FS19

DISTRIBUTED HASH TABLES

Kademlia & co.

T. Bocek

thomas.bocek@hsr.ch

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Agenda

- **Distributed Systems in the News**
- **Distributed Management and Retrieval of Data**
- **Fundamentals of Distributed Hash Tables**
- **DHT Mechanisms and Interfaces**
- **Kademlia**

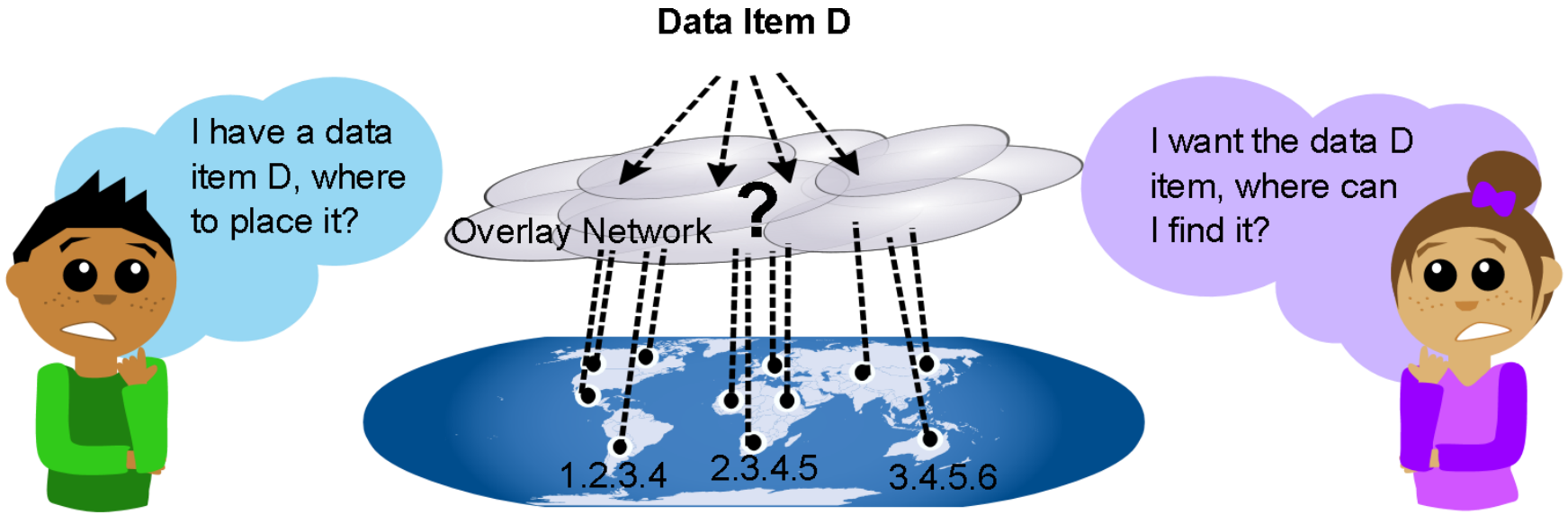
Goals

- **I know the three strategies to locate data in a decentralized system**
- **I know the advantages, disadvantages, and overhead for those strategies**
- **I know how a DHT operates**
- **I know the routing mechanism of Kademlia, its advantages and disadvantages**

Transparency

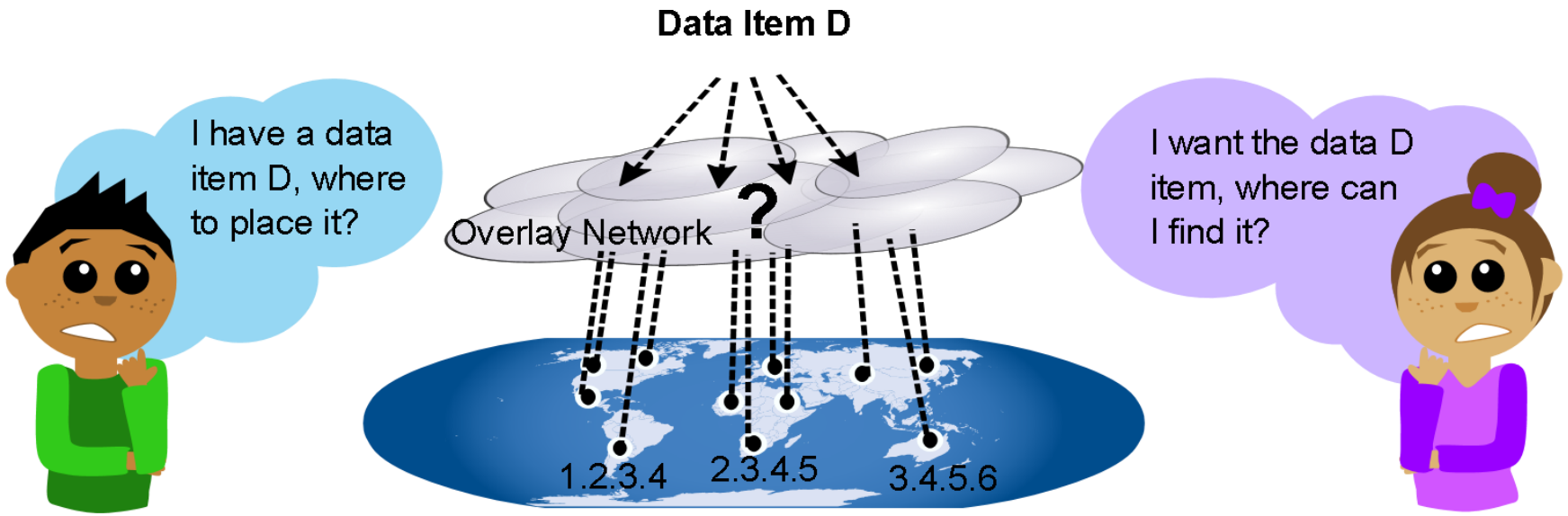
- **Distributed system should hide its distributed nature**
  - Access transparency - users should access resources in a single, uniform way
  - Location transparency – users should not be aware of the physical location
  - Migration, relocation transparency – users should not be aware, that resource have moved
  - 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
  - Failure transparency – users should be aware of recovery mechanisms
- Security transparency – users should be minimally aware of security mechanisms

- **Essential challenge in (most) Peer-to-Peer systems?**
  - **Location of a data item among systems distributed**
    - Where shall the item be stored?
    - How does a requester find the actual location of an item?
  - **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
  - Flooding search
  - Distributed indexing

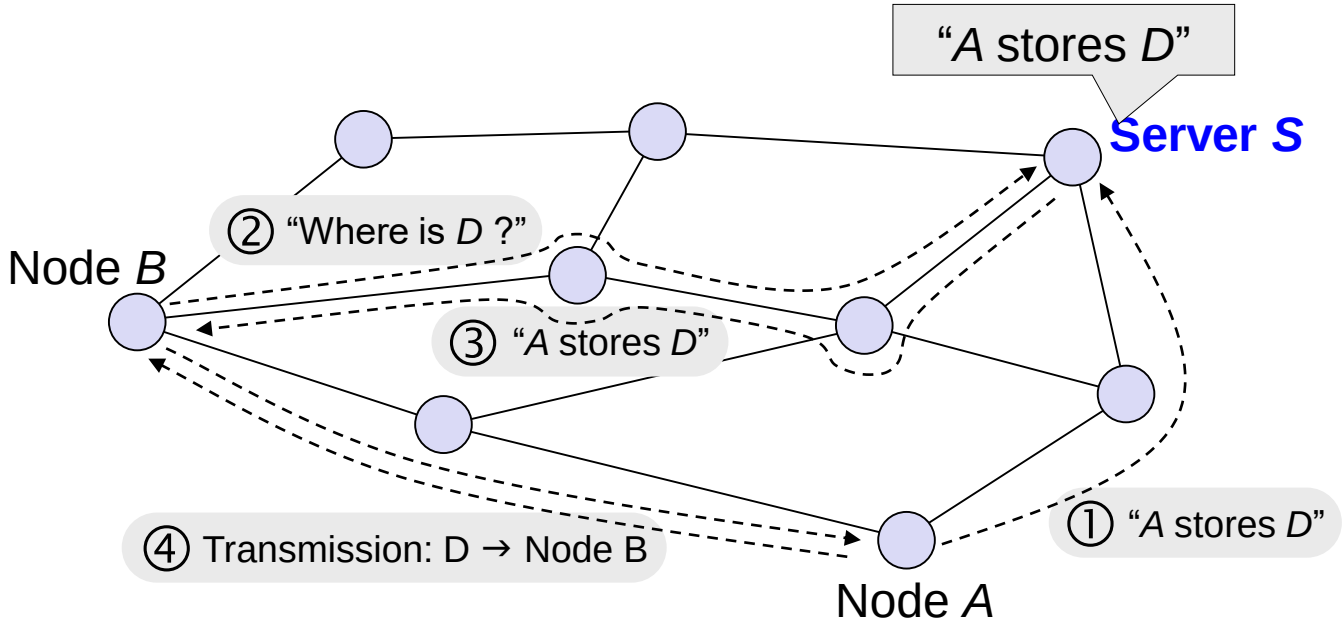


URL: b.socrative.com/student/

Room: HSR

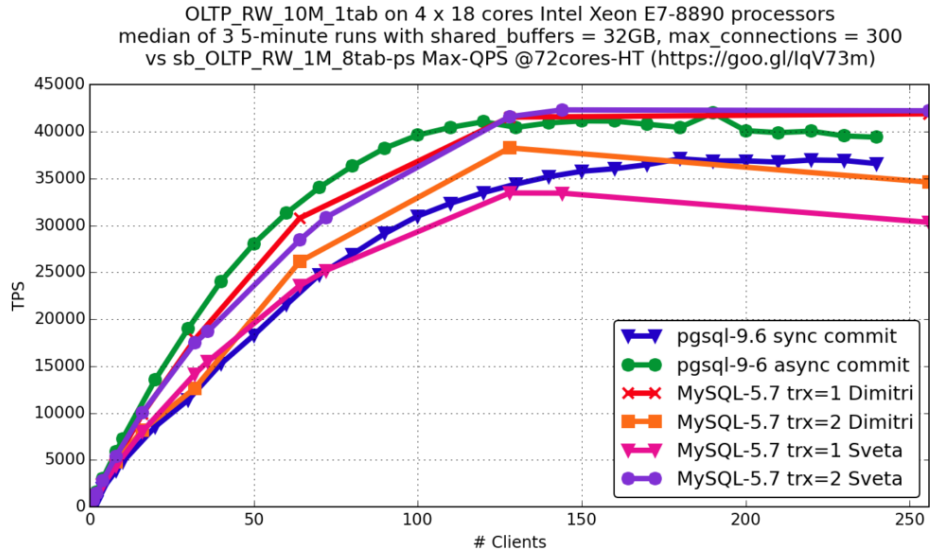
Central Server

- **Simple strategy: Central Server**
  - 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 (2)

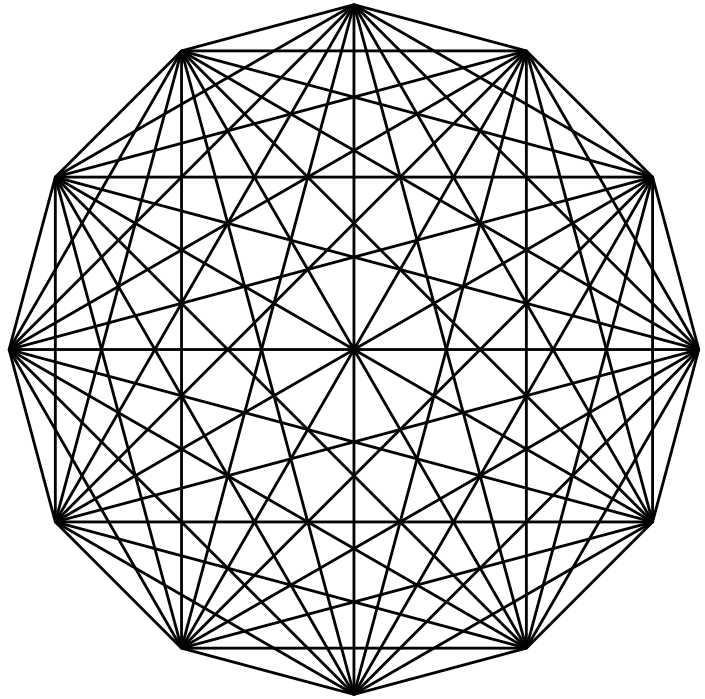
- **Advantages**
  - Search complexity of O(1) – *"just ask the server"*
  - Complex and fuzzy queries are possible
  - Simple and fast
- **Problems**
  - No Scalability
    - O(N) node state in server
    - O(N) network and system load of server
  - Single point of failure or attack (also for lawsuits, or not ;-)
  - (Single) central server not suitable for systems with massive numbers of users
    - Or you scale internally (google, dropbox)
- **But overall, …**
  - Best principle for small and simple applications!



https://www.percona.com/blog/2017/01/06/millions-queries-per-second-postgresql-and-mysql-peaceful-battle-at-modern-demanding-workloads/

Approach II: Flooding Search (1)

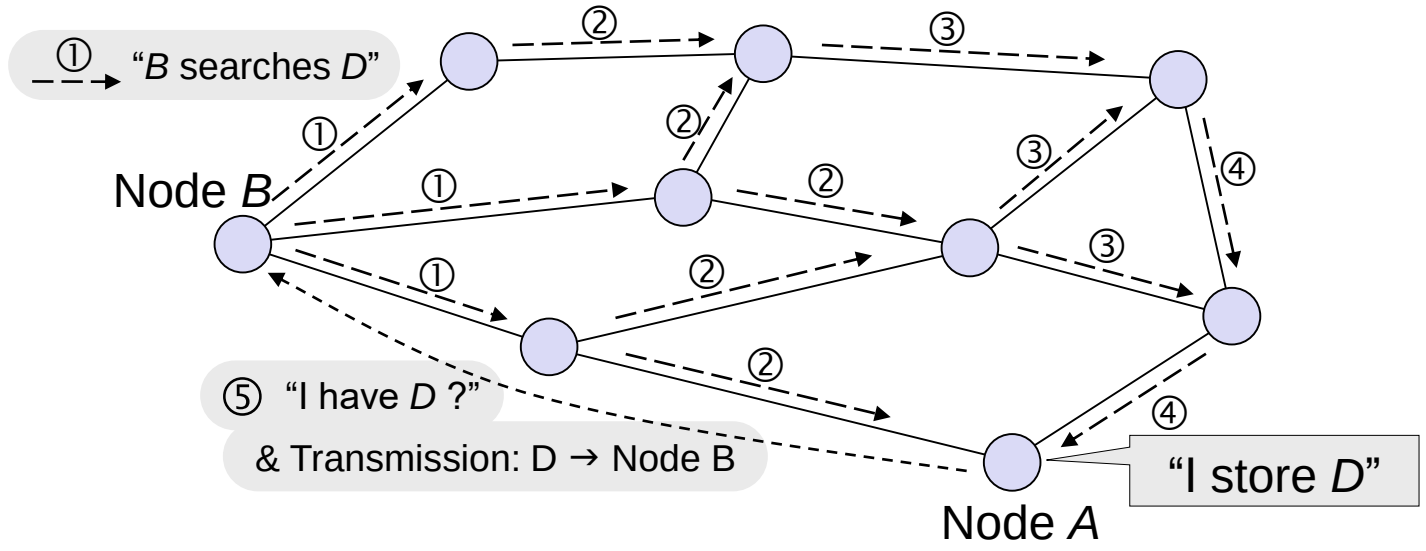
- **Fully-distributed Approach**
  - Central systems are vulnerable and do not scale
  - Unstructured Peer-to-Peer systems follow opposite approach
  - No information on location of a content
- **Retrieval of data**
  - No routing information for content
  - Necessity to ask as much systems as possible / necessary
  - Approaches
    - Flooding: high traffic load on network, does not scale
    - Highest degree search: quick search through large areas – large number of messages needed for unique identification



Approach II: Flooding Search (2)

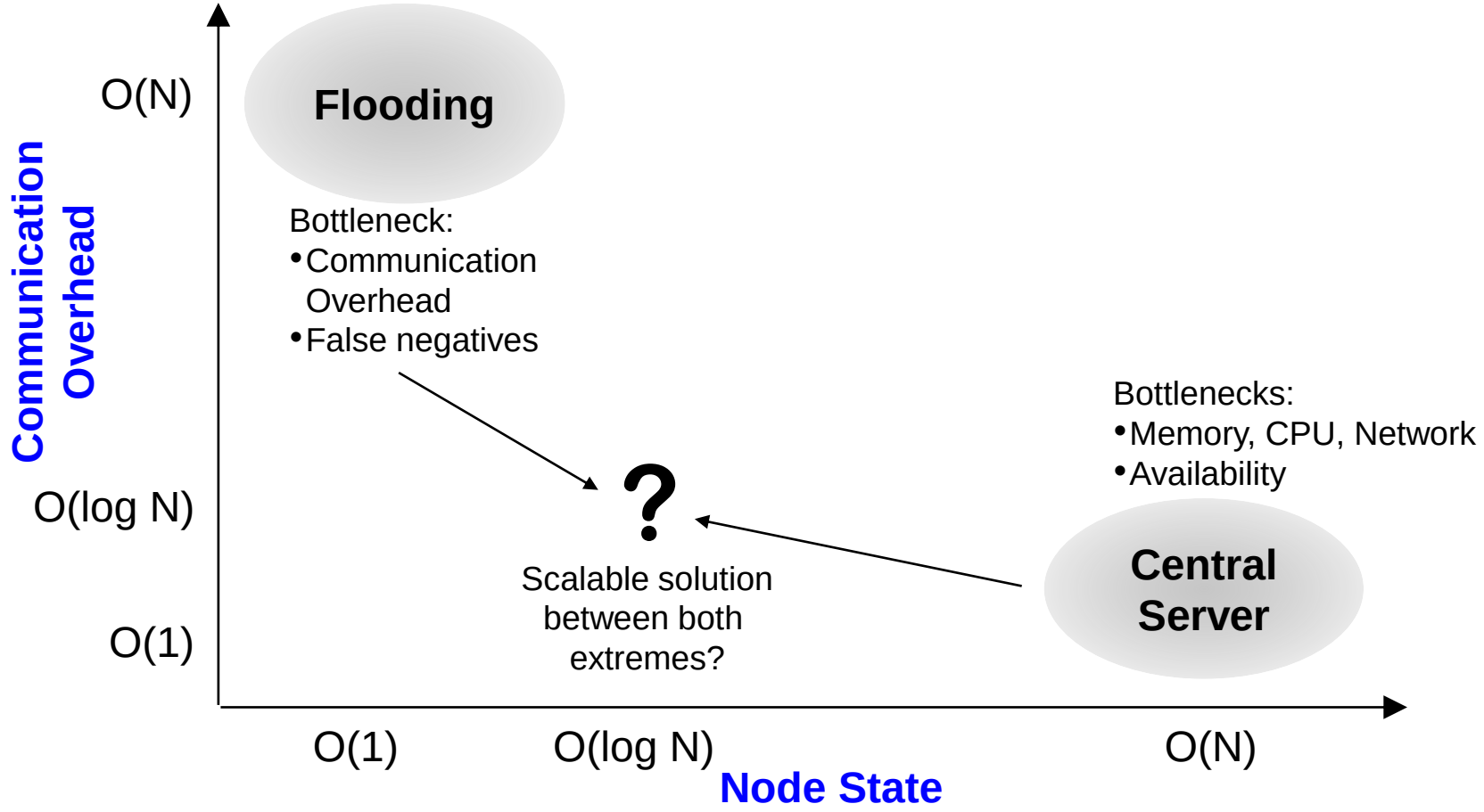
- **Fully Decentralized Approach: Flooding Search**
  - No information about location of data in the intermediate systems
  - Necessity for broad search

- 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



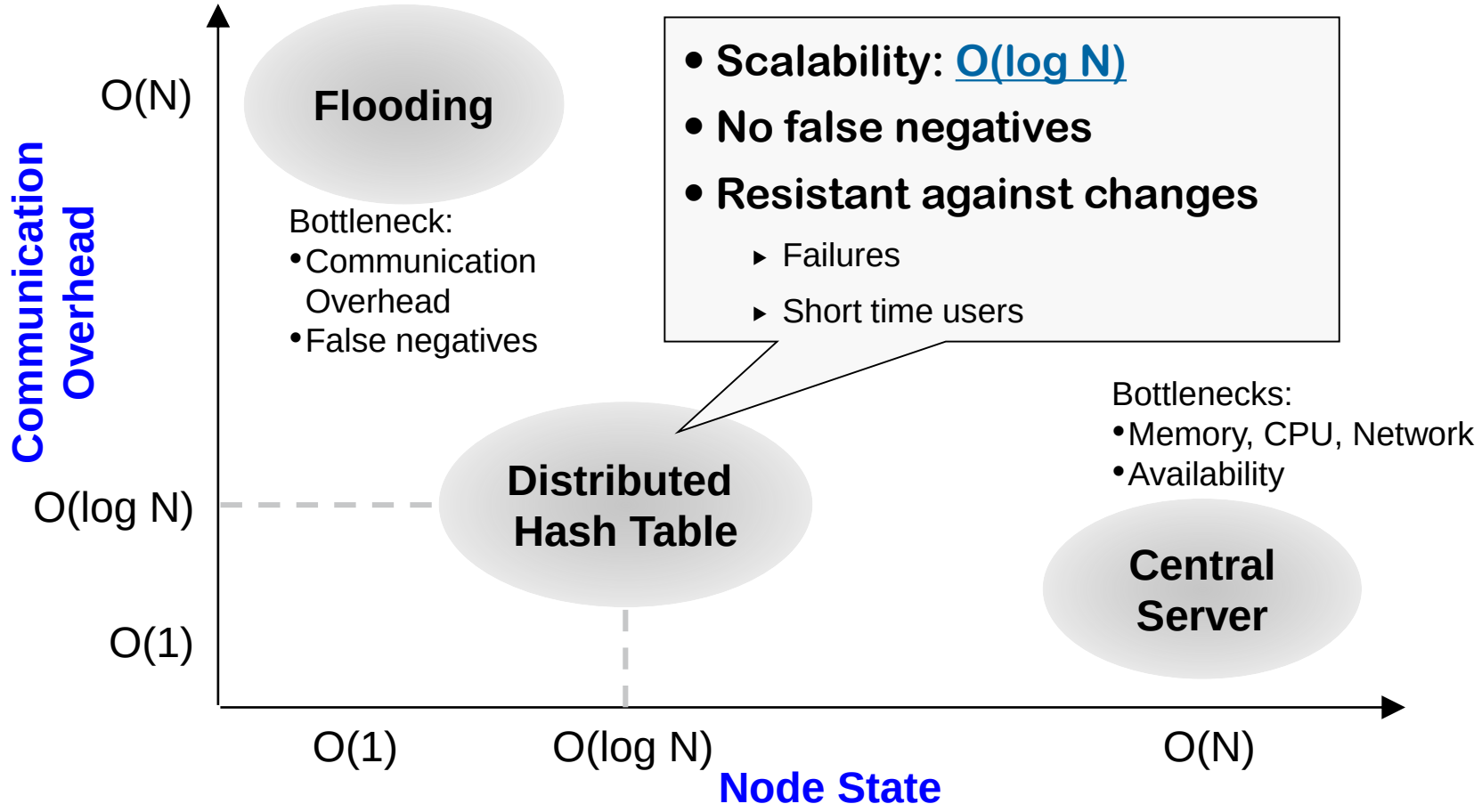
Motivation Distributed Indexing (1)

- **Communication overhead vs. node state**



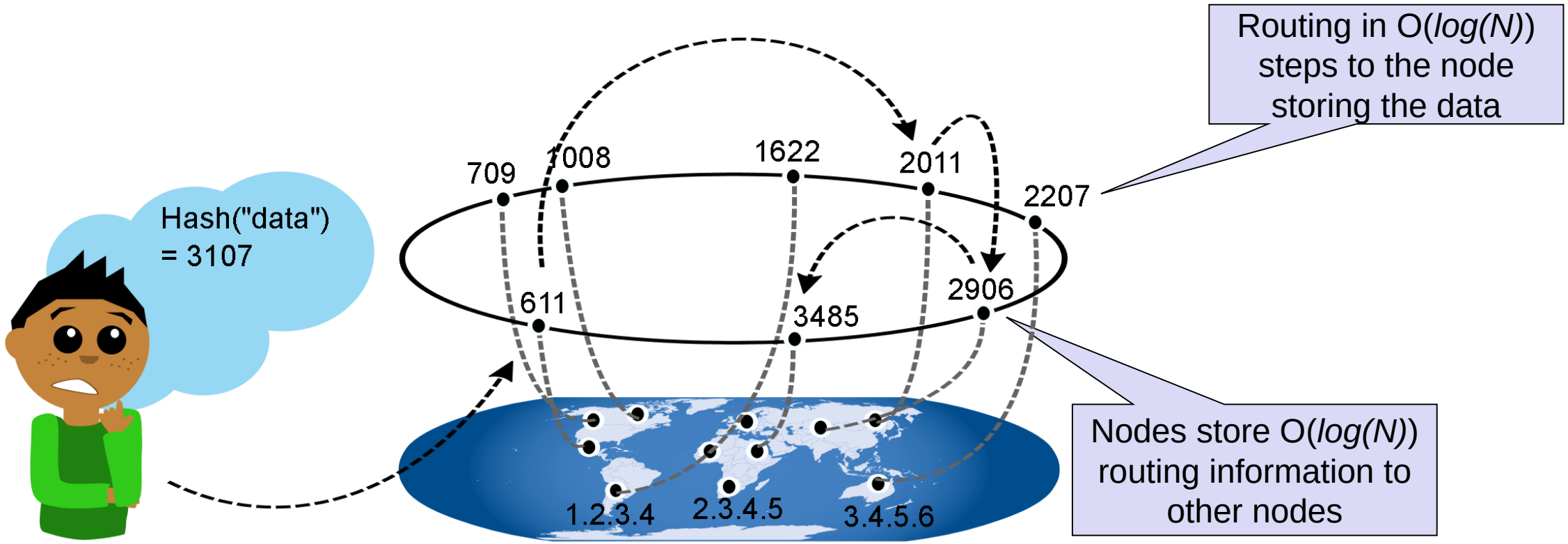
Motivation Distributed Indexing (2)

- **Communication overhead vs. node state**



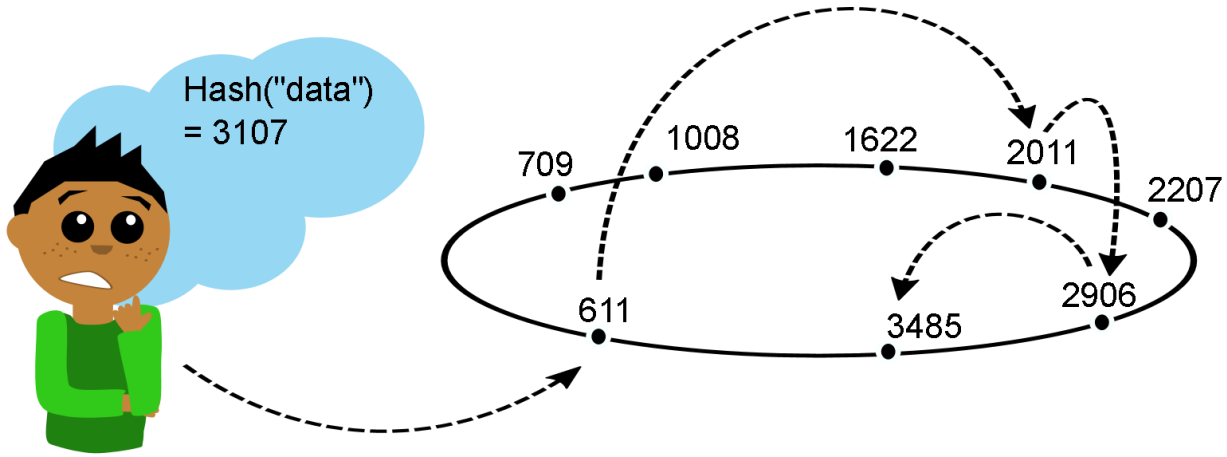
Distributed Indexing (1)

- **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

System	Per Node State	Communi-cation Overhead	Fuzzy Queries	No false negatives	Robust-ness
Central Server	O(N)	O(1)	✓	✓	✗
Flooding Search	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**
  - Bucket = hash(x) mod n
  - If n changes, remapping / bucket changes
  - N changes if capacity is reached
  - Remapping is expensive in DHT!

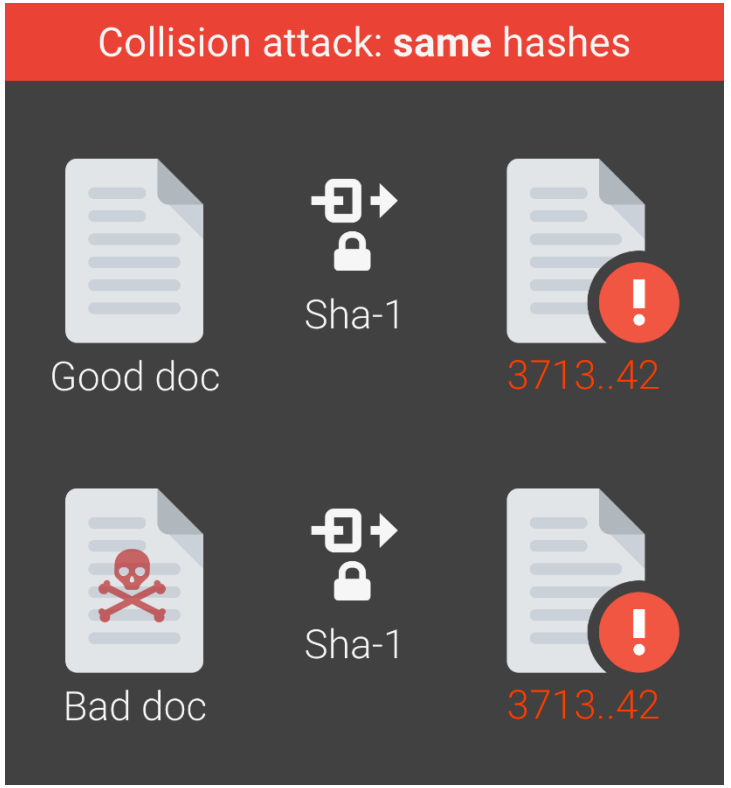
URL: b.socrative.com/student/

Room: HSR

Distributed Management of Data

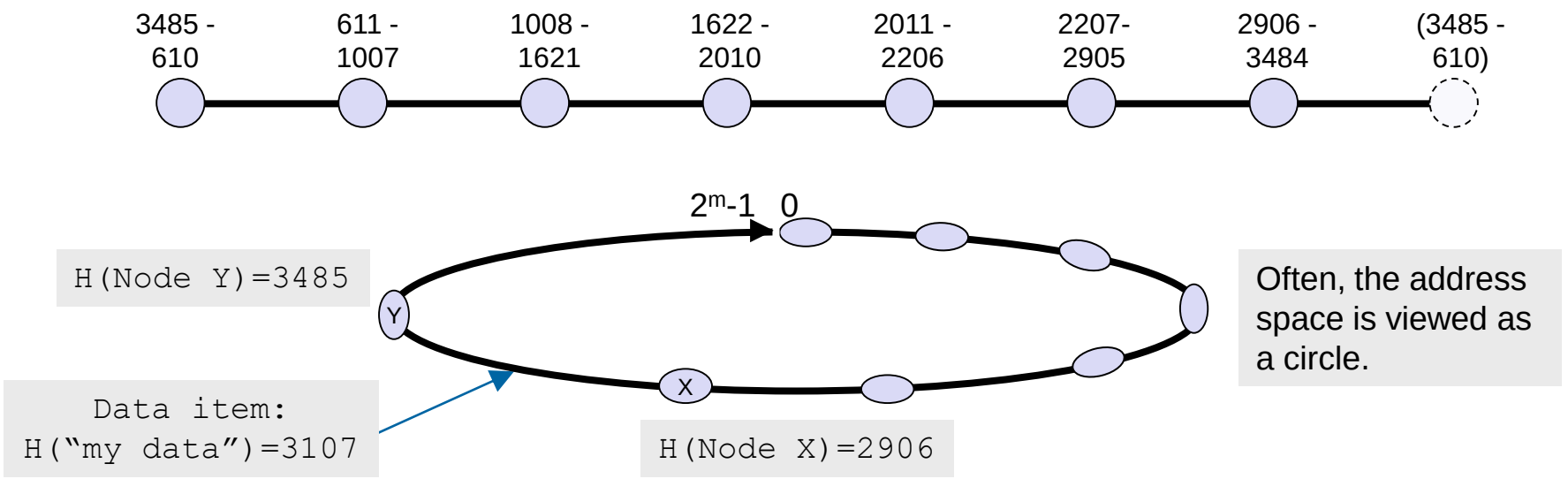
- **1. Mapping of nodes and data into same address space**
  - Peers and content are addressed using flat identifiers (IDs)
    - E.g., Address is public key (256bit) or SHA256 of public key. Content ID = SHA256(content)
  - Common address space for data and nodes
  - Nodes are responsible for data in 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**
  - Search data = first, search for responsible node
  - Store data = first, search for responsible node
    - Not necessarily known in advance

- SHA-1?



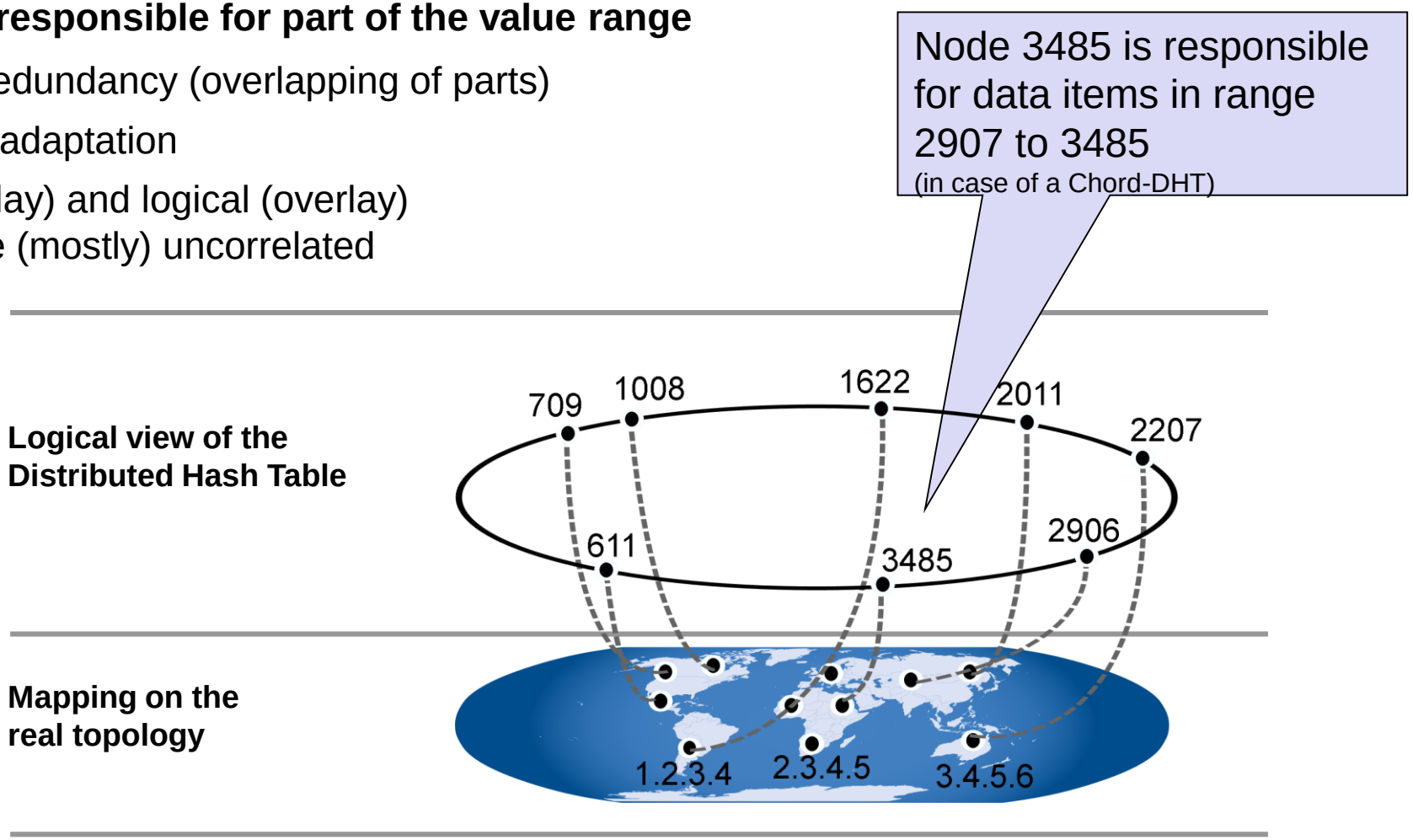
Addressing in Distributed Hash Tables

- **Step 1: Mapping of content/nodes into linear space**
  - Usually: 0, …, 2^m-1 >> number of objects to be stored
  - Mapping of data and nodes into an address space (with hash function)
    - E.g., Hash(*String*) mod 2^m: H(„*my data*“) → 3107



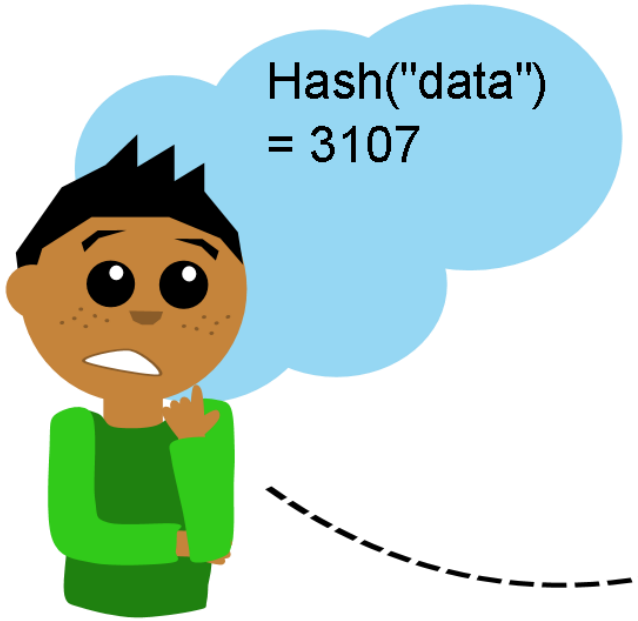
Association of Address Space with Nodes

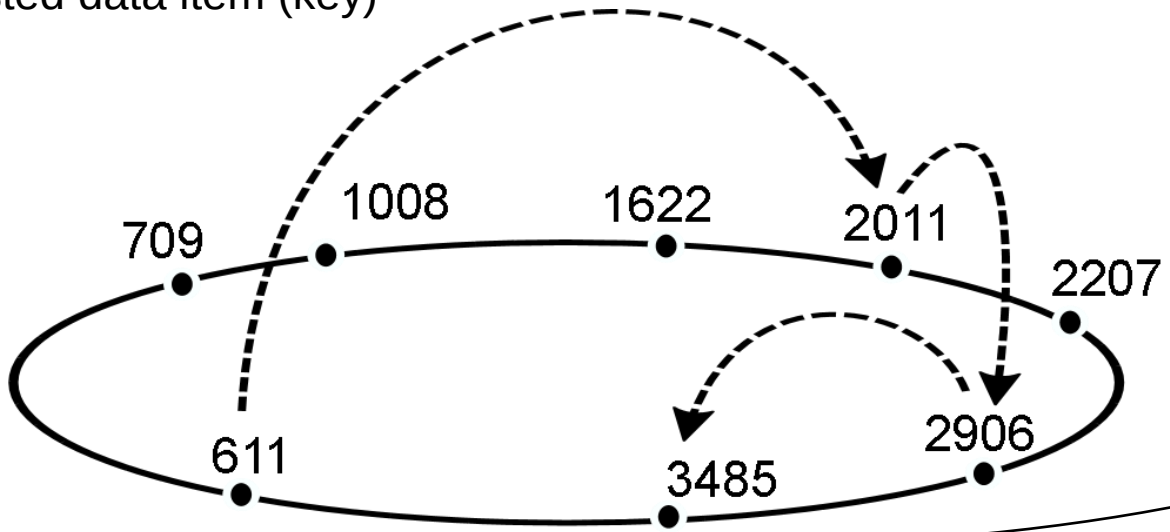
- **Each node is responsible for part of the value range**
  - Often with redundancy (overlapping of parts)
  - Continuous adaptation
  - Real (underlay) and logical (overlay) topology are (mostly) uncorrelated



Routing to a Data Item

- **Step 2: Locating the data / Routing to a K/V-pair**
  - Start lookup at arbitrary node of DHT
  - Routing to requested data item (key)





Node 3485 manages keys 2907-3485,

Initial node (arbitrary)

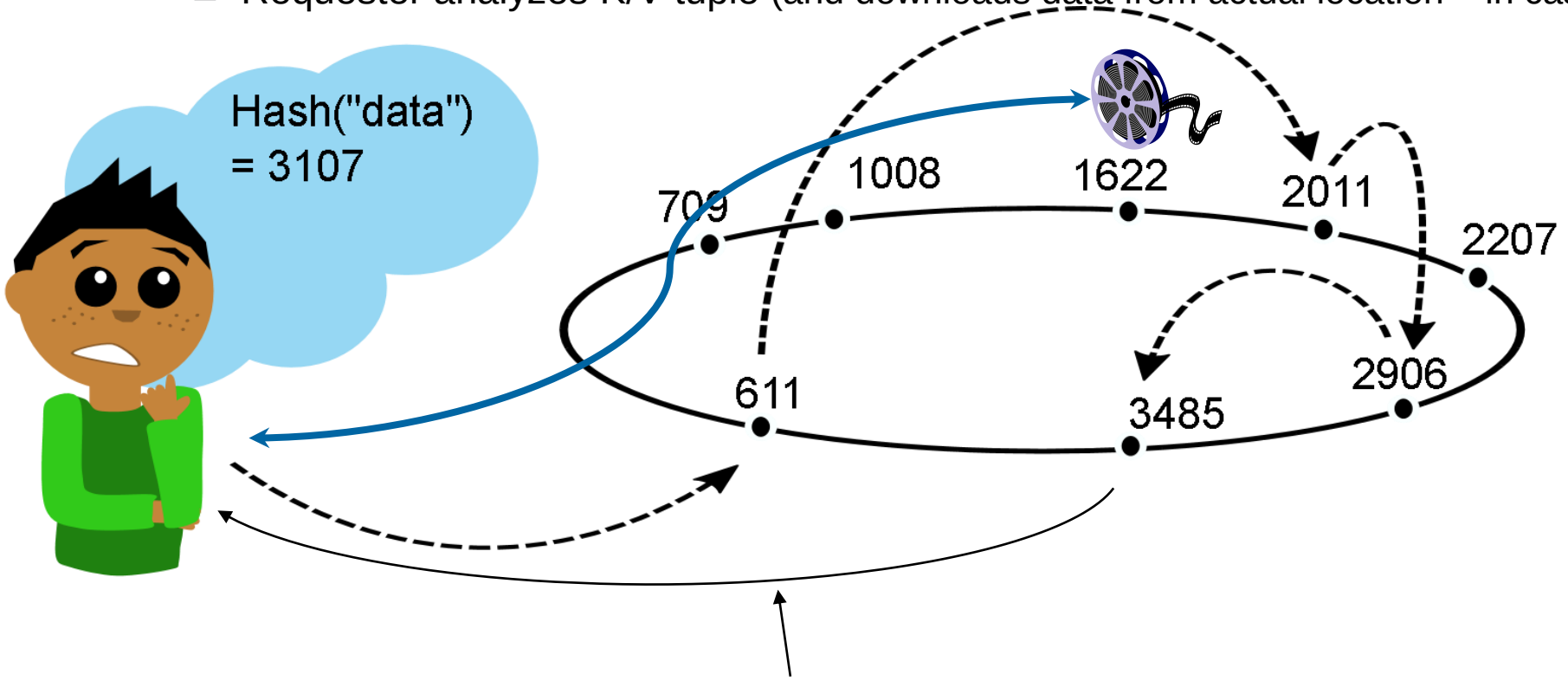
Key = H("*my data*")

(3107, (ip, port))

Value = pointer to location of data

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)

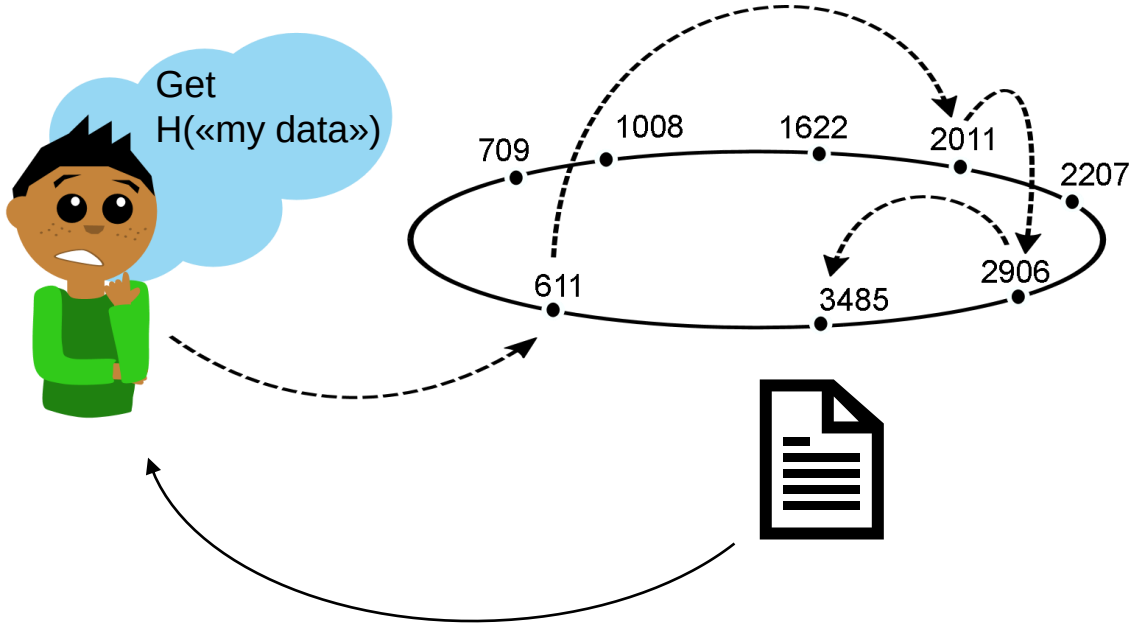


Node 3485 sends
(3107, (ip/port)) to requester

Association of Data with IDs – Direct Storage

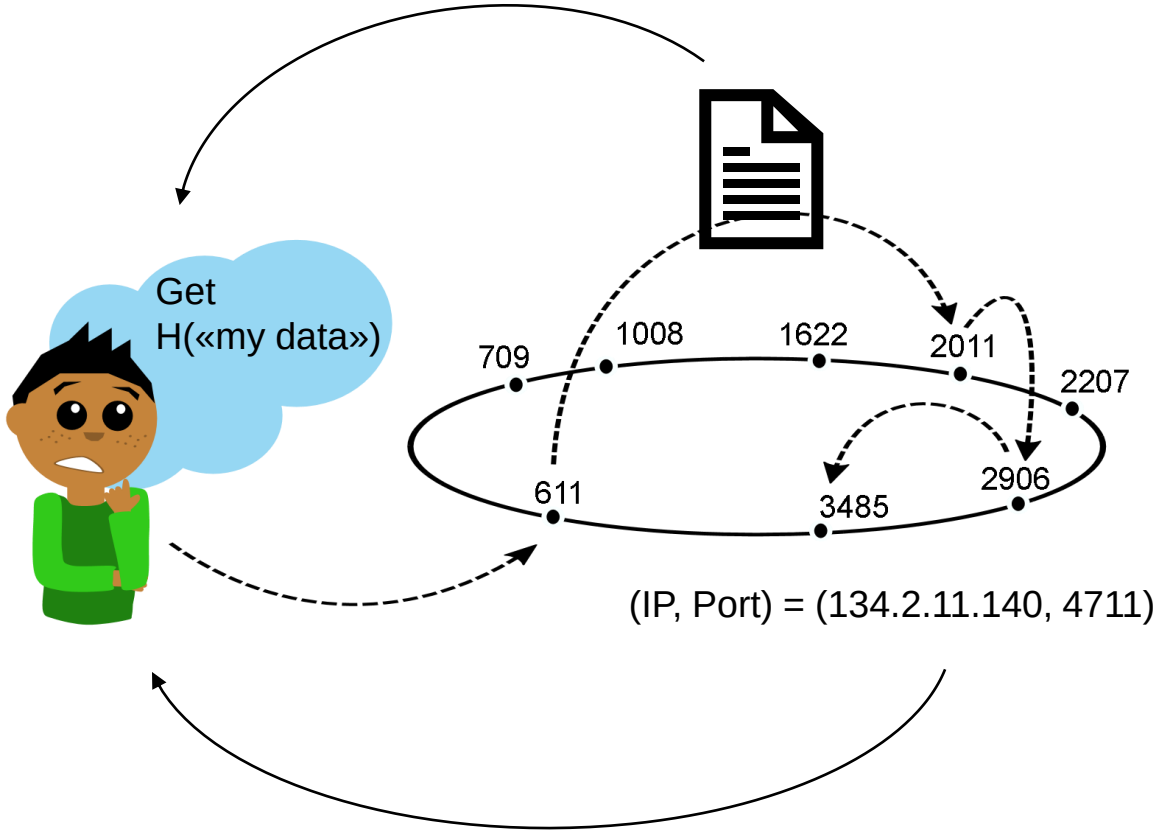
- **How is content stored on the nodes?**
  - Example:
    H(“*my data*”) = 3107 is mapped into DHT address space
- **Direct storage**
  - Content is stored in responsible node for H(“*my data*”)

    → Inflexible for large content – o.k., if small amount data (~KB)



Association of Data with IDs – Indirect (tracker) Storage

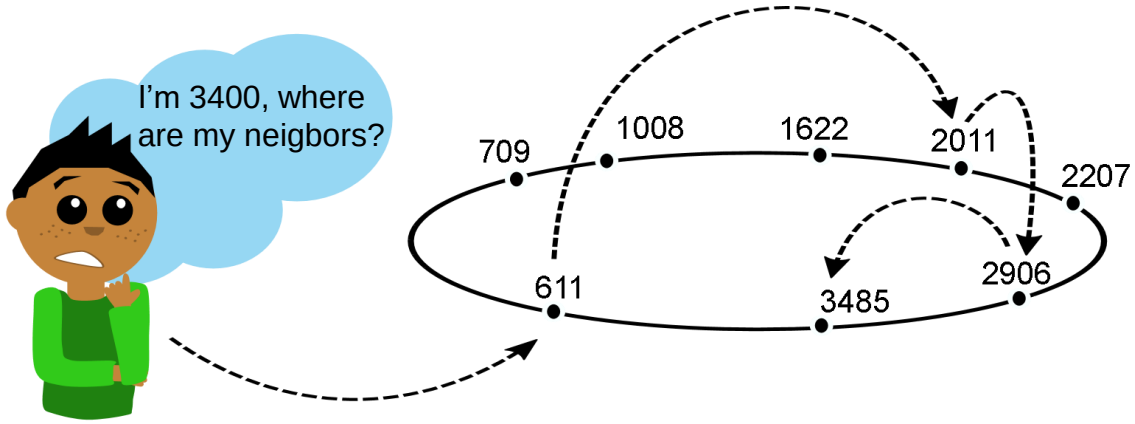
- **Indirect storage**
  - Nodes in a DHT store tuples like (key,value)
    - Key = Hash(„*my data*”) → 2313
    - Value is often real storage address of content: (IP, Port) = (134.2.11.140, 4711)
  - More flexible, but one step more to reach content



Node 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



Properties of DHTs

- **Use of routing information for efficient search for content**
- **Keys are evenly distributed across nodes of DHT**
  - No bottlenecks
  - Failure of nodes can be tolerated
  - Survival of some attacks possible
- **Self-organizing system**
- **Simple and efficient realization**
- **Supporting a wide spectrum of applications**
  - Flat (hash) key without semantic meaning
  - Value depends on application

- **Examples of generic Distributed Hash Tables**
  - Pastry (freepastry, v2.1, 13.3.2009) Microsoft Research, Rice University
  - Chord (v1.0.5, 11.4.2008) UC Berkeley, MIT
  - Tapestry (chimera 1.20 ~2007) UC Berkeley
  - CAN (~2001) UC Berkeley, ICSI
  - P-Grid (v3.2.0_822, 25.11.2008) EPFL Lausanne
  - BEP 44 Storing arbitrary data in the DHT 1KB, many implementations
  - TomP2P (currently working on ED25519 / ChaCha, with KCP)
- **… and there are plenty of others …**

URL: b.socrative.com/student/

Room: HSR

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, Steem, Factom, Tor Onion Services
- **Parallel queries**
  - For one query, α (alpha) concurrent lookups are sent
  - More traffic load, but lower response times

- **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**
  - Kademlia: 160 buckets with size 20 (8)
  - Knows neighbors well, further nodes not that much
  - **If distance can be represented in m bits, bucket m will be used**

Kademlia Example

- **2^3, max size 8, #6 searches for 3**
  - Neighbors of 6, if k=1

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

Routing Table of #6
6 xor 3 = 101b

  - Search for 3, ask 0, neighbors of 0

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

Routing Table of #0
0 xor 3 = 11b

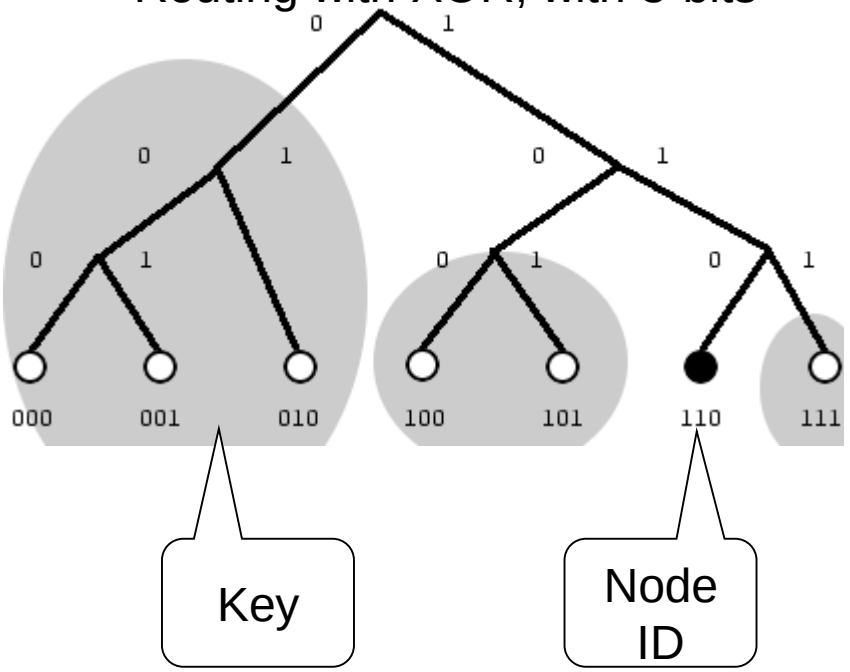
  - Ask 2, neighbors of 2

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



Construction of Routing Table

- **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
  - Check if peer IDs fit better in routing table
- **DHT-based overlay network using the XOR distance metric**
  - Symmetrical routing paths
    (A → B == B → A)
    - due to dXOR(A,B) == dXOR (B,A)

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

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

URL: b.socrative.com/student/

Room: HSR

Questions?

Prof. Dr. Thomas Bocek
thomas.bocek@hsr.ch