panikzettel.htwr-aachen.de

Advanced Internet Technology

Version 2 | 23.07.2026

Jonas Schneider

Contents

1 Preface and Prior Knowledge

This Panikzettel ("panic note") is open source at https://github.com/htwr-aachen/panikzettel . We encourage any and all contribution from students and official sources. See https://htwr-aachen.de/docs/panikzettel .

As this is a communication systems subject, knowledge of IP, TCP, and UDP is assumed. Take a look over the Datkom Panikzettel to refresh. We will structure this Panikzettel along the three major topics: Massively Scalable Systems, Resource-Constrained Systems & Adaptive Communication.

Sidenote: Effort of this Panikzettel

This Panikzettel was fun to create, but was also an immense effort, have fun reading it :)
PS. Most graphs here are generated on the actual parameters of the graphs

2 Massively Scalable Systems

2.1 Peer-to-Peer Systems

Imagine you want to build a system, which operates mainly between peers, such as file sharing, a chat, or something like that. These systems face challenges with scalability and are susceptible to a single-point of failure in a server-based architecture which does not seem necessary when we just communicate between another. Thus, we strive towards a **decentralized self-organizing system** with **decentralized usage of resources**. The lecture gives Definition 1 the long definition, of which parts are important (marked bold).

Definition 1: P2P System

P2P is a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the **edge of the Internet**. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes **must operate outside the DNS system** and have significant or total **autonomy** from central servers.

The essential challenge to solve in P2P systems is the location and search of content. "How do we distribute it and find it later?" With considering our requirements of **scalability**, **robustness** and **consistency**.

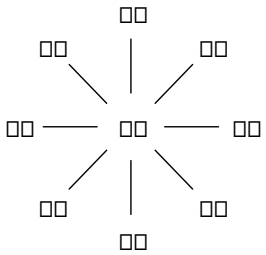


Figure 1:
Client-Server

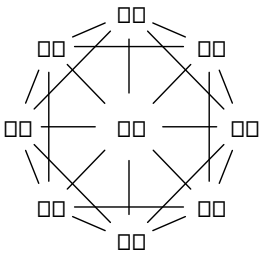


Figure 2:
Centralized P2P

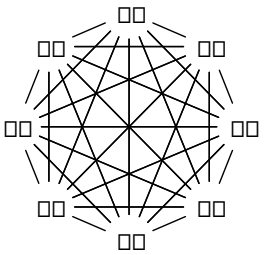


Figure 3:
Pure P2P

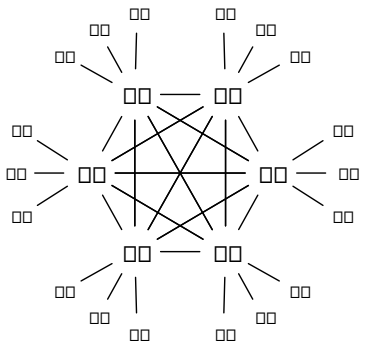


Figure 4:
Hybrid P2P

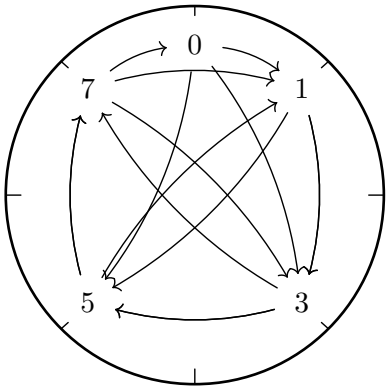


Figure 5:
Structured P2P

2.2 Centralized P2P & Napster

The first ideas of a peer-to-peer systems where realized as a centralized type shown in Figure 2. The most prominent example is the Napster file sharing application. Here we have a central *directory service* which provides coordination of entities. It would for example save that Alice has file `A`. If then Bob wants to retrieve `A`, it would ask the directory which peer has this file, which would give a list of just `[Alice]`. Now if Alice is offline currently, the directory service should organize some sort of replication to combat the total unavailability of the data. Though the directory service is in itself not much better, than the traditional Client-Server model Figure 1 shows.

When we look at scalability the worst counts, worst of in this P2P model, is the directory service. Most peers do not need information about other peers, but the central directory service need $O(n)$ state entries for all peers. This makes this **not scalable**. One advantage however is that, if the directory service answers our search, we can just directory contact Alice without extra communication, thus $O(1)$ communication overhead.

2.3 Pure P2P & Gnutella 0.4

When looking at the complete opposite end shown in Figure 3 or Gnutella 0.4, we see that there is no central server anymore. Here Alice still stores her own file, but when Bob want to retrieve it, he has to ask his neighbors "Requesting `A`". Their neighbors do not know Alice either and send this request to their neighbors. This is called *flooding* and it will quickly overload **any** system. To mitigate a complete overload we limit the Request to a specific time to live (default 7), after which it will be just dropped. This means we do not necessarily find `A` if Alice is further than 7 hops away. Notice however that **each** entity (all are equal) stores constant $O(1)$ information about their peers. But we require up to a $O(n)$ communication overhead before we find Alice (in case an of straight line) which does **not scale** either.

2.4 Hybrid P2P & Gnutella 0.6

Gnutella 0.6 wants to improve the performance and introduced a superpeer type of peer. This are dynamically elected by peers which then just connect to one or more superpeers. When Bob wants to find `A`, he goes to its superpeer. This then does flooding between all superpeers to find it. This improves the method but does not scale either, because there is not organization

and load can be very asymmetric. Asymptotic could still be considered $O(1)$ state and $O(n)$ communication overhead because the superpeer network is again a pure P2P.

2.5 Structured P2P & Chord

Now to the winner a *distributed hash table*. We recognize that we need to achieve a nice distribution of our content to the nodes and use the help of a hash function (h), which does not necessarily be a cryptographically secure hash function, but it **must** map to exactly one $\{0, ..., 2^m - 1\}$ large address space. Then we lay out this address space as a ring shown and give each peer an ID in this address space e.g. $h(\text{Alice}) = 13, h(\text{Bob}) = 7, h(\text{Charlie}) = 3, h(\text{David}) = 15$. The content is then also hashed $h(\texttt{A}) = 0$ and saved by Alice on the next clockwise node on the ring of that id $0 \to 3, 4 \to 7, ...$ e.g. Charlie. This results in Figure 6.

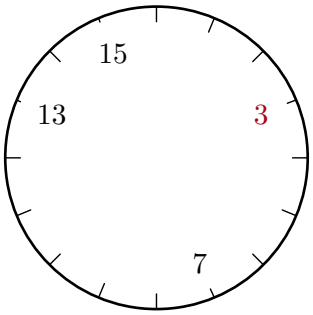


Figure 6: DHT 1. Step

Again Bob wants to find A. Currently he only knows it's hash $h(\texttt{A})$, but for routing we need to know the next nodes. Specifically we create a table of the nodes which own the $j + 2^i$, where j is my ID and $0 \leq i \leq m$. So with each finger we cover increasing distances, which allows us to have a $O(\log n)$ node state.

Finger	Node ID
0	13
1	13
2	13
3	15

Table 1: Finger Table of peer 7 (Bob)

Coming back to Bob, who wants to find the owning node of $h(\texttt{A} = 0$ and starts with going as far as he can without overshooting so $7 \to 13$ in our case (! he does not know 1 is responsible 0, there could be someone in the middle!), Then we check again, 13 knows 15 is closer so $13 \to 15$, and now 15 knows the next node 3 is responsible and sends this back to Bob. This allows routing in $O(\log n)$ steps :D see Figure 7. Then Bob can simply go to the responsible node and retrieve A (Figure 8)

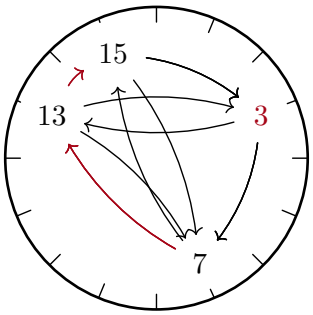


Figure 7: Chord Routing

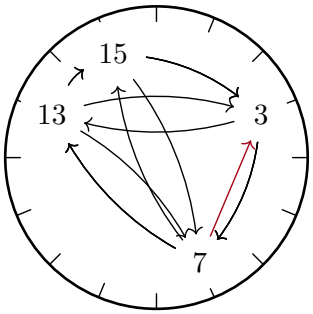


Figure 8: Chord Routing

Algorithmus 0: Chord Routing

Go as far as we can in our finger table, without overshooting, and repeat this until the next node is responsible for the searched content.

There are a couple of nuances that need to be set out.

1. Node Arrival

If you as a new node arrives, get you ID, iterate over the finger table and query the $\mathrm{id} + 2^i$'s responsible node similar to standard id lookup. For successors of your node contact the first successor from the finger table and use his list.

Then we need to signal nodes which have an invalid finger to our successor to update to us. For this we use the same method but backwards $\mathrm{id} - 2^i$. $=> O(\log^2 n)$

2. Storage & Replication

We can distinguish between *direct storage* where we store content directly on the responsible node, or an *indirect storage* where we store at Charlie, that Alice has the content. Content can be replicated with using multiple hashes, and storing it at each (e.g $h(\mathtt{A}), h(h(\mathtt{A})), h(h(h(\mathtt{A})))$. Then when loading choose one to lookup.

3. Node Failure

We do active finger maintenance and periodically check fingers for activity. If a finger is offline, query the alternative like on node arrival. We also need to keep a list of successors of our node, to mitigate the last step, where a responsible node is unavailable.

4. Node Departure

Is handled either exactly like a failure or with a short notification period, where we notify nodes about our pending departure.

2.6 Graph & Network models

This is the theoretical network science groundwork why the DHT and Chord was designed the way it is. A graph $G = (V, E)$ contains vertices and edges

We will look at three different graph categories (Random Graphs , Small World Graphs , Power Law Graphs) and rate them for each $v \in V$ mainly using $\deg(v)$ i.e. the number of edges which are incident to vertex v, the clustering coefficient.

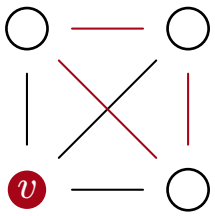
Definition 2: Clustering Coefficient

$$C(v) := \frac{2 \cdot e(v)}{(\deg(v) \cdot (\deg(v) - 1)}$$

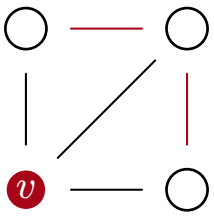
where $e(v) :=$ the number of connection of v's neighbors with each other. The clustering coefficient of a graph is the average over all vertices:

$$C(G) := \frac{\sum_{\{v \in V\}} C(V)}{|V|}$$

Examples:



$$C(v) = \frac{2 \cdot e(v)}{\deg(v) \cdot (\deg(v) - 1)} = \frac{2 \cdot 3}{3 \cdot 2} = 1$$



$$C(v) = \frac{2 \cdot e(v)}{\deg(v) \cdot (\deg(v) - 1)} = \frac{2 \cdot 2}{3 \cdot 2} = \frac{2}{3}$$

Definition 3: Path Length

A path is a sequence of $k \in \mathbb{N}$ vertices $P(v, w) = (v_1, v_2, ..., v_k) \in V \times V \times ... \times V$ where $v_1 = v$ and $v_k = w$ such that $(v_i, v_{i+1}) \in E$ for $0 \leq i \leq k - 1$.

- The path length $|P(v, w)| = k - 1$ is the number of edges in the path.
- The distance $d(v, w)$ is the shortest-path $P(v, w)$ with respect to the path length.

Definition 4: Graph Diameter

The diameter of a graph $G = (V, E)$ is the farthest distance $d(v, w)$ between any two $v, w \in V$

Before going further what are we looking for? We want a small diameter for efficient routing steps but high clustering coefficient, to achieve reliability and stability. We do also need a somewhat balanced node degree.

Random Graphs

We have the Randomized Network (Erdös-Rényi-Model) in which we choose a graph $g_{n,m} \in_R G_{n,m}$ out of all Graphs with exactly $n \in \mathbb{N}$ vertices and $m \in \mathbb{N}$ edges.

Satz 1: Connectedness

With high probability, the graph has a connected component of size $O(n)$ if the average node degree $> O(\log n)$.

Satz 2: Diameter

If the graph is connected, then with high probability $\text{Diameter}(g_{n,m}) = O(\log n)$.

We cannot reasonably generate/choose such a random network, and have to generate it incrementally using the *Gilbert Random Graph* Model. For this generate a graph $g_{n,p}$ with a given number of vertices $n \in \mathbb{N}$ and a probability $0 \leq p \leq 1$ to add an edge between any (v, w) with $v, w \in V$.

Satz 3: Clustering Coefficient

$C(g_{n,p}) \simeq p$ with high probability.

Satz 4: Diameter

If $p > \ln \frac{n}{n}$ a graph $g_{n,p}$ will be connected with high probability.

Power Law / Scale-free Graphs

Random graph do not provide the necessary goals we want to achieve. They have a good (good **always** means $O(\log n)$ here) diameter but do not provide a dense locale structure.

When we look at network in reality, they often provide a so called *power law* distributed node degree.

Definition 5: Power-Law Distributed Graph

In a power-law / scale-free distributed graph the probability that a node is connected to k is $P(k) \sim k^{-\gamma}$. This directly translates to the statements **"The rich get richer"** as some nodes (called hubs) have very high and most other have small degree. It is called *scale-free* due to the probability being completely independent of the scale of the system.

We see that Internet web pages, Internet backbone structure, cooperation between actors, power grids and more follow a power-law distribution.

Algorithmus 0: *Baranbási-Albert* Model

1. Start with a small random network $G = (V, E)$
2. Add a single vertex $G' = (V \cup \{v_{\text{new}}\}, E)$. Choose $m \in \mathbb{N}$ vertices from the probability distribution $\Pi_v = \frac{\deg(v)}{\sum_{w \in V'} \deg(w)}$.

To recap, the more edges it already has, the more probable new edges are.

Small World Graphs

Power-Law networks are great and provide a somewhat dense local structure (even though only the hubs) and a good routing performance. The problem are with the hubs. They have much higher traffic load in a P2P system and are susceptible to targeted attacks. They are however robust against random node failures.

The next graph model are small-world networks, where a dense local structure (e.g. a grid) is used in combination with a few far reaching connections. These connections that make the routing fast.

1. Watts-Strogatz Model

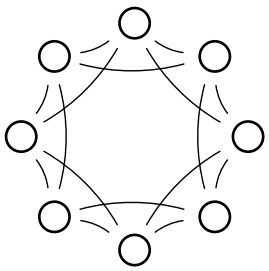


Figure 11:
Watts-Strogatz $k = 2, p = 0$

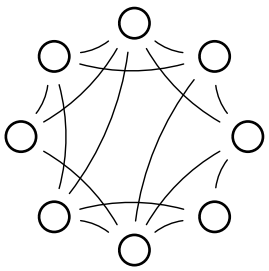


Figure 12:
Watts-Strogatz $k = 2, p = \frac{1}{8}$

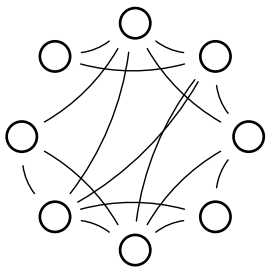


Figure 13:
Watts-Strogatz $k = 2, p = \frac{1}{4}$

Form a ring and connect the next $k \in \mathbb{N}$ vertices together. Then choose random numbers $0 \leq x \leq 1$ and rewire each edge if $x \leq p$ to a uniformly random $w \in_R V$.

2. Kleinbergs Model is similar

Take a fix grid of vertices and **add** (not rewire) a long-distance edge to any $w \in V$ with $P(v, w) \sim d_{m(v,w)}^{-}\alpha$ where d_m is the Manhattan distance (so only along the grid).

Satz 5: Short Paths in Kleinbergs model

The routing algorithm will find 'short' paths if and only if $\alpha = d$, where d is the grid dimension

2.7 Internet Indirection Infrastructure I3

A P2P system for indirection in the Internet. This allows for mobility, multicast, anycast and more, which all use some sort of indirection. The system is implemented as an overlay network i.e. on top of IP in between TCP/UDP.

For this we again use a DHT/Chord addressing and routing system. Each node has its i3 id and places a *trigger* of this id together with a way route to the node (e.g. IP address or DNS entry) in the DHT. As seen in Figure 14 the sender then can send you content, with just knowing your id and nothing else. If the sender and receiver are not part of the i3 network, the content is first send to any i3 node (as the gateway), this then looks up the responsible node for id ID and forwards the traffic. We also use caching to alleviate the extra overhead of ID lookup both for the gateway, and the responsible node.

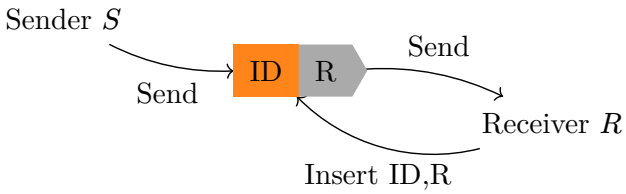


Figure 14: Simple i3 trigger

Let's go over all the applications of this infrastructure:

1. Mobility

Each trigger can have multiple ends, when sending to a trigger each end receives the full content. We can also update the existing ends of a trigger to a new. As such when we move e.g. from Wi-Fi to LTE, we register our new IP address as R_2 end end remove R_1.

To improve this design further, we can have a short time, where both R_1 and R_2 are part of the trigger and form a multicast group. The mobile node now has to detect duplicate packets coming over both routes for a short time, after which we remove R_1 to fully transition.

2. Multicast

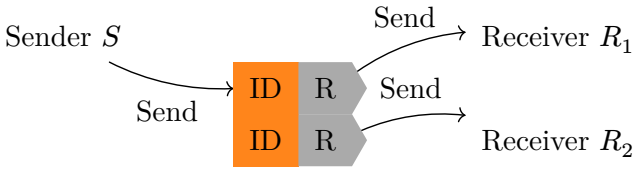


Figure 15: Multicast i3 trigger

We solved multicast already as seen in Figure 15. We can improve scalability concerns, due to a single node having to send to all R by implementing another Indirection and let ID send to ID2 which is again multicast

3. Anycast

As a refresher: In anycast, only one of group of nodes should receive our message. Most often the closest one.

In i3 we can realize this using a joint anycast prefix, and a per member postfix. Then the data is forwarded using a longest prefix match, this also enables a more filtered approach and load-balancing and context-aware selction.

4. Network Services

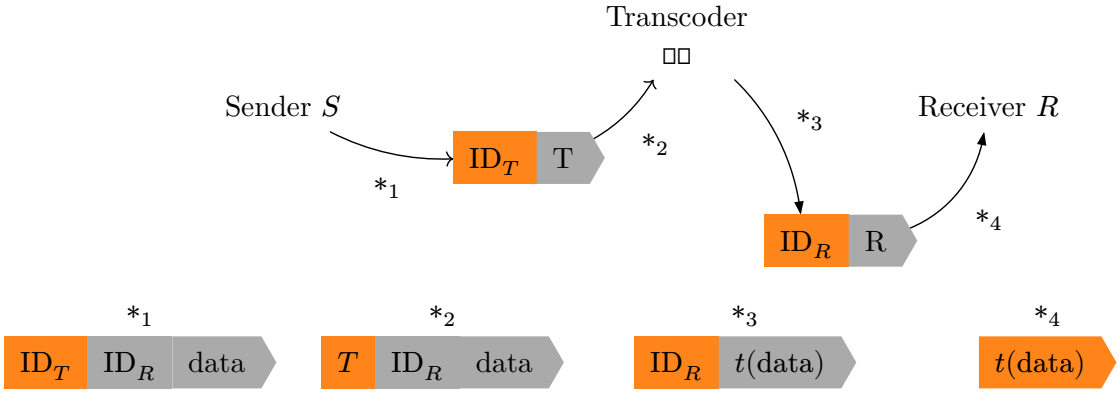


Figure 16: Multicast i3 Sender-initiated Network Services

We can define “middleware” services which can manipulate the data in-transit either as a sender-initiated or receiver initiated.

For this we define the exact types a trigger can be:

i3 destination address	i3 stack	i3 trigger
ID / (IP & Port)	IP/Port | ID	ID | IP/Port | ID_2
This is the end address, upon receive, the address is removed from the stack, data is processed and send to next node	A sequence of i3 addresses	Upon receive, the ID will be replaced by the stack e.g. (IP/Port + ID_2) and forwarded to the address

Server-initiated as shown in Figure 16, Sends the data after a stack of IDs (in this case the transcoder).

In a Receiver-initiated the receiver would place the ID_T into his trigger Which would redirect the traffic first to the transcoder and then to R. ID_R | ID_T | R

2.8 Bitcoin

Sidenote: Warning

Read this section carefully, to shut down any crypto bro who says it revolutionizes the world

A special case of distributed P2P systems are crypto-currencies because even with all that we have a particular problem which is very difficult to solve

Definition 6: Double Spending Problem

It must be impossible to:
1. Create new currency outside rules
2. Copy currency after obtaining it
3. Reuse currency

Image Eve has 5 Melaniacoins The problem is when Eve send Alice and Bob both illegaly 5 Melaniacoins over different links. This illegal transaction can come through if the network is slow between these links, then Alice and Bob both assume they have 5 Melaniacoins.

Bitcoin was the first (kinda) cryptocurrency build on a **public append-only distributed ledger** called *blockchain*. Transactions are only considered valid if they are in the longest valid blockchain.

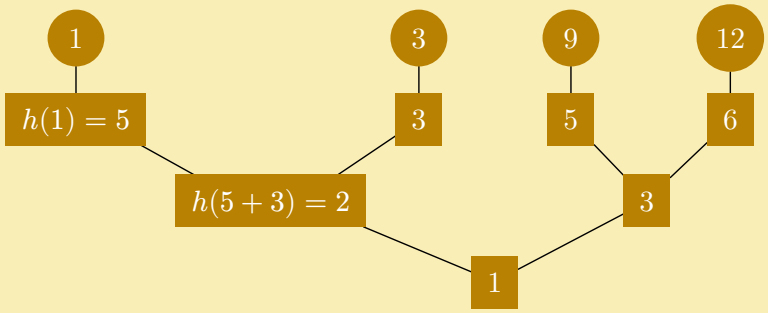
Blockchain

Each block in the blockchain have the following properties:
- ID: hash of this block
- Previous Block Hash: used to link blocks together
- Proof-of-Work: a random nonce to assure ID has k zeros as a prefix of the hash, based on the current threshold $k \in \mathbb{N}$
- Transactions: set of transactions in each block.

- Transaction Merkle Tree allows nodes, that trust parts of a transaction, to verify partial transactions without needing the full content (only the partial merkle node suffices for the other transactions)

Algorithmus 0: Merkle Tree

Say you have a very secure hash function $h(x) = (3x + 2) \bmod 8$. Then construct the merkle tree like this and only save the 1.



We require the hash function to be a cryptographically secure on (see IT-Sec panikzettel)

There are a couple node types:
1. Full nodes verify all blocks and transactions. They must be bootstrapped with the known root blocks hash and an archival node but afterward do not need the full blockchain.
2. Archival nodes save the complete blockchain
3. Backbone nodes are both archival and verifying
4. Miners find new proof-of-works and therefore add new blocks. They pick proposed transactions into new blocks

Step by Step Transaction

| this can become somewhat tricky and is not necessary to understand to pass the exam (pass not 1.0 it). We are also more detailed than the lecture here because it aids understanding

Bitcoins are managed by a wallet software managing a specific *bitcoin address* (or multiple), which is just a (base58-checksumed typo mitigating representation) of a hash a public key, to which the user has the private key.

The ownership of the bitcoins at that address is enforced with signatures of that private key.

Imagine Alice wants to send Bob 10฿.

1. **Creating the transaction**: Alice's wallet software constructs a proposed *transaction*. This transaction is cryptographically signed with her private key and includes
   - **Inputs**: The transaction's inputs reference previous transactions where Alice received bitcoins. These bitcoins are locked in the previous transactions with output scripts, that Alice now needs to unlock to prove she owns them.
   - **Outputs**: Then after unlocking, she is able to create new locks that specify the conditions for spending the bitcoins in the future. In our case, the bitcoins will be lock with Bob's address.

(Do not take the time assertions, e.g. "after" literally here, both inputs, and outputs are both scripts in the same transaction which is proposed as a single unit)

2. **Unlocking / Inputs** / *scriptSig*: Let's assume Alice needs to gather enough to pay Bob. Her wallet finds two previous transactions, in one Alice received 5฿ (ID `f3`) and 10฿ in another (ID `02`). Also assume these were both locked at that time using a standard `P2PKH` output scripts. To unlock them, her wallet creates a corresponding input script (*scriptSig*) for each

   ```
   Push <signature for f3>
   Push <Alices's pubkey>
   ```

   and

   ```
   Push <signature for 02>
   Push <Alics's pubkey>
   ```

   When this new transaction is processed by the network, the provided signature and pubkey will be checked against the conditions of the previous output script to prove Alice's ownership. Now Alice has successfully proven her ownership and unlocked them, ready to be locked again by the output scripts.

3. **Locking / Ouptuts** / *scriptPubKey*: The bitcoins are now in an unlocked state, ready to be locked again by output scripts:
   - Script to send Bob 10฿
   - Transaction fees, say 1฿ for the hard work of the miner (as the time of this writing it would be 0.00000224฿)
   - Script to return the rest 4฿ back to Alice.

   The transaction fee is handled implicitly, simply all not locked bitcoins are the transaction fee at the end.

   The others have to be declared though, for this a couple of output script options are in use

   1. Pay-to-Public-Key-Hash (`P2PKH`) is the standard nowadays which locks the Unspent Transaction Output (UTXO) behind the has of bob's public key.

   ```
   OP_DUB              // Duplicates the public key provided by bob in the intput
   script
   OP_HASH160          // Hashes that duplicate key
   <pubkeyhash of bob>
   OP_EQUALVERIFY      // Checks if the provided hash matches Bob's public key hash
   OP_CHECKSIG         // Checks if the signature provided by bib is valid for this
   transaction
   ```

   2. Pay-to-Public-Key (`P2PK`) is not used anymore as a post-quantum mitigation, but locks directly behind Bob's public key.

   ```
     <pubkey of bob>
     OP_CHECKSIG // Checks if the signature provided by bib is valid for this
   transaction
   ```

   3. Pay-to-Multi-Sig (`P2MS`) is used when more than one signature unlocks the funds.
   4. Pay-to-Script-Hash (`P2SH`) is a more advanced script type that locks the bitcoins not to a public key hash, but to the hash of another script (the redeem script). When the bitcoins are spent, the spender must provide the original script and the inputs that make that script true. This allows for much more complex transaction conditions without cluttering the blockchain with long scripts.
   5. **OP_RETURN** allows for currently 80 bytes of arbitrary data for timestamping and other things (see)

`P2PKH` is used over `P2PK` since, with throwaway bitcoin addresses the public key of the address of bob is only known after using the funds, where they are already send to another unknown address.

Again to recap *scriptSig* (input script) from a new transaction provides the key to unlock a previous *scriptPubKey* (output) script.

4. Alice sends this complete transaction to the bitcoin networks.
5. Miners pick up this transaction and combine it with multiple other into one a proposed block and via brute force find a proof-of-work such that the block hash starts with $n \in \mathbb{N}$ zeros.
6. After a block has been added, wait a few more blocks to be sure there is no fork in the and establish consensus that the transaction is done.

Now Bob is literally a millionaire congrats.

2.9 Cloud

The next three sections concern cloud environments and two mainly cloud applications.

> "What is the cloud anyway?"

Definition 7: Cloud Computing

Cloud computing is a model for enabling ubiquitous, convenient, on- demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

In a nutshell informally it is basically 3-4 (Amazon web services, Google cloud, Microsoft azure, Oracle cloud □) big companies having to much compute resources and making a business out of sharing it while providing additional added value.

For our sakes cloud computing has the following characteristics:
1. Resource pooling: computing resources serve multiple consumers
2. Broad network access: accessible via standard protocols
3. On-demand self-service: automatic provisionment of resources
4. Rapid elasticity: resources can be scaled to demand quickly
5. Measured service: automatic control & optimization of resources via metering capability

Most of the time they operate on a "Pay as much as used" basis and abstract away the underlying infrastructure. For example, you create virtual networks the servers communicate over, regardless whether they are in the same server rack, data-center or country.

Services provided by cloud providers can be in different categories, depending on how much the provider "does for you".

Private / On-Premise	Infrastructure as a Service (IaaS)	Platform as a Service (PaaS)	Software as a Service (SaaS)
Applications	Applications	Applications	Applications
Runtimes	Runtimes	Runtimes	Runtimes
Security & Integration	Security & Integration	Security & Integration	Security & Integration
Databases	Databases	Databases	Databases
Operating System	Operating System	Operating System	Operating System
Virtualization	Virtualization	Virtualization	Virtualization
Server HW	Server HW	Server HW	Server HW
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking
Your problem		Cloud providers problem	

Now we do a quick sidetrack exploration into data-center design of such cloud providers.

Three-Tiered Design

The traditional topology with hierarchical classes of switches.

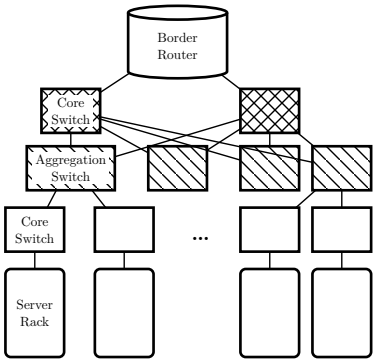


Figure 17: Three-Tiered Datacenter Design

Fat-Tree

Allow easy additions but can result in uneven traffic distribution. Fat-Tree is entirely determined by the port count $K \in \mathbb{N}$ of the switches.

$K = 24$ ports $\rightarrow \frac{K^3}{4} = 3456$ servers

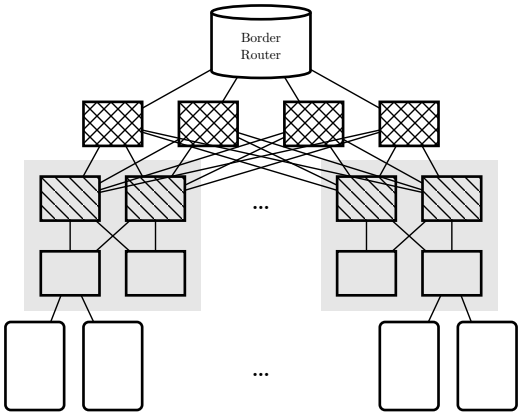


Figure 18: Fat-Tree Datacenter Design

Jellyfish

Random connections between switches can result in more throughput for less switches. There is not really an aggregation or core switch anymore, or any structure.

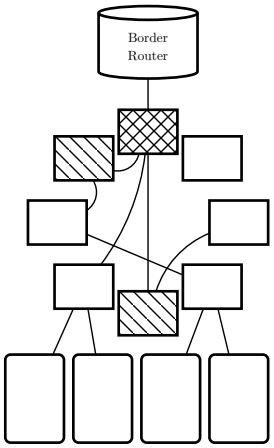


Figure 19: Jellyfish Datacenter Design

Now with the data-center designed we can build application that runs on it. We will see Cassandra as a distributed Database & Map-Reduce for massive distributed computation that can take advantage of the cloud by scaling-out.

Definition 8: Scalability Approaches

1. Scale-Up: Better server to handle more traffic
2. Scale-Out: More servers to handle more traffic

2.10 Cassandra

Satz 6: CAP Theorem (Eric Brewer)

In a distributed system you can only satisfy at most two of the following three properties:
1. Consistency: all nodes have the same data at any time
2. Availability: the system is operational at any time
3. Partition-tolerance: the system continues to operate in spite of network partitions

Regular DBMS such as Postgres usually provide strong consistency but are not available when the network partitions. Cassandra and many such Databases on the other hand only provide a eventual (weak) consistency but high availability i.e. you will not necessarily receive the newest data but always some.

For this, Cassandra again uses a DHT, a so called Cassandra ring and again each instance in the ring is responsible for a part of the address space. However, we store the data at $N-1$ additional nodes to have configurable $N \in \mathbb{N}$ replicas of the data. The biggest difference between this setup and Chord, is that Cassandra is managed by someone and all Cassandra nodes know each other. As such there is no need for the finger routing technique and any onboarding, offboarding logic. Nodes are added and removed manually and they gossip memberships between each other.

Now to receive a data item we can require a certain level of consistency $R \in \mathbb{N}$ and send our request to **any** node in the cluster. Since each nodes knows all others, our requested node will ask the coordinator of the data + some replica nodes **directly**. Now the node waits for R nodes to reply to it and then reply to us the conflict resolved data. Regardless of R the results may be outdated and after replying the requested node will wait for all responses and then initiate a *read repair*.

Writing is very similar, with addition of a *hinted handoff* incase nodes are unavailable, then further nodes are temporarily used.

Cassandra uses a *bloom filter* for a fast key exists lookup **in** each instance. Take $k \in \mathbb{N}$ hash functions $h_0, ..., h_k$ each producing outputs of $b \in \mathbb{N}$ bits and create a bitmap of size $m := 2^b$. For example $h_0(x) := (2x+1) \bmod 12$ and $h_1(x) := (3x+2) \bmod 12$. If we write a new item 9 into the bloom filter we set bits $h_0(9) = 7$ and $h_1(9) = 5$ of the bitmap to 1 and leave the rest as before. If we want to check if item 5 is present we read bits $h_0(5) = 11$ and $h_1(5) = 5$. While bit 5 is set to 1 in our example, bit 11 is not and therefore we can be sure, that 5 is not stored. The probability of a false positives are $\approx \left(1 - e^{\frac{kn}{m}}\right)^k$ with $n \in \mathbb{N}$ being the number of items stored. An optimal k would be $k = \ln 2 \cdot \frac{m}{n}$.

2.11 Map-Reduce

Satz 7: Amdahls formular

The speedup S of a program of which a portion $0 \leq p \leq 1$ can be parallelized by $n \in \mathbb{N}$ cores is bounded by

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

Map-Reduce is a originally google system for massive (20 exabyte/day) computation. The processing is divided into a

1. Map phase: computing an intermediate solution

$$\text{map}(k_{\text{in}}, v_{\text{in}}) \to \text{List}[< k_{\text{out}}, v_{\text{intermediate}} >]$$

2. Reduce phase: collecting and combining solution grouped by k_{out}

$$\text{reduce}(k_{\text{out}}, \text{List}[v_{\text{intermediate}}]) \to \text{List}[v_{\text{out}}]$$

It's easy to see that this provides a great parallelization groundwork since in each stage everything is independent and thus scaleable.

MapReduce chunks the to be processed data into *Mappers* then a *Partitioner* collects the results and partitions reducer tasks/nodes to produce the final output. In high-performance computation ahmdahls law is only theoretically applicable, because there is much overhead on network communication and data distribution. Communication is limited since each Mapper is isolated from the rest of the system and does not need to communicate other than start and completion. Data should be distribution equally to the *Mappers*, but that does not necessarily mean each Mapper task takes the same time. For this MapReduce adopts a *Job Tracker* (Master), *Task Tracker* (Worker) approach with each worker pulling work from the master when finished. For map tasks that take exceptionally long we can implement speculative execution where multiple workers receive the same input and the first wins. The reducers can for example be somewhat equally divided via the usual hash method. An example of MapReduce can seen in Figure 20.

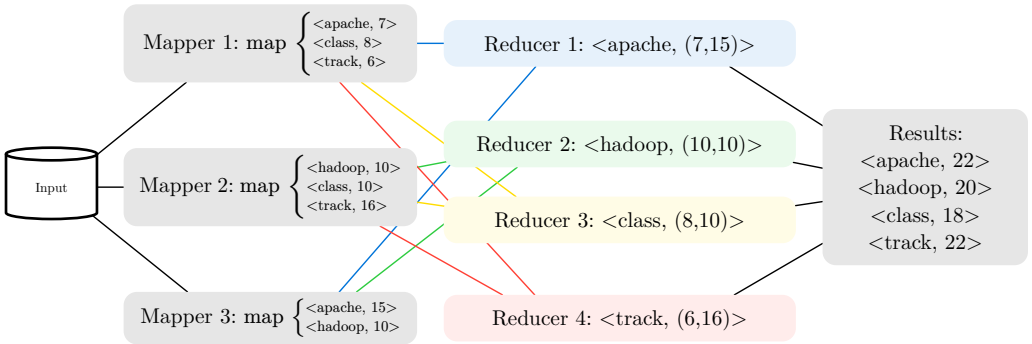


Figure 20: MapReduce Pipeline Example

3 Resource-Constrained Systems & Cyberphysical Systems

Since the invention and spread of the internet there is this idea of interconnected everything. This idea is conceptualized into the Internet of Things (IoT) where devices/microcontrollers in "things" are connected to the network. A cyberphysical system is a digital chip i.e. a microcontroller that has influence in the "real" world either by sensors (allowing measurements) and/or by actuators (motors, mosfets, etc.) that allows control over things.

3.1 Energy Consumption

The chips in these IoT devices vary widely between purposes, price, and other factors. But most of them are severely resource constrained. Either they have little processing power or run on battery power and need to last as long as possible. The lecture example here are "smart" fire detectors which you do not want to charge too often.

The energy consumption depends in most cases on the following things in order

1. Sending wireless transmissions —
2. Receiving wireless transmissions —
3. CPU computation —
4. RAM —
5. Storage — Depends on the type of Storage high speed SSDs can consume up to 10W, HDDs as well, both during full operation. The types of devices we are talking about likeliest only have tiny flash storage inside them.
6. Physical systems — This of course heavily depends on the device. A 1 Hz laser smoke detection costs basically no power. Powering a motor does though.

TODO Unfinished Work

This Panikzettel is only half finished if I find time again I may continue it. Otherwise it is to much work to not upload it anyway.

If you have some time or want to give others a better time learning, continue it and publish it on `https://github.com/htwr-aachen/panikzettel` .

Even sketches, corrections and bullet points help others!

3.2 In-Network Processing

3.3 Multi-Hop Networks

This chapter is covered in the MIT lecture and it's Panikzettel (https://htwr-aachen.de/panikzettel/mit.pdf) and not further summarized here.

3.4 IoT

4 Adaptive Communication