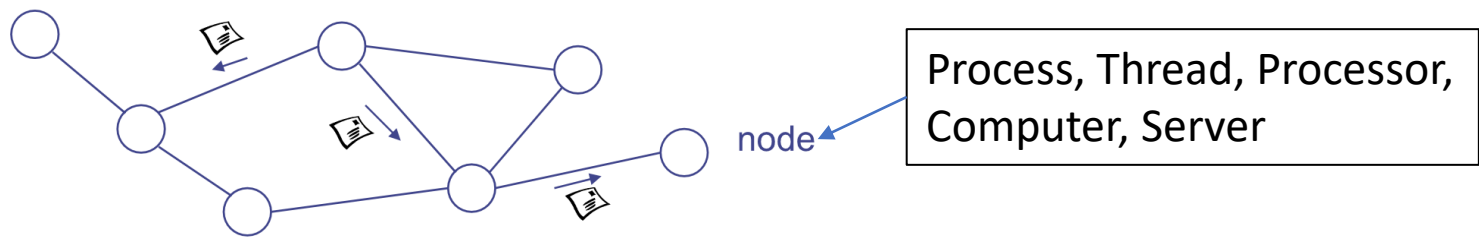


Zusammenfassung

Verteilte Systeme

Hauptproblem: Es treten Fehler auf;
es treten unbekannt lange zeitliche Verzögerungen auf.

Models



Node Model
Send <ul style="list-style-type: none">Strictly blocking (synchronous)Blocking (asynchronous)Strictly non-blocking (asynchronous)
Receive <ul style="list-style-type: none">Blocking (synchronous)Nonblocking (asynchronous)
(bold = common combination)
Unidirectional / bidirectional

Fault models
<ul style="list-style-type: none">Fault (defect in system, e.g. bitflip)Error (actual & intended behavior differ)Failure (system does not behave according to specs; visible from outside the system)



Wire format (serialization)
<ul style="list-style-type: none">Neutral representationE.g. XML, JSON



Node Model		
Type	Sender	Receiver
Unicast	1	1
Broadcast	1	?
Multicast	1	x
Convergecast	x	1

Type A: Message
Packet / Stream of data Do all messages take equally long? When is it allowed to send data? Synchronous / asynchronous / in between (time / size limit for message)

Delivery models
<ul style="list-style-type: none">Exactly once (impossible)At least once (impossible)At most once (feasible)

Type B: Shared Memory
<ul style="list-style-type: none">CentralDistributed

Timing Model
Nodes equally fast / in sync (lock-step model, PRAM)?

Communication

Client (service accessor)

AJAX

- Asynchronous programming paradigm
- Send request, provide callback, wait for event

Request / Reply

- Synchronous**
(block until response)
- Asynchronous**
(do other stuff after request is sent)



Server (service provider)

Frontend

Service 1, 2, ..., n

- Persistent server: started in bootup process (autostart)
- Statefull vs. stateless

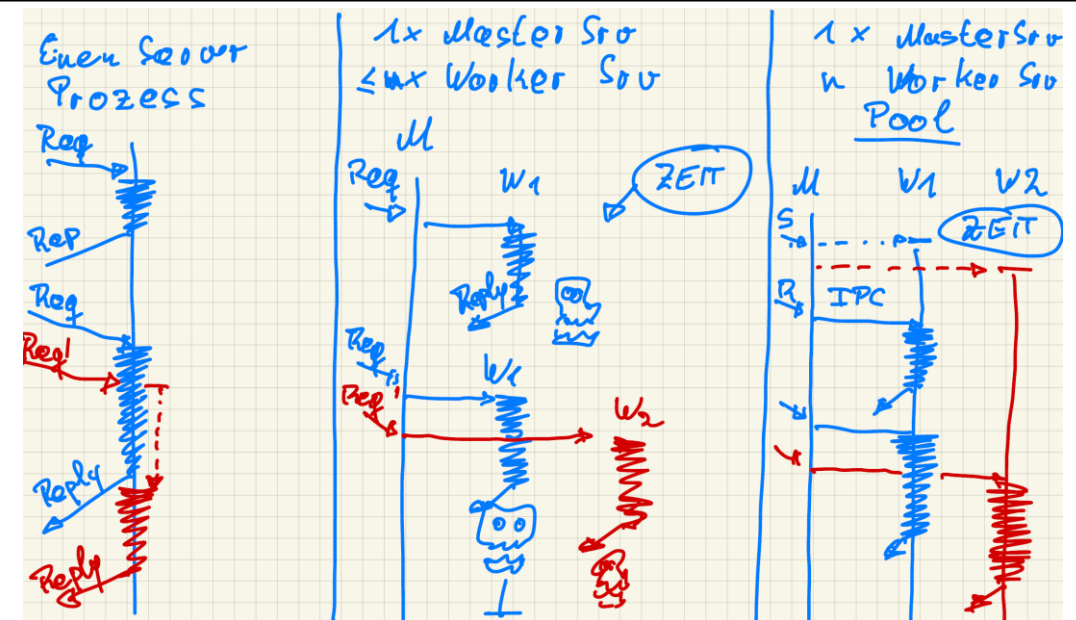
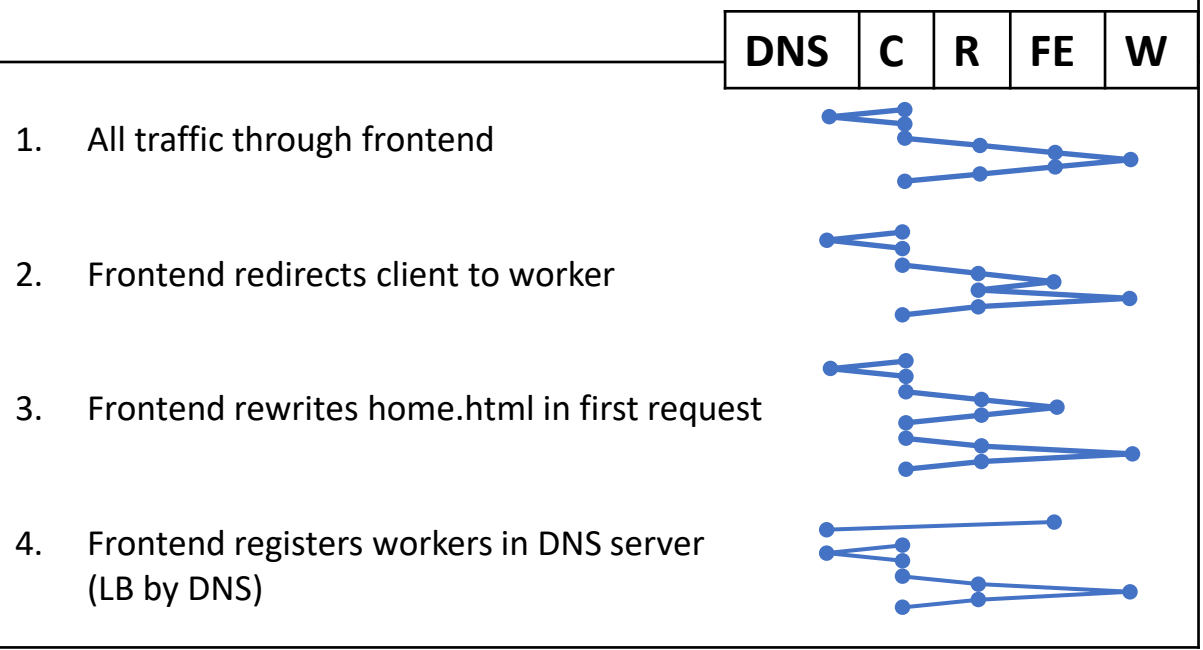


Worker pool (optional)

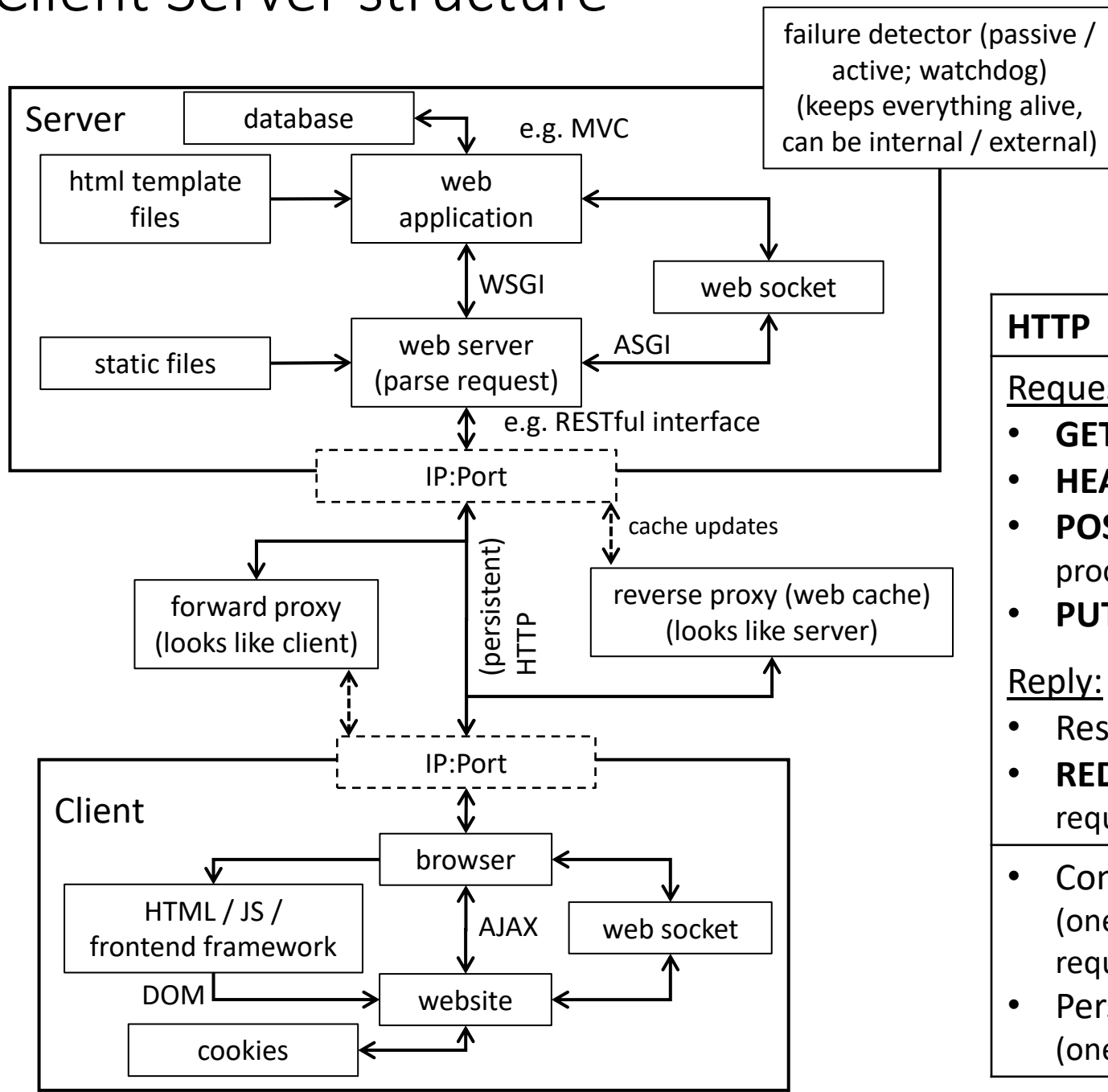
Worker 1, 2, ..., m

- Spawn / kill workers vs. worker pool

How does client frontend worker communication work?



Client Server structure



CRUD

Create
Read
Update
Delete

Uniform Resource Locator (URL)	GET	PUT	PATCH	POST	DELETE
Collection , such as <code>https://api.example.com/resources/</code>	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Not generally used	Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	Delete the entire collection.
Element , such as <code>https://api.example.com/resources/item17</code>	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection, or if it does not exist, create it.	Update the addressed member of the collection.	Not generally used. Treat the addressed member as a collection in its own right and create a new entry within it.	Delete the addressed member of the collection.

HTTP

Request:

- **GET** (obtain content)
- **HEAD** (obtain metadata)
- **POST** (provide URL to program to process data)
- **PUT** (provide URL for Data storage)

Reply:

- Response containing data
- **REDIRECT** (returns new URL to requested data)

- Conventional (one TCP connection for each request / reply)
- Persistent (one TCP connection for each client)

REST

Request:

- **GET**
- **PUT**
- **(PATCH)**
- **POST**
- **DELETE**

(RESTful: Service that implements REST API)

Websocket

- Handshake
- TCP connection between server and client

RPC / RMI

Pass parameters

- **Call by value** (value is copied)
- **Call by reference** (only locally or with log address format)
- **Call by copy & restore** (remote alternative to call by reference, slightly different semantics)

Long address format

- IP : process : logical_address
- Call by reference becomes valid
- Overhead: Everything becomes remote call

Reply cache

- Sequence numbers to decide if response from reply cache
- Only recompute if $f(x)=f(f(x))$

Webservices

- Used in the past (**deprecated!**)
- Interprocess communication
- SOAP (Simple Object Access Protocol)
- Interface is machine readable

Remote calls

- RPC binding
- RMI registry

Applications

Directory service

WebServices & Service description

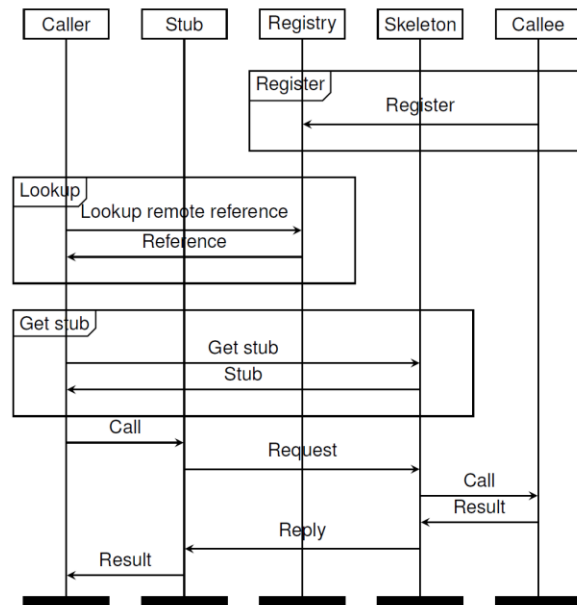
SOAP (representation syntax)

URIs, XML, HTTP

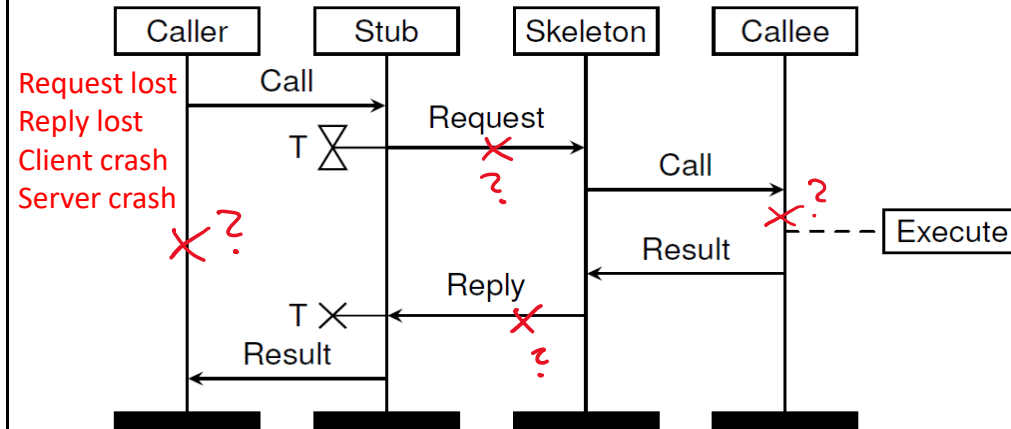
Interface definition

- IDL (language independent)
 - Stub / skeleton (proxies) (can be generated from IDL)
- Transparency
 - Caller & callee see each other
 - Stub / skeleton are transparent
- Marshalling
 - Prepare data for transmission / delivery
 - Done by stub / skeleton

msc RPC – no errors

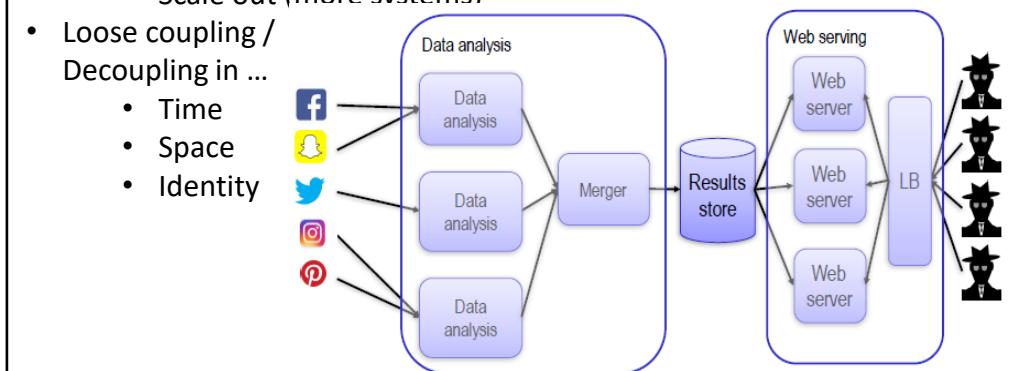


msc RPC – no errors



Microservices

- Stateful / stateless
- Chaining
 - Orchestration (parallel)
 - Choreography (sequential)
- Scalability
 - Scale up (better system)
 - Scale out (more systems)
- Loose coupling / Decoupling in ...



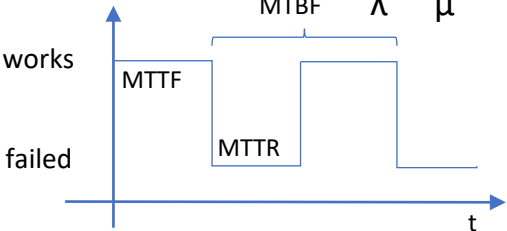
Client Server

Dependability

- Reliability [probability] (MTTF [h])
 $r(t) = P(X > t)$

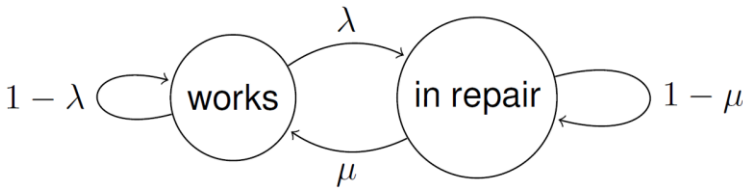
- Availability: $P(\text{works})$
(requirement: system is reparable)

$$P(\text{works}) = \frac{\mu}{\mu + \lambda} = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda} + \frac{1}{\mu}} = \frac{MTTF}{MTTF + MTTR}$$



Steady-state availability [probability]

- λ : Failure probability
- μ : Repair probability

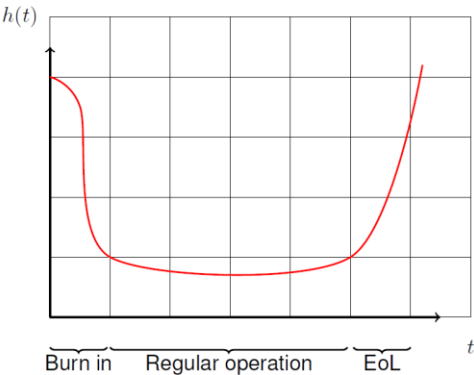


Failure detector (passive / active)

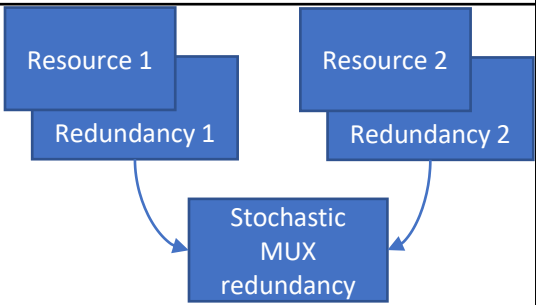
- States
 - Trust
 - Suspect
 - Permanently suspect
- Can differ from actual system state

Hazard rate

$$h(t) = \lim_{\delta t \rightarrow 0} \left(\frac{P(X < t + \delta t | X > t)}{\delta t} \right) \xrightarrow{\text{exp. Vert}} \lambda$$



Stochastic Multiplexing



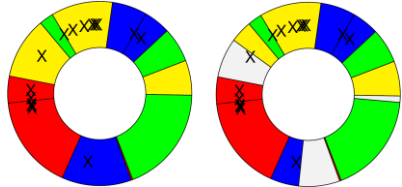
Standby types

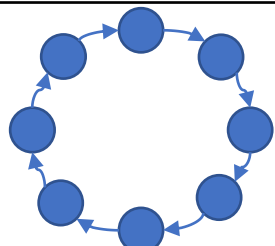
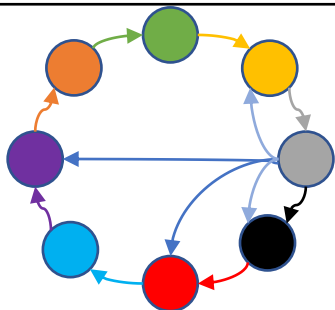
- **Cold** (needs boot)
- **Warm** (is booted)
- **Hot** (same state)
- **Active / Active** (same state + running)

Tier structure

Presentation tier	Logic tier	Data tier
<ul style="list-style-type: none">• User interface• Web: Browser, JavaScript	<ul style="list-style-type: none">• Logical decisions• Command processing• Updating states• Web: Web framework	<ul style="list-style-type: none">• Ground truth for all state• Web: Database

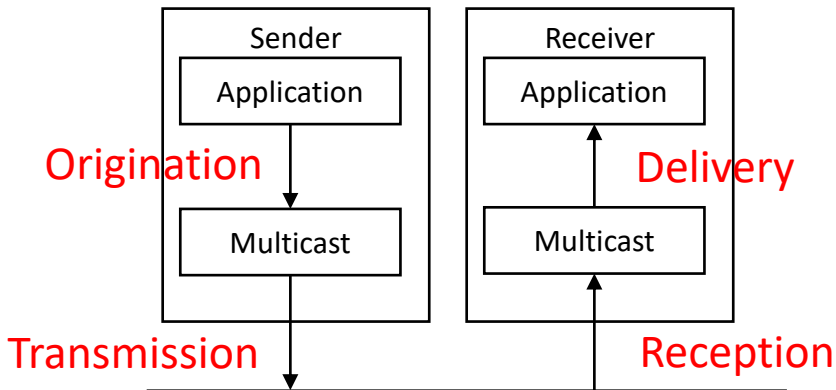
P2P - Simple & Consistent hashing

Hashing (GUIDs and node names come from the same namespace)	
<div>Simple<ul style="list-style-type: none">$selected_node = hash(key) \% num_nodes$Responsibility: Own GUID \rightarrow Next GUIDLarge overhead when <code>num_nodes</code> changesWhen to use: <code>num_nodes</code> does not change</div>	<div>Consistent<ul style="list-style-type: none">Responsibility: Own GUIDs \rightarrow Next GUIDs (common: 100 – 200 GUIDs per node)New node:<ul style="list-style-type: none">Randomly choose GUIDsAsk network who is responsibleTake over responsibilitiesClient needs to store entire server name list locally</div> <div></div>

Distributed hash tables (client does not need server list)																																														
<div>Strawman<ul style="list-style-type: none">• Check if responsible• If not, forward to neighbor</div> <div></div>	<div><p>Own GUID = 65a3; inc. GUID = 64b2</p><table><tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>...</th></tr><tr><td>0</td><td>0...</td><td>1...</td><td>2...</td><td>3...</td><td>4...</td><td>5...</td><td>-</td><td>.</td></tr><tr><td>1</td><td>60...</td><td>61...</td><td>61...</td><td>63...</td><td>64...</td><td>-</td><td>66...</td><td>.</td></tr><tr><td>2</td><td>650...</td><td>651...</td><td>652...</td><td>653...</td><td>...</td><td></td><td></td><td>.</td></tr><tr><td>3</td><td>65A0..</td><td>-</td><td>65A2...</td><td>65A3...</td><td></td><td></td><td></td><td>.</td></tr></table></div> <div>Plaxton<ul style="list-style-type: none">• Comparing GUIDs digit by digit in hexadecimal representation• Rows represent length of GUID• Compared row-wise until first deviation</div>		0	1	2	3	4	5	6	...	0	0...	1...	2...	3...	4...	5...	-	.	1	60...	61...	61...	63...	64...	-	66...	.	2	650...	651...	652...	653...	3	65A0..	-	65A2...	65A3...				.
	0	1	2	3	4	5	6	...																																						
0	0...	1...	2...	3...	4...	5...	-	.																																						
1	60...	61...	61...	63...	64...	-	66...	.																																						
2	650...	651...	652...	653...																																						
3	65A0..	-	65A2...	65A3...				.																																						
<div>Chord<ul style="list-style-type: none">• Check if responsible• Check if direct neighbors are responsible• If not, forward according to fingers• <i>Fingers</i> = $n + 2^{i-1}$ (flip GUID bits)• Each Node is responsible for GUID space in front of its own GUID</div> <div></div>	<div>Pastry (uses Plaxton)<ul style="list-style-type: none">• Joining protocol• Host integration based on underlying network• Departure: copy data to another node• Failure: Redundantly store data on neighbors</div>																																													

Group communication

Multicast protocol



Group membership (views)

Definition

Sequence of group members considered alive

Operations

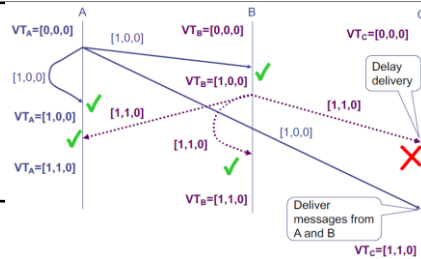
- **Create** group
- **Join** group ($V = V_{old} \cup \{self\}$)
- **Leave** group ($V = V_{old} \setminus \{self\}$)
- **Get** current view
- **Suspect** failed member

Goal

- Define delivery semantics
- Divide between past and future (epochs)

Timing models

Timestamps	Logical time	Lamport time	Vector clock
<ul style="list-style-type: none">• Do not express causality• Can be imperfect	<ul style="list-style-type: none">• Assign every event to totally ordered set T• Causality $L(\text{send}(m)) < L(\text{receive}(m))$• Compatible to <i>happened before</i> relation $x \rightarrow y$ implies $L(x) < L(y)$• Partial order between two events $x \rightarrow_1 y$ y immediately before x	<ul style="list-style-type: none">• Implements logical timestamps• Each process has clock (initialized with 0)• Increment clock for every normal event• Increment clock on send event; attach new clock value to message• Update clock with max(clock, received clock) on receive event; then increment	<p>Functionality</p> <ul style="list-style-type: none">• Implements logical timestamps from every process• Vector of logical timestamps of each process• Own cell is handled analog to lamport time• Other cells are also adapted on receive <p>Can ensure causal ordering!</p> <ul style="list-style-type: none">• Only deliver message if vector timestamp differs in one cell at most• Otherwise, some causally related event may be missing• E.g., CBCAST, causal ordering



Ordering semantics

reliable	FIFO	Atomic	Causal	Highest Name	Description
Yes	No	No	No	reliable	Message is delivered eventually at every receiver
Yes	Yes	No	No	FIFO	Messages originating at one sender are delivered in order at every receiver
Yes	No	Yes	No	Atomic	Messages are delivered in the same order at every receiver
Yes	Yes	Yes	No	Total	FIFO + Atomic
Yes	Yes	No	Yes	Causal	Potentially related messages are delivered in the correct order at every receiver

Pub/Sub Systeme

Events	Matching (Where: all publishers, all subscribers, neutral node)	
Topic based (text metadata)	Subscription & Publication are lists of topics → Match if intersection	
Subject based (key/value metadata)	P = { (a, 5), (b, 7) } S = { (a, 5), (b, [2, 10]) } → match S = { (a, 4), (b, [2, 10]) } → no match Alternative: Match against predicate containing constraints (SQL like) Predicate = [a = "UPB" AND b < 10]	
Content based (look at content)	<ul style="list-style-type: none">• Regex matching• Any other mapping function bool m = match(P, S)	API <ul style="list-style-type: none">• Subscribe (to a set of events)• Publish (events)• Notify (when event matching subscription occurs)
Other (geographics, rate limits)	bool m = match(P, S)	

Loose coupling MQ vs. Pub/Sub	
Decoupling in time Msg is stored even if sender and receiver are inactive	Decoupling in time Sub, pub, notify happen at different points in time (no memory → notify only if already subscribed)
Decoupling in space	
Decoupling in identity	

Central matching server
<ul style="list-style-type: none">• Decoupling in space, time, identity• SPoF
(Content based) event routing
<ul style="list-style-type: none">• Event routing structure• To which neighbor an event is forwarded<ul style="list-style-type: none">• Flooding (send Sub / Notify to all; other only to one)• Routing + Forwarding• Covering of predicates p which select messages m: p1 covers p2 if p2(m) → p1(m) for all m• Alternative: Gossiping<ul style="list-style-type: none">• No routing table• Random forwarding (message eventually reaches destination)• Omq – Filter at subs• Redis -

Message queuing

Architecture

- Sender / Receiver (P2P)
- (Distributed) Queue manager / broker

Leader election algorithm

Assumptions: basic, fault, time

- Simple algorithm
Circle (slide 100)
 - Needs synchronisation
- FloodMax (SpanningTree)
- Raft: see animation + FSA

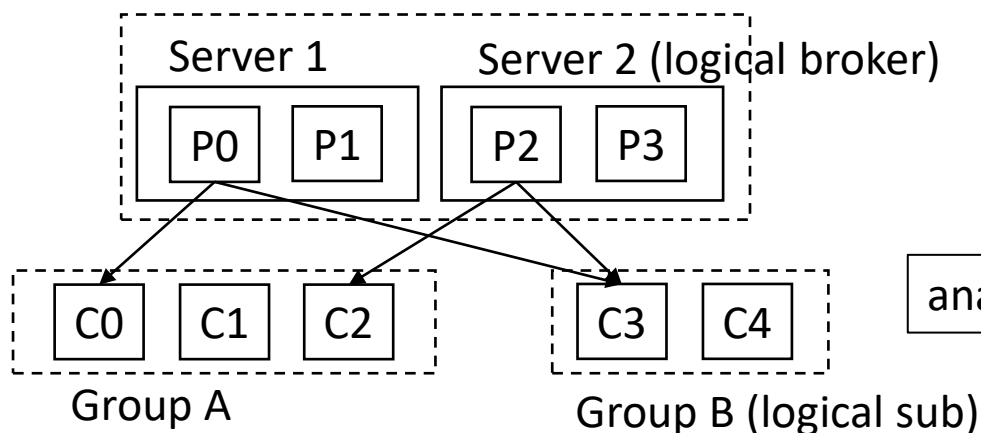
Semantics MQ System

- APPEND
- GET
- POLL
- NOTIFY

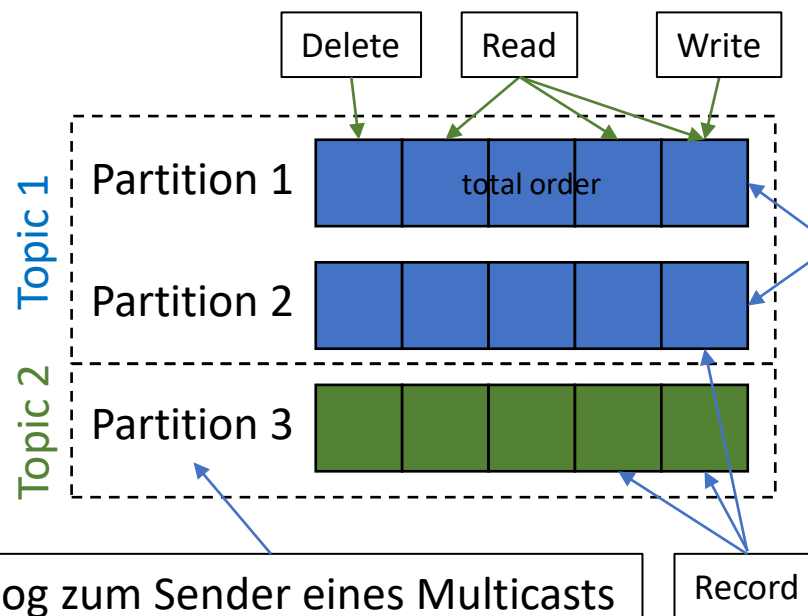
Kafka

- Streams of messages
- Kafka cluster
 - Streams of records (content + metadata)
 - sorted by topic

Kafka Cluster



Broadcast: One group per consumer
Load balancing: Multiple consumer per group

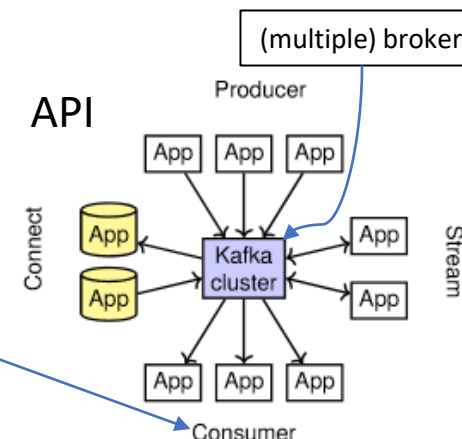


Partition replication

- One leader, multiple followers
- Committed msg: Leader + all working followers have msg in queue
- Load balancing
 - Random, Round-robin

Consumer (position in partition is stored):

- Read msg → update log (at least once)
- Update log → Read msg (at most once)



Distributed storage – Data Centric Consistency Models I

RMI ordering

- **Object knows** it is replicated (ordering inside RMI skeleton)
- **Object does not know** (ordering between Network and Skeleton; common approach)

Quorum (alternative approach to replica management)

- Before: write everywhere
- Now: write only somewhere and ask for latest version
- Consequence: Move some write overhead to read operation
- Example: Read / write ratio
- Higher overlap: Failure tolerance

$$N_R + N_W > N$$
$$N_W > N/2$$

Valid quorum

A	B	C	D
E	F	G	H
I	J	K	L

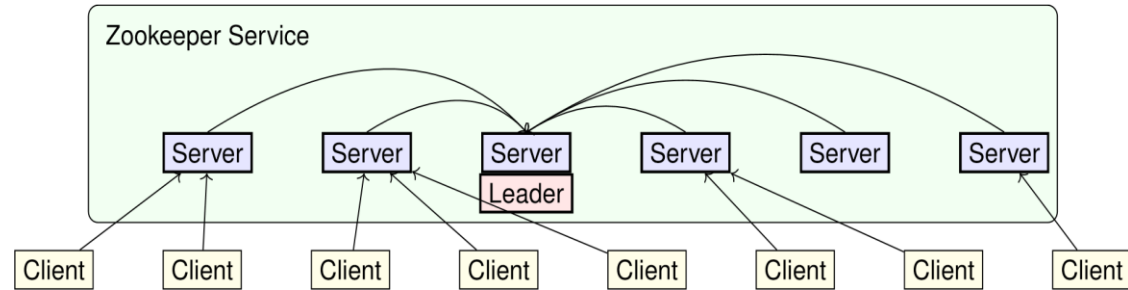
Consistency models (What does up to data mean?) for replicated storage (strong, weak, really weak)



Strict	Sequential	Causal	FIFO	Weak	Release
				<ul style="list-style-type: none">• Synchronization variables vs. data variable• Access synchronization variable only if ALL other write operations at all replicas are done (Sequential consistency for synchronization variables)• Writes to sync variables are grouped together• Consistency only at certain points in time• Ensure synchronicity via “generations” (see view-synchronous->epochs)	<ul style="list-style-type: none">• Weak consistency considers start AND end of synchronization section• Release consistency:<ul style="list-style-type: none">• Enter synchronization area: Get local data up to data• Leave synchronization area: Get remote data up to data

Distributed storage – Data Centric Consistency Models II

Zookeeper



- Distributed coordination service
- High read / write ratio
- Leader and follower servers (leader election)
- Guarantees
 - **Sequentially consistent**
 - **Atomic** (update all or no replica)
 - **Single System Image**
 - **Dependable** (updates are persistent, if enough servers stay alive)
 - **Timely** (consistent within time bounds)
- Operations
 - **Read**: Directly from one server
 - **Write**: Distributed via ZAB (Zookeeper atomic broadcast)

Protocols

- **Primary based protocol**
 - no replication
 - **Local-write protocol**
 - without backup
 - with backup
 - **Remote-write protocol**
 - backup blocking write
 - backup non-blocking write
- **Replicated-write protocol**
 - Naïve
 - **Active replication protocol**
 - only data
 - Replicated objects with **coordinator** which does RMI
- **Quorum based protocol**
 - see above

Update propagation

- **Invalidation protocol**
 - send notification of update (invalidates replica)
- **Update protocol**
 - push based
 - pull based
 - hybrid (leasing)

Distributed transactions (Sequence of read / write operations)

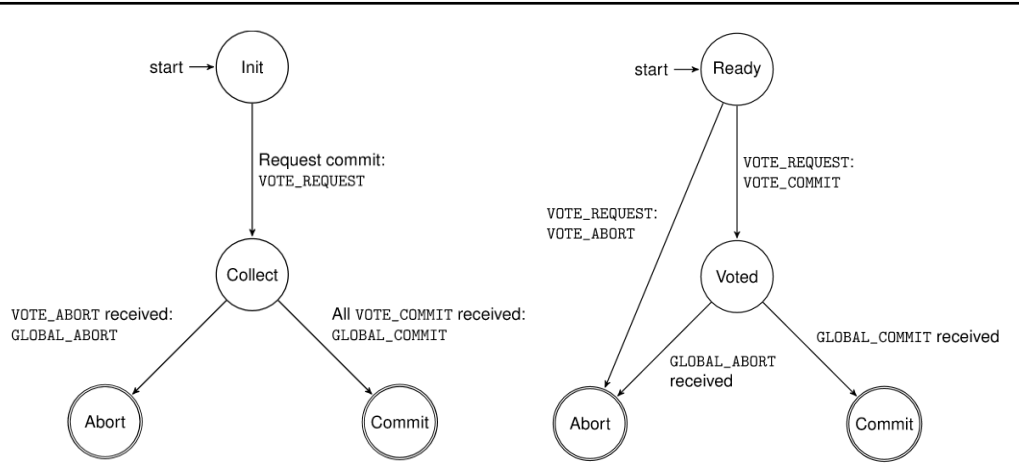
- Distributed snapshot (e.g. for deadlock detection)

Properties of databases when executing transactions (**ACID**)

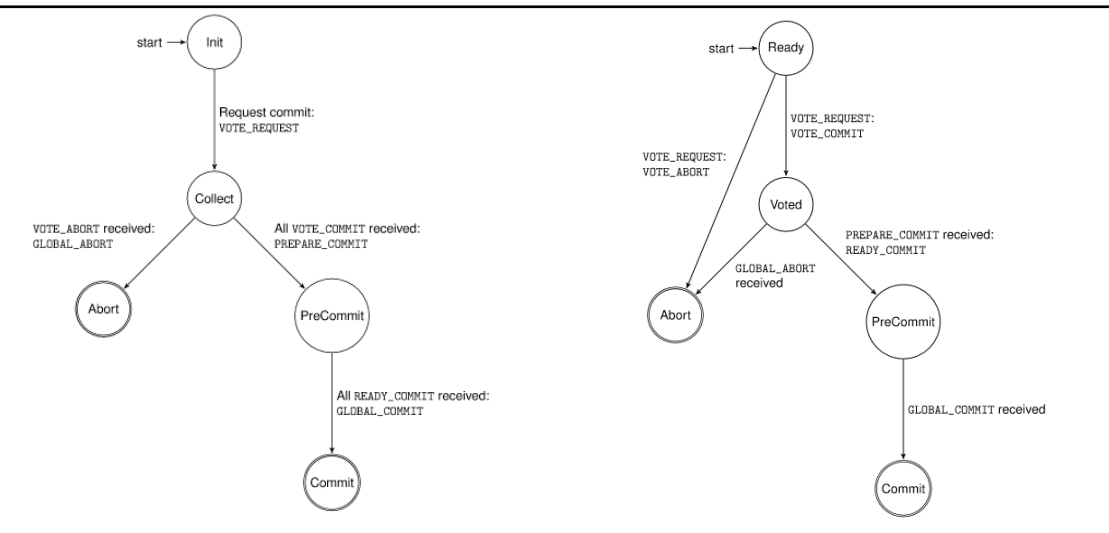
Atomicity (all or nothing)	Consistency	Isolation	Durability	Tradeoff: Concurrency
<ul style="list-style-type: none">• When transaction (t) completes -> every change visible in data store.• When t aborts -> no effect at all	<ul style="list-style-type: none">• ...	<ul style="list-style-type: none">• Transactions operate without effects from concurrently executing transactions being visible to them• T1 reads X multiple times while X is modified by T2	<ul style="list-style-type: none">• After successful t its transactions must be visible to other t.• Save effects in (stable) storage• Recoverability	<ul style="list-style-type: none">• Concurrent processing of ts.• Limited by atomicity, isolation• Goal: ensure serializability

Leader is dead; all live participants are in state VOTED

2PC – Two phase commit (safe, but not always live)



3PC – Three phase commit (better but still not live and save)



Distributed snapshot

- Cut in time
- Goals
 - Detect de
 - Garbage
- How it works?
 - Save own
 - Save even
 - Send mar

Consensus

Consensus
<p>Agreed up decision</p> <ul style="list-style-type: none">Impossible if asynchrony and failure (FLP: Fischer-Lynch-Paterson result)Properties<ul style="list-style-type: none">Agreement: All processes decide for the same value.Termination: All processes eventually decide.Validity:<ul style="list-style-type: none">If all processes start with 0 all must decide for 0.If all processes start with 1 and all messages are delivered all must decide for 1.Scenarios<ul style="list-style-type: none">No failures, messages arrive in bounded time Deterministic decision rulef nodes fail, messages arrive in bounded time FloodSet algorithm: f+1 rounds, deterministic decision rulef nodes are traitors, $n \geq 3f + 1$ mit n number of nodes und f number of traitors

Request lost
Reply lost

Byzantine agreement
<ul style="list-style-type: none">Agreement, Termination, Validity apply only to nonfaulty processesTriple Modular Redundancy (3f) is not enoughCreate consensus when there are f traitor<ul style="list-style-type: none">Feasible case: $n \geq 3f + 1$Synchronous case: $n > 4f$ nodes, in $2(f + 1)$ rounds

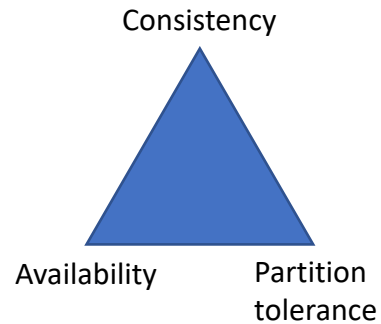
Raft (see http://thesecretlivesofdata.com/raft/)	
<p>Goals</p> <ul style="list-style-type: none">Safety (never return incorrect result)Available (if majority is operable, progress is made)Timing-independent (specific timer values does not matt)	
Leader election	Log replication
<ul style="list-style-type: none">Every nodes starts as followerRandom timeoutTimeout expires → Follower becomes candidateAsk for votes → Candidate becomes leader if majority of votesLeader sends heartbeats which reset follower timeouts	<ul style="list-style-type: none">Heartbeats contain log updatesAcks acknowledge heartbeat and log changeLog entries are committed once majority of followers acknowledge it

Message ordering
Unordered Ordered

Distributed databases

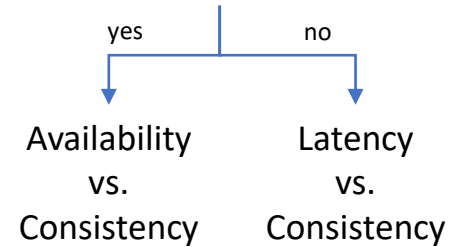
CAP theorem

- 2 of 3 are achievable



PACELEC theorem

Partition?



NoSQL

- Not only SQL

BASE

- Basically Available, Soft state, Eventual consistency
- Give up consistency

Eventual consistency

- System eventually becomes consistent
- In absence of updates

Big data

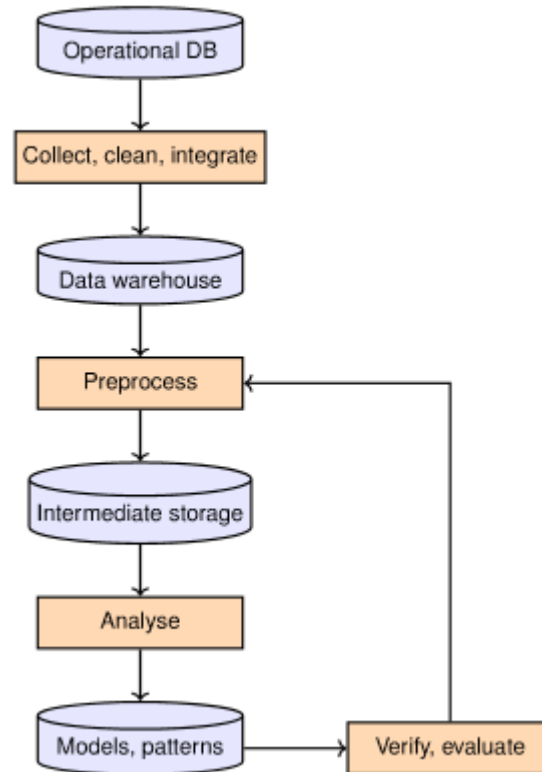
3Vs / 5Vs of data

- Volume
- Variety
- Velocity
- Versatility
- Value

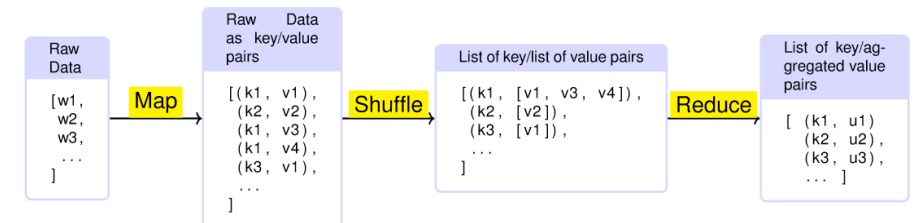
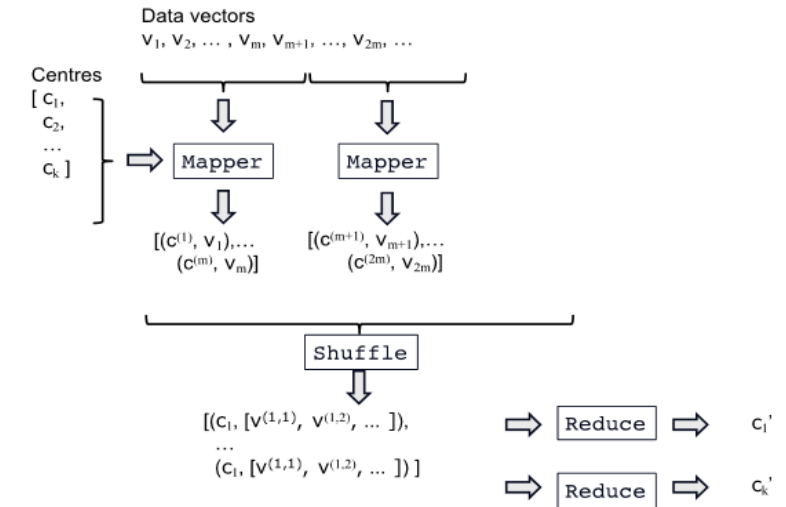
Clustering

- kMeans

Data analytics



MapReduce



Unübersicht

RPC / RMI

- Parameter passing
- Long address format
- Reply cache $f(x)=f(f(x))$
- Bindings
- Interface (Stub/Skeleton)
 - Transparency
 - Marshalling
 - IDL

Microservices

Webservices

Client / Server

- Structure
- Send / receive
- Wire format
- Timing models
- Delivery models
- Fault models / detection

P2P

- Simple hashing
- Consistent hashing
- DHT
 - Strawman
 - Chord
 - Plaxton

Group communication

- Multicast protocol
- Timing models
 - Timestamps
 - Logical time (Lamport time)
 - Vector clocks
- Ordering semantics
- Views (Group membership)

Message queuing

- Exactly once?
- Broker
- Overlay graph
- Kafka...

Pub / Sub

- Event routing
- API
- Matching
 - Topic
 - Subscription
 - Configuration

Websockets

Ordering semantics

- Reliable, FIFO, Atomic, Total

Leader election

Gossiping

Loose coupling

- Time
- Space
- Identity

Distributed storage

Distributed databases

Distributed transactions

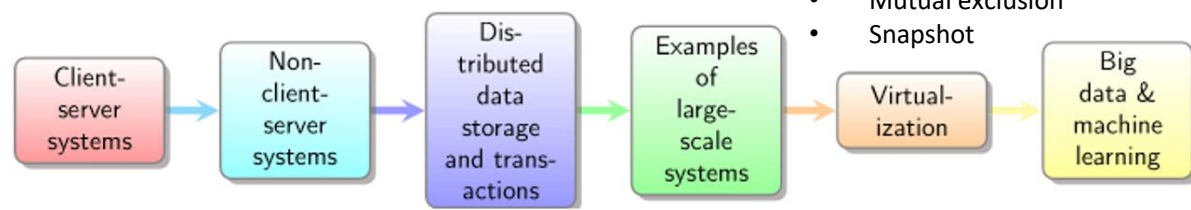
- 2PC / 3PC
- Mutual exclusion
- Snapshot

Consensus

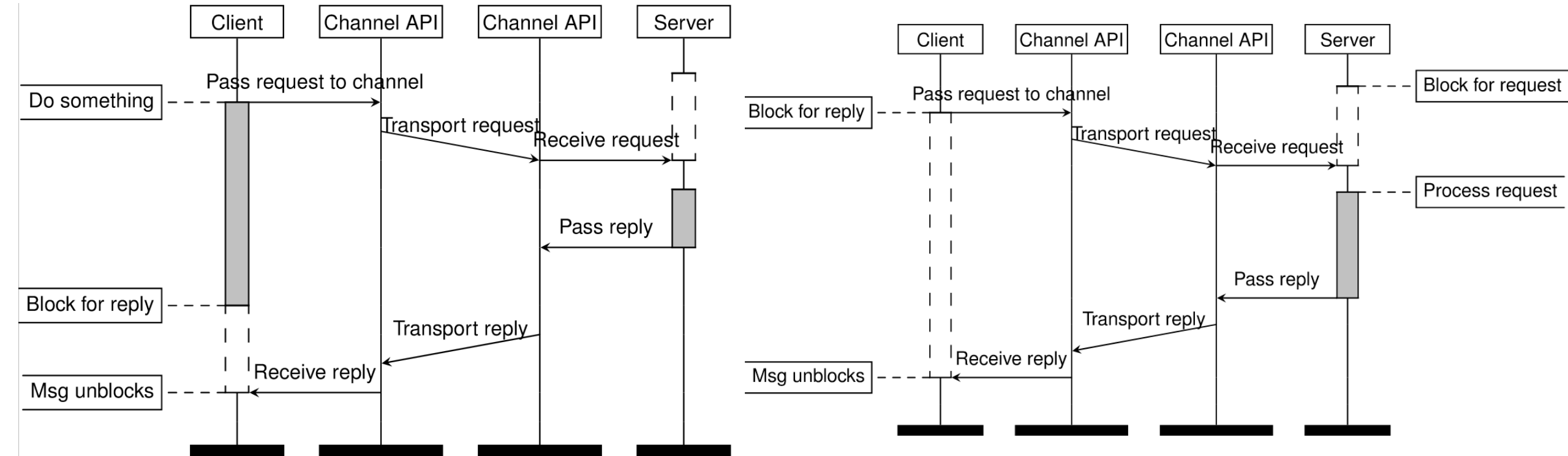
- Properties
 - Agreement
 - Termination
 - Validity
- Raft
 - Leader election
 - Log replication
- Byzantine agreement

Big Data

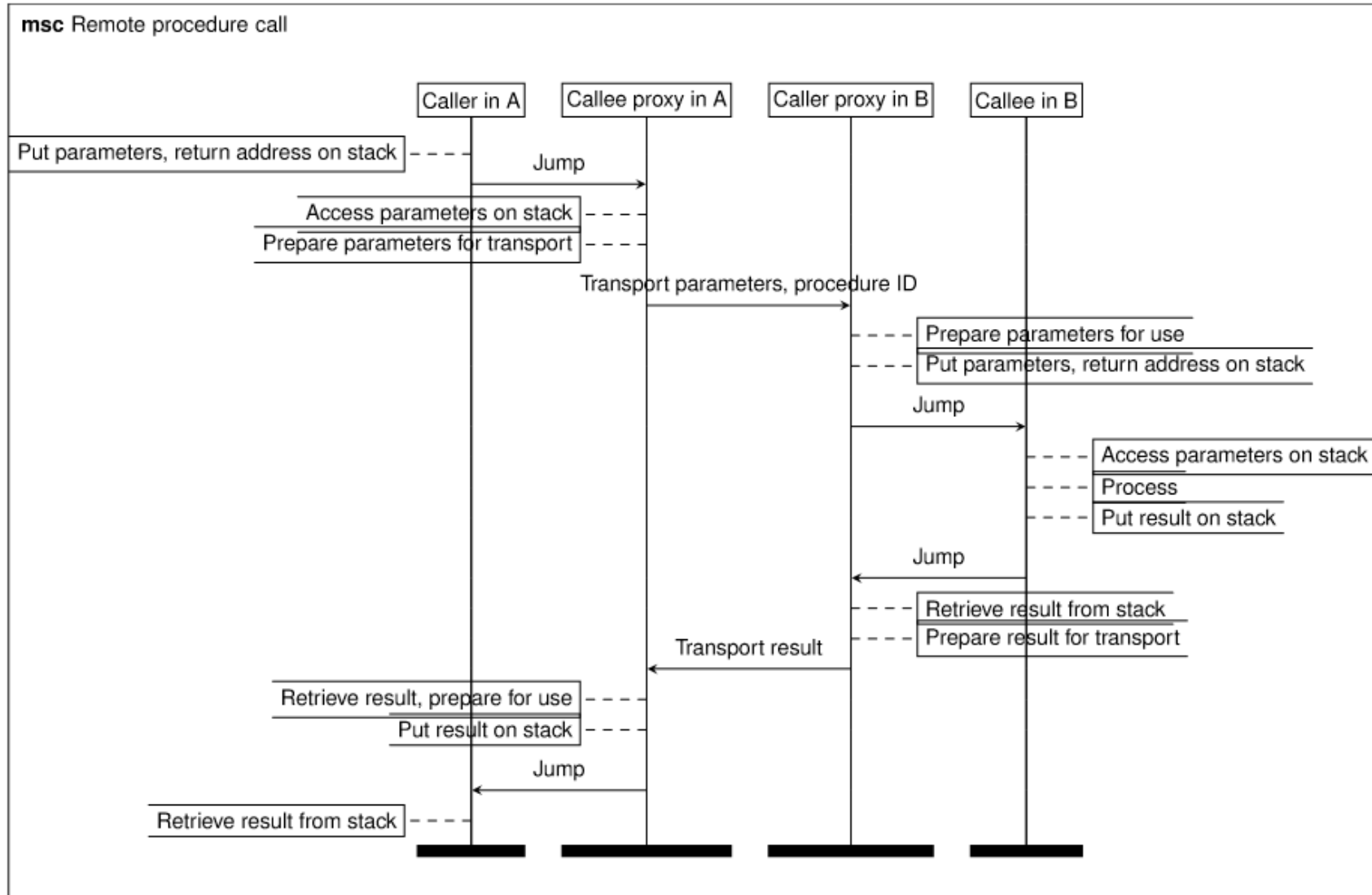
- 3/5Vs
- Data analytics model
- MapReduce (Programming model and algorithm)



Request/Reply: Synchronous vs. Asynchronous



RPC by proxy



Structure of this class

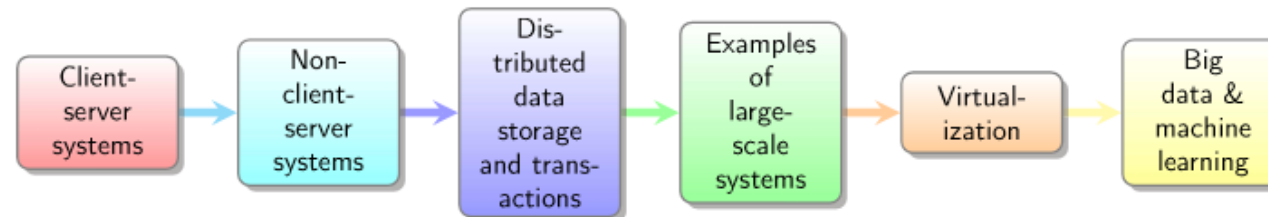


Figure 2: Structure of this class

What can go wrong? Taxonomy

- ▶ **Fault**: a defect in the **system under consideration**
 - ▶ May or may not lead to observable misbehavior
 - ▶ E.g.: an alpha particle flips a bit in a memory cell
- ▶ **Error**: Discrepancy between intended and actual behavior of system
 - ▶ At runtime, errors are the manifestation of a fault in an unexpected state
 - ▶ E.g.: memory cell was written with a 1, subsequent read returns a 0 owing to the bit flit fault
 - ▶ Do not necessarily cause failure
- ▶ **Failure**: System displays behavior contrary to specification
 - ▶ Caused by error
 - ▶ Observable from outside of the system
 - ▶ E.g., incorrect memory value might cause observable misbehavior, or it might be correct (e.g., redundancy)

Improving
Client/Server
systems:
Latency,
throughput,
dependability,
consistency

Holger Karl

Overall
requirements

Dependability

Fault Models

**Determining
metrics**

**Redundancy –
Standby
Failure detection**

Multi-tier
architectures

Improving
throughput

Improving
latency

Summary

Material