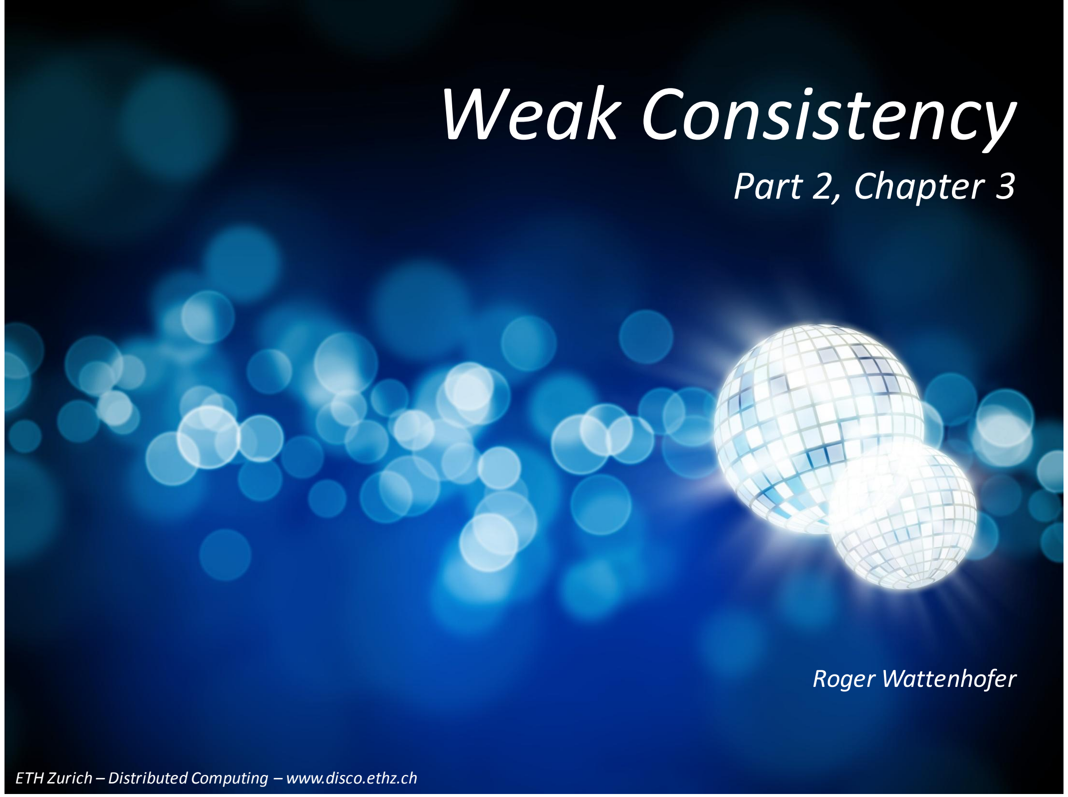


Weak Consistency

Part 2, Chapter 3

Roger Wattenhofer

Overview

- CAP: Consistency or Availability?
- Consistency Models
- Peer-to-Peer
- Distributed Storage
- Bitcoin

How to make sites responsive?

Goals of Replication

- Fault-Tolerance
  - That’s what we have been looking at so far...
  - Databases
  - We want to have a system that looks like a single node, but can tolerate node failures, etc.
  - Consistency is important („better fail the whole system than giving up consistency!“)

- Performance
  - Single server cannot cope with millions of client requests per second
  - Large systems use replication to distribute load
  - Availability is important (that’s a major reason why we have replicated the system...)
  - Can we relax the notion of consistency?

Example: Bookstore

Consider a bookstore that sells its books over the world wide web:

What should the system provide?

- **Consistency**
  For each user the system behaves correctly

- **Availability**
  If a user clicks on a book in order to put it in his shopping cart, the user does not have to wait for the system to respond.

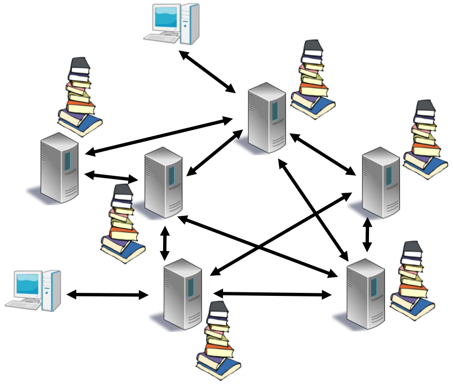
- **Partition Tolerance**
  If the European and the American datacenter lose contact, the system should still operate.

How would **you** do that?

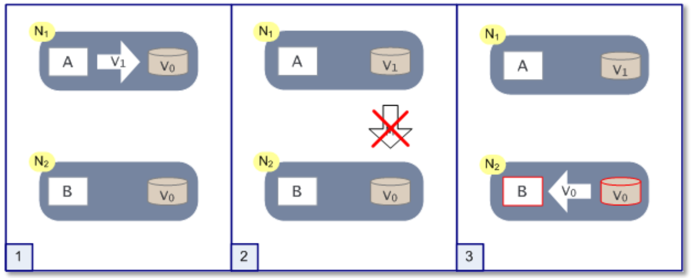
CAP-Theorem

Theorem

It is impossible for a distributed computer system to simultaneously provide Consistency, Availability and Partition Tolerance.
A distributed system can satisfy any two of these guarantees at the same time but not all three.

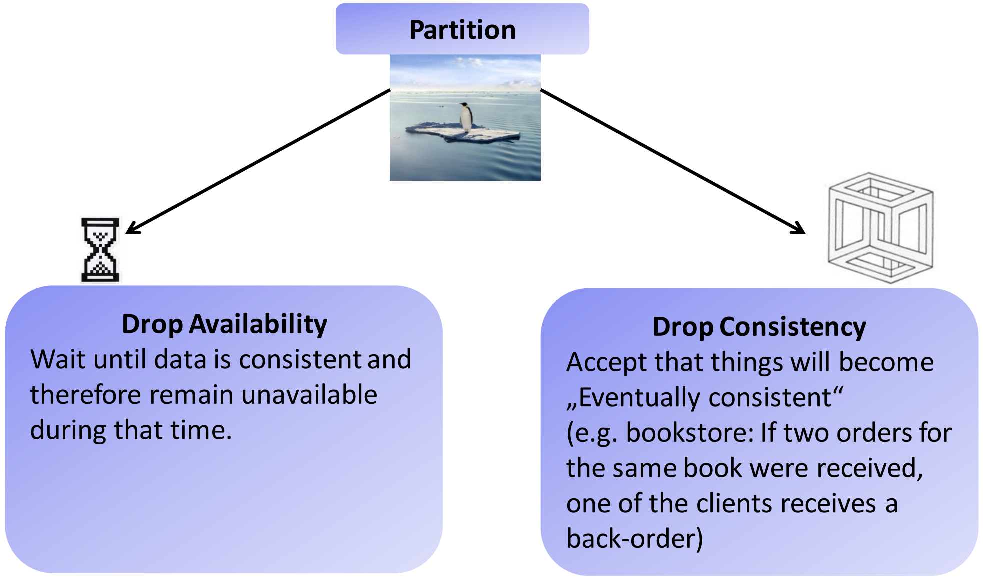


CAP-Theorem: Proof



- N_1 and N_2 are networks which both share a piece of data v.
- Algorithm A writes data to v and algorithm B reads data from v.
- If a partition between N_1 and N_2 occurs, there is no way to ensure consistency and availability: Either A and B have to wait for each other before finishing (so availability is not guaranteed) or inconsistencies will occur.

CAP-Theorem: Consequences



Partition

Drop Availability
Wait until data is consistent and therefore remain unavailable during that time.

Drop Consistency
Accept that things will become „Eventually consistent“
(e.g. bookstore: If two orders for the same book were received, one of the clients receives a back-order)

Again, what would you prefer?

Availability is more important than consistency!

distrib.exe - Application Error

The application failed to initialize properly (0xc0000142). Click on OK to terminate the application.

CAP-Theorem: Criticism

- Application Errors
- Repeatable DBMS errors
- A disaster (local cluster wiped out)

CAP-Theorem does not apply

- Unrepeatable DBMS errors
- Operating system errors
- Hardware failure in local cluster
- A network partition in a local cluster

Mostly cause a single node to fail (can be seen as a degenerated case of a network partition)

This is easily survived by lots of algorithms

- Network failure in the WAN

Very rare!

Conclusion: Better giving up availability than sacrificing consistency

ACID and BASE

ACID

- **Atomicity**: All or Nothing: Either a transaction is processed in its entirety or not at all
- **Consistency**: The database remains in a consistent state
- **Isolation**: Data from transactions that are not yet completed cannot be read by other transactions
- **Durability**: If a transaction was successful it stays in the system (even if system failures occur)

BASE

- **B**asically **A**vailable
- **S**oft State
- **E**ventually consistent

BASE is a counter concept to ACID.

The system may be in an inconsistent state, but will eventually become consistent.

ACID vs. BASE

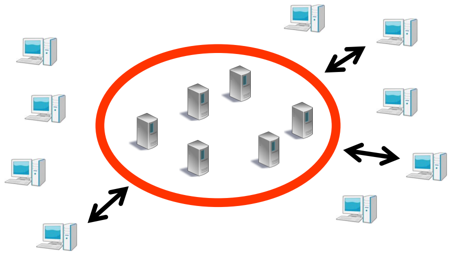
ACID

- Strong consistency
- Pessimistic
- Focus on commit
- Isolation
- Difficult schema evolution

BASE

- Weak consistency
- Optimistic
- Focus on availability
- Best effort
- Flexible schema evolution
- Approximate answers okay
- Faster
- Simpler?

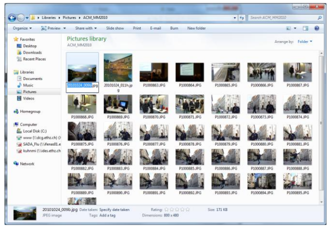
Consistency Models (Client View)



- Interface that describes the system behavior
- Recall: Strong consistency
  - After an update of process A completes, any subsequent access (by A, B, C, etc.) will return the updated value.
- Weak consistency
  - Goal: Guarantee availability and some „reasonable amount“ of consistency!
  - System does not guarantee that subsequent accesses will return the updated value.

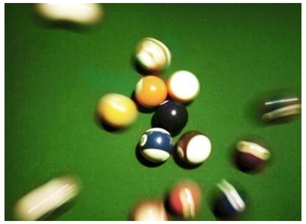
What kind of guarantees would you definitely expect from a real-world storage system?

Examples of Guarantees We Might Not Want to Sacrifice...



- If I write something to the storage, I want to see the result on a subsequent read.

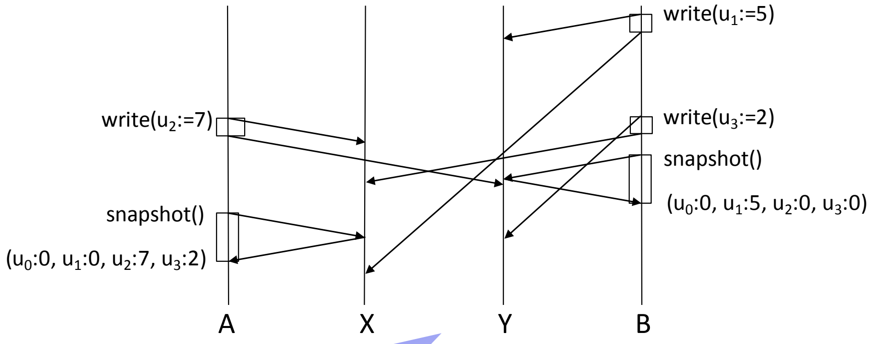
- If I perform two read operations on the same variable, the value returned at the second read should be at least as new as the value returned by the first read.



- Known data-dependencies should be reflected by the values read from the storage system.

Weak Consistency

- A considerable performance gain can result if messages are transmitted independently, and applied to each replica whenever they arrive.
  - But: Clients can see inconsistencies that would never happen with unreplicated data.


write(u_1:=5)
write(u_2:=7)
write(u_3:=2)
snapshot()
(u_0:0, u_1:5, u_2:0, u_3:0)
snapshot()
(u_0:0, u_1:0, u_2:7, u_3:2)
A
X
Y
B

This execution is NOT sequentially consistent

Weak Consistency: Eventual Consistency

Definition

Eventual Consistency

If no new updates are made to the data object, eventually all accesses will return the last updated value.

- Special form of weak consistency
- Allows for „disconnected operation“
- Requires some conflict resolution mechanism
  - After conflict resolution all clients see the same order of operations up to a certain point in time („agreed past“).
  - Conflict resolution can occur on the server-side or on the client-side

Weak Consistency: More Concepts

Definition

Monotonic Read Consistency

If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.

Definition

Monotonic Write Consistency

A write operation by a process on a data item *u* is completed before any successive write operation on *u* by the same process (i.e. system guarantees to serialize writes by the same process).

Definition

Read-your-Writes Consistency

After a process has updated a data item, it will never see an older value on subsequent accesses.

Weak Consistency: Causal Consistency

Definition

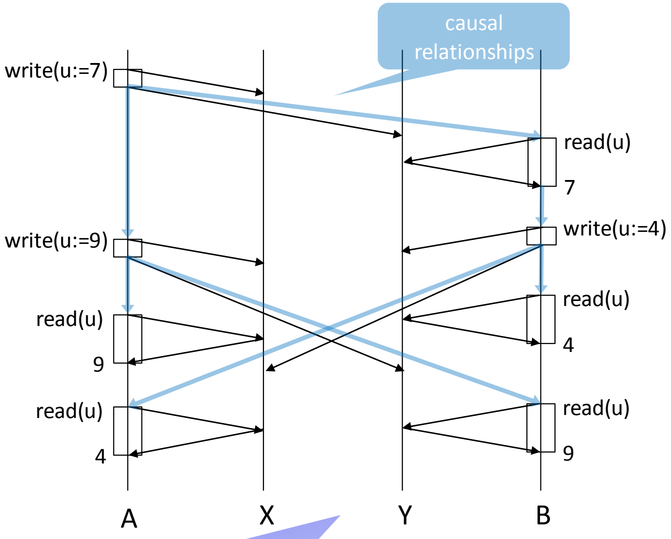
A system provides causal consistency if memory operations that potentially are *causally related* are seen by every node of the system in the same order. Concurrent writes (i.e. ones that are not causally related) may be seen in different order by different nodes.

Definition

The following pairs of operations are causally related:

- Two writes by the same process to any memory location.
- A read followed by a write of the same process (even if the write addresses a different memory location).
- A read that returns the value of a write from any process.
- Two operations that are transitively related according to the above conditions.

Causal Consistency: Example



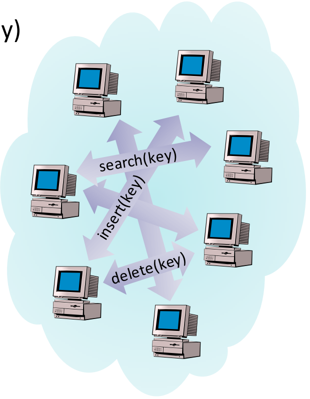
This execution is causally consistent, but NOT sequentially consistent

Large-Scale Fault-Tolerant Systems

- How do we build these highly available, fault-tolerant systems consisting of 1k, 10k,…, 1M nodes?
- Idea: Use a completely decentralized system, with a focus on availability, only giving weak consistency guarantees. This general approach has been popular recently, and is known as, e.g.
  - Cloud Computing: Currently popular umbrella name
  - Grid Computing: Parallel computing beyond a single cluster
  - Distributed Storage: Focus on storage
  - Peer-to-Peer Computing: Focus on storage, affinity with file sharing
  - Overlay Networking: Focus on network applications
  - Self-Organization, Service-Oriented Computing, Autonomous Computing, etc.
- Technically, many of these systems are similar, so we focus on one.

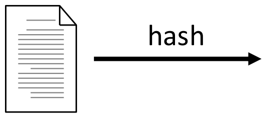
P2P: Distributed Hash Table (DHT)

- Data objects are distributed among the peers
  - Each object is uniquely identified by a key
- Each peer can perform certain operations
  - Search(key) (returns the object associated with key)
  - Insert(key, object)
  - Delete(key)
- Classic implementations of these operations
  - Search Tree (balanced, B-Tree)
  - Hashing (various forms)
- “Distributed” implementations
  - Linear Hashing
  - Consistent Hashing



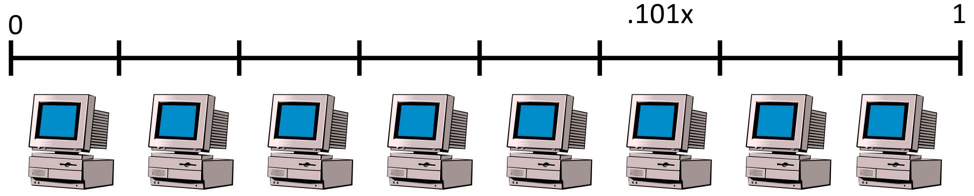
Distributed Hashing

- The hash of a file is its key



.10111010101110011… ≈ .73

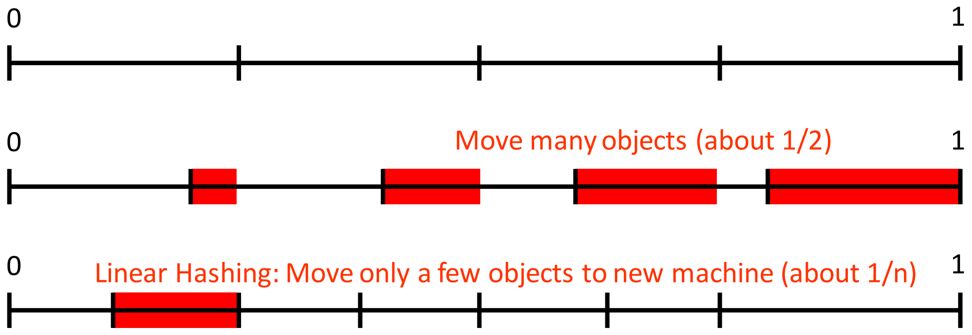
- Each peer stores data in a certain range of the ID space [0,1)



- Instead of storing data at the right peer, just store a forward-pointer

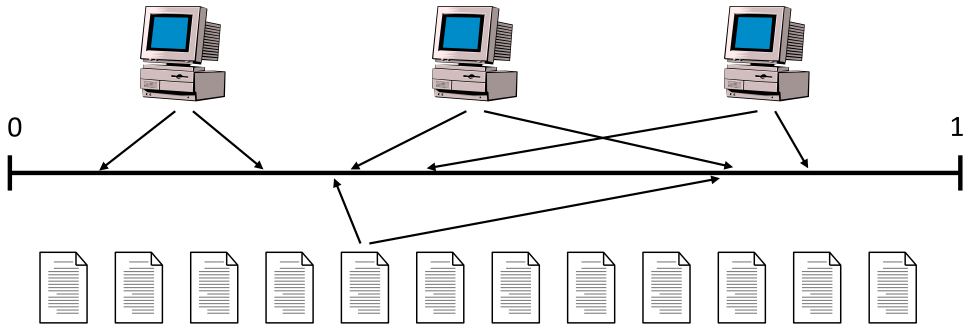
Linear Hashing

- Problem: More and more objects should be stored → Need to buy new machines!
- Example: From 4 to 5 machines


0
1
0
Move many objects (about 1/2)
1
0
Linear Hashing: Move only a few objects to new machine (about 1/n)
1

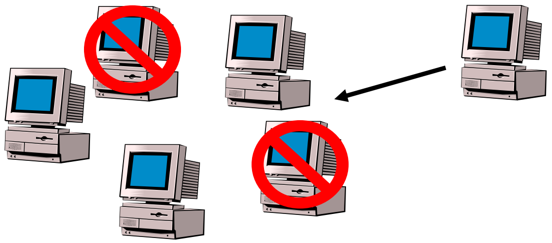
Consistent Hashing

- Linear hashing needs central dispatcher
- Idea: Also the machines get hashed! Each machine is responsible for the files closest to it
- Use multiple hash functions for reliability!

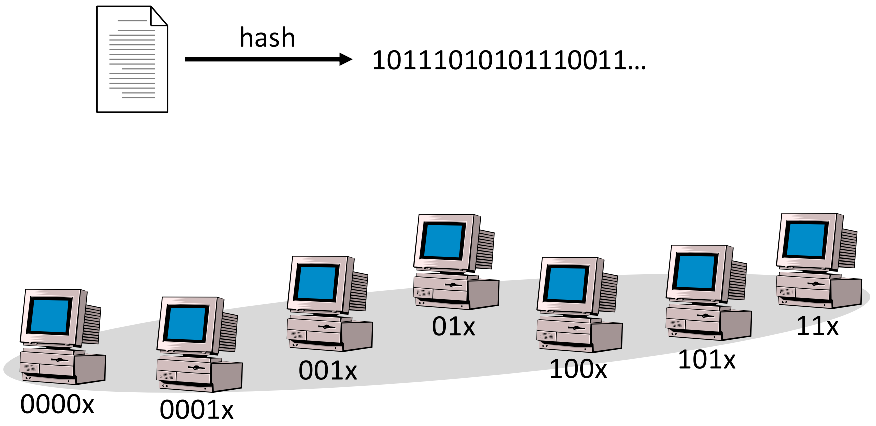


Search & Dynamics

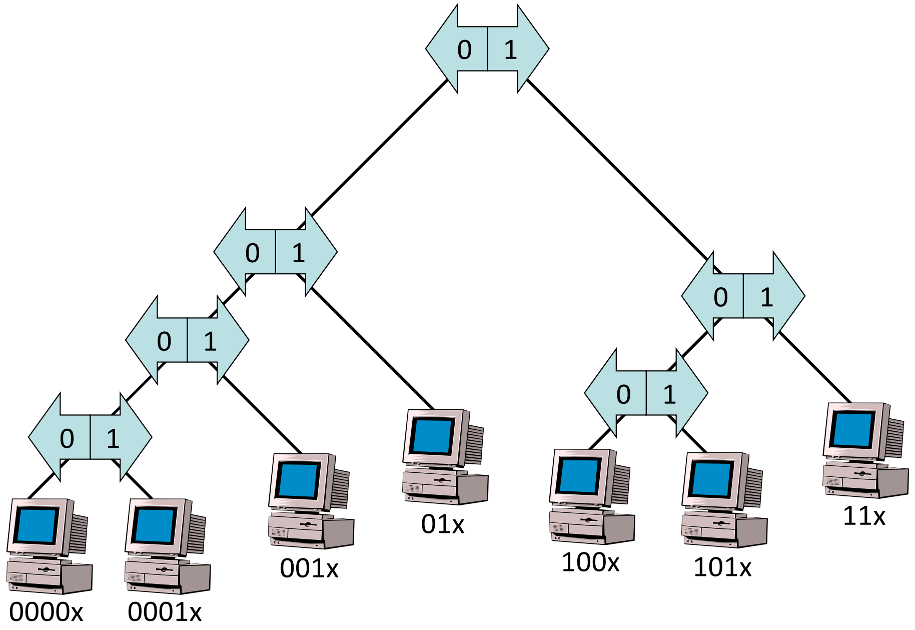
- Problem with both linear and consistent hashing is that all the participants of the system must know all peers…
  - Peers must know which peer they must contact for a certain data item
  - This is again not a scalable solution…
- Another problem is dynamics!
  - Peers join and leave (or fail)



P2P Dictionary = Hashing



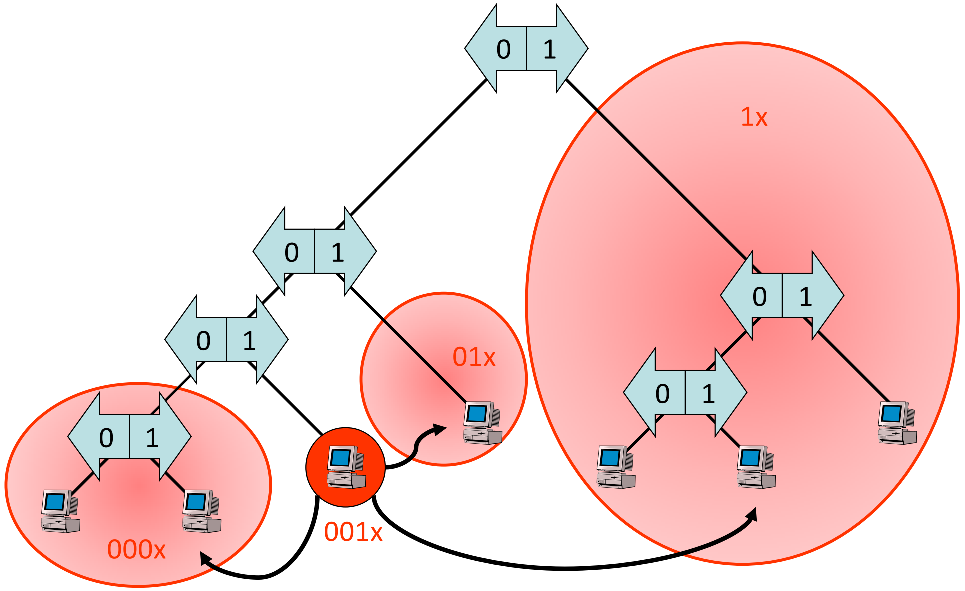
P2P Dictionary = Search Tree



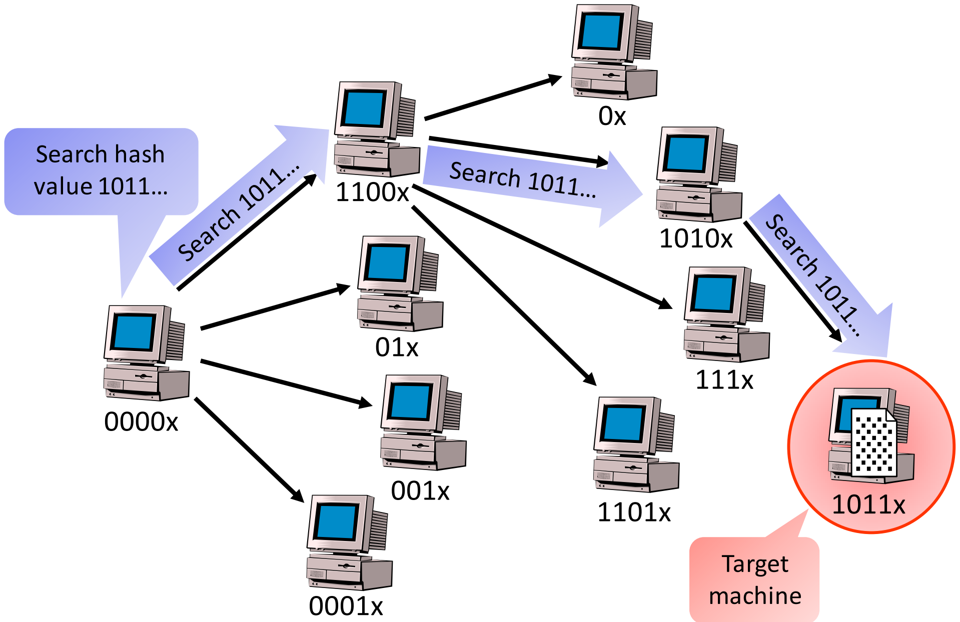
Storing the Search Tree

- Where is the search tree stored?
- In particular, where is the root stored?
  - What if the root crashes?! The root clearly reduces scalability & fault tolerance…
  - Solution: There is no root…!
- If a peer wants to store/search, how does it know where to go?
  - Again, we don’t want that every peer has to know all others…
  - Solution: Every peer only knows a small subset of others

The Neighbors of Peers 001x

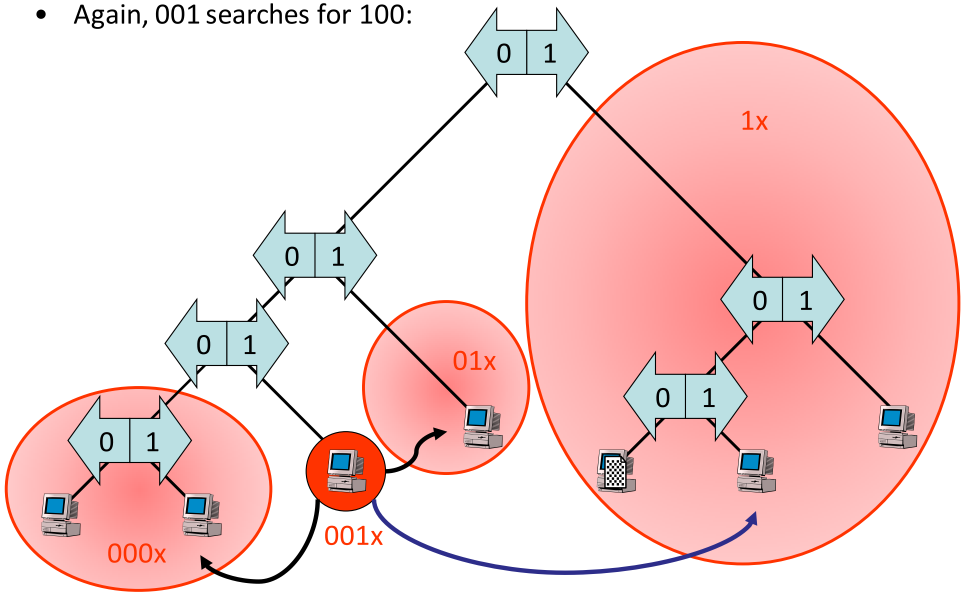


P2P Dictionary: Search



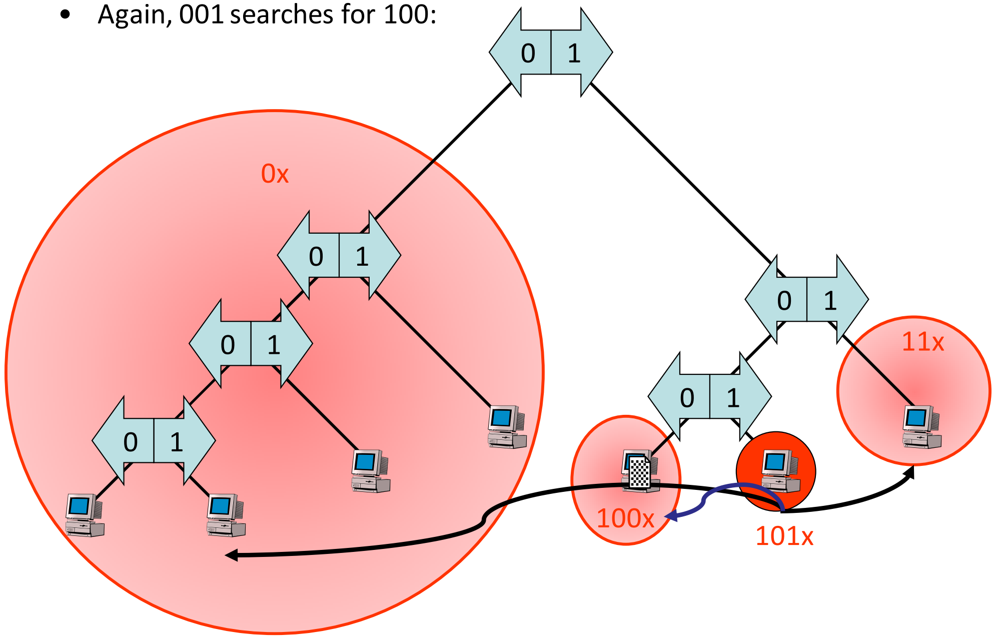
P2P Dictionary: Search

- Again, 001 searches for 100:



P2P Dictionary: Search

- Again, 001 searches for 100:



Search Analysis

- We have *n* peers in the system
- Assume that the “tree” is roughly balanced
  - Leaves (peers) on level $\log_2 n$ ± constant
- Search requires O(log n) steps
  - After kth step, the search is in a subtree on level k
  - A “step” is a UDP (or TCP) message
  - The latency depends on P2P size (world!)
- How many peers does each peer have to know?
  - Each peer only needs to store the address of $\log_2 n$ ± constant peers
  - Since each peer only has to know a few peers, even if *n* is large, the system scales well!

Peer Join

- How are new peers inserted into the system?
- Step 1: Bootstrapping
- In order to join a P2P system, a joiner must already know a peer already in the system
- Typical solutions:
  - Ask a central authority for a list of IP addresses that have been in the P2P regularly; look up a listing on a web site
  - Try some of those you met last time
  - Just ping randomly (in the LAN)

Peer Join

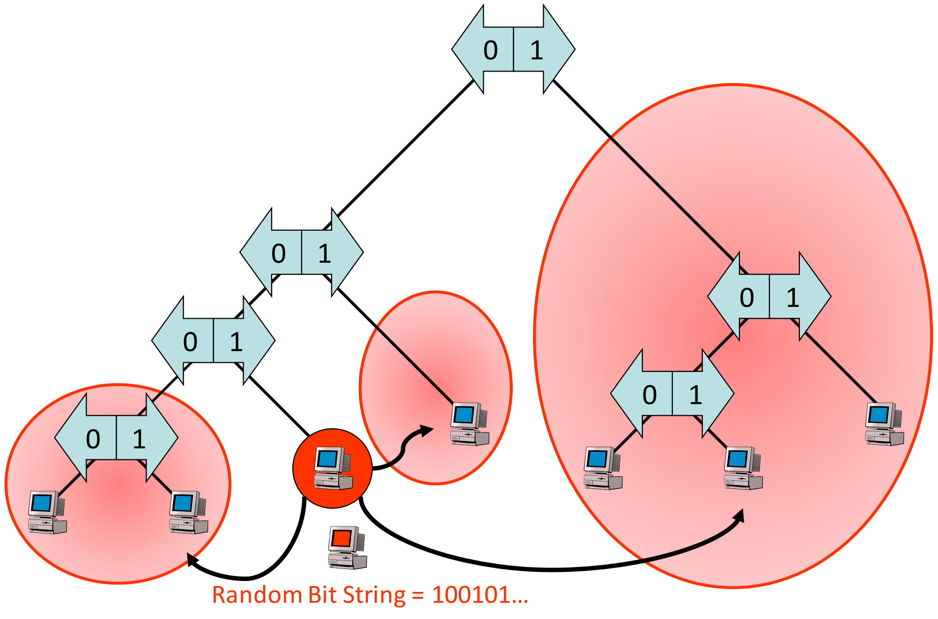
- Step 2: Find your place in the P2P system

Peer ID!

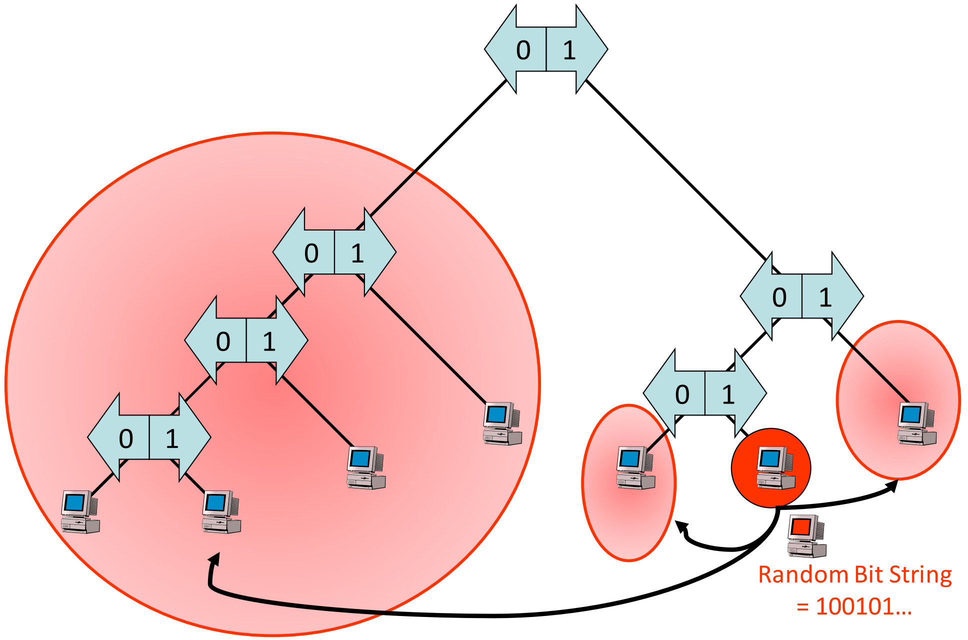
- Typical solution:
  - Choose a random bit string (which determines the place in the system)
  - Search* for the bit string
  - Split with the current leave responsible for the bit string
  - Search* for your neighbors

* These are standard searches

Example: Bootstrap Peer with 001

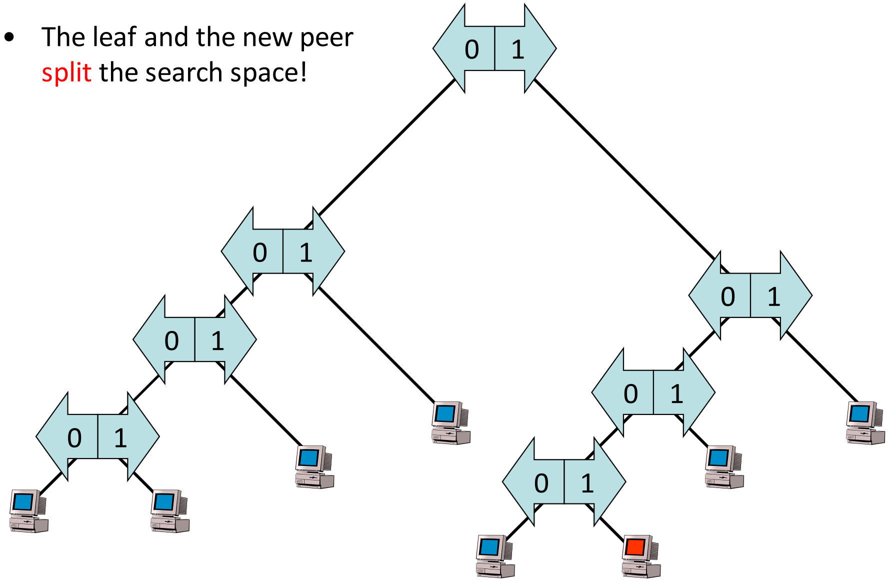


New Peer Searches 100101...

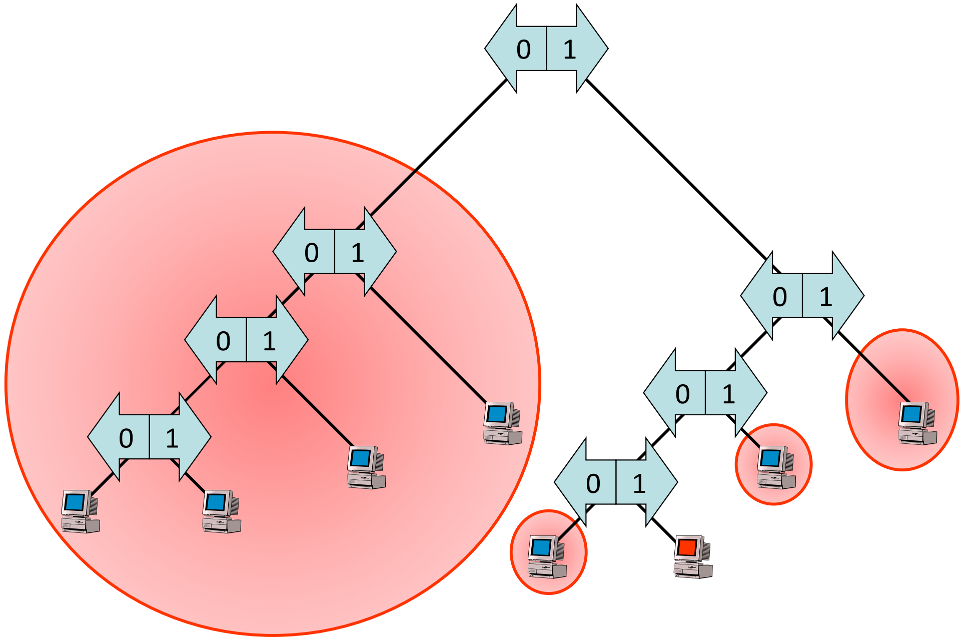


New Peer found leaf with ID 100...

- The leaf and the new peer split the search space!



Find Neighbors



Peer Join: Discussion

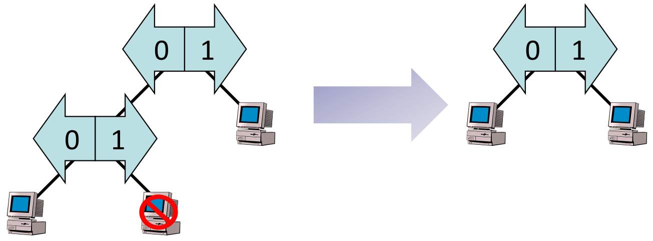
- If tree is balanced, the time to join is
  - O(log n) to find the right place
  - O(log n)·O(log n) = O(log^2 n) to find all neighbors

A regular search…

- It is be widely believed that since all the peers choose their position randomly, the tree will remain more or less balanced
  - However, theory and simulations show that this is not really true!

Peer Leave

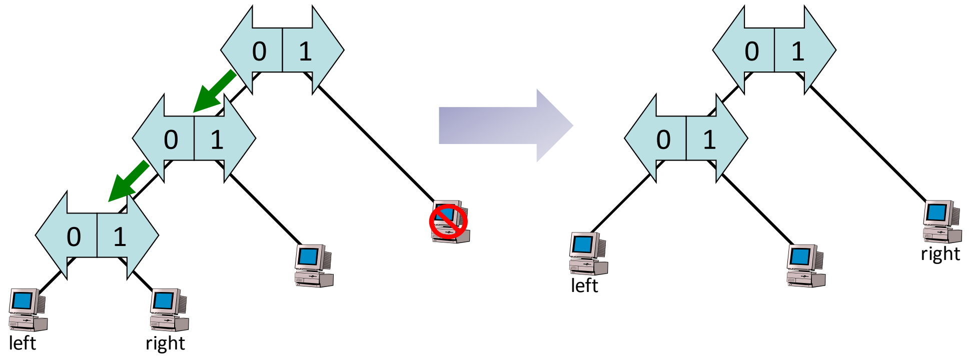
- Since a peer might leave spontaneously (there is no leave message), the leave must be detected first
- Naturally, this is done by the neighbors in the P2P system (all peers periodically ping neighbors)
- If a peer leave is detected, the peer must be replaced. If peer had a sibling leaf, the sibling might just do a “reverse split”:



- If a peer does not have a sibling, search recursively!

Peer Leave: Recursive Search

- Find a replacement:
  1. Go down the sibling tree until you find sibling leaves
  2. Make the left sibling the new common node
  3. Move the free right sibling to the empty spot



Fault-Tolerance?

- In P2P file sharing, only pointers to the data is stored
  - If the data holder itself crashes, the data item is not available anymore
- What if the data holder is still in the system, but the peer that stores the pointer to the data holder crashes?
  - The data holder could advertise its data items periodically
  - If it cannot reach a certain peer anymore, it must search for the peer that is now responsible for the data item, i.e., the peer’s ID is closest to the data item’s key
- Alternative approach: Instead of letting the data holders take care of the availability of their data, let the system ensure that there is always a pointer to the data holder!
  - Replicate the information at several peers
  - Different hashes could be used for this purpose

Questions of Experts…

- Question: I know so many other structured peer-to-peer systems (Chord, Pastry, Tapestry, CAN…); they are completely different from the one you just showed us!

- Answer: They *look* different, but in fact the difference comes mostly from the way they are presented (I give a few examples on the next slides)

The Four P2P Evangelists

- If you read your average P2P paper, there are (almost) always four papers cited which "invented" efficient P2P in 2001:

Chord CAN Pastry Tapestry

- These papers are somewhat similar, with the exception of CAN (which is not really efficient)

- So what are the „Dead Sea scrolls of P2P"?

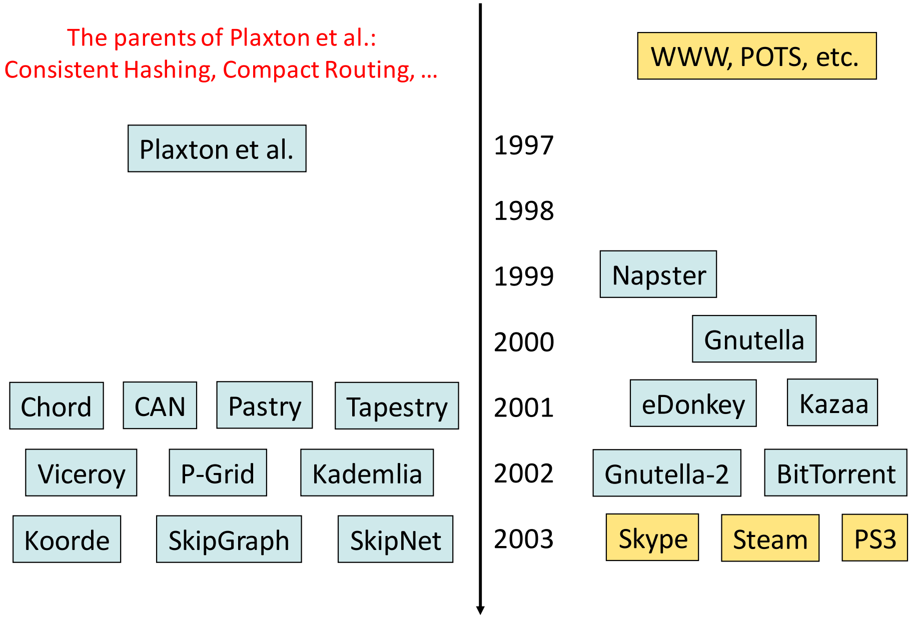
Intermezzo: "Dead Sea Scrolls of P2P"

> „Accessing Nearby Copies of Replicated Objects in a Distributed Environment" [Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa, SPAA 1997]

- Basically, the paper proposes an efficient search routine (similar to the four famous P2P papers)
  - In particular search, insert, delete, storage costs are all logarithmic, the base of the logarithm is a parameter

- The paper takes latency into account
  - In particular it is assumed that nodes are in a metric, and that the graph is of „bounded growth" (meaning that node densities do not change abruptly)

Intermezzo: Genealogy of P2P

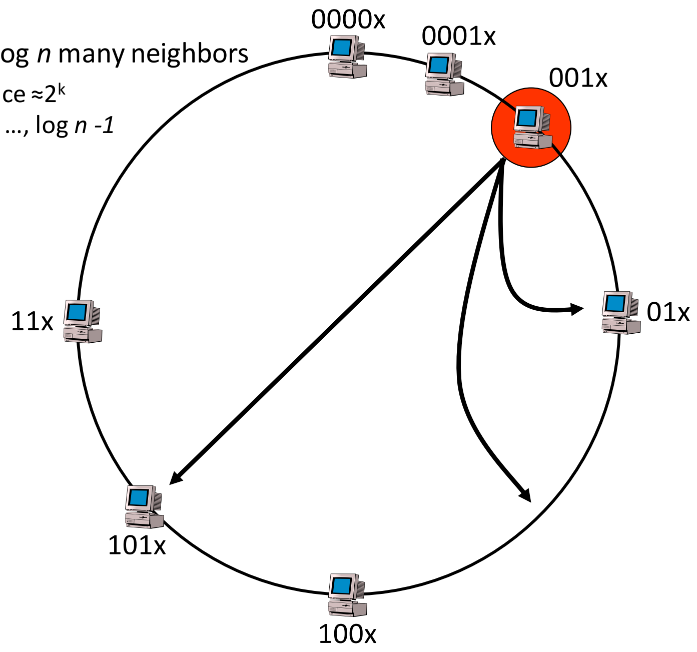


Chord

- Chord is the most cited P2P system
- Most discussed system in distributed systems and networking books, for example in Edition 4 of Tanenbaum's Computer Networks
- There are extensions on top of it, such as CFS, Ivy…

Chord

- Every peer has log *n* many neighbors
  - One in distance ≈2^k for k = 0,1, 2, …, log *n -1*



Example: Dynamo

- Dynamo is a key-value storage system by Amazon (shopping carts)
- Goal: Provide an "always-on" experience
  - Availability is more important than consistency
- The system is (nothing but) a DHT
- Trusted environment (no Byzantine processes)
- Ring of nodes
  - Node n_i is responsible for keys between n_{i-1} and n_i
  - Nodes join and leave dynamically
- Each entry replicated across *N* nodes
- Recovery from error:
  - When? On read
  - How? Depends on application, e.g. "last write wins" or "merge"
  - One vector clock per entry to manage different versions of data

Basically what we talked about

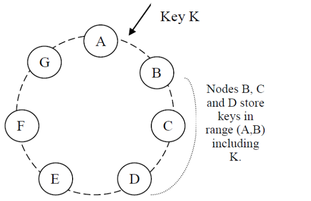
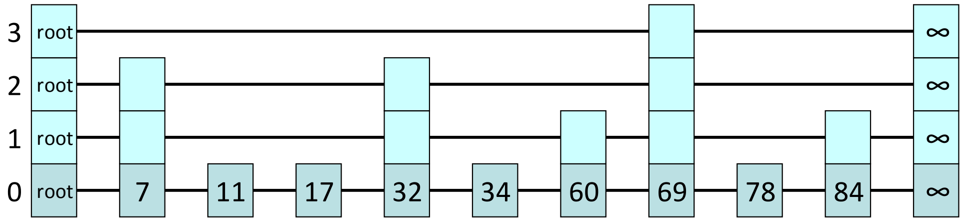


Figure 2: Partitioning and replication of keys in Dynamo ring.

Skip List

- How can we ensure that the search tree is balanced?
  - We don't want to implement distributed AVL or red-black trees…
- Skip List:
  - (Doubly) linked list with sorted items
  - An item adds additional pointers on level 1 with probability ½. The items with additional pointers further add pointers on level 2 with prob. ½ etc.
  - There are $\log_2 n$ levels in expectation
- Search, insert, delete: Start with root, search for the right interval on highest level, then continue with lower levels



Skip List

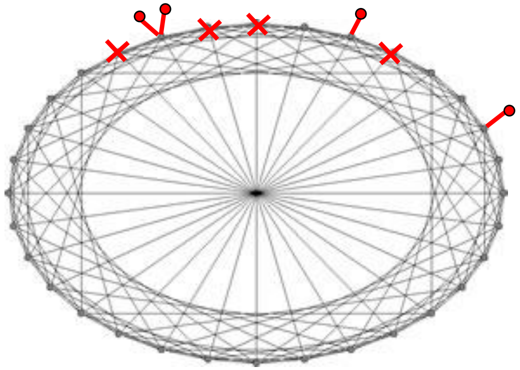
- It can easily be shown that search, insert, and delete terminate in O(log n) expected time, if there are n items in the skip list
- The expected number of pointers is only twice as many as with a regular (doubly) linked list, thus the memory overhead is small
- As a plus, the items are always ordered…

P2P Architectures

- Use the skip list as a P2P architecture
  - Again each peer gets a random value between 0 and 1 and is responsible for storing that interval
  - Instead of a root and a sentinel node ("∞"), the list is short-wired as a ring
- Use the Butterfly or DeBruijn graph as a P2P architecture
  - Advantage: The node degree of these graphs is constant → Only a constant number of neighbors per peer
  - A search still only takes O(log n) hops

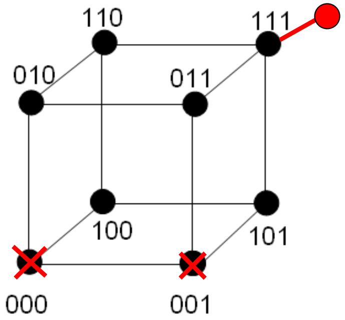
Dynamics Reloaded

- Churn: Permanent joins and leaves
  - Why permanent?
  - Saroiu et al.: „A Measurement Study of P2P File Sharing Systems": Peers join system for one hour on average
  - Hundreds of changes per second with millions of peers in the system!
- How can we maintain desirable properties such as
  - connectivity
  - small network diameter
  - low peer degree?



A First Approach

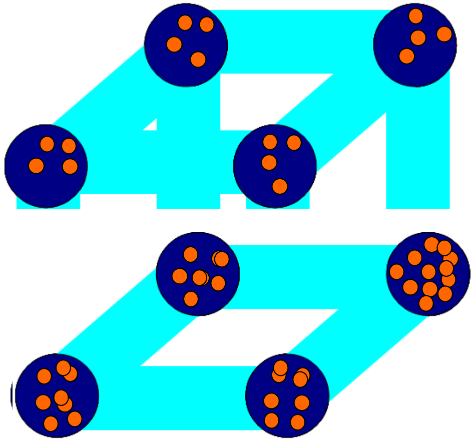
- A fault-tolerant hypercube?
- What if the number of peers is not 2^i?
- How can we prevent degeneration?
- Where is the data stored?
- Idea: Simulate the hypercube!



Simulated Hypercube

- Simulation: Each node consists of several peers

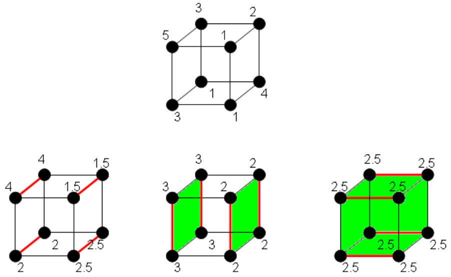
- Basic components:
- Peer distribution
  - Distribute peers evenly among all hypercube nodes
  - A token distribution problem
- Information aggregation
  - Estimate the total number of peers
  - Adapt the dimension of the simulated hypercube



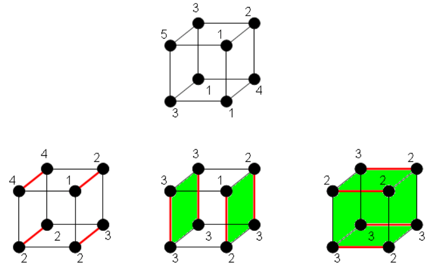
Peer Distribution

- Algorithm: Cycle over dimensions and balance!
- Perfectly balanced after d rounds

Dimension of hypercube



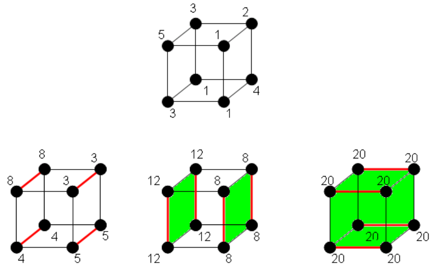
- Problem 1: Peers are not fractional!
- Problem 2: Peers may join/leave during those d rounds!
- “Solution”: Round numbers and ignore changes during the d rounds



Information Aggregation

- Goal: Provide the same (good!) estimation of the total number of peers currently in the system to all nodes
- Algorithm: Count peers in every sub-cube by exchanging messages wih the corresponding neighbor!
- Correct number after d rounds

- Problem: Peers may join/leave during those d rounds!
- Solution: Pipe-lined execution



- It can be shown that all nodes get the same estimate
- Moreover, this number represents the correct state d rounds ago!

Composing the Components

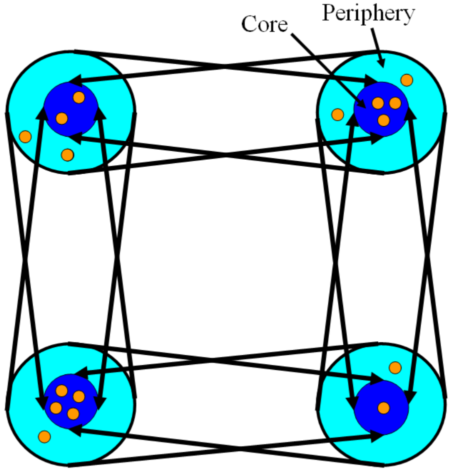
- The system permanently runs
  - the peer distribution algorithm to balance the nodes
  - the information aggregation algorithm to estimate the total number of peers and change the dimension accordingly

- How are the peers connected inside a simulated node, and how are the edges of the hypercube represented?

- Where is the data of the DHT stored?

Distributed Hash Table

- Hash function determines node where data is replicated
- Problem: A peer that has to move to another node must store different data items
- Idea: Divide peers of a node into core and periphery
  - Core peers store data
  - Peripheral peers are used for peer distribution
- Peers inside a node are completely connected
- Peers are connected to all core peers of all neighboring nodes



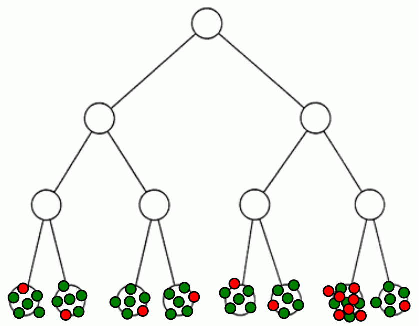
Evaluation

- The system can tolerate O(log *n*) joins and leaves each round
- The system is never fully repaired, but always fully functional!
- In particular, even if there are O(log *n*) joins/leaves per round we always have
  - at least one peer per node
  - at most O(log *n*) peers per node
  - a network diameter of O(log *n*)
  - a peer degree of O(log *n*)

Number of neighbors/connections

Byzantine Failures

- If Byzantine nodes control more and more corrupted nodes and then crash all of them at the same time ("sleepers"), we stand no chance.
- Idea: Assume that the Byzantine peers are the minority. If IDs are chosen uniformly at random, the number of peers in any fraction of the ID space can be bounded with high probability. If there are many Byzantine peers in the same region, they can be detected (because of their unusual high density). This unsafe fraction is ignored.



Bitcoin

- Bitcoin is a decentralized peer-to-peer currency
  - Network of untrusted nodes clears transactions and tracks balances
  - Own currency: "Bitcoins"
  - Can be traded for traditional currencies (EUR, USD, CHF)
- Bitcoin is
  - Global
  - Fast
  - Irreversible
  - Without intermediaries
  - Anonymous/Pseudonymous
- Public record of all transactions (blockchain)
  - Only includes valid transactions
  - Eventually consistent

Where do I sign up?

- Private Key
  - A user locally generates a random private key.
  - Used to prove ownership of Bitcoins.
- Public Key
  - The public key is derived from the private key, using ECDSA (elliptic curve digital signing algorithm).
  - Used to verify ownership of Bitcoins.
- Address
  - Derived from a hash of the public key, by adding a version number and a checksum and encoding it in base58.
  - Users can have an arbitrary number of addresses.
  - The address is then either published (donations) or sent along with an invoice (payments).
  - Users are encouraged to create a new key triplet for each incoming payment.

Transactions

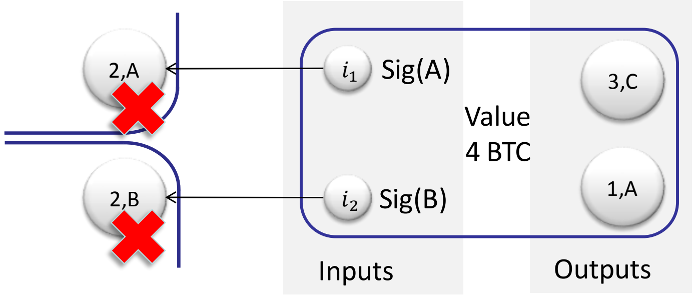
- Bitcoin implements a replicated transaction history
  - A transaction transfers ownership of some coins from one or more sending addresses to one or more receiving addresses.
  - Bitcoins only exist as part of transactions.
  - The current balance of an address is simply the sum of all bitcoins associated with that address, and have not been spent yet.

Transactions continued

- Bitcoin tracks tuples (called *outputs*) of bitcoins and an owner address

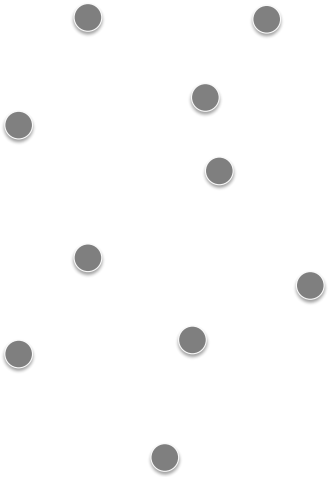
  (value, address)
- A transaction claims some outputs by including *inputs* and a valid signature for the owner address.
- Inputs are references to the transaction that created the claimed output

  (transaction hash, index)



The Bitcoin Network

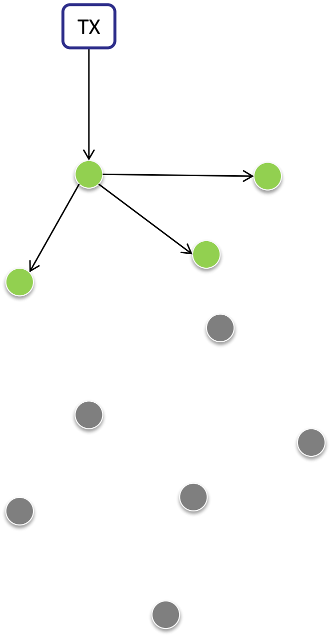
- Bitcoin is a network of nodes, controlled by a multitude of users
- The nodes connect to randomly selected nodes in the network
- Each node holds a full replica of the transaction history (blockchain) and the unspent transaction output set
- Each node verifies each transaction and applies it to its replica if valid



Tracing a Transaction in the Network

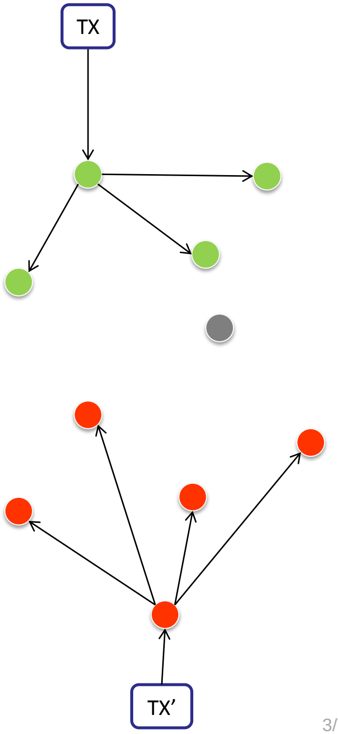
- Transactions enter the Bitcoin network at any node
- Nodes forward transactions to their neighbors in the network
- A node receiving a transaction attempts to apply it to its local replica:
  - Verify signatures
  - Mark the transaction inputs as claimed
  - Create new outputs

What if they are already claimed?



What could possibly go wrong?

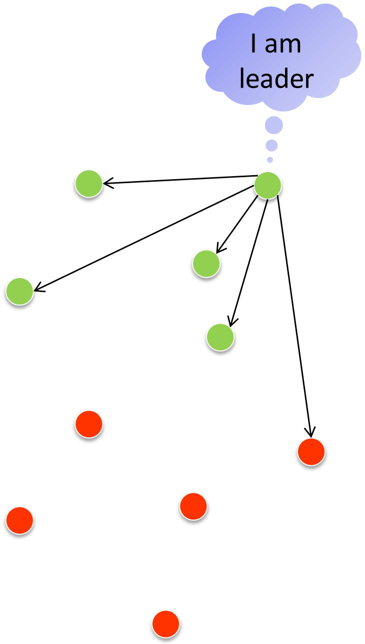
- Transactions can conflict, if they attempt to claim the same output
  - "Double spending" attack
  - Only one of the transactions can be valid at a time
  - Which one to choose?
- We simply decide locally and tentatively apply the first transaction we received
  - But then we are inconsistent
  - How do we re-establish consistency?



Blocks

- Basic idea: Use a leader
- Elect a new leader at regular intervals
- Leader *confirms* a set of transactions* that have not been *confirmed* so far by previous leaders.
- The leader broadcasts a *block*.
- Along the set of transactions the block contains a nonce (a number) and a reference to the previous block.
- But who should be the leader? Let the nodes try to solve some hard problem, the first to solve it is the leader (Proof-of-Work).

*Leaders choose their own transactions, and transactions where the output values is smaller than the input values.



Proof-of-Work

- Bitcoin's leader election is based on finding a Proof-of-Work
- A Proof-of-Work system is characterized by:
  - A non-reversible work function $F(x)$, where x is the block
  - A validity predicate $P(y)$

$$\text{Find } x \text{ so that } P\big(F(x)\big) = True$$

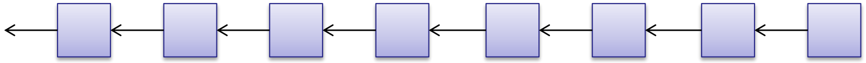
- In Bitcoin

$$\mathrm{F(x)} = SHA256(SHA256(x))$$
$$P(y) \leftrightarrow \text{ first } b \text{ bits of } y \text{ are } 0$$

- b is a "global variable" so that a solution is found about once every 10 minutes in expectation (in the whole network). The variable b is adjusted (-4,…,+4) after 2016 leaders have been elected. Currently b ≈ 60.
- Nodes attempting to find a Proof-of-Work are called *miners*.

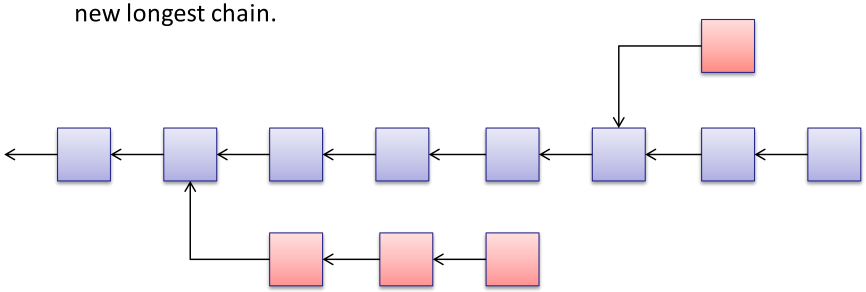
Blockchain

- How to avoid that a leader reverts the decisions of a previous leader?
  - Blocks are chained together (using the reference to the last block).
  - Every transaction confirmed in a block is also confirmed in its successors.
  - If a later block contradicts with a predecessor it is invalid.
  - Blocks therefore incrementally build a consistent transaction history.



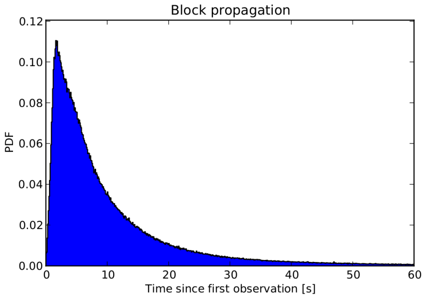
Blockchain Forks

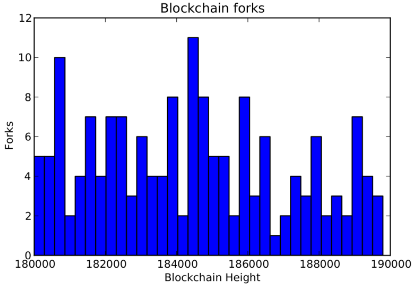
- Are we really any better off than simply using transactions?
  - What happens if multiple blocks are created as a successor to a block?
  - Can’t an attacker simply go back and start its own chain?
- Solution: Extend the longest chain
  - Eventually there will be only one longest chain, and all nodes will agree.
  - An attacker would have to compete with the rest of the network to create a new longest chain.



Do Blockchain Forks happen?

- Slow propagation of blocks in the network causes Blockchain forks
- Time to propagate a block is proportional to the block’s size
- Currently
  - Blocks limited to 1MB in size
  - Time to propagate a block to 50% of the network $\sim$6 seconds
  - Long tail: 95th percentile at 40 seconds
  - In 10,000 blocks we have 163 forks, which is about 1.6%





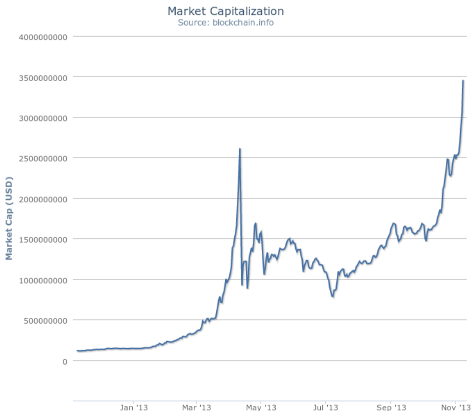
Mining

- Blocks are currently worth $\sim 25'000$ USD
  - Miners are involved in an arms race, fighting over the block reward.
  - Companies are building ever more powerful and sophisticated mining equipment.
  - CPUs were replaced by GPUs, GPUs were replaced by FPGAs and they too were replaced by ASICs
- But mining is not useful in itself
  - It just provides the leader election and limits the influence of individuals
  - Is there a way to make the mining process useful?

There are some attempts, e.g., primecoin, that are more useful, but none of them is searching for a cure for cancer yet.

Bitcoin Today

- Equivalent of > 9 billion USD of bitcoins in circulation
- $\sim 10'000$ active nodes in the network at any time
- $\sim 50'000$ transactions per day
- $\sim 42 \times 10^{18}$ FLOPS total computational power in the network
- Mining with CPU (or even GPU) is not cost efficient



Market Capitalization

Source: blockchain.info

Ongoing Research in Bitcoin

- Pseudonymity / Anonymity
  - The ledger is open for everyone to inspect
  - Isn't this a privacy issue?
- Scalability
  - As more and more people start using Bitcoin the number of transactions grows quickly
  - The current protocol does not scale
- Confirmation speed
  - For a transaction to be confirmed it has to be included in a block
  - A block is found every 10 minutes in expectation
- New Use-cases
  - Smart Property
  - Micropayments
  - Escrow transactions
  - …

History

- Bitcoin was published by Satoshi Nakamoto in 2008
- Satoshi remains to date anonymous, we don't even know whether it's an individual or a group of people.
- By early 2009 the prototype was up and running.
- Satoshi disappeared in 2010, after transferring control to open-source developers, saying that he had "moved on to other things".
- Satoshi did not pre-mine any blocks, the genesis block (0th block) contains a quote from The Times of the 3rd january 2009.

Credits

- Concurrent hashing and random trees have been proposed by Karger, Lehman, Leighton, Levine, Lewin, and Panigrahy, 1997.
- Chord is from Stoica et al. 2001.
- The churn-resistent P2P System is due to Kuhn et al., 2005.
- Dynamo is from DeCandia et al., 2007.
- Byzantine faults in DHTs is from Awerbuch and Scheideler, 2004.
- Bitcoin is due to Nakamoto (likely a pseudonym), 2008.
  - Bitcoin Propagation analysis by Decker and Wattenhofer, 2013

That's all, folks!

Questions & Comments?

Roger Wattenhofer