Practice: Large Systems

Part 2, Chapter 2

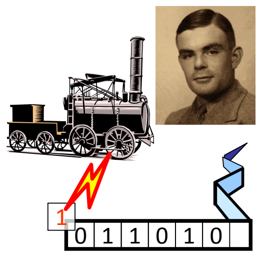
Thomas Locher
Roger Wattenhofer

Overview

- Introduction
- Strong Consistency
  - Crash Failures: Primary Copy, Commit Protocols
  - Crash-Recovery Failures: Paxos, Chubby
  - Byzantine Failures: PBFT, Zyzzyva
- CAP: Consistency or Availability?
- Weak Consistency
  - Consistency Models
  - Peer-to-Peer, Distributed Storage, Cloud Computing
- Computation: MapReduce

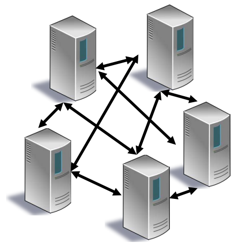
Computability vs. Efficiency

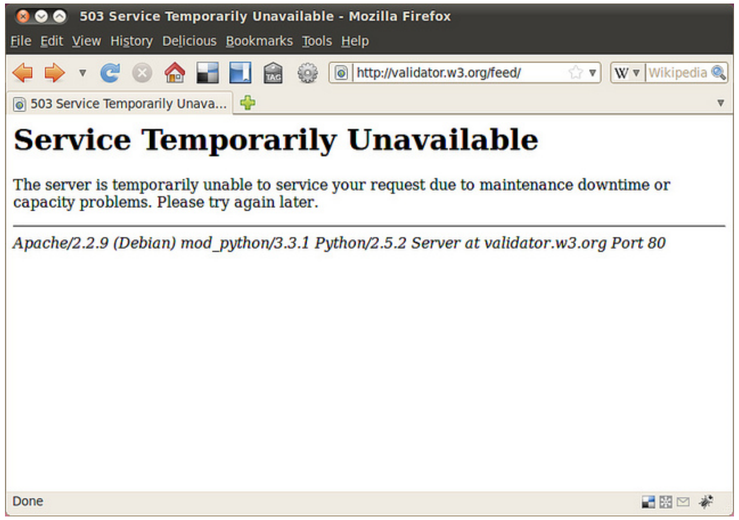
- In the last part, we studied computability
  - When is it possible to guarantee consensus?
  - What kind of failures can be tolerated?
  - How many failures can be tolerated?



Worst-case scenarios!

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



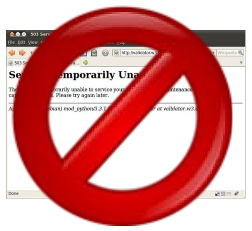

503 Service Temporarily Unavailable - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://validator.w3.org/feed/
Wikipedia
503 Service Temporarily Unava...
Done

Service Temporarily Unavailable

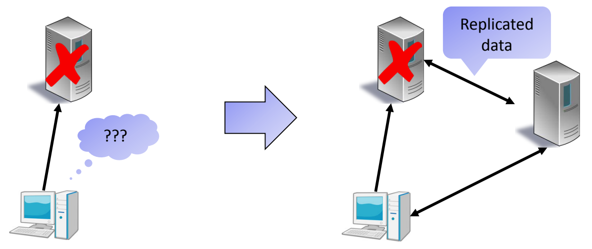
The server is temporarily unable to service your request due to maintenance downtime or capacity problems. Please try again later.

Apache/2.2.9 (Debian) mod_python/3.3.1 Python/2.5.2 Server at validator.w3.org Port 80

Fault-Tolerance in Practice



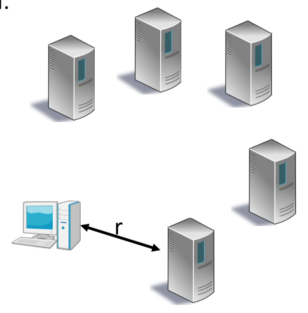
- Fault-Tolerance is achieved through *replication*



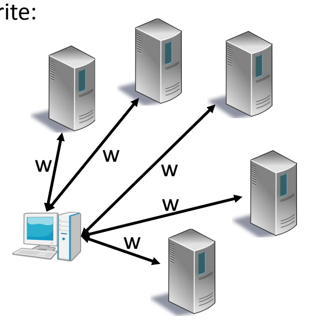
Replication is Expensive

- 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:



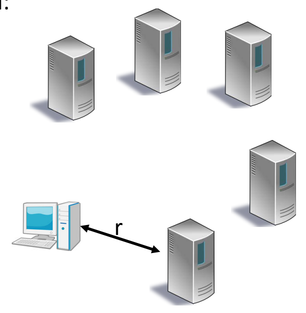
Write:



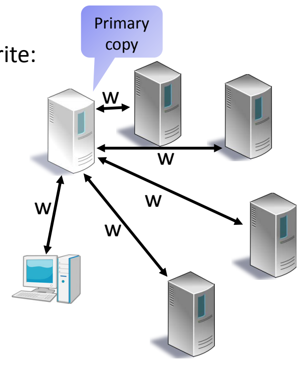
Primary Copy

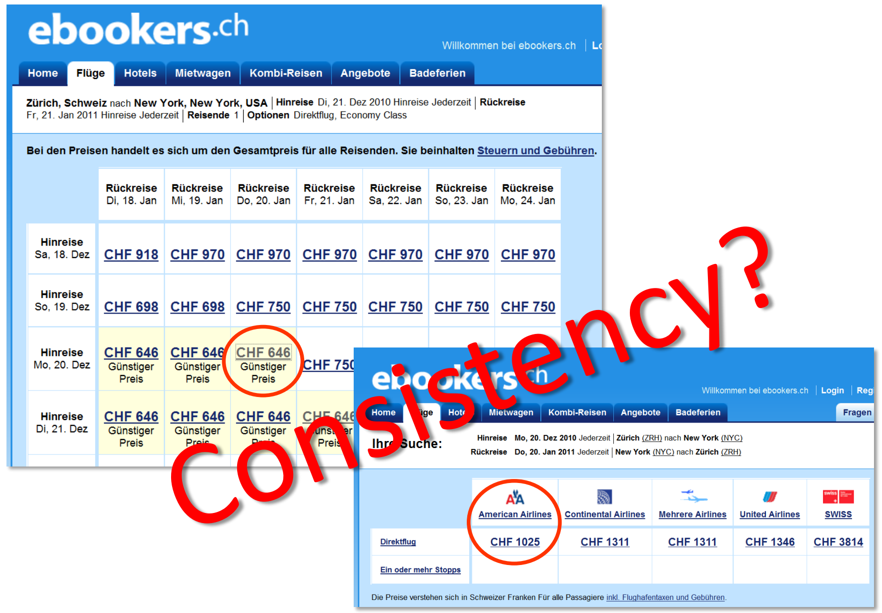
- Can we reduce the load on the clients?
- Yes! Write only to one server (the primary copy), and let primary copy distribute the update
  - This way, the client only sends one message in order to read and write

Read:



Write:





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	CHF 750	[illegible]	[illegible]	[illegible]
Hinreise Di, 21. Dez	CHF 646 Günstiger Preis	CHF 646 Günstiger Preis	CHF 646 Günstiger Preis	CHF 646 Günstiger Preis	[illegible]	[illegible]	[illegible]

Willkommen bei ebookers.ch | Login | Reg

Home | Flüge | Hotels | Mietwagen | Kombi-Reisen | Angebote | Badeferien | Fragen

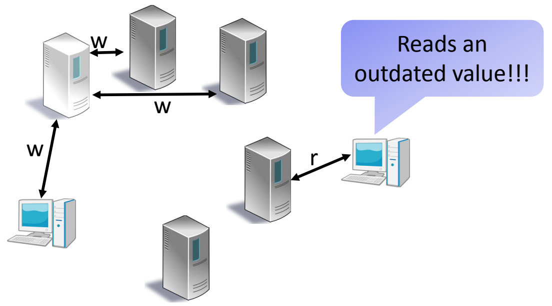
Ihre Suche:
Hinreise **Mo, 20. Dez 2010** Jederzeit | **Zürich** (ZRH) nach **New York** (NYC)
Rückreise **Do, 20. Jan 2011** Jederzeit | **New York** (NYC) nach **Zürich** (ZRH)

	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.

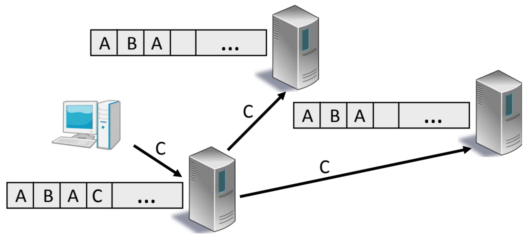
Problem with Primary Copy

- 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



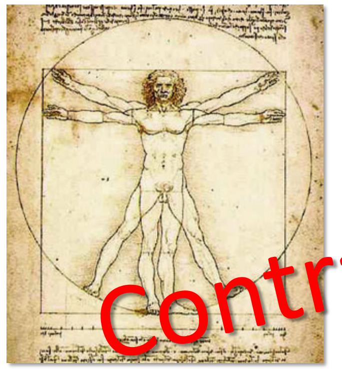
State Machine Replication?

- 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

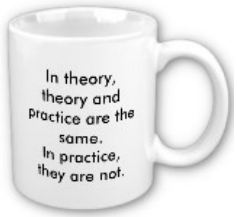


Practice

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

Consensus is required to guarantee consistency among different replicas

From Theory to Practice


In theory, theory and practice are the same. In practice, they are not.

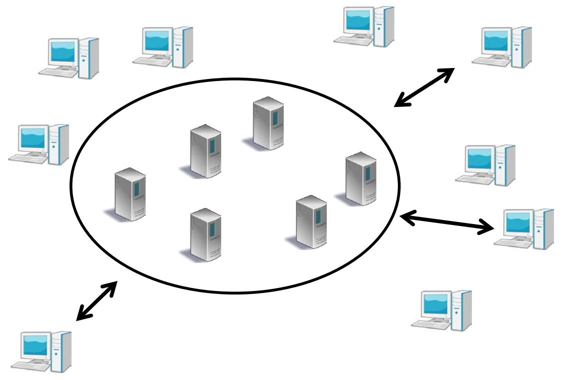
- So, how do we go from theory to practice…?
- Communication is often not synchronous, but not completely asynchronous either
  - There may be reasonable bounds on the message delays
  - Practical systems often use message passing. The machines wait for the response from another machine and abort/retry 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!

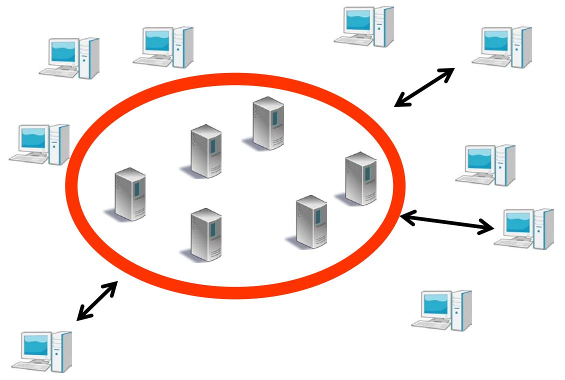
System

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



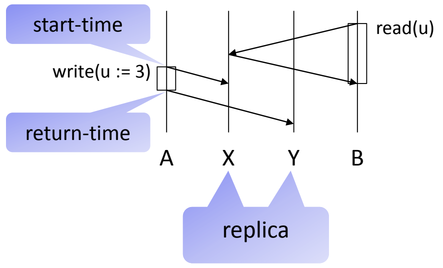
Consistency Models (Client View)

- 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.

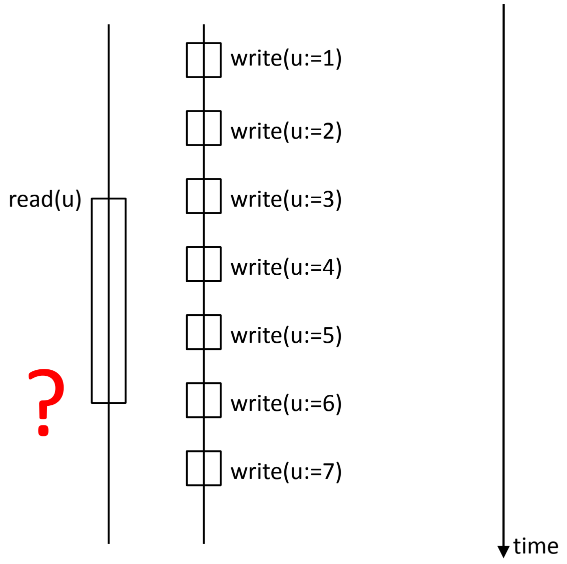


Let‘s Formalize these Ideas

- 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 $T_R - T_s$.

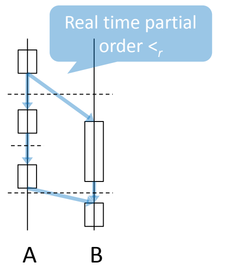


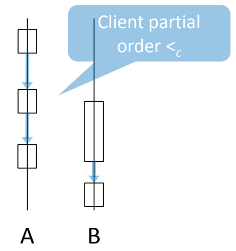
Motivation



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

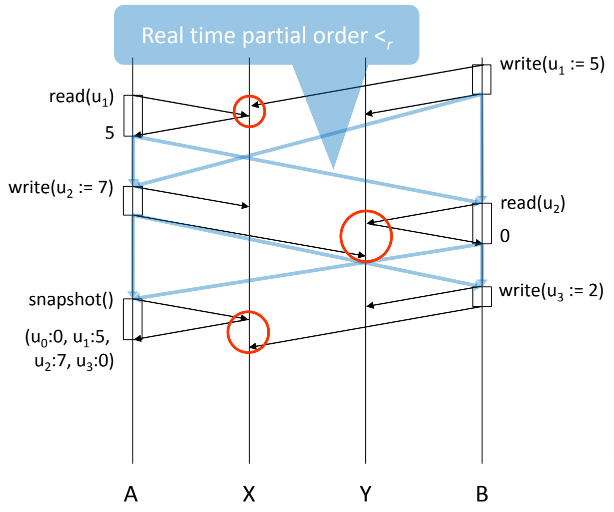
- A replicated system is called linearizable if it behaves exactly as a single-site (unreplicated) system.

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

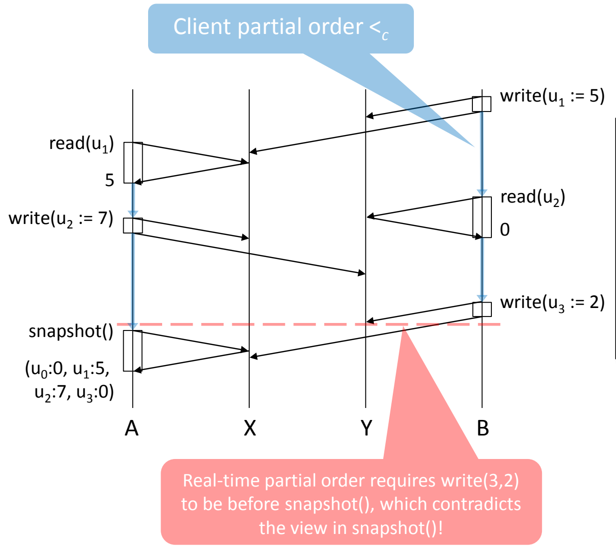
- Orders at different locations are disregarded if it cannot be determined by any observer within the system.
- I.e., a system provides sequential consistency if every node of the system sees the (write) operations on the same memory address in the same order, although the order may be different from 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

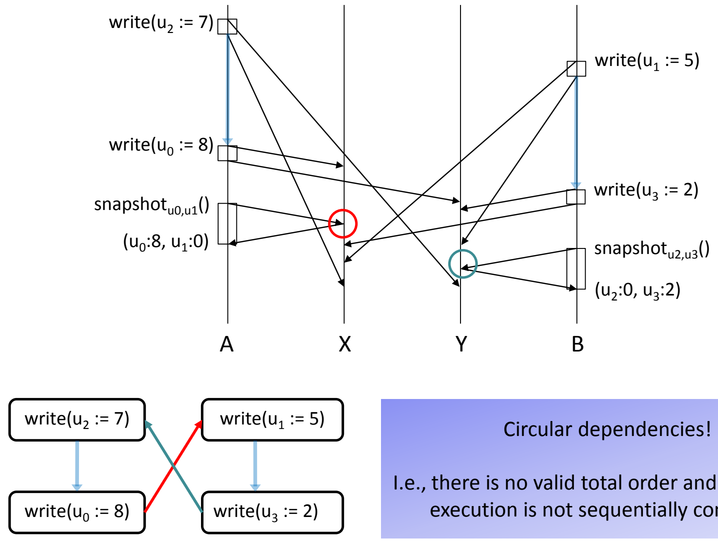


Client partial order $<_c$
write(u_1 := 5)
read(u_1)
5
read(u_2)
0
write(u_2 := 7)
write(u_3 := 2)
snapshot()
(u_0:0, u_1:5, u_2:7, u_3:0)
A
X
Y
B
Real-time partial order requires write(3,2) to be before snapshot(), which contradicts the view in snapshot()!

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)

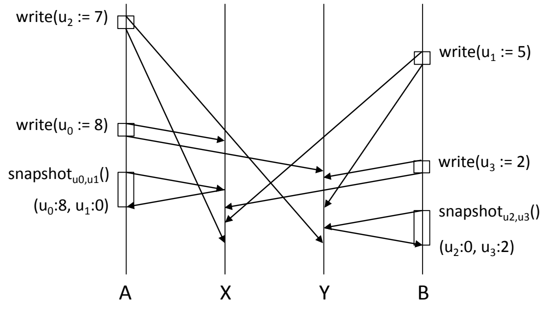
Is Every Execution Sequentially Consistent?



Circular dependencies!

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

Sequential Consistency does not Compose



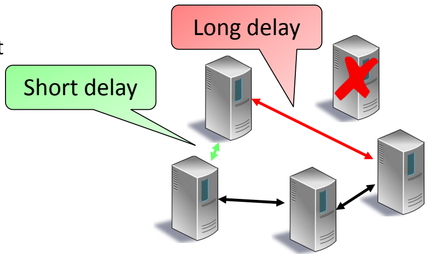
- If we only look at data items 0 and 1, operations are sequentially consistent
- If we only look at data items 2 and 3, operation are also sequentially consistent
- But, as we have seen before, the combination is not sequentially consistent

Sequential consistency does not compose!

(this is in contrast to linearizability)

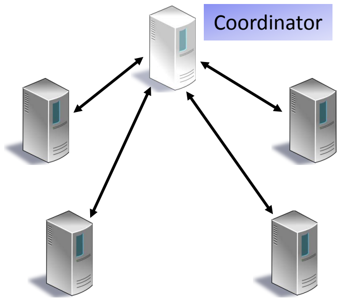
Transactions

- 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!



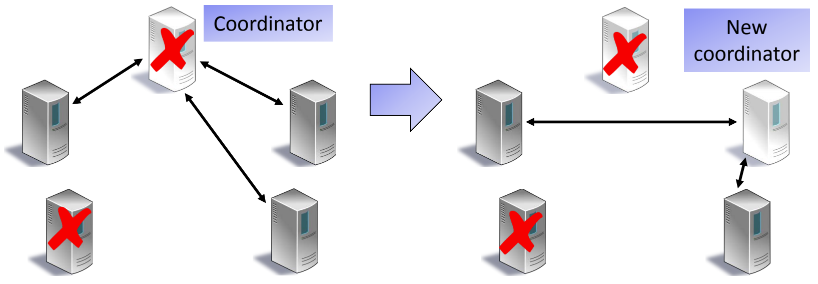
Two-Phase Commit (2PC)

- 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



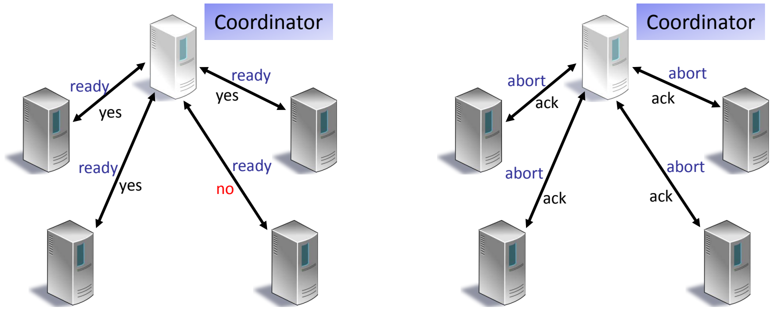
Two-Phase Commit: Failures

- 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

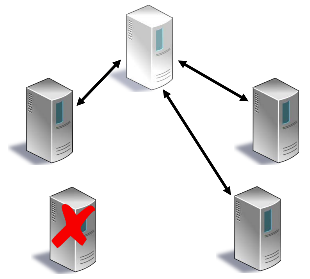
```
Phase 1:

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

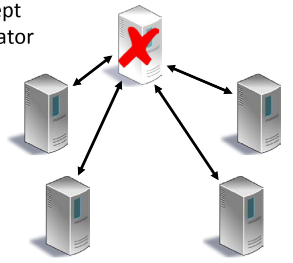


Two-Phase Commit: Analysis

- 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…

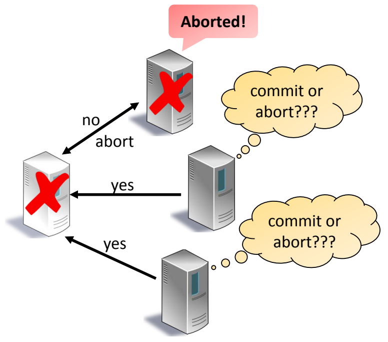
- How does the new coordinator proceed?
  - It must ask the other nodes if a node has already received a commit
  - A node that has received a commit replies yes, otherwise it sends no and promises not to accept a commit that may arrive from the old coordinator
  - If some node replied yes, the new coordinator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



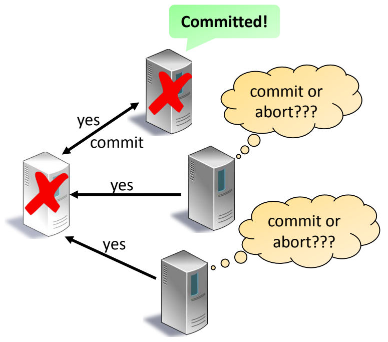
- This works if there is only one failure
- Does 2PC still work with multiple failures…?

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?



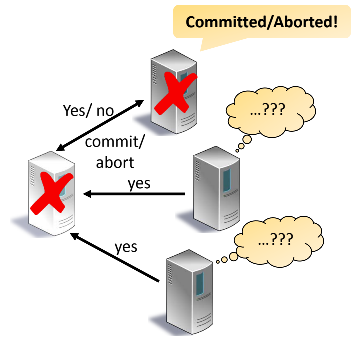
The nodes cannot commit!



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

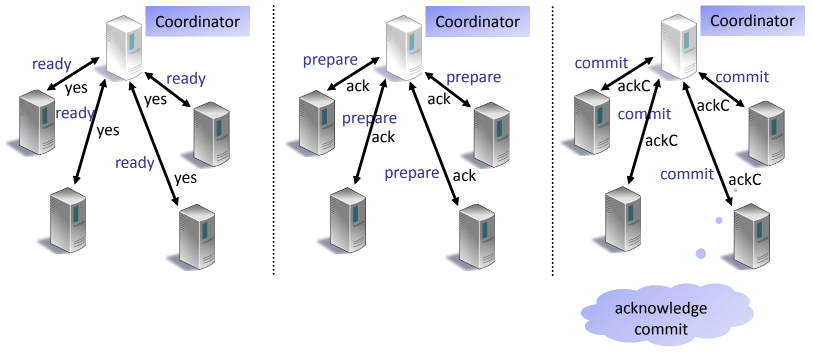
- Solution: Add another phase to the protocol!
  - The new phase precedes the commit phase
  - The goal is to inform all nodes that all are ready to commit (or not)
  - At the end of this phase, every node knows whether or not all nodes want to commit *before* any node has actually committed or aborted!

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



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!

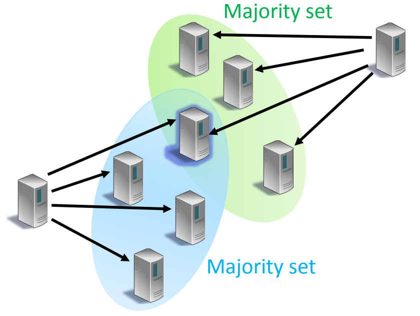
Paxos

- 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

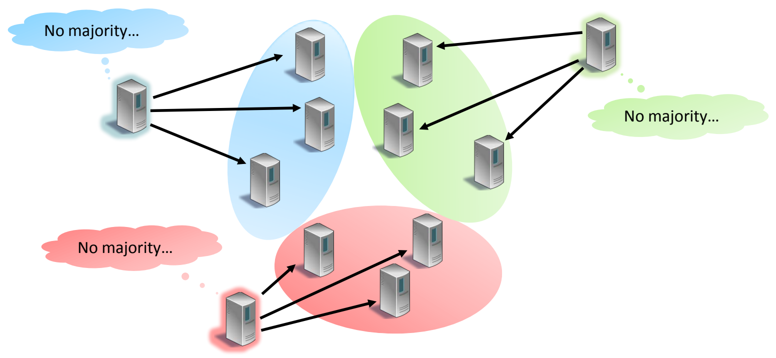
Paxos: Majority Sets

- 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

- 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!

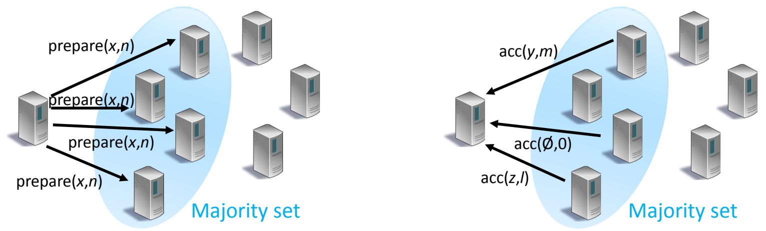
- An acceptor can *accept* any number of proposals
  - An accepted proposal may not necessarily be *chosen*
  - The value of a *chosen proposal* is the *chosen value*
- Any number of proposals can be *choosen*
  - However, if two proposals (x,n) and (y,m) are chosen, then $x = y$

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 $(\emptyset,0)$
  - The proposer learns about accepted proposals!

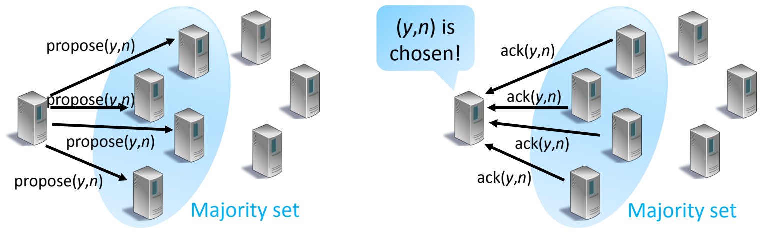
Note that $m < n$!



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$(\emptyset,0)$, 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

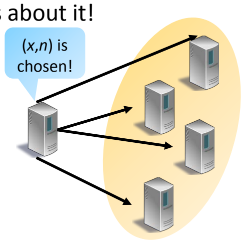
Paxos: Algorithm of Acceptor

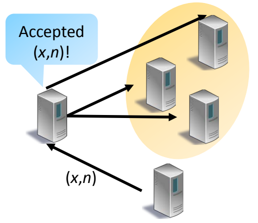
Why persistently?

```
Initialize and store persistently:

$n_{max}$ := 0                                  Largest request number ever received
$(x_{last}, n_{last})$ := (Ø,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…?

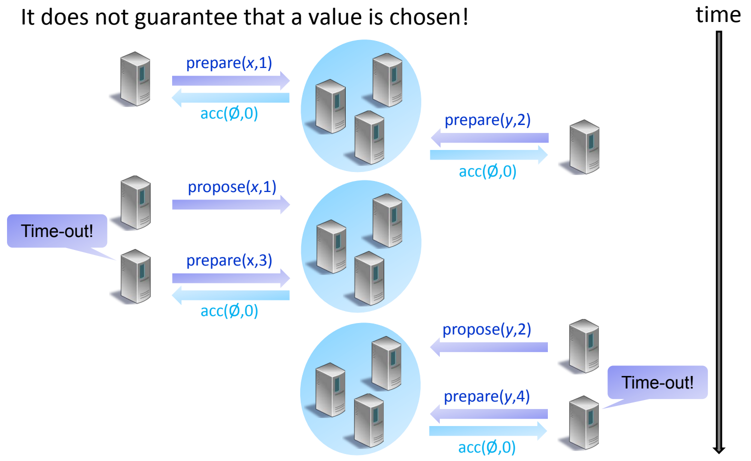
Theorem

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