Technical Report 371/09

ChordNet: Protocol Specification and Analysis

Version 1

Dennis Schwerdel, Matthias Priebe, Paul Müller and Peter Merz

Integrated Communication Systems Lab (**Müller**, Schwerdel)

Distributed Algorithms Group (**Merz**, Priebe)

University of Kaiserslautern

{schwerdel,priebe,pmueller,pmerz}@cs.uni-kl.de

February 23, 2009

Contents

1 Motivation

Peer to peer overlay networks offer a new way of communication. These networks do not need a central component like a server and scale well with the number of participants. Peer-to-peer networks can connect millions of nodes efficiently while client-server solutions would need huge investments in server architecture. With a peer-to-peer overlay the costs of the infrastructure is fairly distributed among all peers.

ChordNet aims to be a state-of-the-art peer-to-peer overlay that efficiently connects even large numbers of nodes. Another design principle for the ChordNet protocol is to be platform-independent and easy to implement.

ChordNet is application neutral. That means that ChordNet simply delivers messages between nodes and manages the network. It does not define what kind of payload the messages contain. It is possible to have nodes running different applications inside the same network.

ChordNet is based on ChordNet but seeks to fix some of the shortcomings of Chord. It uses a 2-tier architecture with super-peer/edge-peer distinction that allows peers with firewalls/NAT[1]. ChordNet improves Chord routing efficiency and deals with special problems that arise when Chord is used as a general peer-to-peer network. Also ChordNet introduces a latency-based routing method for Chord networks.

The long-term goal of ChordNet is to have a large-scale ChordNet overlay network in the Internet which is open for all nodes and applications which speak the ChordNet protocol.

With this specification implementers should be able to create software that uses the ChordNet protocol.

Chapter 2 describes the network structure of ChordNet. A few services on that network are described in chapter 3. Chapter 4 defines the protocol, its data structures and message types in detail, while chapter 5 focuses on the behavior of the nodes. Chapter 6 lists some experiments on the reference implementation.

1.1 Related work

Structured peer-to-peer networks like CAN [14], Pastry [16], Tapestry [19] and especially Chord [17] have been researched a lot in the last years.

Chord has been found to be a very promising structure and was researched in detail. Bidirectional chord structures as described in [6] and [1] are the next evolutionary step for chord networks, because TCP connections as used by the majority of implementations are bidirectional and so bidirectional chord is mostly a change inside the finger tables.

Multi-tier chord architectures like ChordNet have already been described in [11; 20; 8]. Those architectures avoid problems with firewalls/NAT that traditional Chord has.

[15] uses the finger tables of the predecessor and successor nodes to create an initial finger table. This is called fast node joining and reduces the complexity of node joins.

The normal PRS routing optimization was described in [5]. In chapter 2.3.3 an advanced version of PRS routing is introduced. [9; 12] show that using the neighbors of neighbors in

[1] Network Address Translation

the routing method can improve the routing significantly. NoN-routing[2] can be improved by using hash functions as described in [2] but this is not used in ChordNet because it changes finger positioning.

2 Topology

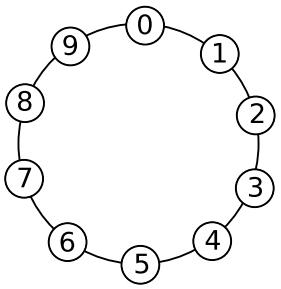
The topology of ChordNet is based on a bi-chord ring. Chord and bi-chord network structures have been described and researched a lot and offer a simple network structure with efficient message delivery. ChordNet uses a different optimization method than normal chord networks, it uses only direct messages and should converge very fast. The bi-chord network structure and its optimization is described in section 2.1.

ChordNet extends the normal chord ring to be a super-peer/edge-peer network. With this extension peers that are restricted by a firewall or NAT router can still participate in the network. The super-peer/edge-peer extension is described in section 2.2.

Messages can be sent to a list or a range of nodes. Thus broadcast and multicast messages are easy to implement. An advanced routing method that uses latency and Neighbor-of-Neighbor information is described in section 2.3.

2.1 Bidirectional Chord ring

Figure 1: Full bi-chord ring with id space $[0 \dots 9]$



In the chord network all nodes $n \in Nodes$ have a unique id $n.id$ in the id space $[0 \dots N-1]$ (with $N = 2^d$ and $d = 60$ normally). The nodes are organized in a ring sorted by their id. The previous (next) id of a node a is defined as $previd(a) = (a.id - 1 + N) \, mod \, N$ and $nextid(a) = (a.id + 1) \, mod \, N$.

The id-distance between two nodes is defined as

$$iddist(a, b) = min((a.id - b.id) \, mod \, N, (b.id - a.id) \, mod \, N)$$

I.e. when the distance would be greater than $N/2$ if measured clockwise the counter-clockwise distance is taken instead.

A node b is called **between** a and c iff $iddist(a, b) + iddist(b, c) = iddist(a, c)$.

[2] Neighbor-of-Neighbor routing

That means that a node is between two nodes iff it lies on the shortest path between those nodes.

Every node a has a set of other nodes called finger nodes. The positions of the finger nodes are defined by

$$a.fingerpos_i = \begin{cases} (a.id - 2^{-i-1}) \, mod \, N & i < 0 \\ (a.id + 2^{i-1}) \, mod \, N & i > 0 \end{cases}$$

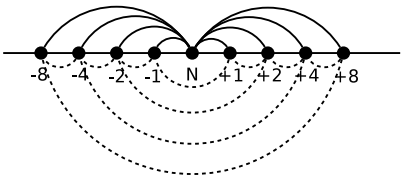
with ($i \in [-d \ldots -1, 1 \ldots d]$). The node for such a position $finger_i$ is selected among all candidate nodes as the one with the smallest id-distance to the position. Candidate nodes for positive (negative) fingers of a node x are the next (previous) node k and all nodes f where k is **between** x and f.

The $finger_1$ is called „successor" and the $finger_{-1}$ is called „predecessor". The successor and predecessor node is by this construction a neighbor of the node in the id space.

2.1.1 Optimization/Maintenance

A goal for local optimization of a node n is to always keep the distance between $n.finger_i.id$ and $n.fingerpos_i$ as small as possible. This is accomplished by exchanging the finger table with finger peers. When a node receives a finger table of a peer, it searches for better nodes for its finger positions inside that finger table.

Figure 2: Finger connections



Good candidates for example for $finger_4$ is the $finger_3$of $finger_3$ and the $finger_{-4}$ of the node of $finger_5$. Also all nodes of the finger tables of the successor and predecessor are good candidates for the own finger table.

In the so-called finger table exchange, each node sends its own finger table to its finger peers and requests their finger tables as a reply. This finger table exchange is done periodically in each node. When a node joins the network it starts the first exchange instantly. In [15] this behavior is described a „fast node joining".

When a finger node disappears or crashes the finger table must be repaired. This happens in three steps:

- First the finger node is removed from the finger table.
- Then its positions are filled by the best nodes among the other finger nodes. For any finger this is normally the previous or next finger.
- The peer executes a finger table exchange.

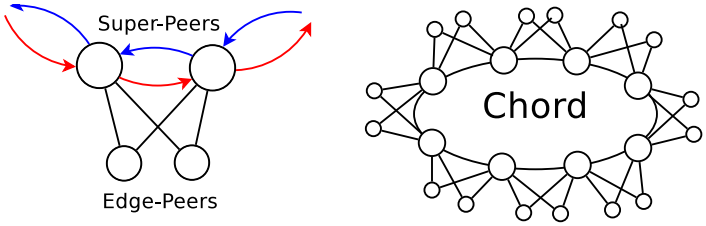
2.2 Edge-Peers/Super-Peers

All nodes described in section 2.1 must be reachable from the public Internet, but some nodes are not. So there are two node types. All nodes that are reachable become super-peers and all others become edge-peers.

Edge-peers are not inside the chord ring. They are located between the two super-peers (called parent nodes) that are closest to the edge-peers id. Each edge-peer has connections to both parent nodes. Edge-peers can not be finger of any node.

The parent nodes keep a list of their edge-peers. A super-peer can assert that between itself and its predecessor/successor only edge-peers exist that are inside the edge-peer list. Edge-peers cannot assert anything about the id space between them and their parents.

Figure 3: Network with super-peers and edge-peers

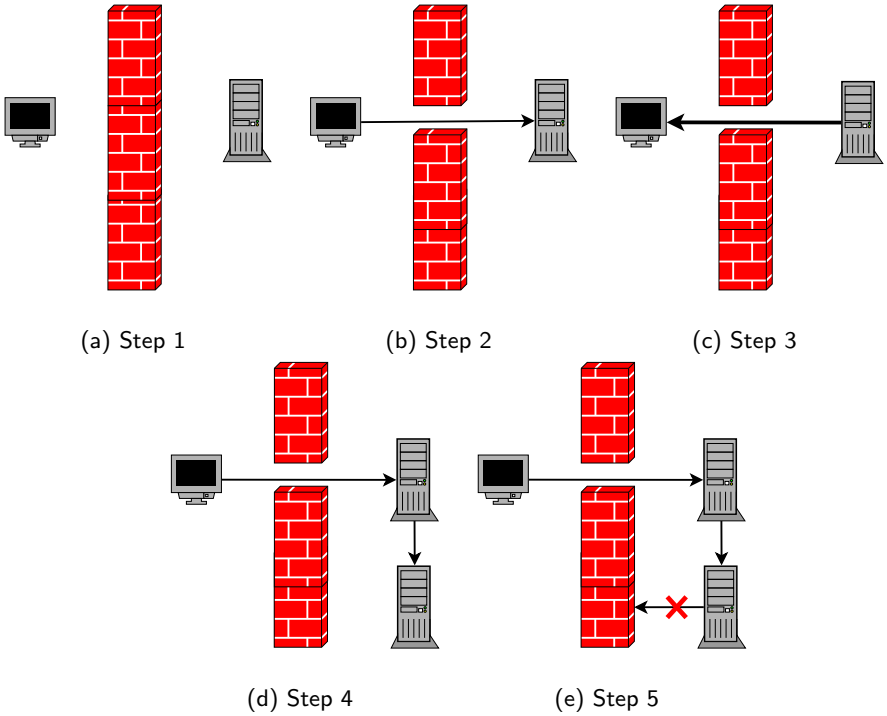


2.2.1 Reachability analysis

The reachability analysis has two goals:

- Check whether a peer is reachable from the public Internet and thus can become super-peer.
- Detect the IP-Address of the peer, as this might not be known to the peer itself.

Figure 4: Firewall/NAT detection



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5

To start a reachability analysis the testee sends a message to a peer inside the network. This peer is called the forwarding peer. The forwarding peer can detect the public address of the testee as it has a direct connection with it. But it can not check the reachability of the testee because that connection might have opened the firewall/NAT for the forwarding peer.

The forwarding peer then chooses a random peer, called testing peer, and sends a message to it, containing the public address of the testee. This peer tries to connect to the testee which will fail if it is not reachable.

- When the connection attempt fails, the testing peer sends a message to the forwarding peer including the public address and a negative reachability flag. The forwarding peer forwards this message to the testee.
- When the connection attempt succeeds, the testing peer sends a message to the testee through this connection, containing the public address and a positive reachability flag.

This analysis needs at least two peers inside the network. That means this analysis can not be done for the first and second peer that enter the network. The first peer is always considered to be reachable, otherwise no one would be able join the network. The second peer is always considered to be unreachable to be on the safe side.

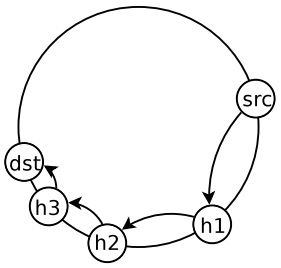
The forwarding peer selects the random peer among its fingers and not from its edge-peers (unless there are no fingers) because the edge-peer might be located behind the same firewall

as the testee. In this situation the testee would be incorrectly classified as reachable.

2.3 Message delivery

Broadcast messages are messages destined to a range of node-ids, and multicast messages are messages destined to a list of node-ids. Broadcasts from a node x to all other nodes have the range of $[nextid(x.id) \ldots previd(x.id)]$. Unicast messages are a special case of multicast messages with only one id in the list. Both multicast and broadcast messages could be seen as a bunch of virtual unicast messages that are bundled to reduce network load. Each of these virtual unicast messages is routed towards its destination by applying a unicast routing method and later regrouping all virtual unicast messages that are to be forwarded to the same peer. The following routing method thus asserts that messages are unicast.

Figure 5: Message delivery



Messages are routed to their destination by forwarding them to the finger nodes. This means that ChordNet uses direct routing. Direct routing means that a message is sent directly from one hop to the next. In contrast iterative routing means that each hop sends information about the next hop to the original sender, which then contacts that peer. A message to a node t is forwarded in a forwarding node s to t directly if t is an edge-peer of s or $t = s$ or to the finger of s that has the smallest distance to t. This a normal greedy routing method that aims to minimize the id-distance to the destination.

Algorithm 1 Routing method

When a message with the destination t reaches the node s the following method is used to select the next hop.

- When t is the same node as s the message has reached its destination.
- When t is an edge-peer of s the message is forwarded to t directly.
- Otherwise let F be the finger table of s.
  $gain\,(f) = iddist\,(s,t) - iddist\,(f,t)$
  Select the finger h which is closest to t.
  The next hop is $h = \arg\max_{f \in F \cup \{s\}} (gain\,(f))$ and the gain is $g = gain\,(h)$.
  - If $g \leq 0$ and s is a super-peer, no finger is closer to t than s and the message can not be delivered.
  - If $g > 0$ or s is an edge-peer, the message is forwarded to h.

When a broadcast is routed the range of ids is split into parts so that every id in that range is forwarded to the peer that fits best. To route a multicast, the list of ids is split analog to the range of ids of a broadcast.

Messages may only be forwarded to a node that is closer to the destination, so the id-distance decreases monotonically with each step. There is only one exception to this rule: When an edge-peer forwards the message to one of its super-peers the distance might increase if the destination is an edge-peer of the same super-peer. I.e. source and destinations are siblings.

After the first step, the message reaches a super-peer or its destination. In all subsequent steps the id-distance decreases monotonically. Thus the routing method is guaranteed to converge.

Nodes assert that no nodes (other than its own edge-peers) are between its predecessor/successor and itself. Broadcasts to non-existing nodes are discarded, while multicast messages are treated specially depending on the message type (Section 4.5 describes the different message types).

2.3.1 Complexity

Normal chord routes only in clockwise direction, so let d be the clockwise id-distance between source and destination. Each routing step i is able to add 2^i to the node id and thus set one bit of d to zero. So the path length is simply the Hamming-norm of the id-distance $H(d)$. The maximal path length for a network with exponent k is $max\,(H\,(x))\ x \in \left[0 \ldots 2^k - 1\right]$ which is k because all ids have k bits. The average path length $avg\,(H\,(x))\ x \in \left[0 \ldots 2^k - 1\right]$ is $\frac{k}{2}$ because the probability for each bit of d to be set is 50%.

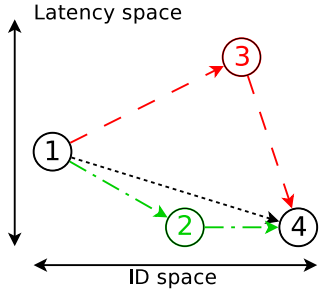
ChordNet routing is a general bi-chord routing. In each routing step the routing node decides whether to route clockwise or counter-clockwise. [4] shows that the maximal path length of this routing method is $\frac{k}{2}$ and the average path length is about $\frac{k}{3}$.

For full networks with $N = 2^k$ nodes the maximal path length is $\frac{\log_2 N}{2}$ and the average path length is about $\frac{\log_2 N}{3}$. The experiments in section 6.1 show that networks with $N \ll 2^k$ also come close to these values when the node ids are evenly distributed.

2.3.2 Proximity Route Selection Optimization

In the example in figure 6 node 1 routes a message with destination node 4. Its fingers are node 2 and node 3. Normal routing would select node 3 as next hop since it has the smallest distance to the destination. But it would be better to route to node 2 because of the latency. The connection from node 1 to node 3 has a higher latency than the connection from node 1 to node 2. Additionally choosing the node 2 will reduce the latency distance to almost 0 while choosing node 3 will even increase the latency distance to the destination.

Figure 6: PRS example



When routing a message in plain chord routing, the finger to forward the message to is selected based on id distance. When a connection to a finger is extremely slow it might be better to avoid that connection and route to a different finger that is not optimal when measured in id distance. To accomplish that, not only the distance to the target is used to select the forwarding finger, but the fraction of skipped distance to the target divided by the latency to the particular finger.

There are two ways to measure the gain.

Algorithm 2 normal PRS Routing method

The routing method in Algorithm 1 is changed.

- Let $lat(x)$ be the function that gives the latency between s and $x \in F$.
- $gain(f) = \begin{cases} \frac{iddist(s,t) - iddist(f,t)}{lat(f)} & f \neq s \\ 0 & f = s \end{cases}$

Algorithm 3 advanced PRS Routing method

With a small change, the PRS routing method could be improved further:

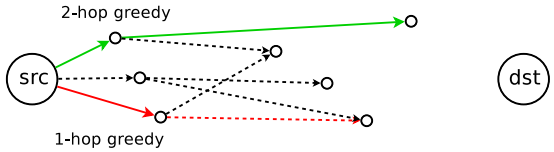
- $gain(f) = \begin{cases} \frac{1/(iddist(f,t)+1)-1/(iddist(s,t)+1)}{lat(f)} & f \neq s \\ 0 & f = s \end{cases}$

Simulations show that with these metrics, messages reach their destinations faster but with more hops taken. Also the advanced version performs better than the normal version. The complexity analysis of section 2.3.1 does not hold for this optimization.

2.3.3 Neighbor of Neighbor (NoN) Routing Optimization

In ChordNet each node exchanges its finger table with its fingers. This information is primarily used to optimize the finger tables of the nodes but it can be used to optimize the routing method as well.

Figure 7: NoN example



In figure 7 is an example of normal routing and NoN-routing. While normal routing is 1-hop greedy, NoN routing uses the finger lists of the neighbors to calculate a 2-hop greedy next hop. This improves the routing.

Algorithm 4 NoN Routing method

The routing method in Algorithm 1 is changed.

- Let $F' = \{f \in F \,|\, iddist(s,t) > iddist\,(f,t)\}$ be the set of fingers that ensure routing convergence.
- Let NoN_f be the Finger-Table of finger f for $f \in F$.
- $gain\,(f,n) = iddist\,(s,t) - iddist\,(n,t)$ for $f \in F$ and $n \in NoN_f \cup \{f\}$
- $gain\,(f) = \begin{cases} \max_{n \in NoN_f \cup \{f\}} (gain(f,n)) & f \neq s \\ 0 & f = s \end{cases}$
- The next hop is $h = \arg\max_{f \in F' \cup \{s\}} (gain\,(f))$.

Instead of choosing the nearest finger to the destination node as next hop, the finger that has the nearest finger to the destination node is chosen. With this extension the routing

method can look one hop ahead.

NoN-routing has been described in detail in [9; 12] and it has been proven that the average path length is in $\Theta\left(\frac{\log_2 N}{\log_2 \log_2 N}\right)$. This algorithm is changed slightly to only route to fingers that are closer to the destination. If that change is not made, the routing method is not guaranteed to converge when finger tables of fingers (NoN-tables) are out of date.

Section 6.1 contains experiments showing a significant improvement by NoN-routing.

2.3.4 Combined PRS and NoN Routing

The previous paragraphs introduced PRS and NoN routing. Those routing extensions can also be combined when latency information is available in the NoN-tables.

Algorithm 5 PRS-NoN Routing method

The routing method in Algorithm 1 is changed.

- Let $F' = \{f \in F \mid iddist(s,t) > iddist(f,t)\}$ be the set of fingers that ensure routing convergence.
- Let NoN_f be the Finger-Table of finger f for $f \in F$.
- Let $lat(f)$ be the function that gives the latency between s and $f \in F$.
  Also let $lat_f(n)$ be the function that gives the latency between $f \in F$ and $n \in NoN_f \cup \{f\}$.
- $gain(f,n) = \frac{iddist(s,t) - iddist(n,t)}{lat(f) + lat_f(n)}$ for $f \in F$ and $n \in NoN_f \cup \{f\}$
- $gain(f) = \begin{cases} \max_{n \in NoN_f \cup \{f\}} (gain(f,n)) & f \neq s \\ 0 & f = s \end{cases}$
- The next hop is $h = \arg\max_{f \in F' \cup \{s\}} (gain(f))$.

Algorithm 6 advanced PRS-NoN Routing method (APN routing)

With a small change, the PRS-NoN routing method could be improved further:

- $gain(f,n) = \frac{1/(iddist(n,t)+1) - 1/(iddist(s,t)+1)}{lat(f) + lat_f(n)}$ for $f \in F$ and $n \in NoN_f \cup \{f\}$

Chapter 6.1 shows that this routing method brings a huge improvement over both PRS and NoN routing.

2.4 Network joining

Network joining consists of two phases. In the first phase the joining position is searched.

When a new node n wants to join the network, it chooses an id and contacts the first peer c_1. When a peer c_i is contacted by n it searches for the closest super-peer to n inside its finger table. Now four cases could happen:

- No super-peer is closer to n than c_i.
  - If c_i has an edge-peer e with $e.id = n.id$, the id of node n is already taken. c_i notifies n about that and n restarts the joining process with another id.
  - Else c_i is the join node and sends a message to n containing the first two finger nodes. If n is between c_i and its predecessor $c_i.pre$ these are $(c_i.pre,\ c_i)$ otherwise these are c_i and its successor $(c_i,\ c_i.suc)$.

Otherwise c_{i+1} has been found and is closer to n than c_i.

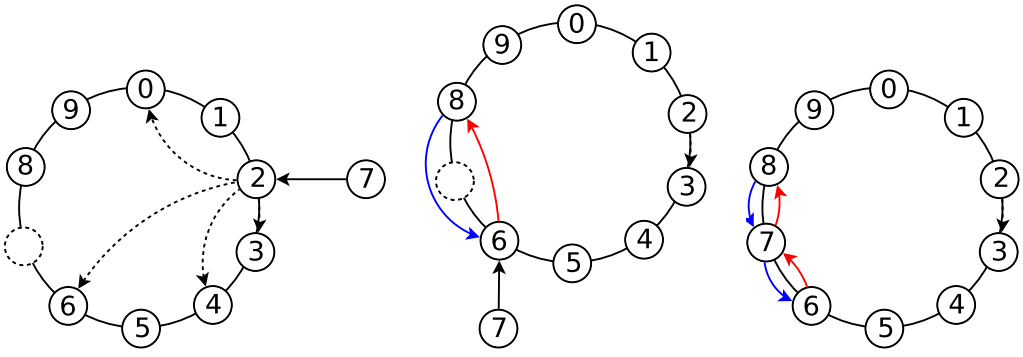
- If $c_{i+1}.id = n.id$ or $c_i.id = n.id$, the id of node n is already taken. c_i notifies n about that and n restarts the joining process with another id.
- Otherwise n is informed about c_{i+1} and contacts it.

This process is guaranteed to converge because n is routed like a message to its future position. Duplicate node-ids are easy to detect and the successor and predecessor finger nodes are always optimal. Thus a node can safely assert that no node in the ring exists between itself and its predecessor and successor except for its own edge-peers.

For the position search iterative routing is used instead of recursive because the joining node is not inside the network yet and cannot receive routed answers. Also the routing extensions could be used for the joining process as well with some limitations:

- PRS routing should not be applied because the latency measurements do not consider the latencies to and from n and thus PRS can not improve the joining process.
- When $n.id$ appears inside the NoN-table of node c_{i+1}, n should still be forwarded to c_{i+1}. In the next step n will be notified about the duplicate id by c_{i+1}.

Figure 8: Joining Node



(a) Step 1 (b) Step 2 (c) Step 3

When a node n knows its joining position and the first two finger nodes (pre, suc) it can start the second phase, the joining process.

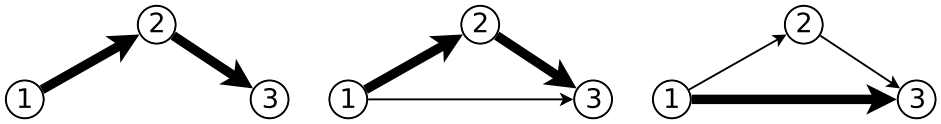
To really join the network, the node n sends a join request to pre and suc, containing self information and an indication whether it wants to become an edge-peer or a super-peer.

When the fingers receive that message, they add n to their finger-list/edge-peer-list and inform the node of the success.

The joining node initiates the first finger table update as described in section 2.1.1.

2.5 Shortcut connections

Figure 9: Shortcut connection



(a) Step 1 (b) Step 2 (c) Step 3

To reduce the number of hops on routes with high traffic, special shortcut connections can be established.

When a node notices that a lot of traffic (e.g. more than 25% of all routed traffic) is routed between two of its finger connections it sends a signal to both fingers containing the address of the other finger and the amount of traffic routed between them.

Each node has a limited number of slots for shortcut connections (e.g. 3 slots). If it receives a signal to establish a shortcut connection and has a free slot or a slot with less traffic it will use that slot to establish the shortcut connection.

To be useful, shortcut connections are treated like fingers for routing purposes. Under normal conditions the shortcut connection will take all the traffic that the signaling node has measured between the two nodes of the shortcut connection.

To measure the traffic, the sizes of forwarded messages are added up in a traffic matrix and reduced over time with a sliding average.

With this optimization, routes with high traffic will shorten over time. When a lot of traffic is routed from a single node to another single node, after some time these nodes will be connected with a shortcut connection and transfer that traffic directly as a result of this optimization.

3 Services

On top of the network that has been described in the previous chapter a set of services can be implemented. In this early specification the feature signaling and the DHT service are specified. Feature signaling is not a real service but merely a method to communicate service support.

3.1 Feature signaling

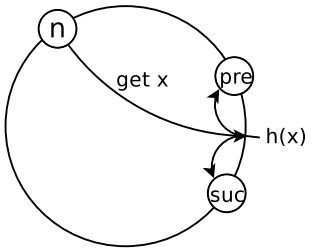
Every connection begins with a message containing a feature list of the peer. With that list future extensions of the protocol could be introduced without becoming incompatible.

With the knowledge of the features of other peers, the network can deny access to peers with lacking features, run compatibility modes or disable extra features that are not supported by all peers.

3.2 DHT Service

The chord network was originally designed to store and lookup data in an efficient way. ChordNet contains a general DHT service that can store value data for any combination of key and type. Also any data stored in the network has a timeout value, after which it expires.

Figure 10: DHT



The data key is converted to a chord id with a hash function. The two peers that surround this id directly are responsible for storing the values for that key of any type.

Data storage and lookup requests are routed to the hash id and end in one of the two peers that surround the hash id. Data storage requests are sent to the other surrounding peer as well when they reach the first one. This peer must detect that the message comes from the other responsible peer and must not forward the message to it again. This is critical to avoid routing loops.

When a node leaves the network that carries stored data it must send the data to one of its neighbors depending on the hash-id of the data key. To do so, a storage request is sent, containing the id of the neighbor instead of the hash-id.

When a node that carries stored data recovers from a lost neighbor, it stores that data inside the new neighbor if the old one carried the data too.

4 Protocol

This chapter is mainly for implementers of the protocol and should define the data types and messages in a way that all implementations are network compatible. The protocol specification aims to be easy to implement.

4.1 Node states

Nodes can have one of the three following states:

Inside: The node is inside the network, can send and receive messages to all other nodes inside the network.

Outside: The node is not inside the network, it can not receive messages from nodes inside the network that are not directly connected to the node.

Joining: The node is half inside half outside the network. It can send messages to other nodes inside the network but might not receive any messages from nodes inside the network that are not directly connected to the node.

4.2 Connections

The whole protocol uses the following assumptions about the network below ChordNet:

- The lower network layer splits data in packets, transmits them reliably to the other side and guarantees order preservation. TCP offers all these features and is a good choice.
- Connections can be built implicitly by sending a message or explicitly.
- Connections are closed automatically when no data is transmitted for a certain time span.
- When a connection is closed by the other side, this is treated as a lost connection.
- Broken connections are detected after a short while.

When a connection is established both sides send a short connection preamble and wait to receive the preamble sent by the other endpoint. The connection preamble string is „`ChordNet\n`" where „`\n`" is the newline character.

The first message that is sent over a new connection right after the preamble is the Ident message containing self information and a feature list.

4.3 General TLV-Format

All message and object classes are encoded using a TLV[3] format. The TLV format consists of three parts:

[3] Type Length Value

Type: a fixed length identifier that defines the type of the object or message. The length of this field and its values are defined in the following sections.

Length: a fixed length field that defines the length of the object or message. The length of this field and its meanings are defined in the following sections.

Value: a portion of data. The size is given in the length field and the meaning is defined by the type field.

Figure 11: TLV-Object

Type	Length	Value
10	000C	48656C6C6F20576F726C6421
Data	12	"Hello World!"

The basic data types are defined as:

- Boolean: `True` or `False`, encoded in one byte as `True=0x01` and `False=0x00`
- Byte: One unsigned byte, values from 0 (`0x00`) to 255 (`0xFF`)
- Short: 2 bytes, big-endian, unsigned, values from 0 to $2^{16} - 1$
- Integer: 4 bytes, big-endian, unsigned, values from 0 to $2^{32} - 1$
- Long: 8 bytes, big-endian, unsigned, values from 0 to $2^{64} - 1$ (Java can only handle signed longs so long values should be limited to $2^{63} - 1$)
- Float: 4 bytes, IEEE single precision floating point number

4.4 Object formats

All objects are encoded using a TLV format. The type field is a Byte value and the length field is a Short value. If an object class contains a field from another object class, this field does only contain the value part of the internal object, not the type and length parts. The length field contains the number of bytes of the value part.

Name	Type	Length	Value
ID	0x00	8	num:Long
Address	0x01	7 or 19	len:Byte address:Byte[len], portnumber:Short
ChordAddr	0x02	15 or 27	address:Address, id:ID
IDRange	0x03	16	rangeStart:ID, rangeEnd:ID
IDList	0x04	2+8*len	len:Short, list:ID[len]
PeerList	0x05	2+(15 or 27)*len	len:Short, list:(ChordAddr,Float)[len]
PingData	0x06	5	stage:Byte, data:Int/Float (depending on stage)
IsSuperPeer	0x07	1	value:Boolean
IsReachable	0x08	1	value:Boolean
Traffic	0x09	4	value:Float
FeatureList	0x0A	4*len	features:Int[len] (len from TLV, can not be part of a compound object)
Data	0x10	len	data:Byte[len] (len from TLV, can not be part of a compound object)
BroadcastDst	0x11	17	flags:Byte, range:IDRange
RoutingDst	0x12	3+8*list.len	flags:Byte, list:IDList
MetaData	0x13	undefined	undefined
DataType	0x20	2	value:Short
DataTimeout	0x21	8	value:Long

Because the length field of any object is an unsigned short value, no object can be longer than 65535 bytes. This limits the size of the objects Data, MetaData, IDList, PeerList and RoutingDst.

Bits for BroadcastDst.flags:

- 1. bit (0x01): **AllSuper**: Message should be sent to all super-peers.
- 2. bit (0x02): **AllEdge**: Message should be sent to all edge-peers.
- All other bits are reserved.

Bits for RoutingDst.flags:

- 1. bit (0x01): **LeftPeer**: Message should be sent to the left super-peer neighbor when the destination id does not exist.

- 2. bit (0x02): **RightPeer**: Message should be sent to the right super-peer neighbor when the destination id does not exist.
- 3. bit (0x04): **Undeliverable**: Message should be returned as undeliverable when the destination id does not exist.
- All other bits are reserved.

Note: The type of the data field in PingData objects depends on the value of the stage field. When the stage is 3 the data field contains the latency measured in seconds as a Float value. In all other stages the data field may contain any Integer value as the value must only be understood by the original sender.

Note: All undefined type constants below 0x80 are reserved for future use. Type constants from 0x80 to 0xFF may be used for custom objects.

Constants for FeatureList:

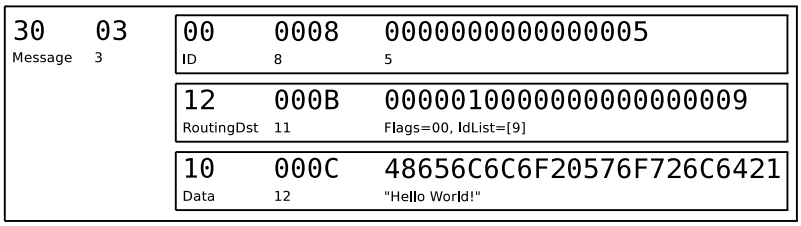
- `0x01`: Protocol version 1, as described in this specification.

4.5 Message formats

All messages are TLV encoded. The type and length fields are Byte values.

The length field gives the number of parameter objects, not the length in bytes. All parameter objects must follow in the given order.

Figure 12: TLV-Message



If any implementation cannot understand a certain message or object class then it must skip that message or object completely. Using the length fields this can even be done for unknown classes without breaking the communication stream.

The list of the message types lists the name first, then the parameters and their types in parenthesis and then the type identifier. If a parameter has a question mark „?" appended, it is optional. A pipe symbol „|" means a choice, where exactly one of the optional parameters must be present. Parenthesis are used for grouping.

All undefined type constants below 0x80 are reserved for future use. Type constants from 0x80 to 0xFF may be used for custom objects.

Messages are sent from a sender to the receiver over a direct connection. Thus the direct sender of a message is considered to be known. Messages are not routed unless stated otherwise.

When a connection is broken but the last sent message could be reconstructed, direct messages should be dropped and routed messages should be routed on another route if not stated otherwise.

Ident(self:ChordAddr, features:FeatureList?) [0x00]

This message informs the receiver about the identity and the features of the sender. It is the first message that is sent on a new connection right after the preamble.

Other messages could be sent right after this message, without waiting for the Ident message of the other side.

Disconnect() [0x01]

This message signals the receiver that the sender wishes to terminate the connection between them and that the connection must not be re-established immediately by the receiver.

This message must be the last message sent on that connection and the connection has to be terminated by the sender after sending that message and by the receiver after receiving the message.

When a connection is terminated by this way, it should be treated like a connection that has been permanently lost. Implementations should not show an error, as this is a normal behavior.

Ping(data:PingData) [0x02]

Ping message with time stamp/latency. This message is part of a ping-pong-peng series to measure the connection latency and keep the connection alive. Data contains stage information and a time stamp/latency.

- Stage 1 (Ping): This message contains a local time stamp value. The meaning of that value might depend on the implementation. When this message is received, a stage 2 ping message with the original time stamp information must be sent to the original sender.
- Stage 2 (Pong): This message is the reply to a stage 1 ping message. It contains the original time stamp from the stage 1 ping message. Upon reception the receiver can measure the connection latency by comparing the current time with the time stamp. When this message has been received and the latency has been calculated a stage 3 ping message with the latency must be sent to the original sender.
- Stage 3 (Peng): This message is the reply to a stage 2 ping message. It contains the measured latency (half RTT) in seconds as a Float value.

Receivers of stage 2 and 3 messages should use the latency to estimate an average latency with a sliding mean.

FindJoinNode(self:ChordAddr) [0x10]

This message is sent from a peer outside of the network to a peer inside the network to find a joining peer. The receiving node searches for a node that is closest to the id of the sender (self.id) in its finger table. This message must be answered with NextJoinNode, JoinHere or DuplicateId.

First the receiving node checks if one of its edge-peers has the same id as the joining node and sends a DuplicateId message with the edge-peer as parameter if so.

If not, the receiving node searches the node that is closest to the joining peer inside his finger table. The found node must be directly connected with the receiver and be closer to the sender than the receiver.

- If the found node has the same id as the joining node, a DuplicateId with that node as parameter is sent.
- If the receiving node is closer to the joining node than all nodes in the finger table, it sends a JoinHere message. The JoinHere message must contain two peers as parameters. If the receiver does not have a predecessor nor a successor, it sends information about itself twice. Else it sends information about its predecessor and itself if the sender is between the receiver and its predecessor, otherwise it sends information about itself and its successor.
- Otherwise it sends a NextJoinNode message containing the found node.

The proof for complexity from section 2.3.1 holds for this joining method as well. A similar method to NoN-routing might be used to speed up joining. Giving the joining peer the address of the NoN peer directly is not a good idea since that peer is not connected to the current peer and might have left the network since last finger-table exchange.

PRS routing should not give any improvement for the joining method as the delay between the receiving node and the next join node does not matter.

NextJoinNode(node:ChordAddr) [0x11]

This message can be the answer to a FindJoinNode message and is sent from a peer inside the network to a peer outside the network. If a node that wants to join the network receives this message, it should send a FindJoinNode message to the given node.

Also it might want to close the connection to the sender, because it is no longer used.

JoinHere(pre:ChordAddr,suc:ChordAddr) [0x12]

This message is the answer to a FindJoinNode message if the joining position is found. The message contains the future predecessor and successor nodes. It is sent from a node inside the network to a node outside the network.

When a node wants to join the network it can send both peers a Joining message to join the network. This has to happen shortly (E.g. 2 seconds) after receiving this message, otherwise the receiver has to search the position again and send FindJoinNode messages to both peers.

DuplicateId(dup:ChordAddr) [0x13]

This message is sent to a node that has an id that already exists inside the network. When a node receives such a message, it must terminate all connections, select another id and join the network again.

Joining(self:ChordNode,superPeer:IsSuperPeer) [0x14]

This message indicates that the sending node wants to join the network. It is only sent to a node inside the network. This message is only sent to the predecessor and successor nodes as given by JoinHere. Those nodes will insert the sender into the network. If superPeer is true the sending node will become a super-peer, otherwise the sending node will become an edge-peer.

When a node receives this message it will check if the sender is between itself and its predecessor or successor. If so it will insert the sender in the finger table or edge-peer list, depending on the value of superPeer and reply with a Joined message. Otherwise it will treat this message as if it was a FindJoinNode message.

After sending this message, the sender state changes to Joining.

Joined() [0x15]

This message is sent to the sender of a Joining message when it has been successfully inserted into the network. The receiver can insert the sender in the finger table and is now part of the network, its state changes to inside the network.

ChangeSuperPeer(peer:ChordAddr) [0x03]

This message is sent from a super-peer to its edge-peers if a new super-peer joins the network between them. The edge-peer should optimize its finger table with the new super-peer given in the field peer and send a Joining message with information about itself to the new super-peer.

This message must only be sent from super-peers to edge-peers and only with super-peers as parameter that are between the sender and the receiver.

After receiving this message the state of the receiving node changes to Joining.

Parting((pre:ChordAddr,suc:ChordAddr)?) [0x04]

This message is sent by a peer that wants to leave the network to all its fingers and edge-peers. When a super-peer sends this message, the parameters pre and suc are present and contain the predecessor and the successor to help edge-peers and other super-peers find a replacement node. When an edge-peer sends this message those parameters are not present.

Any receiving node must remove the sending node from its finger table and edge-peer list and optimize its finger table with the parameters given.

After sending this message, the sender state changes to outside the network.

GetPeerList(peers:PeerList?) [0x05]

This message is the first part of a finger table exchange cycle. It should contain a list of all fingers of the sender and the sender itself if it is a super-peer. All peers in the list must be super-peers.

For each peer x in the list sent by y the latency (half RTT) between x and y (measured in seconds) is sent along with the information about x.

The receiver must answer with a PeerList message containing its finger table. After that it can use the peers in the list to update its finger table.

PeerList(peers:PeerList?) [0x06]

This message is the second part of a finger table exchange cycle. It is identical to a GetPeerList message except that no answer is sent.

StoreData(hash:ID,type:DataType,key:Data,value:Data,timeout:DataTimeout) [0x20]

This message is used to store data inside the chord ring. For any type and key combination a value can be stored.

The data will be stored in both nodes that surround the hash-id directly. Let those nodes be a and b and the receiver r. If r is either a or b, then r knows also the other peer, because that is its predecessor or successor.

- If $r = a$ or $r = b$ the receiver will store the data for at least the given timeout.
  - If $a \neq b$ and $r = a$ and the sender is not b then the receiver has to send the message to b.
  - If $a \neq b$ and $r = b$ and the sender is not a then the receiver has to send the message to a.
- If the receiving node knows a super-peer between the hash-id and itself, it will forward the message to that peer like a routed message.

When a peer stores data, it overwrites all other data with the same key and same type.

GetData(sender:ID,hash:ID,type:DataType,key:Data) [0x21]

This message is used to retrieve data that has been stored in the ring. This message will be routed through the network to a node that has the hash ID or is a neighbor of the hash ID. The data is stored in both nodes that surround the hash-id directly. Let those nodes be a and b and the receiver r. If r is either a or b, then r knows also the other peer, because that is its predecessor or successor.

- If $r = a$ or $r = b$ the receiver will lookup the data and route a GetDataResult message containing it to the id stored in the field sender if the data has been found. Otherwise:
  - If $a \neq b$ and $r = a$ and the sender is not b then the receiver sends the request to b.

  - If $a \neq b$ and $r = b$ and the sender is not a then the receiver sends the request to a.
  - If either $a = b$ or the sender is either a or b, a GetDataResult message containing no data is routed back to the id stored in the field sender.

- If the receiving node knows a super-peer between the hash-id and itself, it will forward the message to that peer like a routed message.

When the receiver gets this message it looks up the data for the given type and key and sends it back to the sender with a GetDataResult message. This message must be answered even when no data is found.

GetDataResult(sender:ID,hash:ID,type:DataType,key:Data,value:Data?) [0x22]

This message is a reply to a GetData message. The sender field contains the ID of the sender of the GetData message. This message will be routed back to the id in the sender field.

If the data has been found then the field value is present and contains that data.

TestReachability(testee:ChordAddr,(forwarder:ChordAddr,pubAddr:Address)?) [0x18]

This message is used to test the reachability of a node. In each test, two of these messages are sent.

The first is sent from the testee to a forwarding peer and does not contain the forwarder and pubAddr fields. When the forwarding peer receives this message, it sets the forwarder field to its own address and the pubAddr to the Internet address of the testee as visible in the connection socket.

The second message is sent from the forwarding peer to the testing peer. The testing peer will try to open a connection to the testee on the address stored in pubAddress. If that works it will send a positive ReachabilityResult through that connection. Else it will send a negative ReachabilityResult back to the forwarding peer.

ReachabilityResult(testee:ChordAddr,reachability:IsReachable,pubAddr:Address) [0x19]

This message transmits the result of reachability analysis.

When a node receives this message and is not the one mentioned in the testee field it must forward the message to that node if it is directly connected to that node.

When the testee receives this message it will save the result and store the public address inside its chord address object.

Message(src:ID,(bdst:BroadcastDst|rdst:RoutingDst),data:Data,meta:MetaData?) [0x30]

This is a general message to transmit application messages. Either a broadcast or a routing destination must be present but not both. Meta contains meta data that can be used to establish special ways of message delivery.

This message must be routed towards the targets. If the path towards the targets splits, the message must also be split. That means the message is copied and the destination field is split so that any id in the old destination is included in the destination field of exactly one

new message and no ids are included in any new destination field that are not present in the original destination.

The destination could contain flags that influence the routing.

When the AllSuper broadcast flag is set, super-peers will process the message, otherwise they must only forward the message but not process it. When the AllEdge broadcast flag is set, the message must be forwarded to edge-peers and they will process the message, otherwise the message will not be forwarded to edge-peers and they must not process it.

When a message is sent to a non-existent node, this can be detected in a node when the target id lies between itself and its predecessor or successor. The flags of the destination determine how such a situation should be handled. If the LeftPeer (RightPeer) flag is set, the message is sent to the super-peer that immediately precedes (succeeds) the destination id in the id ring. That node must detect that it is the left (right) neighbor of the destination and that the flag is set. It must process the message as it was received normally.

When the Undeliverable flag is set, a message sent to a non-existent node must be handled by sending an UndeliverableMessage to the original sender containing the destination and the original data and meta data.

Note: Any combination of the flags is valid. If no flags are set, a message to a non-existent node is simply dropped. If both the LeftPeer and RightPeer flags are set, both neighbors must receive and process the message. It is important the a node does not send a message back to the node it came from because of both flags being set. Also when one neighbor a receives such a message with both flags set, it must unset the flag that could cause the other neighbor b to resend the message to a when it receives it.

UndeliverableMessage(src:ID,dst:RoutingDst,data:Data,meta:MetaData?) [0x31]

This message informs the sender that its message did not reach the destination. This is routed to the id stored in the src field.

The dst field contains a list of unreachable destination ids. If part of the destination ids of a routed message can not be reached, then only those are mentioned in the UndeliverableMessage. This message is not sent for broadcast messages.

The data and meta fields are copied from the original message.

ShortcutIndication(peer:ChordAddr,traffic:Traffic) [0x07]

This message indicates that from the receiving peer to the parameter peer a certain traffic has been measured. When a node receives such a message it should check if a shortcut connection to that peer would improve routing.

5 Behavior

Implementations should keep a list of „important" connections. Important connections are all finger connections and shortcut connections. Those connections are kept alive with Ping messages and if an important connection is lost unintentionally it should be re-built without taking normal actions for lost connections if that succeeds.

Since messages could get lost when connections break or nodes crash, nodes should use timeouts when they wait for replies.

5.1 Network joining

When a node wants to join the network it is supposed to contact a node inside the network and send a TestReachability message and a FindJoinPeer message.

Figure 13: positive reachability test

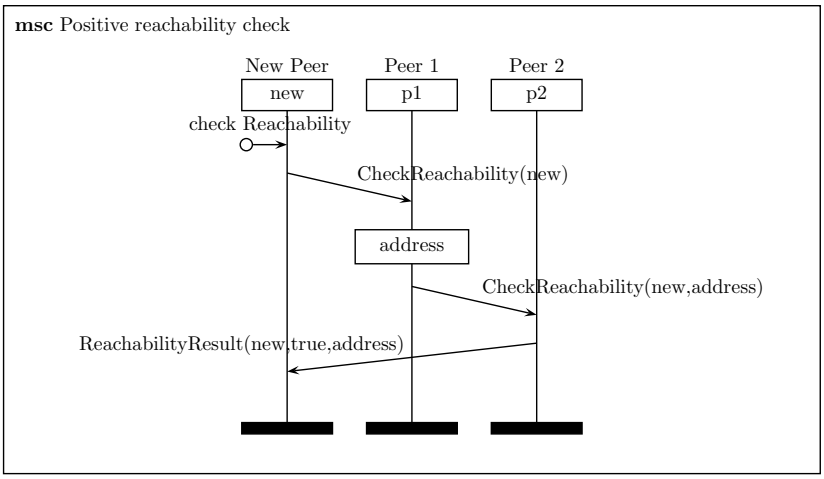
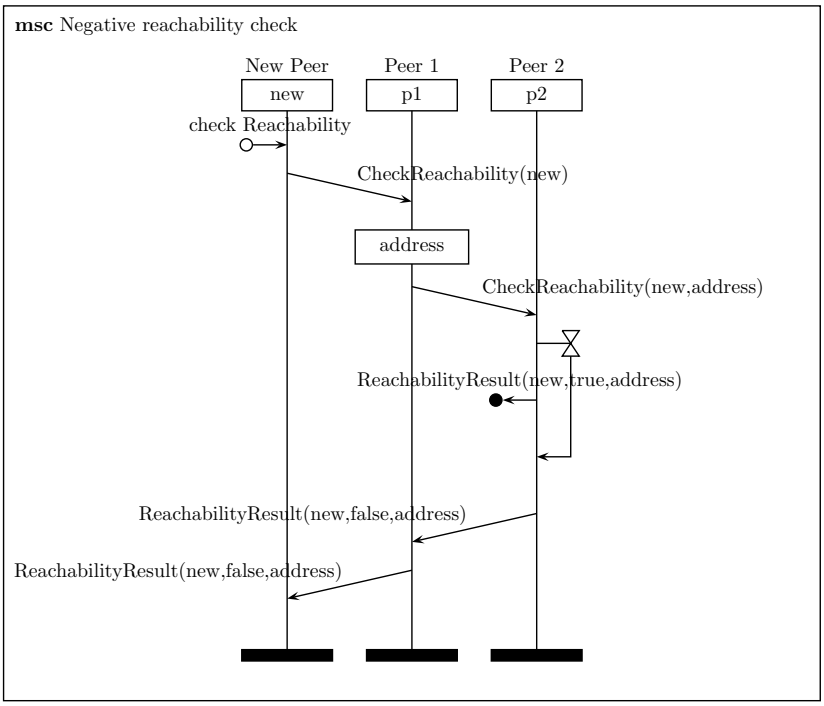
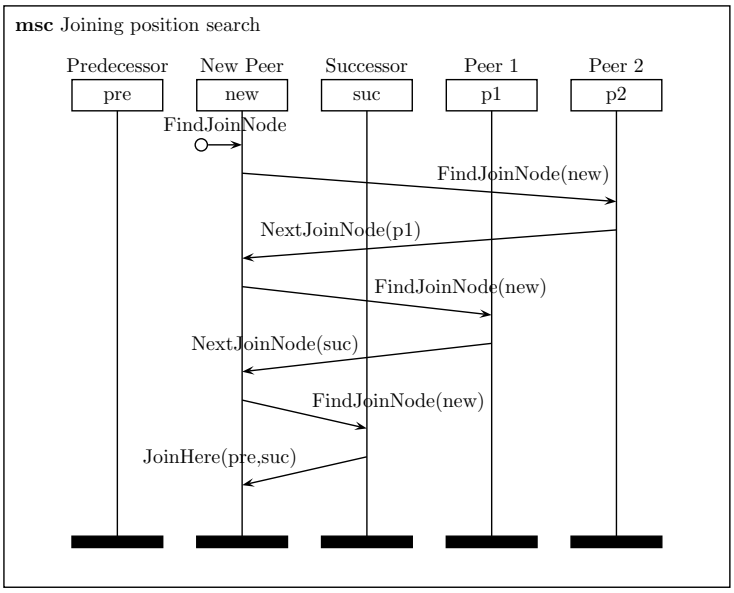


Figure 14: negative reachability test



The TestReachability message starts the reachability analysis and will finally lead to a ReachabilityResult message containing the reachability state and the public address.

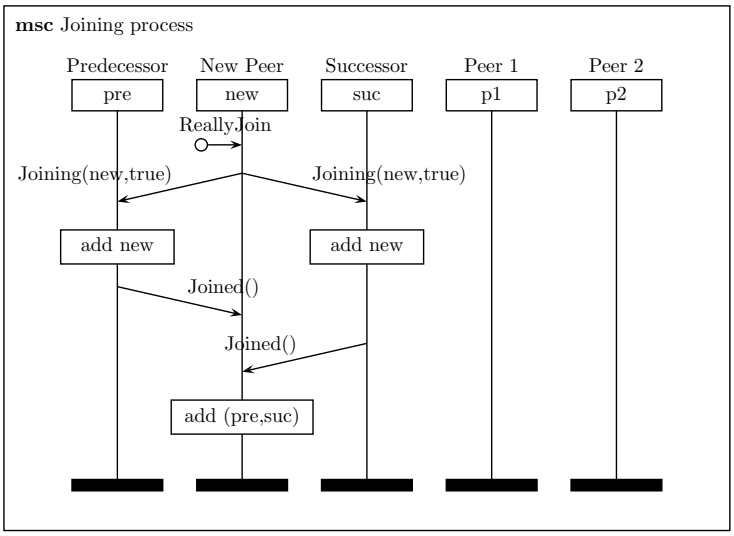
Figure 15: Joining phase 1



The FindJoinPeer message starts off the join position search and will finally lead to a JoinHere message containing the predecessor and successor node.

When the reachability analysis and the position search are finished the node can join the network by sending Joining messages to its predecessor and successor.

Figure 16: Joining phase 2

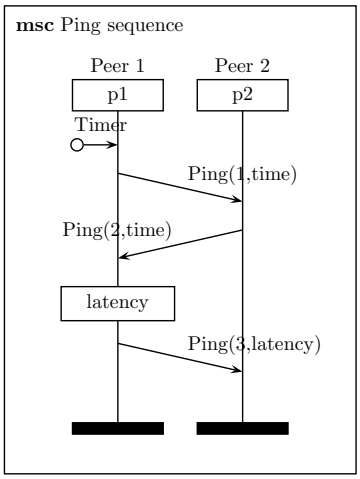


5.2 Optimization/Maintenance

The optimization and maintenance consists of two message sequences that are executed periodically. How often each sequence is executed might depend on the node life cycles.

The ping-pong-peng sequence is used to measure the distances to all connected peers and to keep the connections alive. All nodes send periodically ping messages to all of their „important" connections. Those distances include not only network latency but also load costs of the peers.

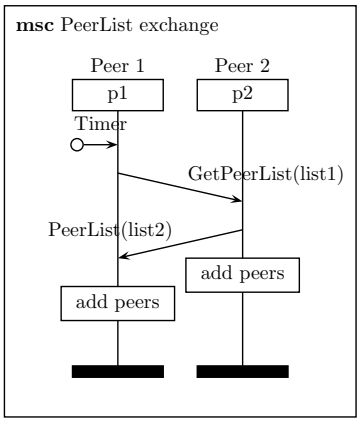
Figure 17: Ping-pong-peng sequence



(Note that all messages are named ping and the stage is given as parameter)

The PeerList exchange sequence is used to optimize the finger table. All nodes send periodically a GetPeerList message to all of their fingers. This message contains the finger table of the sending peer and the answer contains the finger table of the destination which can be used to optimize the local finger table.

Figure 18: Peer list exchange



Algorithm 7 finger table optimization

```
optimizeFingers ( Peer peer, FingerList fingers, Peer self ) {
  for ( int i = fingers.getMinId(); i <= fingers.getMaxId(); i++ ) {
    if ( i != 0 ) optimizeFinger ( peer, fingers, i, self );
  }
}

optimizeFinger ( Peer peer, FingerList fingers, int fingerId, Peer self ) {
  Peer fi = fingers.get(fingerId);
  if ( fi != null ) {
    if ( peer.equals(fi) ) continue;
    ID pos = fingers.fingerPosition(fingerId);
    if ( peer.distanceTo(pos) != peer.distanceTo(self) + self.distanceTo(pos)
      && ( fi.distanceTo(pos) == fi.distanceTo(self) + self.distanceTo(pos) )
        || peer.distanceTo(pos) < fi.distanceTo(pos) )
       fingers.set(fingerId, peer);
  } else fingers.set(fingerId, peer);
}
```

Note: When an edge-peer changes a super-peer during optimization, it must send a Joining message to that new super-peer.

5.3 Connection loss

When a connection is lost and can not be reestablished, the peer must be removed from the finger table and from the edge-peer list.

The remaining entries in the finger table can be used to fill the resulting gaps. Also an immediate finger table update should be initiated to find good substitutes for the lost peer.

Algorithm 8 lost peer algorithm

```
peerLost ( Peer p, FingerList fingers, Peer self ) {
  for ( int i = fingers.getMinId(); i <= fingers.getMaxId(); i++ ) {
    if ( i == 0 ) continue;
    if ( fingers.get(i).equals(p) ) fingers.set(i,null);
  }
  for ( int i = fingers.getMinId(); i <= fingers.getMaxId(); i++ ) {
    if ( i == 0 ) continue;
    Peer fi = fingers.get(i);
    if ( fi == null ) {
      for ( int j = fingers.getMinId(); j <= fingers.getMaxId(); j++ ) {
       Peer fj = fingers.get(j);
       if ( j != 0 ) optimizeFinger ( fj, fingers, i, self );
      }
    }
  }
}
```

5.4 Network parting

When a node with the state Inside or Joining wants to leave the network it has to execute the following steps:

1. Send a Parting message on all connections. After this step the node is not part of any finger table (that also means it has no edge-peers).
2. Forward DHT data to neighbors with StoreData messages.

When all steps have been executed, the node may close its connections and shut down.

6 Experiments & Simulations

6.1 Routed distances and hop count

To analyze the average routed distances and hop counts inside the chord network, simulation series with 30 runs have been executed. Those series use the Meridian[18] latency matrix.

6.1.1 Plain bi-chord routing

Series one researches the plain ChordNet architecture with plain bi-chord routing.

Figure 19: Series 1

Node count	100	200	300	400	500	600	700	800	900	1000
Avg. routed distance	2.45	2.77	2.95	3.08	3.19	3.28	3.35	3.42	3.47	3.53
Avg. hop count	2.43	2.77	2.95	3.09	3.19	3.28	3.36	3.42	3.48	3.53

These results show a small routed distance which comes close to the expected $\frac{\log_2 n}{3}$. Also the standard deviation is very low so the probability is high that these values will be reached.

6.1.2 PRS routing extension

Series two and five research the impact of normal and advanced PRS routing on these values.

Figure 20: Series 2 (normal PRS)

Node count	100	200	300	400	500	600	700	800	900	1000
Avg. routed distance	2.35	2.60	2.76	2.89	2.99	3.06	3.13	3.18	3.23	3.28
Decrease in %	3.83	6.13	6.35	6.15	6.53	6.75	6.80	6.89	7.00	7.24
Avg. hop count	2.95	3.36	3.61	3.78	3.91	4.03	4.12	4.21	4.28	4.34
Increase in %	21.13	21.42	22.27	22.48	22.51	22.77	22.83	22.97	23.07	23.10

These results show that normal PRS routing is able to decrease the routed distance by over 7% while increasing the hop count by over 20%.

Figure 21: Series 3 (advanced PRS)

Node count	100	200	300	400	500	600	700	800	900	1000
Avg. routed distance	2.34	2.64	2.81	2.94	3.03	3.10	3.16	3.22	3.27	3.32
Decrease in %	4.11	4.63	4.68	4.73	5.24	5.47	5.68	5.75	5.75	6.02
Avg. hop count	2.49	2.82	3.01	3.14	3.25	3.34	3.41	3.48	3.54	3.59
Increase in %	2.21	1.81	1.82	1.81	1.76	1.76	1.74	1.74	1.75	1.72

Series 3 shows that advanced PRS routing reduces the routed distance a little less than normal PRS but has no significant hop count increase.

6.1.3 NoN routing extension

Series four researches the impact of NoN routing.

Figure 22: Series 4 (NoN routing)

Node count	100	200	300	400	500	600	700	800	900	1000
Avg. routed distance	2.59	2.77	2.94	3.05	3.13	3.21	3.27	3.32	3.37	3.42
Decrease in %	-6.06	-0.15	0.22	1.17	2.11	2.15	2.49	2.79	2.89	3.15

These results show that NoN routing can reduce the average path length significantly.

6.1.4 Combined PRS-NoN routing

Series five and six research the impact of combined PRS and NoN routing and APN routing.

Figure 23: Series 5 (normal PRS + NoN)

Node count	100	200	300	400	500	600	700	800	900	1000
Avg. routed distance	2.26	2.47	2.61	2.70	2.77	2.83	2.87	2.91	2.95	2.99
Decrease in %	7.42	10.79	11.43	12.42	13.31	13.67	14.29	14.70	15.06	15.38
Avg. hop count	3.18	3.69	3.99	4.20	4.36	4.50	4.61	4.71	4.80	4.88
Increase in %	30.78	33.53	35.09	35.86	36.36	37.09	37.39	37.64	37.96	38.43

These results show a noticeable improvement made by combining NoN and PRS routing extensions but also an enormous hop count increase.

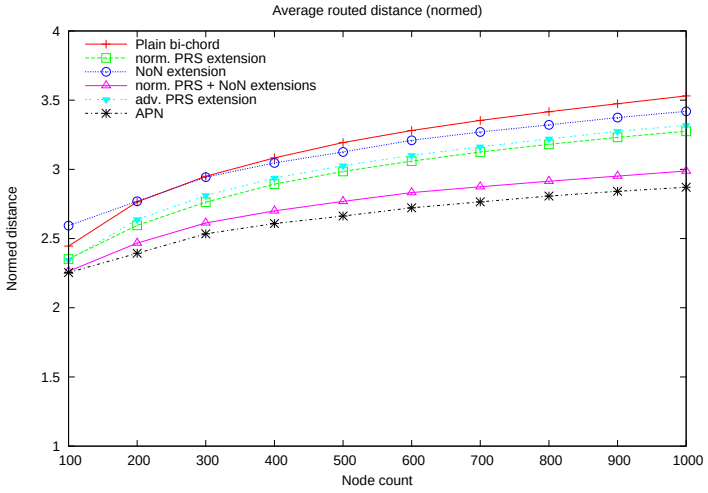
Figure 24: Series 6 (APN routing)

Node count	100	200	300	400	500	600	700	800	900	1000
Avg. routed distance	2.25	2.39	2.53	2.61	2.66	2.72	2.76	2.81	2.84	2.87
Decrease in %	7.81	13.43	14.10	15.38	16.62	17.03	17.56	17.84	18.22	18.69
Avg. hop count	2.53	2.75	2.93	3.04	3.13	3.21	3.28	3.33	3.38	3.43
Increase in %	4.05	-0.64	-0.63	-1.55	-2.16	-2.15	-2.40	-2.58	-2.66	-2.76

These results show a huge improvement made by APN routing while the hop count is even decreased.

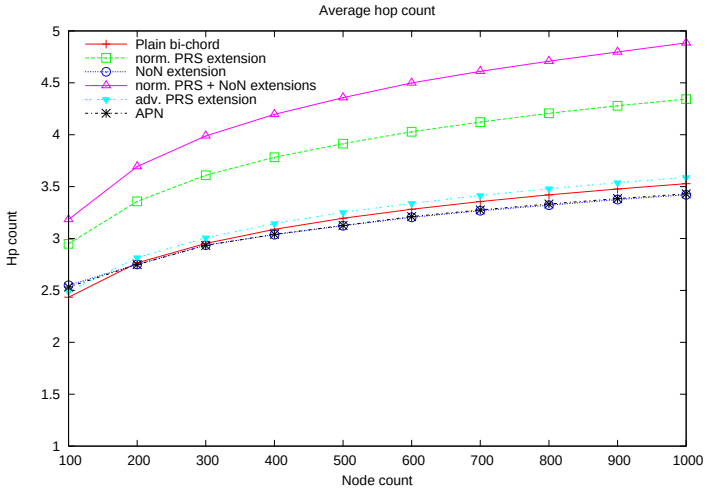
6.1.5 Comparison

Figure 25: Comparison of normed distances



This figure shows that NoN routing has a different complexity than normal routing. It also shows that APN routing is the best method measured in routing distance.

Figure 26: Comparison of hop counts



The hop count is significantly increased by normal PRS routing while advanced PRS routing has nearly no influence on hop counts.

6.2 Finger table size

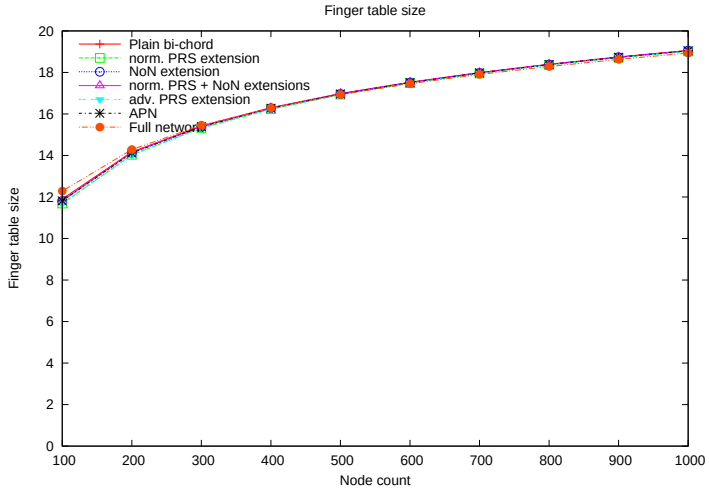
To research the average number of different finger nodes, 30 runs have been executed with different node counts. The id space was $\left[0 \dots 2^{60} - 1\right]$, so 120 finger slots are theoretically available per node.

Figure 27: Finger table size

Node count	100	200	300	400	500	600	700	800	900	1000
Avg different Fingers	11.91	14.17	15.42	16.30	16.98	17.53	18.00	18.40	18.75	19.06

These results show that the size of the network id space has no influence on the number of different fingers when the ids are evenly distributed. The expected value for a full network of N nodes is $2\left(\log_2 N\right) - 1$ as the most extreme fingers are always identical.

Figure 28: Finger table size



If the finger count is too high for a node, the finger definitions can be changed to

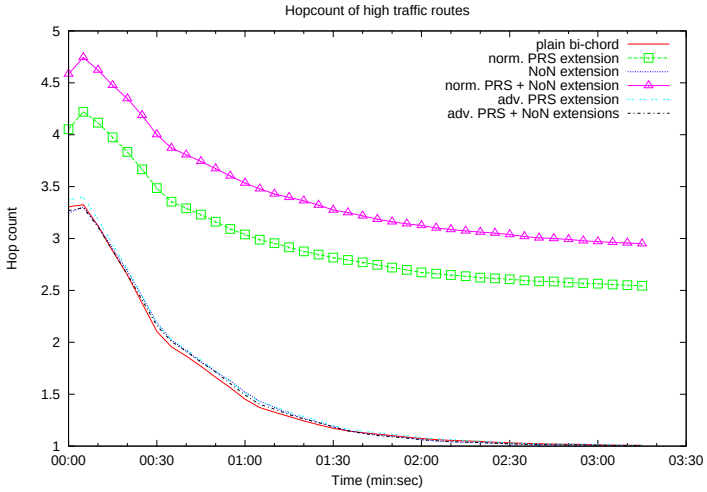
$$fingerpos_i(a) = \begin{cases} (a - 4^{-i-1}) \, mod \, N & i < 0 \\ (a + 4^{i-1}) \, mod \, N & i > 0 \end{cases}$$

to reduce the number of fingers by half at the cost of increased routing distances and hops.

6.3 Shortcuts

Simulations show that shortcut connections are able to reduce the distance and hop count on high traffic routes extremely.

Figure 29: Hop-count with shortcut connections



This figure shows that normal PRS conflicts with shortcut optimization. This happens because the total gain is calculated by the fraction of id-distance gain divided by the costs of the connection. When a finger spans half the distance towards the destination, a shortcut to the destination itself can only be twice as good in skipped id-distance. If the costs for the finger are less than half of those for the shortcut, normal PRS routing will use the finger although the shortcut goes directly to the destination.

This does not happen without PRS and with advanced PRS. With those routing methods the hop count is reduced to 1. In the experiment it took 2:30 minutes to reach 1 hop distances (the shortcut notice interval was 30 seconds).

7 Conclusion

This specification presents a peer-to-peer overlay network with efficient message delivery and maintenance. The network scales well even for large node numbers.

The protocol is platform-independent and easy to implement. Future extensions are easy to include as the protocol uses an extensible TLV format and feature signaling.

Several routing methods and optimizations have been described and analyzed.

7.1 Choice of routing method

Since all those routing methods guarantee the convergence criterion in all routing steps, different routing methods can be combined in one network.

Normal PRS routing optimization should not be used since it hugely increases the hop count. An increased hop count means increased global network load and increased failure probability. Also normal PRS routing conflicts with shortcut connections.

Under normal conditions APN routing should be used as it has the shortest routing distances without increasing the hop count.

7.2 Future work

The protocol specification will be extended to include end-to-end protocols similar to ICMP, UDP and TCP. For those protocols the MetaData field in the message type Message will be used.

Also publish/subscribe communication with topics and message queues as defined by JMS[4] will be specified. These communication types will also use the MetaData field.

Future work will research the possibilities to calculate routing tables on every update of the finger list and keep the routing table static for each message delivery. This could reduce the CPU costs of the routing methods.

The maintenance method used in ChordNet, the APN routing method and the shortcut connections will be analyzed in detail in future works.

Proximity identifier selection might offer an opportunity to further reduce routing distances. When network coordinates of a few nodes are known other nodes could calculate their own coordinates [3; 13]. Based on those coordinates, joining nodes could select an identifier that yields shorter routing distances than a random identifier. Mapping from coordinate to id might be done by binary partitioning the coordinate space as in [10]. This will be researched in a separate work.

[7] proposes a distributed certification system that could be used for ChordNet to establish trusted networking.

[4] Java Message Service (`http://java.sun.com/products/jms/`)

References

1 Hongwei Chen and Zhiwei Ye. Bchord: Bi-directional routing dht based on chord. In *CSCWD*, pages 410–415. IEEE, 2008.

2 Gennaro Cordasco, Luisa Gargano, Mikael Hammar, and Vittorio Scarano. Degree-optimal deterministic routing for p2p systems. In *ISCC*, pages 158–163. IEEE Computer Society, 2005. ISBN 0-7695-2373-0.

3 Manuel Costa, Miguel Castro, Antony I. T. Rowstron, and Peter B. Key. Pic: Practical internet coordinates for distance estimation. In *ICDCS*, pages 178–187. IEEE Computer Society, 2004.

4 Prasanna Ganesan and Gurmeet Singh Manku. Optimal routing in chord. In J. Ian Munro, editor, *SODA*, pages 176–185. SIAM, 2004.

5 P. Krishna Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of dht routing geometry on resilience and proximity. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors, *SIGCOMM*, pages 381–394. ACM, 2003. ISBN 1-58113-735-4.

6 Junjie Jiang, Ruoyu Pan, Changyong Liang, and Weinong Wang. Bichord: An improved approach for lookup routing in chord. In Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam, editors, *ADBIS*, volume 3631 of *Lecture Notes in Computer Science*, pages 338–348. Springer, 2005. ISBN 3-540-28585-7.

7 François Lesueur, Ludovic Mé, and Valérie Viet Triem Tong. A distributed certification system for structured p2p networks. In David Hausheer and Jürgen Schönwälder, editors, *AIMS*, volume 5127 of *Lecture Notes in Computer Science*, pages 40–52. Springer, 2008. ISBN 978-3-540-70586-4.

8 Yin Li, Xinli Huang, Fanyuan Ma, and Futai Zou. Building efficient super-peer overlay network for dht systems. In Hai Zhuge and Geoffrey Fox, editors, *GCC*, volume 3795 of *Lecture Notes in Computer Science*, pages 787–798. Springer, 2005. ISBN 3-540-30510-6.

9 Gurmeet Singh Manku, Moni Naor, and Udi Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized p2p networks. In László Babai, editor, *STOC*, pages 54–63. ACM, 2004. ISBN 1-58113-852-0.

10 Peter Merz and Matthias Priebe. A new iterative method to improve network coordinates-based internet distance estimation. In *ISPDC*, pages 169–176. IEEE Computer Society, 2007.

11 Peter Merz, Steffen Wolf, Dennis Schwerdel, and Matthias Priebe. A self-organizing super-peer overlay with a chord core for desktop grids. In Karin Anna Hummel and James P. G. Sterbenz, editors, *IWSOS*, volume 5343 of *Lecture Notes in Computer Science*, pages 23–34. Springer, 2008. ISBN 978-3-540-92156-1.

12 Moni Naor and Udi Wieder. Know thy neighbor's neighbor: Better routing for skip-graphs and small worlds. In Geoffrey M. Voelker and Scott Shenker, editors, *IPTPS*, volume 3279 of *Lecture Notes in Computer Science*, pages 269–277. Springer, 2004. ISBN 3-540-24252-X.

13 T. S. Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM*, 2002.

14 Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

15 Xiao-Jin Ren, Guo-An Wang, Zhi-Min Gu, and Zhi-Wei Gao. A fast joining operation for highly dynamic chord system. In *ICPADS*, pages 1–5. IEEE Computer Society, 2007.

16 Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001. ISBN 3-540-42800-3.

17 Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

18 Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: a lightweight network location service without virtual coordinates. In Roch Guérin, Ramesh Govindan, and Greg Minshall, editors, *SIGCOMM*, pages 85–96. ACM, 2005. ISBN 1-59593-009-4.

19 Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

20 Stefan Zöls, Zoran Despotovic, and Wolfgang Kellerer. Cost-based analysis of hierarchical dht design. In Alberto Montresor, Adam Wierzbicki, and Nahid Shahmehri, editors, *Peer-to-Peer Computing*, pages 233–239. IEEE Computer Society, 2006. ISBN 0-7695-2679-9.