Blockchain (BlCh)

Algorithms/Mechanisms for Fully Distributed Systems

OST Eastern Switzerland University of Applied Sciences

Thomas Bocek

11.11.2024

BitTorrent

- Last week: pkdns, A DNS server providing self-sovereign and censorship-resistant domain names. It resolves records hosted on the BitTorrent Mainline DHT
  - ~15m users – difficult to get a number now
- BitTorrent Protocol, created by Bram Cohen in 2001
  - Bram Cohen sold BitTorrent Inc to TRON, a blockchain platform that uses a Delegated Proof of Stake (DPoS) governance system, for ~140m led by controversial figure Justin Sun
- Bram Cohen then founded Chia Network, a other blockchain company focused on proof-of-space-and-time (PoST) (mine with your SSD, and with 1 single core)

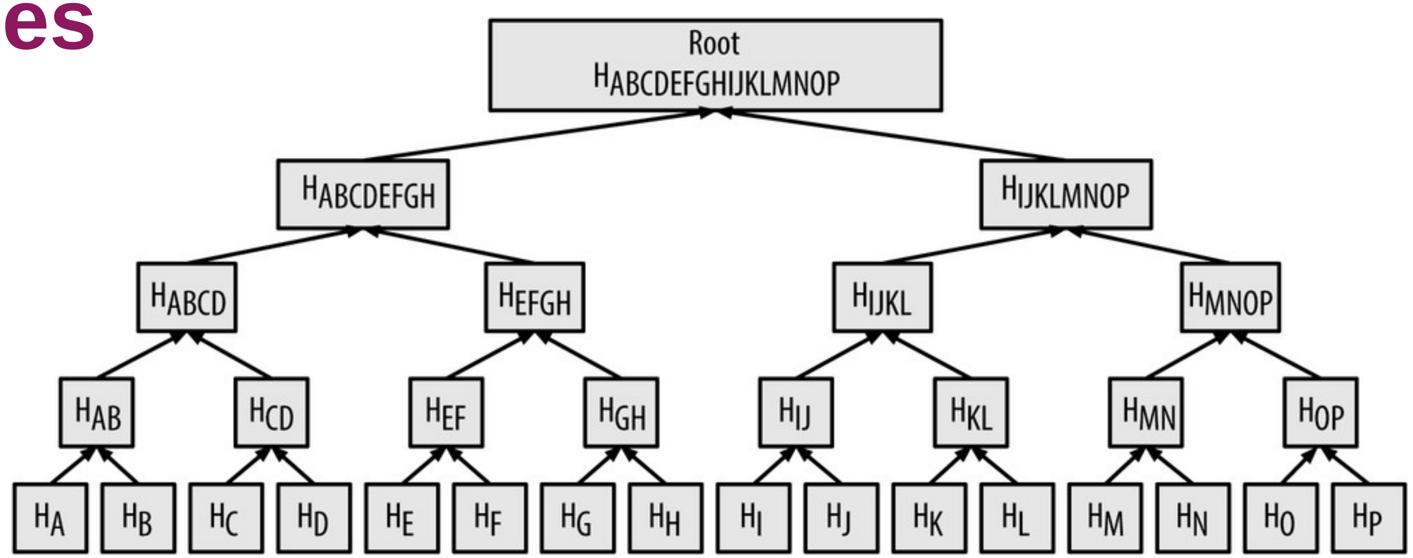
- What is BitTorrent?
  - A peer-to-peer (P2P) file-sharing protocol that enables efficient distribution of large files
  - Breaks files into small pieces for simultaneous downloads from multiple peers
  - Users both download and upload pieces simultaneously, contributing to network efficiency
  - Decentralized architecture reduces server load and bandwidth costs
- Clients: BitComet, DC++, eMule, Filetopia, µTorrent, OnionShare, qBittorrent, Shareaza, Transmission, Tribler, Vuze, WinMX

OST

BitTorrent Key Concepts

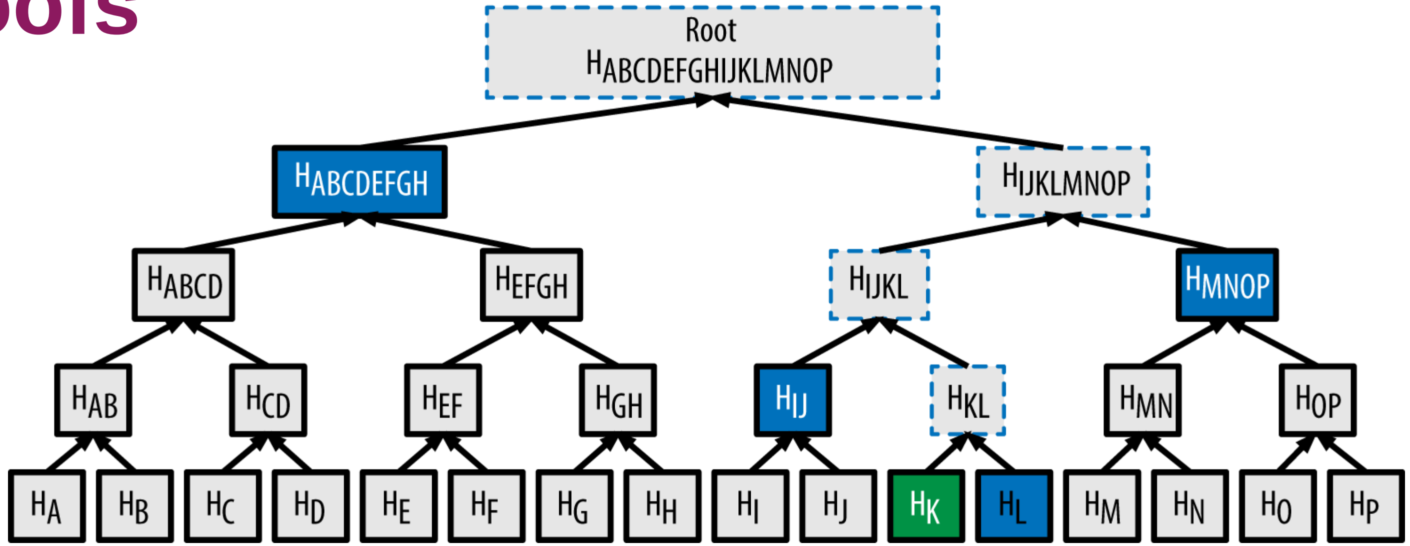
- Core Components:
  - Tracker: Coordinates peers and maintains lists of active users
  - Torrent File: Contains metadata about files and tracker information
  - Seeds: Users with complete file copies
  - Peers/Leechers: Users still downloading pieces
- Many other interesting technical details
  - µTP: Micro Transport Protocol, similar to LEDBAT, Low Extra Delay Background Transport
    - Actively monitors RTT to detect congestion, backing off when other applications need bandwidth
    - "network-friendly" compared to traditional aggressive TCP
  - Bencoding, tit-for-tat, ...
- Many interesting details that we will take a look at
  - Merkle proofs
    - Also used in many blockchains
  - Bloom filters
    - Avoid processing repeated messages, also used e.g., for blockchain light clients
  - DHT / Kademlia
    - Also used in Tor (e.g,onion services) / blockchains (e.g,
    - Polkadot for cross-chain communication, shariding)

Merkle Trees



- A Merkle tree is a binary hash tree containing leaf nodes
- Constructed bottom-up, i.e.,
- Used to summarize all transactions in a block
- To prove that a specific transaction is included in a block, a node only needs to produce hashes, constituting a merkle path connecting the specific transaction to the root of the tree.

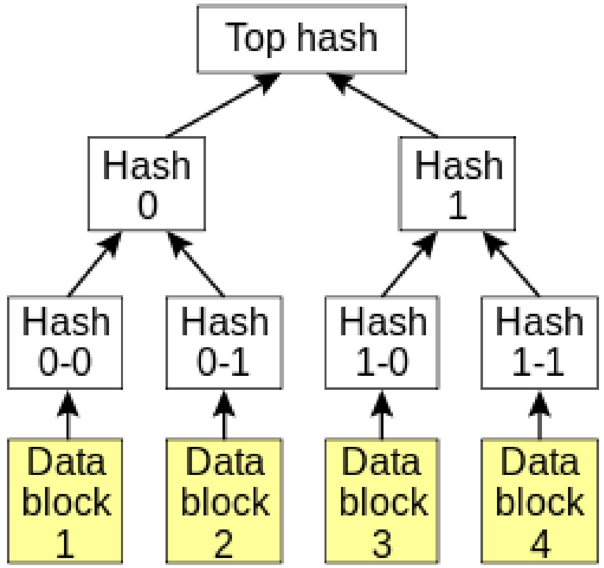
Merkle Proofs



- A node can prove that transaction K is included in the block by producing a merkle path
  - $\boldsymbol{log_2 16} = \boldsymbol{4\ hashes}$ long

BitTorrent: Mechanisms

- Magnet links
  - Magnet is URI scheme, does not point to a centralized tracker
    - No centralized tracker: pointer to DHT
    - General purpose, not only for BT
    - magnet:?xl=1000&dn=song1.mp3&xt=urn:tree:tiger:2A3B…
  - tree:tiger → Hash Tree
    - Tree of hashes (|| → concatenation)
    - hash 0 = hash(hash 0-0 || hash 0-1)
    - hash 1 = hash(hash 1-0 || hash 1-1)
    - Top hash = hash(hash 0 || hash 1)

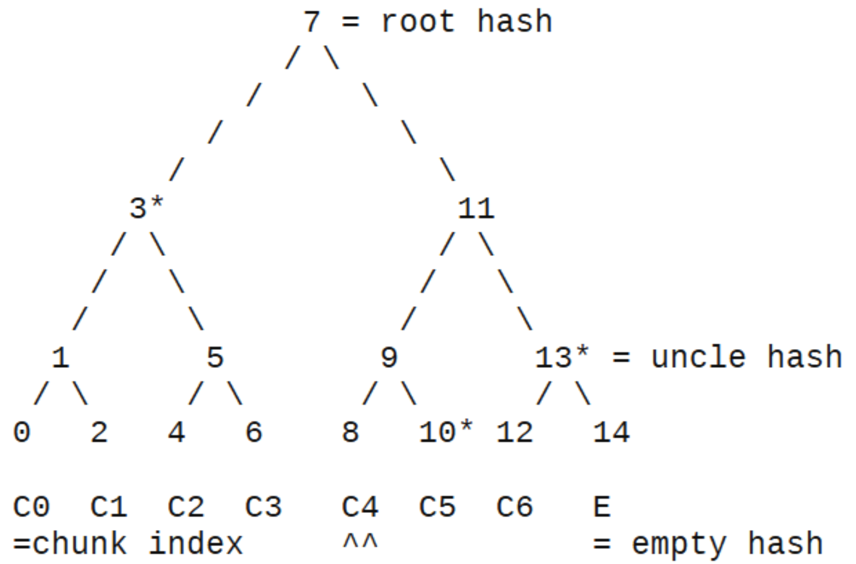


http://en.wikipedia.org/wiki/Hash_tree

BitTorrent: Mechanisms

- Verification
  - Peer A has top hash (root hash)
  - Peer downloads C4 from peer B
    - create hash 8
  - Need hash 10, 13, 3 (uncle hash)
    - Can be from peer B
  - With 8,10,13,3 can create root hash

    → verify this root hash
- Usage: Blockchain, P2P filesharing, git, Amazons Dynamo, ZFS

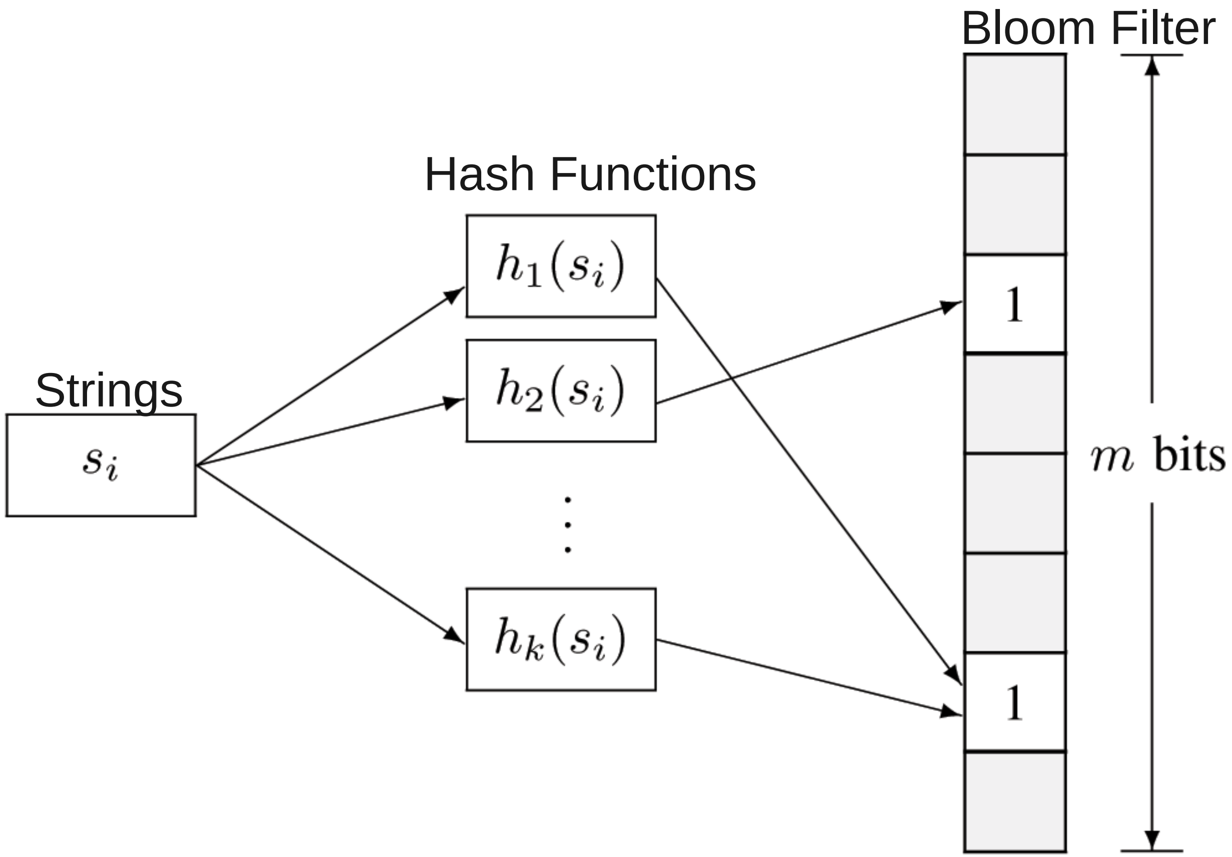


The Merkle hash tree of an interval of width W=8

http://datatracker.ietf.org/doc/draft-ietf-ppsp-peer-protocol/ Section 5.2

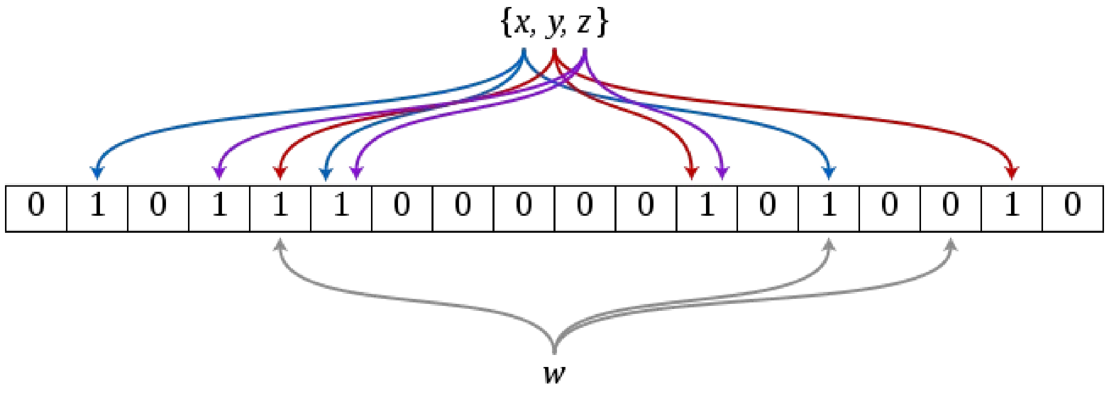
Bloom Filter

- An array of m bits, initially all bits set to 0
- A bloom filter uses k independent hash functions
  - h1, h2, …, hk with range {1, …, m}
- Each input is hashed with every hash function
  - Set the corresponding bits in the vector
- Operations
  - Insertion
    - The bit A[hi(x)] for 1 < i < k are set to 1
  - Query
    - Yes if all of the bits A[hi(x)] are 1, no otherwise
  - Deletion
    - Removing an element from this simple Bloom filter is impossible



Query of an Element, m=18, k=3

- Insert x, y, z
- Query w



http://en.wikipedia.org/wiki/Bloom_filter

- Example for False-positives
  - Insertions
    - Hash („color printer“) => (1,4,6)
    - Hash („digital camera“) => (3,4,5)
    - Bloom filter (1,3,4,5,6)
  - Query
    - Hash („heat sensor“) => (3,4,6)
    - Matches since bits 3,4,6 are all set to 1
  - Online
- False-negative
  - Query
    - Hash (“color printer”) => (1,4,6) , matches (1,3,4,5,6) → no false-negative

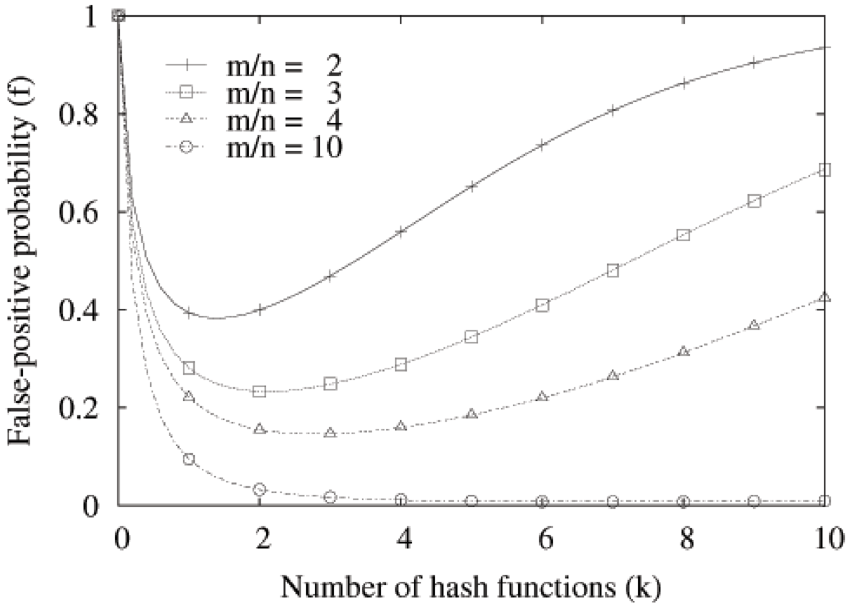
OST

Properties

- Space Efficiency
  - Any Bloom filter can represent the entire universe of elements
    - In this case, all bits are 1
- No Space Constraints
  - Add never fails
  - But false positive rate increases steadily as elements are added
- Simple Operations
  - Union of Bloom filters: bitwise OR
  - Intersection of Bloom filters: bitwise AND

- No false negative, but false positive
- False-positive probability:
  - *n* number of strings; *k* hash functions; *m*-bit vector

$$f = (1 - e^{-\frac{nk}{m}})^k$$



=> Given m/n, there is an optimal number of hash functions (opt. k = m/n ln 2, or k=-$\log_2$(f) (when 50% of the bits are set)

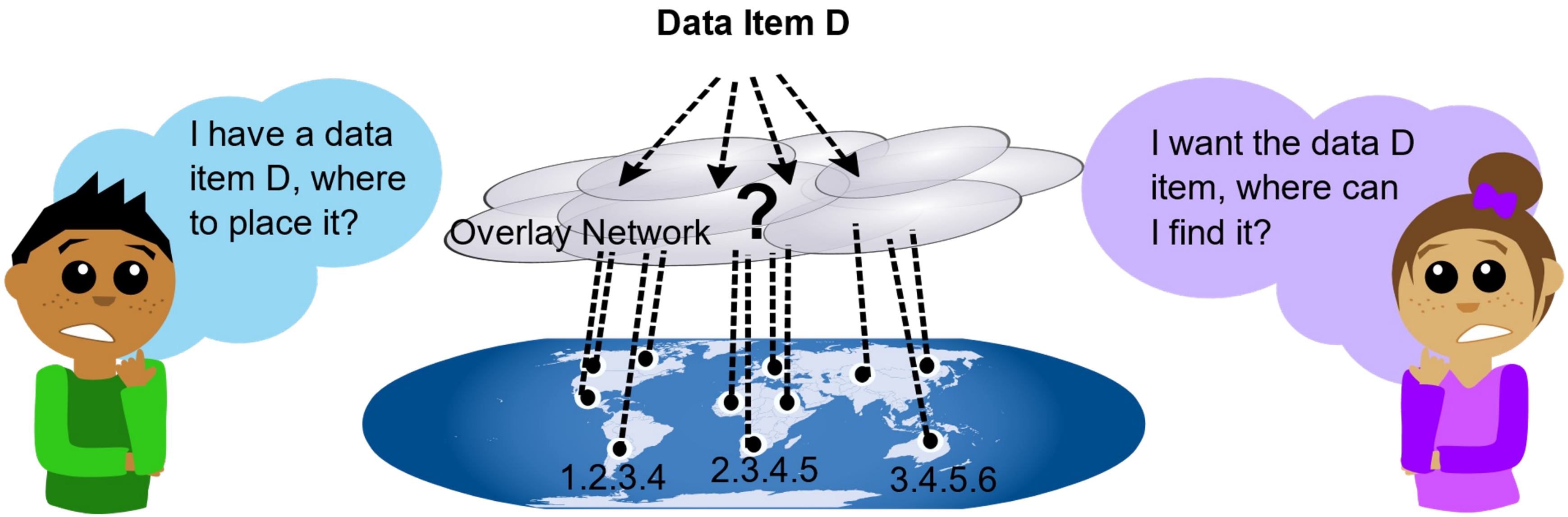
Bloom Filter Variants

- Compressed Bloom Filters
  - When the filter is intended to be passed as a message
  - False-positive rate is optimized for the compressed bloom filter (uncompressed bit vector m will be larger but sparser)
  - However, compression/decompression, more memory
- Generalized Bloom Filter
  - Two type of hash functions gi (reset bits to 0) and hj (set bits to 1)
  - Start with an arbitrary vector (bits can be either 0 or 1)
  - In case of collisions between gi and hj, bit is reset to 0
  - Store more info with low false positive
  - Produces either false positives or false negatives
- Counting Bloom Filters
  - Entry in the filter not be a single bit but a counter
  - Delete operation possible (decrementing counter)
- Scalable Bloom Filter
  - Adapt dynamically to number of elements, consist of regular Bloom filters
  - "A SBF is made up of a series of one or more (plain) Bloom Filters; when filters get full due to the limit on the fill ratio, a new one is added; querying is made by testing for the presence in each filter"
- Others, e.g., Cuckoo filter, VQF
- Usage: e.g., fast search at LinkedIn

OST

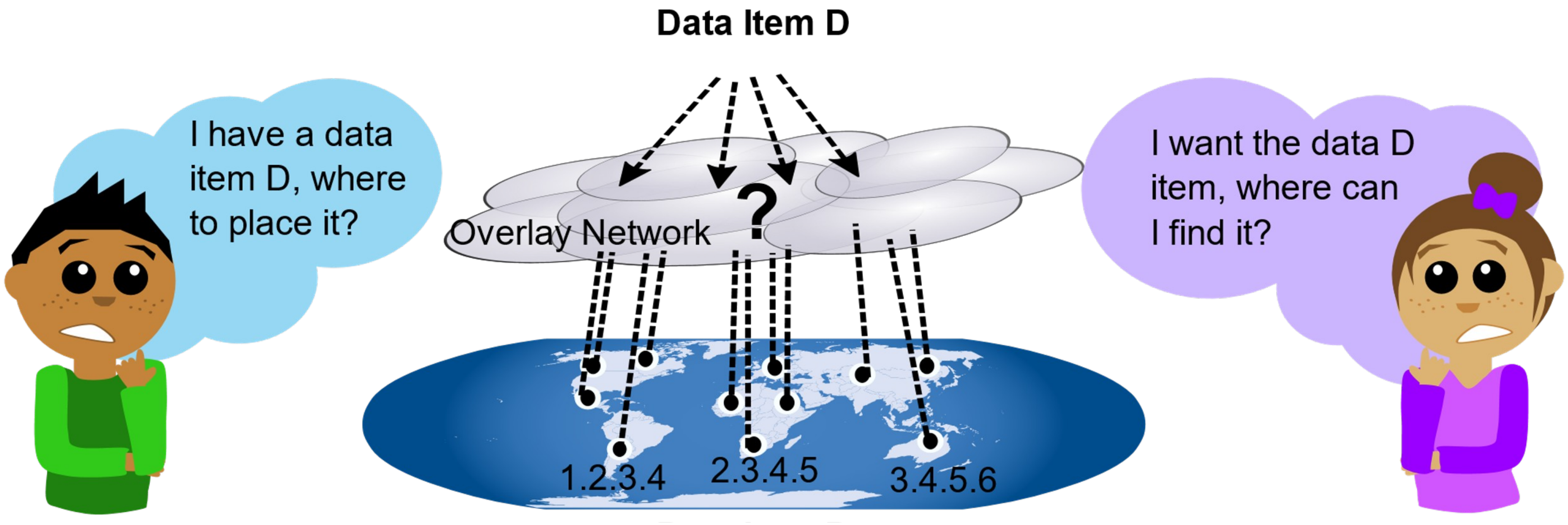
DHT / Kademlia

- Essential challenge in (most) distributed / P2P systems?
  - Location of a data item among systems distributed
    - Where shall the item be stored?
    - How can the item be found?
  - Scalability: keep the complexity for communication and storage scalable
  - Robustness and resilience in case of faults and frequent changes



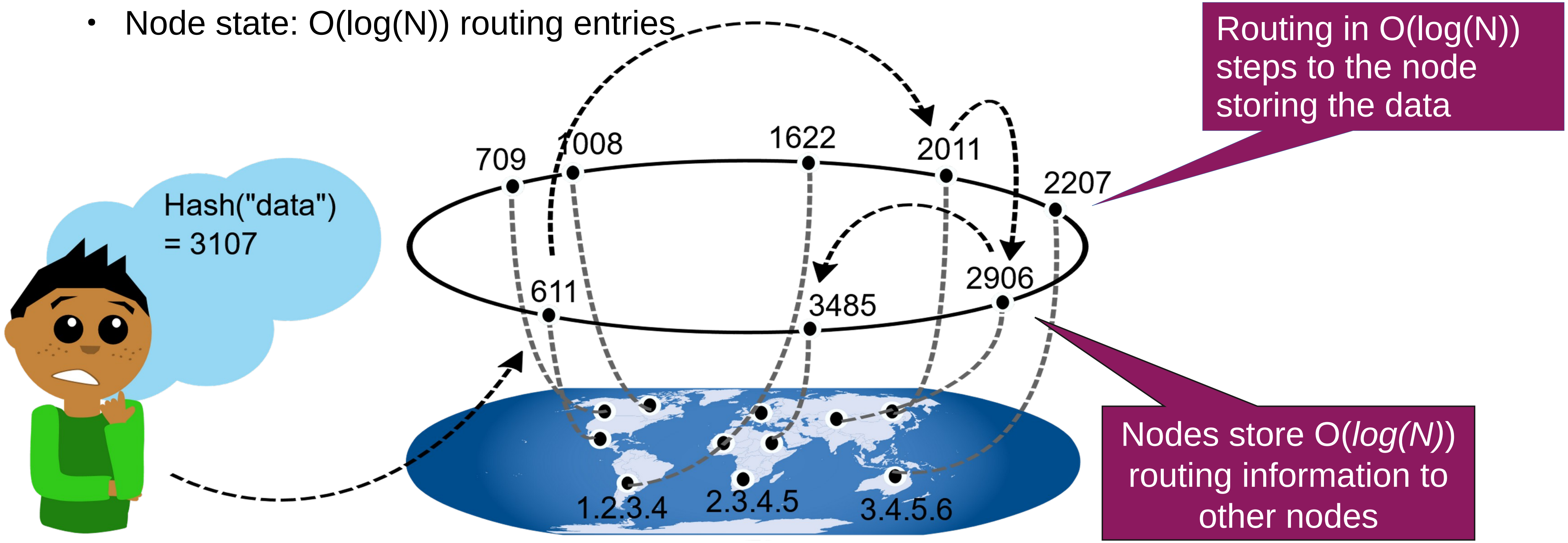
Comparison of Strategies for Data Retrieval

- Strategies to store and retrieve data items in distributed systems
  - Central server (e.g., service registry, reverse proxy - although main use case is load balancing)
  - Flooding search (e.g., layer 2 broadcasting, wireless mesh networks, Bitcoin)
  - Structured indexing (Tor, Bittorrent, IPFS, Apache Cassandra)



Structured Indexing (1)

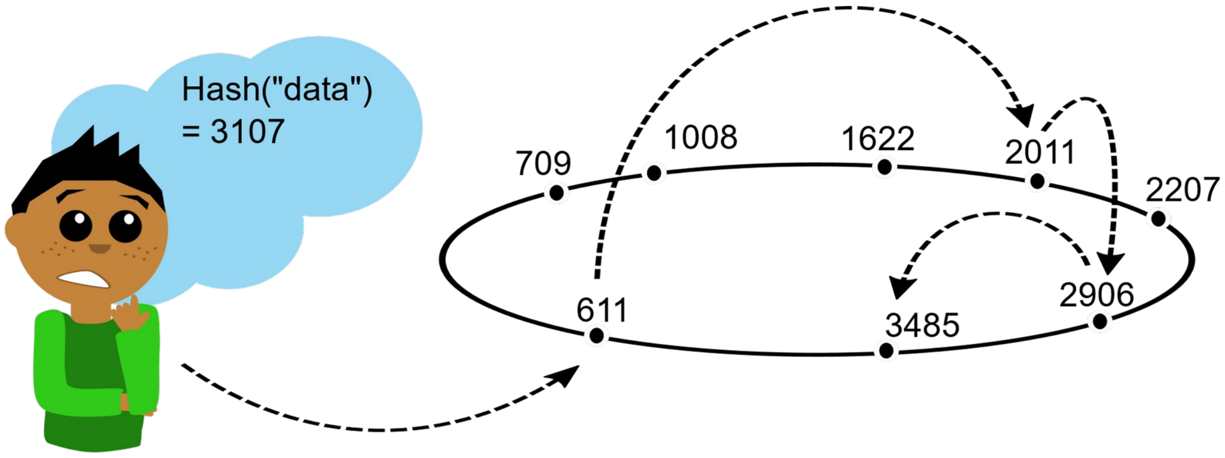
- Goal is scalable complexity for
  - Communication effort: O(log(N)) hops
  - Node state: O(log(N)) routing entries



OST

Structured Indexing (2)

- Approach of structured indexing schemes
  - Data and nodes are mapped into same address space
  - Nodes maintain routing information to other nodes
    - Definitive statement of existence of content
- Problems
  - Maintenance of routing information required
  - Overlay/Underlay



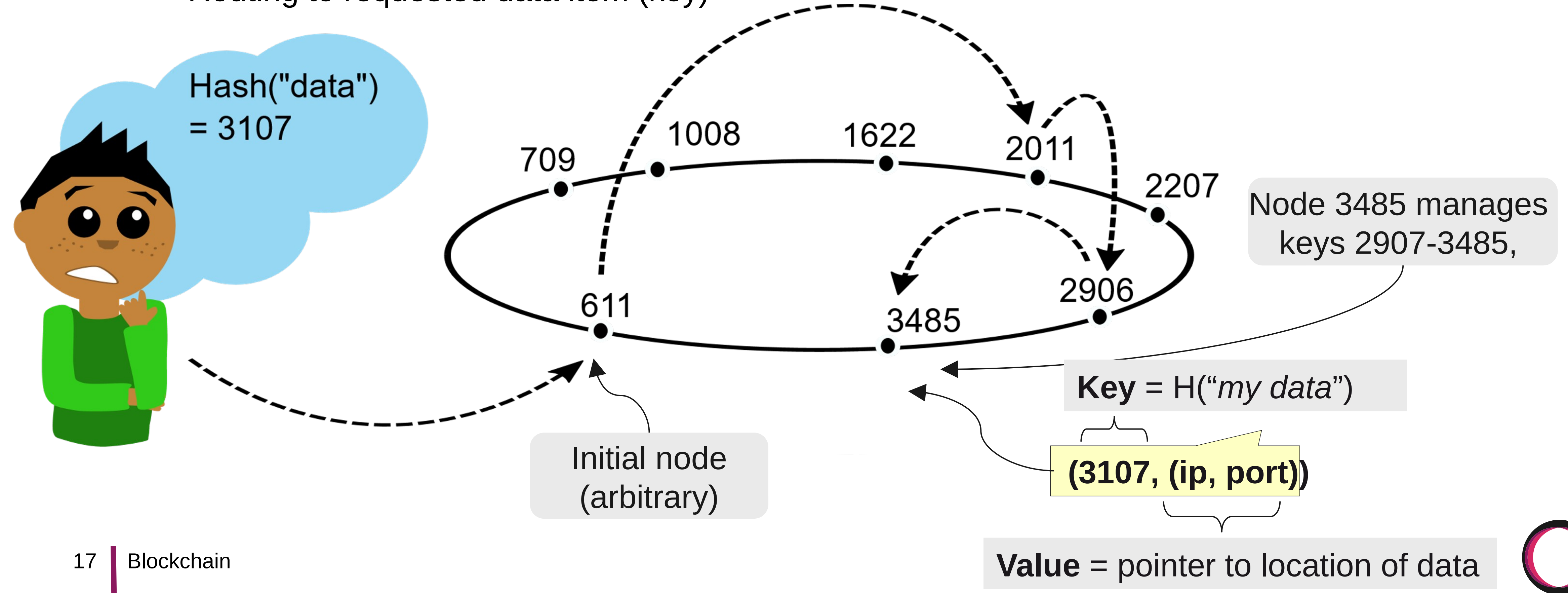
Fundamentals of Distributed Hash Tables

- Challenges for designing DHTs
  - Desired Characteristics
    - Reliability / Scalability
  - Equal distribution of content among nodes
    - Crucial for efficient lookup of content
  - Permanent adaptation to faults, joins, exits of nodes
    - Assignment of responsibilities to new nodes
    - Re-assignment and re-distribution of responsibilities in case of node failure or departure

- Distributed Hash Table
  - Consistent hashing → nodes responsible for hash value intervals
  - More peers = smaller responsible intervals
- Hash Table [link]
  - Modulo hashing
    - Bucket = hash(x) mod n
  - If n changes, remapping / bucket changes
  - N changes if capacity is reached
  - Remapping is expensive in DHT!
    - DHTs reassign responsibility

Routing to a Data Item

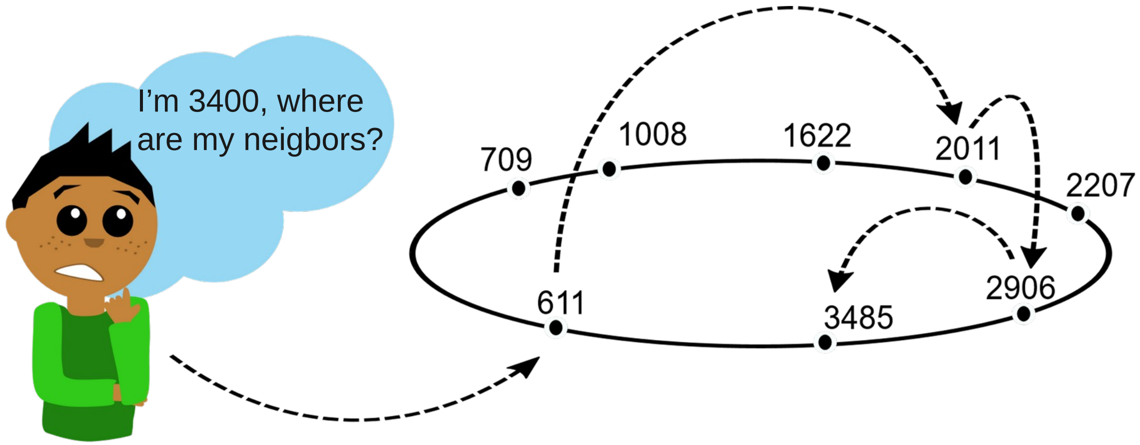
- Locating the data / Routing to a K/V-pair
  - Start lookup at arbitrary node of DHT
  - Routing to requested data item (key)



Join/Leave

- Joining of a new node
  1) Calculation of node ID (normally random / or based on PK)
  2) New node contacts DHT via arbitrary node (bootstrap node)
  3) Lookup of its node ID (routing)
  4) Copying of K/V-pairs of hash range (in case of replication)
  5) Notify neighbors
- Failure of a node
  - Use of redundant K/V pairs (if a node fails)
  - Use of redundant / alternative routing paths
  - Key-value usually still retrievable if at least one copy remains

- Departure of a node
  - Copying of K/V pairs to corresponding nodes
    - Can be before or after unbinding
  - Friendly unbinding from routing environment
    - If unbinding is unfriendly, need for keep-alive messages



Kademlia

- Several approaches to build DHT
  - Distance metric as key difference
    - Chord, Pastry: numerical closeness
    - CAN: multidimensional numerical closeness
    - Kademlia: XOR metric
- Kademlia designed in 2002 by Maymounkov and Mazières
  - Many implementations, application specific
    - BitTorrent (tracker), IPFS, Tor Onion Services
- Parallel queries
  - For one query, α (alpha) concurrent lookups are sent
  - More traffic load, but lower response times
- Preference towards old contacts
  - Study has shown that the longer a node has been up, the more likely it is to remain up another hour
  - Resistance against DoS attacks by flooding the network with new nodes
- Network maintenance
  - In Chord: active fixing of fingers
  - In Kademlia: active maintenance
- DHT-based overlay network using the XOR distance metric
  - Symmetrical routing paths (A → B == B → A)
    - due to XOR(A,B) == XOR (B,A)

Construction of Routing Table

- Each Kademlia node and data item has unique identifier
  - 160 bit (SHA-1)
  - Nodes: Node ID (160bit)
    - Can be calculated from IP address or public key, and data item using secure hash function, or just random
  - Data items: Keys (160bit), hash of data item
- Keys are located on the node whose node ID is closest to the key
  - Knows neighbors well, further nodes not that much
  - Kademlia: 160 buckets with size 20 (8)
  - If distance can be represented in m bits, bucket m will be used

XOR Distance Calculation:

```
ID Node A: 110101
ID Node B: 010001
```

$d_{XOR}(A,B) = d(110101, 010001)$

```
1 1 0 1 0 1
    XOR
0 1 0 0 0 1
      ↓
1 0 0 1 0 0
```

$d_{XOR}(A,B) = 100100_2 = 36_{10}$

Kademlia Example

- 2^3, max size 8, #6 searches for 3

1	2	3
7	4 (or 5)	0 (or 1, 2)

Routing Table of #6
6 xor 3 = 101b

- Neighbors of 6, if k=1

1	2	3
1	2	4 (or 5, 6, 7)

Routing Table of #0
0 xor 3 = 11b

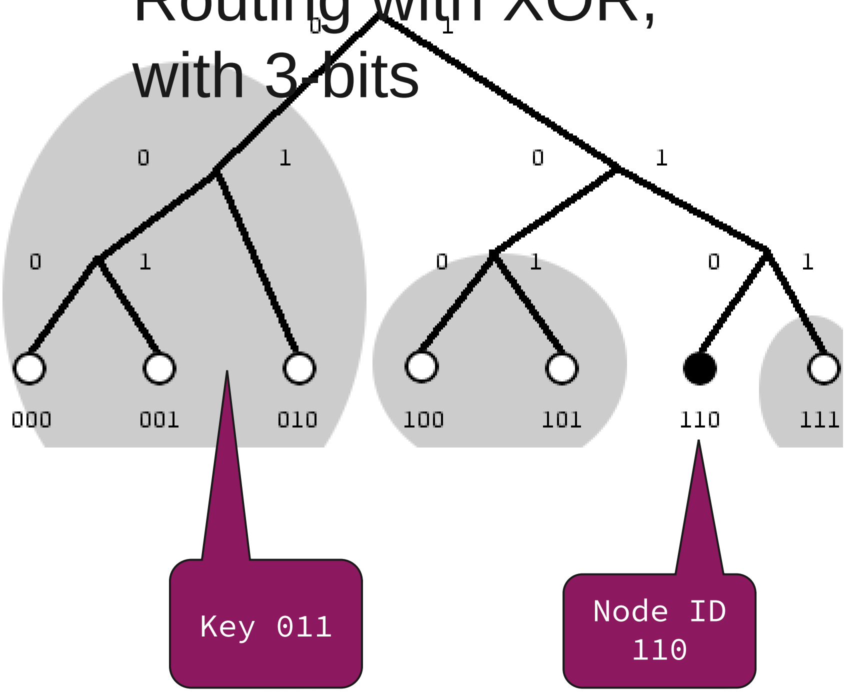
- Search for 3, ask 0, neighbors of 0
- Ask 2, neighbors of 2

1	2	3
-	0 (or 1)	4 (or 5, 6, 7)

Routing Table of #2
2 xor 3 = 1b

- Ask 2, 2 replies 0. 6 figures that there is no closer node, 2 is the closest one (2 xor 3 =1)

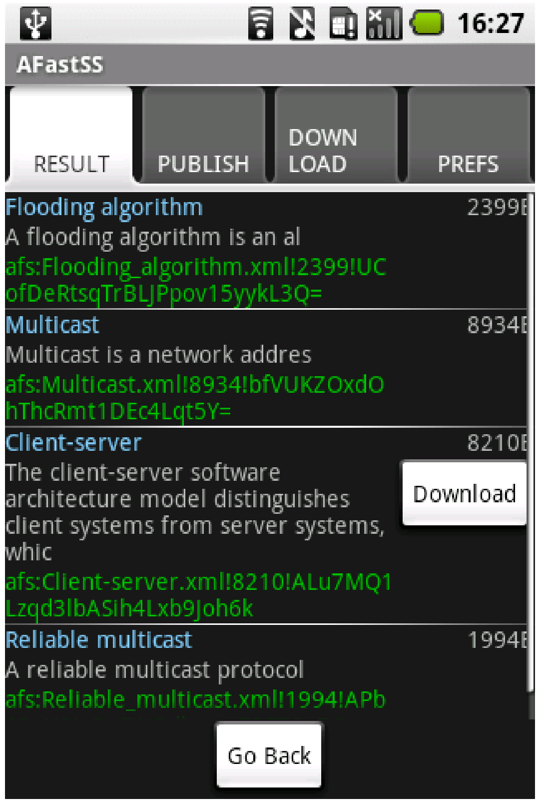
Routing with XOR, with 3 bits

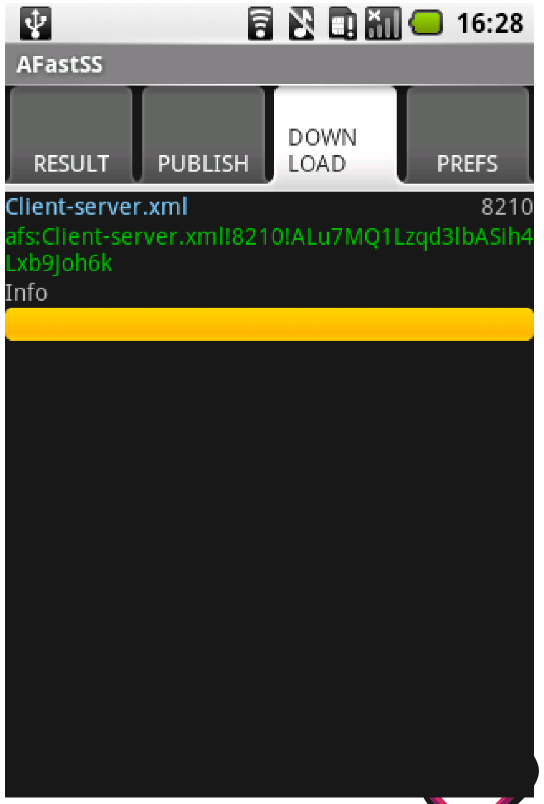


OST

TomP2P

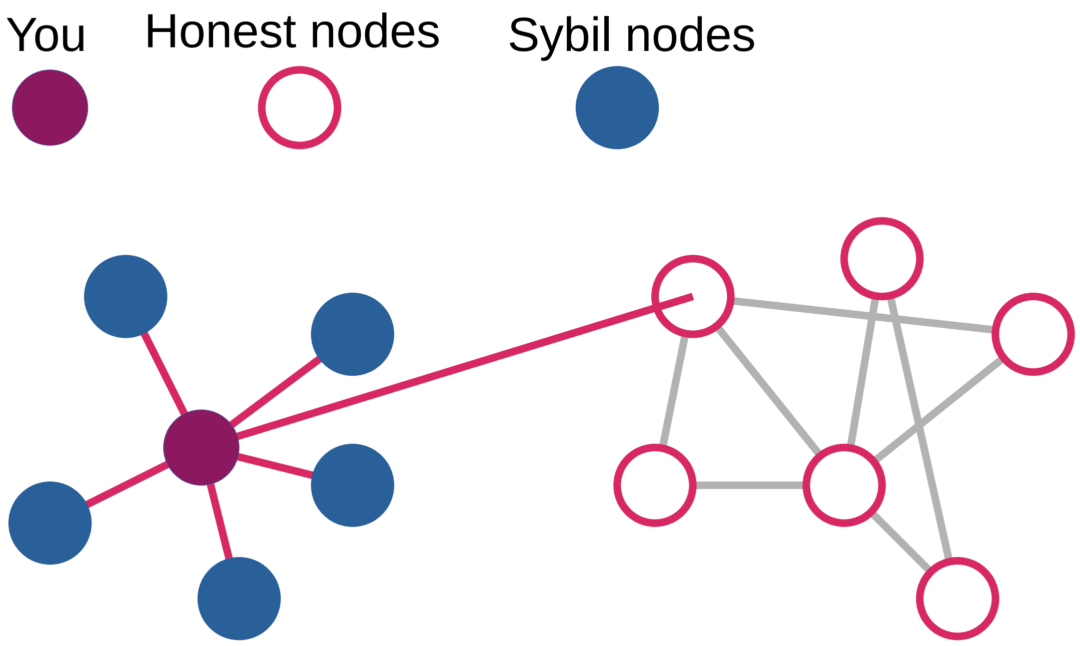
- TomP2P is a P2P framework/library
  - Unmaintained ☹️
  - Implements DHT (structured), broadcasts ([un]structured), direct messages (can implement super-peers)
  - NAT handling: UPNP, NATPMP, relays, hole punching (work in progress)
  - Direct / indirect (tracker / mesh) storage
  - Direct / indirect replication (churn prediction and ~rsync)
- Yes, this is the first Android device, HTC Dream, Sept. 2009





Fully Decentralized Systems

- Always consider Sybil attacks
  - TomP2P, BitTorrent, etc.
    - Data can always disappear
  - Know when data changed



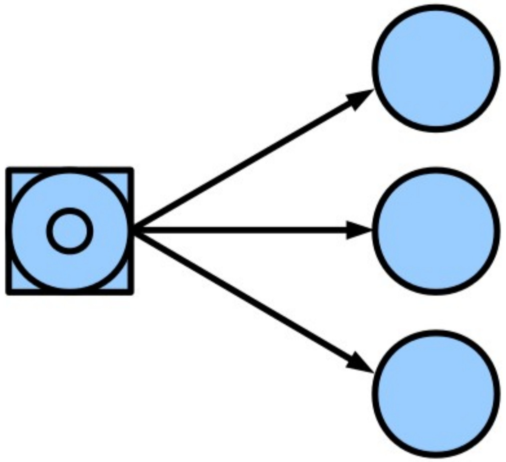
- Sybil attack
  - Create large number of identities
  - Larger than honest nodes
    - Control “close” nodes in a DHT
    - Isolate nodes
- Prevention [source]
  - Creation of identities costs money
  - Always assume data from other nodes may be missing
    - Bitcoin – chain of block, if block is missing, you notice
  - Chain of trust / reputation

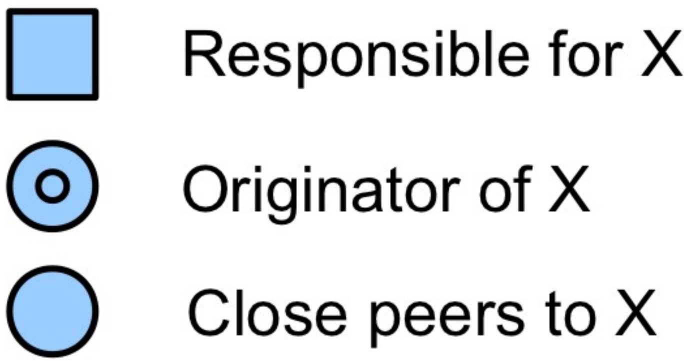
Attacking the DHT

- Example
- Create a key for a data item close to the target:
  Number160.createHash(data).xor(new Number160(0)) – distance 0, perfect match
  Number160.createHash(data).xor(new Number160(1)) – distance 1
  Number160.createHash(data).xor(new Number160(2)) – distance 2
  …
- Or create key of node close to the target
  new PeerBuilder(new Number160(RND)).ports(port).start(), where RND is
  Number160.createHash(data).xor(new Number160(0))
  Number160.createHash(data).xor(new Number160(1))
  …
- Peer can then answer there is no data
- For previously known values / peers (known public key)
  - Cannot change data, but make it disappear

Redundancy in DHTs

- Replication
  - Enough replicas
  - Direct replication
    - Originator peer is responsible
    - Periodically refresh replicas
    - Example: tracker that announces its data
- Problem
  - Originator offline → replicas disappear. Content has TTL

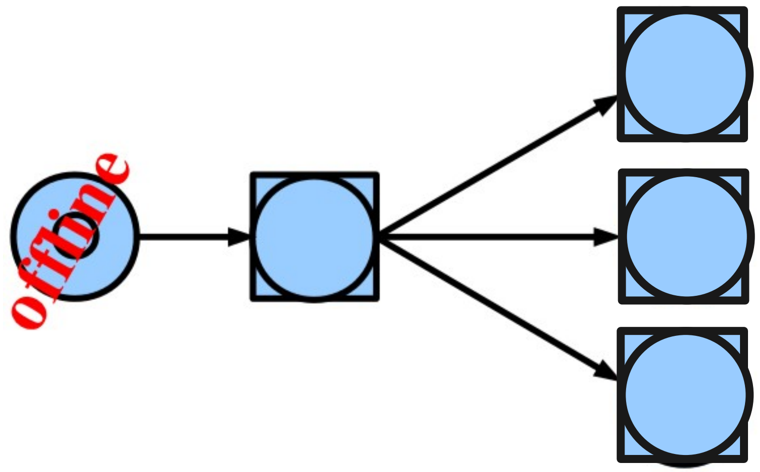




OST

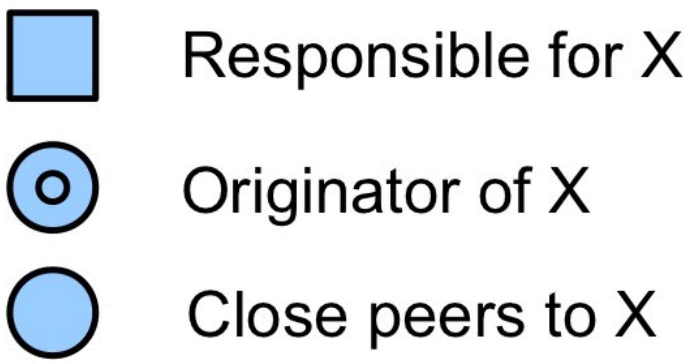
Redundancy in DHTs

- Indirect Replication
  - The closest peer is responsible, originator may go offline vs any close peers are responsible
    - Periodically checks if enough replicas exist
    - Detects if responsibility changes

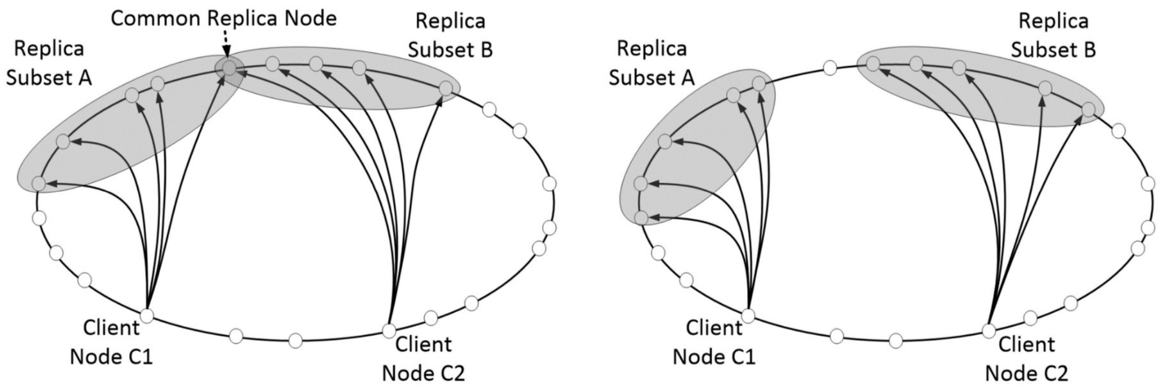


closest vs any

- Problem
  - Requires cooperation between responsible peer and originator
  - Multiple peers may think they are responsible for different versions → eventually solved



Replication and Consistency



- DHTs have weak consistency
  - Peer A put X.1
  - Peer B gets X.1
  - Peer B modifies it puts B.2
- Same time (time in distributed systems):
  - Peer C gets X.1
  - Peer C modifies it puts C.2
- Replication makes it worse
  - Consistency: generic issue in distributed systems, requires typically coordinator
- Multi-Paxos, Raft, ZooKeeper → Leader Election

- vDHT: CoW, versions, 2PC, replication, software transactional memory (STM) → for consistent updates. Works for light churn
  - No locking, no timestamps (replication time may have an influence)
  - Every update – new version
    - get latest version, check if all replica peers have latest version, if not wait and try again
    - put prepared with data and short TTL, if status is OK on all replica peers, go ahead, otherwise, remove the data and go to step 1.
    - put confirmed, don't send the data, just remove the prepared flag
  - Leader is the originator
  - In case of heavy churn, API user needs to resolve