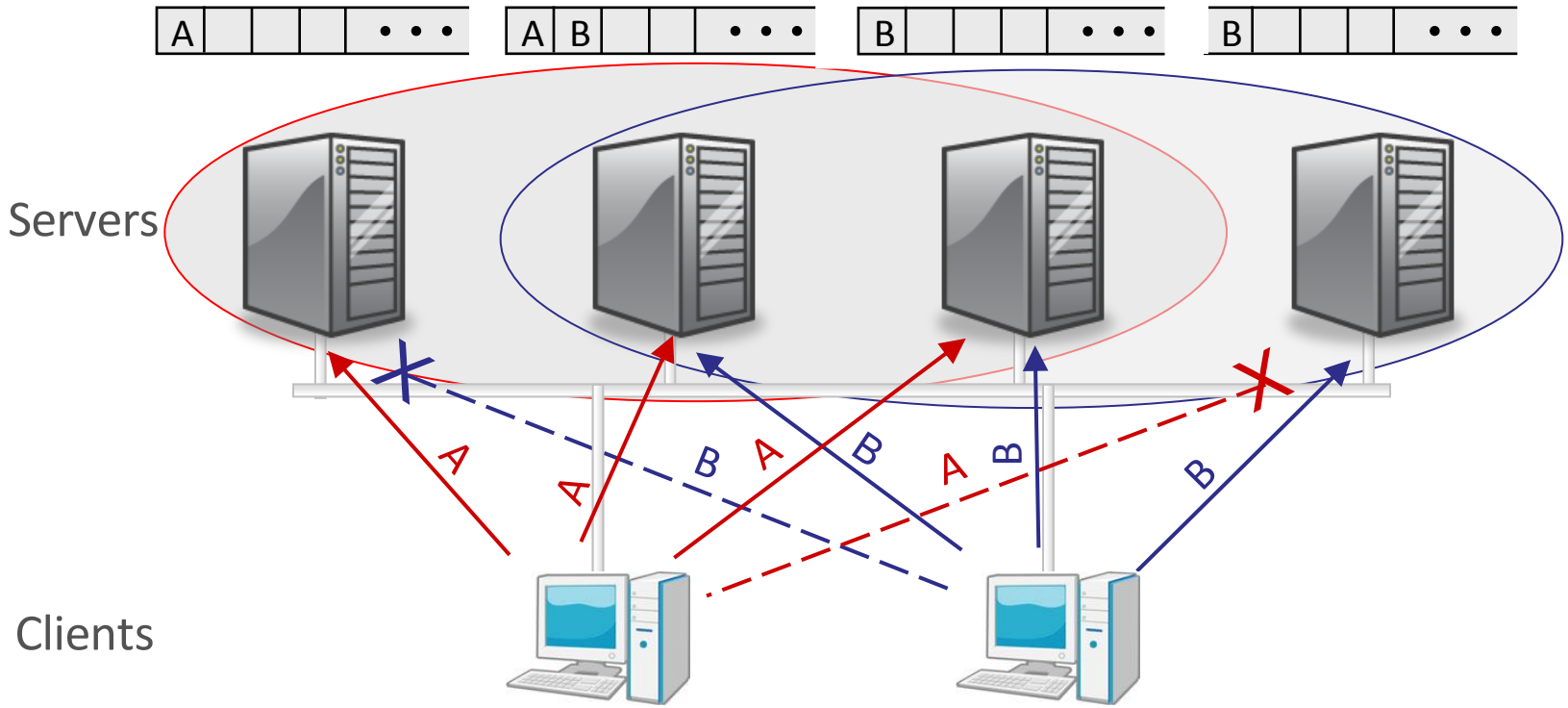
PBFT: Order of Operations

- State replication (repetition): If all servers start in the same state, all operations are deterministic, and all operations are executed in the same order, then all servers remain in the same state!
- 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

PBFT is not as decentralized as Paxos!

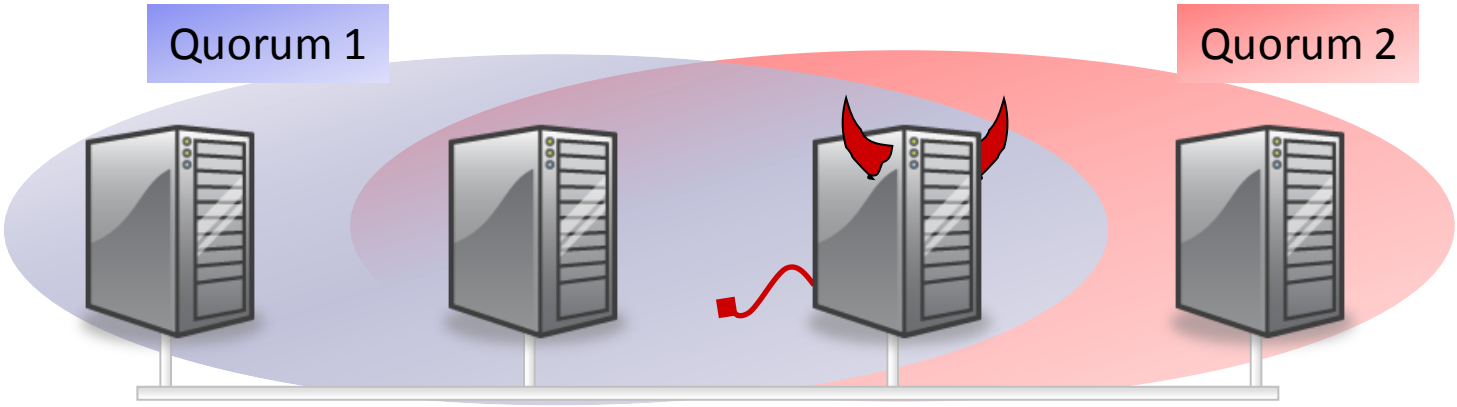
Quorum…?

- Note that we assume in the following that there are *exactly* $n = 3f+1$ servers!

Byzantine Quorums

Now, a quorum is any subset of the servers of size at least $2f+1$

- The intersection between any two quorums contains at least one correct (not Byzantine) server



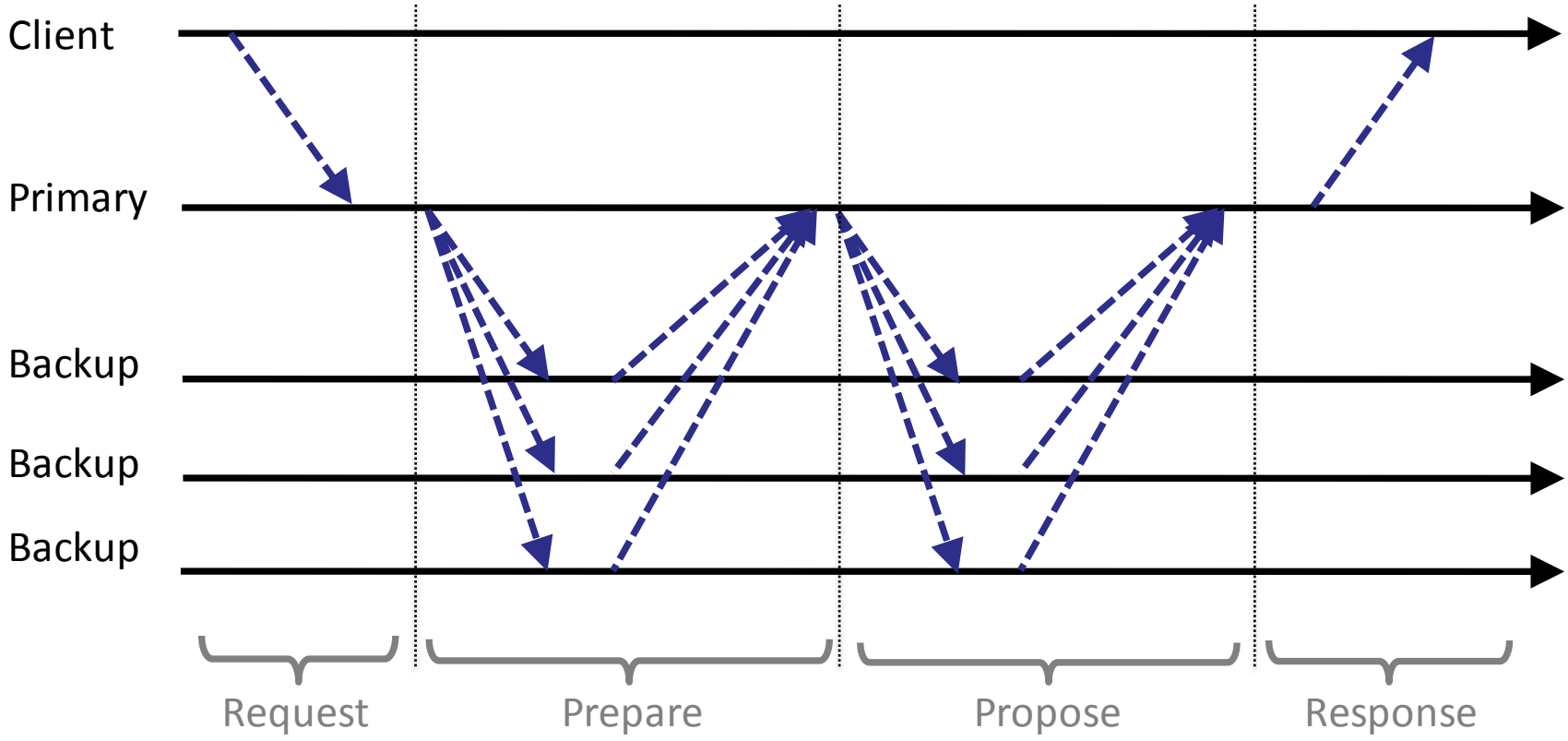
PBFT: Main Algorithm

- PBFT takes 5 rounds of communication
- 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…

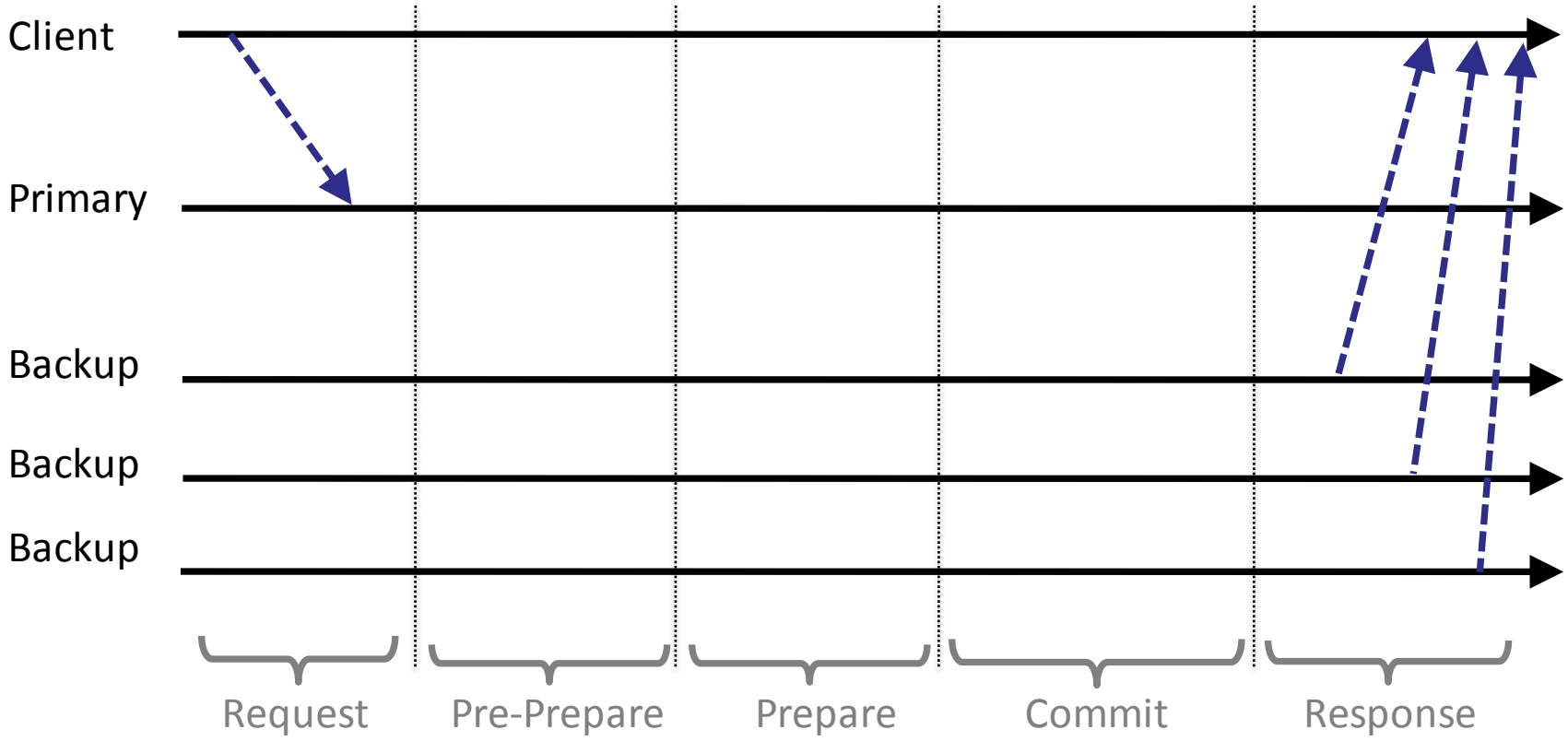
PBFT: Paxos

- 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



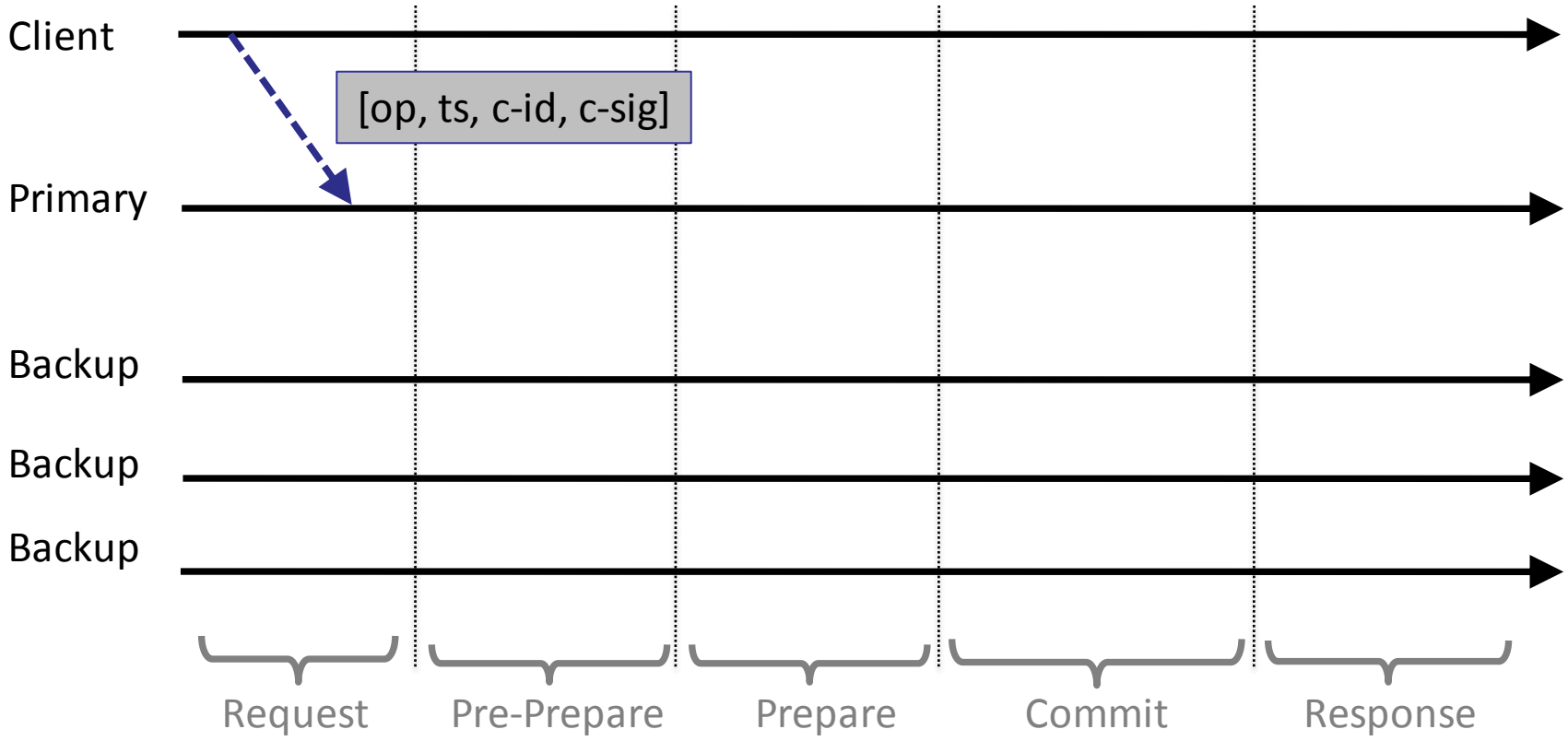
PBFT: Algorithm

- PBFT takes 5 rounds of communication
- The main parts are the three rounds pre-prepare, prepare, and commit



PBFT: Request Phase

- In the first round, the client sends the command op to the primary
- It also sends a timestamp ts, a client identifier c-id and a signature c-sig



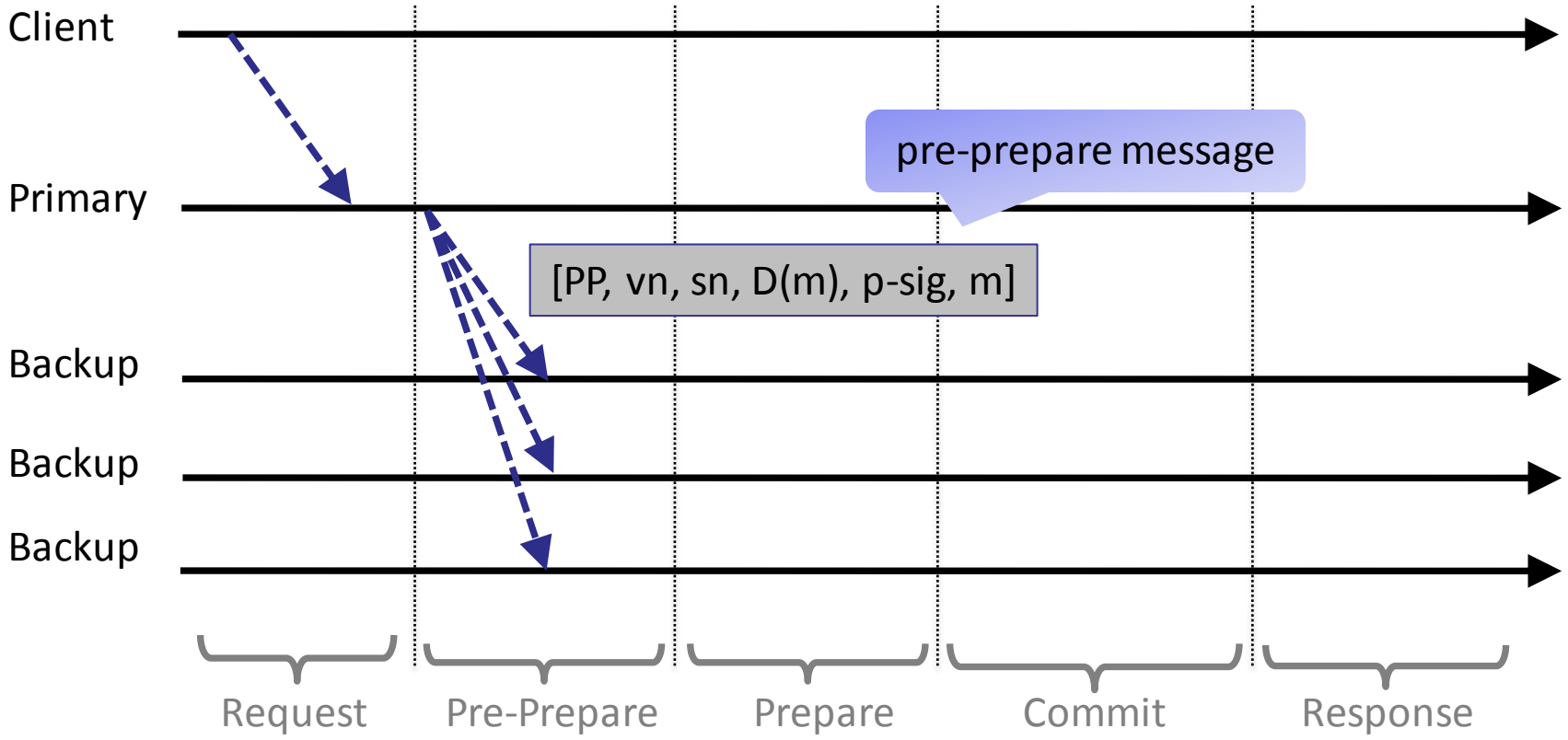
PBFT: Request Phase

- 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

- In the second round, the primary multicasts m = [op, ts, c-id, c-sig] to the backups, including the view number vn, the assigned sequence number sn, the message digest D(m) of m, and its own signature p-sig



PBFT: Pre-Prepare Phase

- The sequence numbers are used to order the commands and the signature is used to verify the authenticity as before

- Why adding the message digest of the client's message?
  - The primary signs only [PP, vn, sn, D(m)]. This is more efficient!

- What is a 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 mod *n*
  - Ideally, all servers are (always) in the same view
  - A view change occurs if a different primary is elected

More on view changes later…

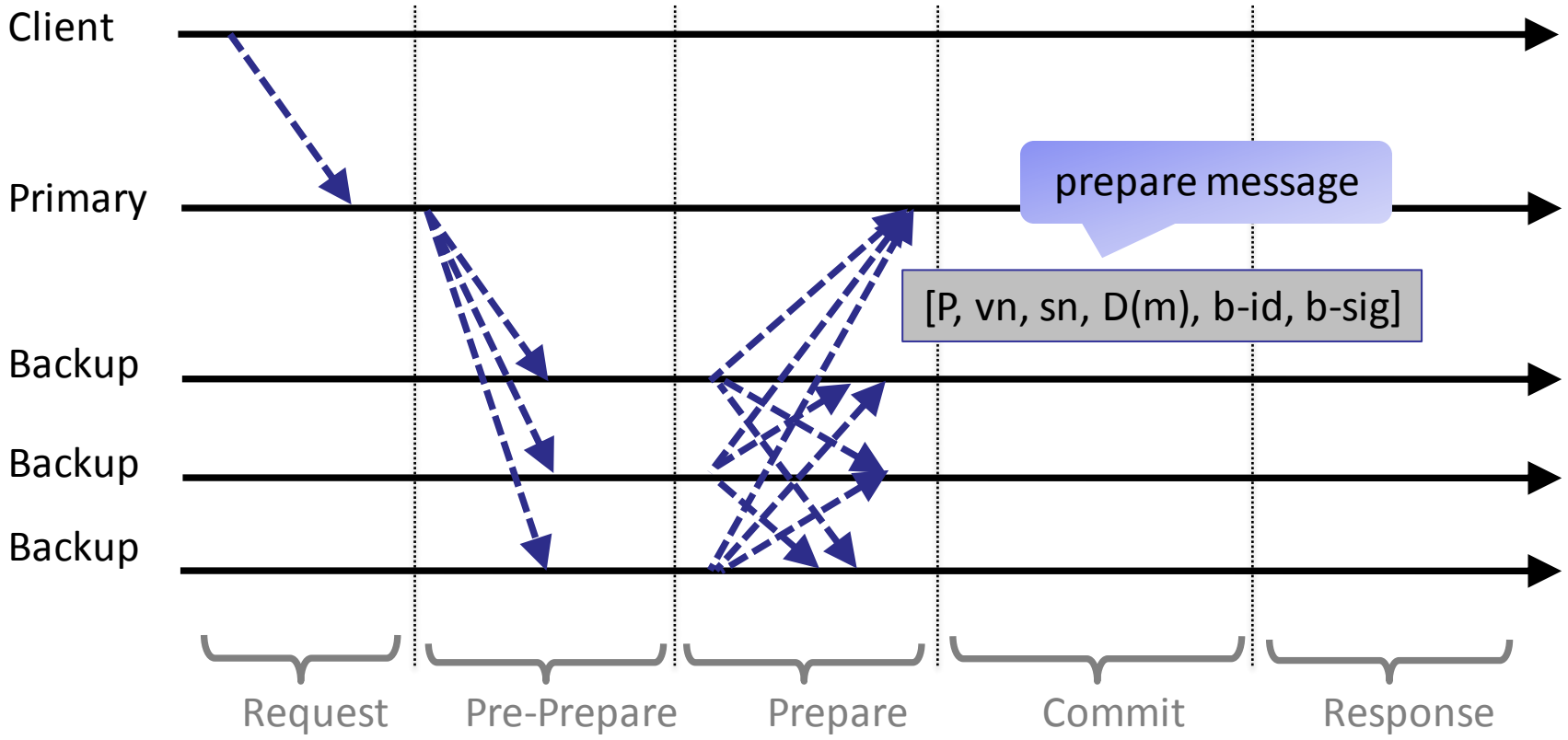
PBFT: Pre-Prepare Phase

- 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
  - The last condition 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



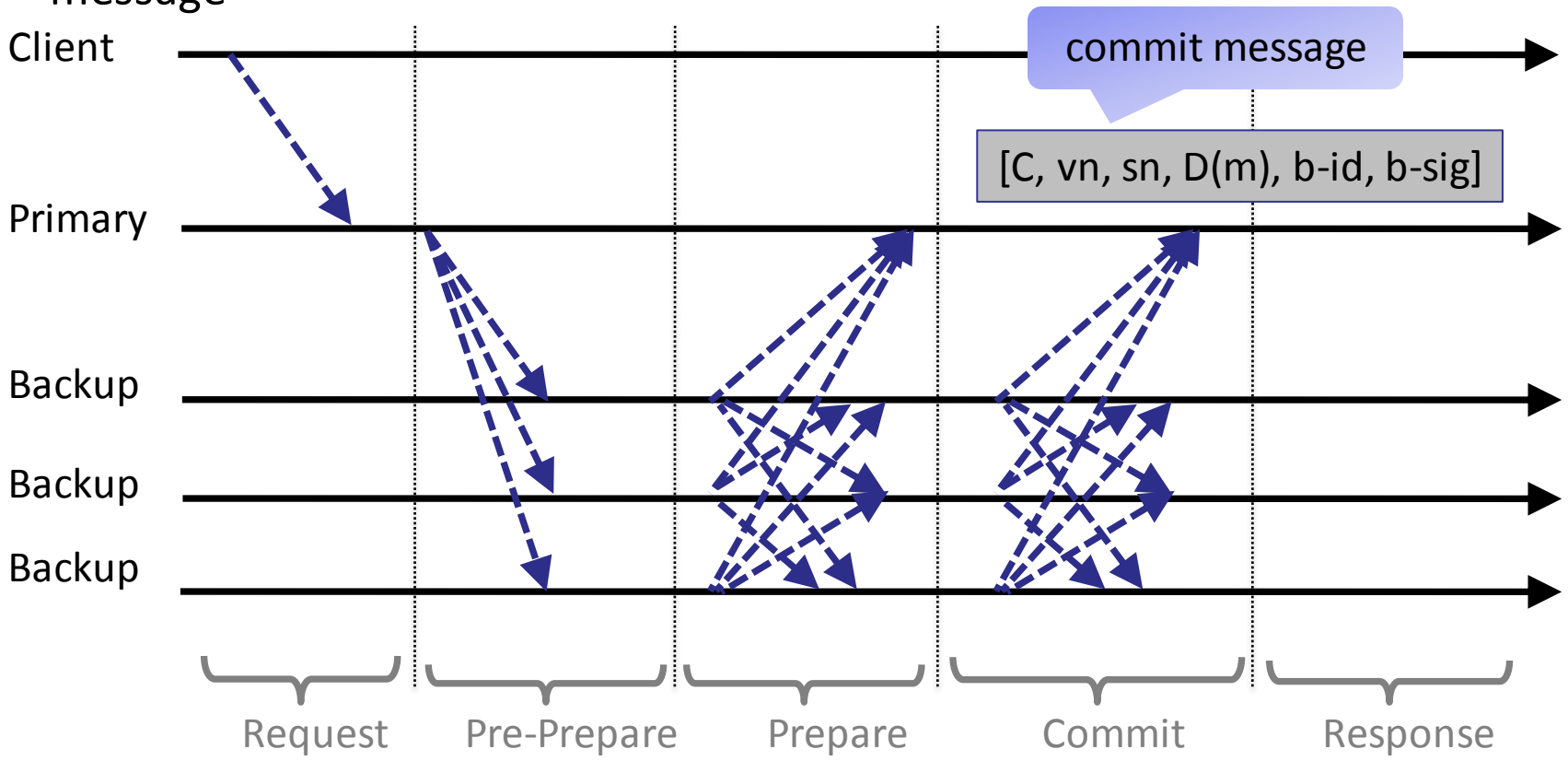
PBFT: Prepare Phase

- 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 a backup b has message m, an accepted pre-prepare message, and $2f$ 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



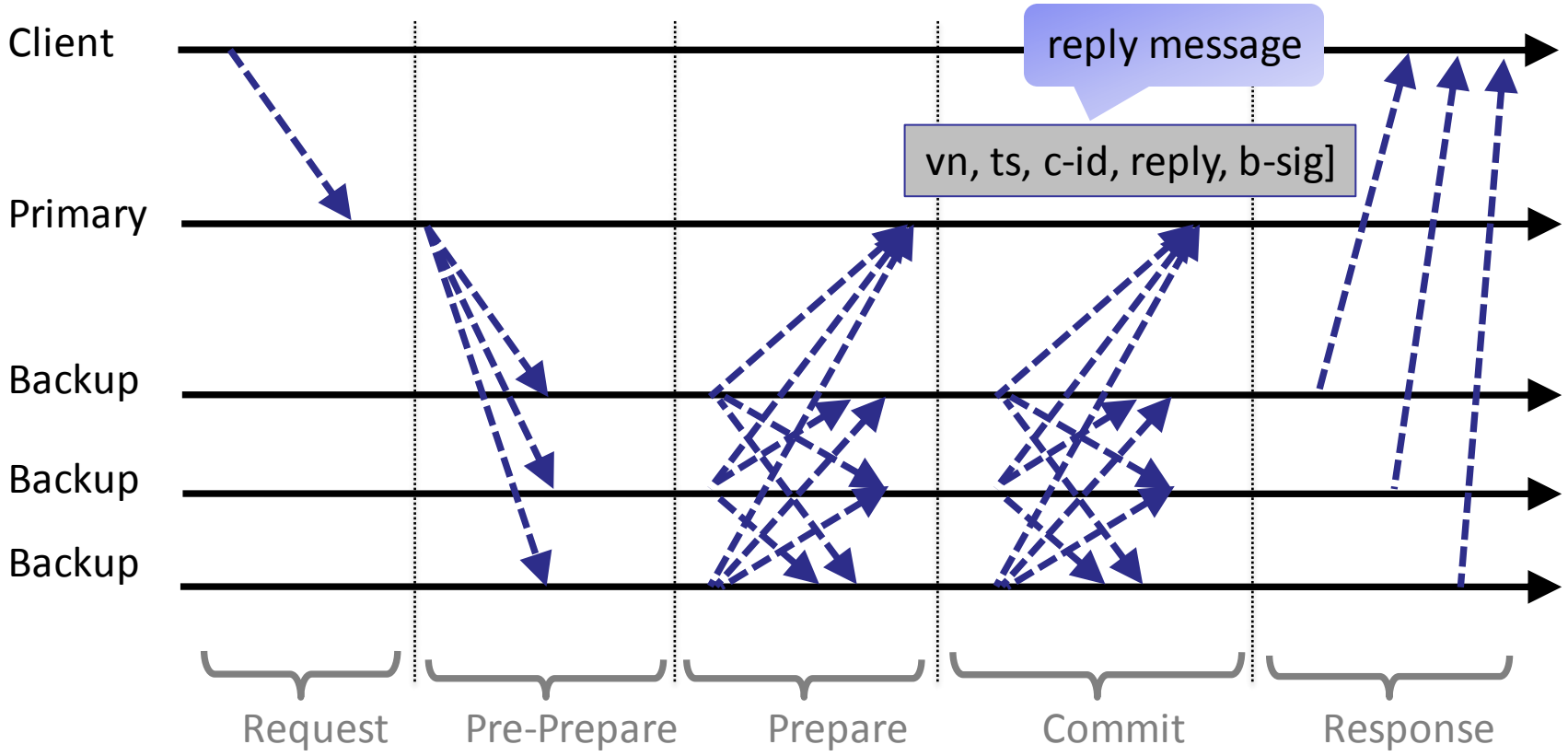
PBFT: Commit Phase

- 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* commit messages, it performs op ("commits") and sends a reply to the client



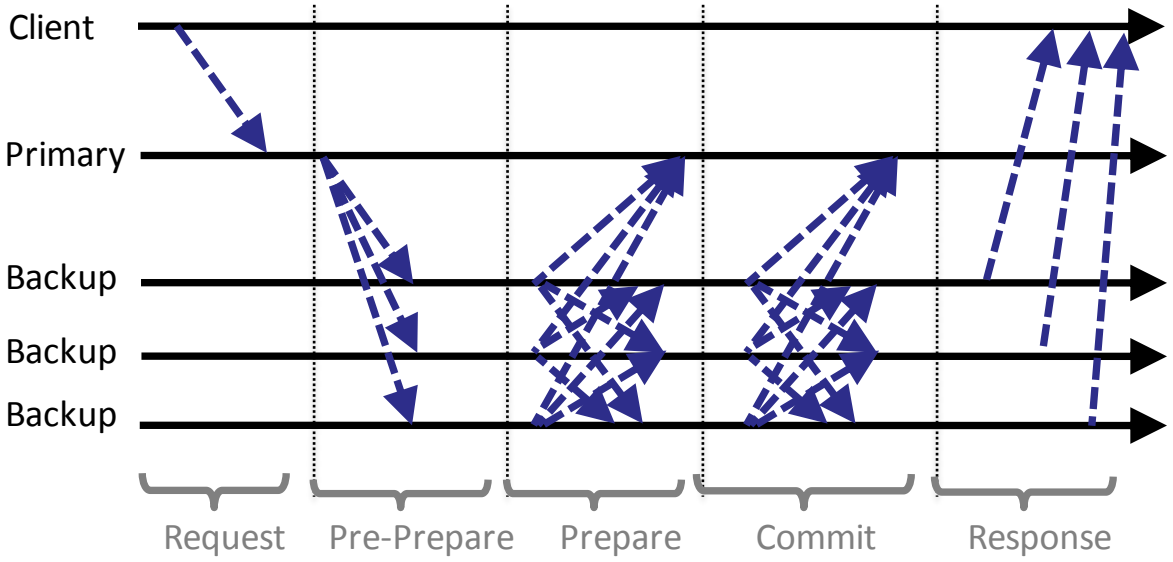
PBFT: Garbage Collection

- 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 checkpoint

  and 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 and sends a prepare message anyway

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 mod n of the next primary and also a certificate for each command for which it accepted 2f+1 prepare messages
  - A certificate simply contains the 2*f*+1 accepted signatures

The next primary!

- When server i receives 2*f* 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≠ m' s.t. D(m) ≠ 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

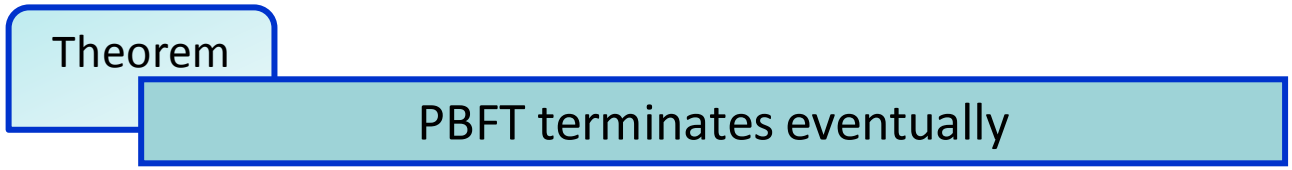
Theorem

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

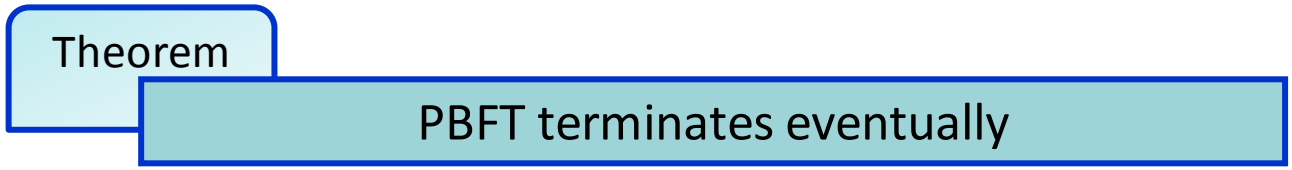
PBFT: Liveness



Proof:

- The primary is correct
  - As we argued before, the algorithm terminates after 5 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



Proof continued:

- 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

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

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

Times are in seconds

- Normal (strict) PBFT is only 26% slower than PBFT without replication
  → Replication does not cost too much!

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

- Normal (strict) PBFT is only 3% slower than NFS-std, and PBFT with read-only lookups is even 2% faster!

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!

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!

Messages do not just "arrive eventually"

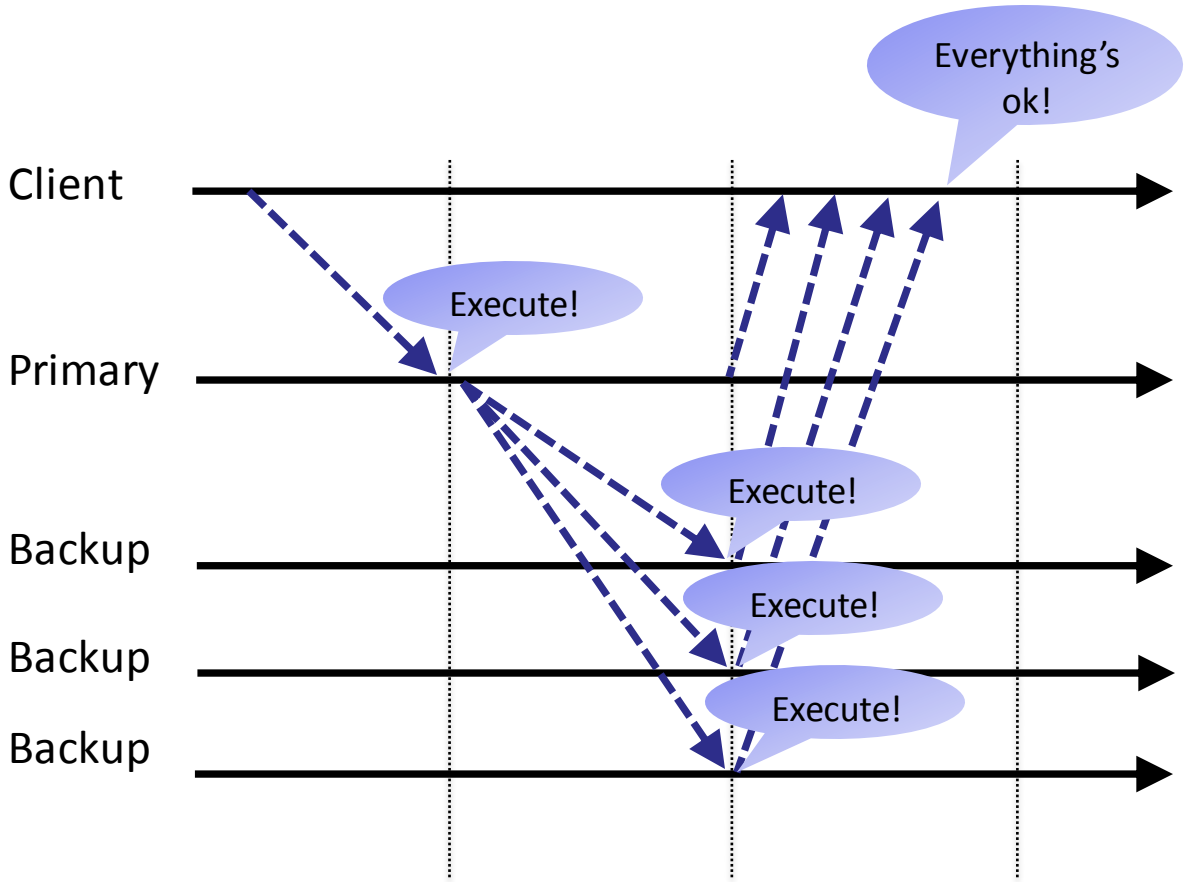
Why?

Zyzzyva

- Zyzzyva 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

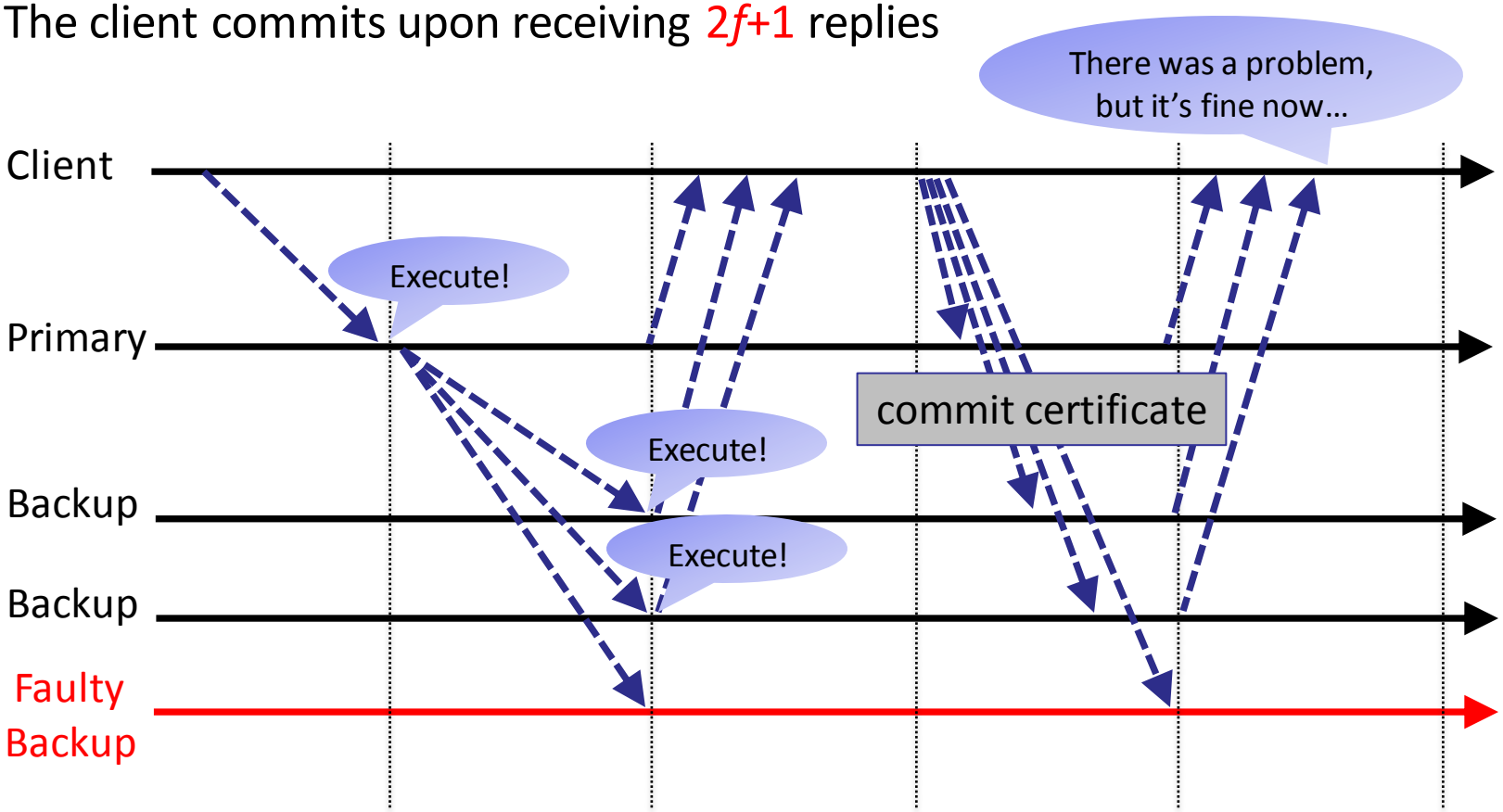
Zyzzyva

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



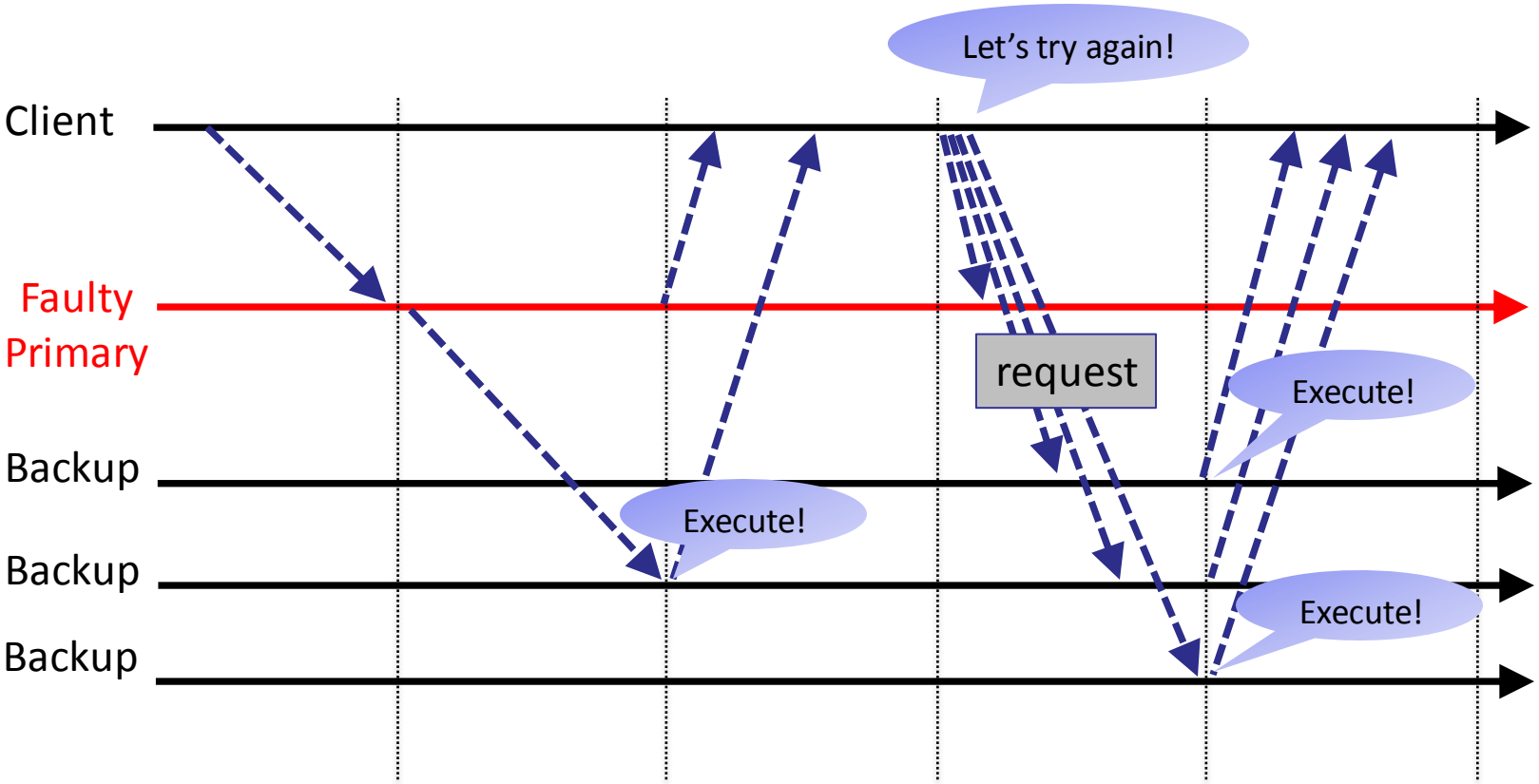
Zyzzyva

- 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



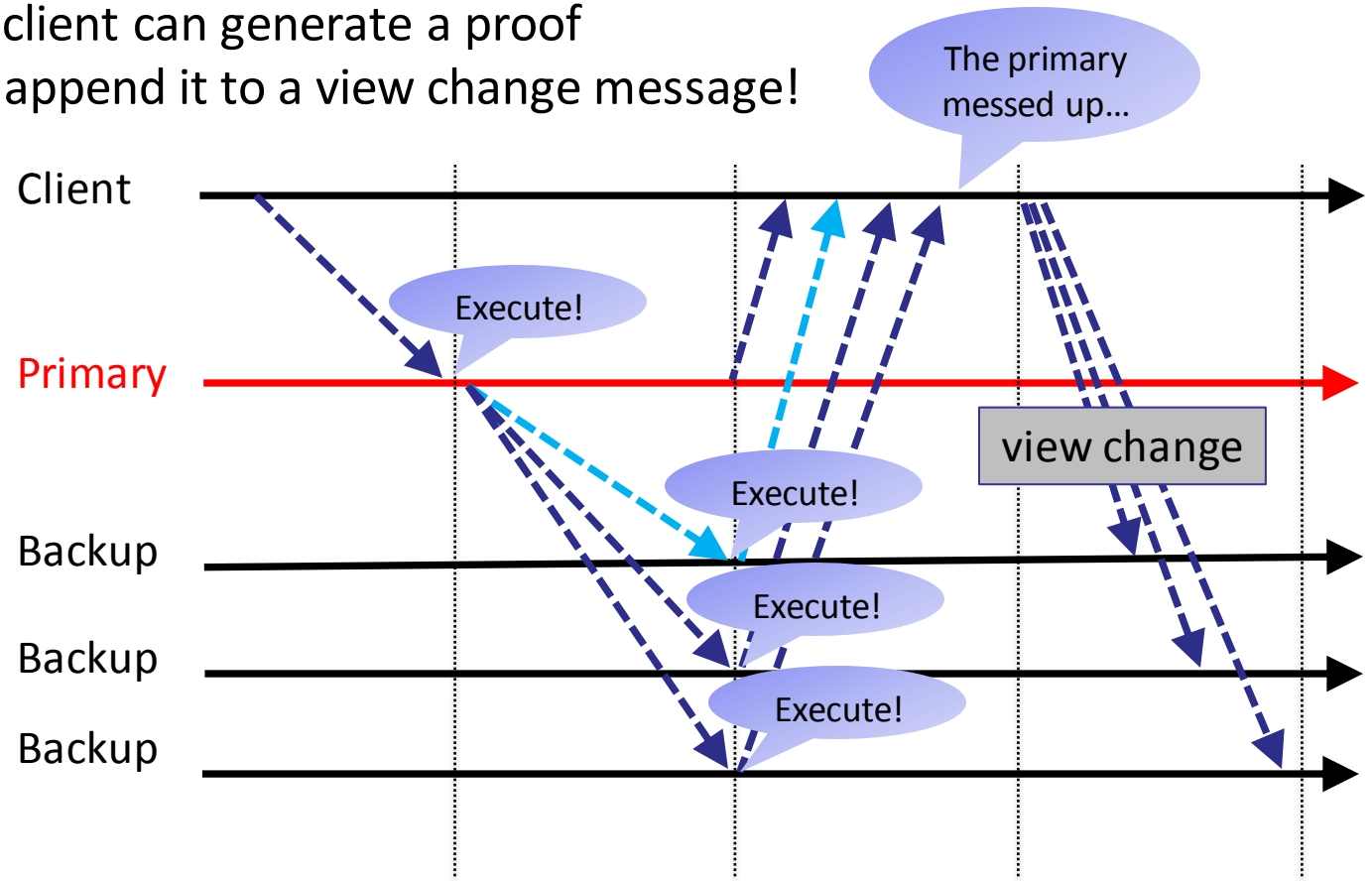
Zyzzyva

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



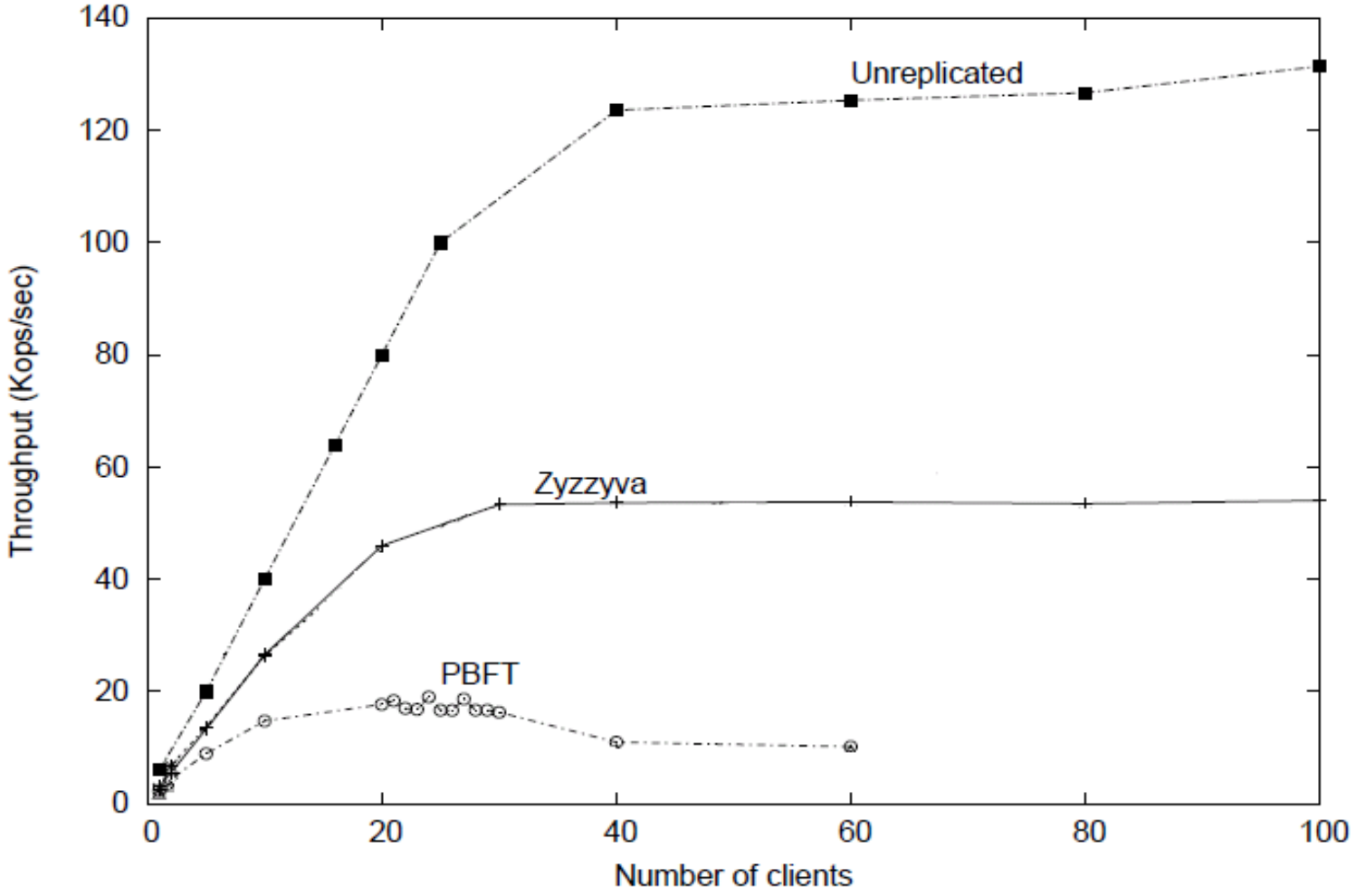
Zyzzyva

- Case 4: The client receives results that indicate an inconsistent ordering by the primary
- The client can generate a proof and append it to a view change message!



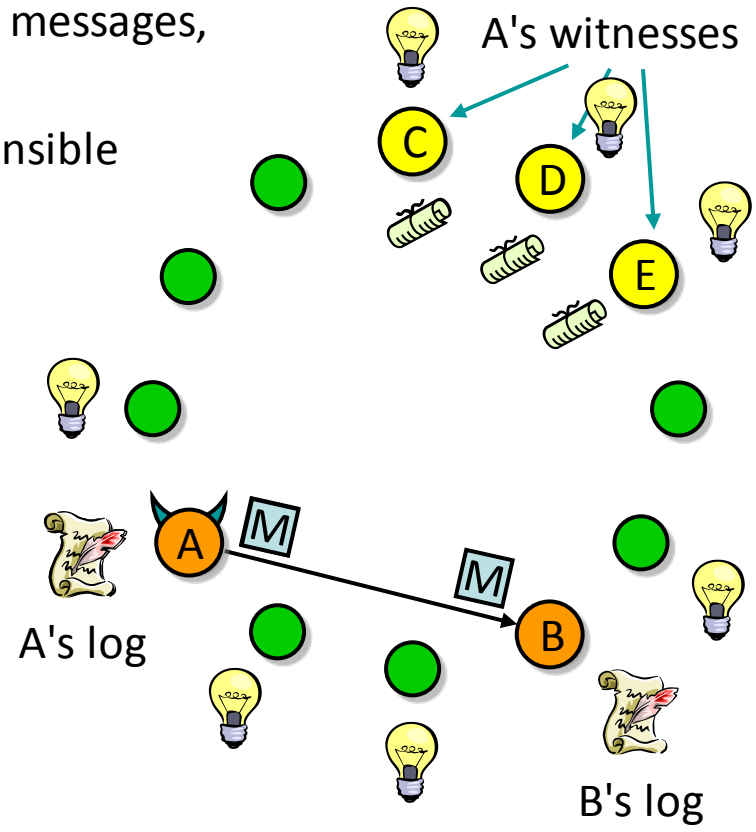
Zyzzyva: Evaluation

- Zyzzyva outperforms PBFT because it normally takes only 3 rounds!



More BFT Systems in a Nutshell: PeerReview

- The goal of PeerReview is to provide accountability for distributed systems
  - All nodes store I/O events, including all messages, in a local log
  - Selected nodes ("witnesses") are responsible for auditing the log
  - If the witnesses detect misbehavior, they generate evidence and make the evidence available
  - Other nodes check the evidence and report the fault
- What if a node tries to manipulate its log entries?
  - Log entries form a hash chain creating secure histories



More BFT Systems in a Nutshell: PeerReview

- PeerReview has to solve the same problems…
  - Byzantine nodes must not be able to convince correct nodes that another correct node is faulty
  - The witness sets must always contain at least one correct node

- PeerReview provides the following guarantees:

1. Faults will be detected
   - If a node commits a fault and it has a correct witness, then the witness obtains a proof of misbehavior or a challenge that the faulty node cannot answer
2. Correct nodes cannot be accused
   - If a node is correct, then there cannot be a correct proof of misbehavior and it can answer any challenge

More BFT Systems in a Nutshell: FARSITE

- "**F**ederated, **A**vailable, and **R**eliable **S**torage for an **I**ncompletely **T**rusted **E**nvironment"
- Distributed file system without servers
- Clients contribute part of their hard disk to FARSITE

- Resistant against attacks: It tolerates $f < n/3$ Byzantine clients
- Files
  - f+1 replicas per file to tolerate f failures
  - Encrypted by the user
- Meta-data/Directories
  - 3f+1 replicas store meta-data of the files
  - File content hash in meta-data allows verification
  - How is consistency established? FARSITE uses PBFT!

More efficient than replicating the files!

Credits

- The Paxos algorithm is due to Lamport, 1998.
- The Chubby system is from Burrows, 2006.
- PBFT is from Castro and Liskov, 1999.
- Zyzyvva is from Kotla, Alvisi, Dahlin, Clement, and Wong, 2007.

That's all, folks!

Questions & Comments?

Roger Wattenhofer