Structured Peer-to-Peer Networks

- The P2P Scaling Problem
- Unstructured P2P Revisited
- Distributed Indexing
- Fundamentals of Distributed Hash Tables
- DHT Algorithms
  - Chord
  - Pastry
  - Can
- Programming a DHT

Graphics repeatedly taken from:
R. Steinmetz, K. Wehrle: *Peer-to-Peer Systems and Applications*, Springer LNCS 3485, 2005
Demands of P2P Systems

- Instant Deployment
  - Independent of infrastructural provisions
- Flexibility
  - Seamless adaptation to changing member requirements
- Reliability
  - Robustness against node or infrastructure failures
- Scalability
  - Resources per node do not (significantly) increase as the P2P network grows
The Challenge in Peer-to-Peer Systems

- Location of resources (data items) distributed among systems
  - Where shall the item be stored by the provider?
  - How does a requester find the actual location of an item?
- Scalability: limit the complexity for communication and storage
- Robustness and resilience in case of faults and frequent changes
Unstructured P2P Revisited

Basically two approaches:

- **Centralized**
  - Simple, flexible searches at server (O(1))
  - Single point of failure, O(N) node states at server

- **Decentralized Flooding**
  - Fault tolerant, O(1) node states
  - Communication overhead \( \geq O(N^2) \), search may fail

But:

- No reference structure between nodes imposed
Unstructured P2P: Complexities

Flooding

- Bottleneck: Communication Overhead
- False negatives

Central Server

- Bottlenecks: Memory, CPU, Network
- Availability

Scalable solution between both extremes?

Communication Overhead

- $O(1)$
- $O(\log N)$
- $O(N)$

O(1) O(\log N) O(N)
Idea: Distributed Indexing

- Initial ideas from distributed shared memories (1987 ff.)
- Nodes are structured according to some address space
- Data is mapped into the same address space
- Intermediate nodes maintain routing information to target nodes
  - Efficient forwarding to „destination“ (content – not location)
  - Definitive statement about existence of content

\[ H(\text{"my data"}) = 3107 \]
Scalability of Distributed Indexing

- 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

$H(\text{"my data"}) = 3107$
Distributed Indexing: Complexities

Flooding
- Bottleneck: Communication Overhead, False negatives
- Scalability: $O(\log N)$
- No false negatives
- Resistant against changes
  - Failures, Attacks
  - Short time users

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

Central Server
- Bottlenecks: Memory, CPU, Network
- Availability

Node State
- $O(1)$
- $O(\log N)$
- $O(N)$

Communication Overhead
- $O(1)$
- $O(\log N)$
- $O(N)$
Fundamentals of Distributed Hash Tables

- Desired Characteristics:
  - Flexibility, Reliability, Scalability

- Challenges for designing DHTs
  - 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
  - Maintenance of routing information
Distributed Management of Data

1. Mapping of nodes and data into same address space
   - Peers and content are addressed using flat identifiers (IDs)
   - 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: $0, ..., 2^m-1 \gg$ number of objects to be stored
- Mapping of data and nodes into an address space (with hash function)
  - E.g., $\text{Hash(String)} \mod 2^m$: $H(\text{"my data"}) \rightarrow 2313$
- Association of parts of address space to DHT nodes

Often, the address space is viewed as a circle.
Mapping Address Space to 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 so far uncorrelated

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

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

Goal: Small and scalable effort

- O(1) with centralized hash table
- 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)
Routing to a Data Item (2)

- Routing to a Key-Value-pair
  - Start lookup at arbitrary node of DHT
  - Routing to requested data item (key) recursively according to node tables

\[ H(\text{"my data"}) = 3107 \]

Initial node (arbitrary)

Node 3485 manages keys 2907-3485,

Key = \( H(\text{"my data"}) \)

\[ (3107, (ip, port)) \]

Value = pointer to location of data
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)

\[
H(“my data“) = 3107
\]

Get\_Data(ip, port)

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

In case of indirect storage: After knowing the actual Location, data is requested
Data Storage

- **Direct storage**
  - Content is stored in responsible node for $H(\text{"my data"})$
  - Inflexible for large content – o.k. for small data (<1KB)

- **Indirect storage**
  - Nodes in a DHT store tuples like (key,value)
    - Key = Hash(\text{"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
Dynamic of a DHT: Node Arrival

Bootstrapping/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 (of overlay)
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
DHT Algorithms

- **Lookup algorithm** for nearby objects (Plaxton et al 1997)
  - Before P2P ... later used in Tapestry

- **Chord** (Stoica et al 2001)
  - Straight forward 1-dim. DHT

- **Pastry** (Rowstron & Druschel 2001)
  - Proximity neighbour selection

- **CAN** (Ratnasamy et al 2001)
  - Route optimisation in a multidimensional identifier space

- **Kademlia** (Maymounkov & Mazières 2002) ...
Chord: Overview

- Early and successful algorithm
- Simple & elegant
  - easy to understand and implement
  - many improvements and optimizations exist
- Main responsibilities:
  - Routing
    - Flat logical address space: 1-bit identifiers instead of IPs
    - Efficient routing in large systems: log(N) hops, with N number of total nodes
  - Self-organization
    - Handle node arrival, departure, and failure
Chord: Topology

- Hash-table storage
  - put (key, value) inserts data into Chord
  - Value = get (key) retrieves data from Chord

- Identifiers from consistent hashing
  - Uses monotonic, load balancing hash function
    - E.g. SHA-1, 160-bit output $\rightarrow 0 \leq \text{identifier} < 2^{160}$
  - $Key$ 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

successor(1) = 1
successor(2) = 3
successor(6) = 0
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
Routing on Ring?

- **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: $O(1/2 \times N)$ hops on average (with $N$ nodes)
  - Node failure breaks circle
Improved Routing on Ring?

- Improved 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 in linear routing progress

- Scalable routing:
  - 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*)

<table>
<thead>
<tr>
<th>$i$</th>
<th>start</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$</th>
<th>start</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$</th>
<th>start</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Chord: Routing

Chord’s routing algorithm:
- Each node $n$ forwards query for key $k$ clockwise
  - To farthest finger preceding $k$
  - Until $n = \text{predecessor}(k)$ 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 in Storage

- Layered design
  - Chord DHT mainly responsible for routing
  - Data storage managed by application
    - persistence
    - consistency

- 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 in 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 fingers “fix_fingers”
  - replace with correct nodes on failures
  - trade-off: maintenance traffic vs. correctness & timeliness
Chord: Failure Tolerance in Routing

- Successor failure during routing
  - Last step of routing can return node failure 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
    "stabilize" uses predecessor pointer
  - 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
Chord: Node Arrival

- Examples for choosing new node IDs
  - random ID: equal distribution assumed but not guaranteed
  - hash IP address & port
  - external observables

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

ID = rand() = 6

entrypoint.chord.org?
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

<table>
<thead>
<tr>
<th>i</th>
<th>start</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

successor list

```
succ(7) = 0
succ(0) = 0
succ(2) = 3
```
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: Performance

- Impact of node failures on lookup failure rate
- lookup failure rate roughly equivalent to node failure rate
Chord: Performance

Moderate impact of number of nodes on lookup latency

Consistent average path length
Chord: Performance

- Lookup latency (number of hops/messages):
  \[ \sim \frac{1}{2} \log_2(N) \]
- Confirms theoretical estimation
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.
Pastry: Overview

- Similar to Chord: Organises nodes & keys in a ring of flat hash IDs $0 \leq \text{ID} \leq 2^{128} - 1$
- Uses prefix-based routing:
  - Interprets identifiers as digit strings of base $2^k$, $k \approx 4$
  - Routing according to "longer prefix match"
  - Result: routing down a tree
- Routing table built according to proximity selection
  - enhanced routing efficiency due to locality
Pastry: Identifier Mapping

- Pastry views $\ell$-bit identifiers as digit strings of base $2^b$
- Example: $\ell = 4$, $b = 2$
- Keys (K..) are stored at closest node (N..) according to prefix metric
- In case of equal distance key is stored on both neighbouring nodes (K22)
Pastry Routing Table

- Contains $\ell/\beta$ rows ("the range of string lengths")
- $2^\beta - 1$ columns ("the digits", one represents the node)
- Cell position approximates pastry node $v$ within overlay, using the index transformation ("·" concatenates):
  $$T(i, j) = prefix(i - 1, hash(v)) \cdot j_\beta,$$
- Cell value maps to corresponding network address
- As there are several nodes with same prefix match: topologically closest selected for routing table

⇒ Proximity Neighbour Selection (PSN)
Prefix-Match Routing Table

Node ID $v = 103220$, $l = 12$, $b = 2$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>031120</td>
<td>1</td>
<td>201303</td>
<td>312201</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110003</td>
<td>120132</td>
<td>132012</td>
</tr>
<tr>
<td>2</td>
<td>100221</td>
<td>101203</td>
<td>102303</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>103031</td>
<td>103112</td>
<td>2</td>
<td>103302</td>
</tr>
<tr>
<td>4</td>
<td>103200</td>
<td>103210</td>
<td>2</td>
<td>103233</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>103221</td>
<td>103222</td>
<td>103223</td>
</tr>
</tbody>
</table>
Routing & Lookup Tables

Three tables:

- Routing – Prefix Match
- Leaf Set – Closest Nodes in Overlay
- Neighbourhood Set – Closest Nodes in phys. Network according to given metric: RTT, Hops, ...

Routing Table

<table>
<thead>
<tr>
<th>Node 103220</th>
<th>031120</th>
<th>1</th>
<th>01303</th>
<th>312201</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>110003</td>
<td>120132</td>
<td>132012</td>
</tr>
<tr>
<td>2</td>
<td>100221</td>
<td>101203</td>
<td>102303</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>103112</td>
<td>2</td>
<td>103302</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>103210</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Leaf Set

103123 103210 103302 103330

Neighborhood Set

031120 312201 120132 101203
Pastry Routing

Step 1: Check, if key k is within the range of the leaf set
- Request forwarded to closest node in leaf set

Step 2: For k not in the range of leaf set, lookup routing table
- Try to identify entry with longer common prefix
- If not available, route to entry closer to key

Note: Routing is loop-free, as forwarding is strictly done according to numerical closeness.
### Pastry Routing Examples

**Node ID** \( v = 103220 \)

<table>
<thead>
<tr>
<th>Routing Table</th>
<th>Leaf Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 031120 1</td>
<td>201303, 312201</td>
</tr>
<tr>
<td>1 0</td>
<td>110003, 120132, 132012</td>
</tr>
<tr>
<td>2 100221</td>
<td>101202, 102303, 3</td>
</tr>
<tr>
<td>3 103112</td>
<td>2, 103302</td>
</tr>
<tr>
<td>4</td>
<td>103210, 2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Key** \( k = 103200 \)

**Key** \( k = 102022 \)

**Key** \( k = 103000 \)
Pastry: Node Arrival

- New node $n$ picks Pastry ID and contacts a Pastry node $k$ nearby w.r.t the proximity metric
- As $k$ is nearby, its neighbourhood set is copied to $n$
- The leaf set is copied from the numerically closest overlay node $c$, which $n$ reaches by a join message via $k$
- The join message is forwarded along nodes with increasingly longer prefixes common to $n$ and will trigger routing updates from intermediate nodes to $n$
- Finally $n$ sends its state to all nodes in its routing tables (active route propagation incl. time stamps)
Pastry: Node Failure

- Node failure arrives at contact failures of tabulated nodes
  - Lazy failure detection
- Pastry provides several redundancies:
  - Routing tables may include several equivalent entries
  - Forwarding may take place to an adjacent entry
- Routing & neighbourhood table repair:
  - Query nodes neighbouring in table rows
  - If unsuccessful: query entries from previous rows
  - Lively routing tables are advertised from new nodes
Pastry: Hop Performance

![Bar Chart]

- Black: PNS N=1000
- Light Gray: PNS N=60000
- Dark Gray: no locality

- X-axis: hop number
- Y-axis: delay

---

49  
http://www.informatik.haw-hamburg.de/~schmidt  
Hochschule für Angewandte Wissenschaften Hamburg  
Hamburg University of Applied Sciences
Pastry: Delay Stretch

![Graph showing delay stretch vs. number of nodes for different scenarios: no locality, predicted, PNS.](image)
Pastry: Summary

- **Complexity**
  - Messages per lookup: $O\left(\log_{2^b} N\right)$
  - Messages per mgmt. action (join/leave/fail): $O\left(\log_{2^b} N\right)/O\left(\log_b N\right)$
  - Memory per node: $O\left(b \cdot \log_{2^b} N\right)$

- **Advantages**
  - Exploits proximity neighbouring
  - Robust & flexible

- **Disadvantages**
  - Complex, theoretical modelling & analysis more difficult
  - Pastry admits constant delay stretch w.r.t. # of overlay nodes, but depends on network topology – Chord’s delay stretch remains independent of topology, but depends on overlay size
CAN: Overview

- Maps node IDs to regions, which partition \(d\)-dimensional space
- Keys correspondingly are coordinate points in a \(d\)-dim. torus: \(<k_1, ..., k_d>\)
- Routing from neighbour to neighbour – neighbourhood enhanced in high dimensionality
- \(d\) tuning parameter of the system
CAN: Space Partitioning

- Keys mapped into $[0,1]^d$ (or other numerical interval)
- Node’s regions always cover the entire torus
- Data is placed on node, who owns zone of its key
- Zone management is done by splitting / re-merging regions
- Dimensional ordering to retain spatial coherence
CAN Routing

- Each node maintains a coordinate neighbour set (Neighbours overlap in \((d-1)\) dim. and abut in the remaining dim.)
- Routing is done from neighbour to neighbour along the straight line path from source to destination:
- Forwarding is done to that neighbour with coordinate zone closest to destination

1's coordinate neighbor set = \{2, 3, 4, 5\}
7's coordinate neighbor set = \{\}
CAN Node Arrival

The new node

1. Picks a random coordinate
2. Contacts any CAN node and routes a join to the owner of the corresponding zone
3. Splits zone to acquire region of its picked point & learns neighbours from previous owner
4. Advertises its presence to neighbours

1’s coordinate neighbor set = \{2,3,4,7\}
7’s coordinate neighbor set = \{1,2,4,5\}
Node Failure / Departure

- Node failure detected by missing update messages
- Leaving gracefully, a node notifies neighbours and copies its content
- On node’s disappearance zone needs re-occupation in a size-balancing approach:
  - Neighbours start timers invers. proportional to their zone size
  - On timeout a neighbour requests ‘takeover’, responded only by those nodes with smaller zone sizes
CAN Optimisations

- Redundancy:
  Multiple simultaneous coordinate spaces - Realities
- Expedited Routing: Cartesian Distance weighted by network-level measures
- Path-length reduction: Overloading coordinate zones
- Proximity neighbouring: Topologically sensitive construction of overlay (landmarking)
- ...
CAN Path Length Evaluation

![Graph showing the number of hops vs. number of nodes for different dimensions (2, 3, 4, 5).]
CAN Path Length Evaluation (2)

Number of nodes = 131,072

- \(d=2, r=2\)
- Increasing dimensions, \#realities=2
- Increasing realities, \#dimensions=2

Number of hops vs Number of neighbors graph for different values of \(d\) and \(r\).
CAN: Summary

- **Complexity**
  - Messages per lookup: $O(N^{1/d})$
  - Messages per mgmt. action (join/leave/fail): $O(d/2 N^{1/d})/O(2d)$
  - Memory per node: $O(d)$

- **Advantages**
  - Performance parametrisable through dimensionality
  - Simple basic principle, easy to analyse & improve

- **Disadvantages**
  - Lookup complexity is not logarithmically bound
  - Due to its simple construction, CAN is open to many variants, improvements and customisations
Implementations / Deployment

- Many concepts & implementations ...
  - Storage Systems
  - Indexing/Naming
  - Content Distribution
  - DB Query Processing, ...

- Real Deployment:
  - Public DHT-Service: OpenDHT
  - Filesharing: Overnet (eDonkey), BitTorrent (newer)
  - Media Conferencing: P2P-SIP
  - Music Indexing: freeDB
  - WebCaching: Coral

- Problems: Overload + Starvation, Need Fairness Balance
Programming a DHT

- Two communication interfaces:
  - One towards the application layer (user of the DHT)
  - One towards other nodes within the DHT
  - Functions similar
- Node-to-Node Interface must be network transparent, choice of:
  - Application layer protocol (using TCP or UDP sockets)
  - Remote procedure calls (RPCs)
  - Remote Method Invocation (RMI/Corba)
  - Web services ...
- Application layer interface may be local or distributed
DHT Data 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)
DHT Self Organisation Interface

- **Join(mykey):** Retrieve ring successor & predecessor (neighbours), initiate key transfer
- **Leave():** Transfer predecessor and keys to successor
- **Maintain predecessor & successor (neighbour) list,** e.g., stabilize in Chord
- **Maintain routing table,** e.g., `fix_fingers` in Chord
Dabek Model

- Layered approach towards a “unified overlay routing”
- Core idea: KBR layer (Tier 0) as a routing abstraction on (interchangeable) structured schemes

- Tier 1: General services
- Tier 2: Higher layer services and applications
Common KBR API

<table>
<thead>
<tr>
<th>Tier</th>
<th>Message Routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>forward(key→K, msg→M, nodehandle→nextHopNode)</td>
</tr>
<tr>
<td></td>
<td>deliver(key→K, msg→M)</td>
</tr>
<tr>
<td>0</td>
<td>route(key→K, msg→M, nodehandle→hint)</td>
</tr>
</tbody>
</table>

Table 5.1: The KBR Message Routing API Calls Implemented on the Corresponding Layers

<table>
<thead>
<tr>
<th>Tier</th>
<th>State Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>update(nodehandle→n, bool→joined)</td>
</tr>
<tr>
<td>0</td>
<td>nodehandle [] local_lookup(key→K, int→num, boolean→safe)</td>
</tr>
<tr>
<td></td>
<td>nodehandle [] neighborSet(int→num)</td>
</tr>
<tr>
<td></td>
<td>nodehandle [] replicaSet(key→k, int→max_rank)</td>
</tr>
<tr>
<td></td>
<td>boolean range(nodehandle→N, rank→r, key→lkey,key→rkey)</td>
</tr>
</tbody>
</table>

Table 5.2: The KBR State Access API Calls Implemented on the Corresponding Layers
References


