DD2451
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

Lecture 10

Peer to Peer Systems

Mads Dam

Autumn/Winter 2011

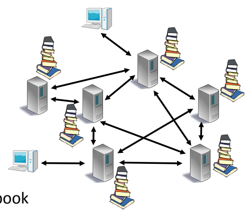
Overview

- Consistency vs availability
  - CAP and ACID vs BASE
- Linear and consistent hashing
- DHT's
- P2P search trees
- Join and leave
- P2P architectures
- Supporting churn

Why Do We Use Replication?

- Fault tolerance:
  - If some nodes fail, information is not lost
  - System should look as if there is no concurrency
  - I.e. it should maintain "consistency"
  - But it should still survive crashes and attacks
- Availability
  - But consistency is expensive
  - In particular for very large systems
  - What if availability is more important than consistency?

Example: Bookstore



- Consider a bookstore which sells its wares over the web:
- Which properties do we want?
- Consistency - for each user the system behaves reliably
- 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 be able to function.

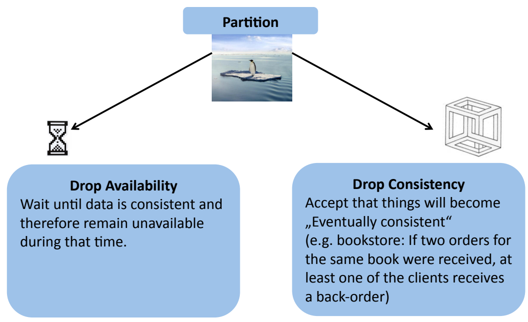
The CAP Theorem

Theorem: It is impossible for a distributed computer system to simultaneously provide **C**onsistency, **A**vailability and **P**artition Tolerance. A distributed system can satisfy any two of these guarantees at the same time but not all three.

Proof: The proof is rather trivial. If a partition occurs into networks N1 and N2, A in N1 writes to v in N1 and B in N2 reads from v. If N1 and N2 are not connected either A and B must wait to synchronize – availability is lost – or the system may become inconsistent.

The second part of the statement: We need to exhibit solutions in each case, but we already have the tools at hand.

CAP Theorem: Consequences



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 which 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 and BASE

ACID

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

BASE

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

Large Networks

- Scalability:
  - Support large numbers of nodes
  - Order of 10^3 – 10^6 nodes
  - May be geographically dispersed
  - Or may be used for e.g. scalable storage inside data centers
- Availability:
  - Emphasis on fast response times
- Survivability:
  - System should survive "reasonable " failure scenarios

Robustness

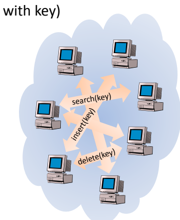
- Large network => failures unavoidable
  - *Churn*: Join and leave of nodes under operation
  - Leaves often without warning
  - Often inherent in P2P networks
  - Allow transient inconsistencies
- Consistency:
  - Large system + churn + transient inconsistency -> system is never in stable state
  - Eventual consistency = consistency if system is stable for sufficiently long (!)
- Byzantine fault tolerance
  - Still relevant to catch attackers, freeriders
  - Some references at end of last slide set

Examples

- Large scale distribution + weak consistency + high availability:
  - 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, self-*, Service-Oriented Computing, Autonomous Computing, etc.
- Technically, many of these systems are similar, so we focus on one.

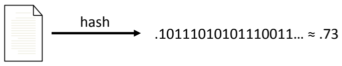
P2P: Distributed Hash Table (DHT)

- 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

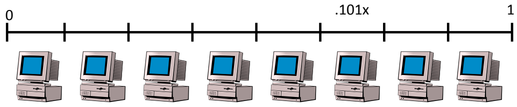


Distributed Hashing

- The hash of a file is its key



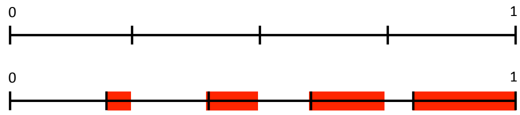
- Each peer stores data in a certain range of the ID space [0,1]



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

Vanilla Hashing

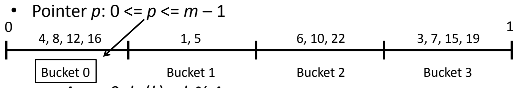
- Problem: More and more objects should be stored → Need to buy new machines!
- Example: From 4 to 5 machines



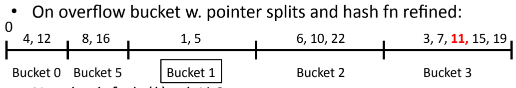
- New bucket needed => about ½ of all objects to be moved
- Is there a hash regime that allows buckets to be dynamically added and removed, with only local effects?

Linear Hashing

- Dynamically updatable hashing scheme
- Initially m buckets
- Initial hashing function $h_0(k) = f(k) \% m$
- Pointer p: $0 <= p <= m - 1$



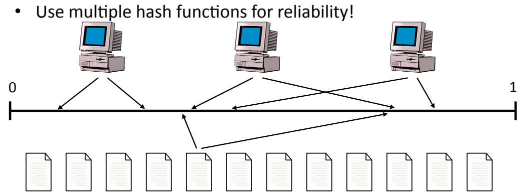
- $m = 4$, $p = 0$, $h_0(k) = k \% 4$
- On overflow bucket w. pointer splits and hash fn refined:



- New hash fn $h_1(k) = k \% 8$, etc etc

Consistent Hashing

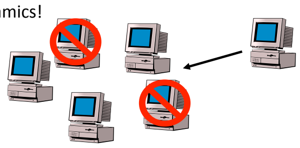
- Linear hashing needs central dispatcher
- Idea: Also the machines get hashed! Each machine is responsible for the files closest to it
- Use multiple hash functions for reliability!



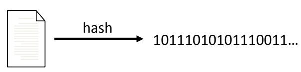
Karger et al: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, STOC 1997

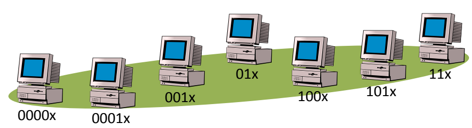
Search & Dynamics

- 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: dynamics!
  - Peers join and leave
  - Or fail . . .



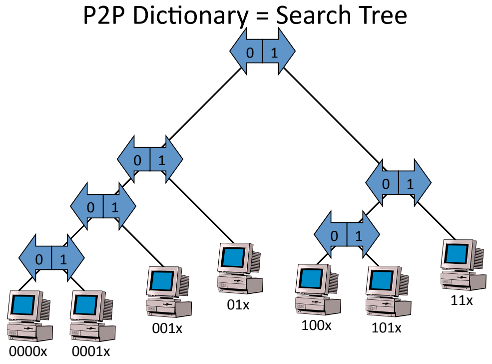
P2P Dictionary = Hashing





Question is: How do we find machine 101x?

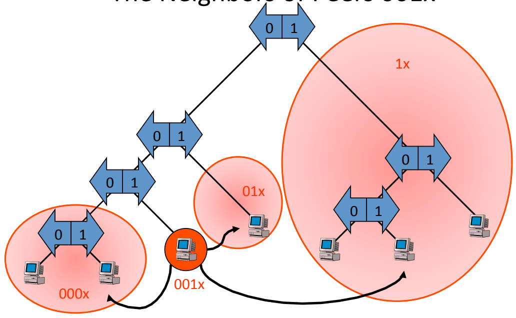
P2P Dictionary = Search Tree



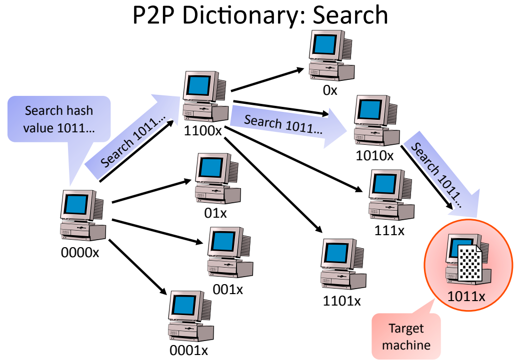
Storing the Search Tree

- 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

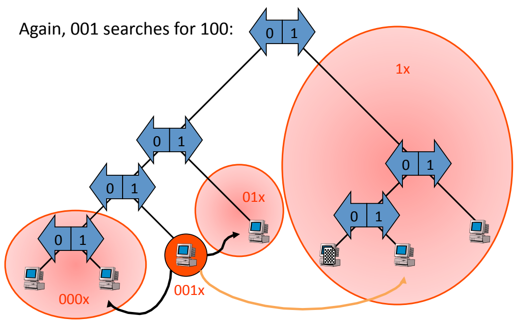


P2P Dictionary: Search



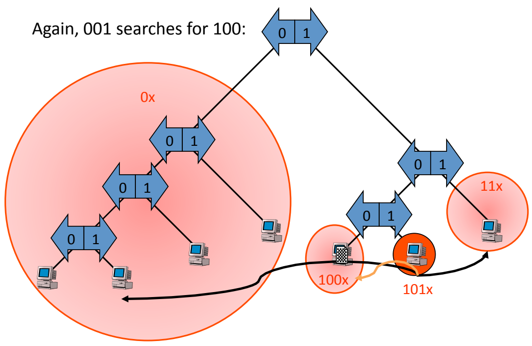
P2P Dictionary: Search

Again, 001 searches for 100:



P2P Dictionary: Search

Again, 001 searches for 100:



Search Analysis

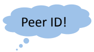
- We have n peers in the system
- Assume that the "tree" is roughly balanced
  - Leaves (peers) on level $\log_2 n \pm$ constant
- Search requires $O(\log n)$ steps
  - After k^{th} step, the search is in a subtree on level k
  - A "step" is a UDP (or TCP) message
  - The latency depends on P2P size (world!)
- How many peers does each peer have to know?
  - Each peer only needs to store the address of $\log_2 n \pm$ 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 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

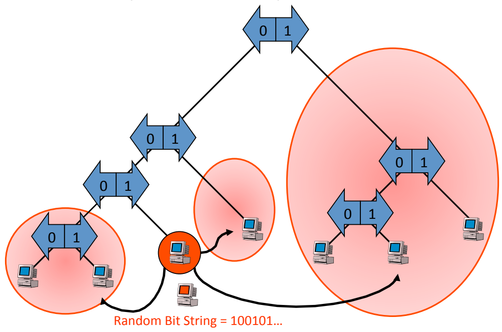
- Step 2: Find your place in the P2P system
- Typical solution:



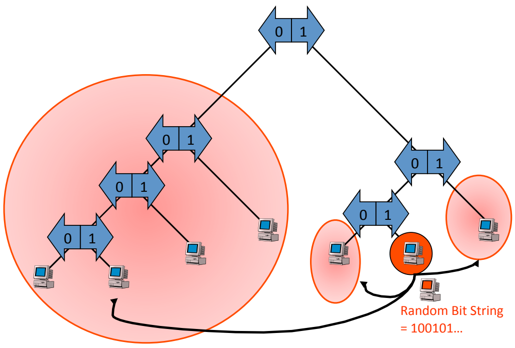
  - Choose a random bit string (which determines the place in the system)
  - Search* for the bit string
  - Split with the current leaf responsible for the bit string
  - Search* for your neighbors

* These are standard searches

Example: Bootstrap Peer with 001

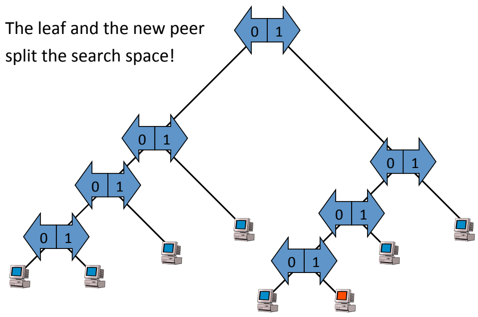


New Peer Searches 100101...

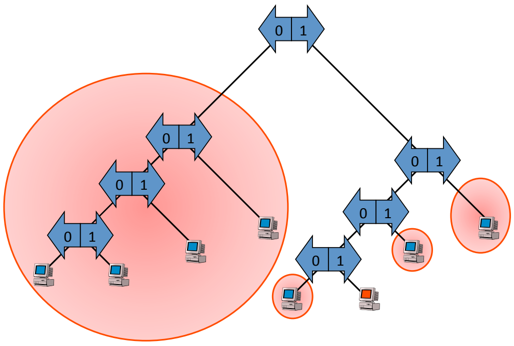


New Peer found leaf with ID 100...

The leaf and the new peer split the search space!



Find Neighbors

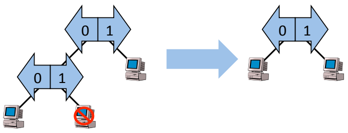


Peer Join: Discussion

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

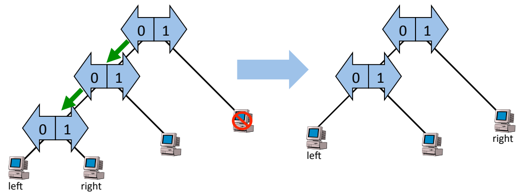
- Since a peer might leave spontaneously, the leave must be detected first
- Typically, all peers periodically ping neighbors
- If a peer leave is detected, the peer must be replaced. If peer had a sibling leaf, the sibling might just do a "reverse split":



- 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



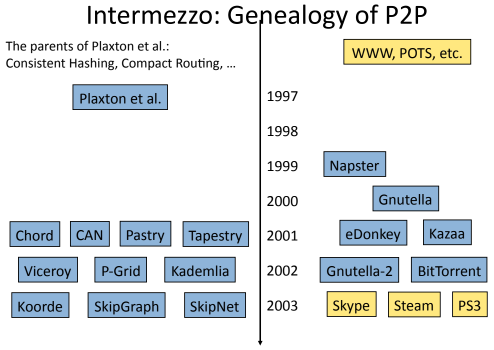
Fault-Tolerance?

- Only pointers to the data is stored
  - If the data holder crashes, hope for backup
- What if 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
- 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

Intermezzo: Genealogy of P2P



Plaxton et al: Accessing Nearby Copies of Replicated Objects in a Distributed Environment, SPAA 1997

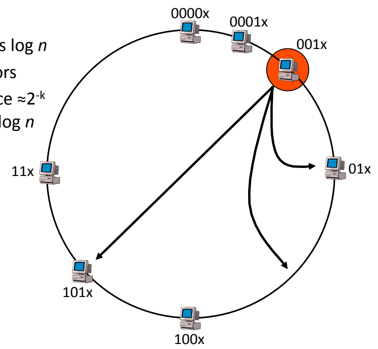
Chord

- 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…

Stoica et al: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, SIGCOMM 2001

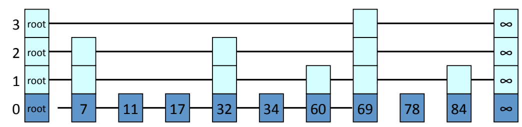
Chord

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



Skip List

- How can we ensure that the search tree is balanced?
- 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

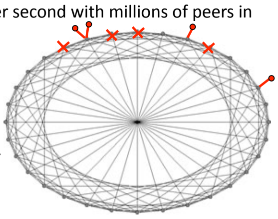
- It can easily be shown that search, insert, and delete terminate in O(log *n*) expected time, if there are *n* items in the skip list
- The expected number of pointers is only twice as many as with a regular linked list, thus the memory overhead is small
- As a plus, the items are always ordered…

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
- Check Wattenhofers chapter for todays lecture

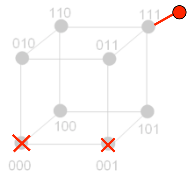
Dynamics Reloaded

- 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:
  - 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!

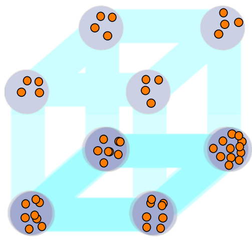


Simulated Hypercube

- 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

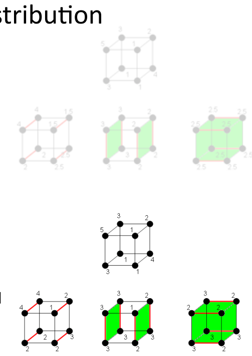


Peer Distribution

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

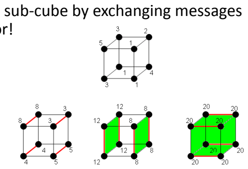
Dimension of hypercube

- 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 presently in the system to all nodes
- Algorithm: Count peers in every sub-cube by exchanging messages with the corresponding neighbor!
- Correct number after *d* rounds
- Problem: Peers may join/leave during those *d* rounds!
- Solution: Pipe-lined execution



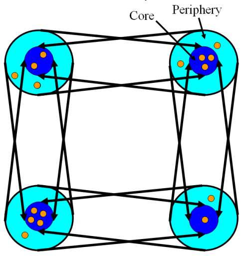
- It can be shown that all nodes get the same estimate
- Moreover, this number represents the correct state d rounds ago!

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 replace 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*)