# Structured Peer-to-Peer Networks

Wolf-Tilo Balke and Wolf Siberski  
31.10.2006

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

## Distributed Hash Tables

<table>
<thead>
<tr>
<th>1. Distributed Management and Retrieval of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Comparison of strategies for data retrieval</td>
</tr>
<tr>
<td>2. Central server</td>
</tr>
<tr>
<td>3. Flooding search</td>
</tr>
<tr>
<td>4. Distributed indexing</td>
</tr>
<tr>
<td>5. Comparison of lookup concepts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. Fundamentals of Distributed Hash Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Distributed management of data</td>
</tr>
<tr>
<td>2. Addressing in Distributed Hash Tables</td>
</tr>
<tr>
<td>3. Routing</td>
</tr>
<tr>
<td>4. Data Storage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. DHT Mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Node Arrival</td>
</tr>
<tr>
<td>2. Node Failure / Departure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. DHT Interfaces</th>
</tr>
</thead>
</table>

| 5. Example: Chord                            |

---

---
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

Comparison of Strategies for Data Retrieval

• Strategies to store and retrieve data items in distributed systems
  ▶ Central server
  ▶ Flooding search
  ▶ Distributed indexing
**Approach I: 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

**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 law suits ;-
- Non-linear increasing implementation and maintenance cost (in particular for achieving high availability and scalability)
- Central server not suitable for systems with massive numbers of users

**But overall, …**

- Best principle for small and simple applications!
Approach II: Flooding Search

- **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
  - Content is only stored in the node providing it

- **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

- **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. Nodes forward request to further nodes (breadth-first search / flooding)
    3. Node A (provider of item D) sends D to requesting node B

---
Motivation Distributed Indexing – I

- Communication overhead vs. node state

![Diagram showing communication overhead vs. node state with O(N), O(log N), and O(1) scaling for flooding and central server.]

Motivation Distributed Indexing – II

- Communication overhead vs. node state

![Diagram showing communication overhead vs. node state with O(N), O(log N), and O(1) scaling for distributed hash table and central server.]

- Scalability: O(log N)
- No false negatives
- Resistant against changes
  - Failures, Attacks
  - Short time users
Distributed Indexing

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

Routing in $O(\log(N))$ steps to the node storing the data

Nodes store $O(\log(N))$ routing information to other nodes

Distributed Indexing

- **Approach of distributed indexing schemes**
  - Data and nodes are mapped into same address space
  - Intermediate nodes maintain routing information to target nodes
    - Efficient forwarding to „destination“ (content – not location)
    - Definitive statement of existence of content

- **Problems**
  - Maintenance of routing information required
  - Fuzzy queries not primarily supported (e.g., wildcard searches)
Distributed Hash Tables

1. Distributed Management and Retrieval of Data
   1. Comparison of strategies for data retrieval
   2. Central server
   3. Flooding search
   4. Distributed indexing
   5. Comparison of lookup concepts

2. Fundamentals of Distributed Hash Tables
   1. Distributed management of data
   2. Addressing in Distributed Hash Tables
   3. Routing
   4. Data Storage

3. DHT Mechanisms
   1. Node Arrival
   2. Node Failure / Departure

4. DHT Interfaces

5. Example: Chord

---

Fundamentals of Distributed Hash Tables I

- Characteristics of Hash Tables
  - Basic idea: keys are mapped via a common function to smaller fingerprints (hashes)
    - Every number defines a position in an array (bucket)
    - Keys mapped onto the same hash are put into the same bucket
    - Look-up works by hashing the query and searching the respective bucket
  - Hash Function
    - Poor choice leads to clustering, i.e. probability of keys mapping to the same hash bucket (collision) is great and the performance degrades
    - Good choices should be easy to compute, result in few collisions, and show a uniform distribution of hash values
  - Hash Tables
    - provide constant-time $O(1)$ lookup on average, regardless of the number of items in the table
Fundamentals of Distributed Hash Tables II

**Challenges for designing Distributed Hash Tables**

- **Desired Characteristics**
  - Flexibility
  - 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 Management of Data

**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
     - Deterministic statement about availability of data
Addressing in Distributed Hash Tables

- **Step 1: Mapping of content/nodes into linear space**
  - Usually: $0, \ldots, 2^m - 1 >>$ number of objects to be stored
  - Mapping of data and nodes into an address space (with hash function)
    - E.g., Hash(String) mod $2^m$: $H(\text{"my data"}) \rightarrow 2313$
  - Association of parts of address space to DHT nodes

```
3485 - 610  611 - 709  1008 - 1621  1622 - 2010  2011 - 2206  2207 - 2905  2906 - 3484  (3485 - 610)
```

- **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

```
Logical view of the Distributed Hash Table
```

```
Mapping on the real topology
```

Node 3485 is responsible for data items in range 2907 to 3485 (in case of a Chord-DHT)
Step 2: Routing to a Data Item

- Step 2: Locating the data (content-based routing)

- Goal: Small and scalable effort
  - $O(1)$ with centralized hash table
    - But: Management of a centralized hash table is very costly (server!)
  - Minimum overhead with distributed hash tables
    - $O(\log N)$: DHT hops to locate object
    - $O(\log N)$: number of keys and routing information per node ($N = \#$ nodes)

---

Step 2: Routing to a Data Item

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

![Diagram showing routing process](image_url)
Step 2: 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)

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 (<1KB)
### Association of Data with IDs – Indirect 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

### Distributed Hash Tables

1. **Distributed Management and Retrieval of Data**
   1. Comparison of strategies for data retrieval
   2. Central server
   3. Flooding search
   4. Distributed indexing
   5. Comparison of lookup concepts
2. **Fundamentals of Distributed Hash Tables**
   1. Distributed management of data
   2. Addressing in Distributed Hash Tables
   3. Routing
   4. Data Storage
3. **DHT Mechanisms**
   1. Node Arrival
   2. Node Failure / Departure
4. **DHT Interfaces**
5. **Example: Chord**
**Node Arrival**

- **Joining of a new node**
  1. Calculation of node ID
  2. New node contacts DHT via arbitrary node
  3. Assignment of a particular hash range
  4. Copying of K/V-pairs of hash range (usually with redundancy)
  5. Binding into routing environment

**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**
  - Partitioning of hash range to neighbor nodes
  - Copying of K/V pairs to corresponding nodes
  - Unbinding from routing environment
Distributed Hash Tables

1. Distributed Management and Retrieval of Data
   1. Comparison of strategies for data retrieval
   2. Central server
   3. Flooding search
   4. Distributed indexing
   5. Comparison of lookup concepts

2. Fundamentals of Distributed Hash Tables
   1. Distributed management of data
   2. Addressing in Distributed Hash Tables
   3. Routing
   4. Data Storage

3. DHT Mechanisms
   1. Node Arrival
   2. Node Failure / Departure

4. DHT Interfaces

5. Example: Chord

### DHT Interfaces

- **Generic interface of distributed hash tables**
  - Provisioning of information
    - Publish(key,value)
  - Requesting of information (search for content)
    - Lookup(key)
  - Reply
    - value

- **DHT approaches are interchangeable (with respect to interface)**

![Diagram of Distributed Hash Table](image)
### Comparison: DHT vs. DNS

#### 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 ...

---

#### Domain Name System vs. Distributed Hash Table

<table>
<thead>
<tr>
<th>Domain Name System</th>
<th>Distributed Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping: Symbolic name → IP address</td>
<td>Mapping: key → value can easily realize DNS</td>
</tr>
<tr>
<td>Is built on a hierarchical structure with root servers</td>
<td>Does not need a special server</td>
</tr>
<tr>
<td>Names refer to administrative domains</td>
<td>Does not require special name space</td>
</tr>
<tr>
<td>Specialized to search for computer names and services</td>
<td>Can find data that are independently located of computers</td>
</tr>
</tbody>
</table>
Conclusions

- 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 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

Next …

- Specific examples of Distributed Hash Tables
  - Chord
    UC Berkeley, MIT
  - Pastry
    Microsoft Research, Rice University
  - CAN
    UC Berkeley, ICSI
  - P-Grid
    EPFL Lausanne

- … and there are plenty of others: Kademlia, Symphony, Viceroy, …
Distributed Hash Tables

1. Distributed Management and Retrieval of Data
   1. Comparison of strategies for data retrieval
   2. Central server
   3. Flooding search
   4. Distributed indexing
   5. Comparison of lookup concepts

2. Fundamentals of Distributed Hash Tables
   1. Distributed management of data
   2. Addressing in Distributed Hash Tables
   3. Routing
   4. Data Storage

3. DHT Mechanisms
   1. Node Arrival
   2. Node Failure / Departure

4. DHT Interfaces

5. Example: Chord

---

Chord

Ion Stoica
Robert Morris
David Karger
M. Frans Kaashoek
Hari Balakrishnan
(2001)
Chord: Overview

- Early and successful algorithm
- Simple & elegant
  - easy to understand and implement
  - many improvements and optimizations exist
  - Ion Stoica et al. in 2001
- Main responsibilities:
  - Routing
    - Flat logical address space: l-bit identifiers instead of IP addresses
    - Efficient routing in large systems: log(N) hops with N total nodes
  - Self-organization
    - Handle node arrival, departure, and failure

Chord: Topology

- Hash-table storage
  - put (key, value) inserts data to Chord
  - Value = get (key) retrieves data from Chord
- Identifiers
  - Derived from hash function
    - E.g. SHA-1, 160-bit output → 0 <= identifier < 2^160
  - Key associated with data item
    - E.g. key = sha-1(value)
  - ID associated with host
    - E.g. id = sha-1 (IP address, port)
Chord: Topology

- Keys and IDs on ring, i.e., all arithmetic modulo $2^{160}$
- (key, value) pairs managed by clockwise next node: successor

![Chord Ring Diagram]

**Chord: Topology**

- Topology determined by links between nodes
  - Link: knowledge about another node
  - Stored in routing table on each node
- Simplest topology: circular linked list
  - Each node has link to clockwise next node
Chord: Routing

- **Primitive routing:**
  - Forward query for key \( x \) until successor(\( x \)) is found
  - Return result to source of query

- **Pros:**
  - Simple
  - Little node state

- **Cons:**
  - Poor lookup efficiency: \( O(1/2 * N) \) hops on average (with \( N \) nodes)
  - Node failure breaks circle

Chord: Routing

- **Advanced routing:**
  - Store links to \( z \) next neighbors
  - Forward queries for \( k \) to farthest known predecessor of \( k \)
  - For \( z = N \): fully meshed routing system
    - Lookup efficiency: \( O(1) \)
    - Per-node state: \( O(N) \)
  - Still poor scalability

- **Scalable routing:**
  - Linear routing progress scales poorly
  - Mix of short- and long-distance links required:
    - Accurate routing in node’s vicinity
    - Fast routing progress over large distances
    - Bounded number of links per node
Chord: Routing

- Chord’s routing table: *finger table*
  - Stores log(N) links per node
  - Covers exponentially increasing distances:
    - Node n: entry i points to successor(n + 2^i) (i-th finger)

- Chord’s routing algorithm:
  - Each node n forwards query for key k clockwise
    - To farthest finger preceding k
    - Until n = predecessor(k) and successor(n) = successor(k)
    - Return successor(n) to source of query
Chord: Self-Organization

- Handle changing network environment
  - Failure of nodes
  - Network failures
  - Arrival of new nodes
  - Departure of participating nodes
- Maintain consistent system state for routing
  - Keep routing information up to date
    - Routing correctness depends on correct successor information
    - Routing efficiency depends on correct finger tables
  - Failure tolerance required for all operations

Chord: Failure Tolerance: Storage

- Layered design
  - Chord DHT mainly responsible for routing
  - Data storage managed by application
    - persistence
    - consistency
    - fairness
- Chord soft-state approach:
  - Nodes delete (key, value) pairs after timeout
  - Applications need to refresh (key, value) pairs periodically
  - Worst case: data unavailable for refresh interval after node failure
Chord: Failure Tolerance: Routing

- **Finger failures during routing**
  - query cannot be forwarded to finger
  - forward to previous finger (do not overshoot destination node)
  - trigger repair mechanism: replace finger with its successor

- **Active finger maintenance**
  - periodically check liveness of fingers
  - replace with correct nodes on failures
  - trade-off: maintenance traffic vs. correctness & timeliness

---

Chord: Failure Tolerance: Routing

- **Successor failure during routing**
  - Last step of routing can return failed node to source of query
    -> all queries for successor fail
  - Store n successors in *successor list*
    - successor[0] fails -> use successor[1] etc.
    - routing fails only if n consecutive nodes fail simultaneously

- **Active maintenance of successor list**
  - periodic checks similar to finger table maintenance
  - crucial for correct routing
Chord: Node Arrival

- New node picks ID
- Contact existing node
- Construct finger table via standard routing/lookup()
- Retrieve (key, value) pairs from successor

---

Examples for choosing new node IDs
- random ID: equal distribution assumed but not guaranteed
- hash IP address & port
- place new nodes based on
  - load on existing nodes
  - geographic location, etc.

Retrieval of existing node IDs
- Controlled flooding
- DNS aliases
- Published through web
- etc.

---

*ID = \$\text{rand}() = 6*$
Chord: Node Arrival

- **Construction of finger table**
  - iterate over finger table rows
  - for each row: query entry point for successor
  - standard Chord routing on entry point

- **Construction of successor list**
  - add immediate successor from finger table
  - request successor list from successor

Chord: Node Departure

- **Deliberate node departure**
  - clean shutdown instead of failure

- **For simplicity: treat as failure**
  - system already failure tolerant
  - soft state: automatic state restoration
  - state is lost briefly
  - invalid finger table entries: reduced routing efficiency

- **For efficiency: handle explicitly**
  - notification by departing node to
    - successor, predecessor, nodes at finger distances
  - copy (key, value) pairs before shutdown
Chord: Summary

- **Complexity**
  - Messages per lookup: $O(\log N)$
  - Memory per node: $O(\log N)$
  - Messages per management action (join/leave/fail): $O(\log^2 N)$

- **Advantages**
  - Theoretical models and proofs about complexity
  - Simple & flexible

- **Disadvantages**
  - No notion of node proximity and proximity-based routing optimizations
  - Chord rings may become disjoint in realistic settings

- **Many improvements published**
  - e.g. proximity, bi-directional links, load balancing, etc.

---

The Architectures of 1st and 2nd Gen. P2P

<table>
<thead>
<tr>
<th>Client-Server</th>
<th>Peer-to-Peer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Server is the central entity and only provider of service and content. 2. Network managed by the Server 3. Client as the lower performance system</td>
<td>1. Resources are shared between the peers 2. Resources can be accessed directly from other peers 3. Peer as provider and requester (Servent concept)</td>
</tr>
<tr>
<td>Example: WWW</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Centralized P2P</th>
<th>Pure P2P</th>
<th>Hybrid P2P</th>
<th>DHT-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. All features of Peer-to-Peer included 2. Central entity is necessary to provide the service 3. Central entity is some kind of index/group database 4. Example: Napster</td>
<td>1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. Example: Gnutella 0.4, JXTA</td>
<td>1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. Dynamic central entities 4. Example: Gnutella 0.6, JXTA</td>
<td>1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. No central entities 4. Connections in the overlay are &quot;fixed&quot; 5. Example: Chord, CAN</td>
</tr>
</tbody>
</table>

1st Gen. 2nd Gen.
Reminder: Distributed Indexing

- Communication overhead vs. node state

**Comparison of Lookup Concepts**

<table>
<thead>
<tr>
<th>System</th>
<th>Per Node State</th>
<th>Communication Overhead</th>
<th>Fuzzy Queries</th>
<th>No false negatives</th>
<th>Robustness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central Server</td>
<td>O(N)</td>
<td>O(1)</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Flooding Search</td>
<td>O(1)</td>
<td>O(N²)</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed Hash Tables</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>