Lecture 3

Distributed Hash Tables (1)

Overlay Networks, Decentralized Systems and Their Applications

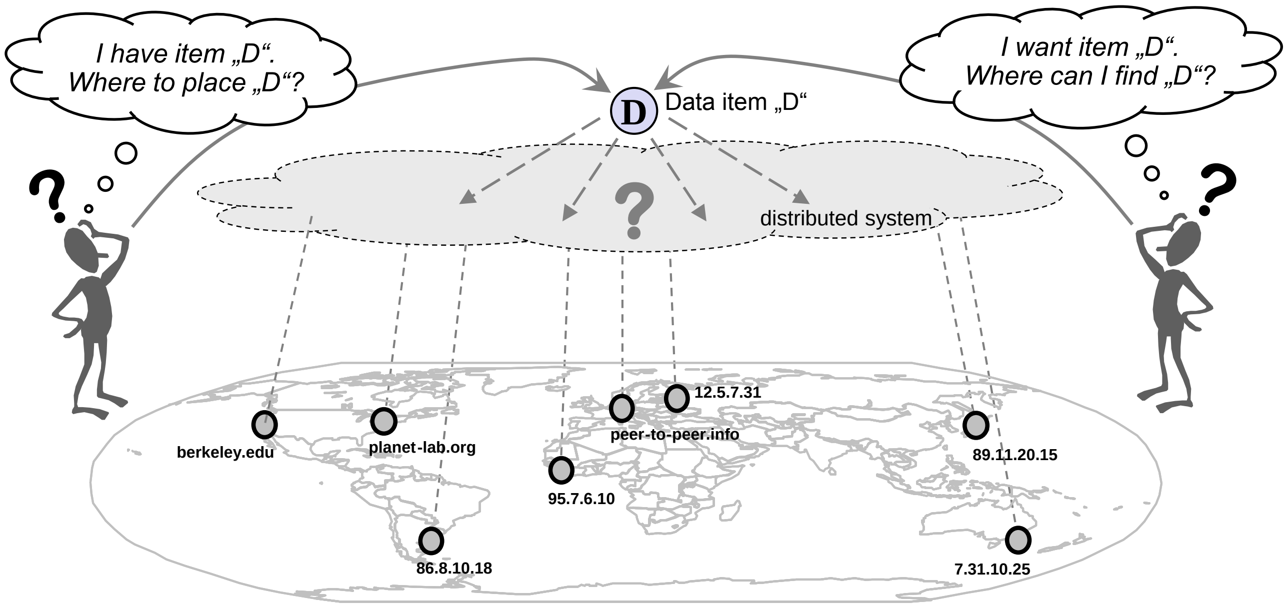
*Original slides for this lecture provided by K. Wehrle, S. Götz, S. Rieche (University of Tübingen)

0. Lecture Overview

1. Distributed Management and Retrieval of Data

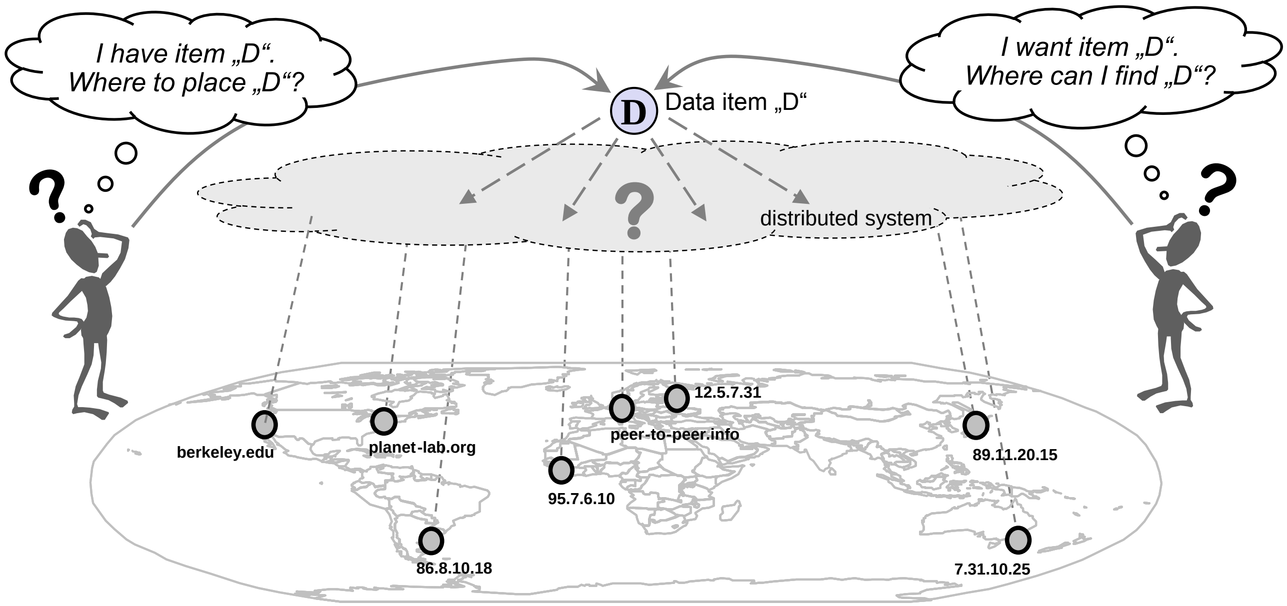
Comparison of Strategies:
Central Server, Flooding, Distributed Indexing

1. Distributed Management and Retrieval of Data



- **Essential challenge in (most) Peer-to-Peer systems?**
  - Location of a data item among systems distributed
    - Where shall the item be stored by the provider?
    - 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

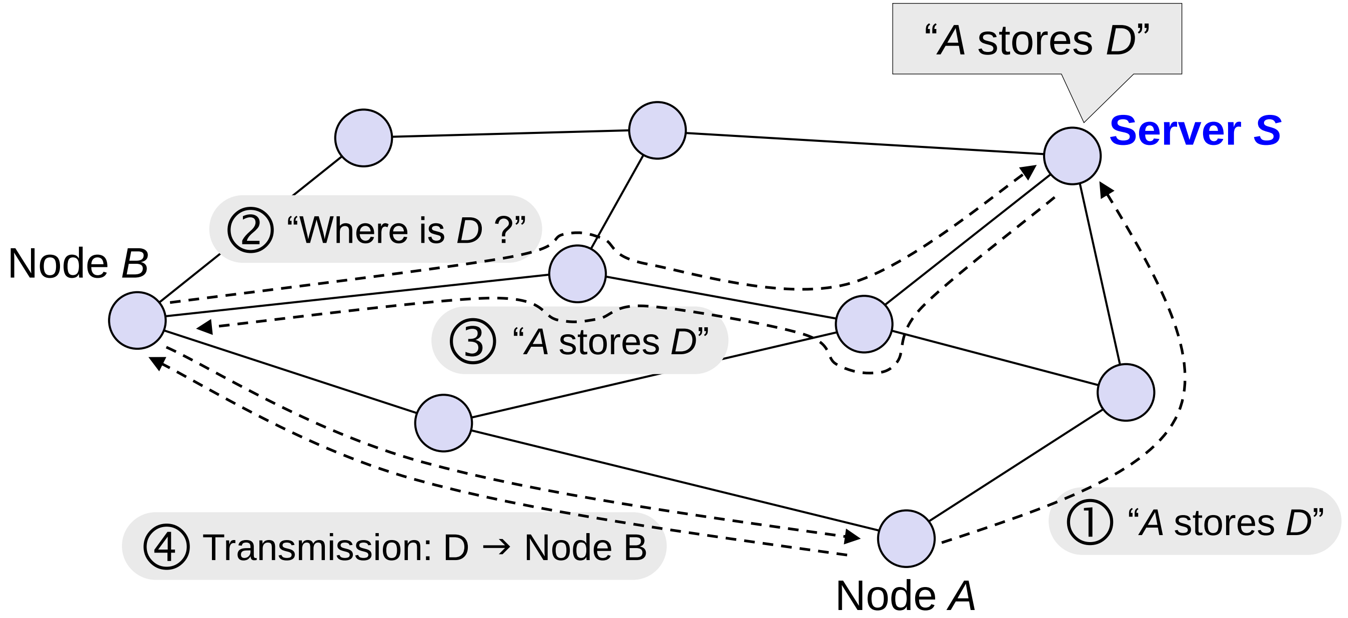
1.1. Comparison of Strategies for Data Retrieval



- **Strategies to store and retrieve data items in distributed systems**
  - Central server
  - Flooding search
  - Distributed indexing

1.2. Approach I: Central Server (1)

- **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



1.2. Approach I: Central Server (2)

- **Advantages**
  - Search complexity of O(1) – *"just ask the server"*
  - Complex and fuzzy queries are possible (i.e. not only exact match queries)
  - 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 ;-)
  - Central server not suitable for systems with massive numbers of users
- **But overall, …**
  - Best principle for small and simple applications!

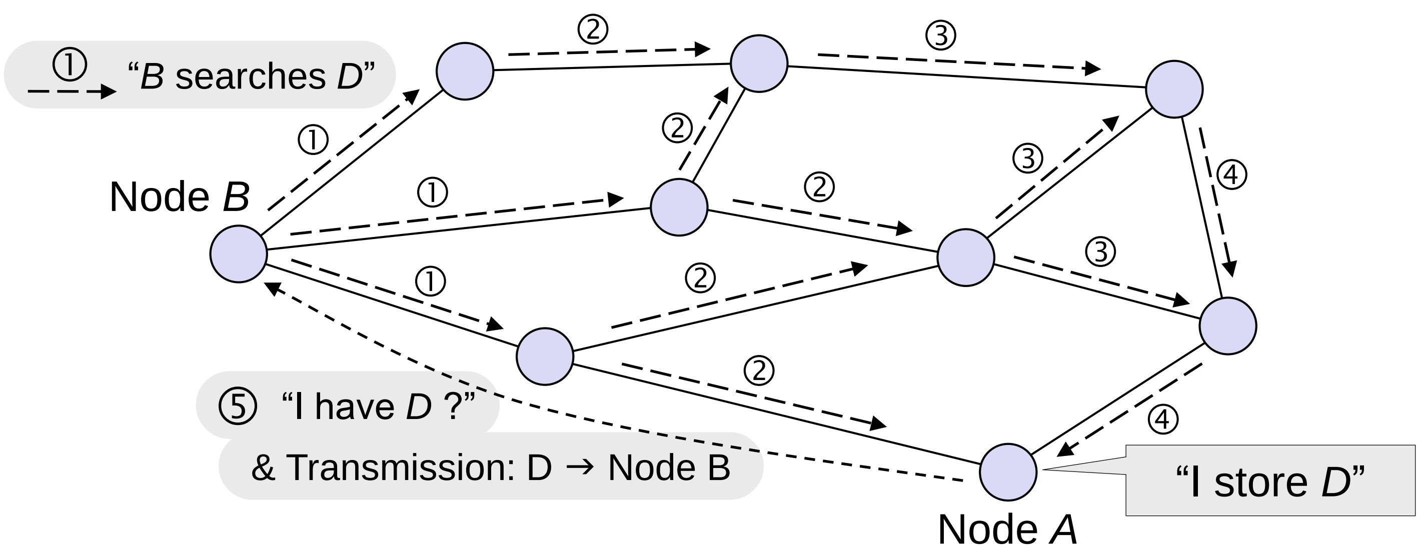
1.3. Approach II: Flooding Search (1)

- **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

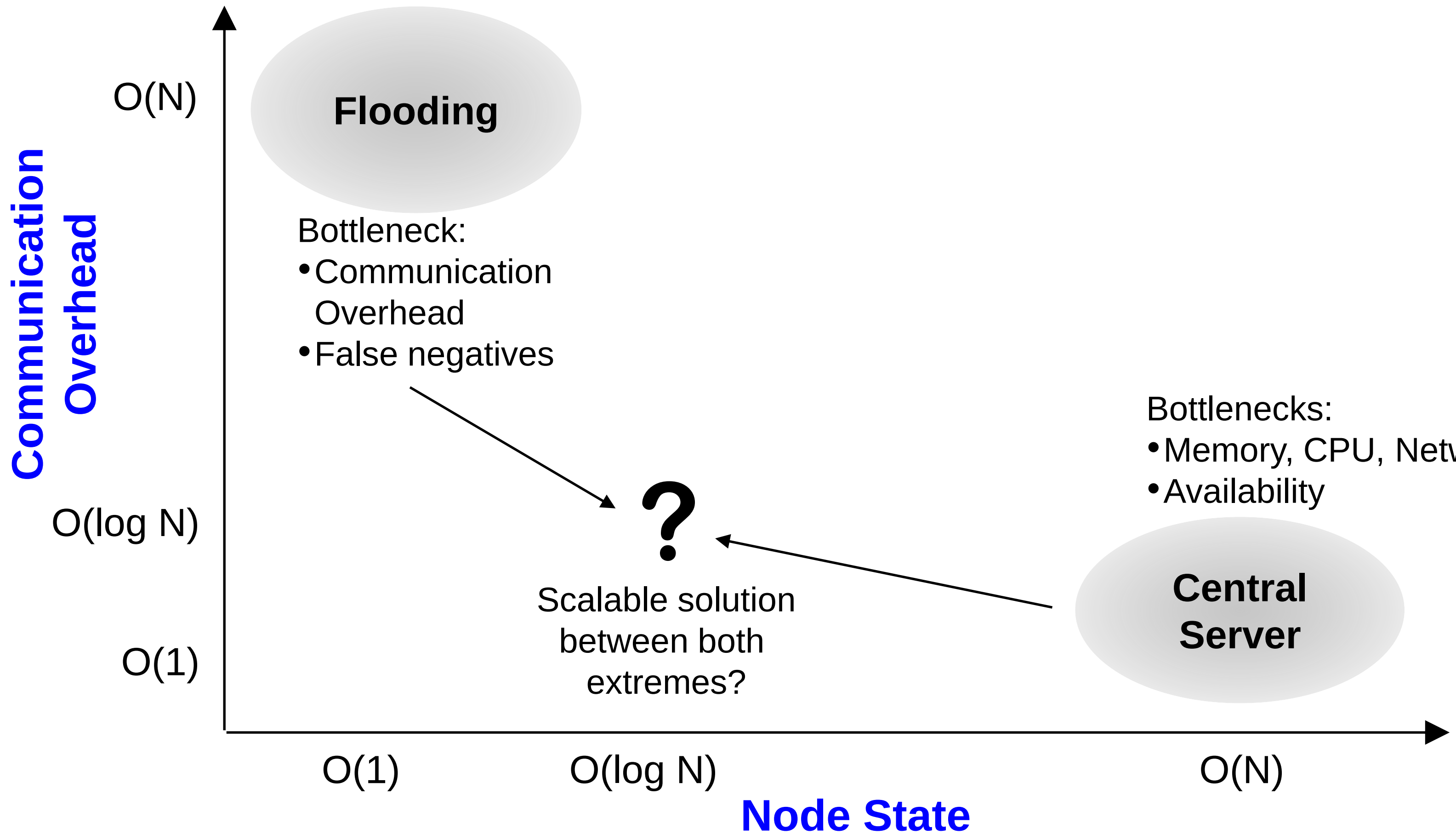
1.3. Approach II: Flooding Search (2)

- **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



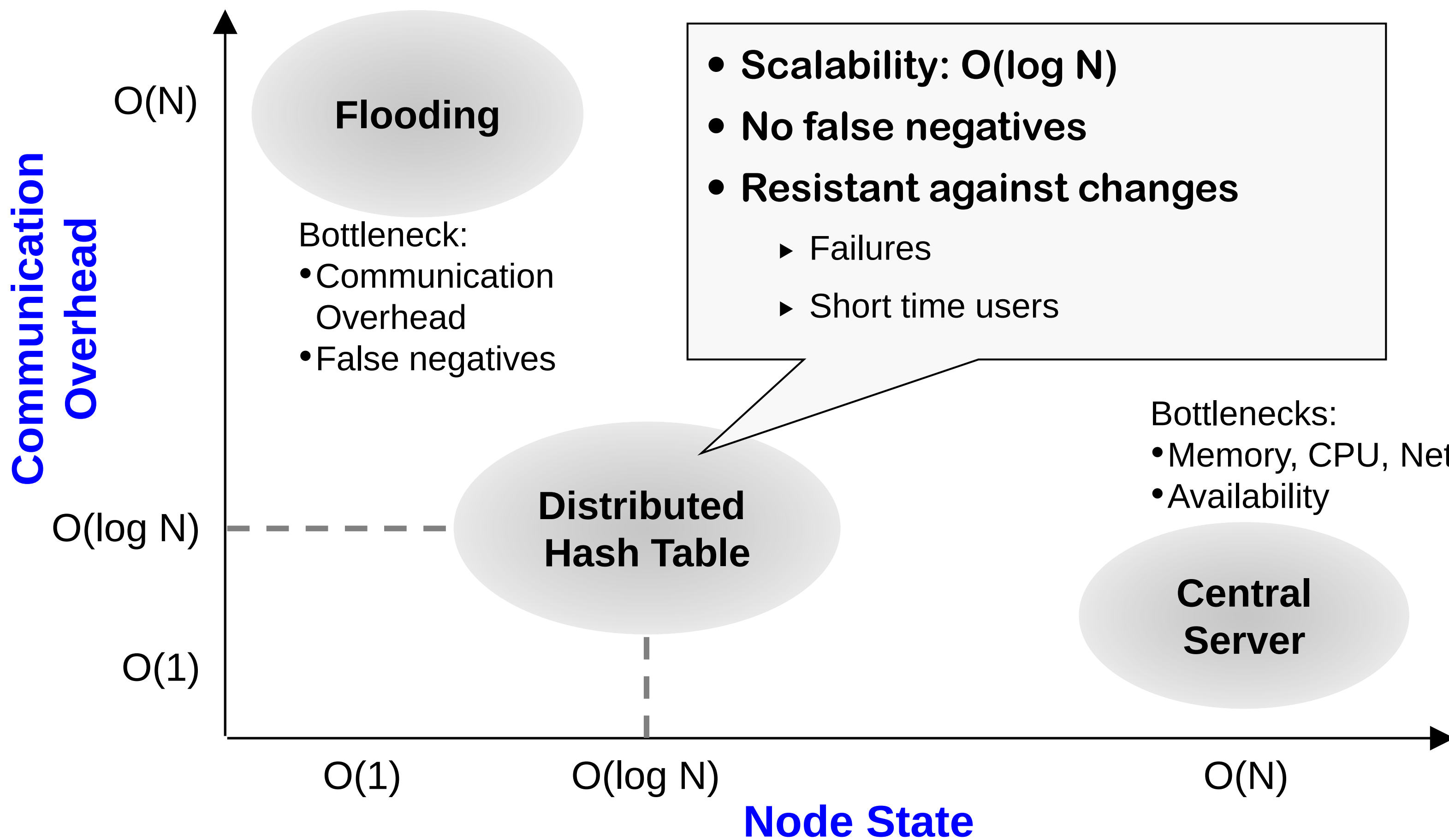
1.4. Motivation Distributed Indexing (1)

- **Communication overhead vs. node state**



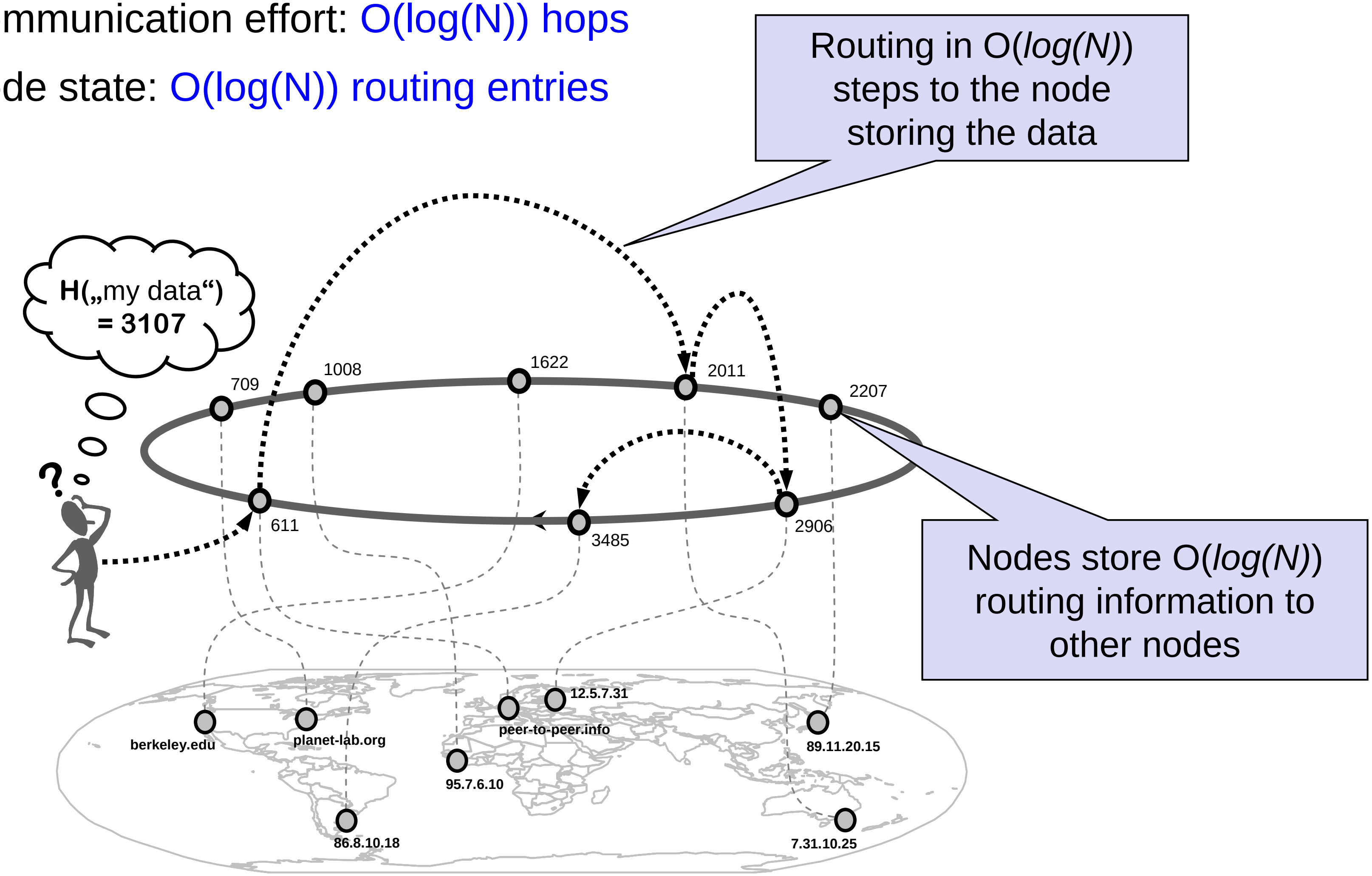
1.4. Motivation Distributed Indexing (2)

- **Communication overhead vs. node state**



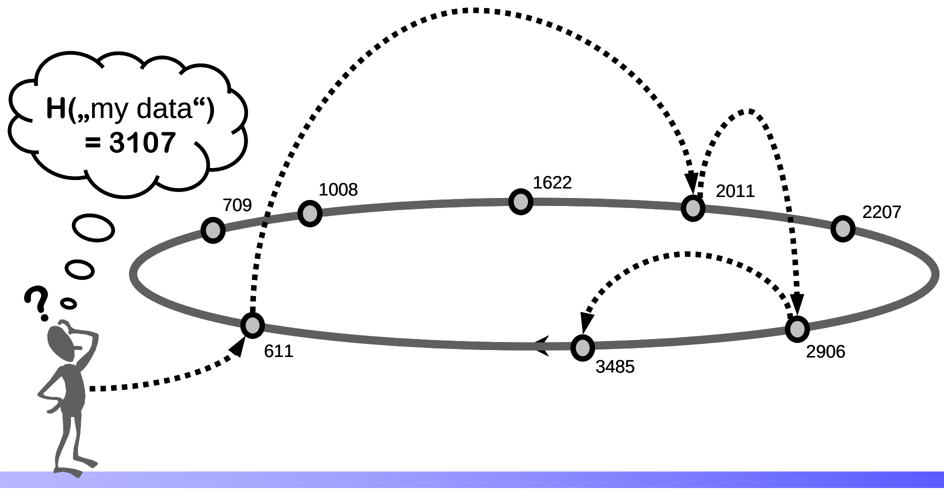
1.4. Distributed Indexing (1)

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



1.4. 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)



1.5. 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)	✗	✓	✓

2. Fundamentals of Distributed Hash Tables

Distributed Data Management
Addressing, Routing, Data Storage

- **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

2.1 Hash Table vs. Distributed Hash Table

- **Hash Table**
  - Bucket = hash(x) mod n
  - If n changes, remapping / bucket changes
  - N changes if capacity is reached
    - HashMap.class: static final float DEFAULT_LOAD_FACTOR = 0.75f;
  - Remapping is expensive in DHT!
    - **Why?**

- **Distributed Hash Table**
  - Consistent hashing → nodes responsible for **hash value intervals**
  - More peers = smaller responsible intervals

Sequence of operations

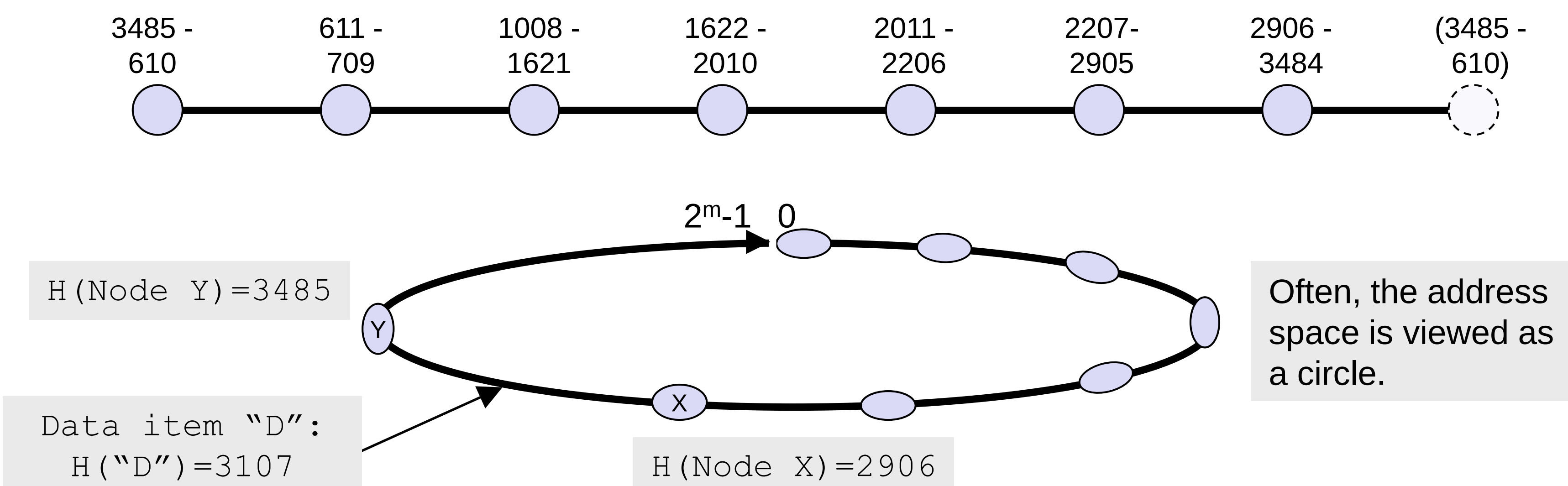
1. Mapping of nodes and data into same address space

- Peers and content are addressed using flat identifiers (IDs)
- 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

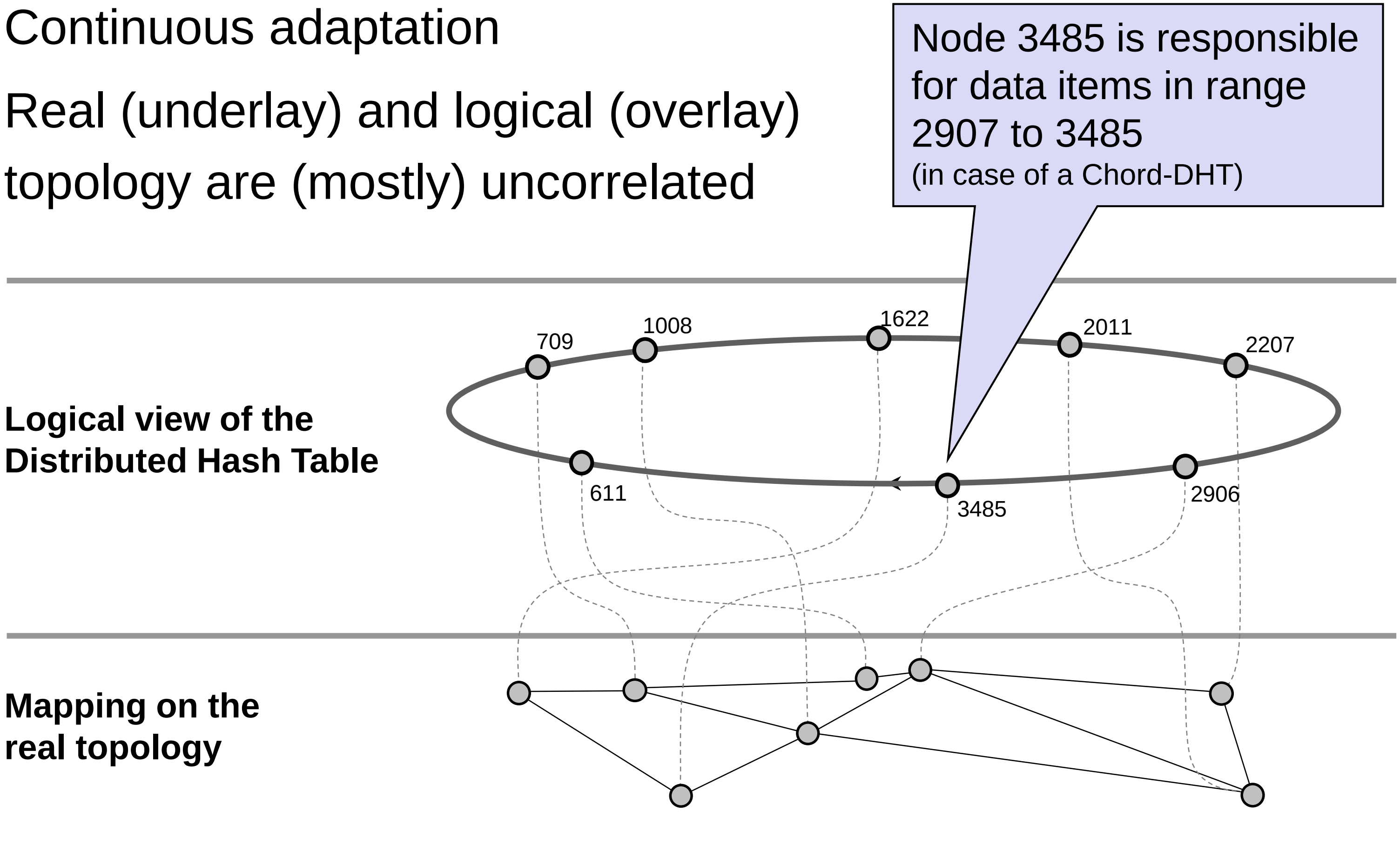
- Search for data = routing to the responsible node
  - Responsible node not necessarily known in advance
  - Node state O(n) vs. O(log n)

- **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*“) → 2313
  - Association of parts of address space to DHT nodes



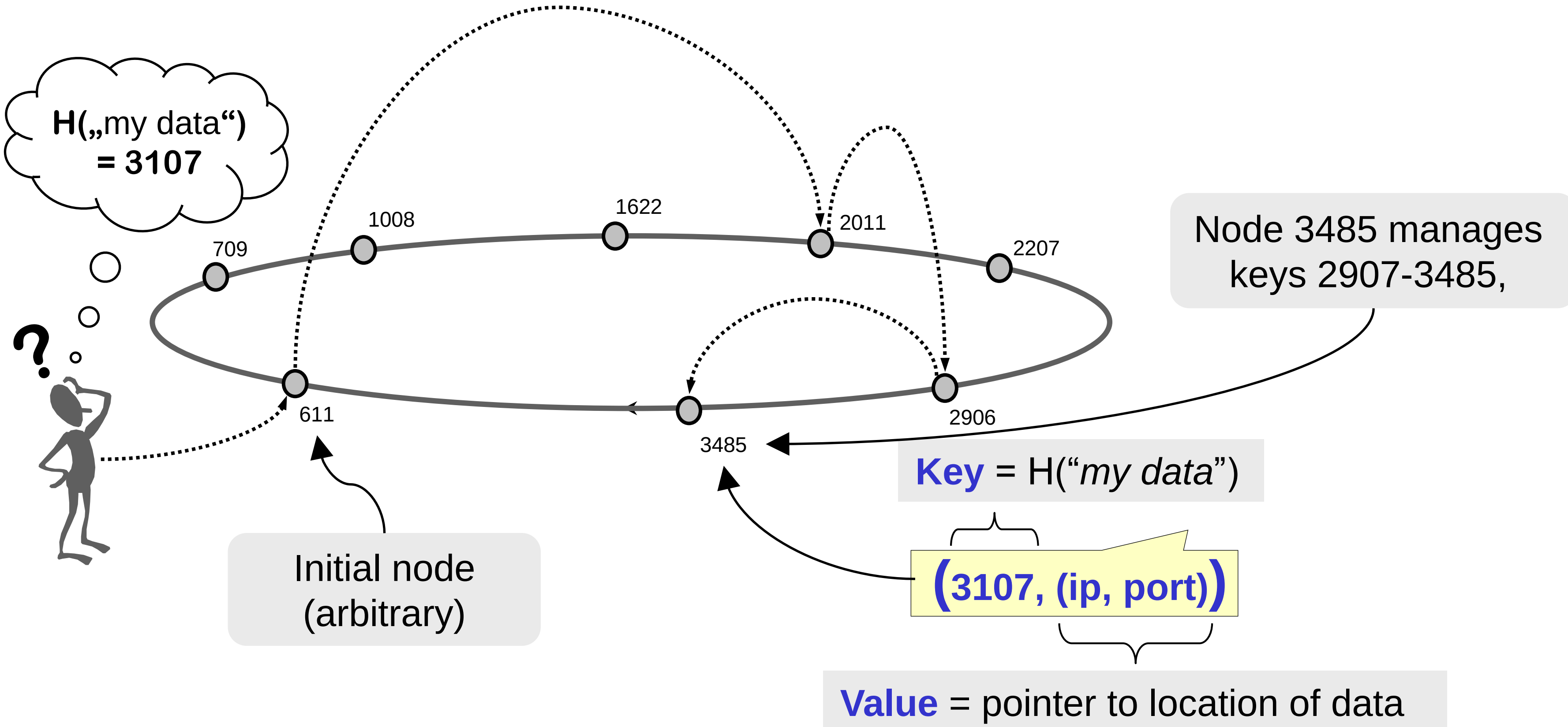
2.2. Association of Address Space with Nodes

- **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



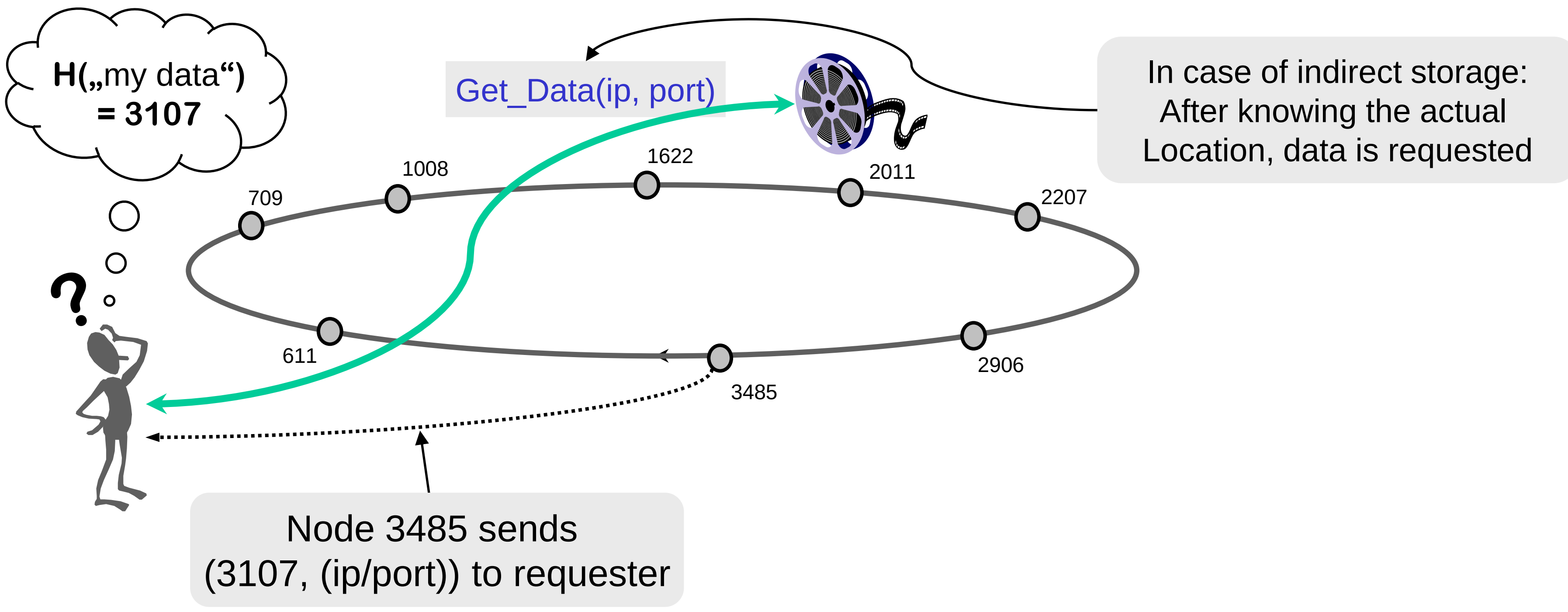
2.3. Step 2: Routing to a Data Item (2)

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



2.3. Step 2: Routing to a Data Item (3)

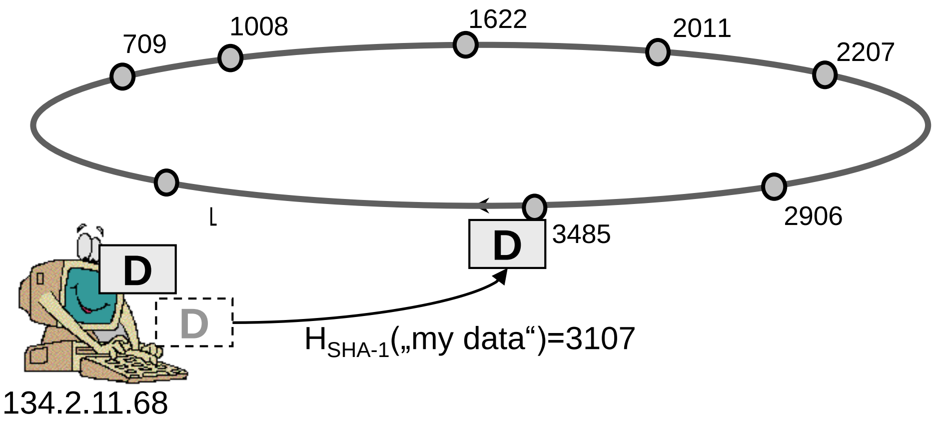
- **Getting the content**
  - K/V-pair is delivered to requester
  - Requester analyzes K/V-tuple
    (and downloads data from actual location – in case of indirect storage)



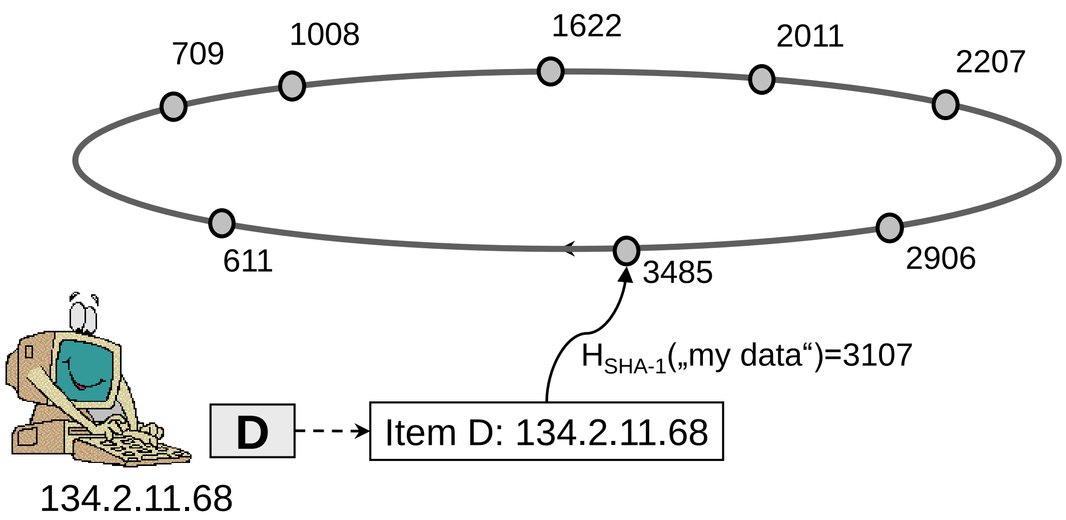
2.4. Association of Data with IDs – Direct Storage

- **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)



- **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

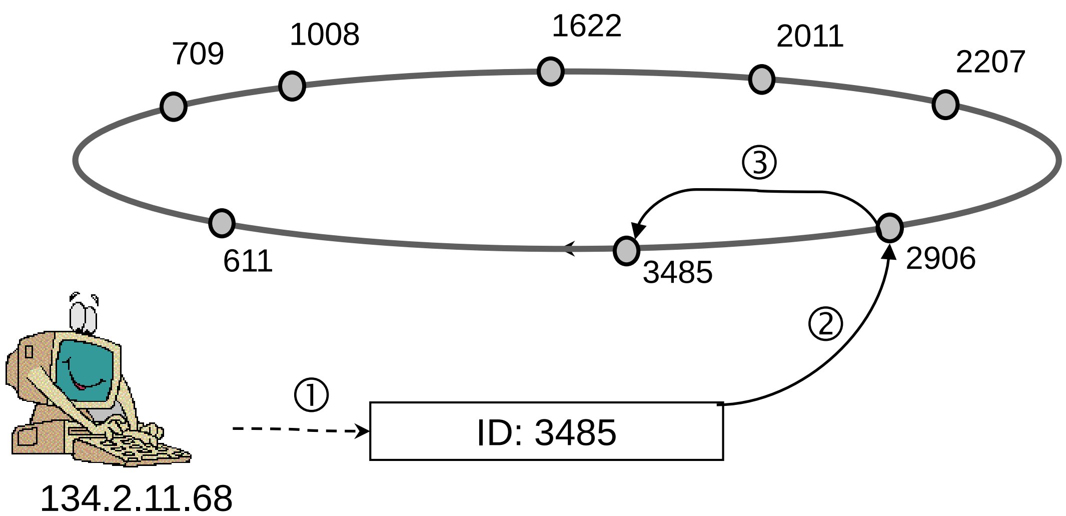


3. DHT Mechanisms

Node Arrival
Node Failure / Departure

3.1. Node Arrival

- **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. Binding into routing environment
  4. Copying of K/V-pairs of hash range (replication)



3.2. Node Failure / Departure

- **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

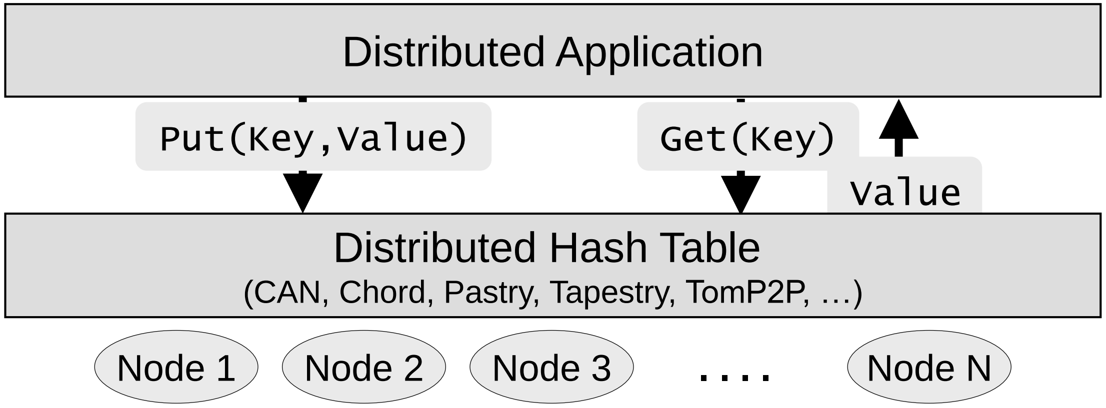
- **Departure of a node**
  - Copying of K/V pairs to corresponding nodes
    - Can be before or after unbinding
  - Friendly unbinding from routing environment
    - If unbinding is unfriendly, need for keep-alive messages

4. DHT Interfaces

Comparison: DHT vs. DNS

Summary: Properties of DHTs

- **Generic interface of distributed hash tables**
  - Provisioning of information
    - Put(key,value)
  - Requesting of information (search for content)
    - Get(key)
  - Reply
    - Value
- **DHT approaches are interchangeable (with respect to interface)**



4.1. Comparison: DHT vs. DNS (1)

- **Comparison DHT vs. DNS**
  - Traditional name services follow fixed mapping
    - DNS maps a logical node name to an IP address
  - DHTs offer flat / generic mapping of addresses
    - Not bound to particular applications or services
    - „*value*“ in *(key, value)* may be
      - an address
      - a document
      - or other data …

4.1. Comparison: DHT vs. DNS (2)

Domain Name System

- Mapping:
  Symbolic name →IP address
- Is built on a hierarchical structure with root servers
- Specialized to search for computer names and services

Distributed Hash Table

- Mapping: key → value
  can easily realize DNS
- Does not need a special server
- Can find data that are independently located of computers

4.2. Summary: Properties of DHTs

- **Use of routing information for efficient search for content**
- **Keys are evenly distributed across nodes of DHT**
  - No bottlenecks
  - A continuous increase in number of stored keys is admissible
  - 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

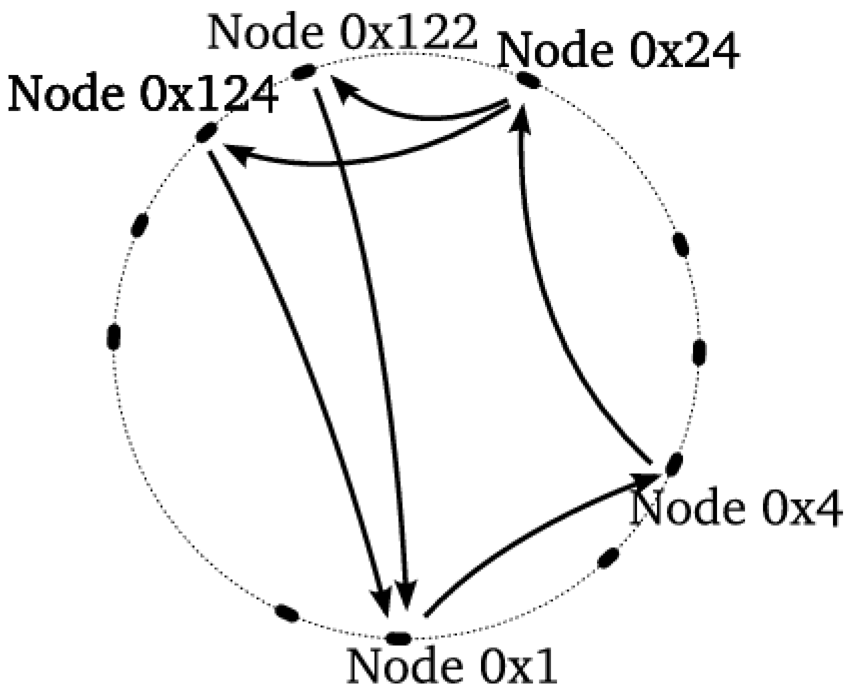
4.3. DHT Implementations

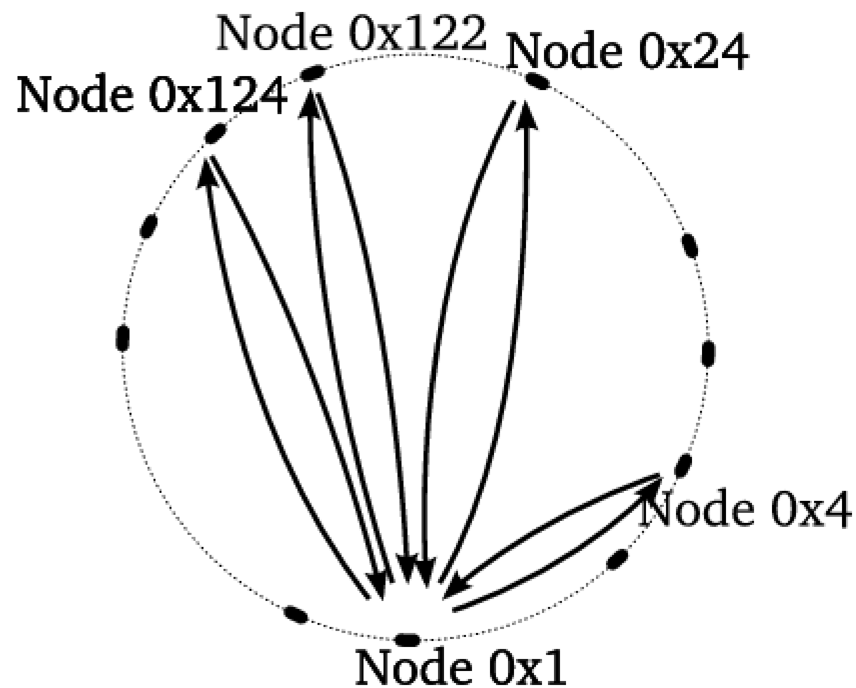
- **Examples of generic Distributed Hash Tables**
  - Pastry (freepastry, v2.1, 13.3.2009)
    Microsoft Research, Rice University
  - Chord (v1.0.5, 11.4.2008)
    UC Berkeley, MIT
  - Tapestry (chimera 1.20 ~2007)
    UC Berkeley
  - CAN (~2001)
    UC Berkeley, ICSI
  - P-Grid (v3.2.0_822, 25.11.2008)
    EPFL Lausanne
  - TomP2P (v5b4, 19.2.2015)
- **… and there are plenty of others …**

5. Routing and API

Routing Information, Routing Procedure, Node Addition and Failure, Common API, TomP2P

- Recursive routing vs. iterative routing





+ online status update

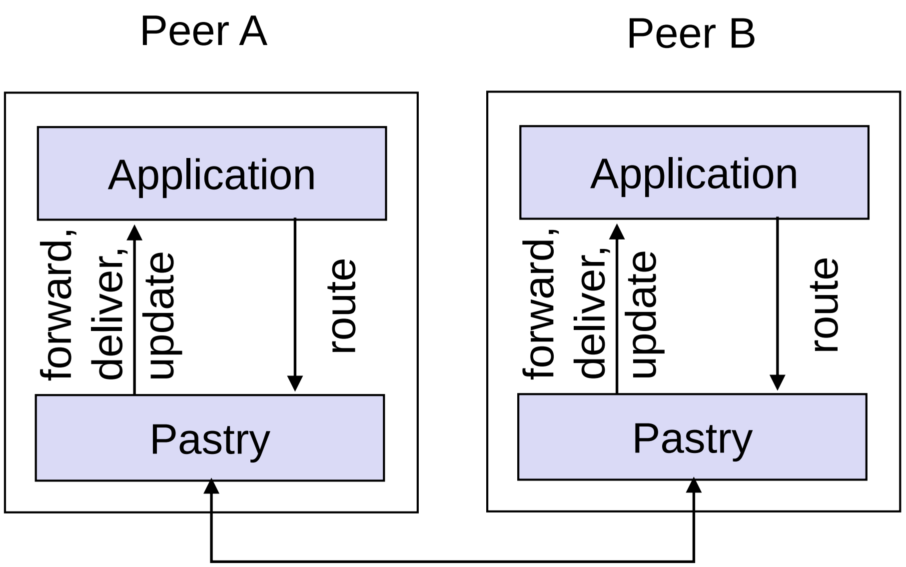
- no tracking of progress

+ control

- neighbor maintenance

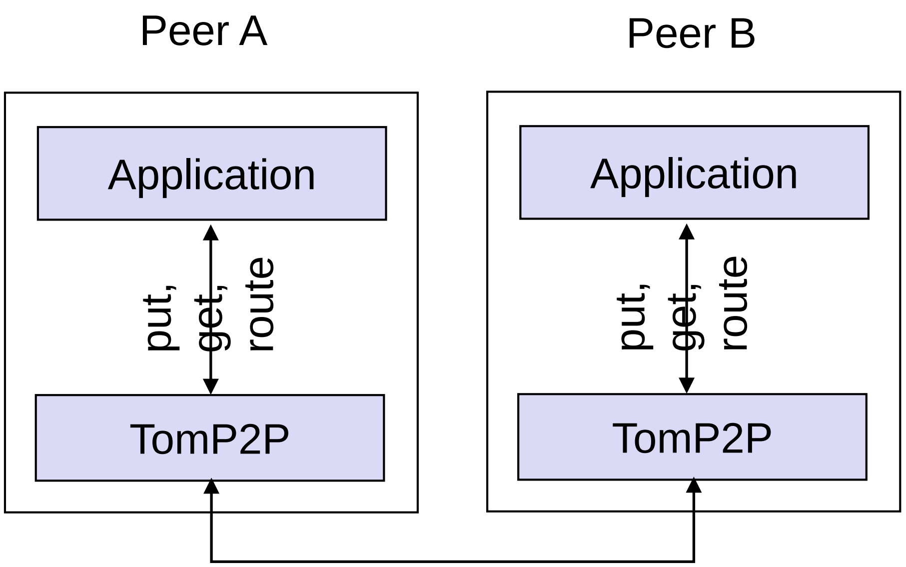
5.5. Common API for Structured P2P Overlays (rec.)

- **Standardized interface between applications and overlays using recursive routing**
  - Implemented by Pastry, others implement it too (Chord, Tapestry, CAN)
- **forward(RouteMessage)**
  - Called just before a message is forwarded
  - Message could be changed or dropped
- **deliver(Id, Message)**
  - Called when a message is received
- **update(NodeHandle, boolean)**
  - Called when a node's leafset changes (node joined or left)
- **route(Id, Message)**
  - Send a message to a node numerically closest to an Id (key)



5.5. Common API for Structured P2P Overlays (iterative)

- **Standardized interface between applications and overlays using iterative routing**
  - Implemented by TomP2P, others implement it too
- **bootstrap (ip)**
  - Called when a node wants to join the network (route)
- **put (id, message)**
  - To store (redundantly) data in the DHT
- **get (id)**
  - To get data from the DHT. May get more than one result
- **Additional features (add/discover/digest)**
  - Differs based on implementation



5.5. Routing and Bootstrap (iterative)

- **Routing essential**
  - get(key) = routing(key) + get(key)
  - put(key, value) = routing(key) + put(key, value)
  - bootstrap(key) = routing(key)
  - add(key, value), digest(key, …), … = routing(key) + add(key,value), digest(key, …), …
- **Routing to ID x**
  - Search local neighbors for closest peer to x → will return v, w
  - Search on peer v for x → will return w, y, …
  - Search on peer w for x → will return y, z, …
  - …
  - Found closest peer(s) → do operations

5.5. Node Addition and Failure (iterative)

- **Node state (routing information) has to be maintained efficiently**
- **Node Arrivals**
  - New node with nodeId X asks nearby node A (bootstrap node) to route to key X
  - As soon as close node is found, replication may kick in
- **Node Failures**
  - nodes periodically exchange keepalive messages
  - If a node does not respond for a time T, it is declared dead
  - Replace node

6. Kademlia

Introduction, Idea, Examples

Concept

6. 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
    - Kad network: eMule and others
    - BitTorrent (tracker)
  - Java-based generic implementations
    - TomP2P
    - Openkad, JDHT

6. Kademlia

- **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
  - Distance is XOR
- **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

6. Kademlia Example

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

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

  - Search for 3, ask 0, neighbors of 0

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

  - Ask 2, neighbors of 2

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

  - Ask 2, 2 tell that there is no closer node, 2 is the closest one (2 xor 3 =1)