Fault-Tolerance: Practice

Chapter 7

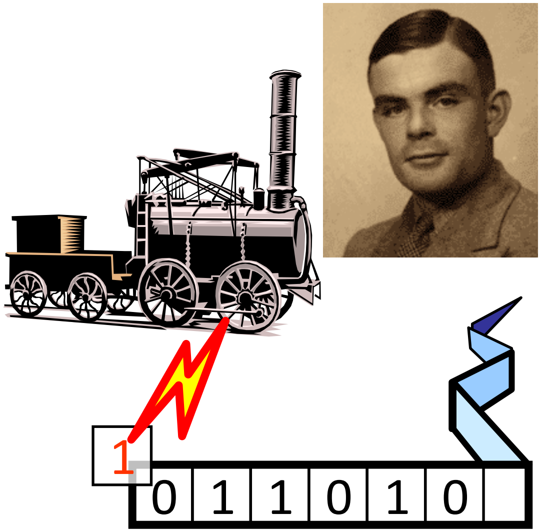
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Overview

- Introduction
- Crash Failures
  - Primary Copy
  - Two-Phase Commit
  - Three-Phase Commit
- Crash-Recovery Failures
  - Paxos
  - Chubby
- Practical Byzantine Fault-Tolerance
- Large-scale Fault-Tolerant Systems

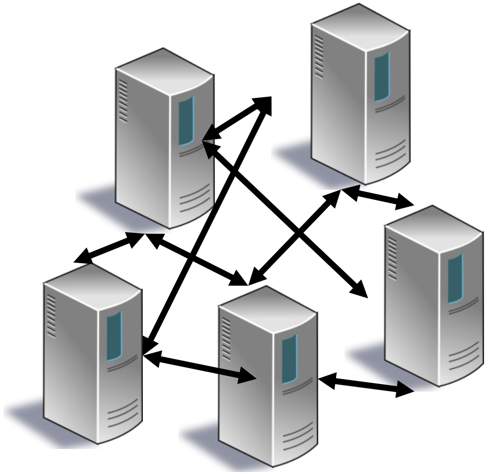
Computability vs. Efficiency

- 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?



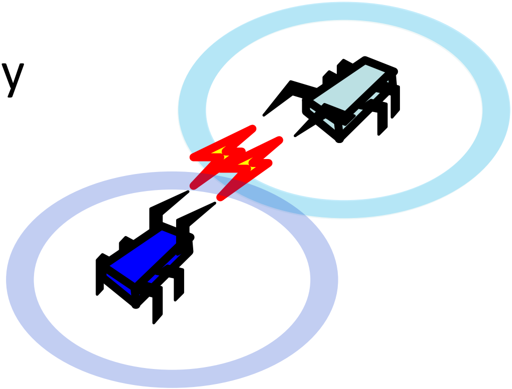
Worst-case scenarios!

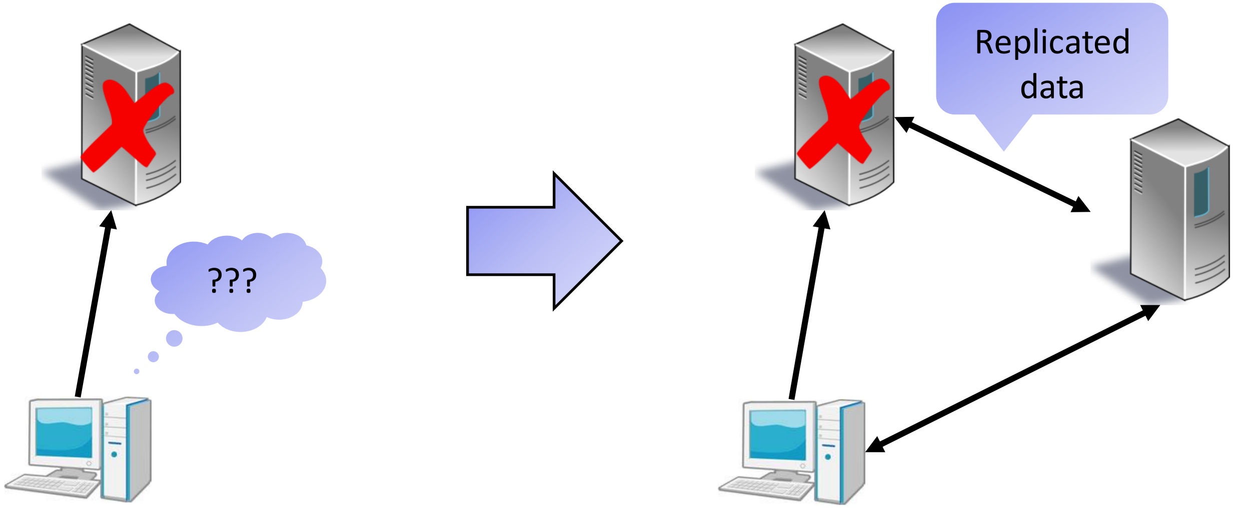
- In this part, we consider practical solutions
  - Simple approaches that work well in practice
  - Focus on efficiency



Fault-Tolerance in Practice

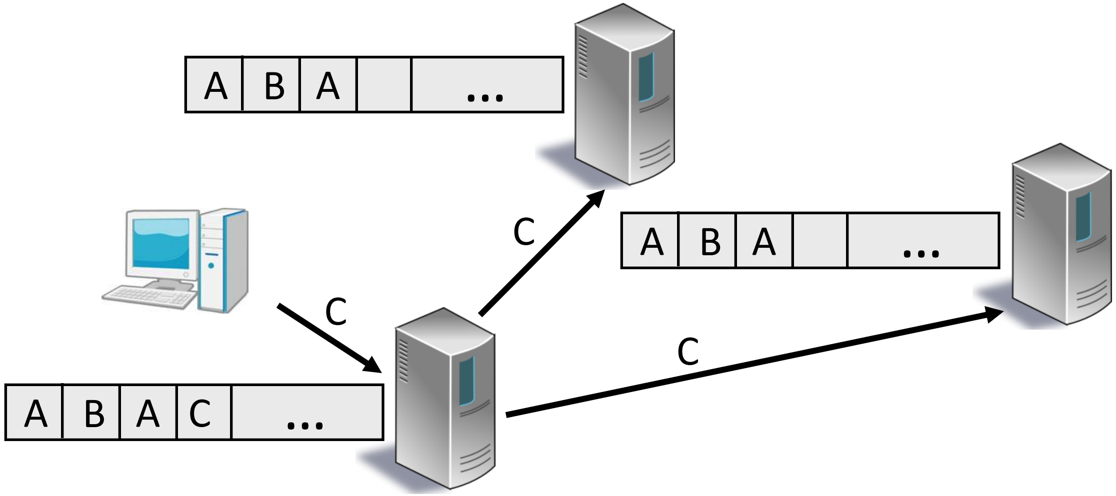
- So far, we studied how to reach consensus in theory
- Why do we need consensus?
- Fault-Tolerance is achieved through *replication*





State Replication

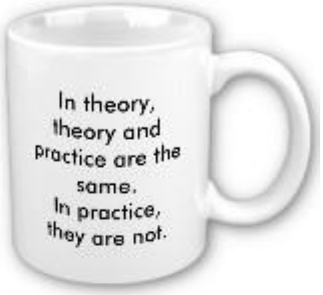
- The state of each server has to be updated in the same way
- This ensures that all servers are in the same state whenever all updates have been carried out!



- The servers have to agree on each update

  → Consensus has to be reached for each update!

From Theory to Practice

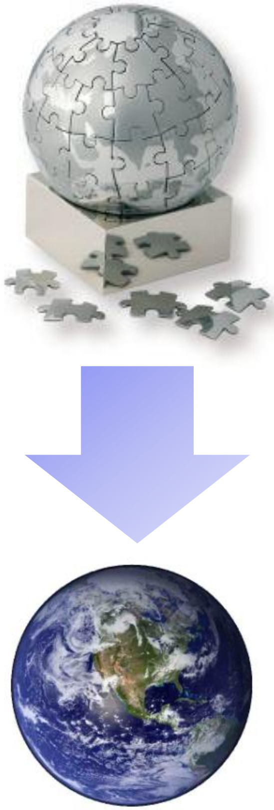


- We studied a lot of theoretical concepts
  - Communication: Synchronous vs. asynchronous
  - Communication: Message passing vs. shared memory
  - Failures: Crash failures vs. Byzantine behavior
- How do these concepts translate to the real world?
  - Communication is often not synchronous, but not 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!

From Theory to Practice

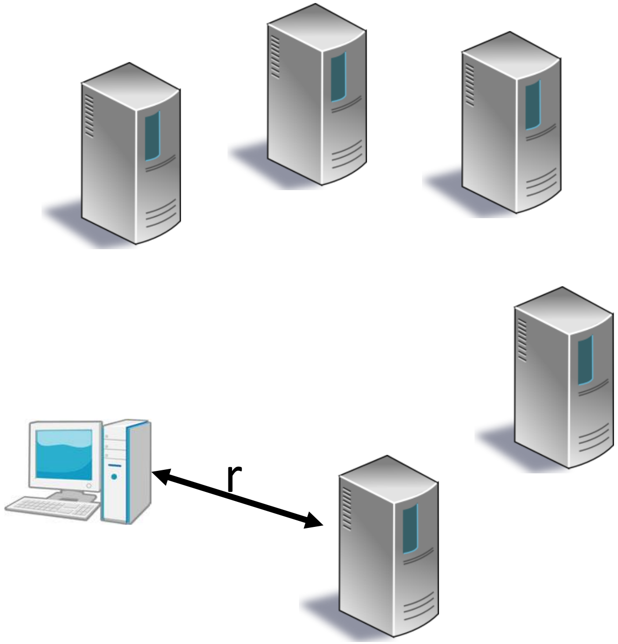
- We studied some impossibility results
  - Impossible to guarantee consensus using a deterministic algorithm in asynchronous systems even if only one node is faulty
- But we want to solve consensus in asynchronous systems!
- So, how do we go from theory to practice…?
  - Real-world algorithms also make assumptions about the system
  - These assumptions allow us to circumvent the lower bounds!
- In the following, we discuss techniques/algorithms that are (successfully) used in practical systems
  - We will also talk about their assumptions and guarantees



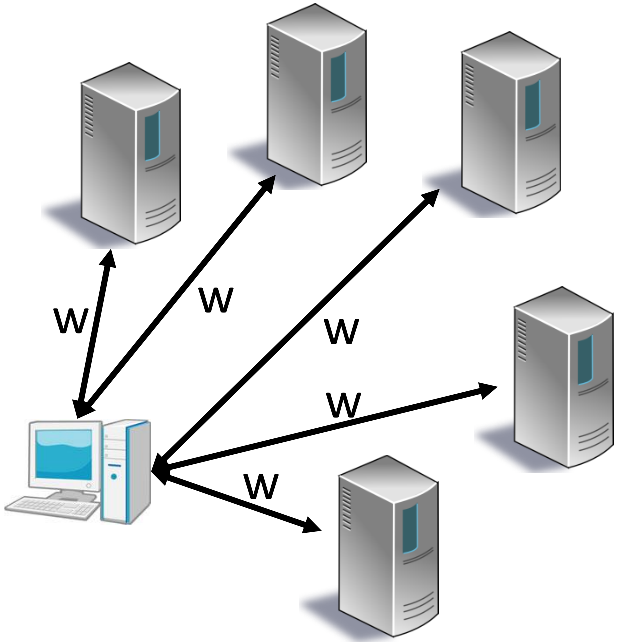
Replication is Expensive

- 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:



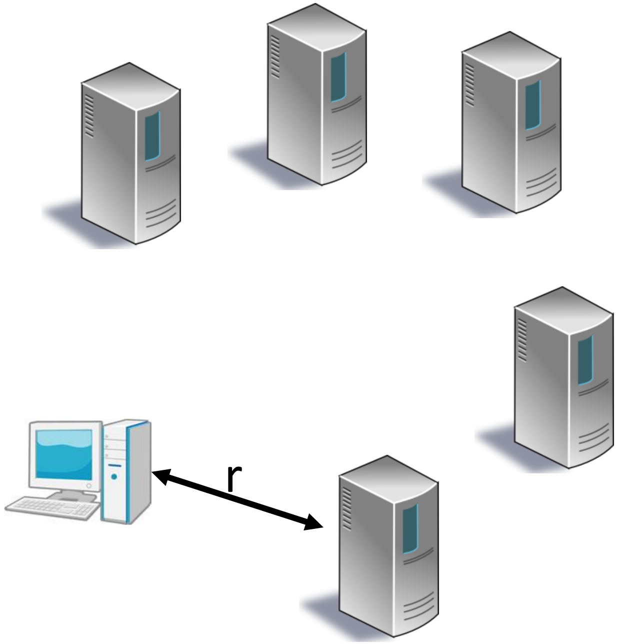
Write:



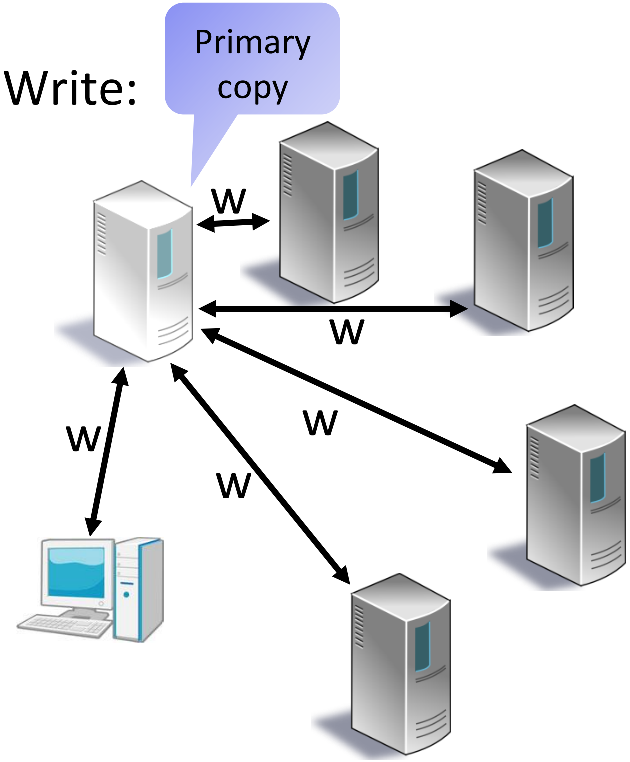
Primary Copy

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

Read:

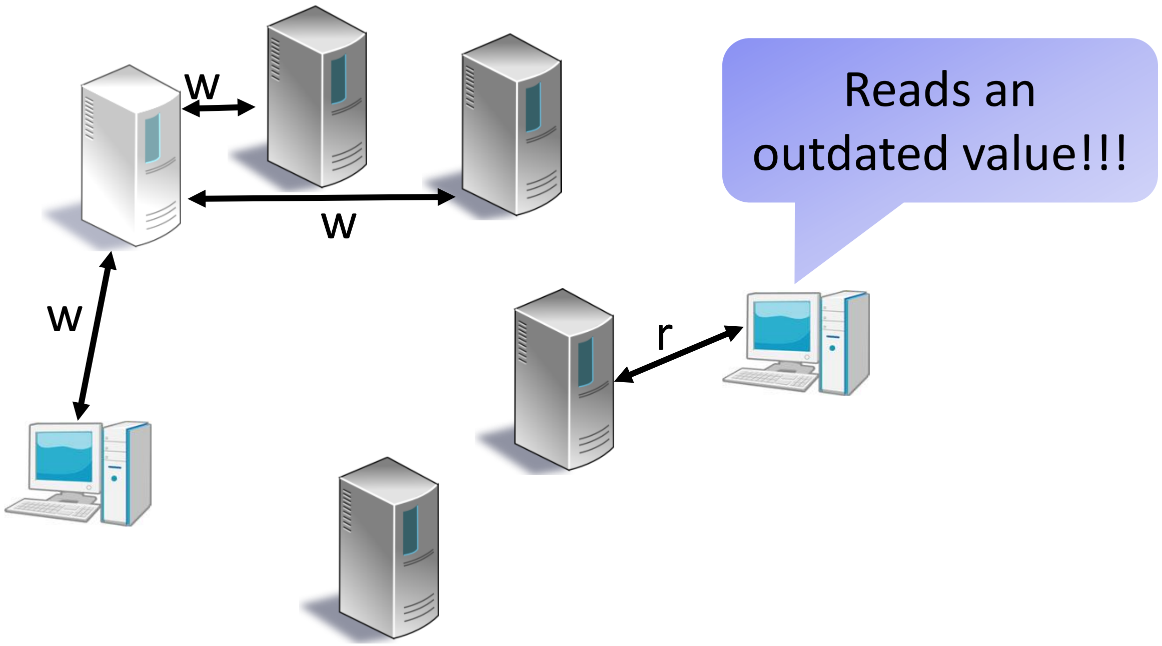


Write:



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

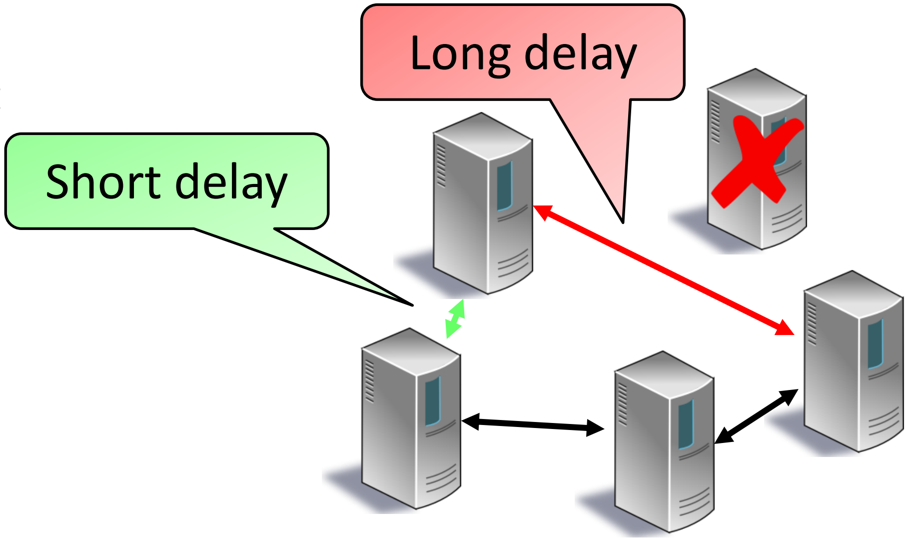


Reads an outdated value!!!

Transactions

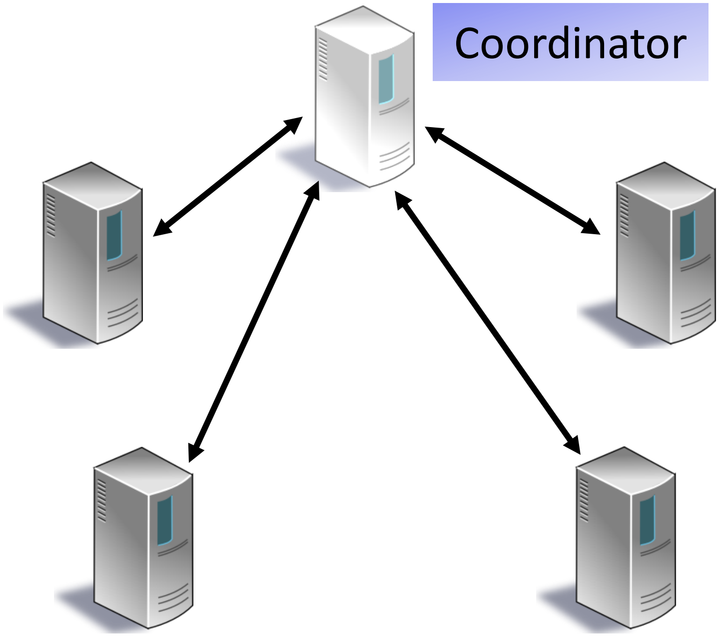
- In order to achieve consistency, updates have to be atomic
- A write has to be an atomic transaction
  - Updates are synchronized

- 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!



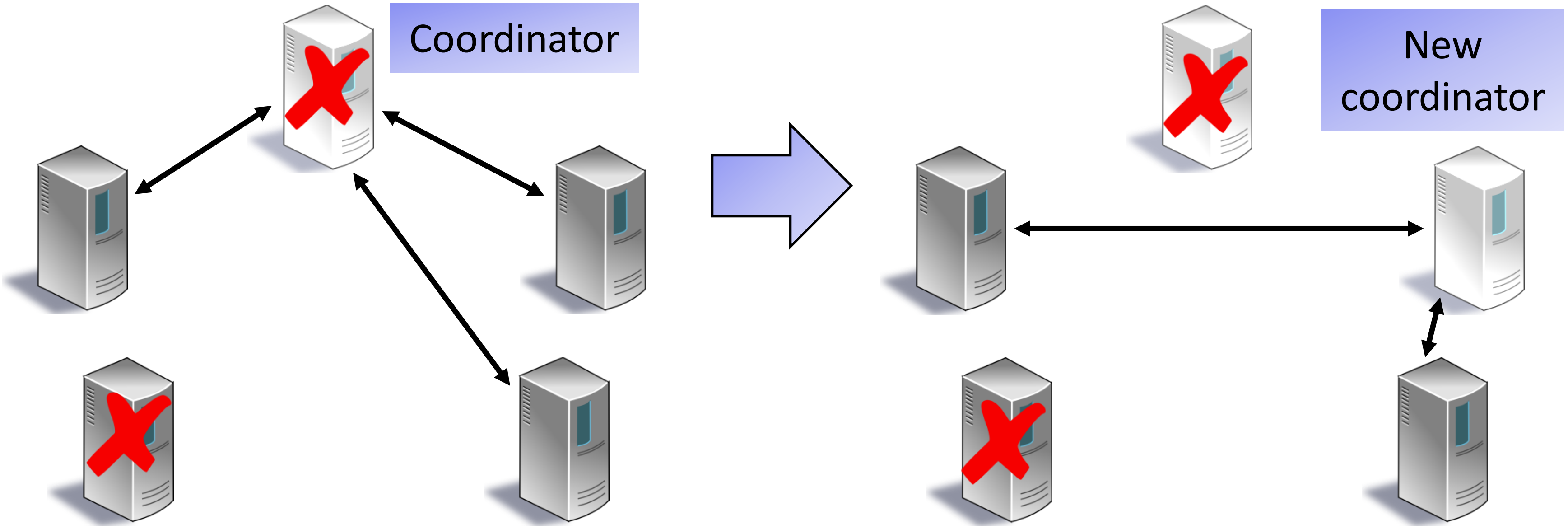
Two-Phase Commit (2PC)

- 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



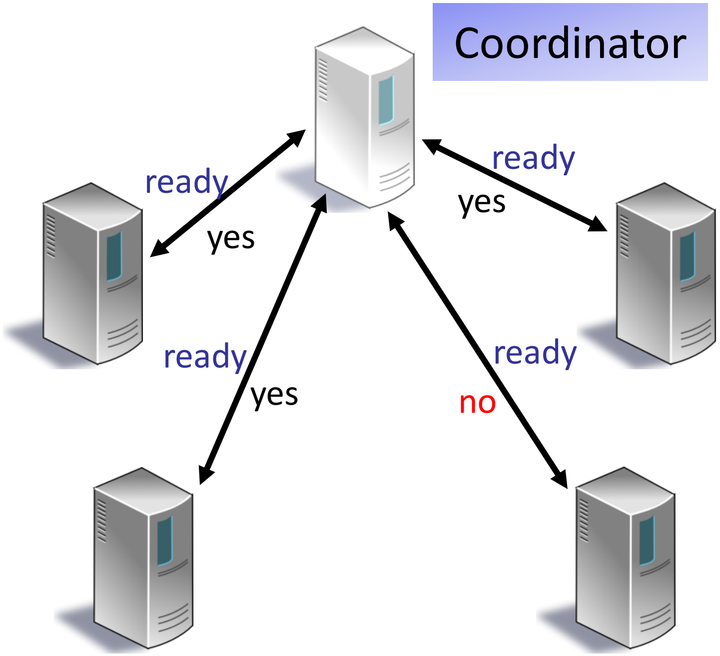
Two-Phase Commit: Failures

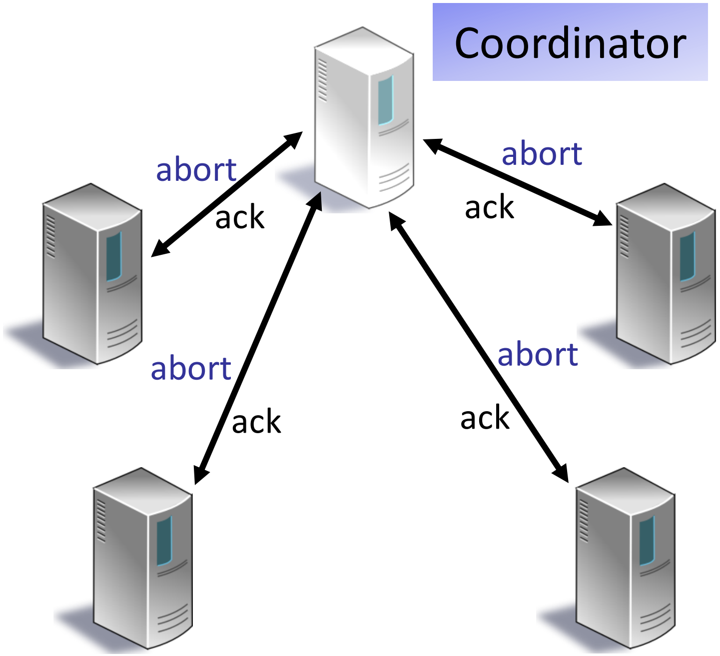
- Fail-stop model: We assume that a failed node does not re-emerge
- Failures are detected (instantly)
  - E.g. time-outs are used in practical systems to detect failures
- If the coordinator fails, a new coordinator takes over (instantly)
  - How can this be accomplished reliably?



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

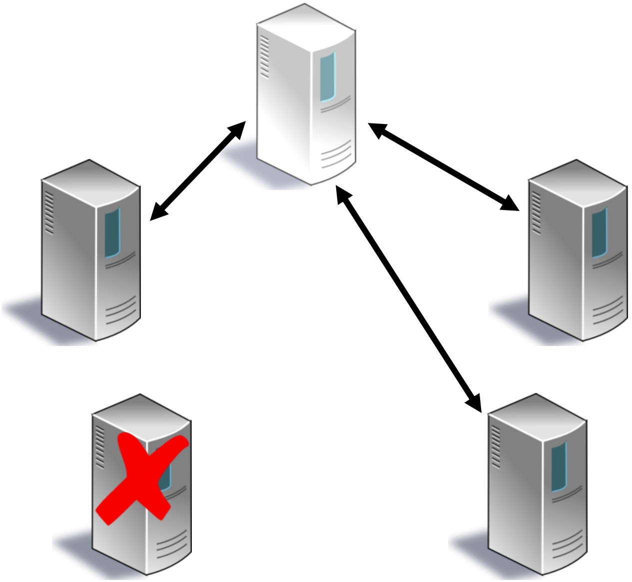
```
If a node receives ready from the coordinator:
If it is ready to commit
    Send yes to coordinator
else
    Send no to coordinator
```

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

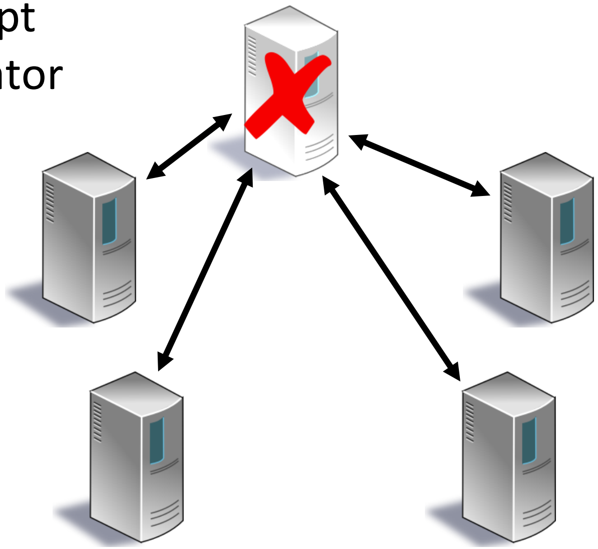


Two-Phase Commit: Analysis

- 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…

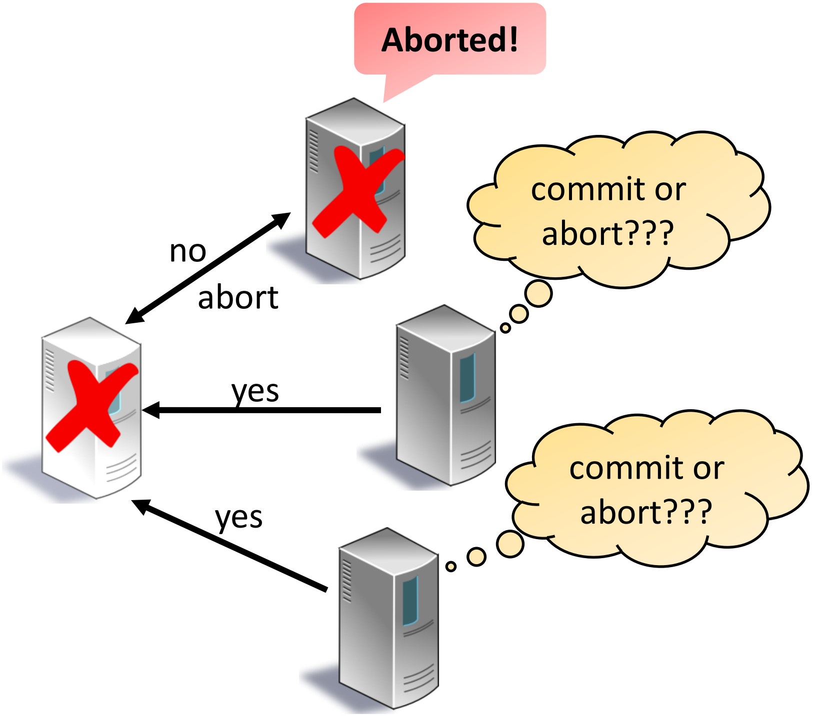
- How does the new coordinator proceed?
  - It must ask the other nodes if a node has already received a commit
  - A node that has received a commit replies yes, otherwise it sends no and promises not to accept a commit that may arrive from the old coordinator
  - If some node replied yes, the new coordinator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



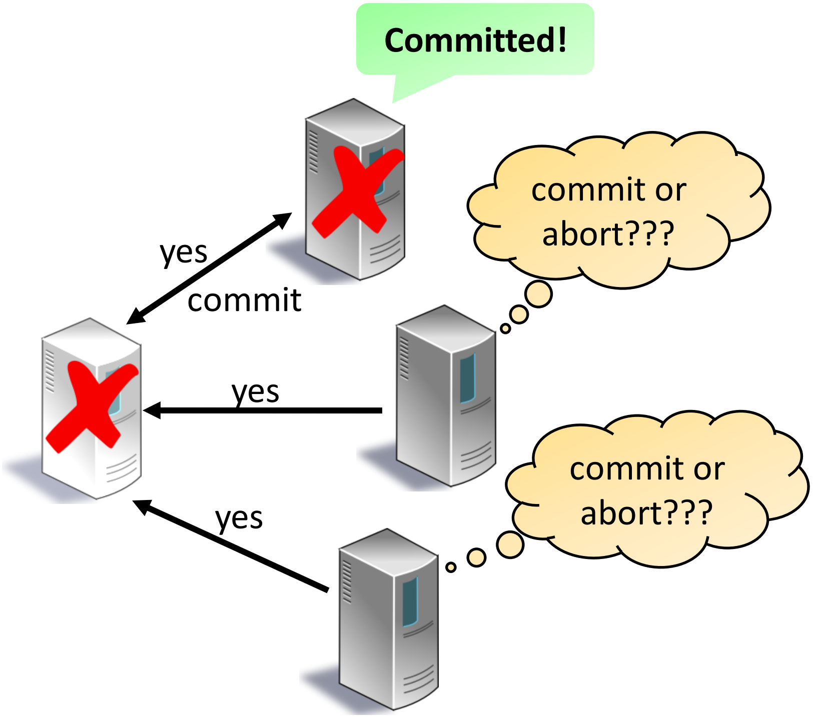
- This works if there is only one failure
- Does 2PC still work with multiple failures…?

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?



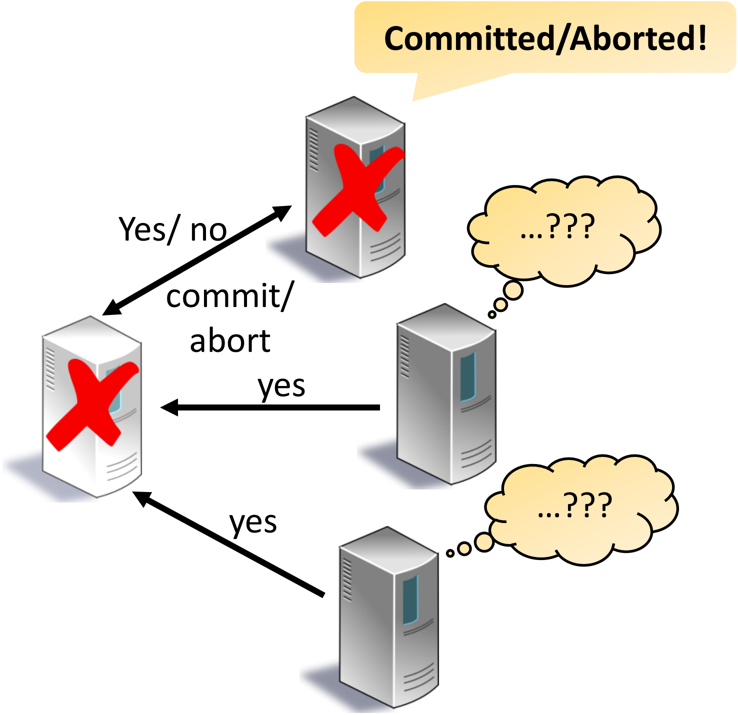
The nodes cannot commit!



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

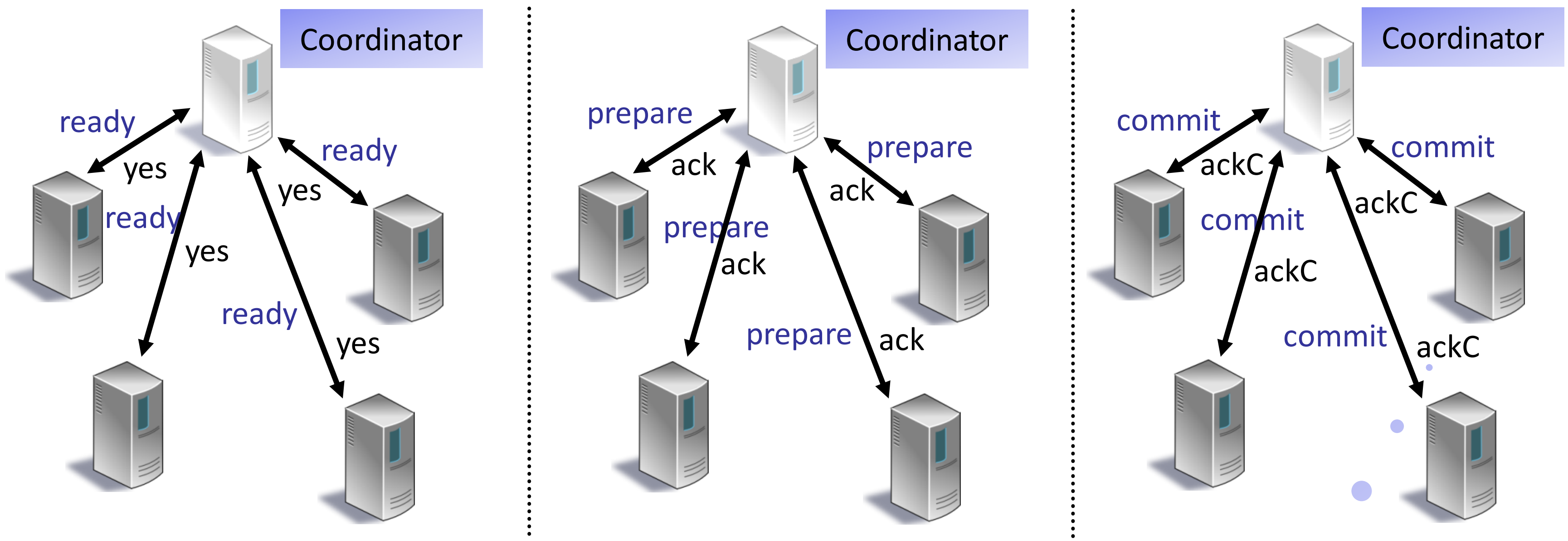
- Solution: Add another phase to the protocol!
  - The new phase precedes the commit phase
  - The goal is to inform all nodes that all are ready to commit (or not)
  - At the end of this phase, every node knows whether or not all nodes want to commit *before* any node has actually committed or aborted!

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



acknowledge
commit

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

- 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!

In order to solve consensus, you first need to solve consensus…

  - 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

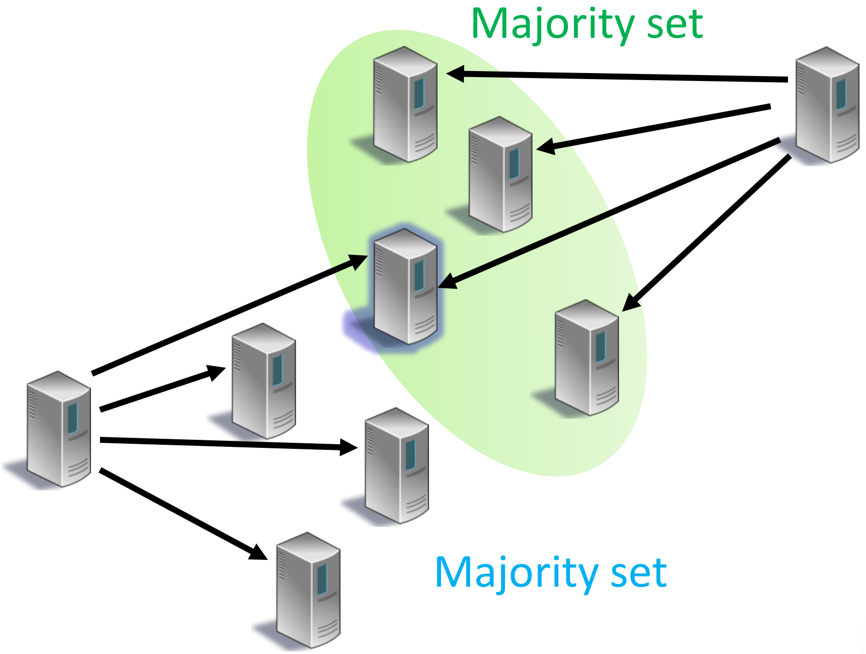
- Historical note
  - In the 1980s, a fault-tolerant distributed file system called “Echo” was built
  - According to the developers, it achieves “consensus” despite any number of failures as long as a majority of nodes is alive
  - The steps of the algorithm are simple if there are no failures and quite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

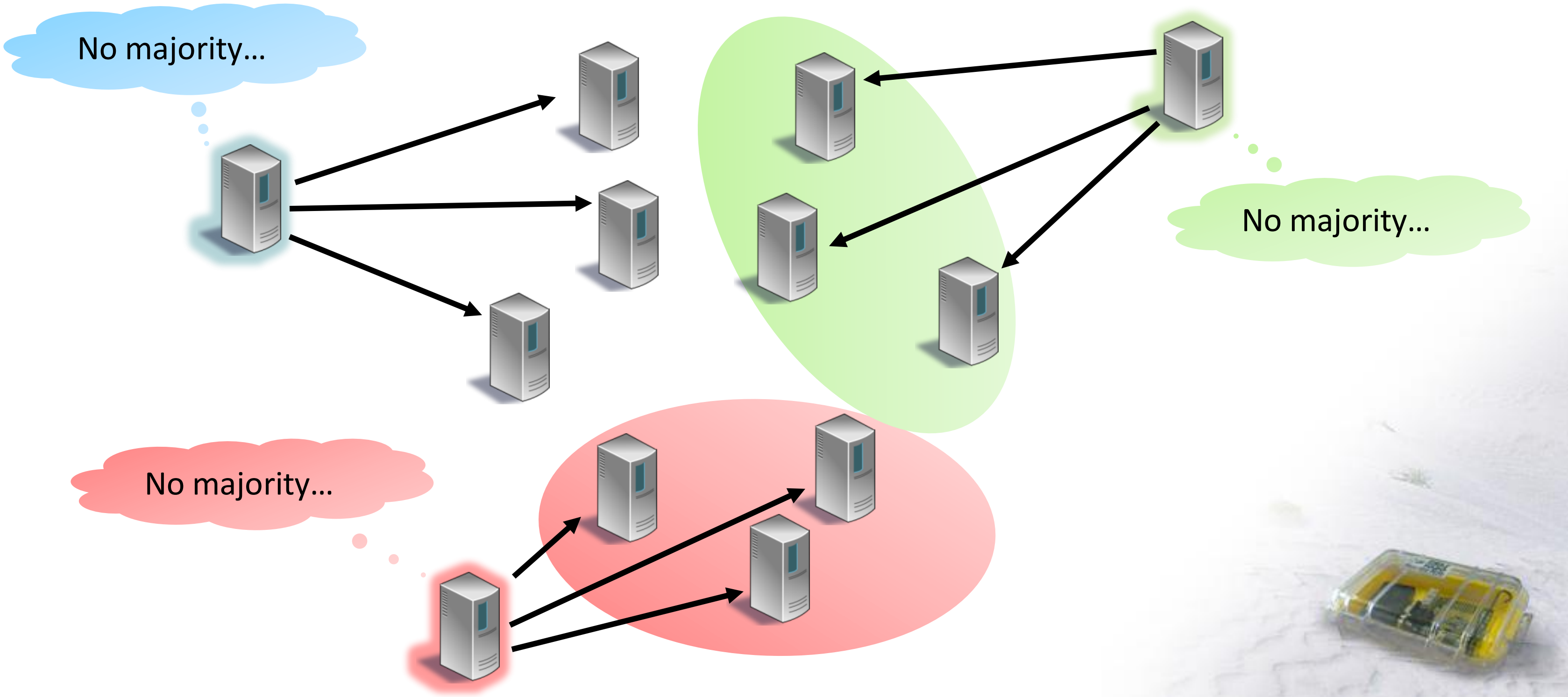
Paxos: Majority Sets

- 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

There are three roles

- Each node has one or more 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!

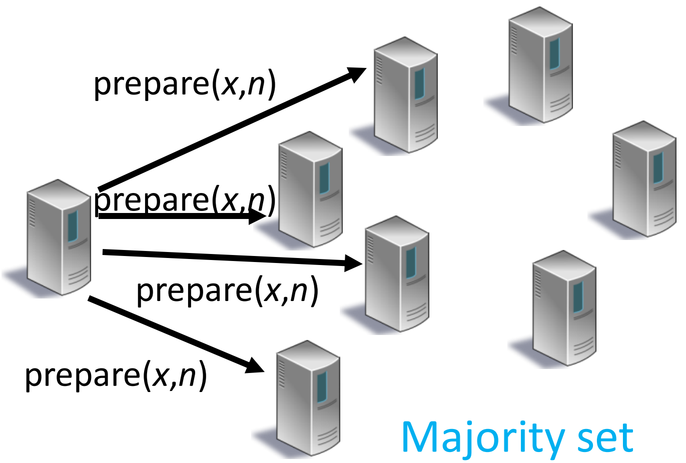
- An acceptor can *accept* any number of proposals
  - An accepted proposal may not necessarily be *chosen*
  - The value of a *chosen proposal* is the *chosen value*
- An acceptor can even *choose* any number of proposals
  - However, if two proposals (x,n) and (y,m) are chosen, then $x = y$

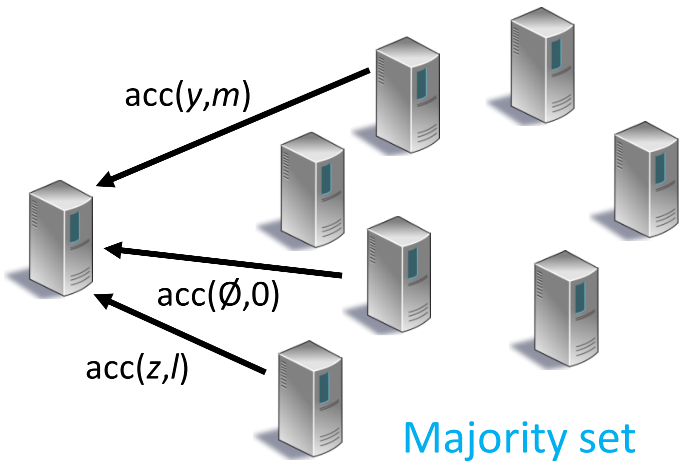
Consensus: Only one value can be chosen!

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 $m < n$!

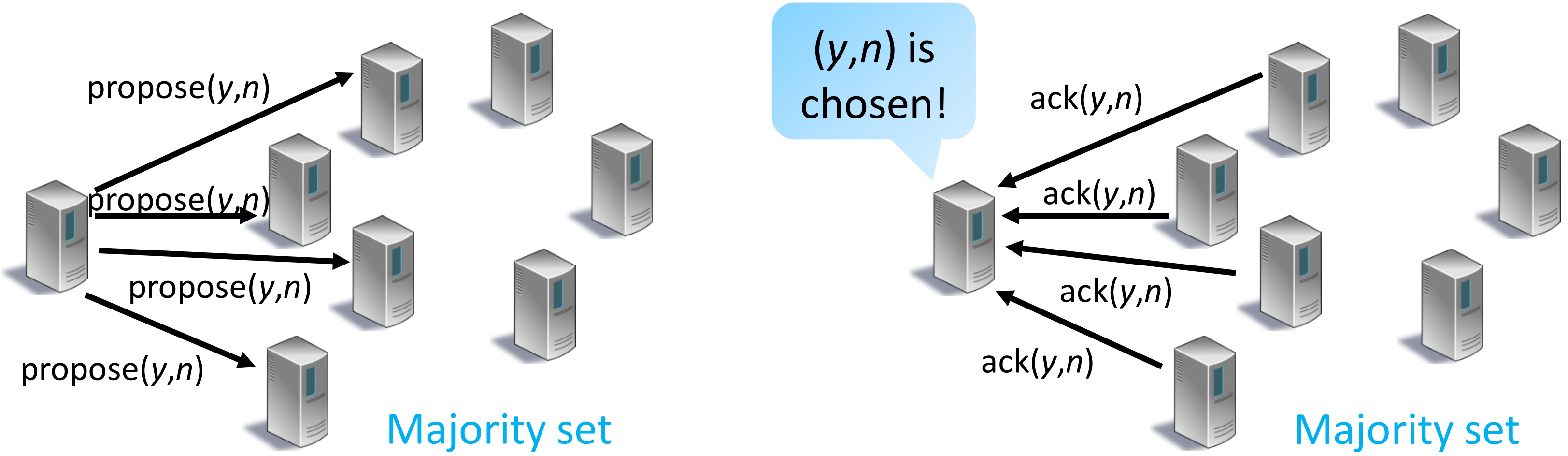




This is the first phase!

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($y$,$n$)
        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
$(x_{last}, n_{last})$ := (Ø,0)
```

Largest request number ever received

Last accepted proposal

Acceptor receives prepare (x,n):

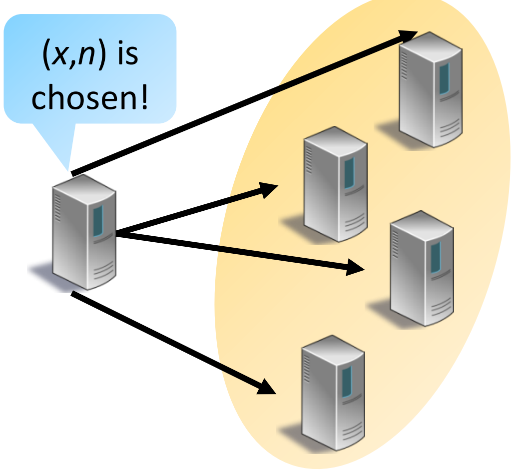
```
if $n > n_{max}$ then
    $n_{max} := n$
    Send acc($x_{last}, n_{last}$) to the proposer
```

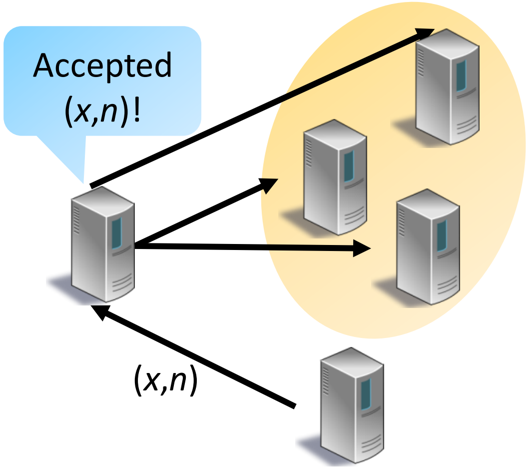
Acceptor receives proposal (x,n):

```
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 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,m) for which $m > n$ it holds that $x = y$

Proof:

- Assume that there are proposals (y,m) for which $m > n$ and $x \neq y$. Consider the proposal with the smallest proposal number m
- 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 $m > n$, the nodes in S must have received prepare(y,m) after (x,n) has been accepted
- This implies that the proposer of (y,m) would also propose the value x unless another acceptor has accepted a proposal (z,l), $z \neq x$ and $n < l < m$
- However, this means that some node must have proposed (z,l), a contradiction because $l < m$ and we said that m is the smallest p.n.!

Paxos: Theorem

Theorem

If a value is chosen, all nodes choose this value

Proof:

- Once a proposal (x,n) is chosen, each proposal (y,m) that is sent afterwards has the same proposal number, 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…?

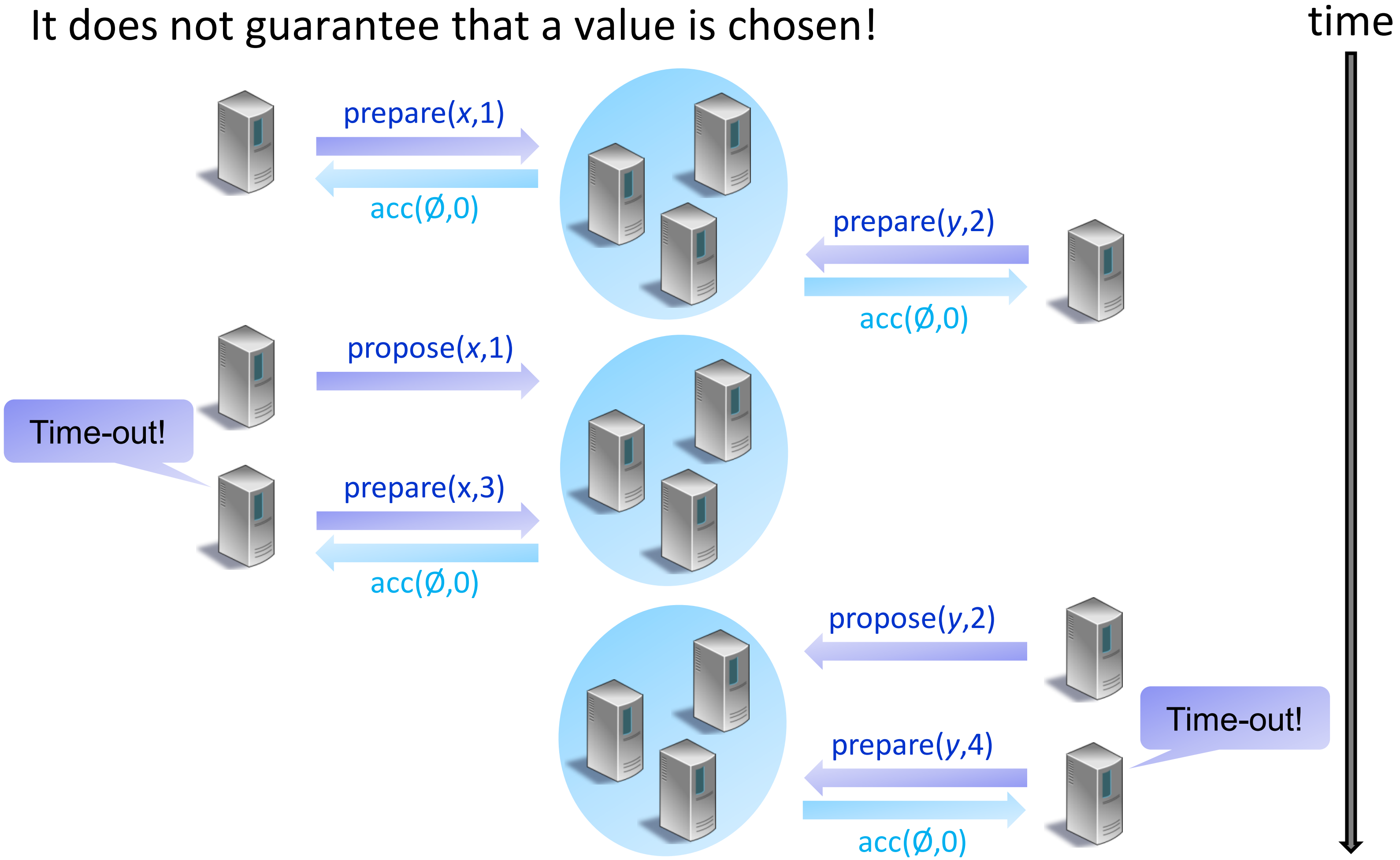
Theorem

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

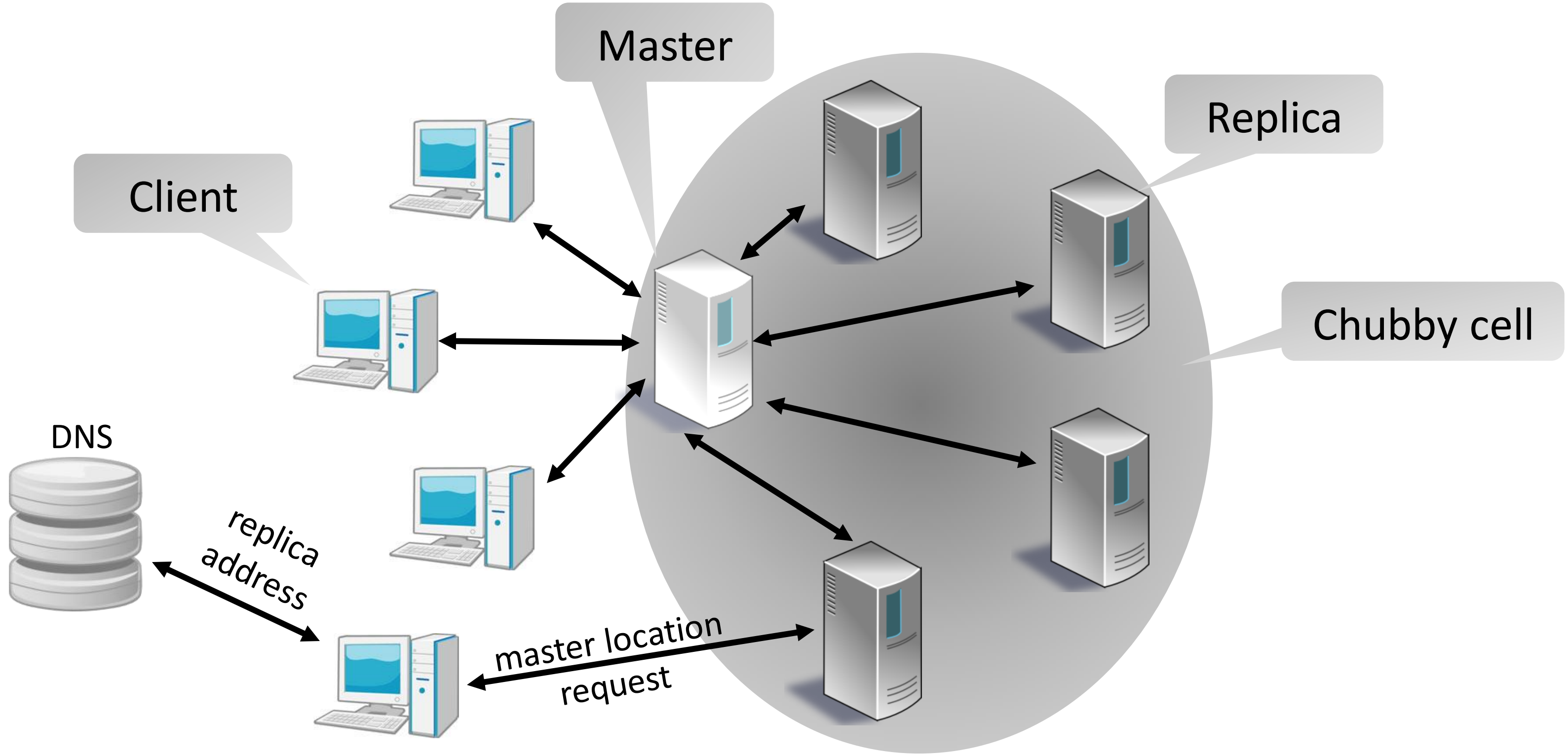
  “was quite annoyed at how humorless everyone working in the field seemed to be”
- After a few years, some people started to understand the importance of the algorithm
- After eight years, Leslie Lamport submitted the paper again, basically unaltered. It got accepted!

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)

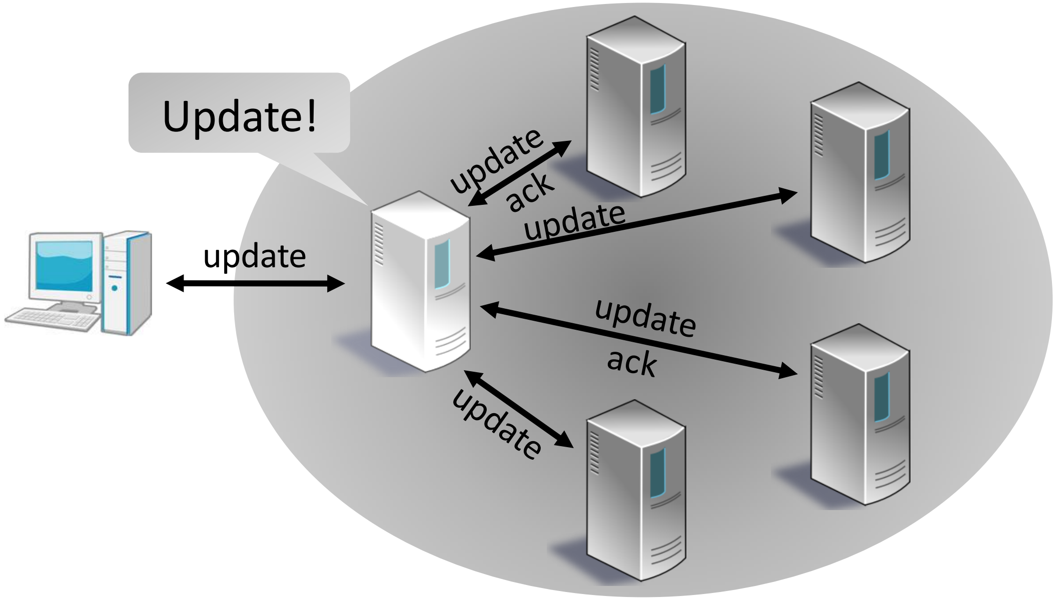
Chubby: System Structure

- 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

- 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

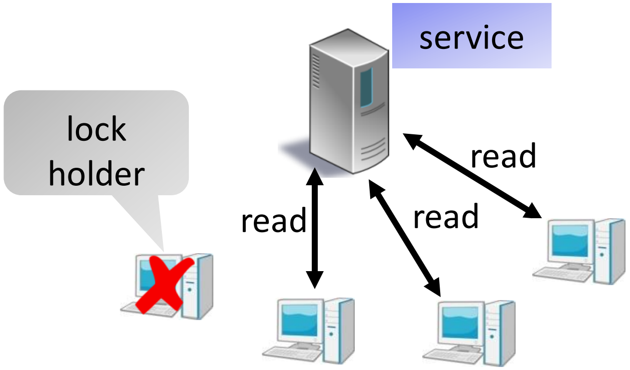
- The master remains the master for the duration of the master lease
  - Before the lease expires, the master can renew it (and remain the master)
  - It is guaranteed that no new master is elected before the lease expires
  - However, a new master is elected as soon as the lease expires
  - This ensures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

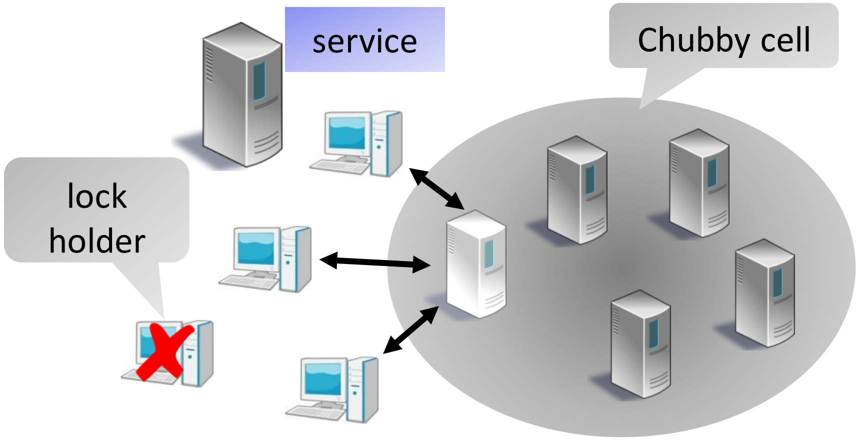
Chubby: Locks

- 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:

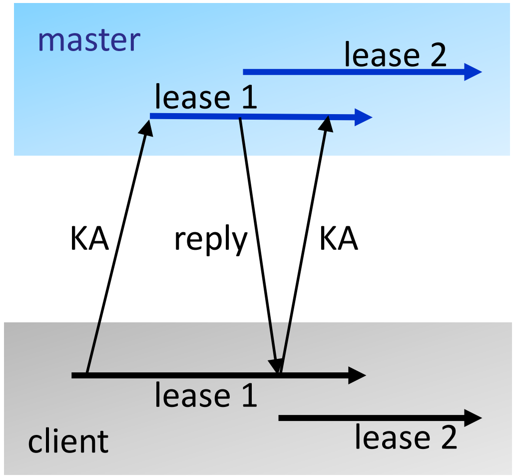


Mandatory:



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

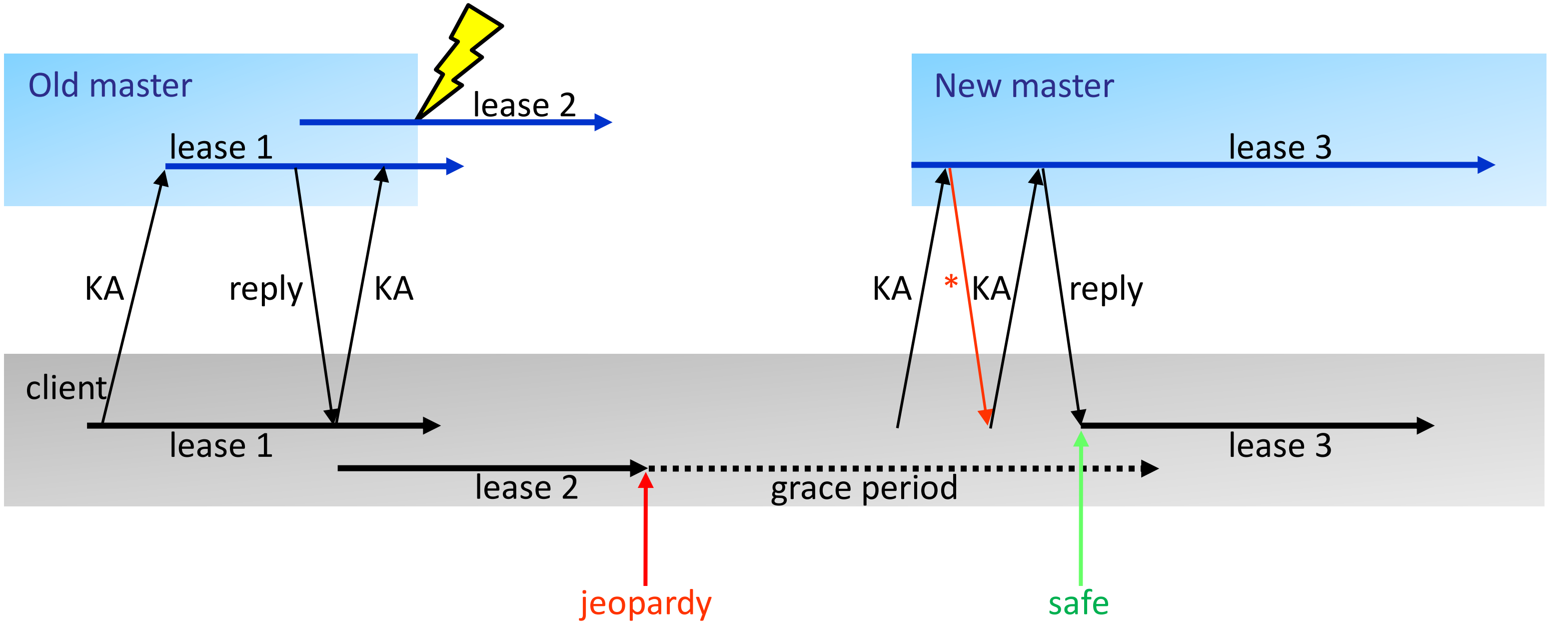
- The client maintains a local lease timeout
  - The client knows (roughly) when it has to hear from the master again
- If the local lease expires, the session is in jeopardy
- As soon as a session is in jeopardy, the grace period (45s by default) starts
  - If there is a successful KeepAlive exchange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

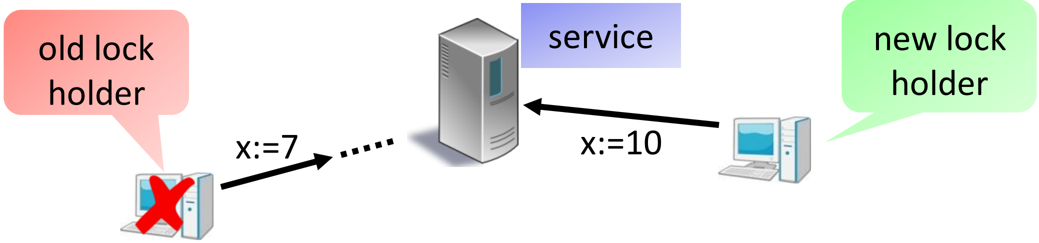
Chubby: Master Failure

- 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 inform all clients about failure → Clients flush cache
  - All operations can proceed

We omit caching in this lecture!

Chubby: Locks Reloaded

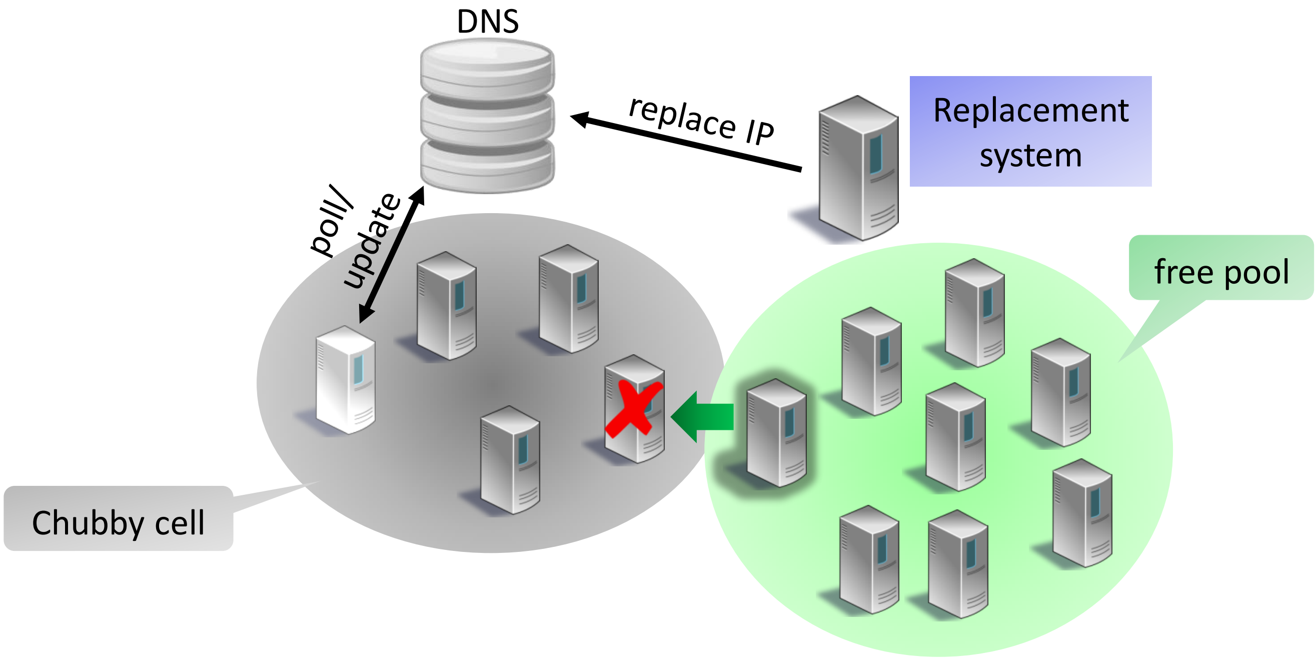
- What if a lock holder crashes and its (write) request is still in transit?
  - This write may undo an operation of the next lock holder!



- 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

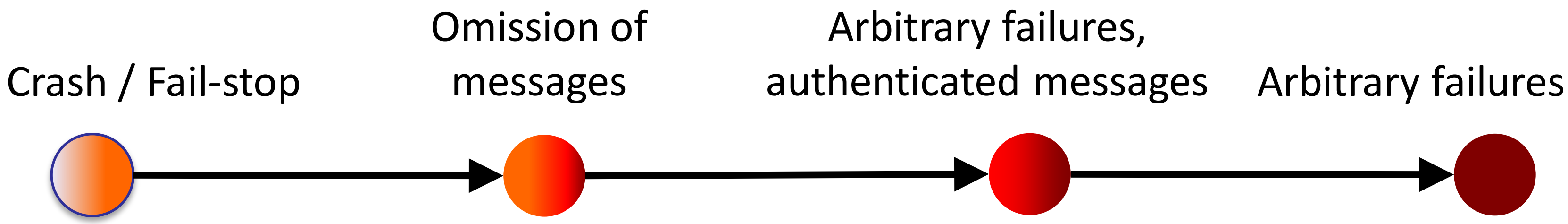


Chubby: Performance

- 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

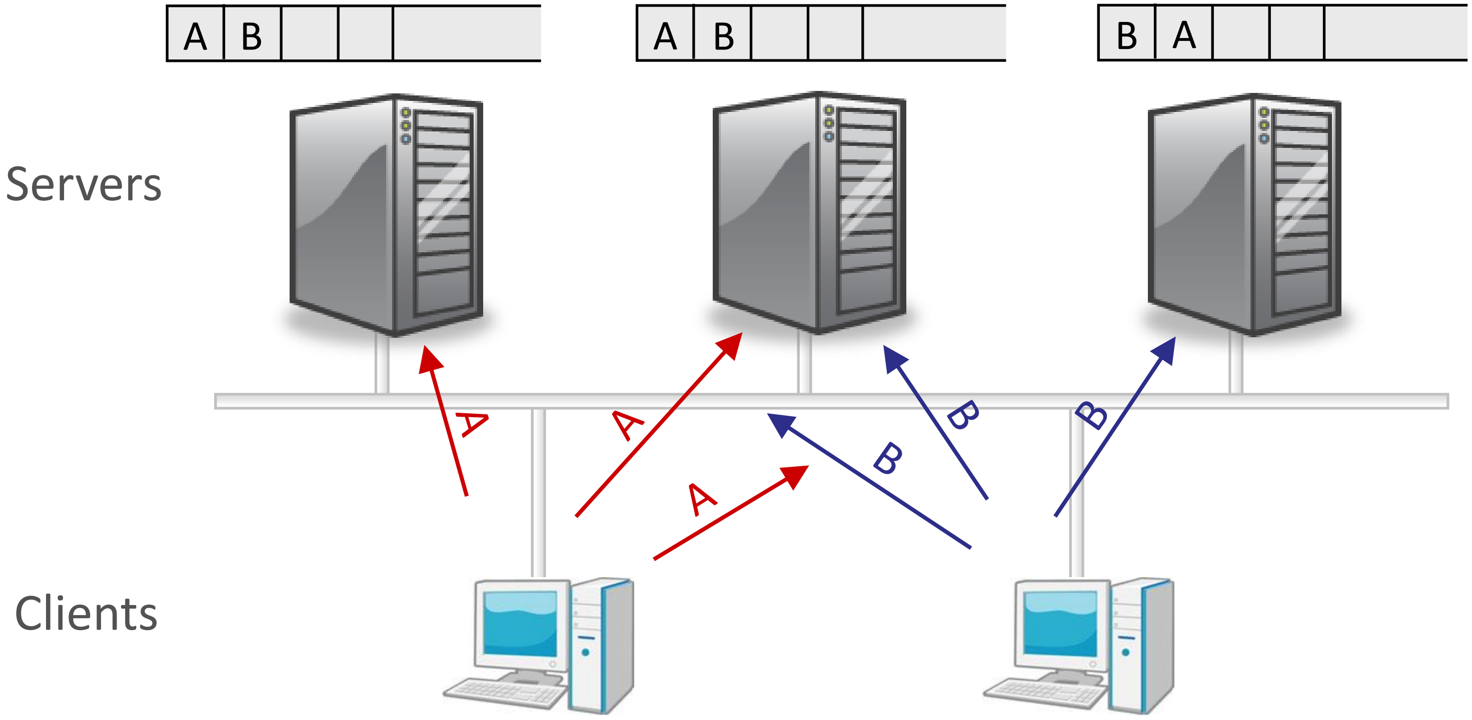
- We are now going to study the Practical Byzantine Fault-Tolerant (PBFT) system
- The system consists of clients that read/write data stored at n servers

- 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

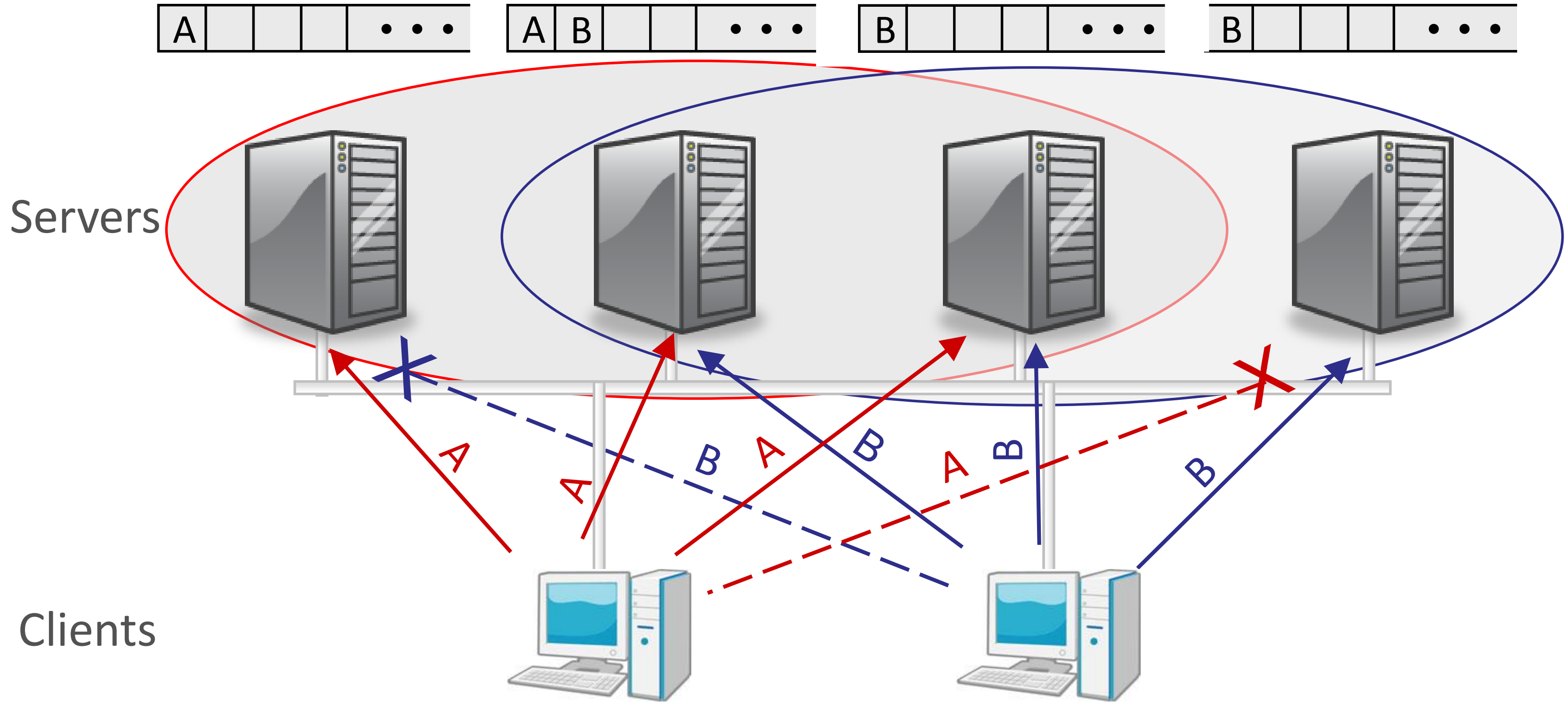
PBFT: Order of Operations

- 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 decentralized like Paxos!

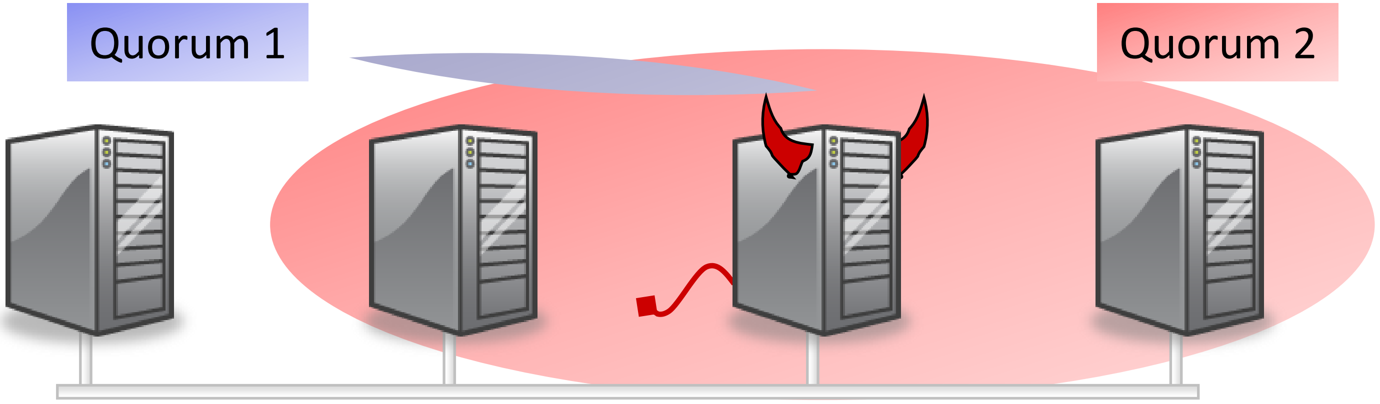
Quorum…?

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

Quorums

- In law, a quorum is a the minimum number of members of a deliberative body necessary to conduct the business of the group
  - In a majority vote system, e.g., any majority is a quorum

- In our case, 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 server



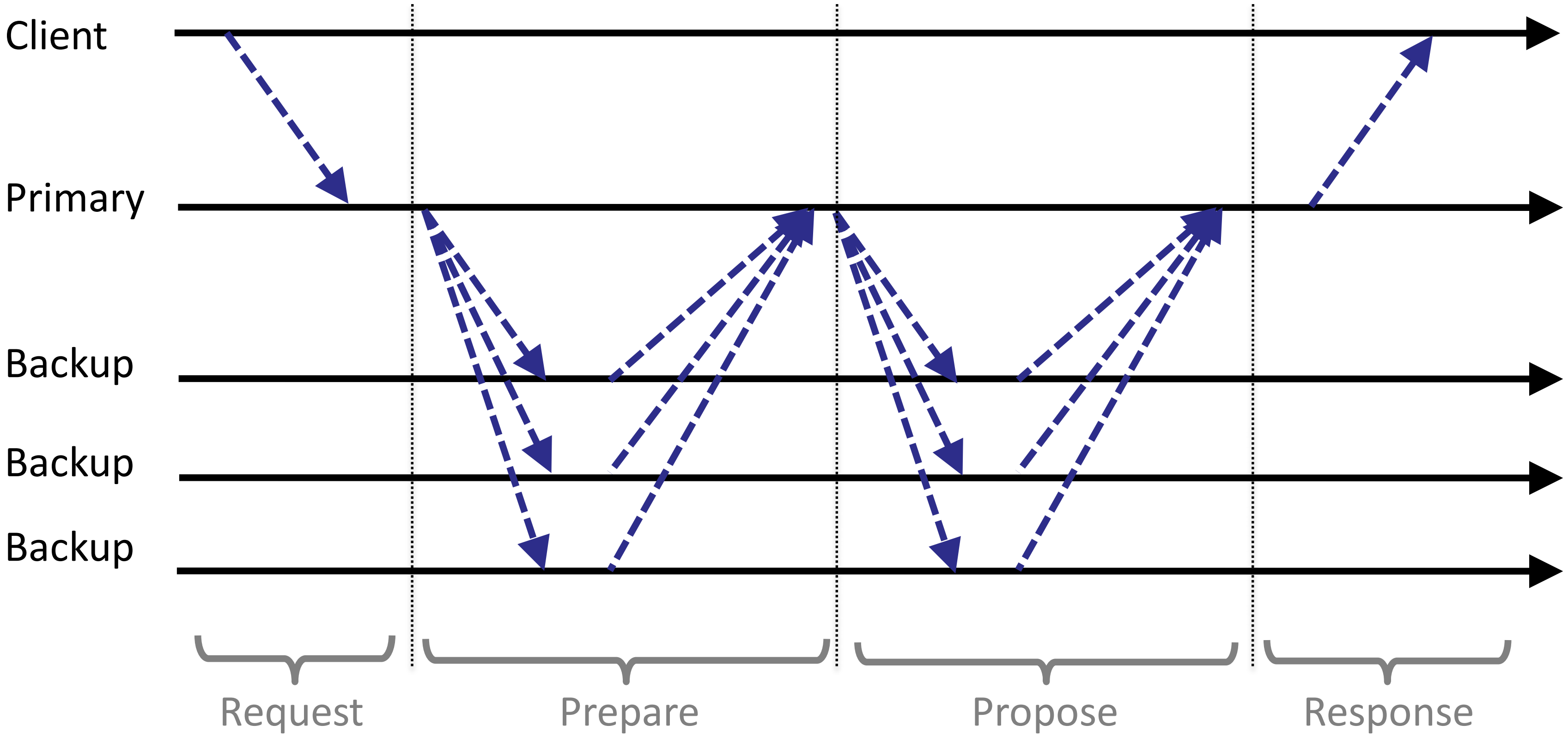
PBFT: Main Algorithm

- 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…

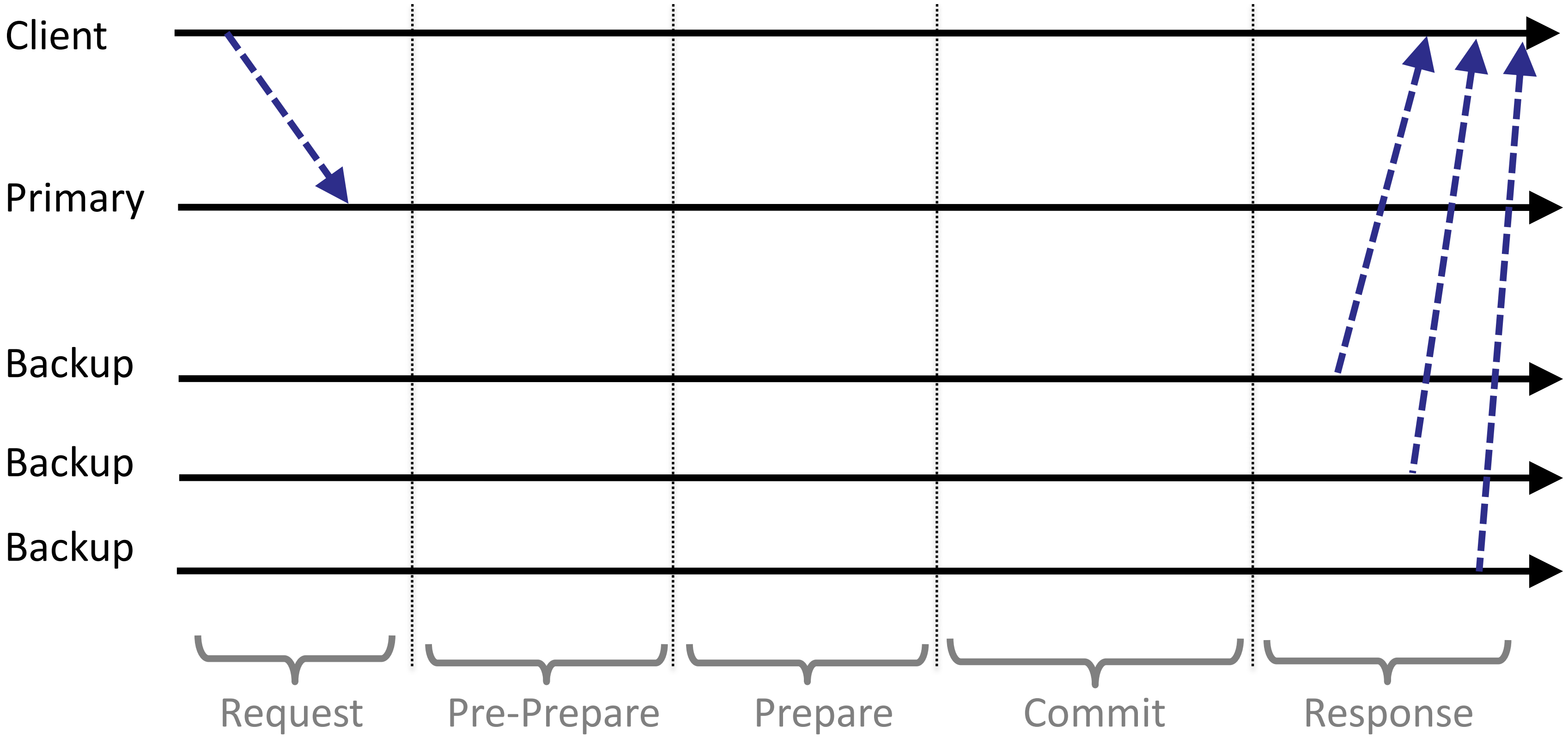
PBFT: Paxos

- 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



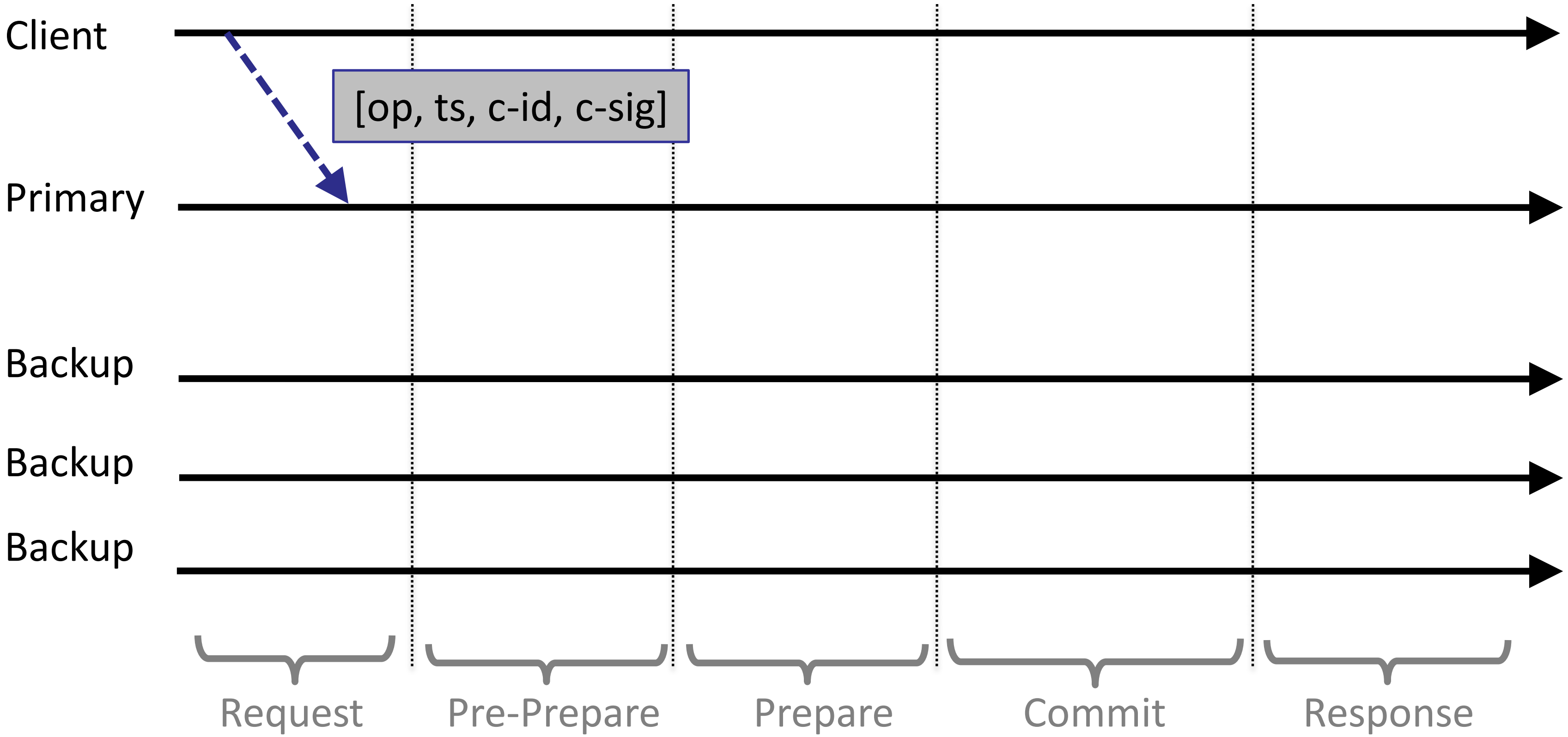
PBFT: Algorithm

- 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



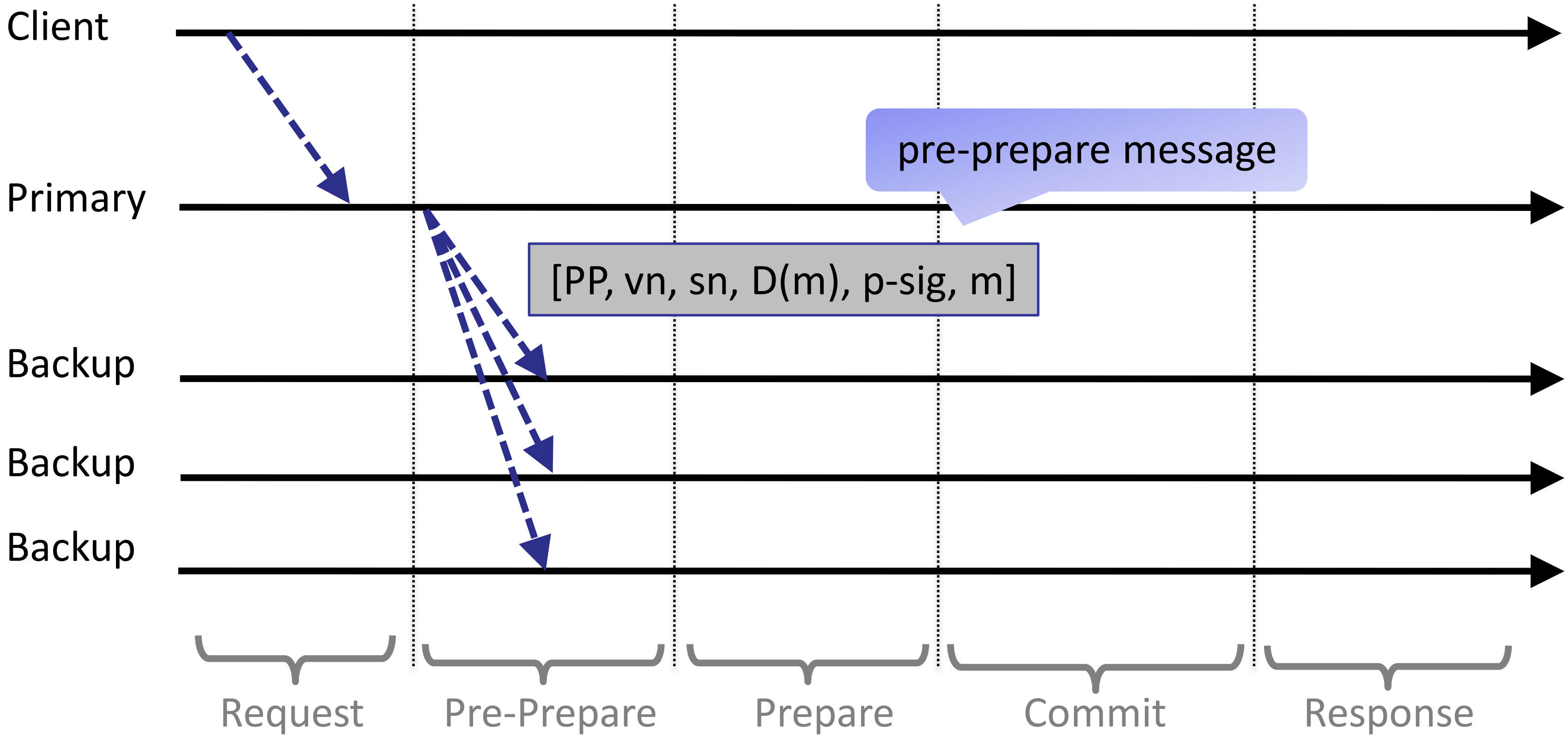
PBFT: Request Phase

- 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, cid, 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…

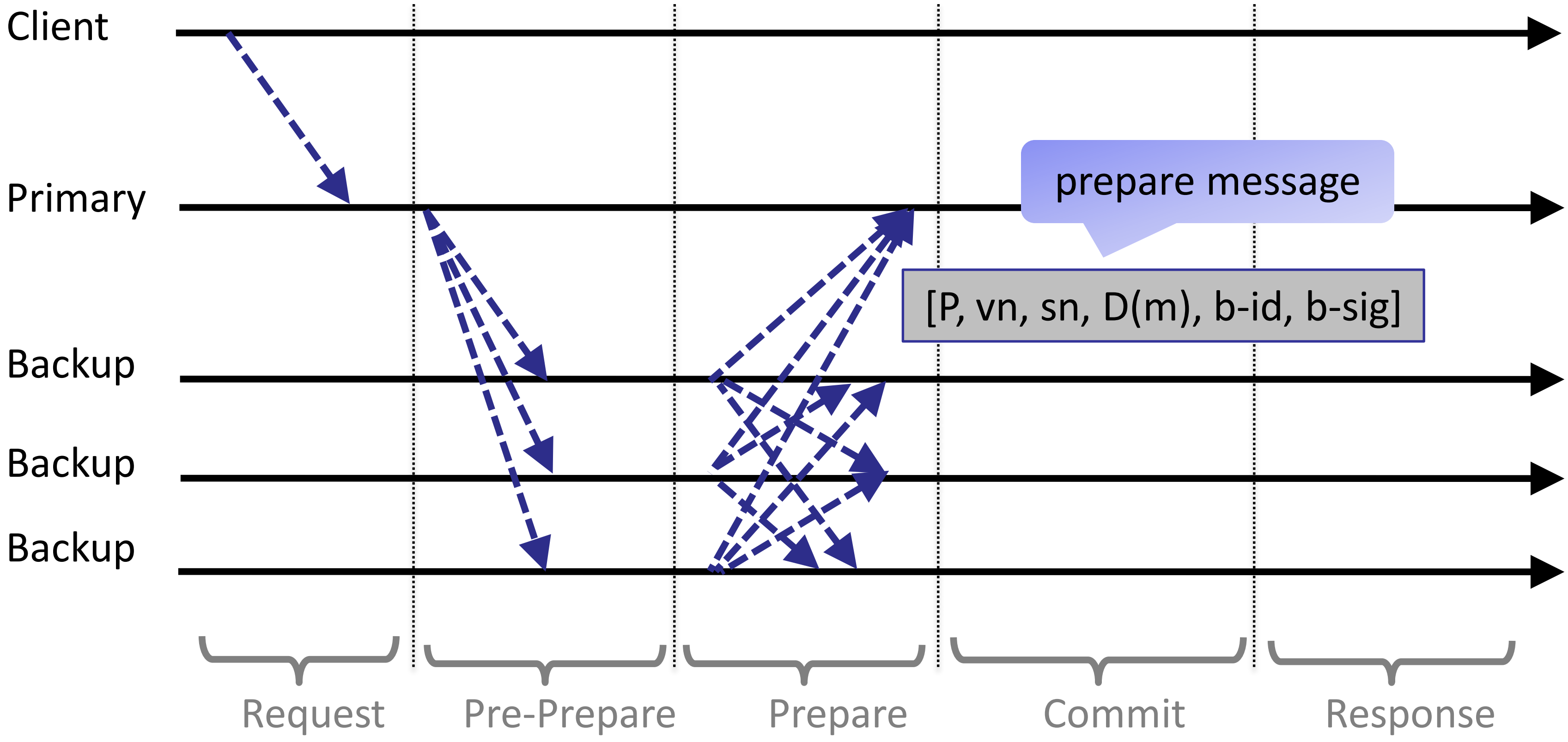
PBFT: Pre-Prepare Phase

- 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



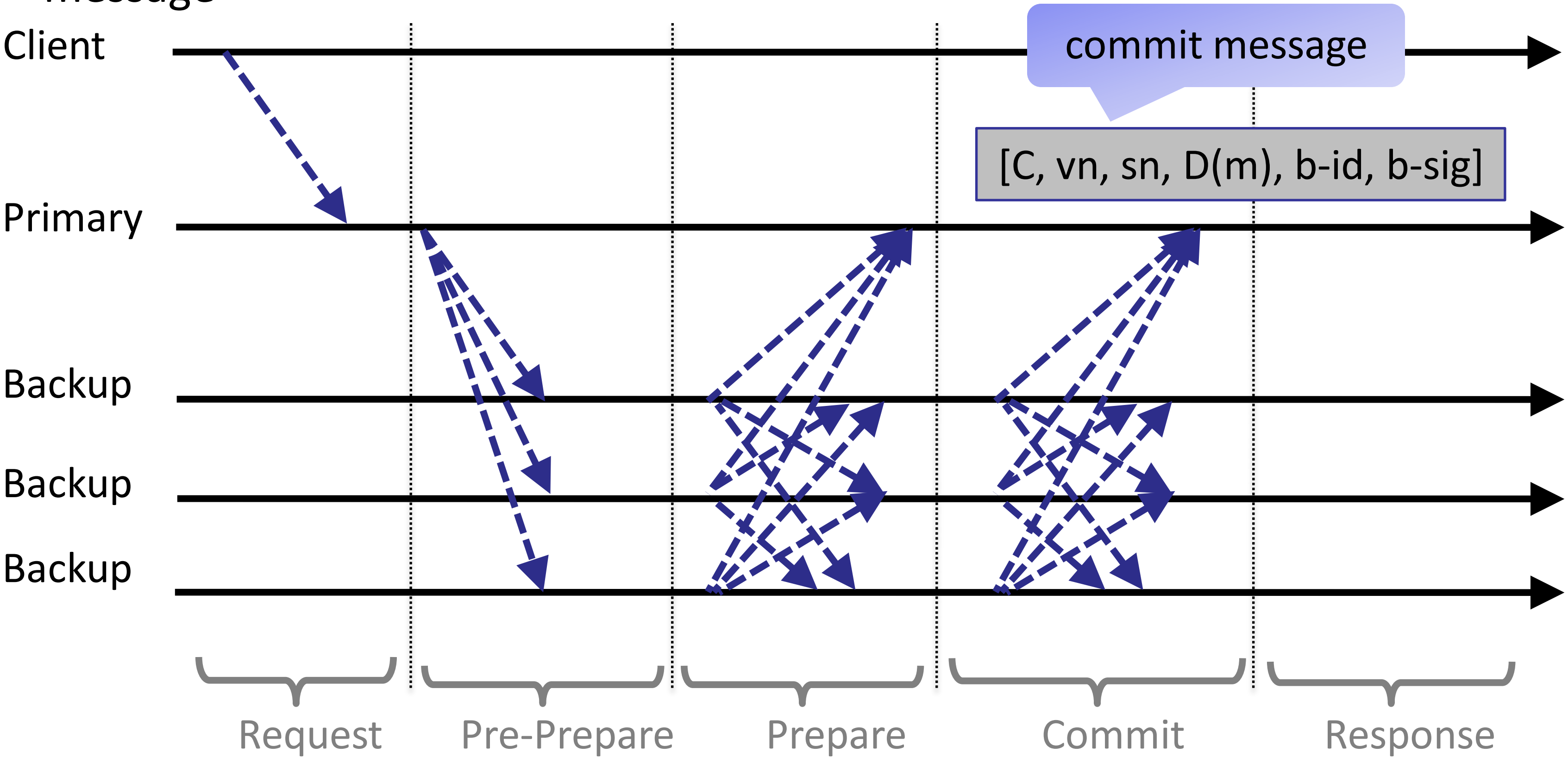
PBFT: Prepare Phase

- 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



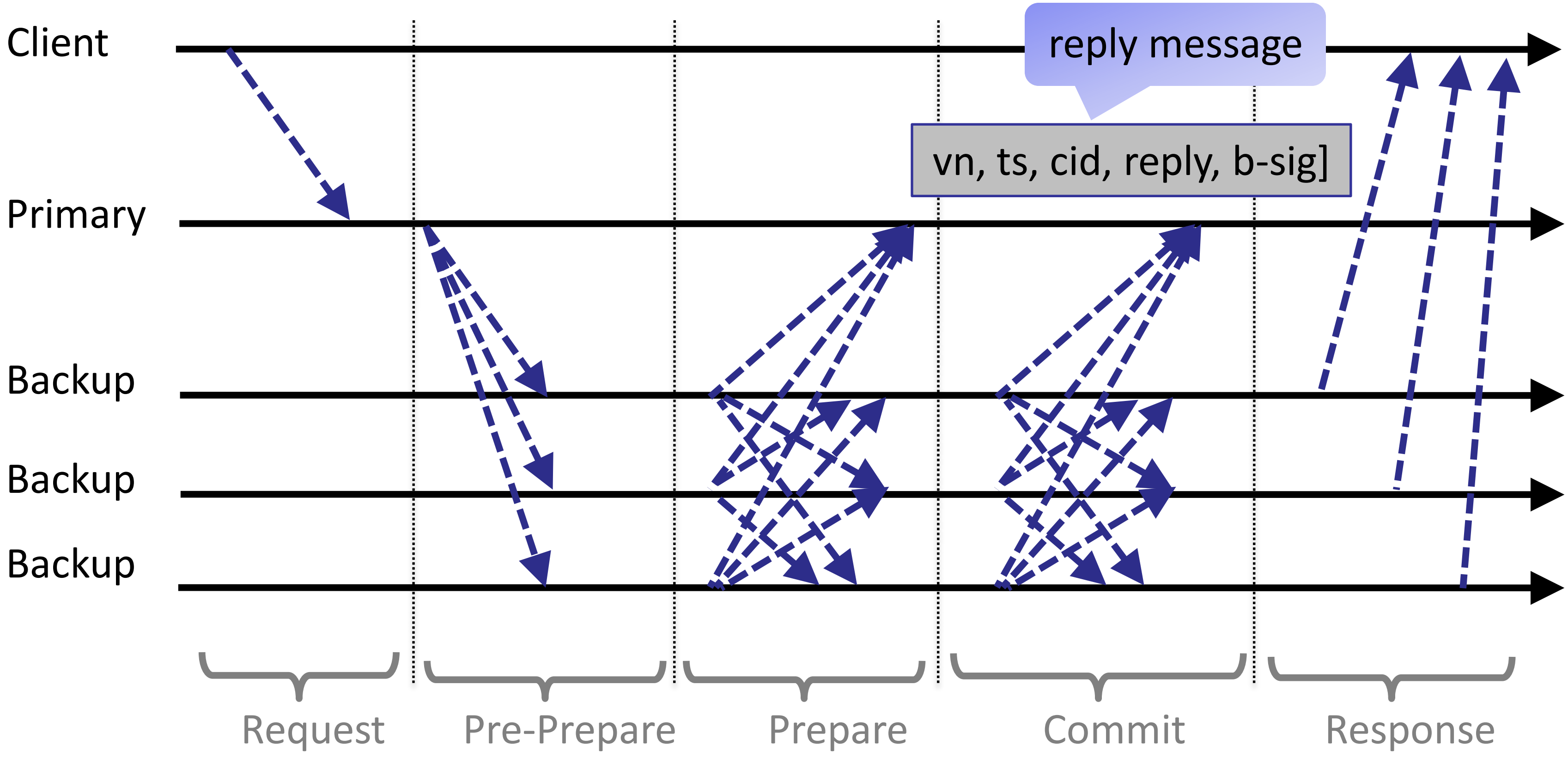
PBFT: Commit Phase

- 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



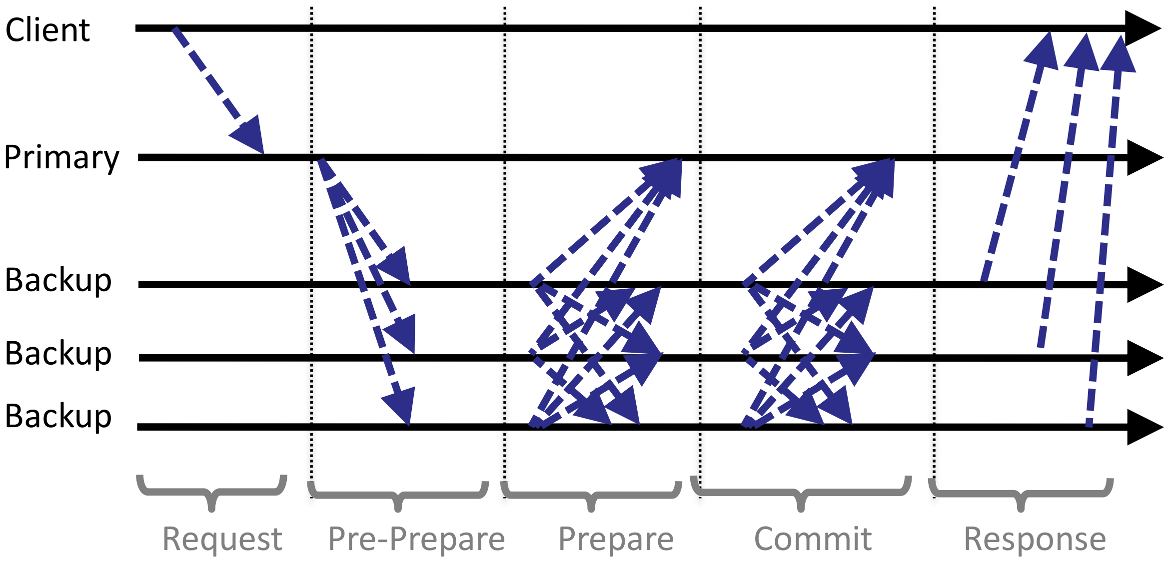
PBFT: Garbage Collection

- 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

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 $2f+1$ accepted signatures

The next primary!

- 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≠ 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

Theorem

PBFT terminates eventually

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

Theorem

PBFT terminates eventually

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:

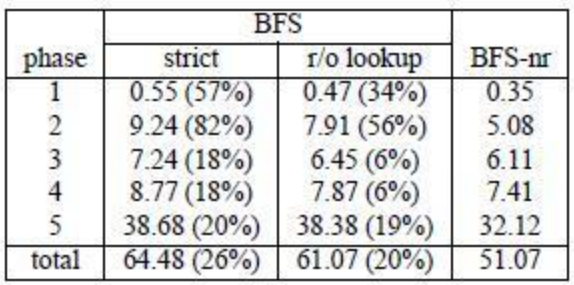
1. Create subdirectories recursively
2. Copy a source tree
3. Examine the status of all the files in the tree without examining the data
4. Examine every byte in all the files
5. Compile and link the files

- 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

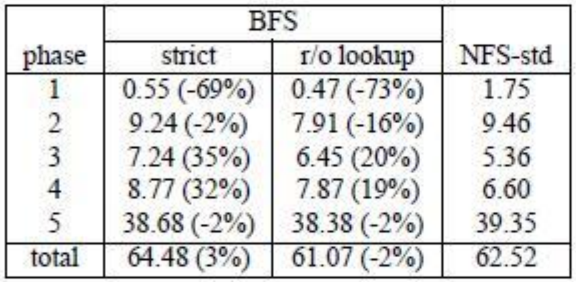
PBFT: Evaluation

- 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
- 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

Times are in seconds



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

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 impossibiliy 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?

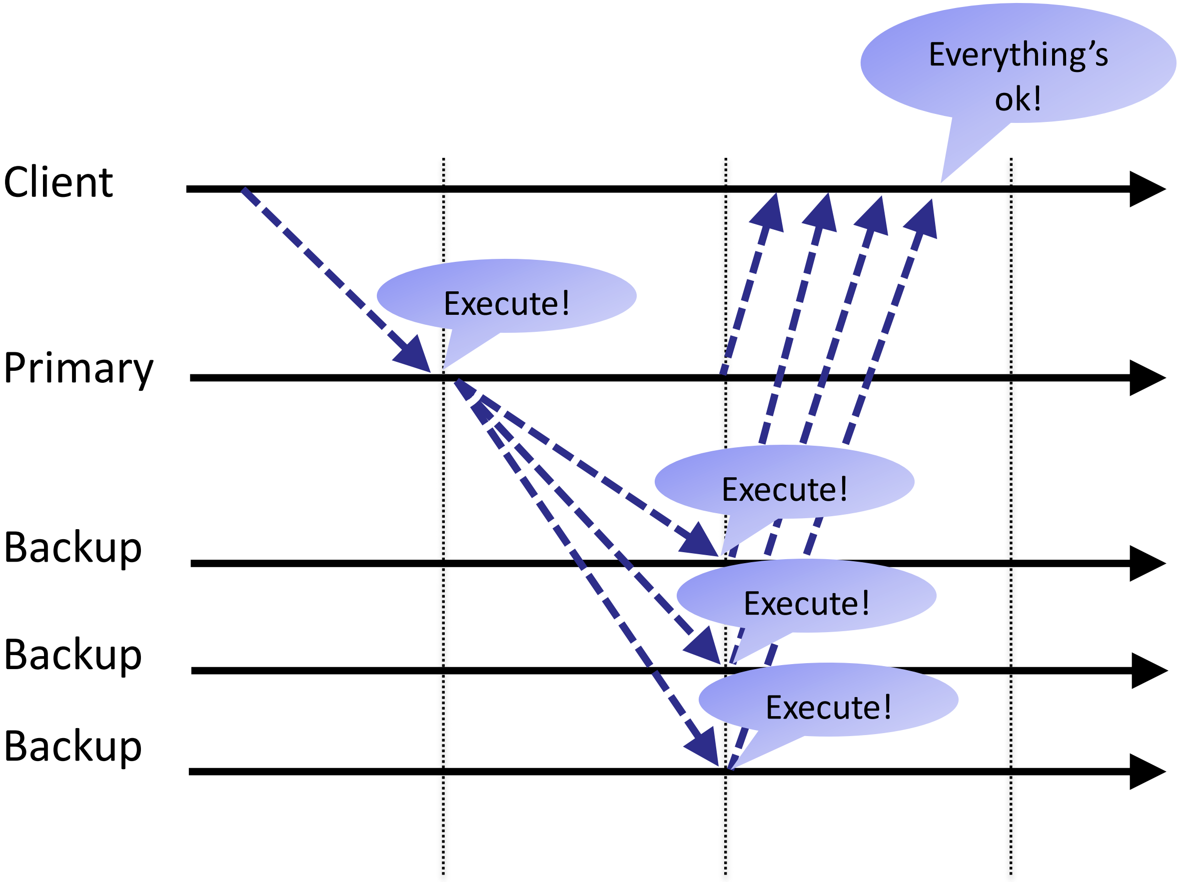
Zyzzyva

- 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

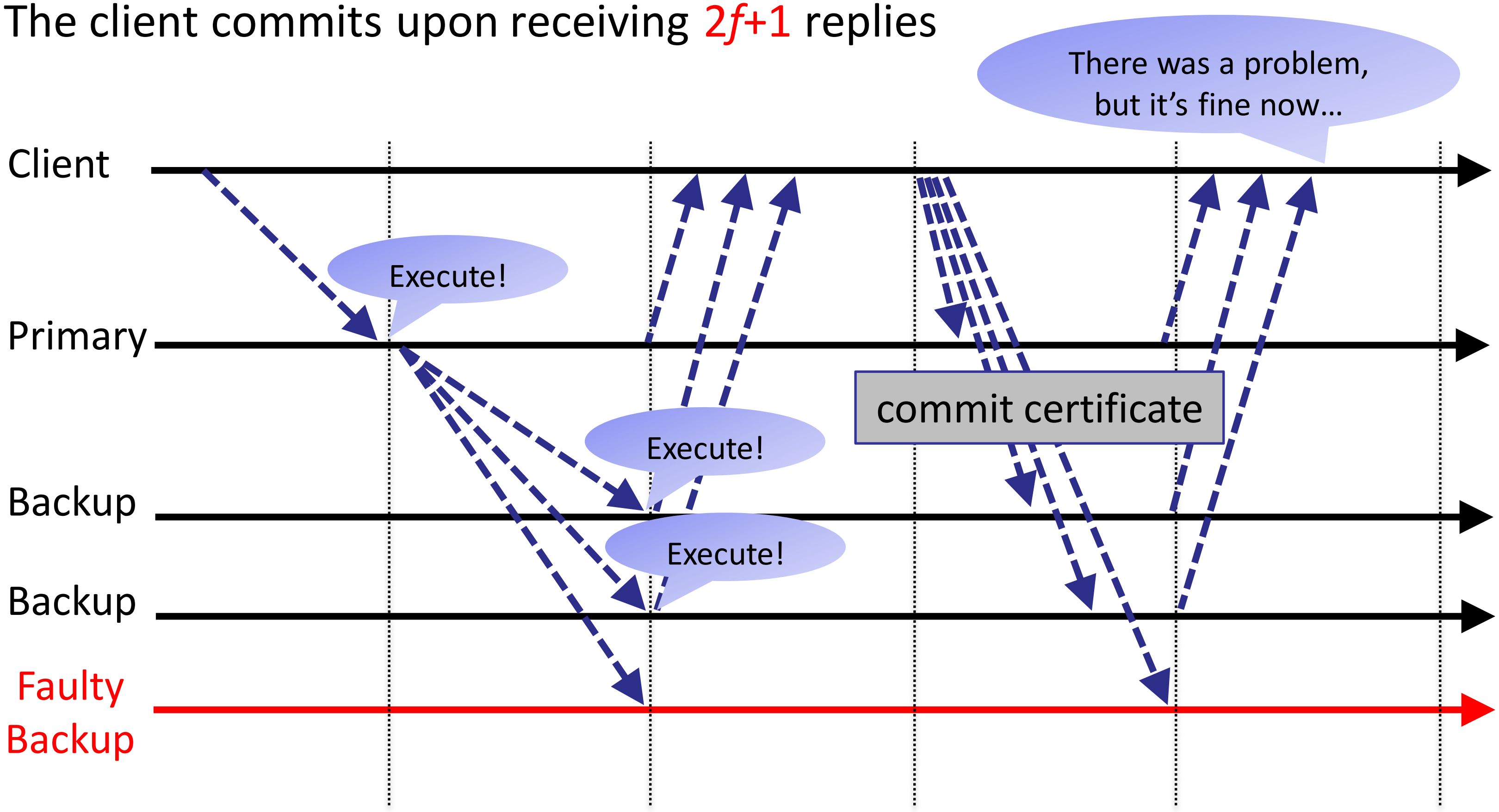
Zyzzyva

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



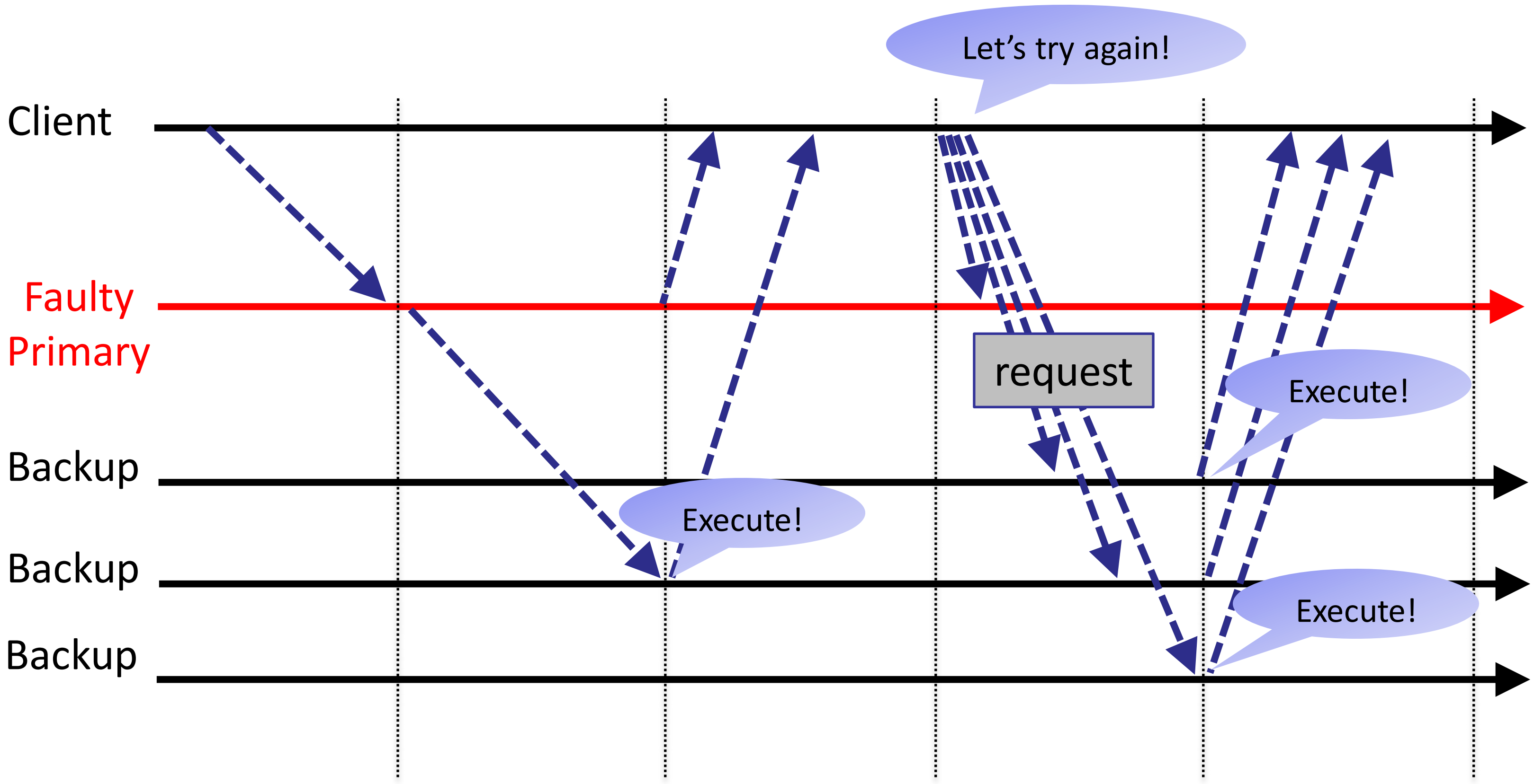
Zyzzyva

- 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



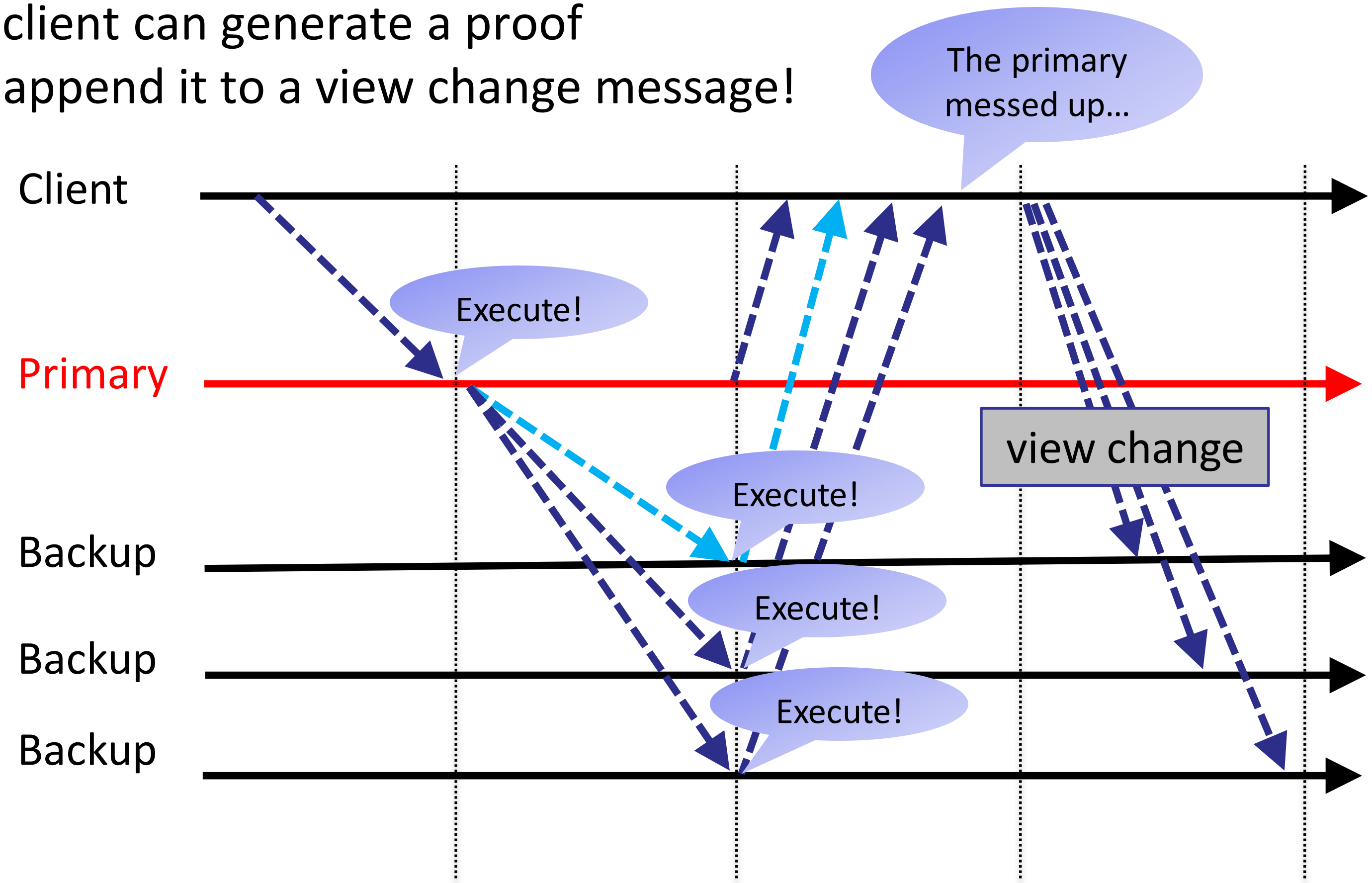
Zyzzyva

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



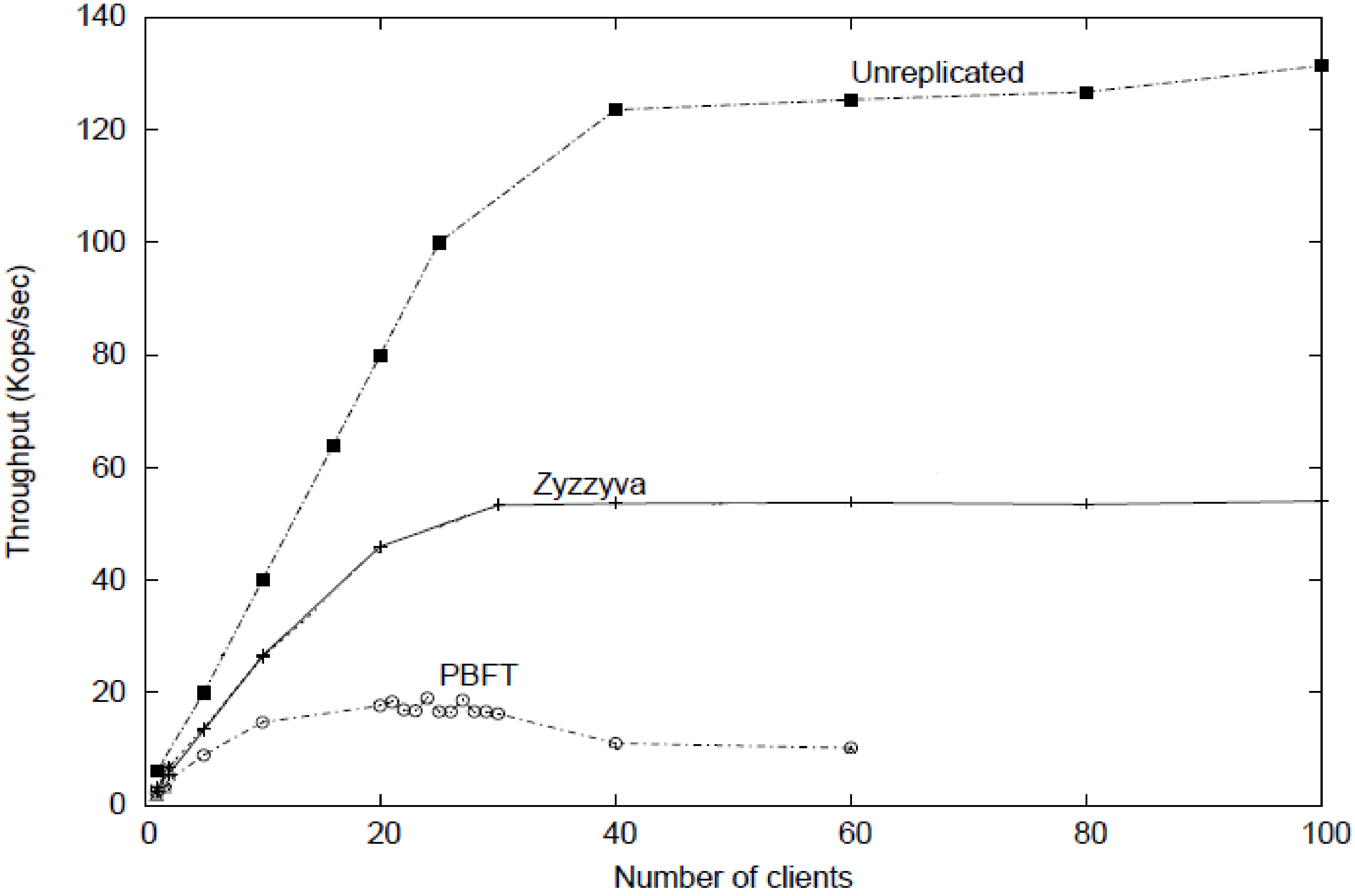
Zyzzyva

- 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!



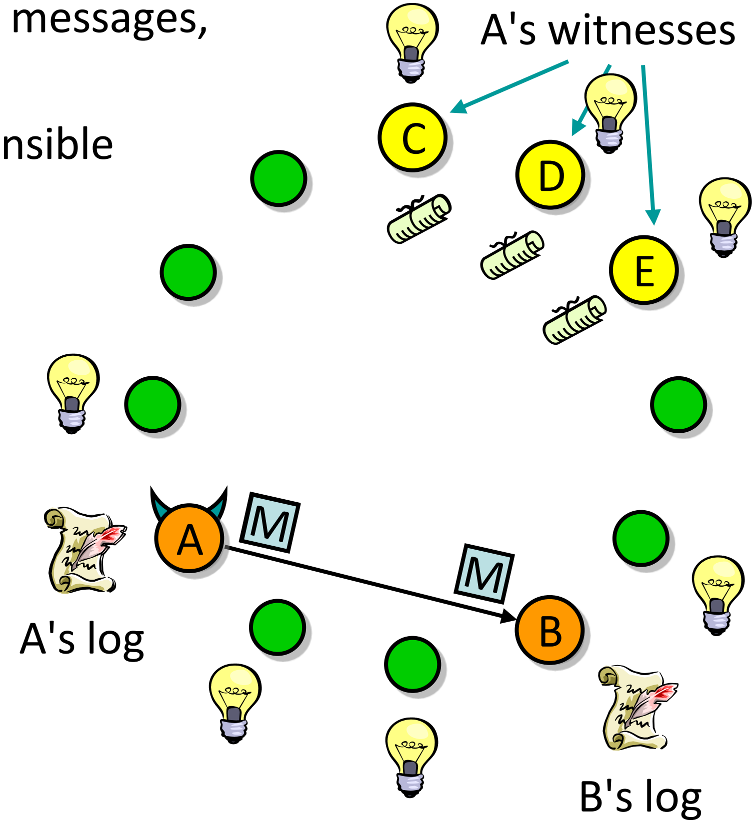
Zyzzyva: Evaluation

- 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!

Large-Scale Fault-Tolerant Systems

- The systems discussed so far have one thing in common:
  They do not scale!
  - More and larger messages have to be exchanged when the size of the systems increases
- Is it possible to create an efficient fault-tolerant system consisting of 1k, 10k,…, 1M nodes?

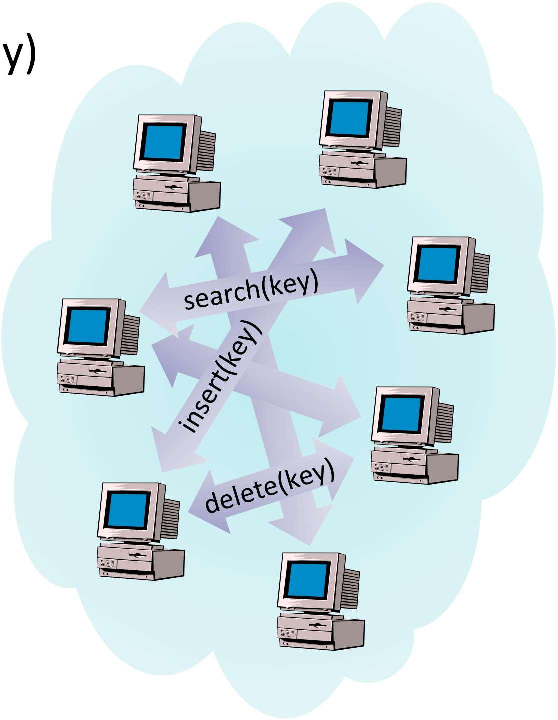
- Idea
  - Instead of a primary- (or view-)based approach, use a completely decentralized system
  - Each node in the system has the same rights and the same power as its other "peers"
  - This networking paradigm is called peer-to-peer (P2P) computing
- Note that this paradigm/model is completely different
  from what we studied on the previous 100+ slides!

P2P: Distributed Hash Table (DHT)

- Data objects are distributed among the peers
  - Each object is uniquely identified by a key
- Each peer can perform certain operations
  - Search(key) (returns the object associated with key)
  - Insert(key, object)
  - Delete(key)

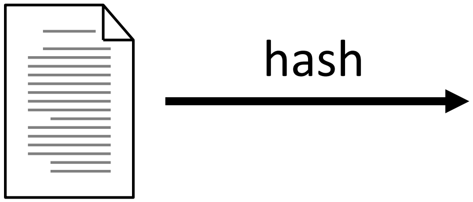
- Classic implementations of these operations
  - Search Tree (balanced, B-Tree)
  - Hashing (various forms)

- "Distributed" implementations
  - Linear Hashing
  - Consistent Hashing



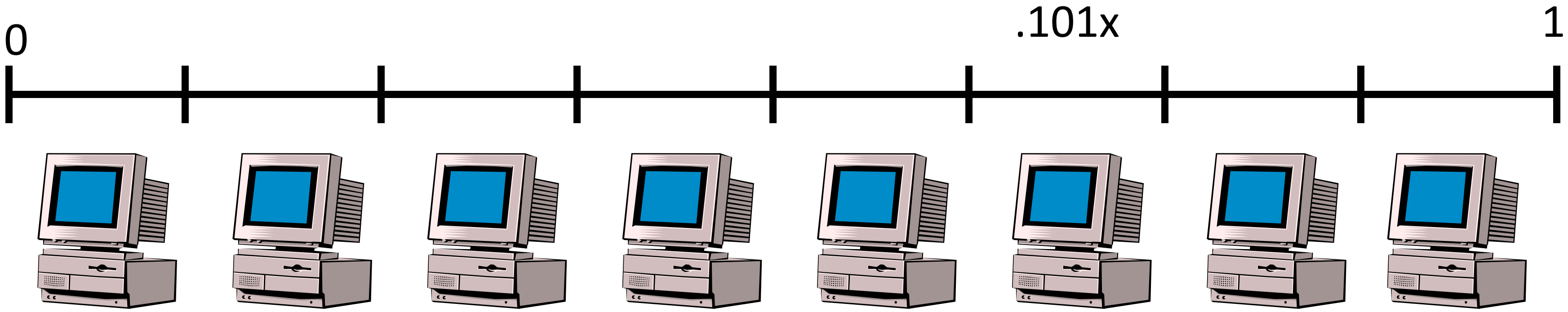
Distributed Hashing

- The hash of a file is its key



.1011101010111 0011… ≈ .73

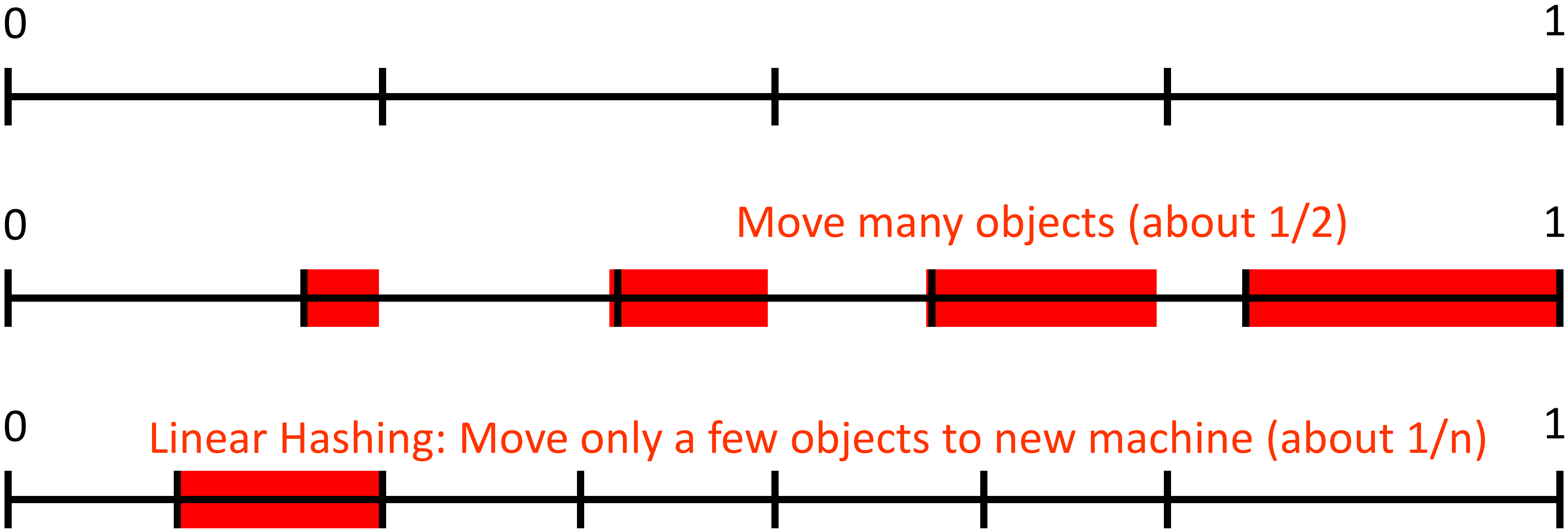
- Each peer stores data in a certain range of the ID space [0,1]



- Instead of storing data at the right peer, just store a forward-pointer

Linear Hashing

- Problem: More and more objects should be stored → Need to buy new machines!
- Example: From 4 to 5 machines

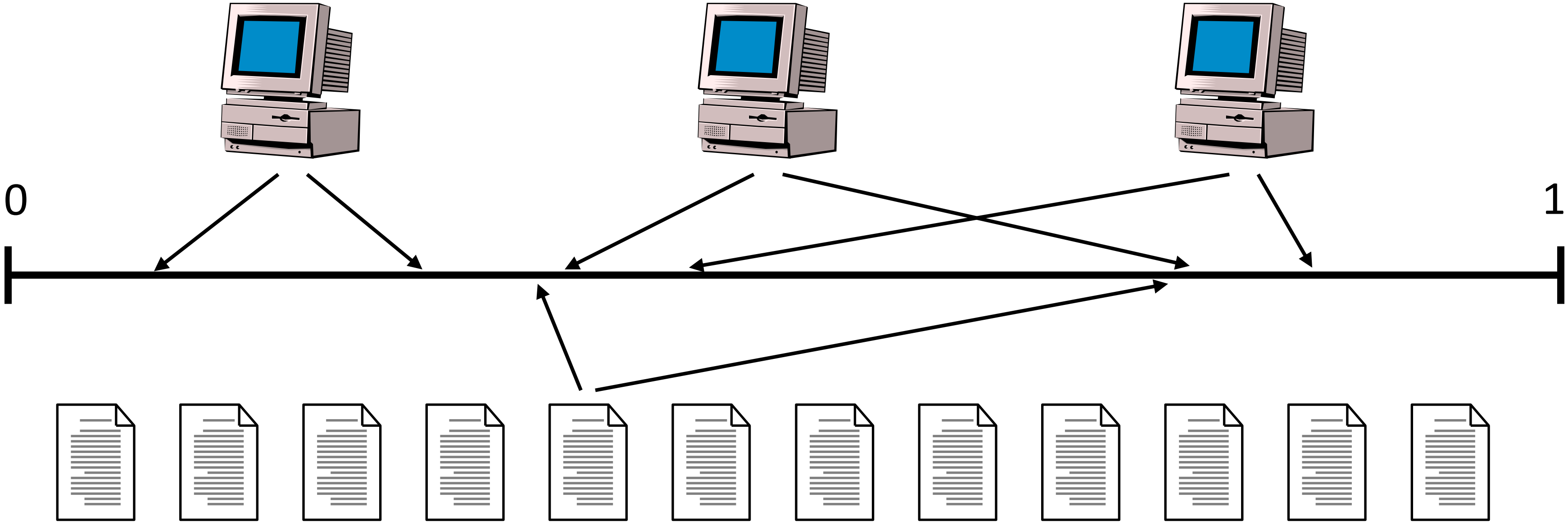


Move many objects (about 1/2)

Linear Hashing: Move only a few objects to new machine (about 1/n)

Consistent Hashing

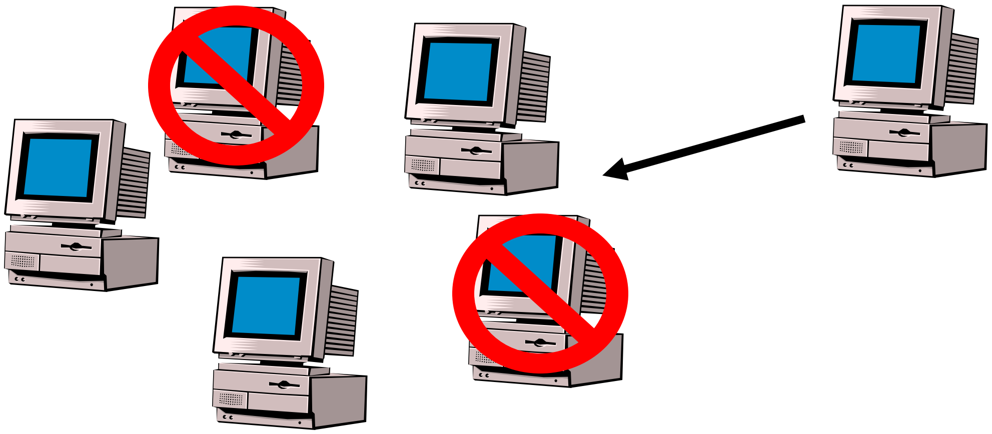
- Linear hashing needs central dispatcher
- Idea: Also the machines get hashed! Each machine is responsible for the files closest to it
- Use multiple hash functions for reliability!



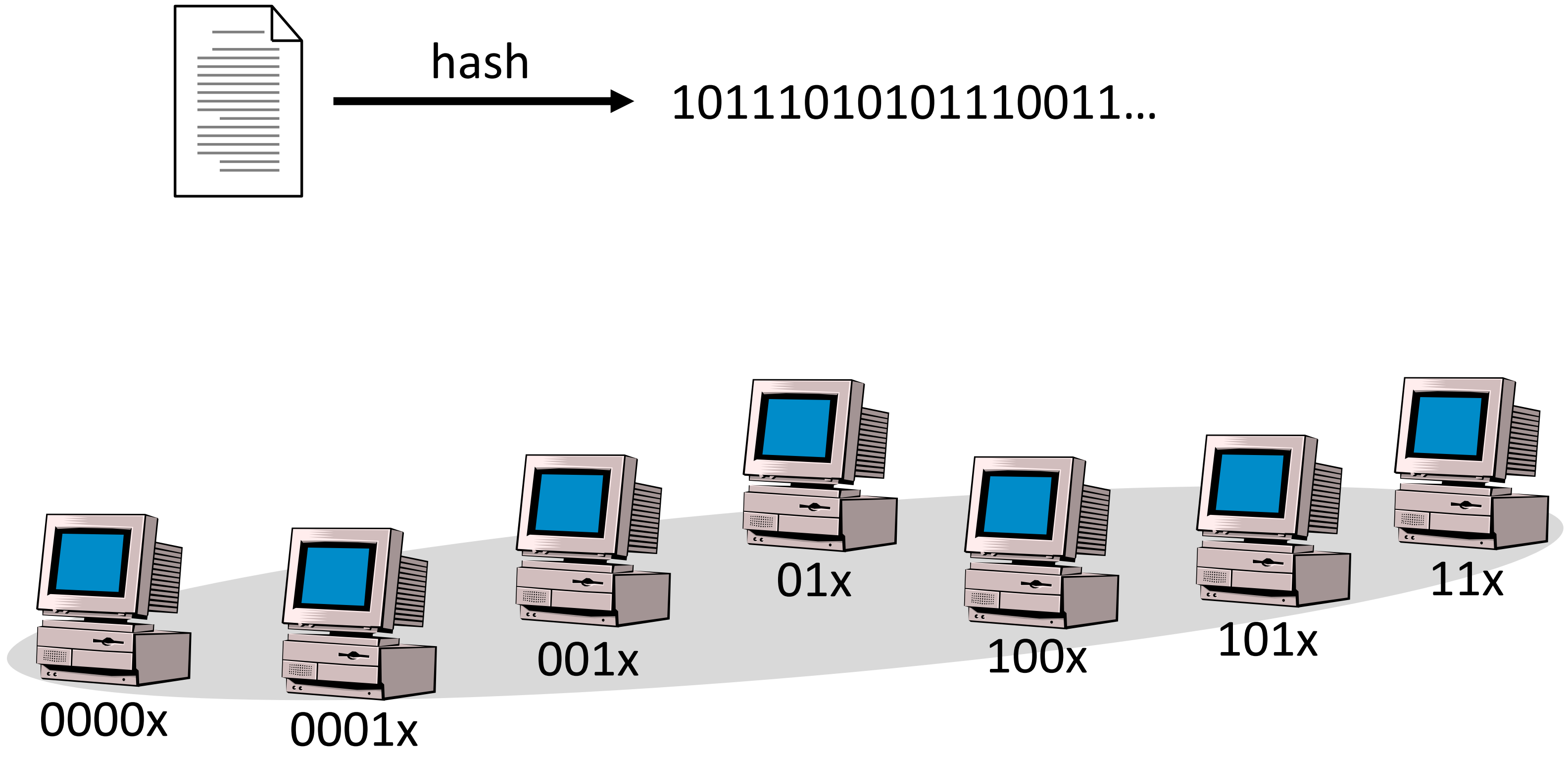
Search & Dynamics

- Problem with both linear and consistent hashing is that all the participants of the system must know all peers…
  - Peers must know which peer they must contact for a certain data item
  - This is again not a scalable solution…

- Another problem is dynamics!
  - Peers join and leave (or fail)



P2P Dictionary = Hashing



hash

10111010101110011…

0000x

0001x

001x

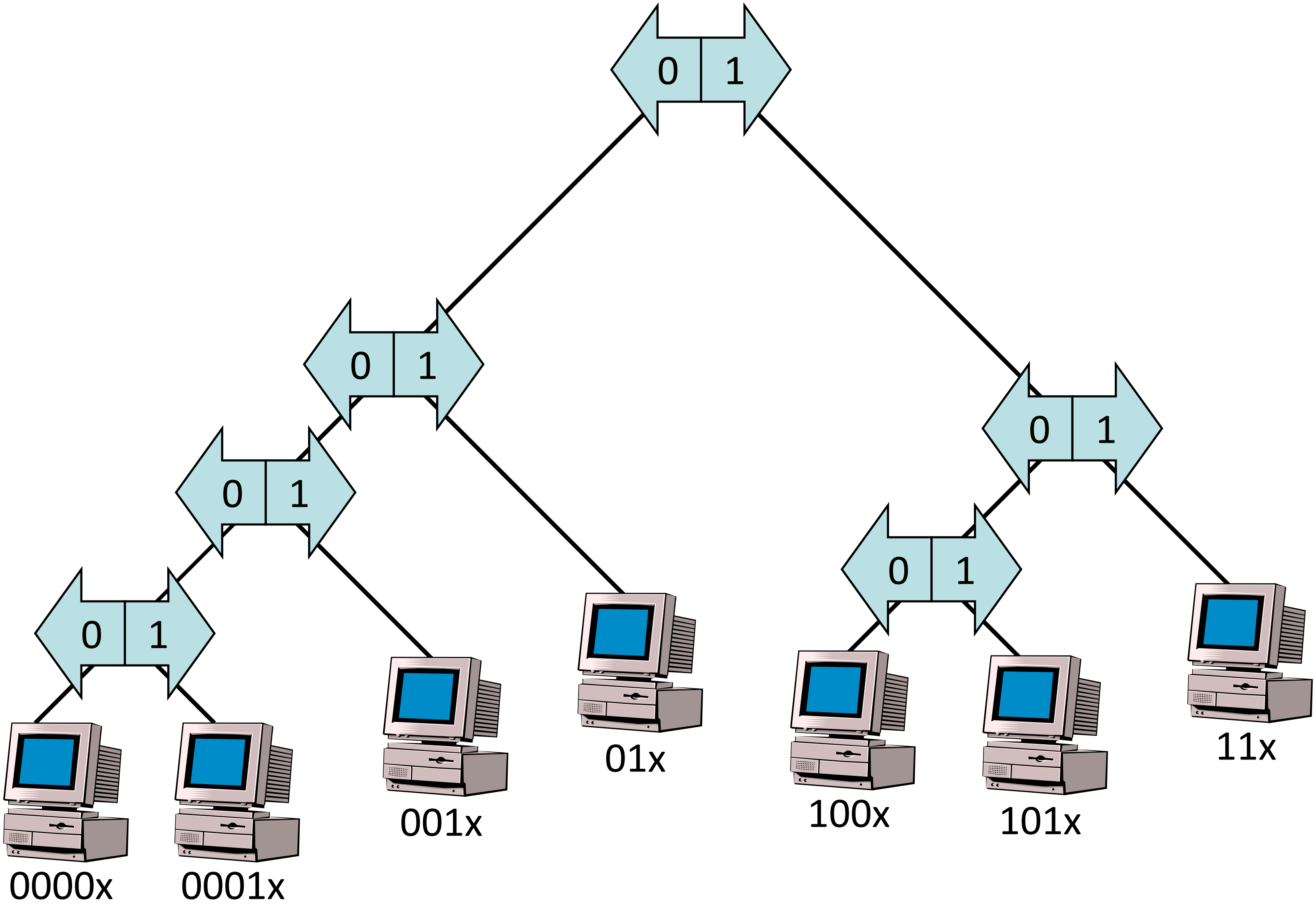
01x

100x

101x

11x

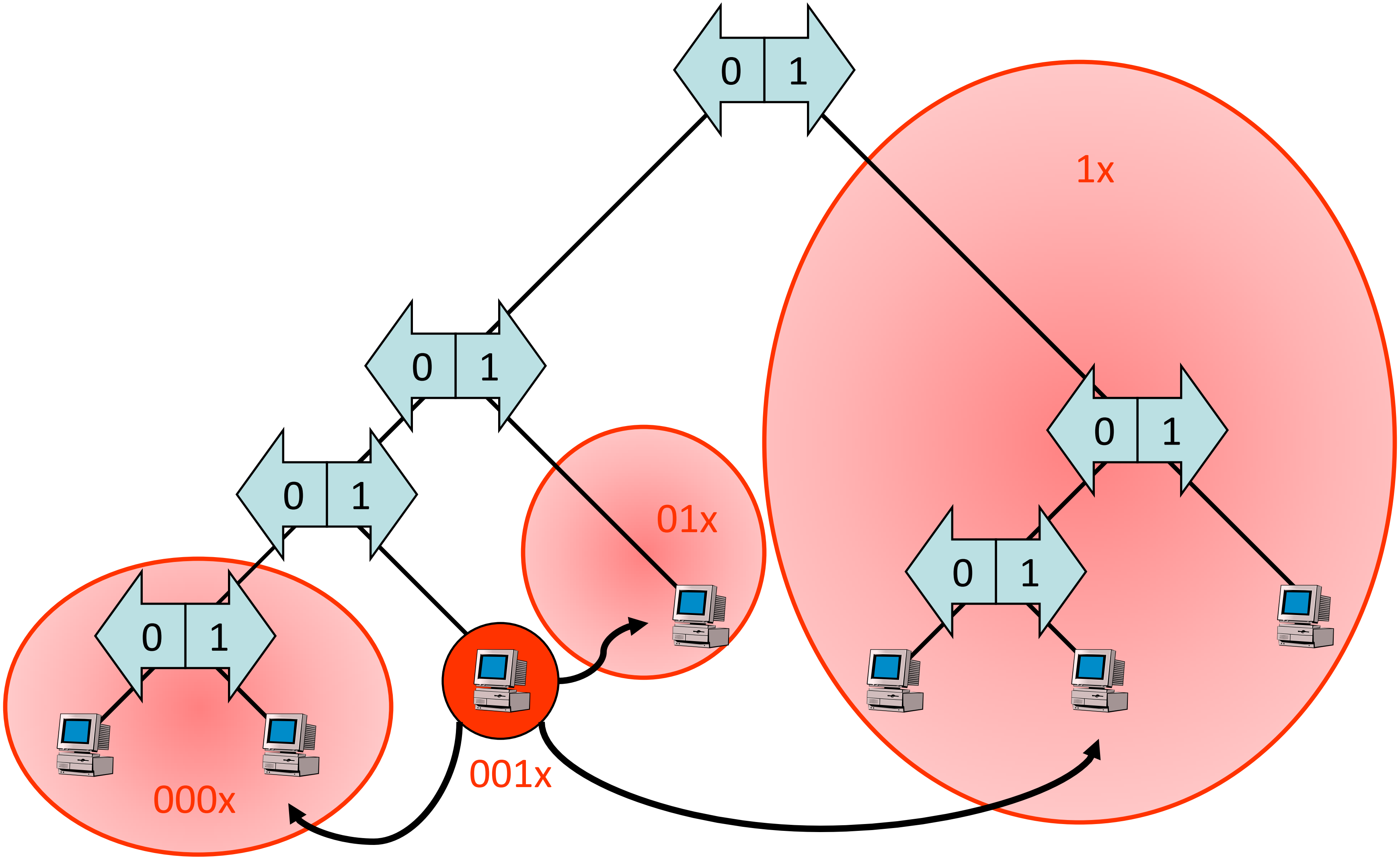
P2P Dictionary = Search Tree



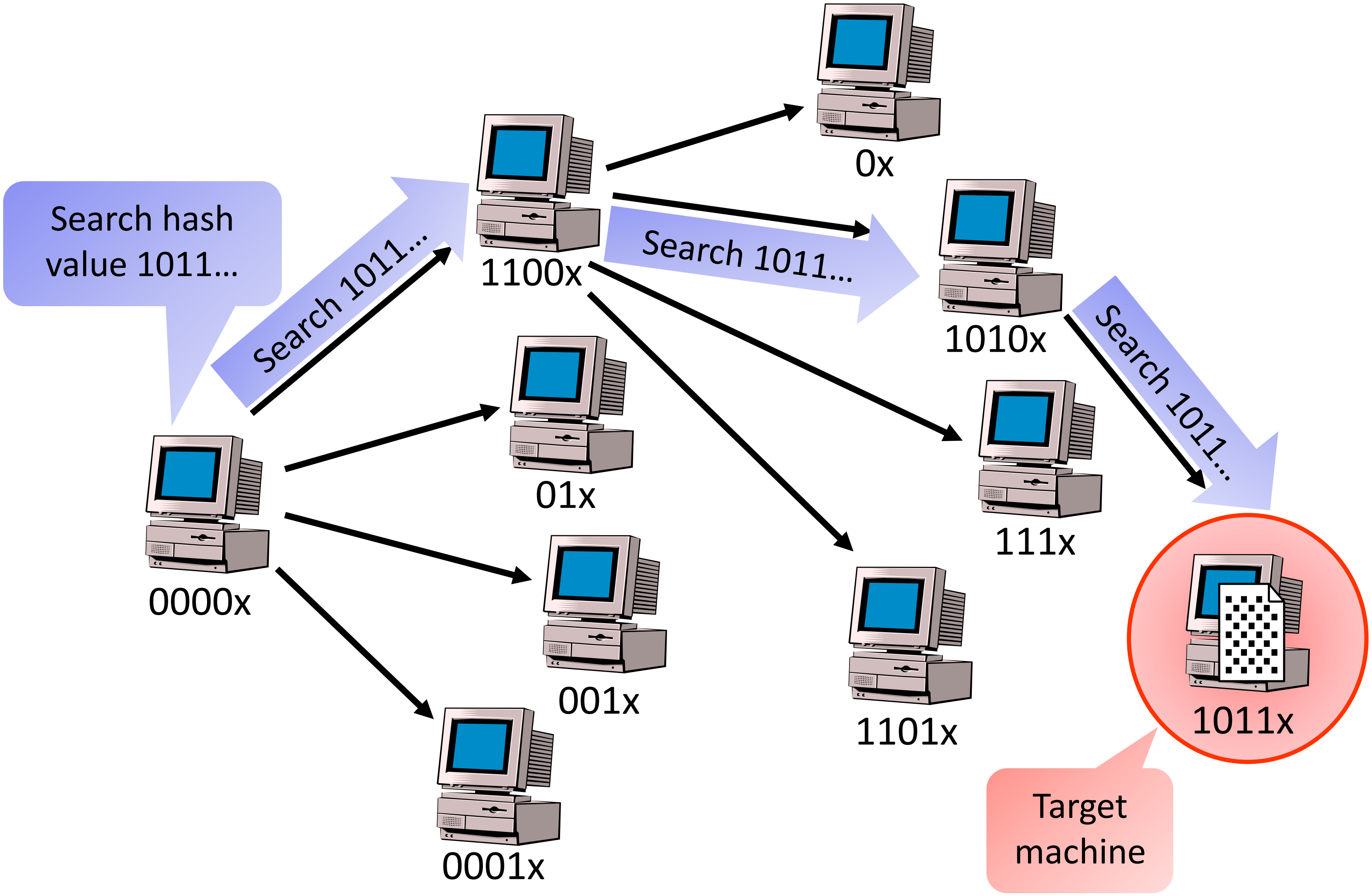
Storing the Search Tree

- Where is the search tree stored?
- In particular, where is the root stored?
  - What if the root crashes?! The root clearly reduces scalability & fault tolerance…
  - Solution: There is no root…!
- If a peer wants to store/search, how does it know where to go?
  - Again, we don't want that every peer has to know all others…
  - Solution: Every peer only knows a small subset of others

The Neighbors of Peers 001x

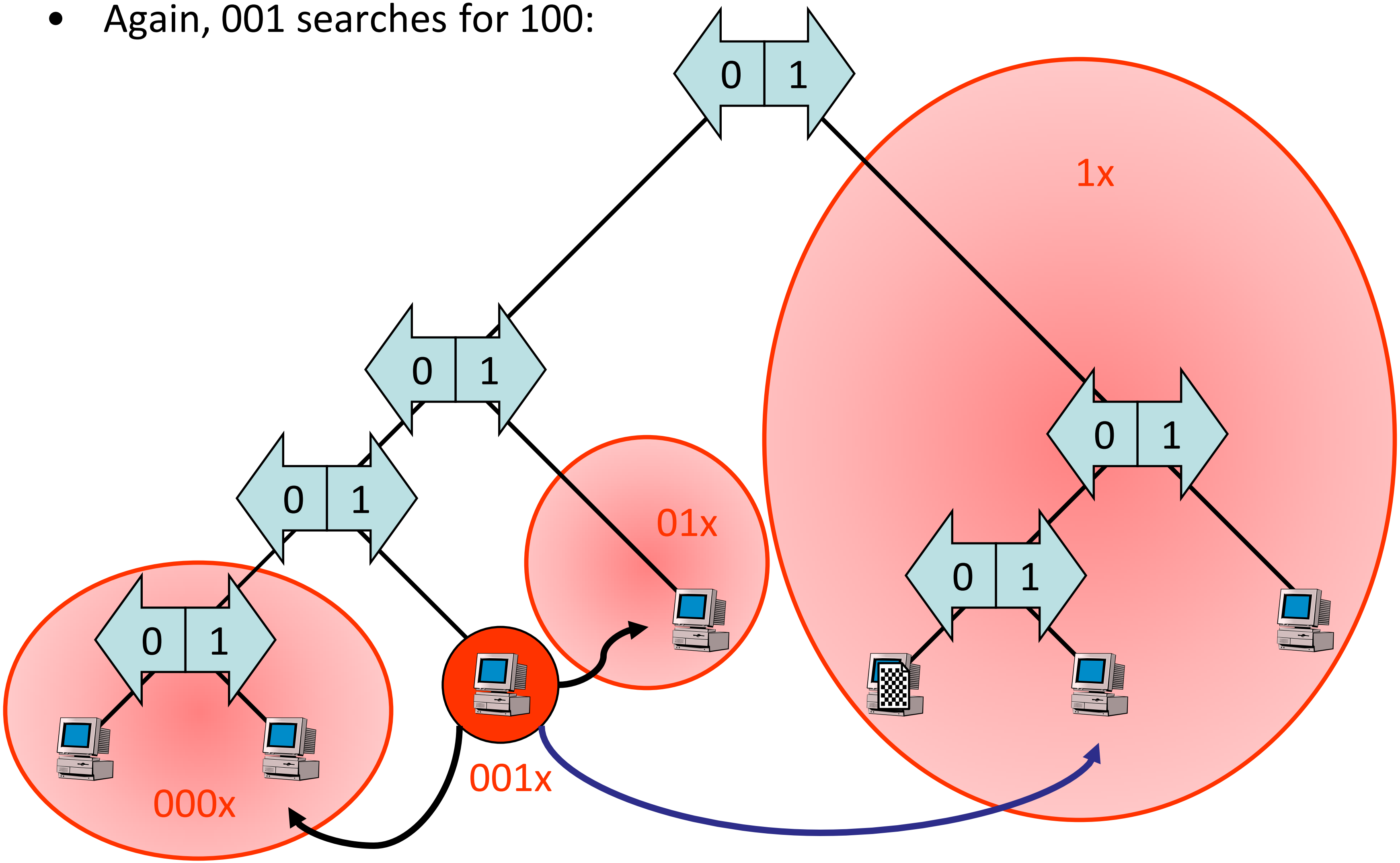


P2P Dictionary: Search



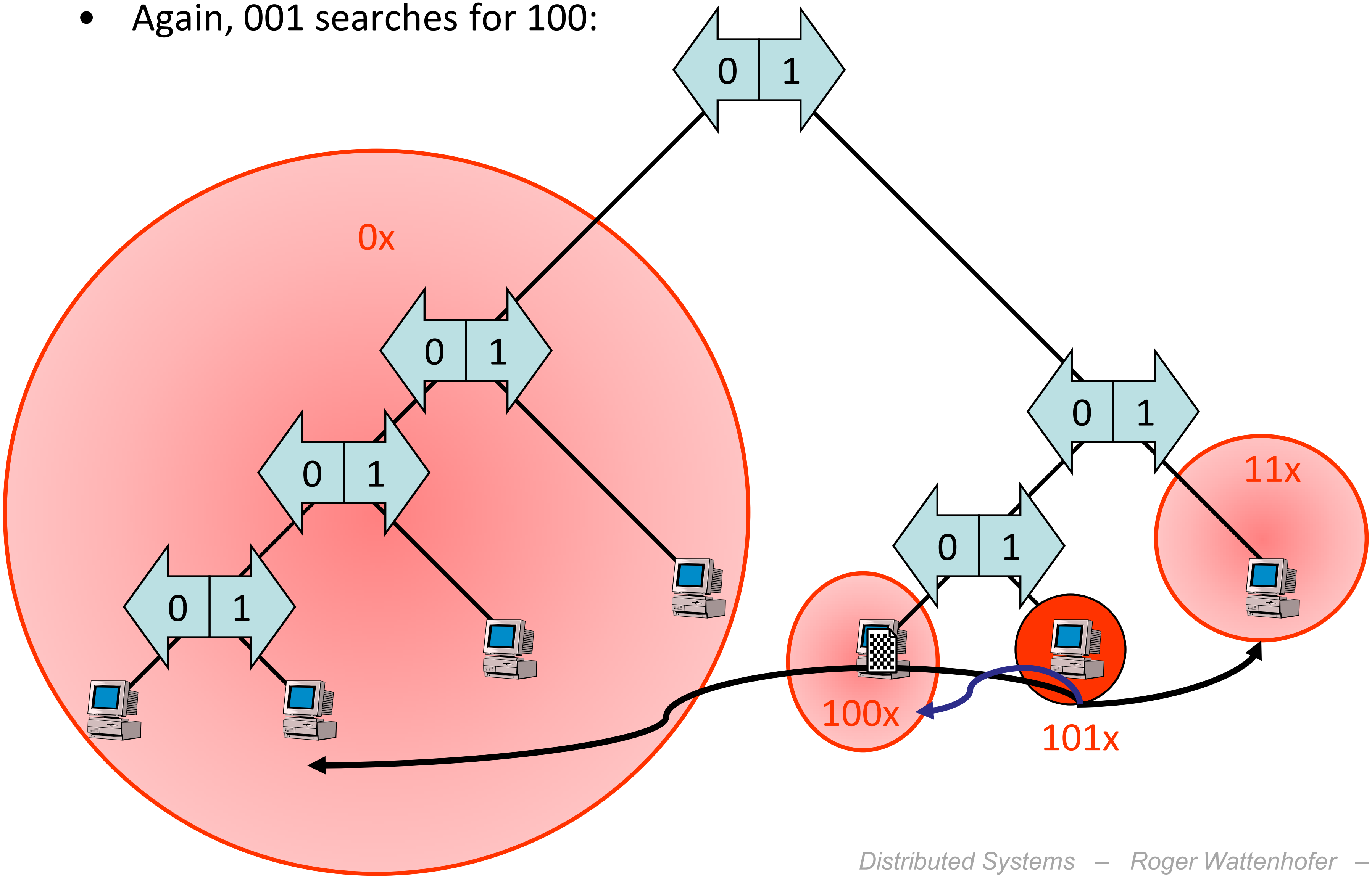
P2P Dictionary: Search

- Again, 001 searches for 100:



P2P Dictionary: Search

- Again, 001 searches for 100:



Search Analysis

- We have *n* peers in the system
- Assume that the “tree” is roughly balanced
  - Leaves (peers) on level $\log_2 n$ constant

- Search requires O(log n) steps
  - After k^{th} step, the search is in a subtree on level k
  - A “step” is a UDP (or TCP) message
  - The latency depends on P2P size (world!)

- How many peers does each peer have to know?
  - Each peer only needs to store the address of $\log_2 n$ constant peers
  - Since each peer only has to know a few peers, even if *n* is large, the system scales well!

Peer Join

- How are new peers inserted into the system?

- Step 1: Bootstrapping

- In order to join a P2P system, a joiner must already know a peer already in the system
- Typical solutions:
  - Ask a central authority for a list of IP addresses that have been in the P2P regularly; look up a listing on a web site
  - Try some of those you met last time
  - Just ping randomly (in the LAN)

Peer Join

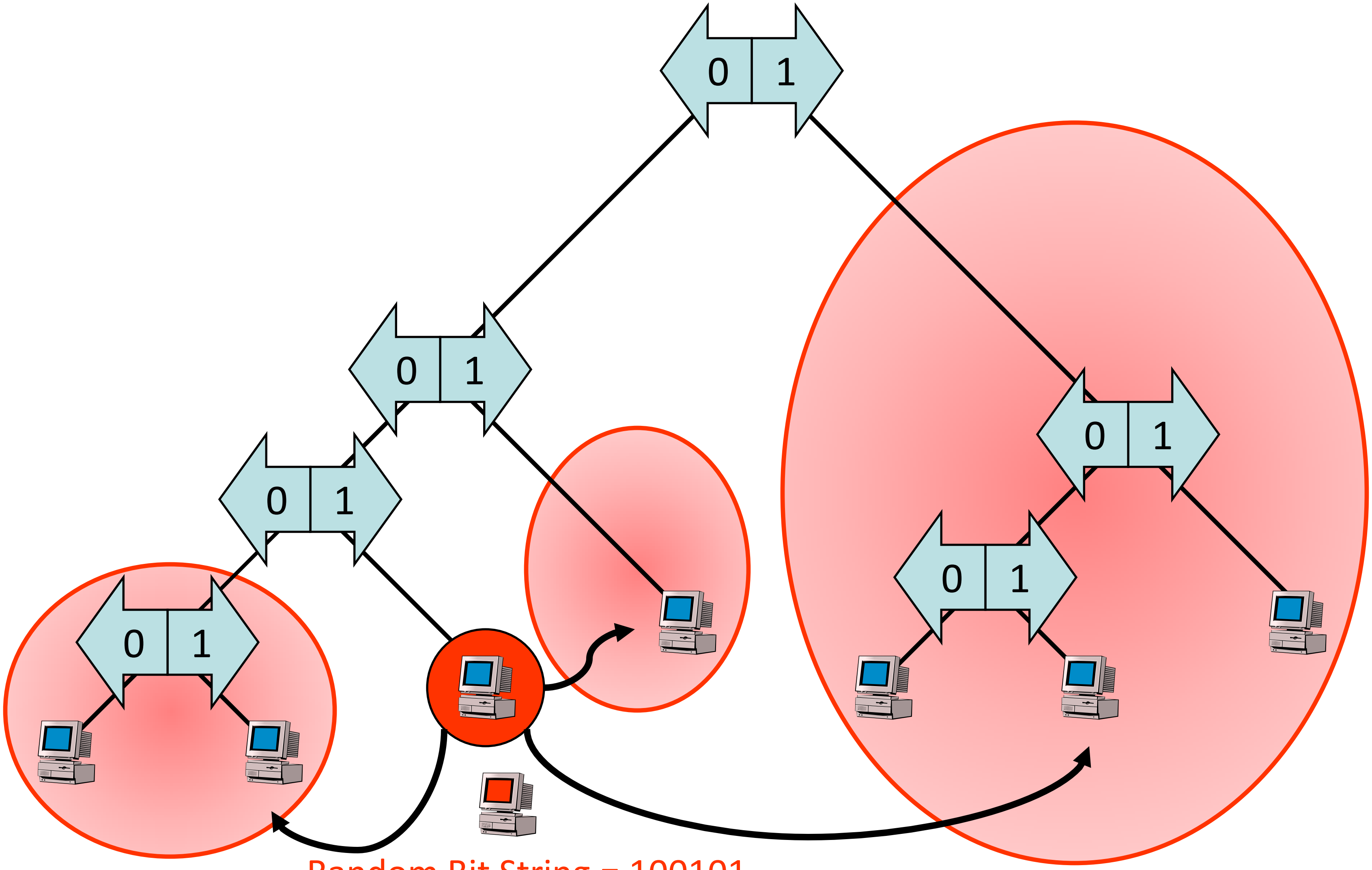
- Step 2: Find your place in the P2P system

Peer ID!

- Typical solution:
  - Choose a random bit string (which determines the place in the system)
  - Search* for the bit string
  - Split with the current leave responsible for the bit string
  - Search* for your neighbors

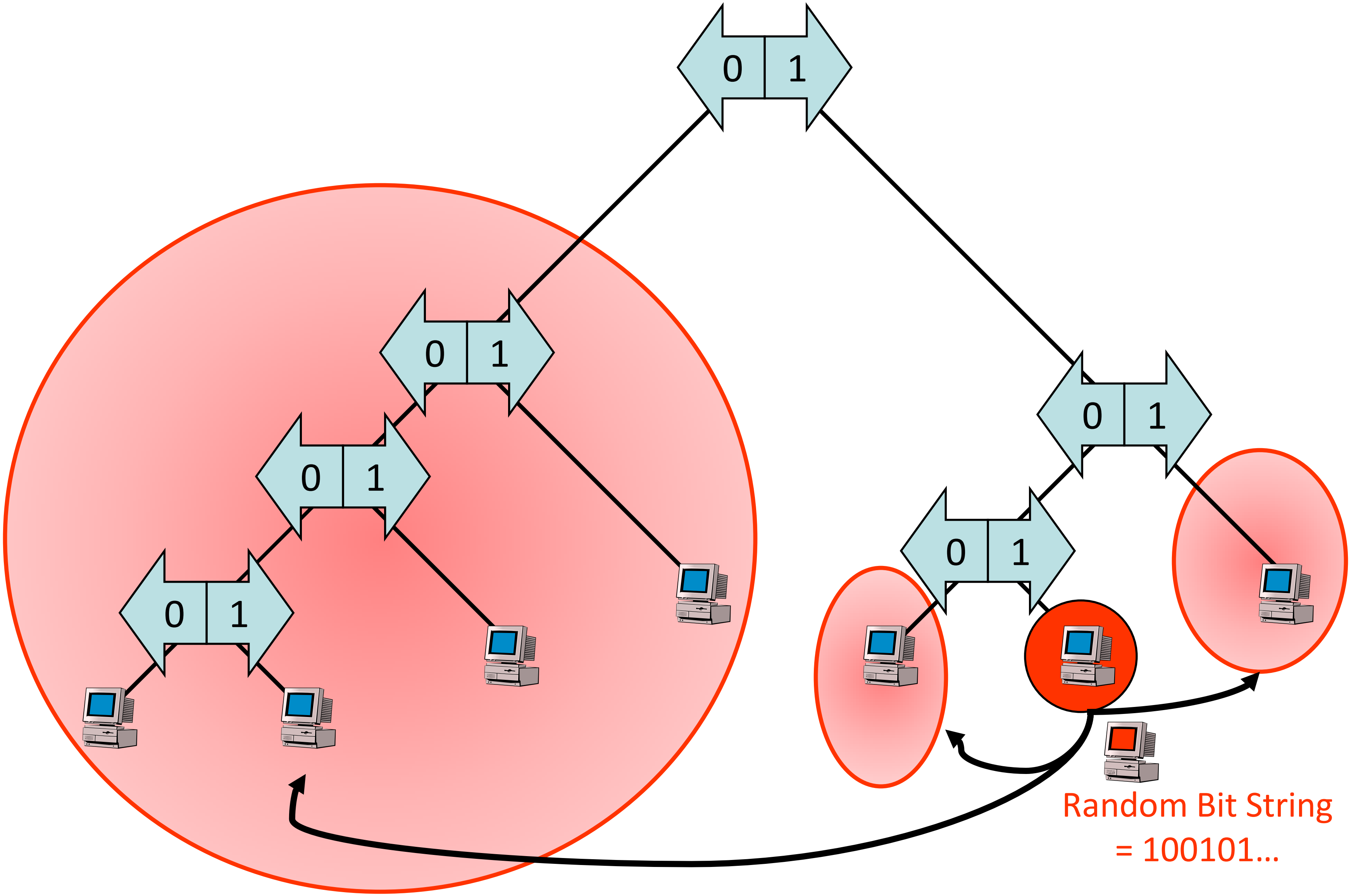
* These are standard searches

Example: Bootstrap Peer with 001



Random Bit String = 100101…

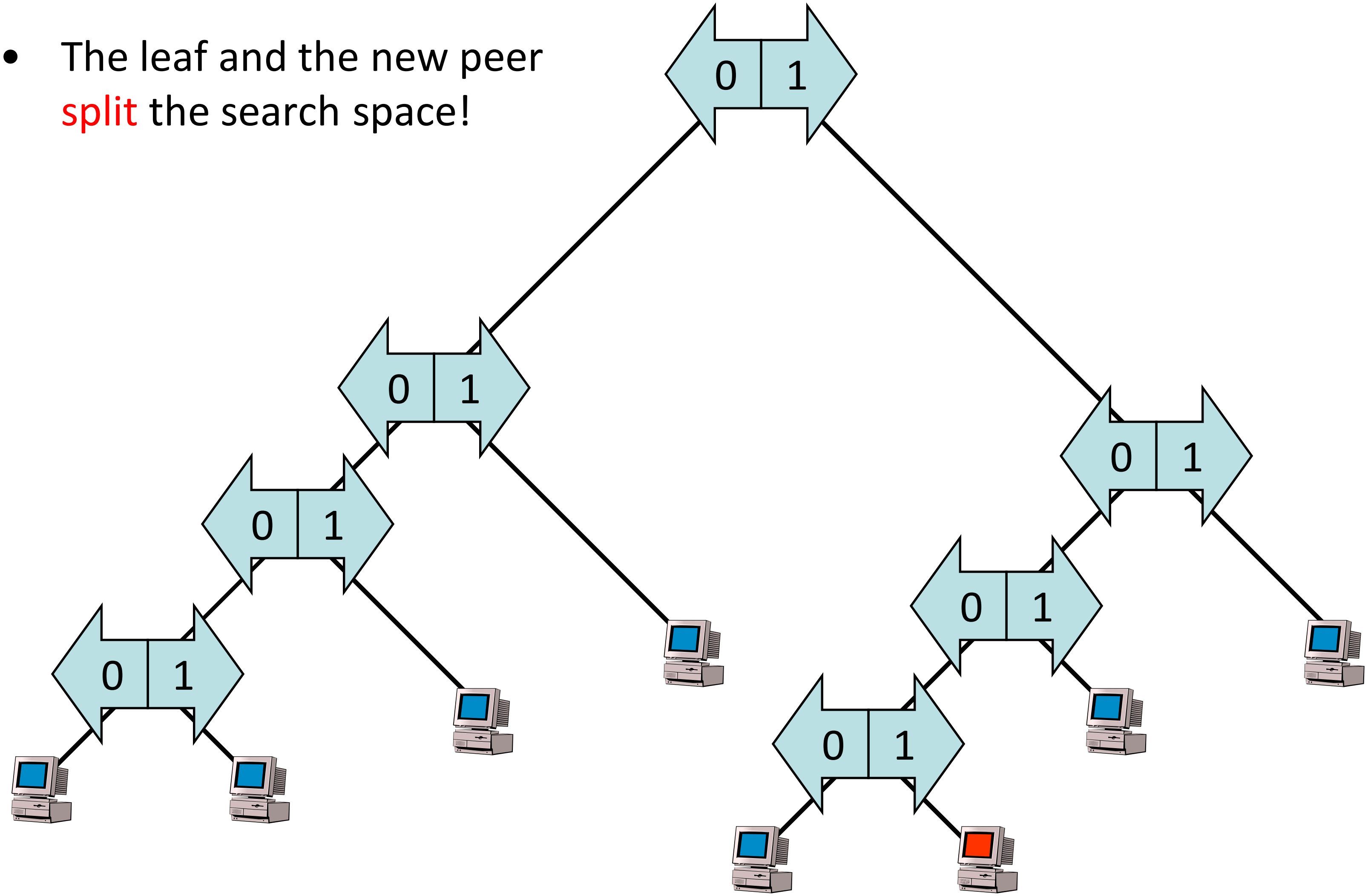
New Peer Searches 100101...



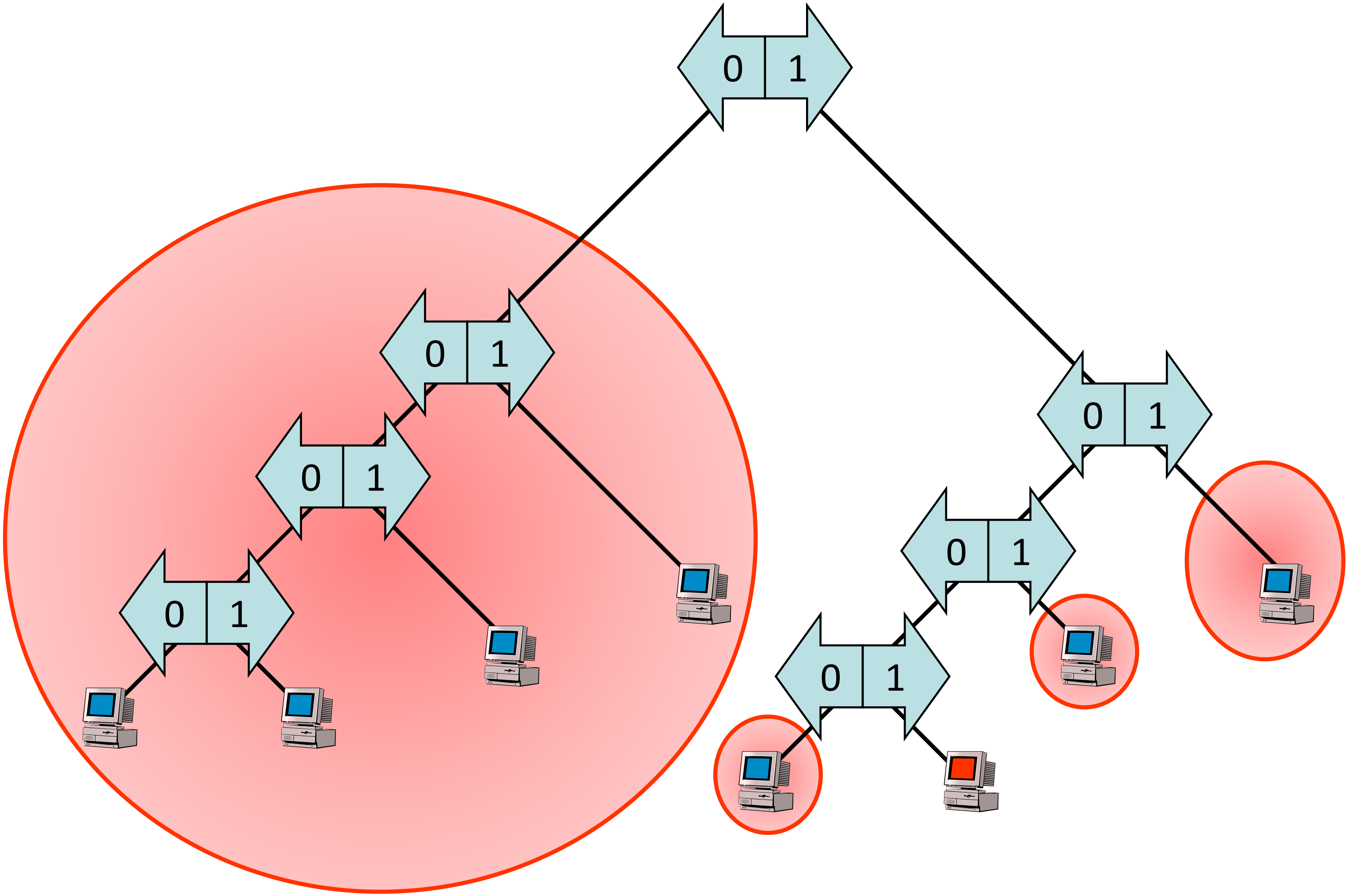
Random Bit String
= 100101...

New Peer found leaf with ID 100...

- The leaf and the new peer split the search space!



Find Neighbors



Peer Join: Discussion

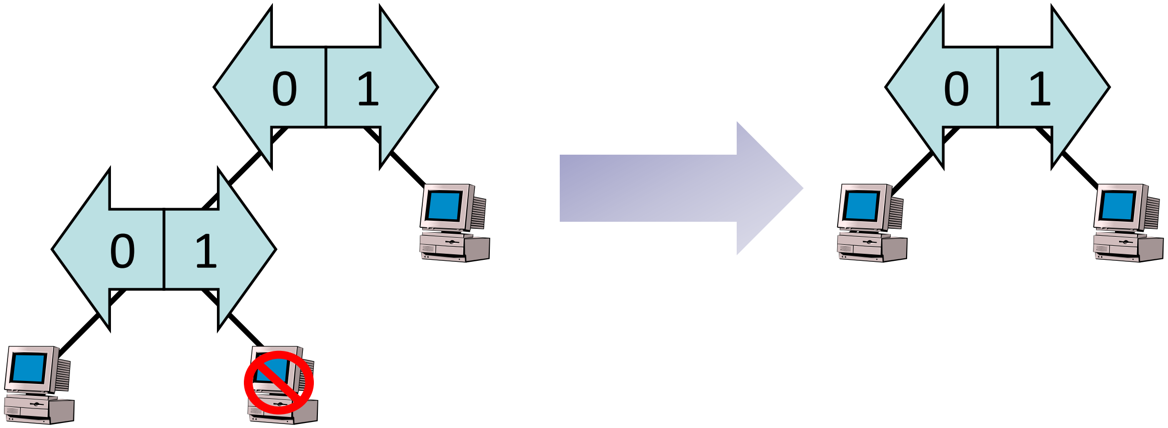
- If tree is balanced, the time to join is
  - O(log n) to find the right place
  - O(log n)·O(log n) = $O(\log^2 n)$ to find all neighbors

A regular search…

- It is be widely believed that since all the peers choose their position randomly, the tree will remain more or less balanced
  - However, theory and simulations show that this is not really true!

Peer Leave

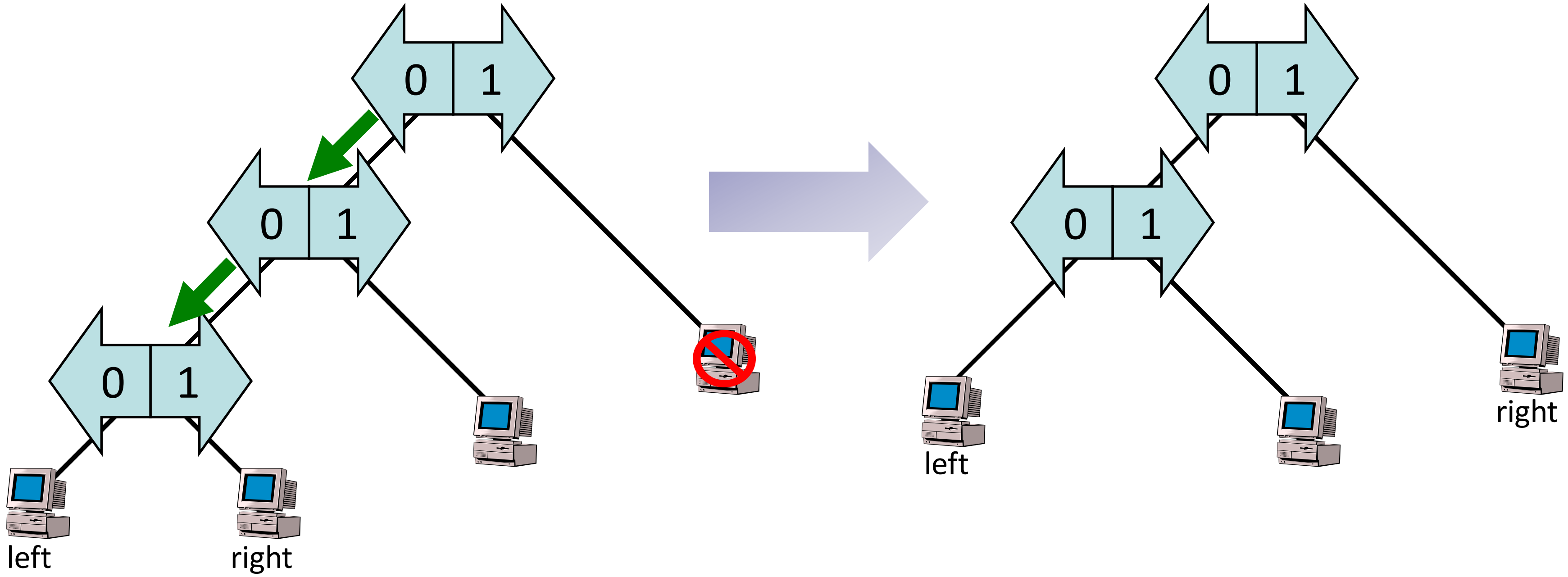
- Since a peer might leave spontaneously (there is no leave message), the leave must be detected first
- Naturally, this is done by the neighbors in the P2P system (all peers periodically ping neighbors)
- If a peer leave is detected, the peer must be replaced. If peer had a sibling leaf, the sibling might just do a "reverse split":



- If a peer does not have a sibling, search recursively!

Peer Leave: Recursive Search

- Find a replacement:
  1. Go down the sibling tree until you find sibling leaves
  2. Make the left sibling the new common node
  3. Move the free right sibling to the empty spot



Fault-Tolerance?

- Typically, only pointers to the data is stored
  - If the data holder itself crashes, the data item is not available anymore
- What if the data holder is still in the system, but the peer that stores the pointer to the data holder crashes?
  - The data holder could advertise its data items periodically
  - If it cannot reach a certain peer anymore, it must search for the peer that is now responsible for the data item, i.e., the peer's ID is closest to the data item's key
- Alternative approach: Instead of letting the data holders take care of the availability of their data, let the system ensure that there is always a pointer to the data holder!
  - Replicate the information at several peers
  - Different hashes could be used for this purpose

Questions of Experts…

- Question: I know so many other structured peer-to-peer systems (Chord, Pastry, Tapestry, CAN…); they are completely different from the one you just showed us!

- Answer: They *look* different, but in fact the difference comes mostly from the way they are presented (I give a few examples on the next slides)

The Four P2P Evangelists

- If you read your average P2P paper, there are (almost) always four papers cited which “invented” efficient P2P in 2001:

Chord CAN Pastry Tapestry

- These papers are somewhat similar, with the exception of CAN (which is not really efficient)

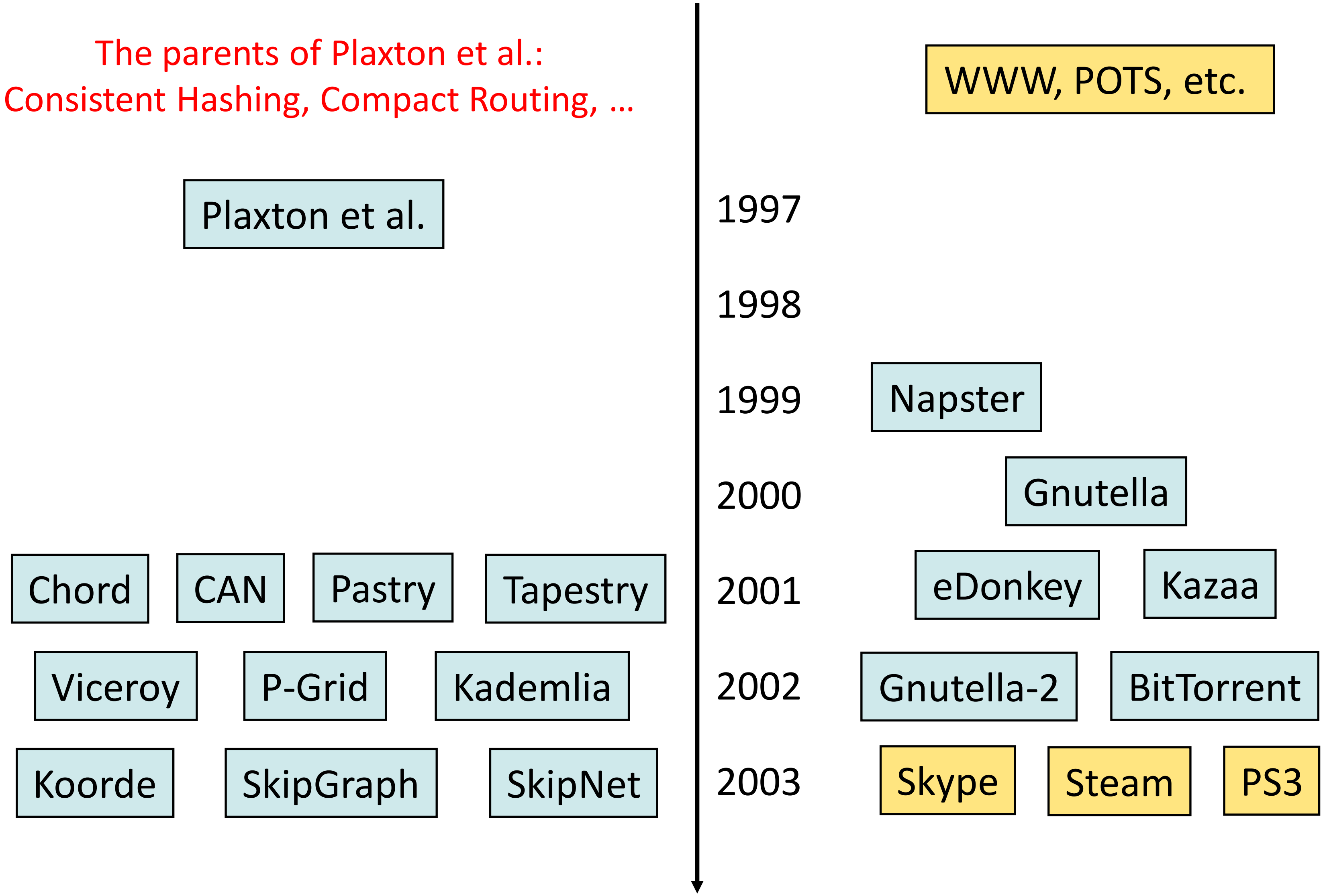
- So what are the „Dead Sea scrolls of P2P”?

Intermezzo: “Dead Sea Scrolls of P2P”

„Accessing Nearby Copies of Replicated Objects in a Distributed Environment“ [Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa, SPAA 1997]

- Basically, the paper proposes an efficient search routine (similar to the four famous P2P papers)
  - In particular search, insert, delete, storage costs are all logarithmic, the base of the logarithm is a parameter
- The paper takes latency into account
  - In particular it is assumed that nodes are in a metric, and that the graph is of „bounded growth“ (meaning that node densities do not change abruptly)

Intermezzo: Genealogy of P2P



The parents of Plaxton et al.:
Consistent Hashing, Compact Routing, …

WWW, POTS, etc.

Plaxton et al.

1997

1998

1999 Napster

2000 Gnutella

Chord CAN Pastry Tapestry 2001 eDonkey Kazaa

Viceroy P-Grid Kademlia 2002 Gnutella-2 BitTorrent

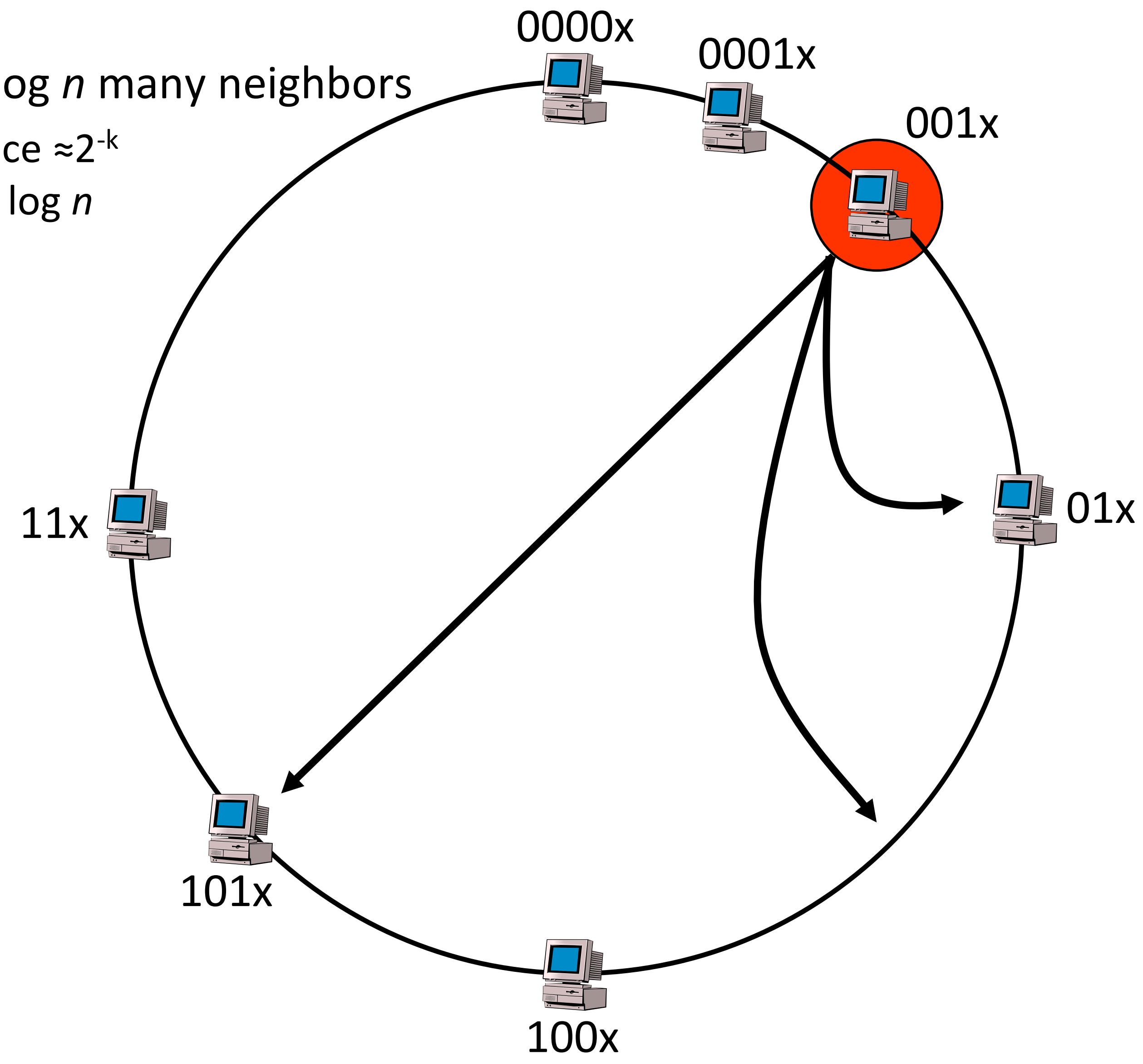
Koorde SkipGraph SkipNet 2003 Skype Steam PS3

Chord

- Chord is the most cited P2P system [Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, SIGCOMM 2001]
- Most discussed system in distributed systems and networking books, for example in Edition 4 of Tanenbaum’s Computer Networks
- There are extensions on top of it, such as CFS, Ivy…

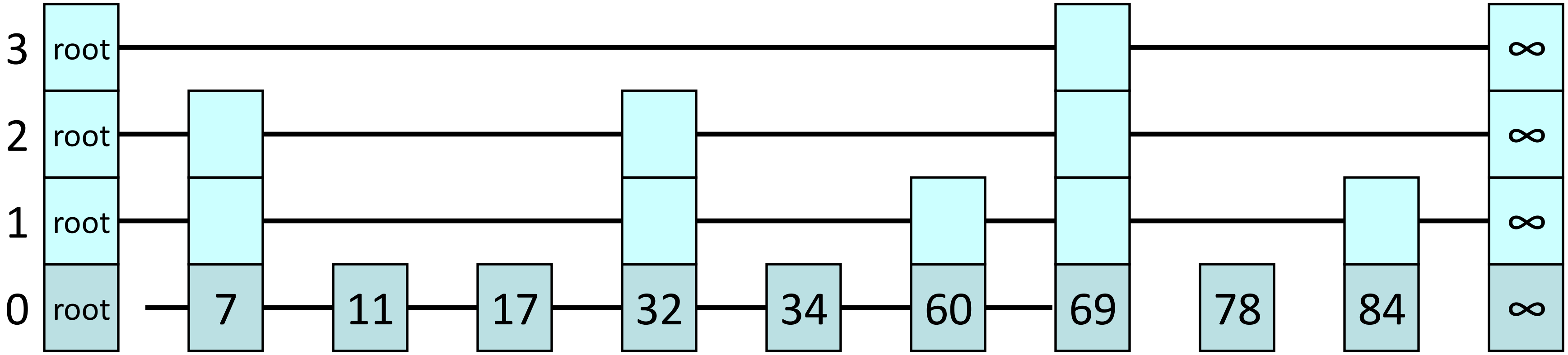
Chord

- Every peer has log *n* many neighbors
  - One in distance ≈2^{-k} for k=1, 2, …, log *n*



Skip List

- How can we ensure that the search tree is balanced?
  - We don’t want to implement distributed AVL or red-black trees…
- Skip List:
  - (Doubly) linked list with sorted items
  - An item adds additional pointers on level 1 with probability ½. The items with additional pointers further add pointers on level 2 with prob. ½ etc.
  - There are $\log_2 n$ levels in expectation
- Search, insert, delete: Start with root, search for the right interval on highest level, then continue with lower levels



Skip List

- It can easily be shown that search, insert, and delete terminate in O(log n) expected time, if there are n items in the skip list
- The expected number of pointers is only twice as many as with a regular linked list, thus the memory overhead is small
- As a plus, the items are always ordered…

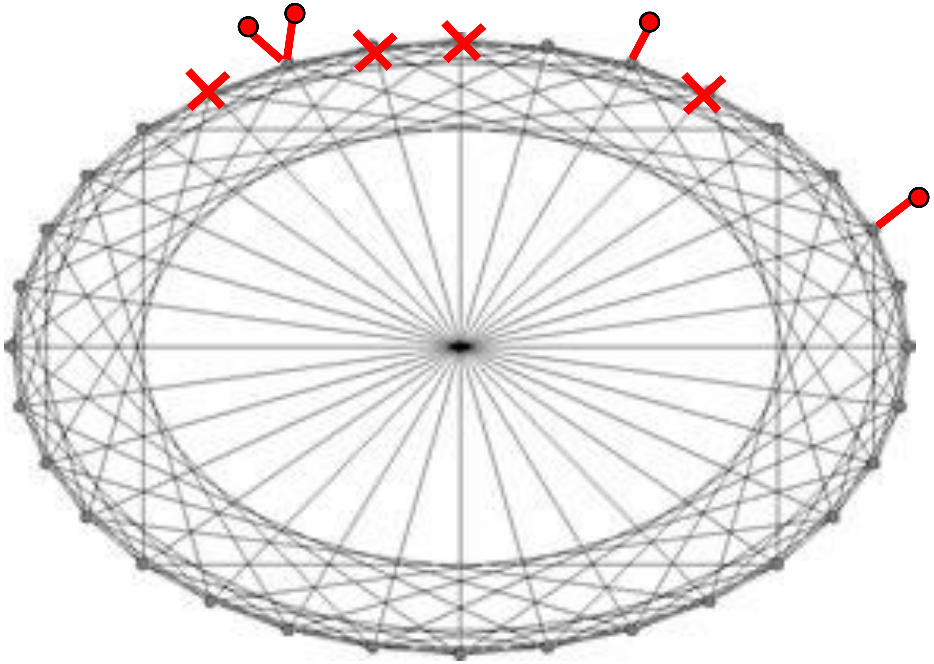
P2P Architectures

- Use the skip list as a P2P architecture
  - Again each peer gets a random value between 0 and 1 and is responsible for storing that interval
  - Instead of a root and a sentinel node (“∞”), the list is short-wired as a ring
- Use the Butterfly or DeBruijn graph as a P2P architecture
  - Advantage: The node degree of these graphs is constant → Only a constant number of neighbors per peer
  - A search still only takes $O(\log n)$ hops

Dynamics Reloaded

- Churn: Permanent joins and leaves
  - Why permanent?
  - Saroiu et al.: „A Measurement Study of P2P File Sharing Systems“: Peers join system for one hour on average
  - Hundreds of changes per second with millions of peers in the system!

- How can we maintain desirable properties such as
  - connectivity
  - small network diameter
  - low peer degree?

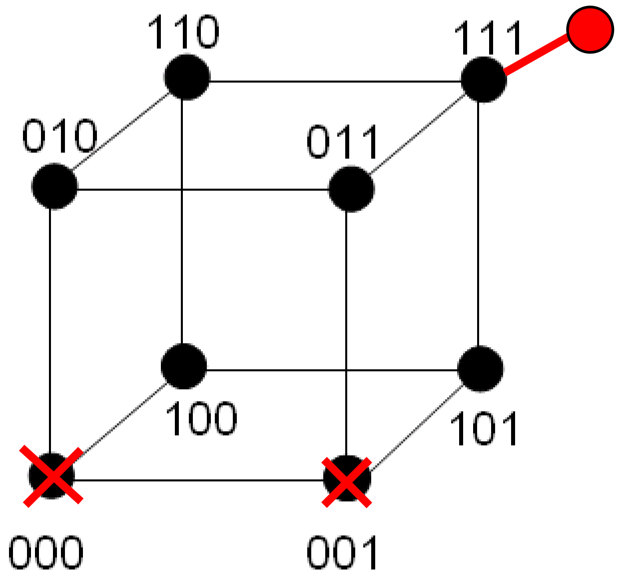


A First Approach

- A fault-tolerant hypercube?

- What if the number of peers is not 2^i?
- How can we prevent degeneration?
- Where is the data stored?

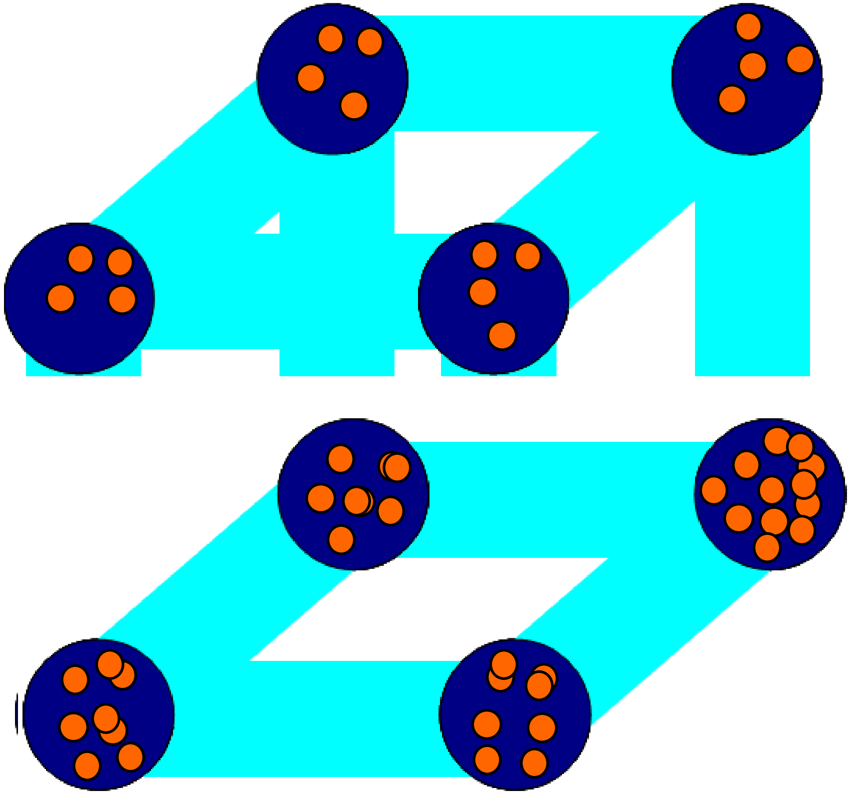
- Idea: Simulate the hypercube!



Simulated Hypercube

- Simulation: Each node consists of several peers

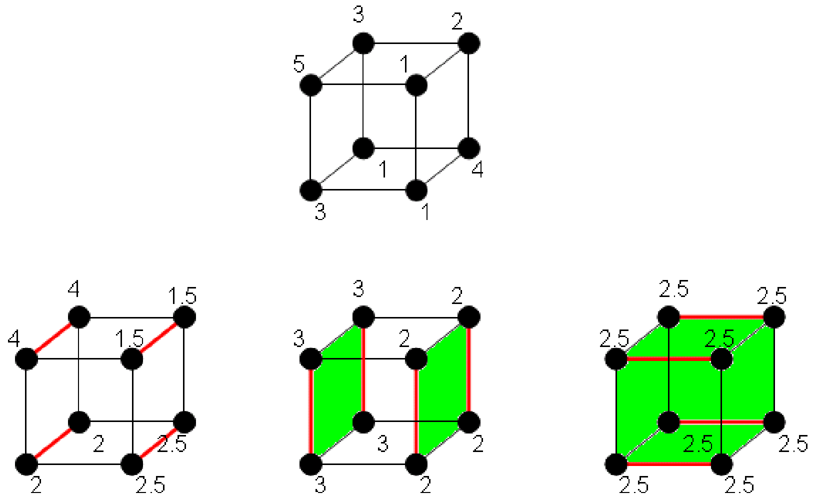
- Basic components:
- Peer distribution
  - Distribute peers evenly among all hypercube nodes
  - A token distribution problem
- Information aggregation
  - Estimate the total number of peers
  - Adapt the dimension of the simulated hypercube



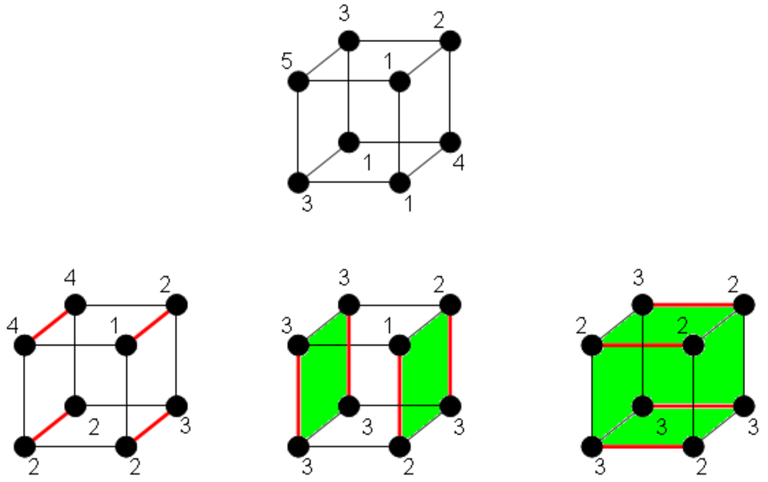
Peer Distribution

- Algorithm: Cycle over dimensions and balance!
- Perfectly balanced after d rounds

Dimension of hypercube

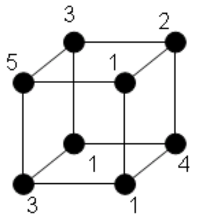


- Problem 1: Peers are not fractional!
- Problem 2: Peers may join/leave during those d rounds!
- "Solution": Round numbers and ignore changes during the d rounds

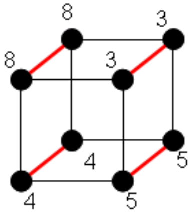


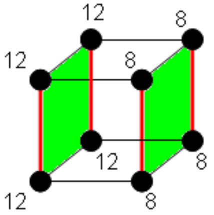
Information Aggregation

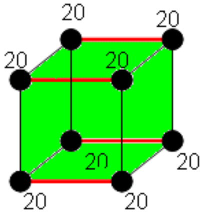
- Goal: Provide the same (good!) estimation of the total number of peers presently in the system to all nodes
- Algorithm: Count peers in every sub-cube by exchanging messages wih the corresponding neighbor!
- Correct number after d rounds



- Problem: Peers may join/leave during those d rounds!
- Solution: Pipe-lined execution







- It can be shown that all nodes get the same estimate
- Moreover, this number represents the correct state d rounds ago!

Composing the Components

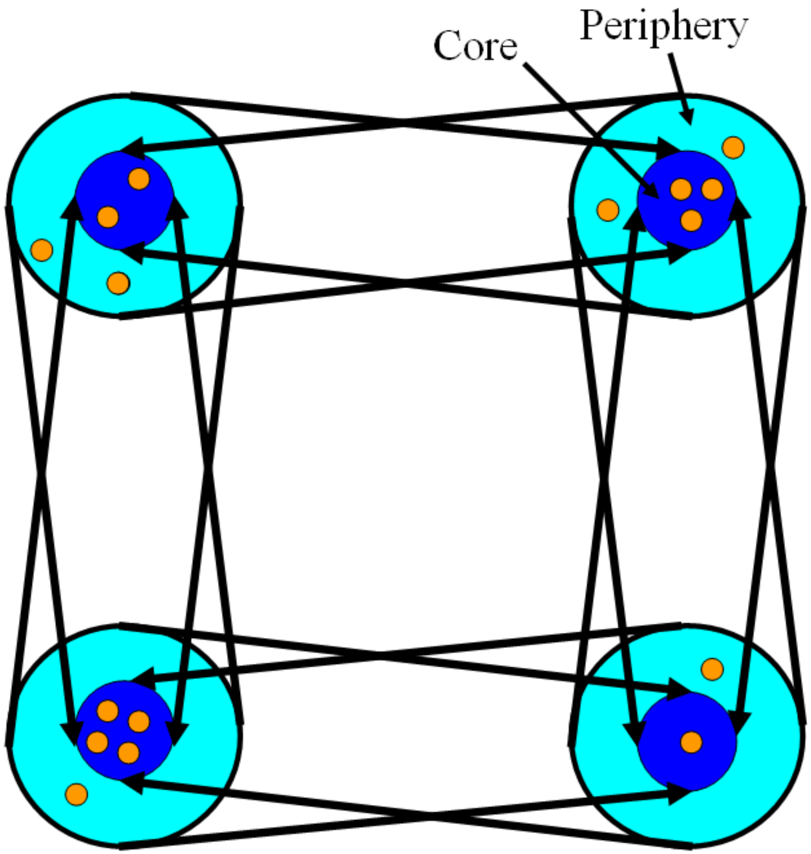
- The system permanently runs
  - the peer distribution algorithm to balance the nodes
  - the information aggregation algorithm to estimate the total number of peers and change the dimension accordingly

- How are the peers connected inside a simulated node, and how are the edges of the hypercube represented?

- Where is the data of the DHT stored?

Distributed Hash Table

- Hash function determines node where data is replicated
- Problem: A peer that has to move to another node must replace store different data items
- Idea: Divide peers of a node into core and periphery
  - Core peers store data
  - Peripheral peers are used for peer distribution
- Peers inside a node are completely connected
- Peers are connected to all core peers of all neighboring nodes



Evaluation

- The system can tolerate O(log *n*) joins and leaves each round

- The system is never fully repaired, but always fully funtional!

- In particular, even if there are O(log *n*) joins/leaves per round we always have
  - at least one peer per node
  - at most O(log *n*) peers per node
  - a network diameter of O(log *n*)
  - a peer degree of O(log *n*)

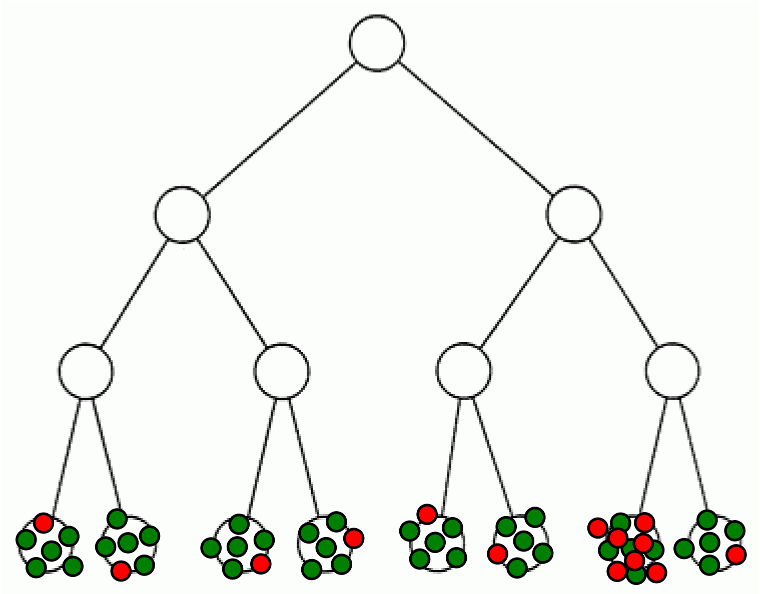
Number of neighbors/connections

Byzantine Failures

- If Byzantine nodes control more and more corrupted nodes and then crash all of them at the same time ("sleepers"), we stand no chance.

- "Robust Distributed Name Service" [Baruch Awerbuch and Christian Scheideler, IPTPS 2004]

- Idea: Assume that the Byzantine peers are the minority. If the corrupted nodes are the majority in a specific part of the system, they can be detected (because of their unusual high density).



Selfish Peers

- Peers may not try to destroy the system, instead they may try to benefit from the system without contributing anything
- Such selfish behavior is called free riding or freeloading

- Free riding is a common problem in file sharing applications:
- Studies show that most users in the Gnutella network do not provide anything
  - Gnutella is accessed through clients such as BearShare, iMesh…
- Protocols that are supposed to be “incentive-compatible”, such as BitTorrent, can also be exploited
  - The BitThief client downloads without uploading!

- Many techniques have been proposed to limit free riding behavior
  - Source coding, shared history, virtual currency…
  - These techniques are not covered in this lecture!

A Large-Scale System in a Nutshell: Dynamo

- Dynamo is a key-value storage system by Amazon
- Goal: Provide an “always-on” experience
  - Availability is more important than consistency
- The system is (nothing but) a DHT
- Trusted environment (no Byzantine processes)
- Ring of nodes
  - Node n_i is responsible for keys between n_{i-1} and n_i
  - Nodes join and leave dynamically
- Each entry replicated across N nodes
- Recovery from error:
  - When? On read
  - How? Depends on application, e.g. “last write wins” or “merge”
  - One vector clock per entry to manage different versions of data

Basically what we talked about

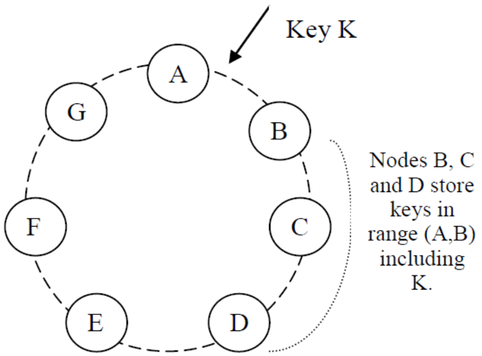


Figure 2: Partitioning and replication of keys in Dynamo ring.

Summary

- We have studied a few practical consensus algorithms
- In particular we have seen
  - 2PC
  - 3PC
  - Paxos
  - Chubby
  - PBFT
  - Zyzzyva, PeerReview, FARSITE
- We also talked about techniques to handle large-scale networks
  - Consistent hashing
  - Skip lists
  - Coping with dynamics
  - Dynamo

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.
- PeerReview is from Haeberlen, Kouznetsov, and Druschel, 2007.
- FARSITE is from Adya et al., 2002.
- Concurrent hashing and random trees have been proposed by Karger, Lehman, Leighton, Levine, Lewin, and Panigrahy, 1997.
- The churn-resistent P2P System is due to Kuhn et al., 2005.
- Dynamo is from DeCandia et al., 2007.