Zusammenfassung Distributed Systems

What is a Distributed System

↳ multiple Computers communicating via a network and working together to solve a task

↳ resources and processes get spread across all machines

Design Goals

→ 3 main aspects to consider:

- Storage
- Communication
- Computation

→ objective is to find ways to mitigate complexity

→ Goals to keep in mind:

- Scalability
- Fault-tolerance
- Data Consistency
- Distribution Transparency

Scalability

↳ Ability to adapt to new size & scope requirements

Limiting Factors:

- Computation Power
- Storage Space
- Bandwith

Solutions

- vertical scaling (scaling up)
  ↳ adding more resources to one machine
- horizontal scaling (scaling out)
  ↳ adding more machines

Dimensions

- Size Scalability → nr of users/processes
- geographical scalability → distance between nodes
- administrative scalability → administrative domains
  ↳ security, policies...

Fault-tolerance

↳ the ability to continue operating even if components of the system fail

Problem Facets

- Availability → always ready to use
- Reliability → always available
- Safety → controlled response on failure
- Maintainability / Recoverability → easy to repair

Solutions

- recovery checkpoints
- Data replication

Data Consistency

↳ Does the replicated Data match the other copies

- **Strong consistency**
  ↳ retrieves most recently written value

- **Eventual consistency**
  ↳ after a write it takes time for all copies to be upto date
    ↳ read only gets a previously written value, may be old.

Distribution Transparency

↳ Is the distribution of processes & resources visible to users
↳ hidding away stuff

Types of transparency

- Access
- Location
- Relocation
- Migration
- Replication
- Concurrency
- Failure

↳ hard to hide all (poformance, latencies etc.)

Fundamental Challenges

false assumptions:

- The network is - reliable - Secure - homogeneous
- The topology does not charge
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

=> keep in mind when developing and account for them

CAP Theorem

↳ it is impossible for a Distributed Data store to simoultanously have more then two of the following properties:

C - **Consistency** → every read returns the most recent value (or error)

A - **Availability** → every request returns a non error value

P - **Partition Tolerance** → continuous operation even on failures

=> focus on two and the third will be lacking

Map Reduce

↳ programming model for processing large data sets accros multiple machines (also possible on single machines)

Process:

1. Map
   ↳ Data set is converted to a set of data, where elements get broken down into key-value pairs
2. Reduce
   ↳ After mapping, takes all created tuples and combines them into the smallest possible set of tuples

Advantages

- Scalability → data can be split into smaler chunks for mapping → parallel processing
- Fault-tolerance → auto retry on failure and data is replicated
- Flexibility → structure of data doesn't matter

Limitations

- High overhead from splitting and organizing
- Botlleneck of disk read & write (no in memory processing)
- Latency

Remote Procedure Call (RPC)

↳ client makes a request, server carries out the request and returns data to client. client doesn't know the process of the server to get the data.

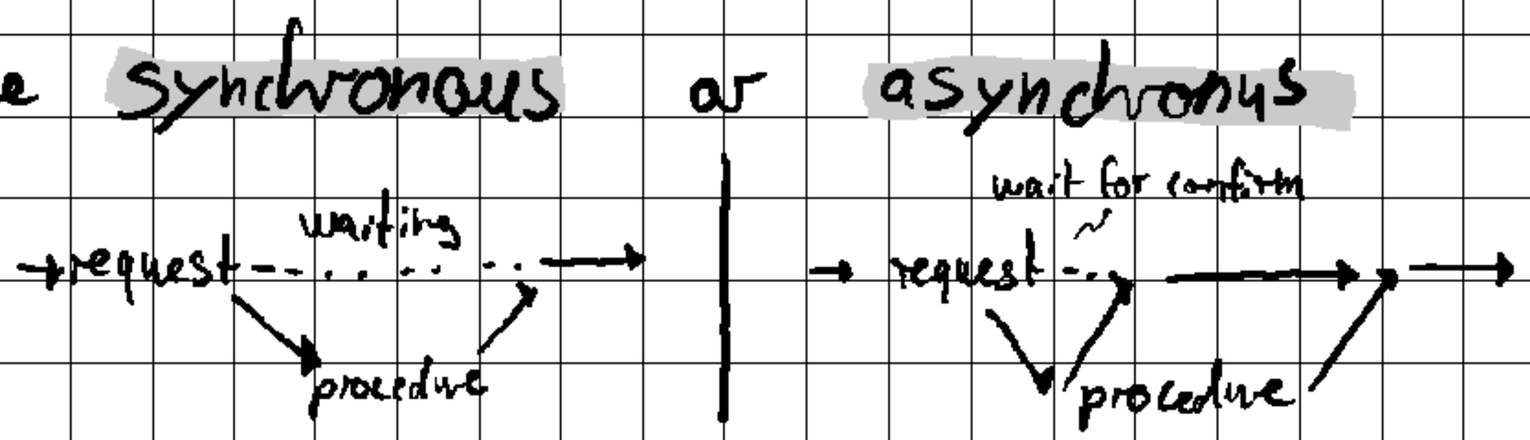
In Distributed Systems this procedures get carried out on different machines.

A machine does this by calling a **Stub** looks like the procedure but packets the request and sends it to the sever stub.

client → client stub → server stub → procedure

⇒ for the client it looks like he did it himself.

- RPCs can be **synchronous** or **asynchronous**



challerges

- access transparency cannot be realized
  ↳ process may crash / delay
- client & server written in different languages
  ↳ need to agree on uniform encoding

Representational State Transfer (REST)

↳ architectural style for building scalable, performant and maintainable systems

Constraints

client - server

- client → lightweight user interface, no stored data
- server → stores data & implements functionality

Stateless

↳ all requests contain the needed data for the server to be able to fulfill the task

↳ no state saved on server

Cache

↳ a client can cache responses to display them instead of running the same request again (request has to be cachable)

Uniform Interface

↳ Components interact through a general uniform interface that hides away their implementation details

Layered System

↳ hirarchial layers of components with knowledge of only the neighboring layer

Code on Demand (optional)

↳ Client can be extended to fetch and execute Code

REST API

→ must be self descriptive

↳ Hypermedia-driven Interactions

↳ transitions to next state must be advertised in the hypermedia (ex: google.com/search?q=searchterm)

Richardson Maturity Model

↳ way to asses the RESTfullness of an API

- **Level 0** : The Swamp of plain old xml (POX)

↳ one URI, one HTTP method

↳ functionality is driven through body

- **Level 1** : Resources

↳ multiple URIs, single HTTP method

↳ URI defines resource (resource-oriented)

- **Level 2** : HTTP Verbs

↳ multiple URIs, multiple HTTP methods, status codes headers

↳ multiple Endpoints with individual resources

- **Level 3** : Hypermedia Controls

↳ responses provide links to related actions

↳ self descriptive API

⇒ majority of Webservices are Level 2

Interaction Style of Web Services

RPC connectors

↳ system executes a Subroutine in another system

↳ website button calls specific method on Web Server

REST connectors (RESTful)

↳ HTTP request centred around resource

↳ Place order via Website button, get a new URL returned with order number to tracle order

Remote Data Access connectors (RDA)

↳ directly access remote data structure to manipulate it

↳ Website Button creates database entry & retrieves Id for tracking

Event connectors (Eventfull)

↳ trigger Event → subscribes to update of Event Status

↳ Website Button triggers Payment event, subscribes to a chargeListener that gets trigged by a PaymentSucess event

Process

↳ The abstraction of a running program

↳ allows execution of multiple programs on the same CPU

Processes are manged by the OS. It has a process table with one entry for each process

↳ process control block

Process control block

↳ conteins information about the state of the process and all the other information needed for running (file-, memory- management, accounting → time used, etc...)

Address Space

↳ each Process has its own address space

↳ no Interferance with other processes

- Text-Segment → program code (read only)
- Data-Segment → variables and constants
- Heap-Segment → dynamically allocated memory
- Stack-Segment → one frame per procedure (method) waiting to be executed

↳ a process can create child processes to help with its job

Threads

↳ multiple threads allow a process to run tasks simultaniously

Multiple threads operate in the same address space but with their own stack

Thread Scheduler

↳ manages threads

↳ multiplexes them on the CPU based on their state (runnable, waiting, blocked, terminated etc.)

Drawbacks

- no protection from OS when using same memory
- one thread fails → all threads fail

Use in Distributed Systems

- Hide Network latency
  ↳ Speed up through parallel processing on multiple servers
- Concurrency
  ↳ Interactions with multiple machines

Mobile Coding Style

↳ Code that can be sent and executed on a different machine (migrating code)

Also possible for process migration → Passing not only code, but also the state

⇒ client and/or server can initiate migration

Reasons

- poformance → distribute load (all machine have an equal over-all load
  ↳ minimize network communication
  → process where the data is needed
- privacy & security → move the code to the protected data
- flexibility → move code to where it is needed

Remote Evaluation

↳ execute code remotly because of a lack of resources

Code-on-Demand

↳ client lacks the code, but has the resources

Mobile Agents

↳ move code autonomously throughout a network

↳ both client & server can initiate migration

Distributed Computation: Spark

↳ Generalization of MapReduce into Data Flows

Data flows can - include multiple stages

- use diverse transformations (not only map, reduce)

Resilient Distributed Data Set (RDD)

↳ main abstraction in Spark

- Immutable → collection is read only
- Resilient → built in fault-tolerance
- cacheable in memory across machines

Two Types of Operations

- transformations → input RDD output new RDD
  ↳ Special transformations : cache() & persist()
    ↳ for checkpointing & avoid redundant computations
- actions → operations to compute a RDD to return a value
  ↳ Common actions :
    - count() → return number of elements in RDD
    - collect() → return elements of RDD
    - saveAsTextFile → saves the RDD to external storage as a text file

4 Steps of Spark execution

Step 1
RDD Objects
- build directed asyclic graph

⇨

Step 2
DAG Scheduler
- Split RDD Graph into stages of tasks
- Submit each stage as ready

⇨

Step 3
Task Scheduler
- Launch tasks
- retry failed or straggeling tasks

⇨

Step 4
Spark Worker
- execute tasks
- store and serve blocks

Partitioning in Spark

too few partitions:

- less concurrency than what could be achieved with same computing power
- greater memory preassue for large partitions

too many partitions:

→ too much schedule overhead

tuning

- lower bound → ex: at least 2x number of CPU cores in cluster
- upper bound → ex: ensure tasks take at least 100 ms

⇨ Spark supports **multi-stage processing** with **in-memory caching** between transformations.
Lineage Graphs allows Spark to optimize execution and recover from failure by recomputing only lost partitions

Distributed Storage

- horizontal scalability
  - ↳ distribute data over many servers (sharding)
  - ↳ many servers → faults becom rule, rather than exception
    - ↳ fault-tolerance → replicate the data
      - ↳ replication → potential inconsistencies
        - ↳ better consistency → more communication
          - ↳ more communication → lower performance

⇒ no silver bullet, just trade-offs

Good vs Bad consistency

Good → clients always read the same date (strong)

Bad → clients may read different data (weak)

Google File System (GFS)

- files are split into 64MB chunks
- chunks are distributed across multiple chunk servers
- each chunk has multiple replicas (3+)
- single master manages file namespace & chunk distribution
- client interacts with master and chunk servers through RPCs

↳ master can be a huge Bottleneck

↳ minimize master involvement

reading from a file

↳ request → get addresses of replicas → go to address of nearest

writing to a file

→ master selects a primary replica & secondaries

↳ client pushes data to closest

↳ closest sends it to the next etc..

↳ after all replica have received the data, they report to the primary

↳ client gives the commit order to the primary

↳ primary sends write request to Secondaries

↳ Secondaries reply to primary with success/error/timeout

↳ primary reports to client with success/error

additional Features

- Garbage Collector
  - ↳ files are marked for deletion and the space is then periodically reclaimed
- Snapshot Capability
  - ↳ create quick snapshots of files or directories by copying metadata instead of the actual data

Influence

↳ GFS was inspired by Hadoop Distributed File System (HDFS)
  ↳ better Consistency than GFS

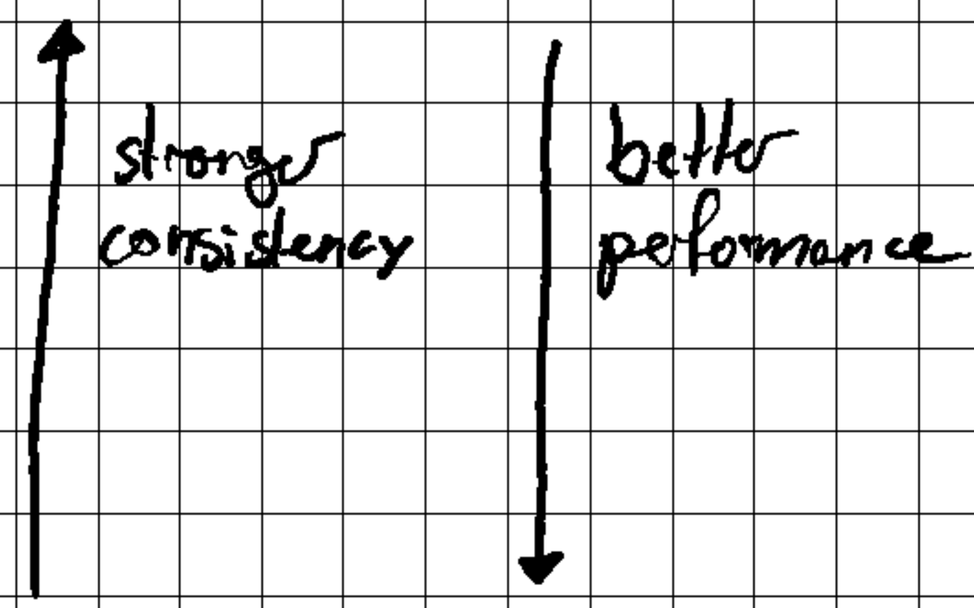
Successor

↳ GFS was replaced by Colossus

Data Consistency Models

↳ ways to ensue consistent data over multiple machines

- Linearizability
- Sequential Consistency
- Causal Consistency
- Eventual Consistency



⇒ Its a trade-off convenience vs speed

Operation f manipulates an object

↳ Operation starts at time f_s
stops at time f_e

Event E is a set of operations on one or more nodes

⇒ if two operations f & g have $f_e < g_s$ then
occurs f before g ($f \rightarrow g$)
if than $a \rightarrow b$ then also $f \rightarrow h$

If events have no connection and occur at the same time
they are concurrent ($a \| b$)

Linearizability

↳ strongest consistency model

↳ all operations occur instantaneously

⇒ each operation f appears instantaneously between f_s and f_c

An Event E is Linearizable if thre is a sequential execution of an operation with its linearization points

Sequential Consistency

↳ ensures operations appear to occur in some order and all nodes agree on that order

An Event is sequential consistent if there is a way for a set of operations to appear in a order that matches the outcome

↳ sequentially consistent

Causal Consistency

↳ related operations will always appear in order

⇒ ex: social media posts may be seen in different orders but comments related to one post will always be in the correct order.

⇒ if related → consistent, if not they can be but dont have to

Eventual Consistency

↳ if no new updates are made, eventually all replicas will be in the same state

⇒ Sometimes a user may read an old value. After some time the user will read the current value

Logical Clock

Strong logical clock

↳ if Event A happens before B, then the timestamp of A has to be smaller than the one of B

↳ timestamps reflect causal relationships

Lamport's Logical Clock

- each node maintains a counter
- local operation encrements the counter
- Sending a message → the Sender attaches his clock
- receiving a message → node clock is set higher than the received clock and its own clock

↳ not a strong logical clock

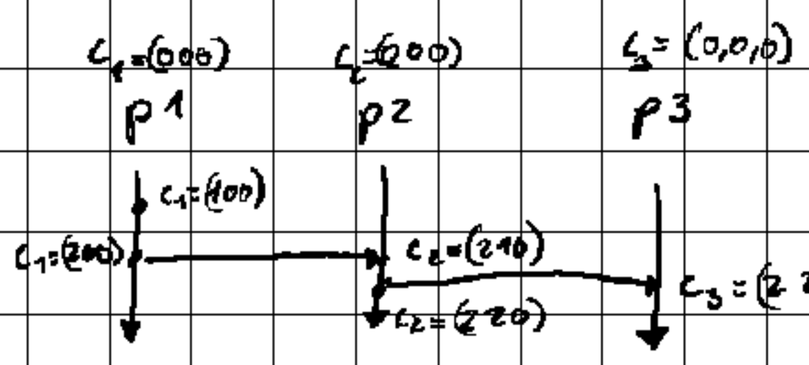
Vector Clock

↳ each node keeps a vector with one component for each node.

If node 1 executes an operation, it increments the component 1 of the vector.

If a message is sent, the vector is attached and the receiver increments its own component and copies the value of each component, if it is higher than it is in his vector.

↳ Strong logical clock



Fault-tolerance

↳ in distributed systems, it is common for a server to fail

Types of failure

- crash failure → server stops
- omission failure → sever fails to respond
- timing failure → serve response is outside accepted time
- response failure → incorrect response
- arbitary (Byzantine) failure → arbitary response at arbitary time

The Byzantine Generals Problem

- similar setting to two generals problem

↳ 3 or more generals have a secure connection, but some generals may be malicious and send false data

No one knows if the other is malicious.

Assumptions about a Distributed System:

- Synchronous → known fixed time limit on tasks & messages
- asynchronous → no fixed time limits
- partially synchronous → Some time limits (not precise)

↳ we can use heartbeats / timeouts to detect crashes

Primary - Backup Protocol

↳ each object has a primary replica that coordinates write operations

→ primary is usually fixed but can be reasigned if needed

⇒ Sequential consistency

⇒ single point of failure

Quorum - based Protocol

↳ a majority of nodes have to agree on an operation

↳ vote before taking action

↳ consensus through majority

⇒ can only recover if a minority of nodes fail

Quorum = set of replicas that respond to a request

RAFT Consensus Algorithm

↳ consensus via a leader

- each node can be one of 3 states
  ↳ leader, follower, candidate
- each node has a log
  ↳ log is replicated to all nodes by the leader
- leader manages operations from clients

Leader election

A leader sends periodicall heartbeats and each other node has a random time that it waits for that heartbeat.
If time to receive heartbeat runs out, node becomes candidate and sends out vote requests.
Other nodes vote on the new leader and on a majority the candidate becomes the new leader.

↳ only nodes with an up-to-date log can become leader

↳ if an old leader gets a heartbeat of a newer leader it becomes a follower.

⇒ a log entry gets only comitted if majority of nodes received it.

Distributed Transactions

↳ handle operations that manipulate multiple objects

We use ACID Transaction to maintain data integrety

ACID:

- Atomic → transaction is all or nothing if one operation fails none are applied
- Consistent → transaction goes from one state to another (all system restrictions apply)
- Isolated → transactions are isolated from each other
- Durable → if a transaction completes, the changes are permanent

Lock-based Protocols

↳ controll concurrent access via locks

- exclusive Locks → read and write permission of a single client
- shared locks → read permission for multiple clients

Two-Phase Commit Protocol (2PC)

↳ nodes of a system have to agree on a proposed transaction

Phase 1 : Voting

- coordinator proposes a transaction
- nodes will respond "yes" if they are ready to commit or "no" if they cannot
- Participants will then hold resources and lock required data and will then wait for instructions

Phase 2 : Decision

- If all participants responded with "yes", the coordinator will send a global commit message
- On global commit, nodes execute operations and release locks
- on global abort, nodes will abort operations and release locks

Problems

→ coordinator fail before global commit/abort → nodes are blocked

→ machines have to save current state incase of failure

⇒ widely used, but slow

2PC over Raft

↳ We can use Raft to replicate coordinator and participants of 2PC

↳ high availability & atomic commits

Google Spanner

↳ uses 2PC over Paxos

Paxos → shards data across machines (similar to Raft)

↳ partitions tables horizontally into tablets

↳ tablets are distributed

⇒ operations are slower (still semantically equivalent)

⇒ favours consistency over availability → CP System

Blockchain

Bitcoin addresses the issue of reaching consensus in presence of the Byzantine Poblem by using two key inovations :

- public ledger
  - ↳ public ledger of all transactions (blockchain)
    - ↳ multiple transactions get grouped into a block (size 1MB) this block gets hashed using the output hash of the previous block, creating a new hash that is used to encrypt the next block etc... (new block every 10 min)
- proof of work leader election
  - ↳ candidates race to solve a puzzle
    - ↳ first to solve wins and proceeds to a block of validated transactions
      - ↳ winner also collects transaction fees and some newly created coins

⇒ puzzle : find a value that makes the result hash of a block stat with N zeros

→ value is called Nonce

⇒ if two find the value at the same time, the chain is forked. New blocks get added to the first they see
  - ↳ shorter block gets discarded.

Distributed Intelligence

↳ concept of autonomous agents and multi-agent systems

autonomous agent

↳ can observe environment using sensors and act using actuators.

multi-agent system (MAS)

↳ autonomous agent in a shared environment, interacting

↳ collaborate → work on the same goals

↳ cooperate → work on compatible goals

↳ compete → work on incompatible goals

characteristics:

- decentralized control
- distributed capabilities
- dependencies / conflicts between agents
- coordinated behavior

Semantic Web

↳ link data in a way that makes it understandable for machines by adding various standards

⇒ RDF → resource description Framework

knowledge Graphs

↳ Graph where nodes represent entities of interest and the edges represent their relations

⇒ queries can go along the edges

SOLID (Social Linked Data)

↳ conventions and tools to decentralize social data

↳ users store their data in pods (personal online data storage) and give permission to other apps to use this data