



OST

Eastern Switzerland
University of Applied Sciences

Blockchain (BICh)

Fully Distributed Systems: Introduction and Algorithms

Thomas Bocek

30.10.2022

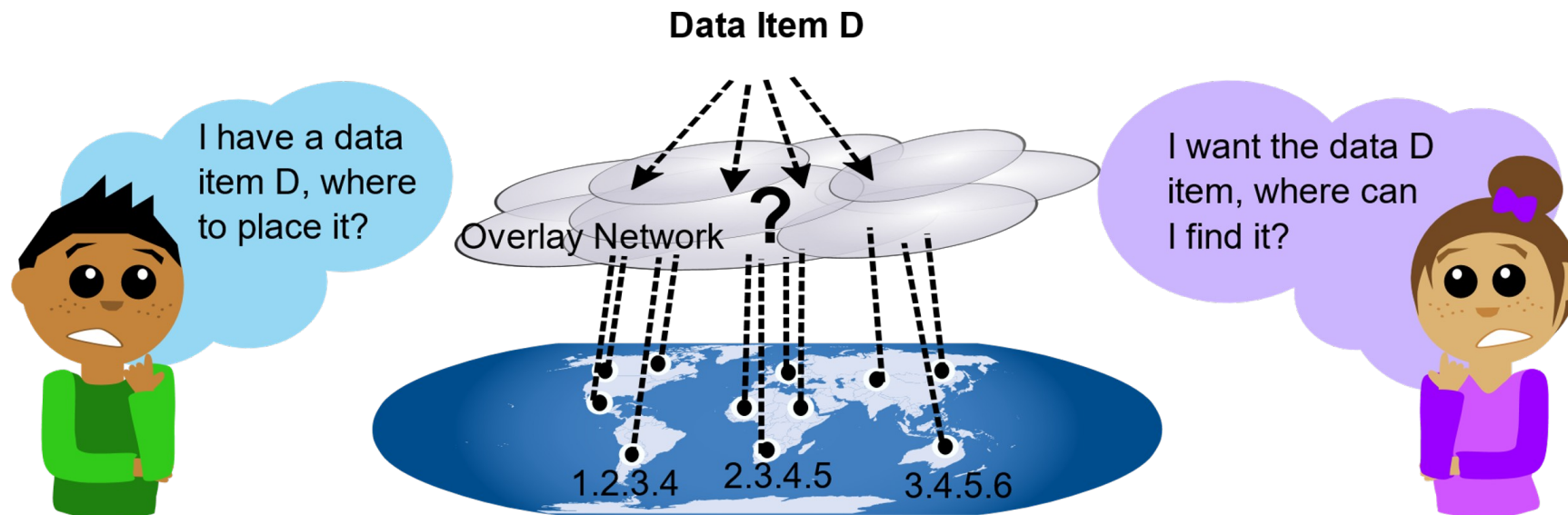
Recall Transparency Principles (DSy, FS22)

- Distributed system should hide its distributed nature
 - Location transparency – users should not be aware of the physical location
 - Access transparency - users should access resources in a single, uniform way
 - Replication transparency – users should not be aware about replicas, it should appear as a single resource
 - Concurrent transparency – users should not be aware of other users
 - ...
- More/other transparencies [here](#), [here](#), [here](#)



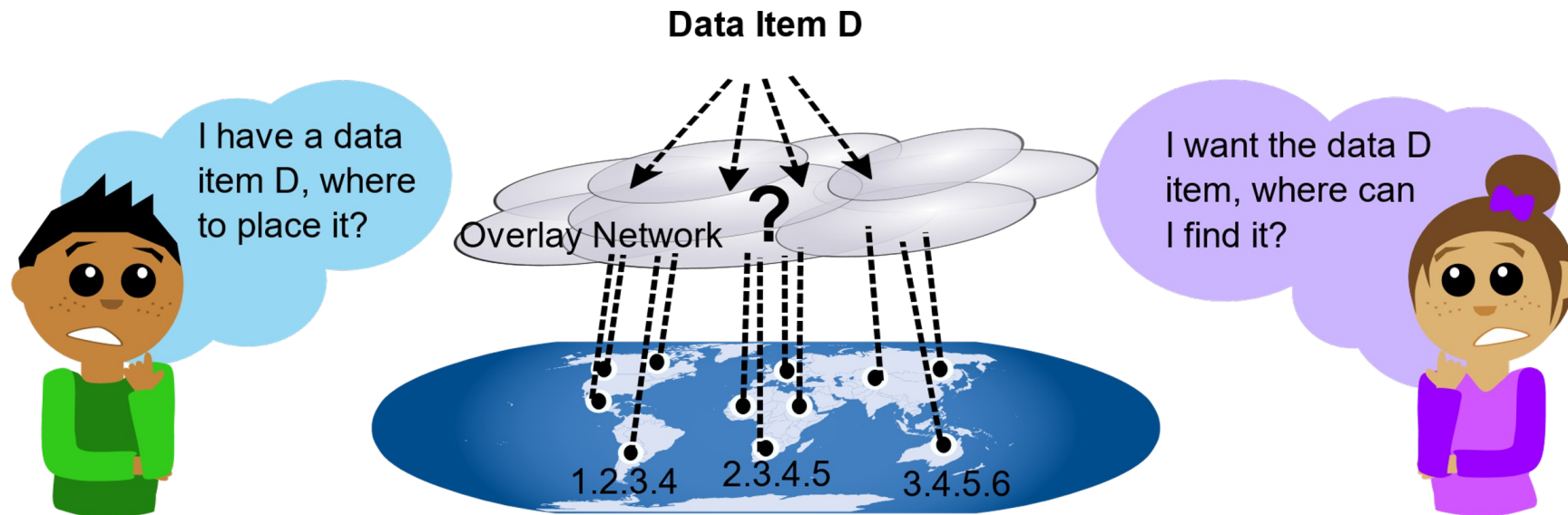
Distributed Management and Retrieval of Data

- 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



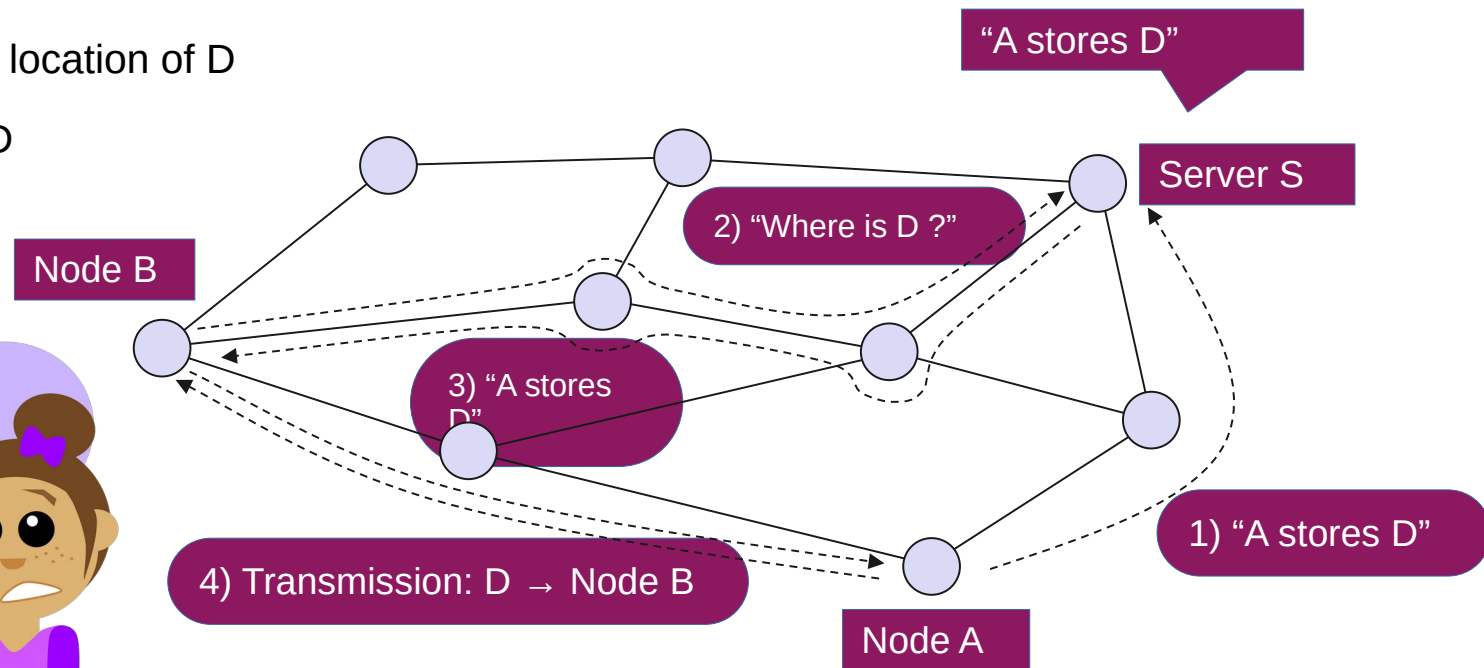
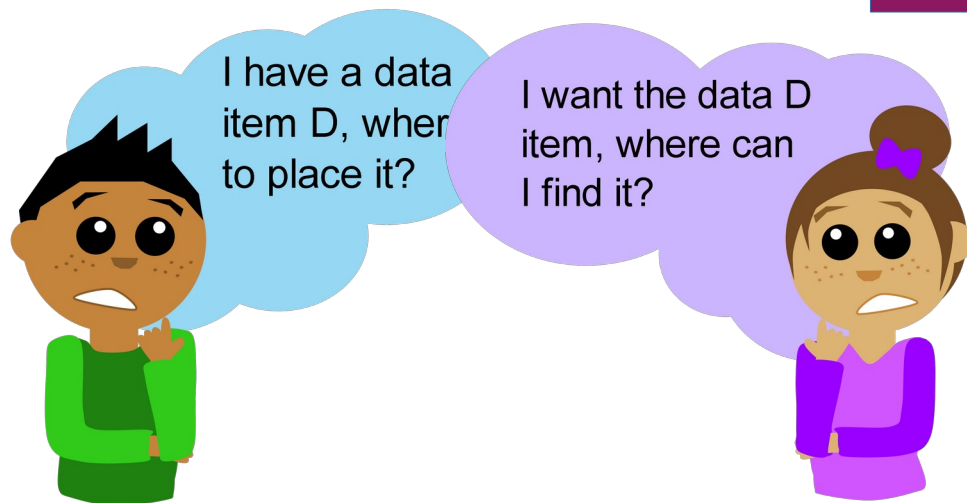
Comparison of Strategies for Data Retrieval

- 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**)
 - Distributed indexing (**Tor**, **Bittorrent**, **IPFS**, **Apache Cassandra**, **Dynamo**, **Atek** [discontinued])



Central Server

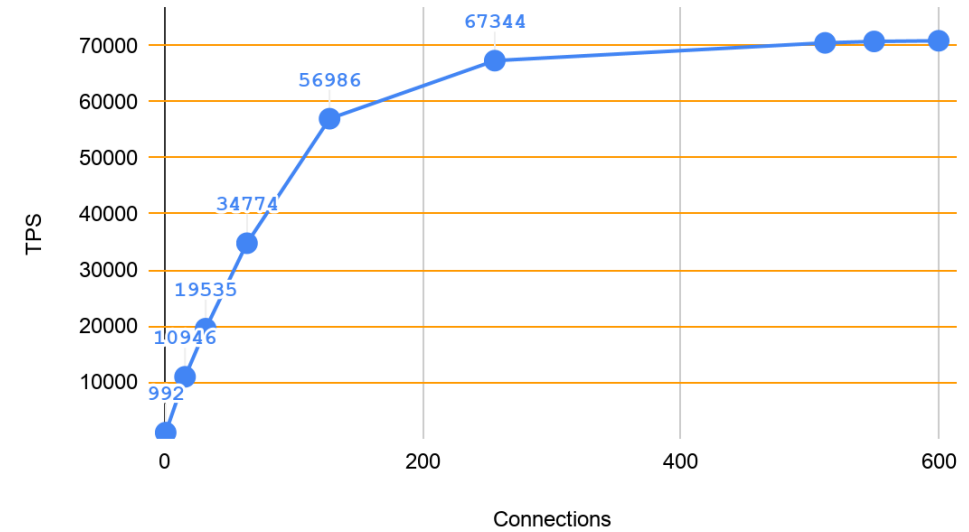
- Simple strategy: Central Server (can be powerful – vertical scaling!)
 - Server stores information about locations
 - 1) Node A (provider) tells server that it stores item D
 - 2) Node B (requester) asks server S for the location of D
 - 3) Server S tells B that node A stores item D
 - 4) Node B requests item D from node A



Approach I: Central Server

- Advantages
 - Search complexity of $O(1)$ – “just ask the server”
 - Complex and fuzzy queries are possible
 - Simple and fast
- Problems
 - No Scalability
 - $O(N)$ node state in server
 - $O(N)$ network and system load of server
 - Single point of failure or attack
 - (Single) central server not suitable for systems with massive numbers of users
- But overall, ...
 - Best principle for small and simple applications!

**EDB Advanced Server 12 (Redwood mode)
TPS vs. Connections**

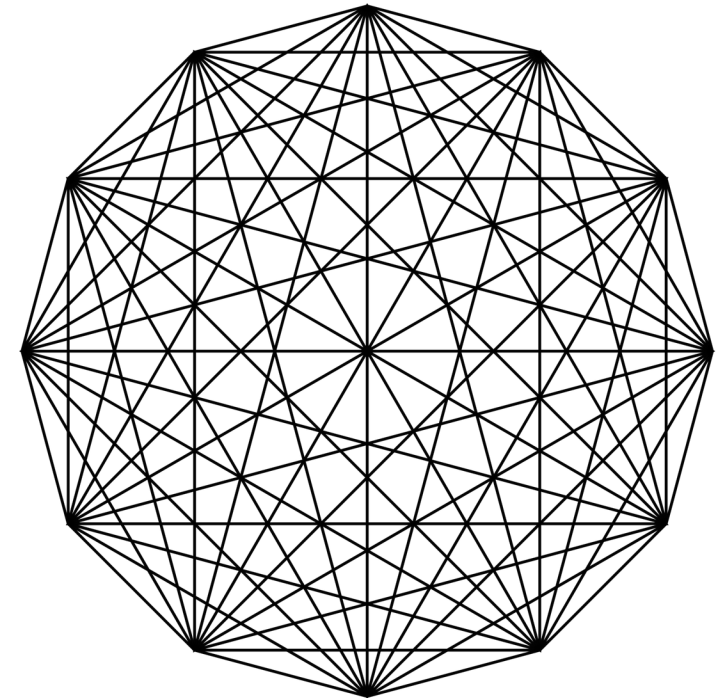


<https://www.enterprisedb.com/blog/pgbench-performance-benchmark-postgresql-12-and-edb-advanced-server-12>

- AMD EPYC: 48 core, 384GB RAM, 4xNVM SSD
- ~70k TPS

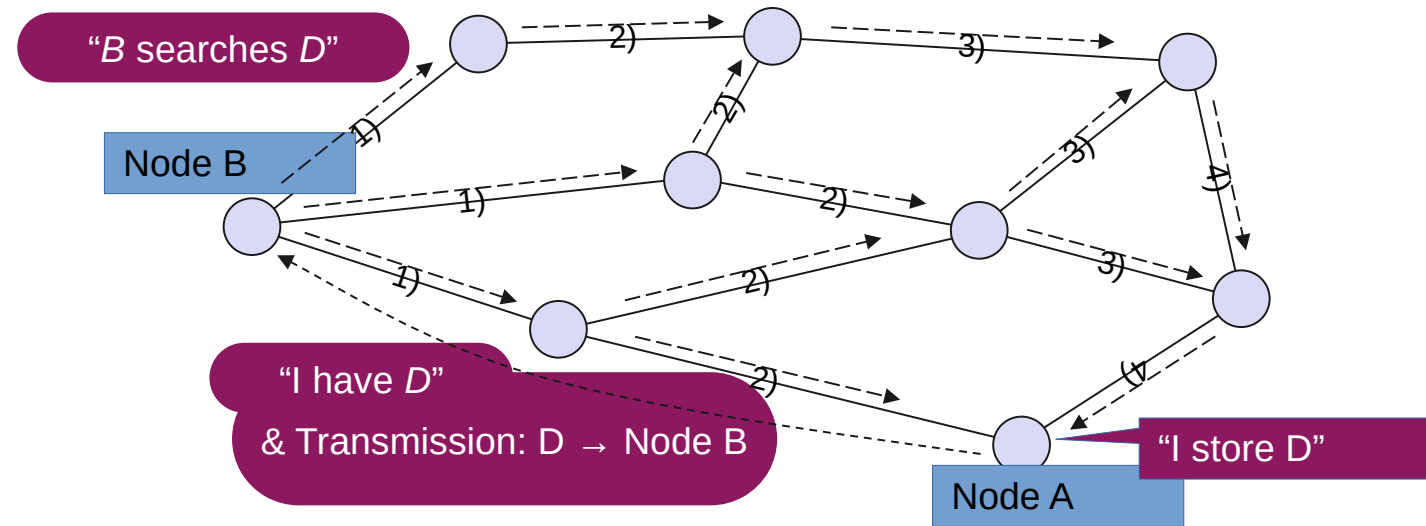
Approach II: Flooding

- Fully-distributed Approach
 - Opposite approach of approach I
 - No information on location of a content
- Retrieval of data
 - No routing information for content
 - Necessity to ask as much systems as possible / necessary
 - Approaches
 - Highest degree search: quick search through large areas
 - Random walk
 - Flooding: high traffic load on network, scalability issues (mechanism required to stop spamming, e.g. TX fee)
 - No guarantee to reach all nodes

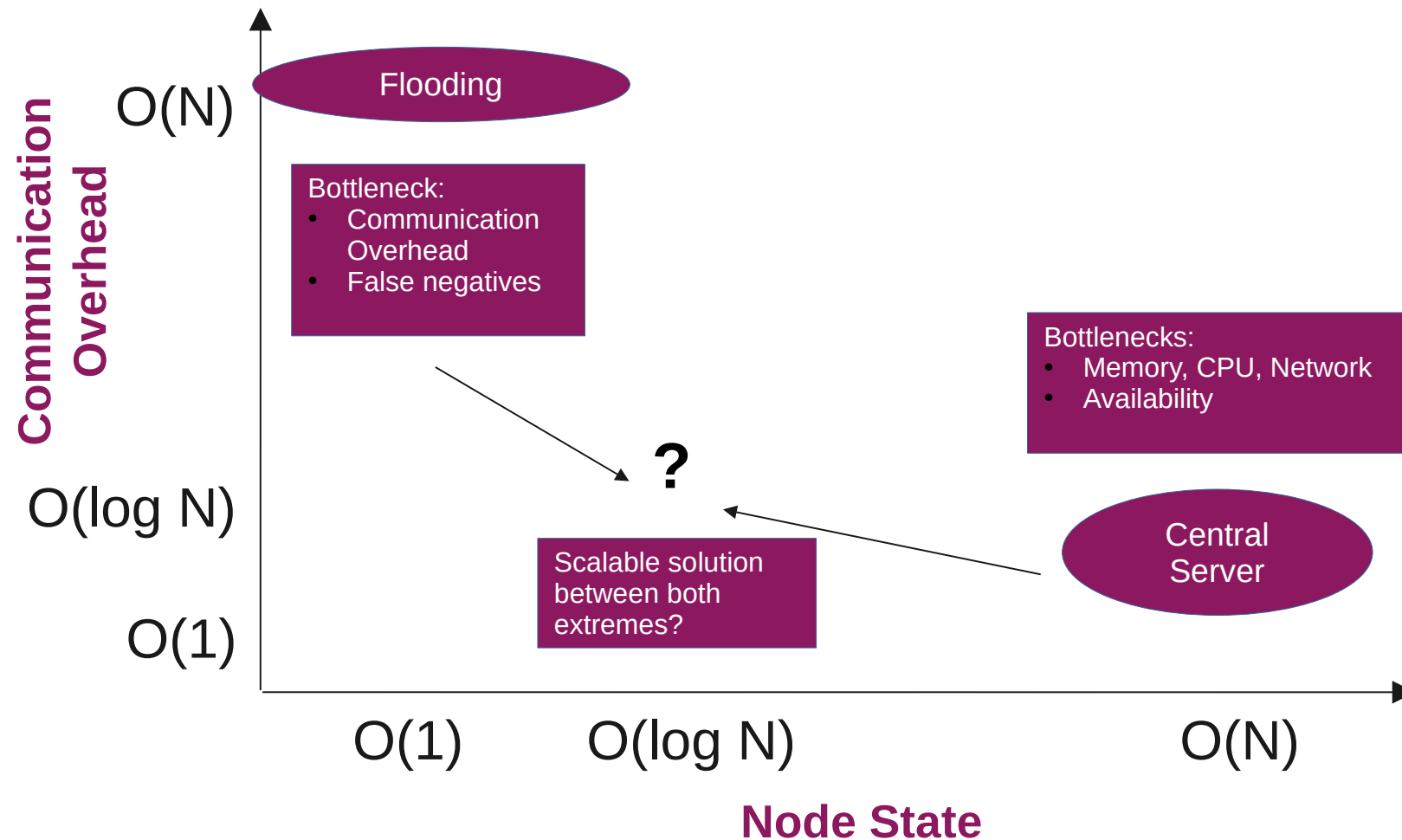


Approach II: Flooding Search

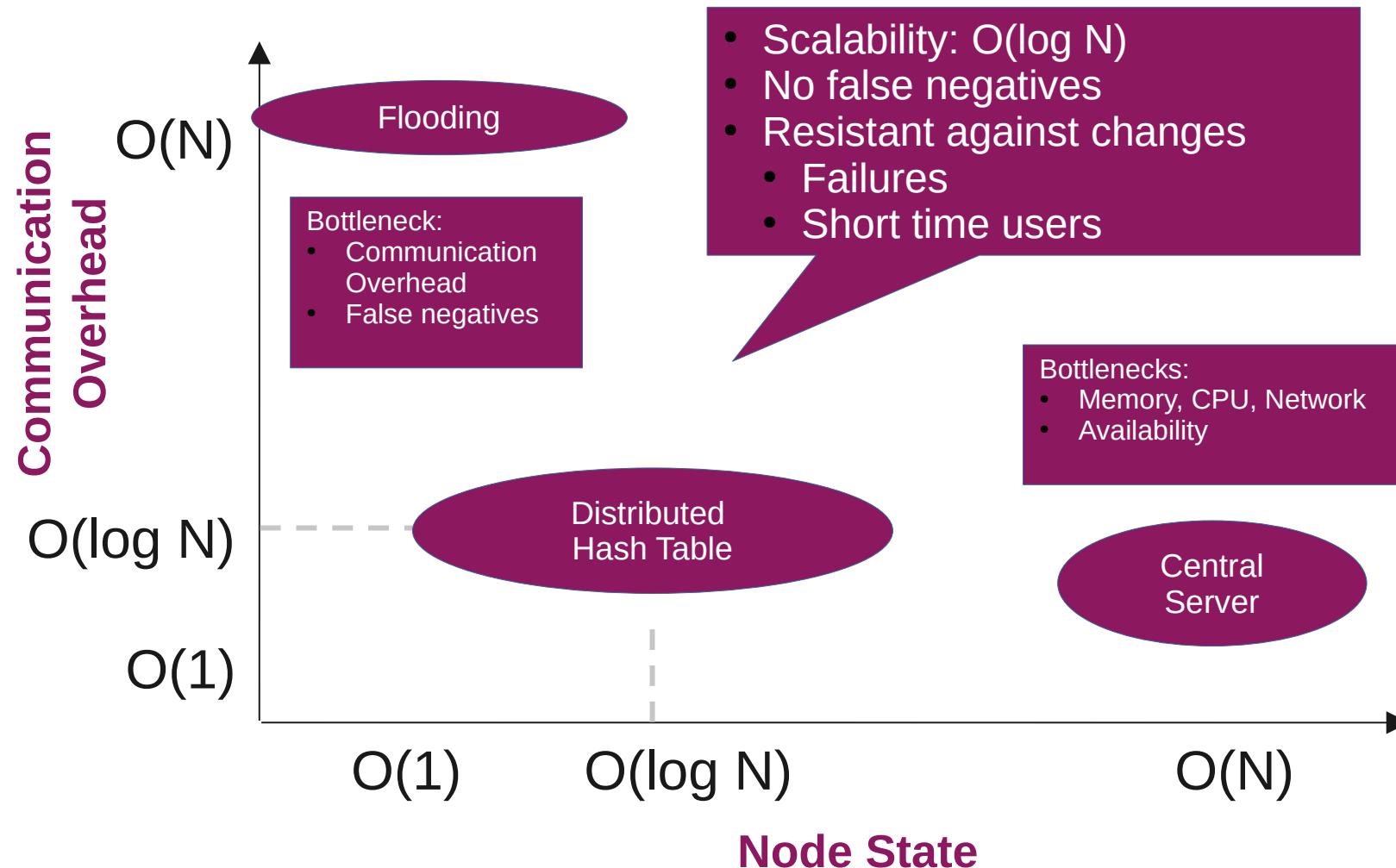
- Fully Decentralized Approach: Flooding Search
 - No information about location of data in the intermediate systems
 - Flood with search term, or flood all the data (Bitcoin)
 - Flood all the data: search local
- 1) Node B (requester) asks neighboring nodes for item D
- 2-4) Nodes forward request to further nodes (breadth-first search / flooding)
- 5) Node A (provider of item D) sends D to requesting node B



Motivation Distributed Indexing (1)

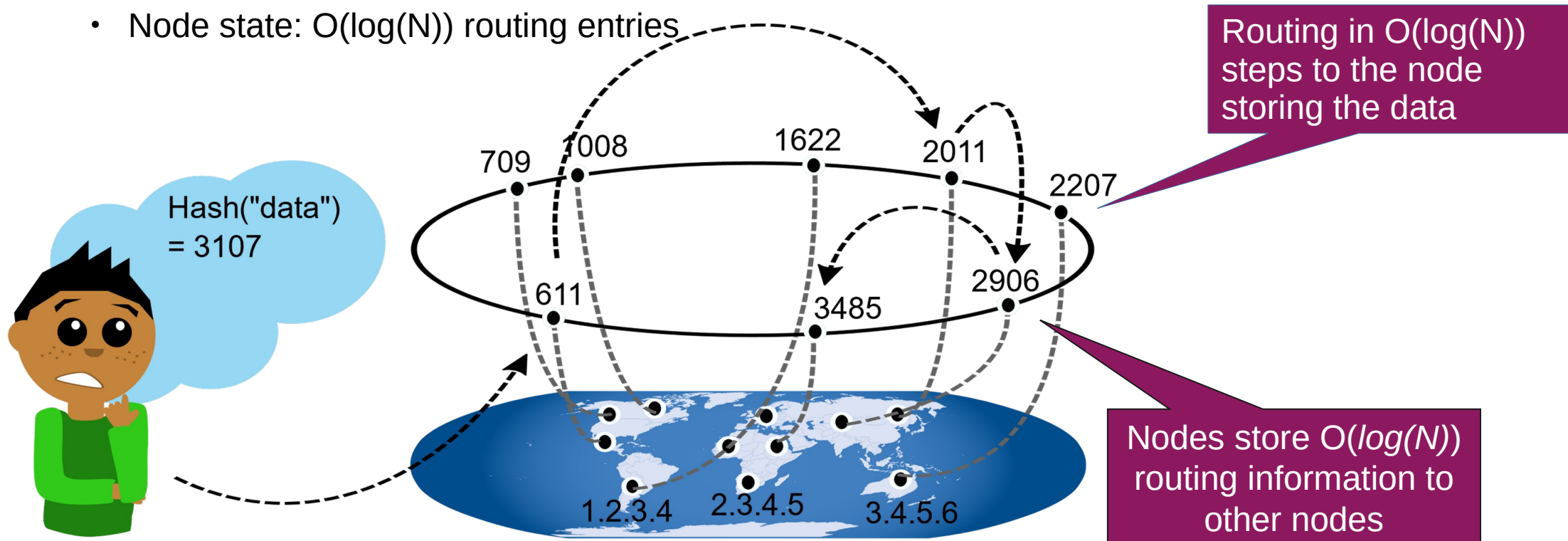


Motivation Distributed Indexing (1)



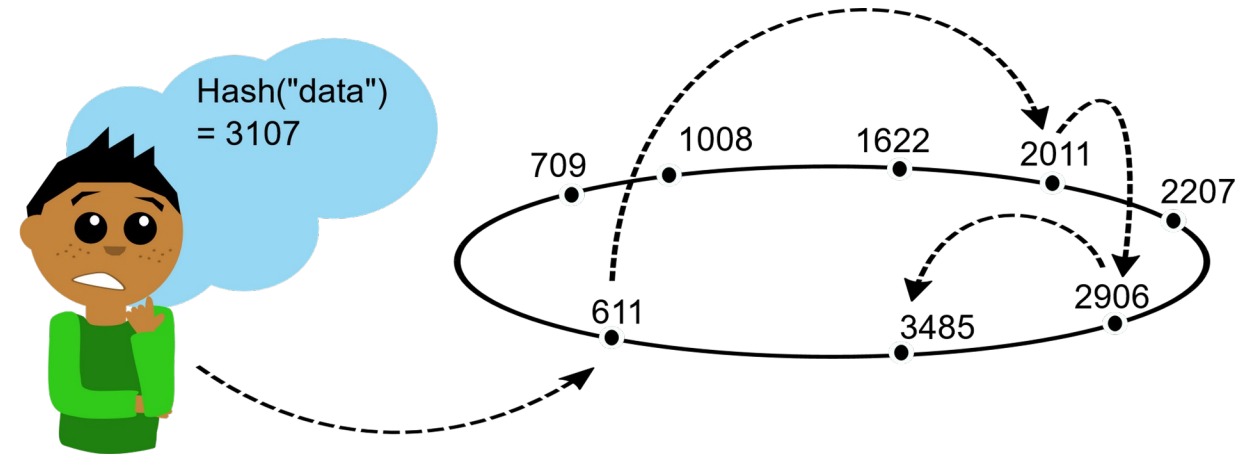
Distributed Indexing (1)

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



Distributed Indexing (2)

- Approach of distributed 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
 - Fuzzy queries not primarily supported (e.g., wildcard searches)



Comparison of Lookup Concepts

- **Big O notation:** classify computer algorithms

System	Per Node State	Communication Overhead	Fuzzy Queries	No false negatives	Robustness / horizontal scalable
Central Server	$O(N)$	$O(1)$			()
Flooding	$O(1)$	$O(N)$		()	
Distributed Hash Tables	$O(\log N)$	$O(\log N)$	()		

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**
 - $\text{Bucket} = \text{hash}(x) \bmod n$
 - If n changes, remapping / bucket changes
 - N changes if capacity is reached
 - Remapping is expensive in DHT!
 - DHTs reassign responsibility

Distributed Management of Data

1. Mapping of nodes and data into same address space

- Peers and content are addressed using flat identifiers (IDs)
 - E.g., Address is public key (256bit) or SHA256 of public key. Content ID = SHA256(content)
- Common address space for data and nodes
- Nodes are responsible for data in certain parts of the address space
- Association of data to nodes may change since nodes may disappear

2. Storing / Looking up data in the DHT

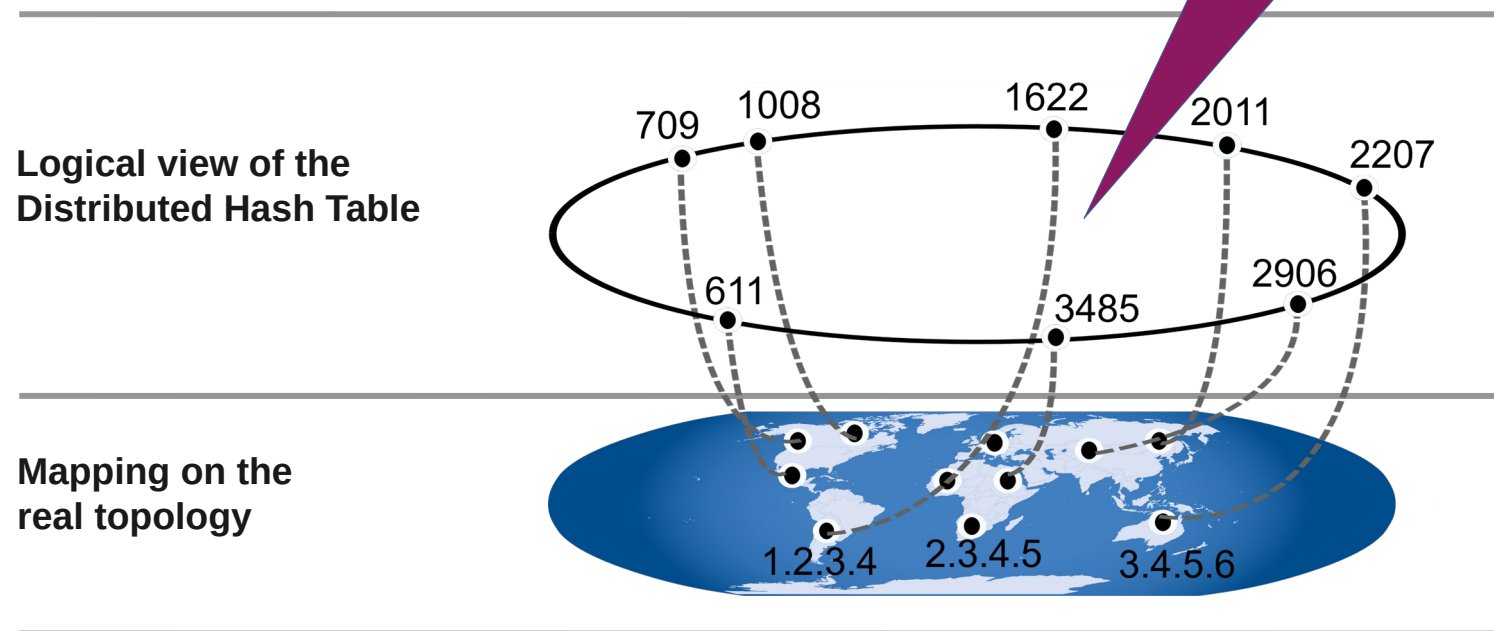
- Store data = first, search for responsible node
 - Not necessarily known in advance
- Search data = first, search for responsible node

Association of Address Space with Nodes

- Each node is responsible for part of the value range
 - Often with redundancy (overlapping of parts)
 - Continuous adaptation
 - Real (underlay) and logical (overlay) topology are uncorrelated

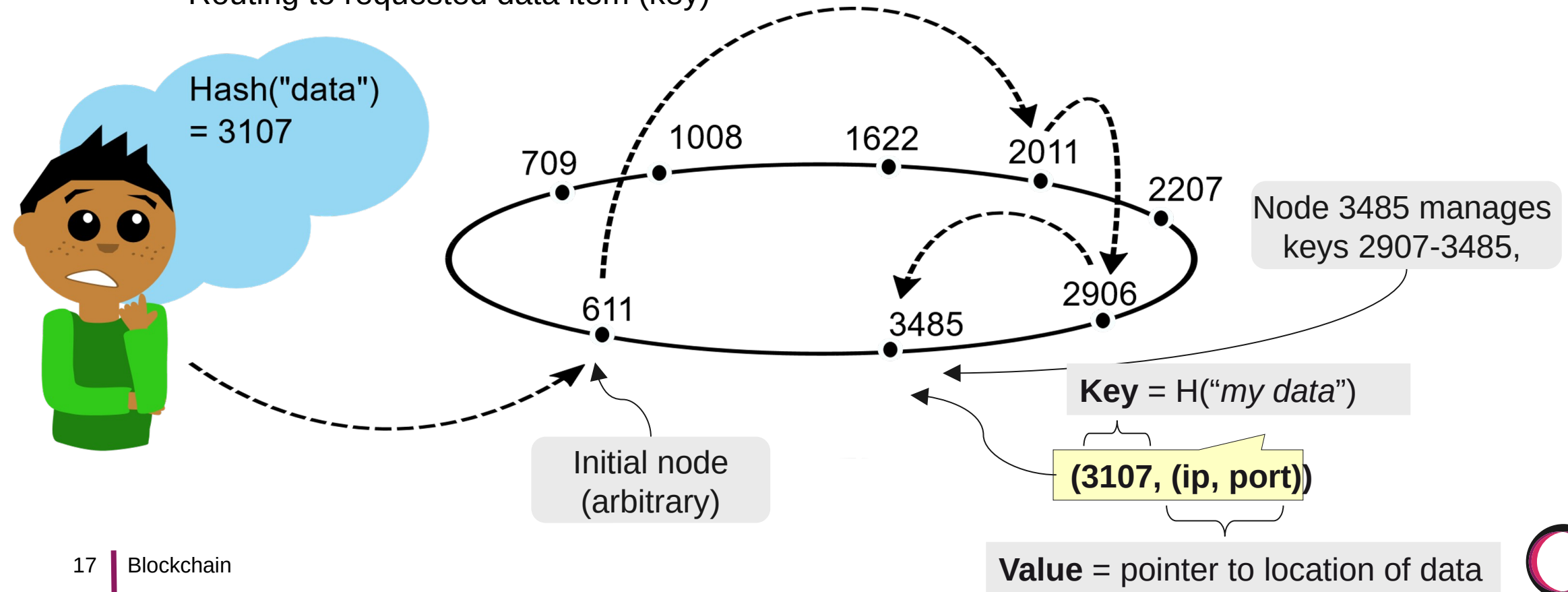
Node 3485 is responsible for data items in range 2907 to 3485

(in case of a Chord-DHT)



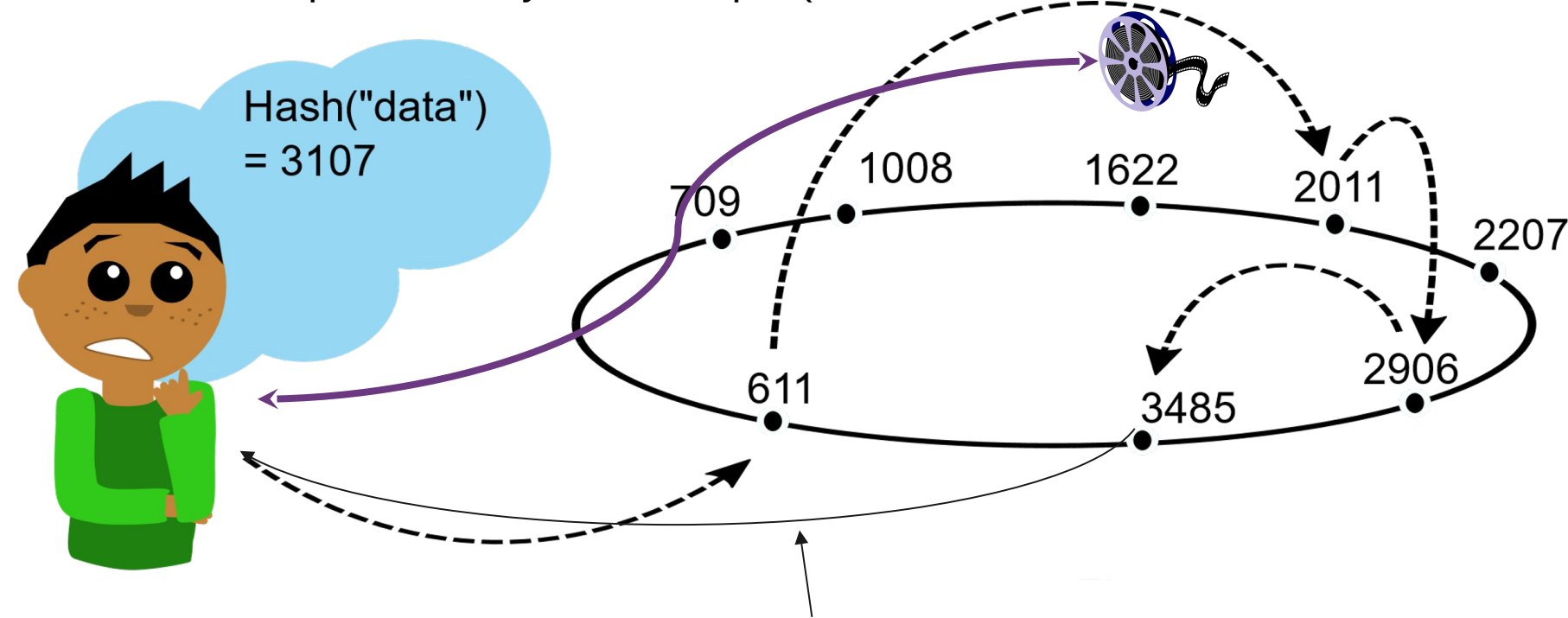
Routing to a Data Item

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



Routing to a Data Item

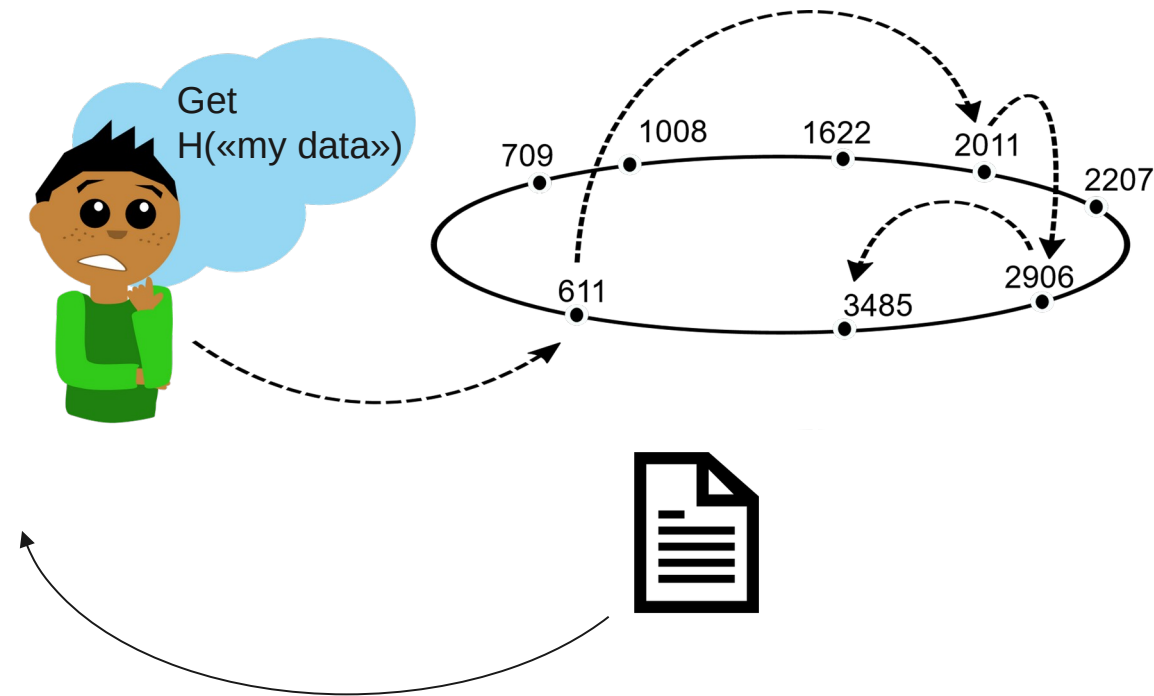
- Getting the content
 - K/V-pair is delivered to requester
 - Requester analyzes K/V-tuple (and downloads data from actual location – in case of indirect storage)



Node 3485 sends
(3107, (ip/port)) to requester

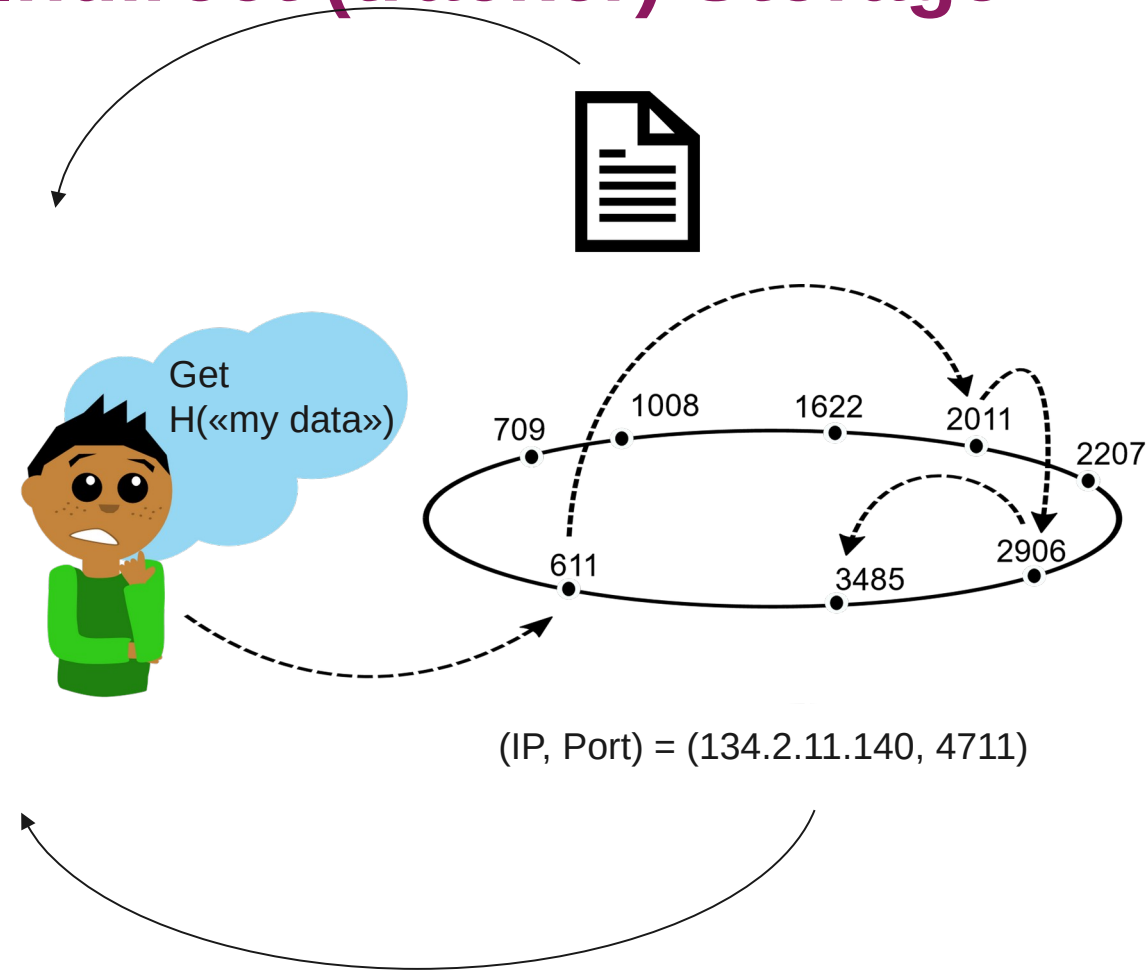
Association of Data with IDs – Direct Storage

- How is content stored on the nodes?
 - Example:
H("my data") = 3107 is mapped into DHT address space
- Direct storage
 - Content is stored in responsible node for H("my data")
 - → Inflexible for large content – o.k., if small amount data (~KB) or used internally



Association of Data with IDs – Indirect (tracker) Storage

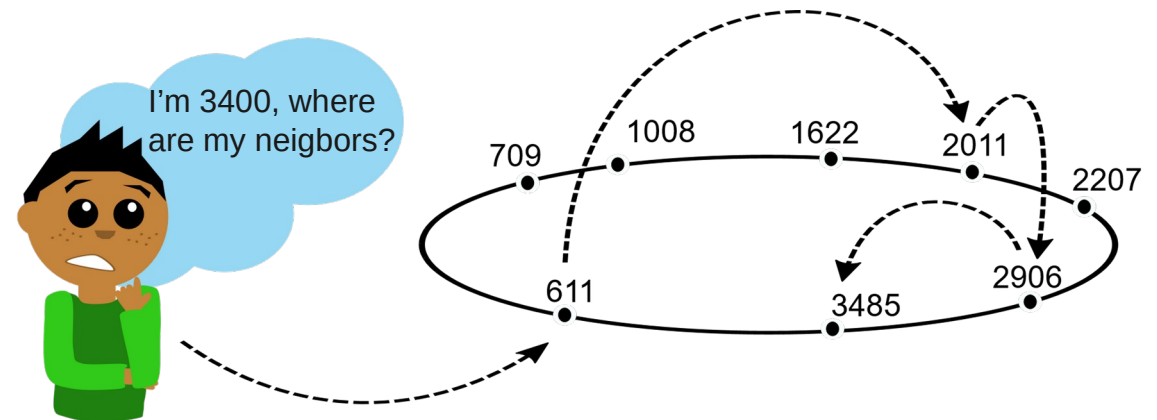
- Indirect storage
 - Nodes in a DHT store tuples like (key,value)
 - Key = Hash(„my data”) → 2313
 - Value is often real storage address of content:
(IP, Port) = (134.2.11.140, 4711)
 - More flexible for large data, but one step more to reach content



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
($A \rightarrow B == B \rightarrow A$)
 - 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

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

1 1 0 1 0 1

XOR

0 1 0 0 0 1

↓

1 0 0 1 0 0

$$d_{\text{XOR}}(A,B) = 1 0 0 1 0 0_2 = 36_{10}$$

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
 $6 \text{ xor } 3 = 101\text{b}$

- Neighbors of 6, if $k=1$

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

Routing Table of #0
 $0 \text{ xor } 3 = 11\text{b}$

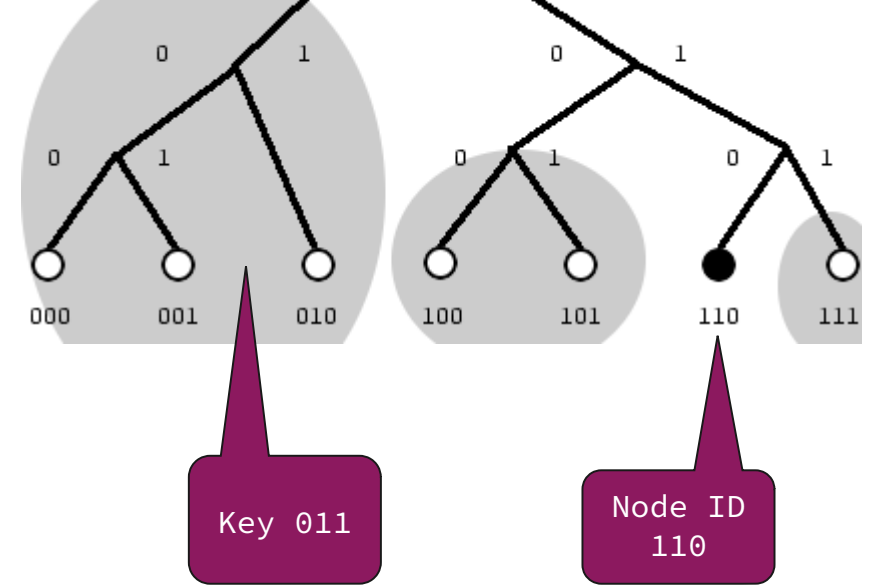
- Search for 3, ask 0, neighbors of 0
- Ask 2, neighbors of 2

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

Routing Table of #2
 $2 \text{ xor } 3 = 1\text{b}$

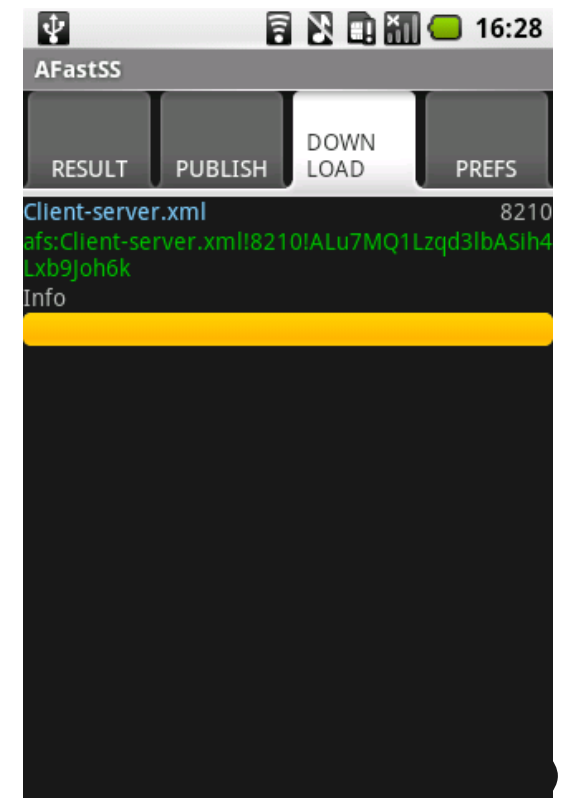
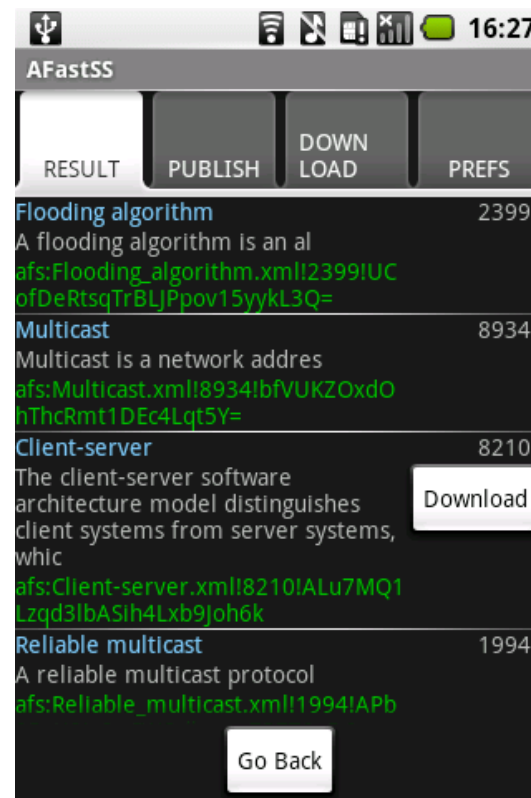
- Ask 2, 2 replies 0. 6 figures that there is no closer node, 2 is the closest one ($2 \text{ xor } 3 = 1$)

Routing with XOR,
with 3-bits



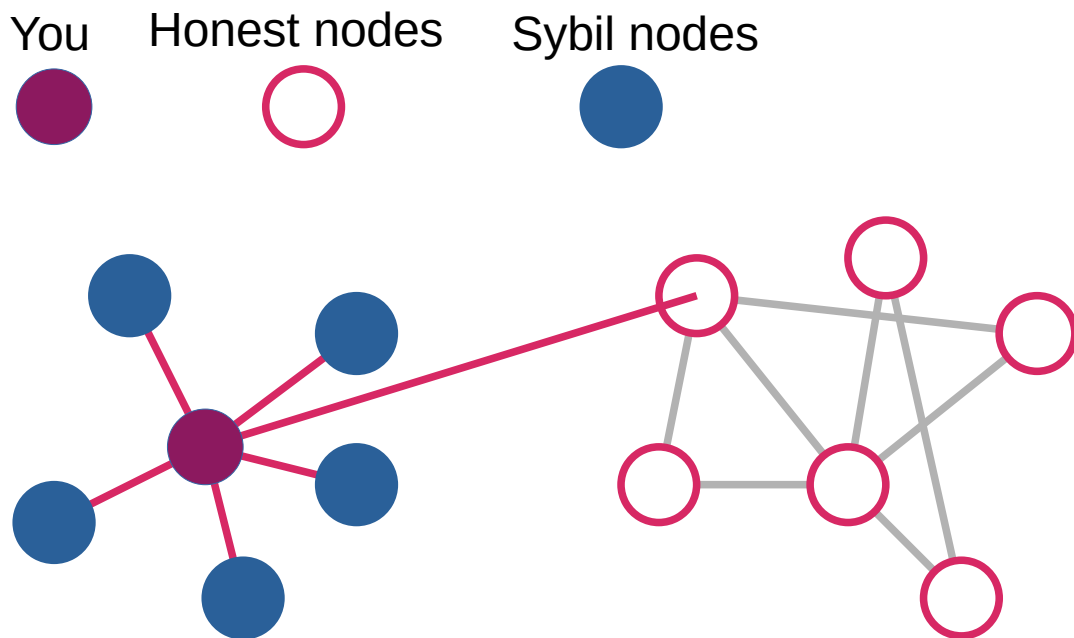
TomP2P

- 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



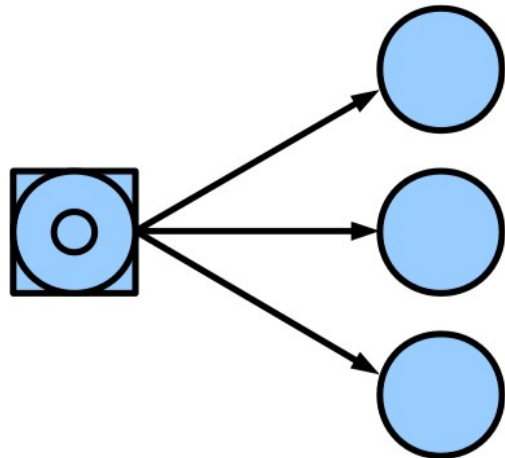
- Sybil attack
 - Create large number of identities
 - Larger than honest nodes
 - Control “close” nodes in a DHT
 - Isolate nodes
- Prevention [[source](#)]
 - Creation of identities costs money
 - Always assume data from other nodes may be missing
 - Bitcoin – chain of block, if block is missing, you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

Attacking the DHT

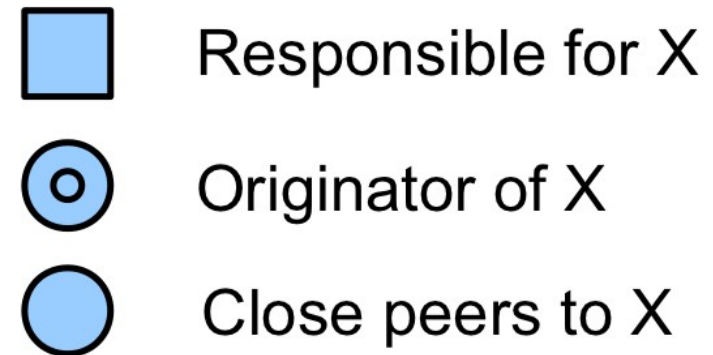
- Example
- Create a key for a data item close to the target:
Number160.createHash(data).xor(new Number160(0)) – distance 0, perfect match
Number160.createHash(data).xor(new Number160(1)) – distance 1
Number160.createHash(data).xor(new Number160(2)) – distance 2
...
- Or create key of node close to the 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



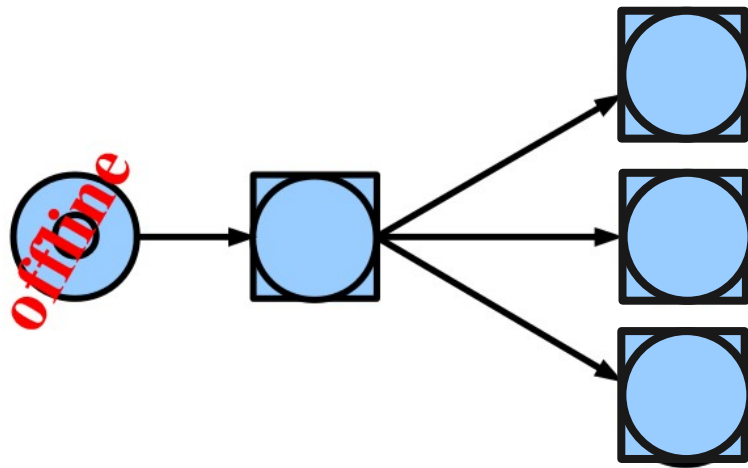
- Problem
 - Originator offline → replicas disappear. Content has TTL



Redundancy in DHTs

- Indirect Replication

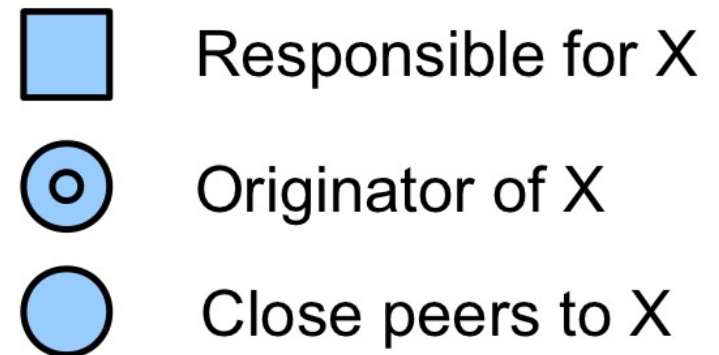
- The closest peer is responsible, originator may go offline vs any close peers are responsible
 - Periodically checks if enough replicas exist
 - Detects if responsibility changes



closest vs any

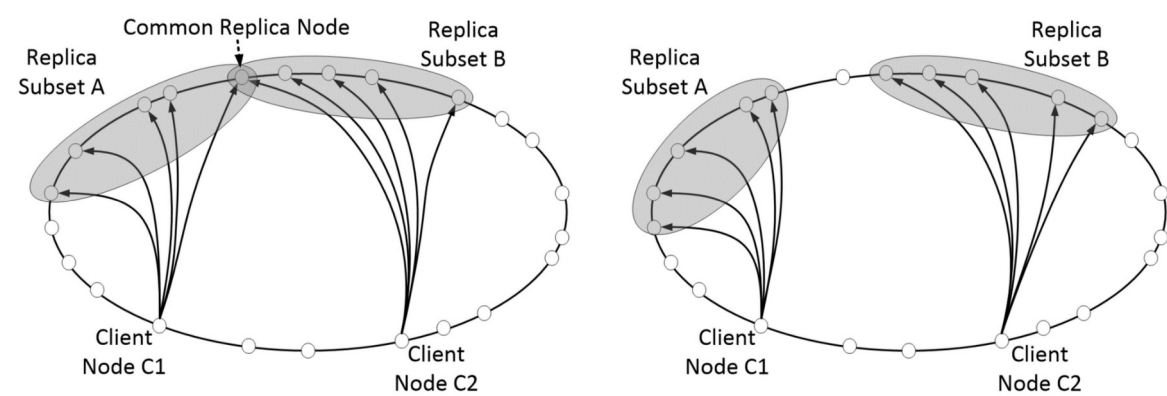
- Problem

- 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