How to DSYS :)

4. januar 2025 10:42 Marius

Lectures

Lecture 2 (Concurrency) - Threads, Race conditions, Deadlocks, Starvation

Lecture 3 (Computer Networks) - TCP, UDP, 3-way handshake,

Lecture 4 (The Web & Microservices) - RPC, space coupling, Services, REST api, Idempotence, Request, response, gRPC, Microservices

Lecture 5 (Physical Time) - Real Clocks, Time Synchronization, Delay D, External & Internal sync, Christians Algorithm, Berkeley Algoritm, NTP, Physical time

Lecture 6 (Logical Time) - Proccesses actions & events, Happend-before relation, Causality, Lamport Clocks, Vector Clocks, Global state, Snapshot algorithm,

Lecture 7 (Distributed mutual exclusion & Elections) -

- Distributed Mutual Exclusion: Critical section, Properties - Safe, Live, order; Central Server, Token Ring, Ricart & Agrawala, Bandwidth, Client Delay, Throughput, Maekawa algorithm, Fault-tolerance,
- Elections: Properties - safety, liveness, Ring algorithm, Bully algorithm,

Lecture 8 (Consensus, BP & CAP) - Consensus, Algorithms for consensus - Dolev-Strong, Paxos, Raft; Byzantine Generals, Byzantine Faults, FLP Result, CAP Theorem,

Lecture 9 (Replication) - Replication, The 5 stages of replication, Correctness of replication - Linearisability, Sequential consistency; Leader based replication & replic's, Primary failure, Leader-based (passive) replication, Leaderless (active) replication,

Lecture 10 (Raft) - Raft properties, Raft (how it works)

Algorithms

Keywords

General:
Asynchronous system
Synchronous system

(3) Computer Networks:
Race conditions
Deadlock
TCP
Sliding window
UDP
3-way handshake

(4) Microservices and Web:
RPC
Space coupling, space uncoupling
Concurrency
Idempotence
gRPC
Microservices

(5) Physical time:
Time Synchronization
Delay D
External & Internal sync
Christians Algorithm
Berkeley Algoritm
NTP

(6) Logical Time:
Proccesses actions & events
Happend-before relation
Causality
Lamport Clocks
Vector Clocks
Global state
Snapshot algorithm

(7) Distributed Mutual Exclusion and Elections:
Distributed Mutual Exclusion: Critical section,
Properties - Safe, Live, order;
Central Server,
Token Ring,
Ricart & Agrawala,
Bandwidth,
Client Delay,
Throughput,
Maekawa algorithm,
Fault-tolerance,

Elections: Properties - safety, liveness;
Ring algorithm,
Bully algorithm,

(8) Consensus, Byzantine Generals, and CAP Theorem:
Consensus,
Algorithms for consensus - Dolev-Strong, Paxos, Raft;
Byzantine Generals,
Byzantine Faults,
FLP Result,
CAP Theorem,

(9) Replication:
Replication, The 5 stages of replication,
Correctness of replication - Linearisability, Sequential consistency;
Leader based replication & replic's,
Primary failure,
Leader-based (passive) replication,
Leaderless (active) replication,

(10)
Raft
Properties of raft

Lecture 2 (Concurrency)

Threads

In go we can run mutilple threads with goroutines that can run in parallel. These threads can access the same memory and ressources.

- Can lead to race conditions and deadlocks.

Race condition

- If two people are trying to write at the same time
- We cant be sure of the outcome.

We can use locks to fix this. However, this can lead to deadlocks.

Deadlocks
- When threads are wating forever

Starvation
If one process always get allowed access to the resouce and all others dont, becayse they wait.
Solution is to randomize, make a priotiry or timeouts.

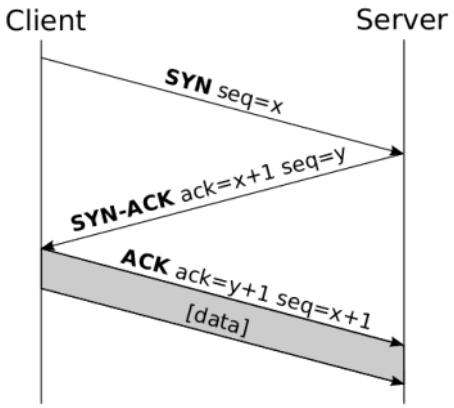
Lecture 3 (Computer Networks)
When two processes are communicating, messeages can fail - dropped lost…
To fix this we can resend, use timeouts, Acknowledgments, sequence numbers and so on.

User Datagram Protocol (UDP)
Is unreliable and messages may be lost
Send datagram.
Is simple, fast, had little overhead and is stateless.

Transmission Control Protocol (TCP)
Is reliable. Uses the 3-way handshake to establish connection

Guarentees:
- In-order-delivery: data packets are delivered to the application in the same order they were sent.
- Reliable Delivery: Data is delivered accuratly and in the correct order.
- Included error detection and correction.

TCP: 3-way Handshake
Uses a random seqence number
Used to ensure that server and client is in sync. Ready to send ordered data.
SYN = lets talk
ACK = Next expected sequence number (+1)



Sliding window
Can send mutiple data at once, without getting a aknowledgement back (more efficient).

Lecture 4 (The Web & Microservices)

- **Service** - Doing something for someone else when requested, here an IT system **does something** for another IT system on request

- **RPC** - Remote Procedure Call, ie a **call**, that leads to procedural **code executed on another computer**, connected over the network

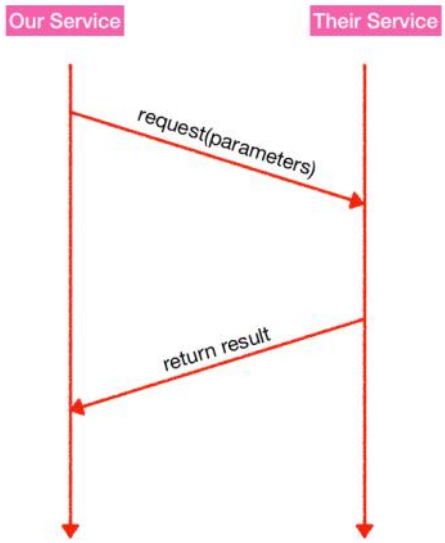
- **API** - Application Programming Interface, ie the things you need to do, to call code on another computer

Microservices
Is splitting processes into smaller parts
Can be independently deployed and are loosly coupled
Use concurrency

- Microservices are small, independent, and loosely coupled.
- Each service is a separate codebase, managed by a small dev team
- Services can be deployed independently
- Services are responsible for persisting their own state
- Services communicate by using well-defined APIs
- Internal implementation details are hidden

No tech login, scalable
But are more complex, harder to test we gotta ensure data integrity

RPC
Like execution a function on a remote server like it was local.
We make a request and get a respond

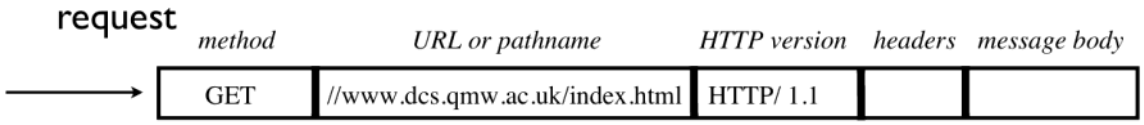


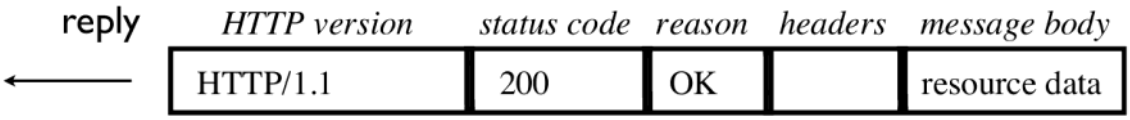
We need to chose what to do with lost messages

At-least-once
Ensure that a message gets delivered at least once and maybe more times. Prioritze ensureing delivery over avoiding duplicates

We want to achieve exactly once, but is very hard.

HTTP





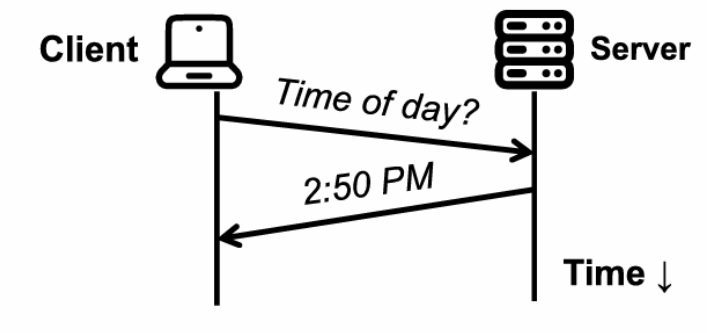
gRPC
Go RPC

Synchronous
- Time decoupled, space decoupled
- Synchronous multiplies downtime
- Your system is the product of downtime of components!
- Synchronous implies waiting
- In reality, you don't really need it

Lecture 5 (Physical Time)

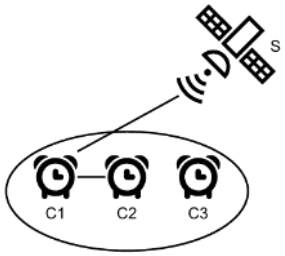
Real clocks differ and drift.

There is a delay if we ask another server what time it is.



In ideal network, if we know the delay d and the send time T, we can do d + T
The reciever can then find the time by D + T

Real networks
In real life, networks are asynchronous, meaning delays are unpredictable (unbounded) and messages may never arrive.



Internal synchronization
Internal sync means that all clocks in a distributed systems are in sync with eachother within a bound D.

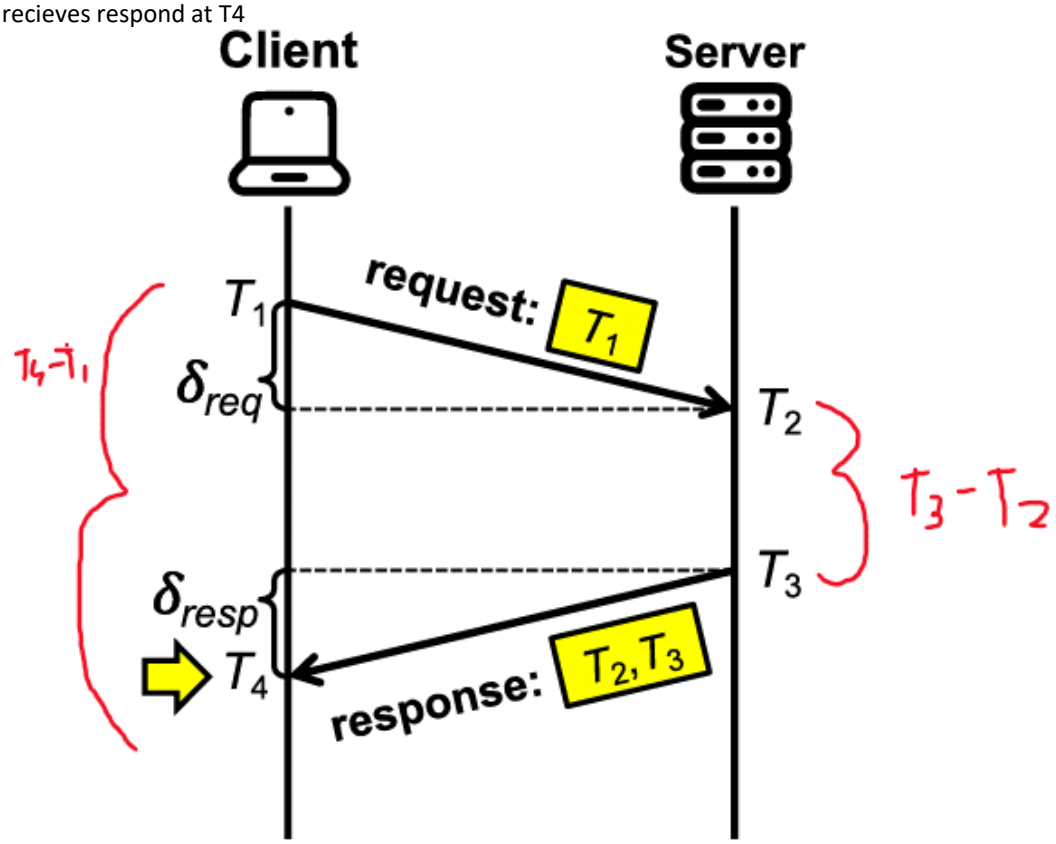
External synchronization
External sync means that a clock is in sync with a reference clock

External does imply internal. If all clocks are externaly sync with a reference clock within the bound D. They must all also be within bound D of eachother.

Internal does not imply external. If all clocks in a system is in sync within bound D, the external clock is not nessesarily within D.

Christians Algorithm
Is an algorithm for synchronizing clocks in distributed systems.

Clients sends a request to Server, with timesamp T1
Server recieves at T2
Server sends respond at T3
Client recieves respond at T4



Assuming that the network delay from client -> server and for server -> client i symmetric we can estimate the delay in total as : δ=(T4−T1)−(T3−T2)

δ is the delay for the request and respond to arrive. To get one, we do δ/2

The client should therefore sync its clock to T3 + δ/2

BAM! The client have synced with the server.

Berkeley algorithm (variant of christians algorithm)

1. A server request timestamps from all clients.
2. It then uses christians algorithm to adjust for delays (between server and each client)
3. It then takes the average time.
4. It then sends out the new calculated time to all clients.

Good when there is no external clock.

Lecture 6 (Logical Time)

Happens-before relation

Describe how events are ordered.

If e -> f, e happend before f.
If f send to g, f happend before g
It is transtive, meaning e also happend before g.

(if you can draw

If two event are not in a happend before relation, they concurrent

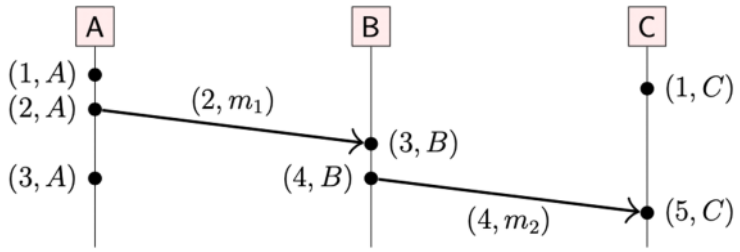
Causality

Is the relationship between events, where one event can cause the other. If a happend before b, it encodes that a potentially caused b.

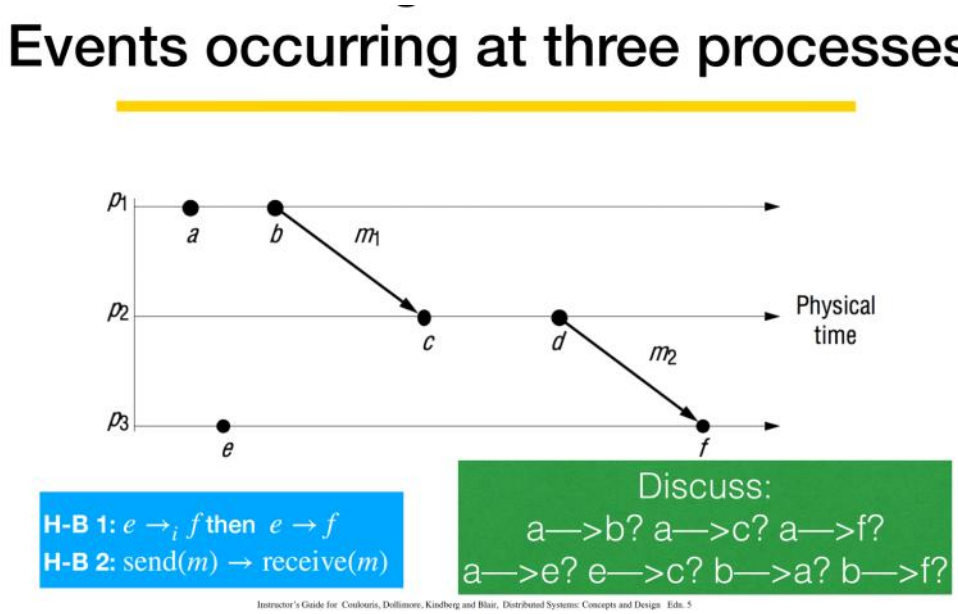
Lamport clocks
Designed to capture causality.

Every process has a counter, that captures when an event happen.
When sending a message to another process, include the timstamp.

- The process that recieve the message should then take the max() of its and the messages timetamp and + 1.

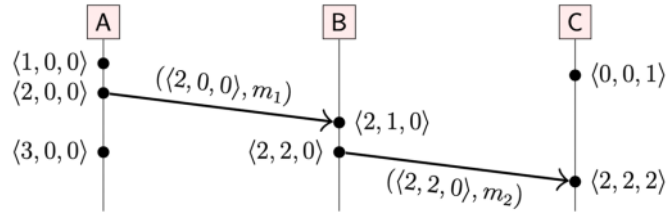


Lamport clocks are comparable if they are in a total order.



Green box:
We know that a -> b, a->c, a->f
We dont know if a->e, e->c
We know that b do not imply a.

Vector clocks



Each process has a vector time. It is more detalied than lamport clocks because it keep tracks of each timestamp in each process. Each vector clock has an array of length corresponding to the amount of processes

Disadvantages:
• Storage and message payload proportional to the number of processes
• Dimension N is unavoidable for concurrency check between two events
• More efficient with e.g. Matrix Clocks (not covered today)'

Lecture 7 (Distributed mutual exclusion & Elections)

Critical section
Part of the code that is accessed as shared ressource.
Shared database

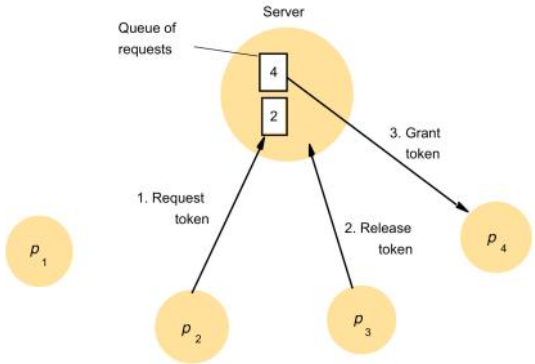
Distributed mutual exclusion
Ensure that only one process can access the crititical section at a time
- If not it can lead to race conditions and deadlocks

We assume the systems is asynchronous, have reliable messages delivery and processes cannot crash.

Properties of distributed mutual exclusion
- ME1 (safe): at most one process in CS at any time
- ME2 (live): request to enter/exit eventually succeeds
- ME3 (order): entry to CS respects happens-before of enter() calls

PROTOCOLS FOR ACHIEVING DISTRIBUTED MUTUAL EXCLUSION:

Central Server
The central server manages a queue for acces to the critical section.
When a process want access to CS it request a request token. The server can then give an Grant token to the process that can access the CS. Once a process is done, and sends a release message to the server



- ME1: it is safe
- ME2: it is liveness
- ME3: Ordering depends on how you implement it. Request could come in parallel - we cant be sure which came first.
- If the server fails, it all fails.

Token ring
No single server
The processes solve the problem by them self.
The processes are like a ring, it has one it recieves from and one it send to.

They have a token that they send around. The process that have the token, can access the critical section (data).

- If the process does not want to access the critical section, and they have the section, they send it to the next.

ME1: It is safe
ME2: They can all at somepoint access the critical section
ME3: No direct order. The tokens has to be passed around.
This solution is relying on the topology (the ring), if one process fails, the ring is broken.

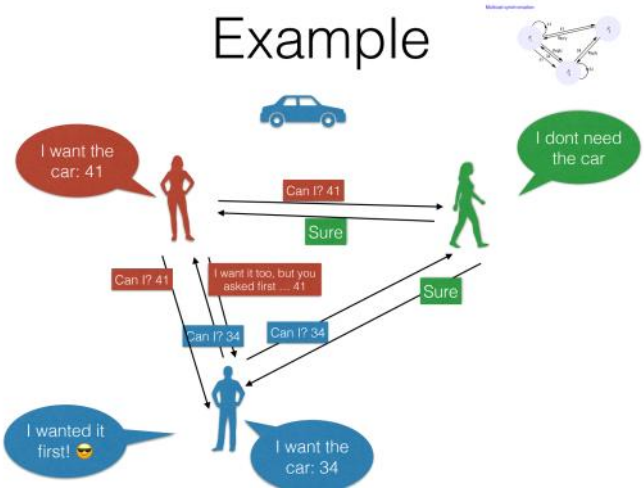
Ricart & Agrawala

Your get rid of two things from the other solutions

- No central server
- They are not organised in a particular way, so the processes does not need a ring topology.

The basic principle:

- It has broadcasting (sending a message to all the other proccess)
- When it want to access the CS, it makes a multi-broadcast to all other processes.
  - We use lamport stamps.
- The messages are revieved by the other processes in a queue (general queue) where they will be ordered by their timestamp.
- Its kinda like working. They all decide who is next to the CS.
- If two processes want the process at almost the same time, they see who was the smallest lamport timestamp.



ME1, ME2, ME2 all good

What happens if one process fails?

- Go by voting and not everyone needs to reply.
- Leader election

Comparing the different protocols

We look at:

- Bandwidth (max number of messages)
- Client delay (delay for entry() exit() operations)
- Sync delay (time between entry/exit() operations)

	Server	Token-ring	Ricart & Agrawala	
Bandwidth	2 1	(continuous)	2(N-1) N-1	enter() exit()
Client delay	2 (rt) 0	0…N (avg .5N) 1	2(N-1) 0	enter() exit()
Sync delay	2 (rt)	1…N (avg .5N)	1	

rt = roundtrip time

Elections

We want to elect a leader (consensus - agreeing on who should be the leader)

- Any process should be able to call the election if a leader crashed (we need a safety system)
- There may be concurrent calls

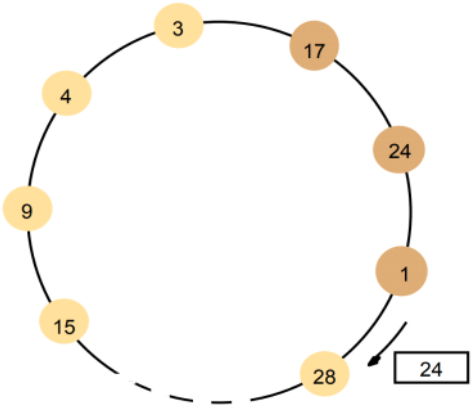
We have the properties:

- E1(safety): there is always a leader and they all agree on this leader. No two processes should have different leaders.
- E2(liveness): The algorithms works and a leader will eventually be elected.

SOLUTION

Ring

Assume: Synchronous system, no failures, logical ring structure,
Highest process id wins elections, if a messages circles around until it comes back to a process, it is the winner.



Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown in a darker colour

Because its a sync setting we have a handshake and we know if something doesnt respond.
If one crashed, we could have double links

Evaluation
Bandwidth: nr. of messages sent: Worst-case, single election: 3N-1
Turn-around: max nr. of sequential messages in a run: Worst-case, single election: 3N-1

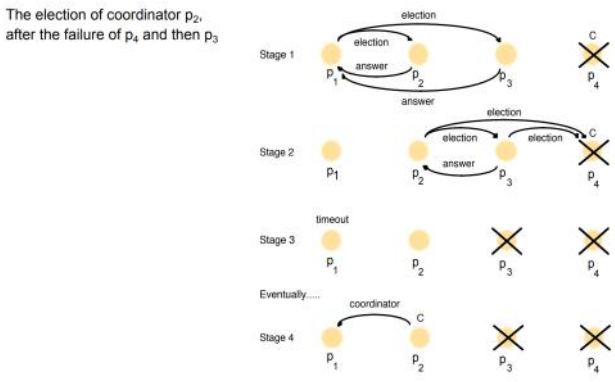
Bully

Assume sync, crashes are allowed, can detect crashed by timeouts
All processes know of each other

It is called bully, because among the processes that are allowed, the one with highest ID is the leader.
Highest id process elects itself and send it all other with coordinator messages to all.

There are times way u may thing u are the leader, but eventually u could be bullied down by a higher id.
If you dont get a answer from all other, then you are the one with the highest ID.

- U send a request, and see if you have the highest ID.



P4 crashes. P1 realises, so it calls for an election, and ask p2 and p3.
Everyone knows each other ID. So a high ID wont ask lower id if they want to be the leader. They only ask upwards. Here p4 is dead, so it cant be the leader.
So the process that realises the crashes, will ask all the higher processes.

If a process does not respond by some timout, it is assumed dead

Lecture 8 (Consensus, BP & CAP)

Consensus
Is about making computers and processes agree on stuff
In distributed systems we have no shared clock or memory.

A network can be:
- Asynchronous () (Impossible to achieve deterministic consensus if a single process can fail)
- Partially Synchronous ()
- Synchronous

The Conensus Problem
You have a set of nodes / processes. They need to agree on something / some value from some set D.
- They reach a consensus when they all agree. Fx

- The protocol must satisfy:
  Termination: each p_i sets its d_i
  Agreement: each correct p_i decides on the same d_i
  Integrity: If correct p_i proposes value d_i, then p_i decides on d_i.

Termination: Each process finish running the consensus algorithm.
Agreement: Each process decides on the same value.
Integrity: If a process at some point propose a value it sticks with it.
- Need to hold for processes that are "correct" / not faulty

Algorithms for consensus
- Dolev-Strong (simple)
- Paxos
- Raft
- Byzantine generals (variant)

Byzantine Generals

To reach consensus we need some assumptions, else its basicly impossible:
Reach even tho some processes might be faulty.

- Only one process, the commander, proposes a value. (The commander is responsible for suggesting a value to other processes)
- Agreement: Correct processes must agree on some value. (All correct and non faulty processes must agree on the same value)
- Integrity: If the commander is correct, they agree on his value. (If the commander is not faulty, all correct processes must agree)

Byzantine faults
Means that a node might be corrupt and send malicous messages or act in other wierd ways.

Byzantine fault tolerance
Be able to defend against failures that can stop the system from working correctly.

Byzantine generals and Consensus are equivalent
- **From Consensus to Byzantine Generals**:
  - If we have a protocol for Consensus, we can use it to solve the Byzantine Generals problem.
  - The commander sends its proposed value to all processes.
  - Each process then runs the Consensus protocol (C) with the value it received from the commander.
  - This ensures that all correct processes agree on the same value, achieving consensus despite potential Byzantine faults.

- **From Byzantine Generals to Consensus**:
  - If we have a protocol for Byzantine Generals, we can use it to solve the Consensus problem.
  - Assume that the majority of processes are correct.
  - Run the Byzantine Generals protocol once for each process to propose a value.
  - Then, take the majority value from the results of these runs.
  - This ensures that all correct processes agree on the same value, achieving consensus.

Byzantine Generals in a synchronous system
Key points / problems:

- Nodes may be malicous
- Communication between nodes is private, meaning messages may not be broadcasted to all
- There is no solutions with 3 processes and 1 failure (Number of processes N must be 3 times greater than the number of faulty processes)

NO SOLUTIONS IF N <= 3f
- f is faulty processes

FLP Result
A therom that states that in a asynchronous system with crash-stop failures, there is no deterministic consensus algorithm that is guarenteed to terminate. (impossible within some time)

- **Asynchronous System**: A system where there is no global clock, and processes execute at arbitrary speeds.
- **Crash-Stop Failures**: A type of failure where a process may stop functioning (crash) but does not behave maliciously. (in async you cannot determine between a slow and a crashed process.
- **Deterministic Consensus Algorithm**: An algorithm that, given the same initial state and inputs, always produces the same output.

Workarounds:
- Fault masking (Mask the effect of faults, by fx replicating data across mutiple nodes so if one process crash the system can continue)
- Failure detectors
- Randomised behaviour

SUMMARY

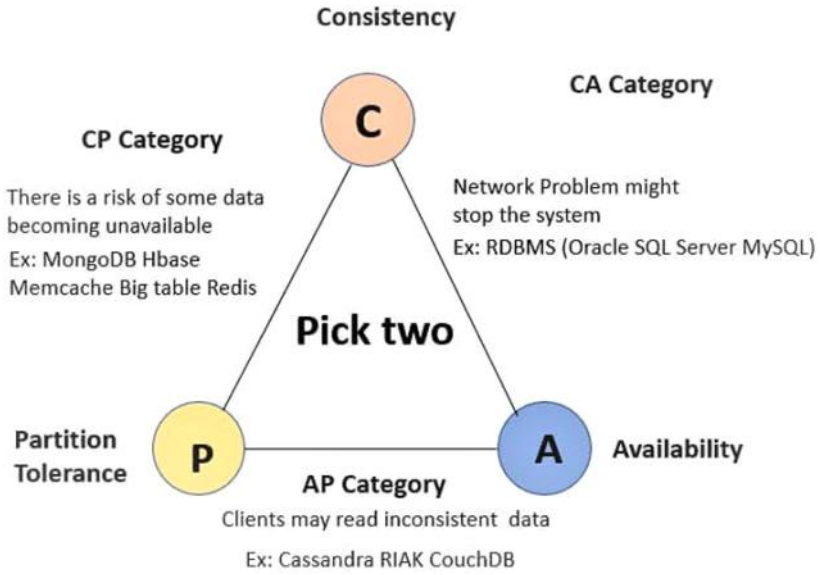
1. We gotta agree on a value
2. Consensus is possible for N > 3f in a sync system
3. Impossible with a single process failure in async system

CAP

- Consistency,
- Availability
- Partition tolerance

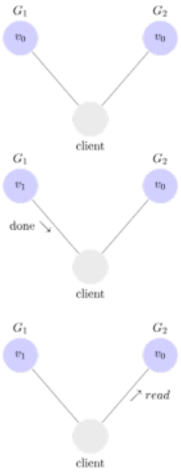
It states its impossible to achieve all three in a distributed data store.

- **Consistency**: Every read receives the most recent write or an error.
- **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **Partition Tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.



CAP Theorem - proof

- Assume for contradiction that there exists a system that is consistent, available, and partition tolerant
- We start by partitioning the system
- Phase α_0:
  - Client requests $G_1 . write(v_1)$ **(available)**
  - G_1 must respond, but cant replicate data **(network partition)**
- Phase α_1:
  - Client requests $G_2 . read() -> v_0$ **(available)**
  - System has an **inconsistent** execution. Contradiction!



If there is a partition of the network and make a rquest to out of data node, it either give you old data or says it cant respond (aka it violates one of them).

Lecture 9 (Replication)

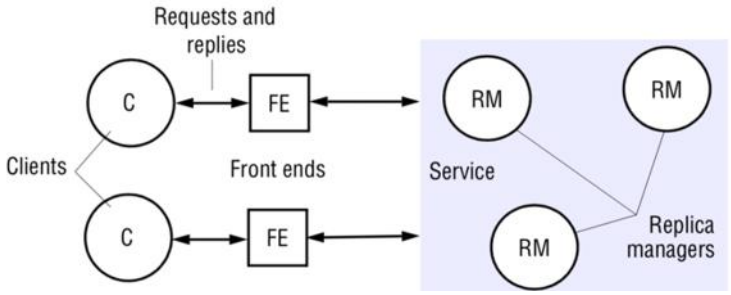
Replication is the process of creating and maintaining mutiple copies of data or services across different nodes.
We do it to improve:

- Fault tolerance
- Availability
- Performance

We will look at it in a asynchronous system (no-handshake, message delivered unknown)
Only crash-stop failures (When something fails it wont come back to live)

Basic architecutre

A basic architectural model for the management of replicated data



Blue server:
- Collection of many nodes (replicas)
- The service it provised is replicated over many nodes

The clients cannot se the many nodes.

If two clients are talking to different replicas - how do we make sure its the same updated data?

The 5 stages of replication

The client should not be affected by how the server is implemented.
- Request
  - Client send som request and will later get a response (last point)
  - Alot of stuff can happen in between:
- Coordination
  - Replicas inside the server have to coordinate. Which request do we do first?
  - Sometimes its impossible to figure out what to do first
- Execution
- Agreement
  - What if we recieve to different request we need to agree on something.
  - Consensus - They have to agree on something.
- Respone

Fault tolerance

Is tolerant up to f failures

Replicas follow specific rules and protocol to maintain consistency and correctness

Correctness - The result even with <= f failures, should be the same as if it was a single server

- f+1 replica managers can sustain f crashes

Linearisability

- For any execution of the system and we do an execution - we can always find a order on the system of the request.
- We want the interleaving to be consistent with the real life order of the operations.
  (If i take the system as a whole algorithm, then it does the same as running the replicated system. - Like if there were only one node in the server).
  It ensures that operations on shared data appear to take effect instantaneously at some point between their invocation and response

  There should be a real life order

Sequential consistency

We can achieve this as long as there is order from each clients perspective. It appear to be in order.

- Linearisable implies sequentially consistent
  [Real-time respects program order]
- Sequentially consistent does not imply linearisable:

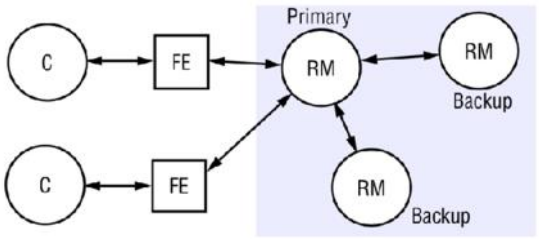
Client 1:	Client 2:
$setBalance_B(x, 1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y, 2)$	

We have two main types of replication:

Leader-based (passive) replication
Good for correctness, but a little slow

We have a primary replica where everything has to go though
We have backup replicas, if the main one fails

Figure 18.3 The passive (primary-backup) model for fault tolerance

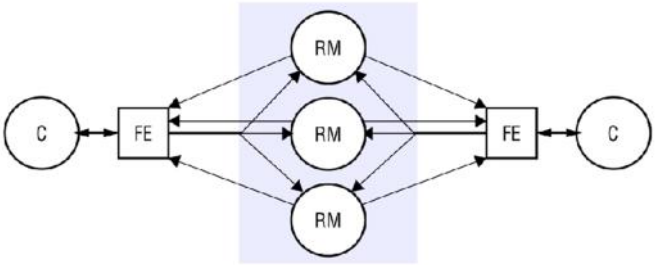


- 1. **Request**: The front end issues the request, containing a unique identifier, to the primary replica manager.
- 2. **Coordination**: The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request, and if so it simply resends the response.
- 3. **Execution**: The primary executes the request and stores the response.
- 4. **Agreement**: If the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
- 5. **Response**: The primary responds to the front end, which hands the response back to the client.
- **Think**: why do we wait for acknowledgements?

Correctness under primary failure
If the primary replicas fails, it must be replaced by one backup replica.

- The primary must be replaced by a unique backup
- The replica managers that survive must agree on which operations had been performed at the point when the replacement primary takes over.
- f+1 replica managers can sustain f crashes
- **Beware**: split brain scenario - if 2 clients begin using 2 backups

Leaderless (Active) replication
This does not guarentee Linearisability



When a client connect, it connects to all replicas. All replicas process the request individually and then they all send a respond back. The client now has the burden of figureing out the right data (take the one most responded with)

- 1. Request
  The front adds unique identifier to the request, reliably-ordered multicasts it to the replica managers, The front end is assumed to fail by crashing at worst.
  It does not issue the next request until it has received a response.
- 2. Coordination:
  The group communication system delivers the request to every correct replica manager in the same (total) order.
- 3. Execution:
  Every replica manager executes the request deterministically.
  The response contains the client's unique request identifier.
- 4. Agreement: Unnecessary due to multicast
- 5. Response:
  Each replica manager sends its response to the front end.
- This achieves sequential consistency, not linearisability

Passive (Leader-Based) Replication

- **Primary-Backup Model**: One primary replica handles all write operations and propagates changes to backup replicas.
- **Linearizability**: Achieves linearizability by ensuring that all updates are propagated to backups in a strict order. This means the system behaves as if there is a single, correct copy of the data, and all operations appear to occur instantaneously at some point between their invocation and their response.
- **Sequential Consistency**: Since linearizability implies sequential consistency, passive replication also achieves sequential consistency.

Active (Leaderless) Replication

- **All Replicas Equal**: All replicas can handle write operations, and changes are propagated to all replicas.
- **Sequential Consistency**: Achieves sequential consistency by ensuring that all replicas process requests in the same order. This means the operations of all processes are executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by the program.

- **Linearizability**: Does not achieve linearizability because it does not guarantee that operations appear to occur instantaneously at some point between their invocation and their response.

Lecture 10 (Raft)

- Consists of two phases:
  - Leader Election
  - Log Replication

Properties ensured by Raft

Election Safety: at most one leader can be elected in a given term. §5.2
Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3
Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3
Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4
State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3