Diplomarbeit

Eine Kommunikations-Infrastruktur für Middleware in Super-Peer-basierten Desktop Grids

Dennis Schwerdel
Matrikelnummer 348458

30. September 2008

Betreuer:
Juniorprofessor Dr.-Ing. Peter Merz
AG Verteilte Algorithmen
Fachbereich Informatik
Technische Universität Kaiserslautern

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In Zeiten von Dual-Core und Quad-Core Desktop-Prozessoren haben Computer sehr viel Rechenleistung, die bisher weitgehend ungenutzt bleibt. Mit Hilfe moderner Peer-to-Peer-Netzwerke[23] können diese Computer dezentral vernetzt werden. Dadurch wird das Potenzial vieler zusammen arbeitender Desktop-Rechner erstmals nutzbar, um große, verteilte Arbeiten auszuführen.

Die Einsatzgebiete eines solchen Desktop Grids sind zahlreich. In den Bereichen Informatik, Biologie, Physik und Maschinenbau[13] wird Rechenleistung für schwer zu lösende Probleme benötigt. So könnte ein weltweites Desktop Grid entstehen[25]. Freiwillige Internetnutzer können ihre überschüssige Rechenleistung zur Verfügung stellen. Forschungseinrichtungen können sich anschließen, um bei Bedarf im gemeinsamen Netzwerk schnell verteilte Berechnungen auszuführen. Unternehmen könnten mit der freien Rechenleistung der Desktoprechner ihrer Angestellten am Desktop Grid teilnehmen und die Rechenkapazität des Grids in ihrer Entwicklungsabteilung verwenden. Ein solches globales Desktop Grid könnte eine Alternative zu Super-Computern werden[3].

In dieser Arbeit wird eine Peer-to-Peer-Kommunikations-Infrastruktur entworfen, die es erlaubt, ein solches Desktop Grid zu realisieren. Aufgabe der Infrastruktur ist es, mit den einzelnen Rechnern ein robustes, dezentrales und effizientes Netzwerk aufzubauen und zu verwalten.

Zu diesem Zweck wird zuerst in Kapitel 2 eine abstrakte Netzwerkschicht erstellt und darauf aufbauend die Kommunikations-Infrastruktur. Die Netzwerkstruktur, die den Kern des Netzwerks bildet, wird in Kapitel 3 entwickelt. In Kapitel 4 wird das Verfahren zum Optimieren des Netzwerks vorgestellt. Kapitel 5 beschreibt, wie die verteilte Liste aller Knoten des Kernnetzwerks verwaltet wird. Kapitel 6 beschäftigt sich mit einem Dienst, der die Verbindung zum Netzwerk wieder aufnimmt, wenn diese abbricht. Permanente IDs werden in Kapitel 7 entwickelt. Kapitel 8 beschreibt, wie all diese Module zusammenwirken, um das Gesamtnetzwerk zu erstellen.

In Kapitel 9 wird die Funktionsweise des Netzwerks anhand verschiedener Szenarien näher erklärt. Die Testergebnisse des Netzwerks in realen Situationen werden in Kapitel 10 dargestellt.

1.2 Netzwerktopologie

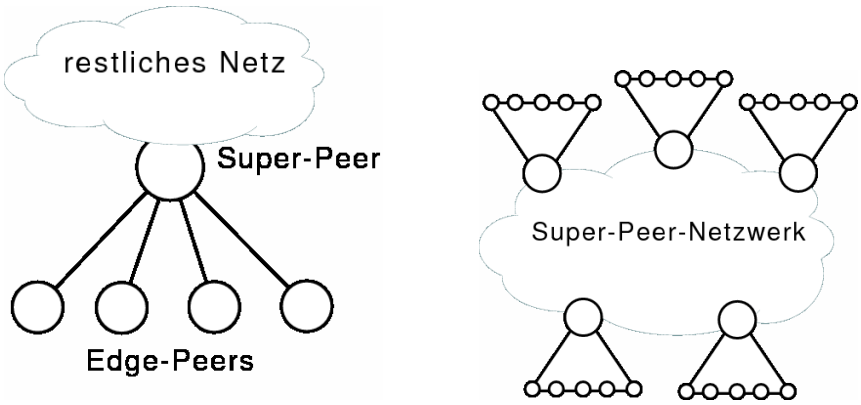
Das Netzwerk besteht aus Rechnern (genannt Knoten, Nodes oder Peers), die über das Internet vernetzt sind. Die Knoten können jederzeit das Netzwerk verlassen oder ausfallen. Daher ist es nicht möglich, zentrale Komponenten zu verwenden, ohne die Ausfallsicherheit des Gesamtnetzwerks zu verlieren.

Es kann auch vorkommen, dass Knoten durch NAT[1] oder Firewalls nicht von anderen Knoten erreichbar sind. Zu diesem Zweck gibt es zwei Klassen von Knoten: Super-Peers[27] und Edge-Peers. Super-Peers können von anderen Knoten erreicht werden und

[1] Network Address Translation

übernehmen Infrastruktur-Aufgaben im Kern des Netzwerks. Sie sind für die Weiterleitung von Nachrichten zuständig und stellen so Kontakt zwischen Knoten her, die sich gegenseitig nicht erreichen können. Edge-Peers sind Knoten, die keine Infrastruktur-Aufgaben übernehmen. Sie sind mit einem Super-Peer verbunden und kommunizieren ausschließlich über ihn.

Abbildung 1: Netzwerkstruktur



(a) Netzwerkstruktur aus Sicht eines Edge-Peers

(b) Netzwerkstruktur mit mehreren Super-Peers

1.3 Knotenidentifikationen

Zur Identifikation von Knoten gibt es mehrere Datenklassen und Adresstypen.

Node beinhaltet die IP-Adresse und den Server-Port des Knotens.

PersistentId ist die eindeutige, dauerhafte ID eines Knotens. Die ID besteht aus einem kurzen Byte-Array. So kann die ID sehr flexibel generiert werden:

- aus einem Zufallsgenerator
- aus einem eindeutigen Namen mit `String.getBytes()`
- als Fingerprint eines öffentlichen Schlüssels

Chord-ID ist eine eindeutige Nummer eines Super-Peers. Alle Edge-Peers eines Super-Peers und der Super-Peer selbst haben die gleiche Nummer. Die Bedeutung der Nummer wird in Kapitel 3 näher beschrieben.

PeerAddress besteht aus der PersistentId und der Chord-ID des Knotens.

SCPeer beinhaltet das Node-Objekt, das PeerAddress-Objekt und die Vivaldi-Koordinaten des Knotens. Vivaldi wird in Kapitel 4.1 näher beschrieben.

1.4 Anforderungen

Die Aufgabenstellung fordert ein robustes, selbst-organisierendes Netzwerk über das effizient Nachrichten versendet werden können. Die Software zur Realisierung des Netzwerks soll als Programmbibliothek (Library) von anderen Programmen einfach benutzbar sein. Um plattformunabhängig zu sein, soll die Library in der Programmiersprache Java geschrieben werden. Dabei soll die Realisierung problemneutral sein. Die Nutzung der Library soll nicht durch Lizenzen eingeschränkt werden.

1.5 Related Work

BOINC [1] ist ein Netzwerk zur Jobverteilung und wird insbesondere von SETI@home [2] benutzt. Im Gegensatz zu dem hier entwickelten Netzwerk wird aber ein starrer Client-Server-Ansatz verwendet.

Active MQ [4] bietet das Verteilen von Nachrichten in einem statischen Netzwerk.

Chord wurde in [24] beschrieben. In diesem Paper wird der Aufbau eines Chord-Netzes und die Verfahren zum Erstellen des Netzwerks und zum Routen von Nachrichten vorgestellt. Das beschriebene Netzwerk ist zum Speichern von Daten gedacht und wurde diesbezüglich untersucht.

Das Verfahren zur Auswahl von Super-Peers (SPSA) wurde in [16] beschrieben. Chord-SPSA [18] ist eine Anpassung von SPSA speziell für die Verwendung mit Chord. Während SPSA von einem vollständig verbundenen Super-Peer-Netz ausgeht und mit dieser Annahme das Netzwerk optimiert, berücksichtigt Chord-SPSA die Eigenschaften der Chord-Struktur.

Vivaldi, ein Verfahren zur Berechnung von Netzwerkkoordinaten, wurde in [10; 8] beschrieben. Ein zu dem hier verwendeten Broadcast-Verfahren ähnliches Verfahren wurde in [14] entwickelt. Ein Vorgänger des hier entwickelten Netzwerks wurde in [17] vorgestellt. PlanetLab [9; 6; 21] ist ein Forschungsgrid mit mehreren hundert Knoten, das für die Tests verwendet wurde. Die Middleware für die Jobverteilung, die langfristig auf dieser Netzwerkstruktur realisiert werden soll, wird in [15] beschrieben.

2 Abstrakte Netzwerkschicht

Um das Netzwerk zu realisieren, wurde zuerst eine abstrakte Netzwerkschicht geschaffen. Das abstrakte Netzwerk bildet das Versenden von Nachrichten zwischen Knoten auf das reale Netzwerk ab. Das reale Netzwerk mit Sockets, IP-Adressen und Ports wird auf die Kommunikation zwischen Knoten abstrahiert.

Eine wesentliche Funktionalität des abstrakten Netzwerks ist das Versenden von Java-Objekten als Nachrichten. Dadurch soll eine Abstraktionsstufe erreicht werden, die es erlaubt, einfach verteilte Anwendungen zu erstellen.

2.1 Annahmen über das Netzwerk

Für die Realisierung wird vorausgesetzt, dass das Netzwerk Daten in Paketen versendet, und dass dabei kein Vertauschen, Duplizieren oder Verfälschen dieser Pakete auftritt. Wenn ein sichtbarer Paketverlust auftritt, wird dies zeitnah erkannt und führt zu einem Verbindungsabbruch.

TCP[2] bietet all diese Merkmale, daher wurde die abstrakte Netzwerkschicht auf TCP aufbauend realisiert. TCP korrigiert Paketverluste selbst, sodass kein sichtbarer Paketverlust auftritt.

2.2 Realisierung

Jeder Knoten öffnet einen Serversocket, auf dem er erreichbar ist. Der Port des Sockets muss beim Öffnen angegeben werden. Für Edge-Peers ist der Serversocket lediglich ein Merkmal zur Unterscheidung mehrerer Instanzen auf dem gleichen Host. Die öffentliche IP-Adresse und der Server-Port bilden zusammen ein Node-Objekt.

```
1 public final class Node {
2   public InetSocketAddress socketAddress;
3 }
```

Diese Informationen können in jedem Knoten manuell konfiguriert oder automatisch mit STUN ermittelt werden. `STUN.request()` liefert dazu einen Datensatz der Klasse `PublicAddress`. Das STUN-Verfahren wird im nächsten Abschnitt näher beschrieben.

```
1 public class PublicAddress {
2   public boolean reachable;
3   public InetAddress publicAddress;
4 }
```

Jeder Knoten wartet auf der in seinem Node-Objekt angegebenen Adresse auf eingehende Verbindungen mit einem eigenen Thread der Klasse `AbstractNetwork.ServerThread`.

Für jede eingehende Verbindung wird ein eigener Verbindungs-Thread der Klasse `AbstractNetwork.Connection` erzeugt. Dieser Thread wartet auf eingehende Nachrichten auf dieser Verbindung und verarbeitet diese. Ein Fehler beim Empfang von Nachrichten wird als Verbindungsabbruch behandelt und die Verbindung wird geschlossen.

Um zu einem anderen Knoten eine Verbindung herzustellen, wird ein neuer Socket erstellt und ein neuer Thread der Klasse `AbstractNetwork.Connection` erzeugt, um diese Verbindung zu verwalten.

Alle Verbindungen eines Knotens werden in einer Zuordnungstabelle gespeichert, die einem Node-Objekt die zugehörige Connection zuordnet. Dadurch können bestehende

[2] Transmission Control Protocol, beschrieben in RFC 793

Verbindungen zum Versenden von Nachrichten verwendet werden. Da bei Verbindungen, die vom ServerThread erstellt wurden, kein Node-Objekt der Gegenseite bekannt ist (es fehlt die Information über den Port des Server-Sockets des Clients), wird dieses in einer Nachricht übertragen. Dazu wird beim Aufbauen einer Verbindung als erstes eine `HelloMessage` gesendet, die das Node-Objekt des Senders beinhaltet.

Beim Senden einer Nachricht wird eine bestehende Verbindung verwendet oder falls keine Verbindung zum Zielknoten besteht, eine neue Verbindung hergestellt, um die Nachricht zu senden. Der Benutzer des Netzwerks muss also nicht explizit Verbindungen herstellen, um Nachrichten zu übermitteln.

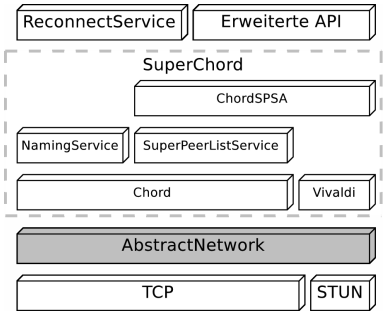
Um defekte Verbindungen schnell zu erkennen, wird für die Verbindung ein Lese-Timeout aktiviert. Die Länge des Timeouts kann mit der Konfigurationsvariablen `superchord.net.sotimeout` eingestellt werden. Wird innerhalb dieser Zeitspanne keine Nachricht empfangen, wird dies als Verbindungsabbruch behandelt. Diese Einstellung ist wichtig, da der für TCP-Verbindungen verwendete Timeout vom Betriebssystem abhängig ist.

Damit nun wichtige Verbindungen auch im Leerlauf gehalten werden können, ohne dass dieser Timeout sie trennt, wird ein Keep-Alive-Mechanismus verwendet. Der `KeepAliveThread` überprüft periodisch alle wichtigen Verbindungen und sendet, falls nötig, eine Nachricht der Klasse `HeartbeatMessage`. Empfängt ein Knoten eine solche Nachricht, sendet er diese, falls nötig, wieder zurück, um auf seiner Seite einen Timeout zu verhindern. Dadurch können Verbindungen einseitig als wichtig angesehen werden. Welche Verbindungen wichtig sind und welche nicht, muss von darüberliegenden Schichten definiert werden.

Zum Versenden von Nachrichten wird die Java-Serialisierung verwendet. Alle zu versendenden Objekte müssen daher `Serializable` sein. Die wichtigsten Nachrichtenobjekte sind zudem noch `Externalizable`, um die Performance zu steigern.

Das abstrakte Netzwerk ist vom Rest der Middleware unabhängig und kann auch als Basis für andere Anwendungen dienen. Die restliche Middleware baut auf dieser Grundlage auf.

Abbildung 2: Blockschaubild der Komponenten



2.2.1 STUN

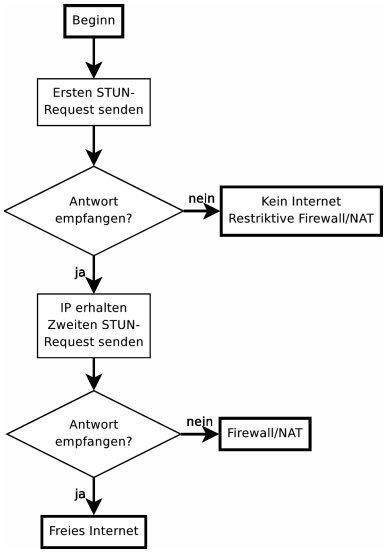
Zur Erkennung von NAT und Firewalls und um die externe Adresse eines Knotens zu ermitteln, wird STUN[3] verwendet. STUN ist heutzutage weit verbreitet und wird im Bereich "Voice over IP" sehr stark verwendet[11]. Das Ermitteln der externen Adresse könnte der Bootstrap-Peer übernehmen; das Erkennen von NATs bzw. Firewalls benötigt allerdings ein Netzwerk aus mehreren Computern. So könnte ein mit STUN vergleichbarer Dienst erst von einem Netzwerk mit zwei oder mehr Super-Peers erbracht werden.

Die Implementierung des STUN-Clients ist sehr simpel und unterstützt nur den benötigten Teil der Funktionalität von STUN. Die Funktionsweise von STUN wird in RFC 3489 beschrieben und hier nur kurz dargestellt.

Es werden zwei Anfragen per UDP[4] an den STUN-Server gesendet. Die erste Anfrage (Abb. 4b) wird mit einem Paket beantwortet, das die öffentliche Adresse beinhaltet (Abb. 4c). Kommt diese Antwort nicht innerhalb einer gewissen Zeitspanne an, so gilt das STUN-Verfahren als gescheitert, weil keine Internet-Verbindung besteht oder eine zu restriktive Firewall den Netzwerkverkehr unterbindet.

Wenn das Paket ankommt wird daraus die öffentliche IP-Adresse ausgelesen und ein weiteres Paket versendet (Abb. 4d). Dieses Paket erbittet eine Antwort von einer anderen Adresse (andere IP und anderer Port) als der Zieladresse des Paketes. Kommt die Antwort an, so ist der Knoten extern erreichbar, ansonsten blockiert eine Firewall oder ein NAT nicht angeforderte Pakete - also Pakete deren Absender nicht vorher kontaktiert wurde (Abb. 4e).

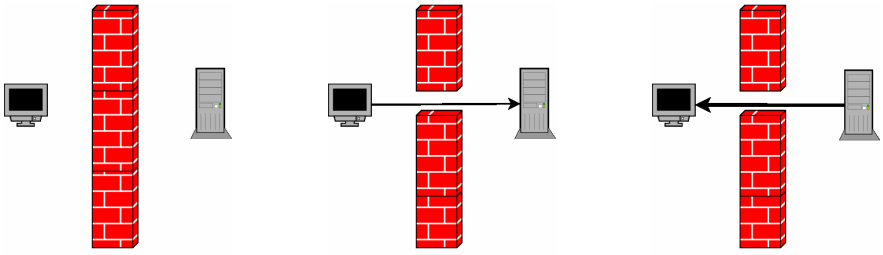
Abbildung 3: STUN-Verfahren



(a) Ablaufdiagramm: STUN

[3]Simple traversal of UDP over NATs
[4]User Datagram Protocol

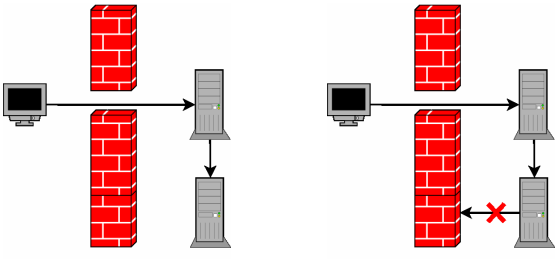
Abbildung 4: Erkennung von NAT



(a) Ausgangszustand

(b) Anfrage vom Client

(c) Antwort vom Server

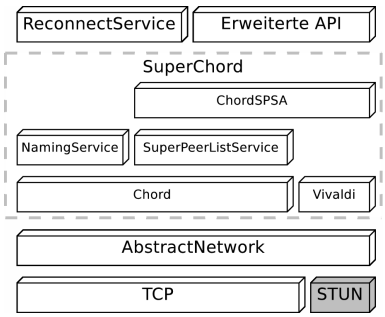


(d) 2. Anfrage vom Client

(e) Antwort vom 2. Server

Der verwendete STUN-Server und der verwendete Timeout können mit den Parametern `superchord.stun.server` und `superchord.stun.timeout` eingestellt werden. Mit den Parametern `superchord.public_address` und `superchord.reachable` können die ermittelten Werte manuell eingestellt bzw. überschrieben werden. Werden beide Parameter angegeben, wird STUN komplett deaktiviert.

Abbildung 5: Blockschaubild der Komponenten



2.2.2 Serialisierung

Um Java-Objekte über eine Netzwerk-Verbindung zu senden, müssen diese serialisiert werden. Hierzu gibt es mehrere Möglichkeiten.

XStream[7] bietet die Möglichkeit Java-Objekte in XML[5] zu konvertieren und so über das Netzwerk zu versenden. Um die XML-Elemente einzeln aus dem Datenstrom zu lesen, muss dann ein spezieller Parser wie z. B. XPP[6] genutzt werden. XML-Dom[7] kann nur vollständige Dokumente parsen und kann nicht genutzt werden, um mehrere XML-Elemente über eine Verbindung zu senden. XStream benötigt beim Serialisieren und Parsen sehr viel Speicher, der nur nach einiger Zeit wieder durch Garbage Collection freigegeben wird. Dadurch wird XStream für diese Anwendung unbenutzbar.

Java bietet mit `ObjectOutputStream`/`ObjectInputStream` selbst eine Möglichkeit, um Objekte zu serialisieren und zu versenden. Um die Größe der serialisierten Objekte zu minimieren, nutzt Java "Object Sharing". Dabei merken sich Sender und Empfänger alle bereits übertragenen Objekte. Soll ein Objekt dann ein zweites Mal übertragen werden, wird lediglich eine Referenz übertragen und dann beim Empfänger ersetzt. Für kurze Serialisierungen ist dies praktisch aber für dauerhafte Verbindungen wird dies zu einem Memory Leak, da der Objektspeicher ständig anwächst.

`ObjectOutputStream` kennt speziell zu diesem Zweck eine Methode, um Objekte "unshared" zu versenden. Tatsächlich werden, wie in der Spezifikation beschrieben, nun keine Objekte mehr gespeichert. Allerdings wird mit dieser Methode anstelle des Objektes eine Null-Referenz für jedes gesendete Objekt gespeichert, was ebenfalls ein Memory Leak darstellt. Dieses Fehlverhalten ist Sun seit Anfang 2007 bekannt[8], besitzt aber keine Priorität. Um auch dieses Memory Leak zu umgehen, muss nach jedem Versenden eines Objektes der Objektspeicher geleert werden. Alternativ dazu wurde ein inoffizieller Patch für die fehlerhafte Java-Klasse erstellt. Der Patch befindet sich im Anhang zu dieser Arbeit.

Abgesehen von den Problemen mit Memory Leaks, benötigt auch die Java-Serialisierung während ihrer Arbeit relativ viel Speicher. Um auch dieses Problem zu beheben, wird `Externalizable` verwendet, was ein Serialisieren einzelner Klassen per Hand erlaubt.

2.3 Interface

Das Interface des abstrakten Netzwerks und das Listener-Interface sind wie folgt definiert:

[5] Extensible Markup Language
[6] XML Pull Parser
[7] Document Object Model
[8] Sun Bug-Tracking-System, Bug ID: 6525563

```
1  public class AbstractNetwork {
2    public void bind(int port) throws IOException;
3    public Node getNode();
4    public void setListener(Listener l);
5    public static PublicAddress getPublicAddress();
6    public synchronized void openConnection(Node n);
7    public void closeConnection(Node n);
8    public Set<Node> getConnections();
9    public void close();
10   public boolean sendMessage(Node n, Message m);
11   public Integer getLatency(Node n);
12 }
```

Mit der Methode `bind(int port)` kann der Serversocket auf einen bestimmten Port gebunden werden. Wird 0 als Port angegeben, wird ein zufälliger, freier Port gewählt. Die Methode `getNode()` liefert das Knotenobjekt des Serversockets. Mit `setListener(Listener l)` kann man den Listener setzen, der über eingehende Nachrichten und Verbindungsverlust informiert. `getPublicAddress()` liefert die öffentliche IP-Adresse und die Information, ob der Server aus dem Internet erreichbar ist.

Mit den Methoden `openConnection(Node n)` und `closeConnection(Node n)` können Verbindungen explizit aufgebaut oder geschlossen werden. `getConnections()` liefert eine Menge aller verbundenen Knoten. Mit `getLatency(Node n)` kann die mit den Keep-Alive-Nachrichten gemessene RTT[9] in Millisekunden abgerufen werden.

Die Methode `sendMessage(Node n, Message m)` sendet eine Nachricht an einen Knoten und baut dafür, wenn nötig, eine Verbindung auf.

Mit `close()` kann das gesamte abstrakte Netzwerk beendet werden. Ein derart beendetes Netzwerk kann nicht mehr mit `bind()` gestartet werden, da der Thread, der Keep-Alive-Nachrichten sendet, mit `close()` ebenfalls beendet wird.

```
1  public interface Listener {
2    public void onConnectionBroken(Node n);
3    public void onIncomingMessage(Node n, Message m);
4  }
```

Wenn eine Verbindung ungewollt abbricht und nicht wieder hergestellt werden kann, wird die Callback-Methode `onConnectionBroken(Node n)` aufgerufen. Trifft eine neue Nachricht ein, wird `onIncomingMessage(Node n, Message m)` aufgerufen.

2.4 Nachrichtenübermittlung

Wird eine Nachricht an `sendMessage()` übergeben, wird zuerst überprüft, ob das Ziel der Knoten selbst ist, und die Nachricht dann direkt zugestellt. Danach wird bei Bedarf

[9] Round-Trip-Time

eine Verbindung mit `openConnection()` aufgebaut bzw. die Verbindung zum Zielknoten aus der Zuordnungstabelle `connections` geladen. Nun wird die Nachricht an `Connection.sendMessage()` übergeben.

In `Connection.sendMessage()` wird die interne Methode `Connection.writeMessage()` aufgerufen und dort die Nachricht serialisiert und übertragen.

Auf der Gegenseite wird die Nachricht von `Connection.run()` mit `Connection.readMessage()` wieder in ein Objekt umgewandelt. Wenn die Nachricht keine `HeartbeatMessage` ist, wird sie an `incomingMessage()` übergeben und dort an `Listener.onIncomingMessage()` weitergeleitet.

3 Chord-Verfahren

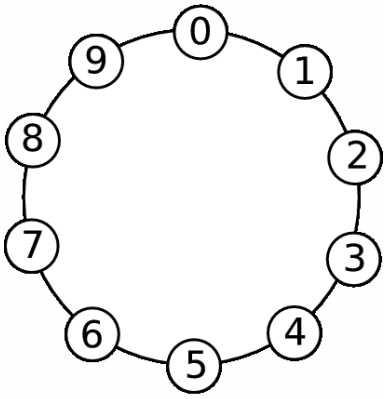
Chord wurde schon mehrfach in der Literatur behandelt und wird hier nur kurz vorgestellt. Teile dieses Kapitels sind aus [22] übernommen.

Chord ist ein strukturiertes P2P-Netzwerk [24]. Es bietet eine sehr robuste Struktur und effizientes Routing. Deshalb wird diese Struktur in abgewandelter Form für das Super-Peer-Netzwerk der Middleware verwendet. Da Chord in seiner eigentlichen Form eine DHT[10]-Struktur ist, wird hier eine abgewandelte Form dieser Struktur verwendet. Insbesondere wird im Normalbetrieb kein Hash-Verfahren angewendet, sondern direkt auf Nummern (IDs) gearbeitet.

3.1 Netzwerkstruktur

Für Chord müssen alle Knoten je eine ID aus einem bestimmten Bereich $[0..N-1]$ haben. Im Allgemeinen wird N als $N = 2^d$ für ein beliebiges $d \in \mathbb{N}$ gewählt. Der Exponent d kann mit dem Konfigurationsparameter `superchord.chord.dim` eingestellt werden. Die IDs sollten dabei gleichverteilt aus dem ID-Bereich gewählt werden. In Kapitel 4.3.1 wird anderes Verfahren zur Wahl von IDs beschrieben.

Abbildung 6: Ein vollbesetzter Chord-Ring mit N=10

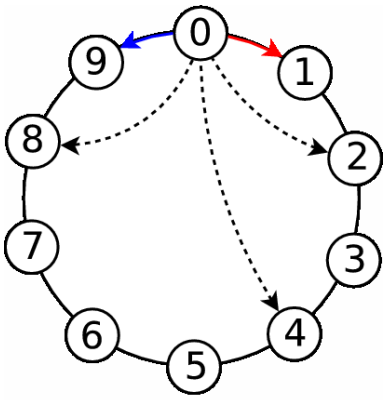


[10] Distributed Hash Table

Jeder Knoten hält eine Menge von Verbindungen F zu anderen Knoten, die Finger genannt werden. Es gibt $d = |F|$ Finger (F_0 bis F_{d-1}), die wie folgt definiert sind: F_i eines Knotens mit der ID k ist der Knoten, dessen ID $(k+2^i)\, mod\, 2^d$ beträgt oder als nächste auf diese folgt. Der erste Finger (F_0) wird auch Nachfolger (oder Successor) genannt, da er nach dieser Definition der Knoten mit der nächsten ID im Ring ist. Zusätzlich zu den Fingern wird noch ein Vorgänger-Knoten P (Predecessor) gespeichert, sodass der Nachfolger des Vorgängers eines Knotens dieser Knoten selbst ist.

Aus Successor- und Predecessor-Verbindung ergibt sich ein doppelt verbundener Ring, der alle Knoten beinhaltet; dieser Ring wird auch Chord-Ring oder Successor-Ring (in Successor-Richtung) genannt. Die restlichen Finger F_1bis F_{d-1} können als Abkürzungen in diesem Ring gesehen werden.

Abbildung 7: von Knoten 0 ausgehende Finger



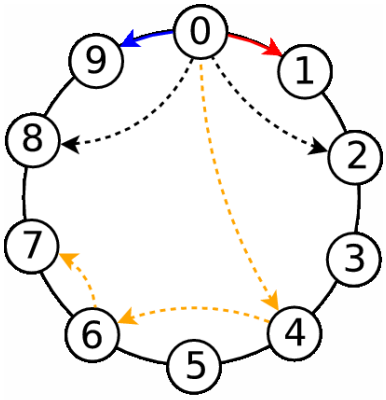
In Abbildung 7 ist ein vollbesetztes Chord-Netzwerk mit $N = 10$ Positionen dargestellt. Alle Verbindungen eines Knotens sind am Beispiel des Knotens 0 eingezeichnet. Knoten 0 hat die Finger $F_0 = 1$, $F_1 = 2$, $F_2 = 4$ und $F_3 = 8$. Der Finger F_0 zum Knoten 1 ist der Nachfolger (Successor). Knoten 9 ist der Vorgänger P (Predecessor) von Knoten 0.

3.2 Routing

Will Knoten S eine Nachricht an einen Knoten D senden, so wählt er den Finger F_i aus, der am nächsten an D liegt, aber noch im Bereich $(S..D]$ (in Richtung aufsteigender IDs) liegt. Zu diesem Finger wird nun die Nachricht weitergeleitet. Findet S keinen solchen Finger F_i (weil sogar sein Nachfolger F_0 hinter D liegt), muss er annehmen, dass es keinen Knoten mit der ID von D gibt. Im Worst-Case kann eine Nachricht immer über den Nachfolger weitergeleitet werden; die restlichen Finger beschleunigen also nur das Verfahren, sind aber nicht unbedingt nötig.

In Abbildung 8 wird der Routing-Verlauf einer Nachricht von Knoten 0 an Knoten 7 dargestellt. Im ersten Routingschritt wählt Knoten 0 aus seinen Fingern ($F_0 = 1$, $F_1 = 2$, $F_2 = 4$ und $F_3 = 8$) denjenigen aus, der am nächsten an Knoten 7, aber nicht dahinter liegt; das ist Knoten 4. Dieser wählt Knoten 6 aus seinen Fingern ($F_0 = 5$, $F_1 = 6$, $F_2 = 8$,

Abbildung 8: Routing-Schritte von Knoten 0 zu Knoten 7



$F_3 = 2$) aus. Knoten 6 leitet nun die Nachricht an Knoten 7 weiter.

3.2.1 Implementierung

```
1  boolean isBetween(long start, long key, long end) {
2    if (start <= end) return start <= key && key <= end;
3    return start <= key || key <= end;
4  }
```

```
1  Node closestPrecedingFinger(long nodeId) {
2    Node fi = self;
3    for ( Node f: finger ) if ( isBetween(fi.id, f.id, nodeId) ) fi = f;
4    return fi;
5  }
```

Das Routing-Verfahren beruht auf der Methode closestPrecedingFinger(). Diese Methode liefert immer einen Knoten, dessen ID zwischen der eigenen ID und der ID des Zielknotens liegt oder einen der beiden Knoten selbst. Es werden immer alle Finger überprüft und der beste ausgewählt. Passt kein Finger, so wird der Knoten selbst zurückgeliefert.

```
1 void routeMessage(RoutedMessage msg) {
2   Node node = closestPrecedingFinger(msg.toId);
3   if ( msg.toId != self.id && isBetween(predecessor, msg.toId, self) ) node
        = predecessor;
4   if ( node == self && msg.toId != self.id ) // handle non-existant target
5   else sendTo(node, msg);
6 }
```

Das Routingverfahren nutzt closestPrecedingFinger(), um den nächsten Knoten auszuwählen aus. Dieser liegt immer zwischen dem Zielknoten und dem Knoten selbst.

Wenn node = self gilt, kann die Nachricht nicht mehr weitergeroutet werden, da der Successor hinter dem Zielknoten liegt. Es wird im Chord-Netzwerk angenommen, dass zwischen einem Knoten und seinem Nachfolger keine Knoten existieren. Deshalb werden Nachrichten, die nicht für den routenden Knoten selbst bestimmt sind, aber nicht mehr geroutet werden können, normalerweise verworfen. Für bestimmte Nachrichtentypen, insbesondere Anwendungsdaten, erfolgt in diesem Fall eine Antwort vom routenden Knoten an den Absender.

Die dritte Zeile ist eine Optimierung. Wenn der Zielknoten zwischen dem Vorgänger und dem Knoten selbst liegt oder der Vorgänger der Zielknoten ist, wird die Nachricht zum Vorgänger geschickt. Dadurch wird das aufwändigere Routen in die Successor-Richtung vermieden. Im nächsten Abschnitt wird gezeigt, dass das Routen zum Vorgänger ohne diese Optimierung den Worst-Case darstellen würde.

3.2.2 Aufwand

Aufgrund der Konstruktion des Netzwerks überbrückt der Finger i eines Knotens einen ID-Bereich von 2^i, wenn er optimal positioniert ist. Hat das Netzwerk $N = 2^d$ viele ID-Positionen, so hat jeder Knoten d Finger – Finger 0 bis $d - 1$.

Wird eine Nachricht geroutet, so werden die Finger in absteigender Reihenfolge verwendet. D. h. der erste Routingschritt benutzt den Finger $d - 1$, der zweite den Finger $d - 2$ und der letzte den Finger 0. In jedem Routingschritt wird der jeweilige Finger nur benutzt wenn er nicht hinter der Ziel-ID liegt. So können mit d Routingschritten $2^{d-1} + 2^{d-2} \ldots + 2^1 + 2^0 = 2^d - 1 = N - 1$ viele IDs überbrückt werden.

Dadurch ergibt sich direkt der theoretische Worst-Case. Dies ist eine Nachricht an den Vorgänger, da hier $N-1$ viele IDs überbrückt werden müssen und so alle d Routingschritte verwendet werden müssen. D. h. im Worst-Case benötigt das Routing-Verfahren $d = log_2 N$ viele Routingschritte, also logarithmischen Aufwand in Bezug auf die ID-Anzahl. Der theoretische Fall des Vorgängers wurde im Routing-Verfahren speziell behandelt und benötigt nur einen Routingschritt.

Jeder Routingschritt verringert den Bereich der noch nicht überbrückten IDs mindestens um die Hälfte. Liegt die Ziel-ID in der hinteren Hälfte, muss der Routing-Schritt ausgeführt werden, ansonsten nicht. Die Wahrscheinlichkeit dafür beträgt bei gleichverteilten vollbesetzten IDs genau 50%. Daher benötigt das Routing-Verfahren im Mittel des Normalfalls

höchstens halb so viele Routingschritte wie im Worst-Case – also $\frac{1}{2}log_2N$ viele.

Im Normalfall ist das Netzwerk nicht voll besetzt. Sind von den N IDs weniger als $K = 2^k$ besetzt, beträgt der mittlere Abstand zweier benachbarter Knoten mindestens $D = \frac{N}{K} = 2^{d-k}$ IDs. Das bedeutet, dass im Mittel die letzten $d - k$ Routingschritte, nicht benötigt werden. Diese Finger überbrücken einen kleineren ID-Bereich als D, d. h. im durchschnittlichen Fall wurde der Zielknoten vorher bereits erreicht. Nun sind also nur noch $d - (d - k) = k$ viele Routingschritte relevant. Jeder dieser Schritte wird mit einer Wahrscheinlichkeit von 50% benötigt, was die erwartete Anzahl an Routingschritten auf $\frac{1}{2}log_2K$ reduziert.

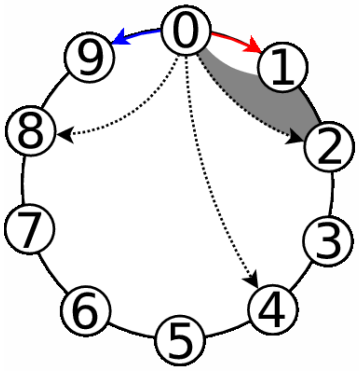
3.3 Broadcast

Das entwickelte Verfahren erlaubt es, eine Nachricht an einen begrenzten Bereich von IDs zu senden. Da der Bereich für IDs bekannt ist, lässt sich dies einfach für Broadcasts verwenden, indem der gesamte Bereich, exklusive der eigenen ID, adressiert wird.

Eine Broadcast-Nachricht enthält alle Eigenschaften einer direkt gesendeten Nachricht (Sender und Payload) und hat zusätzlich eine ID-Bereichsangabe (start und end). Wenn ein Knoten nun eine Broadcast-Nachricht empfängt, spaltet er den Broadcast-Bereich an den IDs seiner Finger auf und leitet den Broadcast an seine Finger mit diesen aufgespaltenen Teil-Bereichen weiter. Wenn ein Broadcast-Bereich nur noch einen Knoten beinhaltet, kann er nicht weiter aufgespalten werden, daher terminiert das Verfahren beim Empfänger, wenn der Broadcast-Bereich keinen anderen Knoten als ihn selbst beinhaltet.

Im Beispiel hat Knoten 0 die Finger $F_0 = 1$, $F_1 = 2$, $F_2 = 4$ und $F_3 = 8$. Er sendet einen Broadcast, indem er den Broadcast-Bereich auf [1..9] – also alle anderen Knoten – setzt.

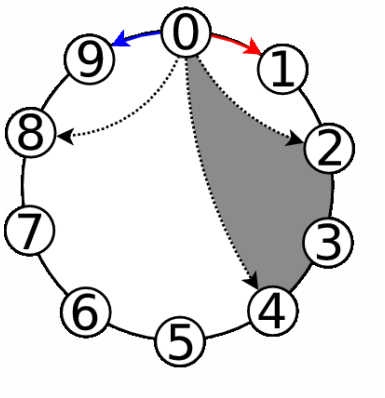
Abbildung 9: 1. Broadcast-Teilbereich



Den Bereich zwischen ihm und seinem Nachfolger kann er beim Broadcast ignorieren, da angenommen wird, dass keine Knoten in diesem Bereich existieren. Sein erster Finger F_0 ist der Knoten 1, was schon im Broadcast-Bereich liegt. Also beginnt der erste Teilbereich bei 1. Der Finger F_1 ist Knoten 2, was auch noch im Broadcast-Bereich liegt. Knoten

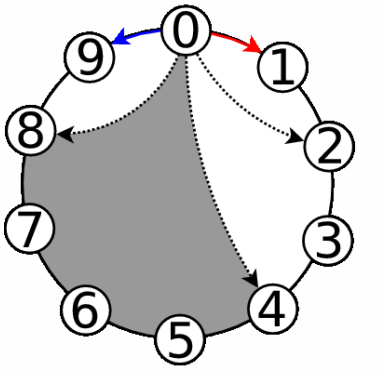
2 befindet sich bereits im zweiten Teilbereich; der erste Teilbereich endet vor Knoten 2. Knoten 0 sendet also einen Broadcast an Knoten 1 mit dem Broadcast-Bereich [1..1].

Abbildung 10: 2. Broadcast-Teilbereich



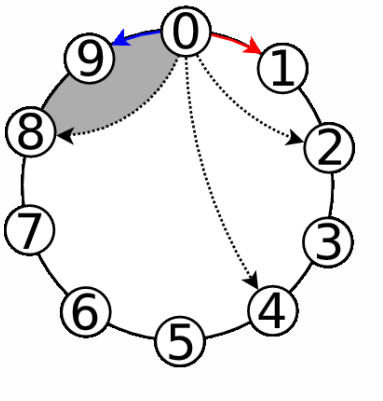
Es wurde bereits festgestellt, dass Knoten 2 der Beginn des nächsten Teilbereichs ist. Der nächste Finger nach Knoten 2 ist $F_2 = 4$. Dieser befindet sich noch innerhalb des Broadcast-Bereichs [1..9], daher ist das Ende des zweiten Teilbereichs und der Beginn des dritten Teilbereichs gefunden. Knoten 0 sendet eine Broadcast-Nachricht mit Broadcast-Bereich [2..3] an Knoten 2. Knoten 2 muss diesen Bereich weiter aufspalten, wenn er den Broadcast empfängt und selbst weiterleitet.

Abbildung 11: 3. Broadcast-Teilbereich



Knoten 4 ist der Beginn des dritten Teilbereichs. Der nächste Finger, $F_3 = 8$, ist noch im Broadcast-Bereich und wird der nächste Teilbereichstrenner. Die nächste Broadcast-Nachricht geht an Knoten 4 mit dem Bereich [4..7].

Abbildung 12: 4. Broadcast-Teilbereich



Knoten 8 ist der Beginn des letzten Teilbereichs. Da Knoten 8 der letzte Finger innerhalb des Broadcast-Bereichs ist, endet der Teilbereich am Ende des gesamten Broadcastbereichs. Der letzte Teilbereich lautet also [8..9].

Nun sieht man, dass der gesamte Broadcastbereich [1..9] in diesem Schritt überdeckungsfrei und vollständig in die Teilbereiche [1..1], [2..3], [4..7] und [8..9] aufgeteilt wurde.

Das Verfahren zum Weiterleiten eines Broadcasts funktioniert fast genau wie eben beschrieben. Der einzige Unterschied ist, dass der Broadcast-Bereich nicht erst ermittelt wird, sondern gegeben ist.

Die jeweiligen Finger führen dieses Verfahren mit den ihnen übergebenen Teilbereichen erneut durch, bis alle Bereiche nur noch einen Knoten beinhalten und der Broadcast terminiert.

3.3.1 Implementierung

```
1 public void broadcastMessage (PeerAddress src, Serializable msg) {
2   if ( !insideNetwork() ) return; // noone to broadcast to
3   // initialize the broadcast
4   forwardBroadcast(src, getSuccessor().id, getPreviousId(self().id),
        MessageType.Application, msg);
5 }
```

```
1  forwardBroadcast(PeerAddress fromId, long startId, long endId, MessageType
       type, Serializable payload) {
2    if ( successor == self ) return;
3    if ( isBetween(self.id, endId, successor.id) ) return;
4    if ( isBetween(self.id, startId, successor.id) ) startId = successor.id;
5    while ( isBetween(self.id, startId, endId) ) {
6      Node before = closestPrecedingFinger(startId);
7      long lastId = endId;
8      for ( Node f : finger ) if ( f != before ) {
9        if ( isBetween(startId, f.id, lastId) ) lastId = getPreviousId(f.id);
10     }
11     if ( before != self.id ) sendTo(before, new BroadcastMessage(fromId,
           startId, lastId, type, payload));
12     else break;
13     startId = getNextId(lastId);
14   }
15 }
```

Das Broadcast-Verfahren sendet Nachrichten an einen Bereich von Knoten-IDs. Beim Weiterleiten einer solchen Nachricht wird der Bereich in mehrere Teilbereiche aufgespalten. Die Teilbereiche beginnen immer mit einem Finger und enden vor dem nächsten Finger bzw. vor dem Knoten selbst.

Zeile 2 überprüft, ob ein Broadcast unmöglich ist, weil der Knoten nicht Teil eines Netzwerks ist. Zeile 3 bricht die Methode ab, falls das Ende des Broadcast-Bereichs vor dem Nachfolger liegt und Zeile 4 verschiebt den Anfang auf den Successor, da angenommen wird, dass zwischen einem Knoten und seinem Successor keine weiteren Knoten existieren.

In der Schleife werden die Teilbereiche zwischen den Fingern ermittelt und neue Broadcast-Nachrichten mit diesen Bereichen an die jeweiligen Finger versendet. Dazu wird zuerst der erste Knoten vor dem Beginn des Bereichs gesucht. Danach wird in der for-Schleife der erste Finger nach diesem Knoten, der aber noch in dem Bereich liegt, gesucht. Das Ende des Bereichs wird auf die ID direkt vor diesem Finger gesetzt; der Finger selbst ist der Beginn des nächsten Bereichs.

Als Optimierung könnte man das Ende des Broadcast-Bereichs auf den Vorgänger vorverlegen, falls es zwischen dem Vorgänger und dem Knoten liegt. Es ist unwahrscheinlich, dass in diesem Bereich Knoten liegen, da der Vorgänger durch die Stabilisierung und den Verbindungsaufbau gewartet wird. Allerdings werden dann, falls doch Knoten in diesem Bereich existieren, diese mit gerouteten Nachrichten erreicht, aber mit einem Broadcast nicht. Dies könnte auftreten, wenn ein neuer Knoten bereits Nachfolger seines Vorgängers ist aber noch nicht Vorgänger seines Nachfolgers.

3.3.2 Aufwand

Bei dem entwickelten Broadcast-Verfahren wird der ID-Bereich eines Broadcasts in jedem Schritt so aufgespalten, dass der Broadcast jede ID auf dem gleichen Weg erreicht, den auch eine geroutete Nachricht nehmen würde.

Dadurch benötigt das Broadcast-Verfahren so viele Schritte, um alle Knoten zu erreichen, wie das Routing-Verfahren. Allerdings muss hier die längste Strecke zu einer ID betrachtet werden; die Wahrscheinlichkeit von 50%, dass ein Routing-Schritt gespart werden kann, entfällt. So ergeben sich bei einem Netzwerk mit K Knoten im Mittel $log_2 K$ Schritte und im Worst-Case $log_2 N$ Schritte bei einem Netzwerk mit N IDs.

Vom Sender werden Nachrichten an alle anderen $K - 1$ Knoten gesendet. Die Routing-Pfade dieser Nachrichten bilden einen Spannbaum. Da auf gemeinsamen Pfaden die Nachrichten für verschiedene Knoten in einer Nachricht gesammelt versendet werden, werden genausoviele Nachrichten versendet, wie es Kanten in diesem Spannbaum gibt. Bei K Knoten im Netzwerk hat der Spannbaum $K - 1$ Kanten. Es werden also bei K Knoten immer $K - 1$ Nachrichten versendet.

Das hier verwendete Verfahren ist optimal in Bezug auf die Anzahl versendeter Nachrichten, wenn kein gemeinsames Medium genutzt werden kann.

3.4 Stabilisierung

Der Stabilisierungszyklus hat zwei Aufgaben:

- Den Successor-Ring aufrechterhalten
- Die Finger optimieren

Die Stabilisierung wird periodisch ausgeführt. Die Periodendauer wird an die Veränderungen des Netzwerks angepasst.

Finden innerhalb einer Periodendauer viele Veränderungen statt, wird die Stabilisierung häufiger ausgeführt. Die minimale Periodendauer und die Schwelle für die Verkürzung der Periodendauer können mit den Parametern `superchord.chord.min_stabilize_interval` und `superchord.chord.stabilize_high_changes` eingestellt werden. Der Faktor für die Veränderung des Intervalls in diesem Fall kann mit `superchord.chord.stabilize_high_changes_adjust` konfiguriert werden.

Wenn innerhalb einer Periodendauer wenig Veränderungen stattfinden, wird die Stabilisierung seltener ausgeführt. Die maximale Periodendauer und die Schwelle für die Verlängerung der Periodendauer können mit den Parametern `superchord.chord.max_stabilize_interval` und `superchord.chord.stabilize_low_changes` eingestellt werden. Der Faktor für die Veränderung des Intervalls in diesem Fall kann mit `superchord.chord.stabilize_low_changes_adjust` konfiguriert werden.

```
1  void stabilize() {
2    sendTo(getSuccessor(), MessageType.NodeInfo, self());
3    sendTo(getPredecessor(), MessageType.NodeInfo, self());
4    checkSuccessor(getPredecessor());
5    for ( ChordNode n : finger ) checkSuccessor(n);
6    fingerToCheck = (fingerToCheck + 1) % finger.length;
7    routeMessage(new RoutedMessage(con.getSelfAddress(), new PeerAddress (
         fingerStart(fingerToCheck)), MessageType.FindSuccessor, null,
         InvalidTargetPolicy.PredecessorOnInvalid));
8    sendTo(finger[fingerToCheck], MessageType.GetFingerList, null);
9    if ( fingerToCheck == 0 ) routeMessage(new RoutedMessage(con.
         getSelfAddress(), new PeerAddress (getPreviousId(self().getChordId())),
          MessageType.FindPredecessor, null, InvalidTargetPolicy.
         PredecessorOnInvalid));
10   sendTo(getPredecessor(), MessageType.GetFingerList, null);
11 }
```

In der zweiten Zeile und dritten Zeile sendet der Knoten Informationen über sich selbst an seinen Nachfolger und seinem Vorgänger. Dadurch soll der Nachfolger aktualisiert werden, falls ein Knoten zwischen dem Nachfolger und dem Knoten selbst liegt. Durch diese beiden Nachrichten wird der Successor-Ring optimiert.

Die Zeilen 4 und 5 suchen in allen bekannten Knoten nach einem besseren Nachfolger. Dies dient hauptsächlich der Rettung des Successor-Rings bei sehr hohen Fluktuationen.

In Zeilen 6, 7 und 9 wird für eine Finger-Position, bzw. die Predecessor-Position (Zeile 9) der passende Knoten mit einer gerouteten Nachricht gesucht. Der Finger, der gesucht wird, wird per Round-Robin gewählt. Der letzte Parameter gibt an, dass bei einer nicht existierenden ID (InvalidTarget) der erste existierende Vorgänger dieser ID die Nachricht bekommen soll.

Zeile 8 und 10 fordern die Fingerlisten von anderen Knoten an, um die eigene Fingerliste zu verbessern.

3.5 Verbindungsaufbau

Beim Verbindungsaufbau sendet ein neuer Knoten N eine FindJoinNode-Nachricht an seinen Bootstrapping-Peer, um seine zukünftige Position zu finden. Empfängt ein Knoten K eine solche Nachricht, so sucht er mittels closestPrecedingFinger() den Finger F_i aus, der am nächsten an N liegt. Nun können drei Fälle auftreten:

1. $F_i = N$: Ein Knoten mit der ID des neuen Knotens N existiert bereits. Um zu verhindern, dass zwei gleiche Knoten-IDs im Netzwerk vorhanden sind, wird N abgelehnt. N sollte dann zufällig eine neue ID ermitteln und das Verfahren erneut starten.

2. $K \neq F_i$: Der Zwischenknoten K hat einen Finger F_i gefunden, der näher an der zukünftigen Position von N ist als K. Da N nicht Teil des Netzwerks ist, kann K lediglich mit einer nextJoinNode-Nachricht über F_i informieren und die Verbindung

beenden. N wendet sich dann mit der FindJoinNode-Suche nach seinem Vorgänger an F_i.

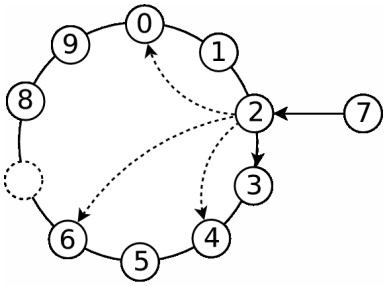
3. $K = F_i$: Die Methode closestPrecedingFinger() liefert K selbst. Das kann nur passieren wenn N zwischen K und seinem Nachfolger S liegt. Das wiederum bedeutet, dass der zukünftige Vorgänger von N gefunden ist. Nun wird N zwischen K und S eingefügt:

   - K sendet eine joinHere-Nachricht an N mit Informationen über S. Wenn N diese Nachricht empfängt, setzt er K als seinen Vorgänger ein und S als seinen Nachfolger.
   - K setzt N als seinen Nachfolger ein.
   - K sendet eine Nachricht an S mit Informationen über N. Wenn S diese Nachricht empfängt, setzt er seinen Vorgänger auf N.

Wenn N seine Position gefunden hat, beginnt er mit der Suche nach seinen Fingern.

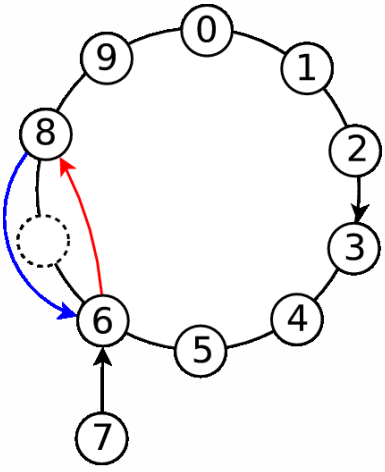
Das Verfahren konvergiert bei einer Netzwerkgröße von n immer nach spätestens $log_2(n)$ Schritten. Der Beweis ist analog zu dem Beweis der Routing-Komplexität, da ein neuer Knoten genau wie eine Nachricht durch das Netzwerk „geroutet" wird.

Abbildung 13: Verbindungsaufbau Schritt 1



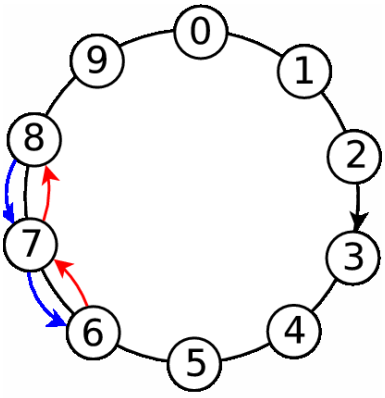
In diesem Beispiel will der neue Knoten 7 das Netzwerk betreten. Die Position 7 ist auch noch als einzige frei. Knoten 7 wendet sich mit einer findJoinNode-Nachricht an Knoten 2 als ersten Kontakt. Knoten 2 ermittelt Knoten 6 als nächsten Kontaktknoten mittels closestPrecedingFinger() und sendet demnach eine nextJoinNode-Nachricht mit Informationen über Knoten 6 an Knoten 7.

Abbildung 14: Verbindungsaufbau Schritt 2



Knoten 7 wendet sich dann als nächstes an Knoten 6. Dessen Nachfolger ist Knoten 8, deshalb liefert das closestPrecedingFinger()-Verfahren Knoten 6 selbst. Der Knoten beginnt also mit dem oben beschriebenen Verfahren und fügt Knoten 7 ein.

Abbildung 15: Verbindungsaufbau Schritt 3



Knoten 7 ist nun in das Netzwerk aufgenommen und beginnt die Suche nach seinen Fingern. Dazu sendet er eine Anfrage nach der Fingerliste an seinen Predecessor und Successor.

3.6 Interface

Die Klasse `Chord` enthält das beschriebene Chord-Verfahren und den aktuellen Zustand (z. B. Finger-Tabelle) eines Knotens.

```
1  class Chord {
2    Chord(ChordConnector con);
3    void setConnector(ChordConnector con);
4
5    static long getPreviousId(long id);
6    static long getNextId(long id);
7    int getTimeout();
8    ChordNode self();
9    ChordNode getPredecessor();
10   ChordNode getFinger(int i);
11   int getFingerCount();
12   ArrayList<ChordNode> getFingers();
13   ChordNode getSuccessor();
14   boolean insideNetwork();
15   long fingerStart(int i);
16   static boolean isBetween(long start, long key, long end);
17   ChordNode closestPrecedingFinger(long node);
18   ChordNode proximityClosestPrecedingFinger(long node);
19
20   void routeMessage(PeerAddress src, PeerAddress dest, Serializable msg,
         InvalidTargetPolicy policy);
21   void routeMessage(PeerAddress src, AddressList dest, Serializable msg,
         InvalidTargetPolicy policy);
22   void broadcastMessage(PeerAddress src, Serializable msg);
23   void limitedBroadcastMessage(PeerAddress src, Serializable msg, long count
       );
24
25   void join(Node node);
26   void leave();
27
28   void connectionLost(ChordNode peer);
29   void timeout();
30   void receiveMessage(ChordMessage msg);
31 }
```

Die Methoden `getNextId(long id)` und `getPreviousId(long id)` berechnen die nächste bzw. vorhergehende ID im Modulo-Ring (also $id + 1$ und $id - 1$, wenn kein Überlauf stattfindet). `fingerStart(int i)` berechnet die gewünschte Position des Fingers Nummer i.

Mit den Methoden `self()`, `getPredecessor()`, `getSuccessor()`, `getFinger(int i)`, `getFingerCount()` und `getFingers()` lassen sich Informationen über den Knoten selbst, den Vorgänger und Nachfolger und die anderen Finger auslesen. Mit `getTimeout()` lässt sich das Intervall für die Stabilisierungszyklen in Millisekunden auslesen.

Mit `routeMessage(...)`, `broadcastMessage(...)` und `limitedBroadcastMessage(...)` lassen sich Nachrichten versenden. Mit `join(Node node)` wird Kontakt zum Netzwerk hergestellt und mit `leave()` das Netzwerk verlassen.

Um Multicasts zur Verfügung zu stellen, gibt es eine Version der Methode `routeMessage(...)`, die eine Liste von IDs akzeptiert. Der Multicast wurde so implementiert, dass

die Liste der IDs, ähnlich wie bei Broadcasts, beim Routen aufgespalten wird.

Die Methoden `connectionLost(ChordNode peer)`, `timeout()` und `receiveMessage(ChordMessage msg)` werden aufgerufen, um Chord über die entsprechenden Ereignisse zu informieren, wobei `timeout()` das Stabilisierungsverfahren auslöst. In Kapitel 8 wird näher beschrieben, wie Chord in das Gesamtnetzwerk eingebunden ist.

```
1  interface ChordConnector {
2    void sendMessage(Node dst, ChordMessage msg);
3    void setTimeout(int millis);
4    ChordNode getSelfNode();
5    PeerAddress getSelfAddress();
6    Random getRandom();
7    void onApplicationMessage(PeerAddress sender, PeerAddress target, Object
         payload);
8    void onDuplicateId(ChordNode node);
9    void onInvalidTarget(PeerAddress address, Object originalMessage);
10   void closeConnection(ChordNode node);
11   void onJoined();
12   void onNewChordPeer(ChordNode peer);
13   double getDistance(ChordNode peer);
14   int numberOfChordPeers();
15 }
```

Das Interface `ChordConnector` wird benötigt, damit Chord (das Chord-Verfahren) die Umwelt über bestimmte Ereignisse informieren, bzw. Informationen bekommen kann.

Die Methoden `sendMessage(Node dst, ChordMessage msg)` und `setTimeout(int millis)` werden benötigt, damit Chord Nachrichten senden und einen Timeout setzen kann. Mit `getSelfNode()` und `getSelfAddress()` kann Chord Informationen über seinen eigenen Knoten bekommen.

Das Event `onApplicationMessage` wird ausgelöst, wenn Chord eine Nachricht an den eigenen Knoten bekommt. `onDuplicateId` wird aufgerufen, wenn die eigene ID bereits im Ring vorhanden ist und der Knoten das Netz nicht betreten konnte. Das Event `onInvalidTarget` wird ausgelöst, wenn eine von diesem Knoten versendete Nachricht nicht zugestellt werden konnte. `onJoined` und `onNewChordPeer` bedeuten, dass der Knoten bzw. ein anderer Knoten das Netzwerk erfolgreich betreten hat.

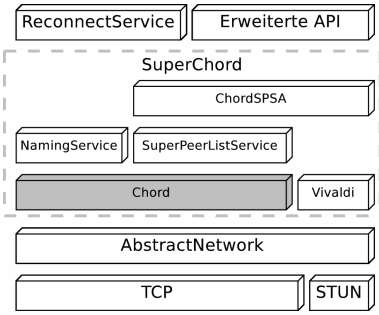
Die Methode `closeConnection(ChordNode n)` wird aufgerufen, wenn Chord viele Verbindungen in Folge öffnet, um anzuzeigen dass diese wieder geschlossen werden können. Mit `getDistance(ChordNode peer)` kann Chord die von Vivaldi geschätzte Distanz zu einem Knoten ermitteln.

```
1  interface ChordNode extends Serializable {
2    long getChordId();
3    Node getNode();
4    boolean chordEquals(ChordNode other);
5  }
```

3.7 Einbettung in das Netzwerk

Die Implementierung von Chord ist abgekapselt, um die Verfahren einfach und wiederverwendbar zu halten. Zu diesem Zweck benutzt die Klasse Chord keine Klassen des Gesamt-Netzwerks außer `PeerAddress`, `Node` und `SCPeer`. Die Klasse `SCPeer` wird durch das Interface `ChordNode` benutzt.

Abbildung 16: Blockschaubild der Komponenten



Die Hauptklasse des Netzwerks ruft Methoden von `Chord` auf und wird über das Callback-Interface `ChordConnector` über Events informiert.

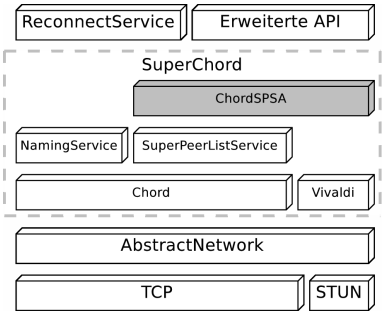
4 Chord-SPSA

SPSA[16] ist ein Verfahren, um in einem Super-Peer-Netzwerk Super-Peers auszuwählen und jedem Edge-Peer einen Super-Peer zuzuweisen. Das Verfahren nutzt Vivaldi-Koordinaten [10; 8], um Latenzen zwischen den Peers abzuschätzen und so ein optimales Netz zu erzeugen. Chord-SPSA [18] ist eine Verbesserung des SPSA-Verfahrens, die speziell auf die Netzwerkstrukturen mit einem Chord-Netzwerk als Kern zugeschnitten ist. Zudem bietet Chord-SPSA mit PIS[11] und PRS[12] noch zwei Verbesserungen für Chord selbst, die ebenfalls in [18] beschrieben sind.

Im Gegensatz zu dem in der Literatur beschriebenen Verfahren, wird hier auf das Versenden der Hub-Liste (hier Super-Peer-Liste) beim Betreten des Netzwerks verzichtet. Ein Knoten betritt das Netzwerk indem er Edge-Peer eines beliebigen Super-Peers wird. Er bekommt dann direkt die Super-Peer-Liste seines Super-Peers gesendet und wird sich im ersten Reorganisationsintervall automatisch den besten Super-Peer aussuchen. Wird ein Edge-Peer anstellte eines Super-Peers kontaktiert, lehnt dieser die Verbindung ab und sendet in der entsprechenden Nachricht die Super-Peer-Liste mit.

[11] Proximity Identifier Selection
[12] Proximity Route Selection

Abbildung 17: Blockschaubild der Komponenten



4.1 Vivaldi-Verfahren

Vivaldi ist ein Verfahren zur Bestimmung von mehrdimensionalen Netzwerkkoordinaten, mit denen Latenzen geschätzt werden können.

Das Verfahren basiert auf einem Modell, in dem die Knoten mit Federn verbunden sind. Die Längen der Federn entsprechen den gemessenen Netzwerklatenzen. Die Koordinaten der Knoten ergeben sich dann als Position im Modell, wenn ein Kräftegleichgewicht aller Federn herrscht.

Vivaldi wird in Zyklen ausgeführt. In jedem Zyklus wird die Entfernung (Latenz) zu einem anderen Peer gemessen und die Koordinaten entsprechend angepasst. Die Messung erfolgt in einer Ping-Pong-Sequenz und eine weitere Peng-Nachricht informiert den Sender der Pong-Nachricht über das Ergebnis. Wenn ein Messergebnis vorliegt, wird es mit der Distanz der Koordinaten verglichen und dann die Koordinaten entlang des Distanzvektors angepasst.

Vivaldi selbst verwendet einen Faktor, um zu verhindern, dass neue Peers mit ihren ungenauen Koordinaten die bestehenden Netzwerkkoordinaten von anderen Peers stark verändern. Dieser Mechanismus wurde durch eine Fehlermessung ersetzt. Bei dem Vergleich zwischen Messwerten und Koordinatendistanzen wird mit einem gleitenden Mittelwert ein relativer Fehler ermittelt. Beim Anpassen der Koordinaten wird der Fehler des Knotens selbst und der des Partnerknotens der Messung mit eingerechnet. Dadurch wird unabhängig von der Laufzeit eines Knotens erreicht, dass Distanzen zu stark fehlerbehafteten Koordinaten nicht zu starken Veränderungen der eigenen Koordinaten führen und andererseits stark fehlerhafte Koordinaten stark korrigiert werden.

Empfängt ein Knoten mit Koordinaten $\overrightarrow{K_1}$ und Fehlerwert e_1 ein Vivaldi-Update von einem anderen Knoten mit Koordinaten $\overrightarrow{K_2}$ und Fehlerwert $e_2$und einer gemessenen Distanz von d, passt er seine Daten wie folgt an: Zuerst wird ein Richtungsvektor $\overrightarrow{D} = \overrightarrow{K_2} - \overrightarrow{K_1}$ berechnet und dann auf die Länge von 1 normiert: $\overrightarrow{D_1} = \overrightarrow{D}/|\overrightarrow{D}|$. Aus der Differenz der geschätzten und der gemessenen Distanz wird eine Verzerrungskraft berechnet: $f = |\overrightarrow{D}| - d$. Aus den Fehlerwerten wird ein Anpassungsfaktor berechnet: $\Delta = \frac{e_1}{e_1+e_2}$. Nun werden die Koordinaten angepasst: $\overrightarrow{K_1} := \overrightarrow{K_1} + \overrightarrow{D_1} * f * \Delta * MAXFORCE$, wobei MAXFORCE ein weiterer Anpassungsfaktor ist, der die Wirkung herunterskaliert. Zusätzlich wird der eigene Fehlerwert angepasst: $e_1 := e_1 * (1 - \Delta * AVERAGINGSPEED) + \frac{f}{|\overrightarrow{D}|} * \Delta *$

AVERAGINGSPEED. AVERAGINGSPEED ist ein Faktor der angibt, wie stark der aktuell gemessene Fehlerwert den gespeicherten Fehlerwert beeinflusst.

```
1  void onReceiveMessage(Node src, VivaldiMessage msg) {
2    switch ( msg.stage ) {
3      case Ping:  sendTo(src, VivaldiMessage.pong(myCoord, msg.millis,
           coordError));
4                  break;
5      case Pong:  long diff = System.currentTimeMillis() - msg.millis;
6                  update(msg.coord, (double)diff, msg.coordError);
7                  sendTo(src, VivaldiMessage.peng(myCoord, diff, coordError));
8                  break;
9      case Peng:  update(msg.coord, msg.millis, msg.coordError);
10                 break;
11   }
12 }
```

```
1  void update(Coord other, double dist, double otherCoordError) {
2    Coord dir = Coord.distVector(myCoord, other);
3    dir = dir.div(dir.vectorLength());
4    double force = getDist(other) - dist;
5    double delta = Math.min(1.0, coordError/(coordError+otherCoordError));
6    Coord x = dir.mul(force*delta*MAX_FORCE);
7    myCoord.set(myCoord.add(x));
8    coordError = (1-delta*AVERAGING_SPEED)*coordError+delta*AVERAGING_SPEED*
       Math.abs(force/dist);
9  }
```

Die Anzahl der Dimensionen der Koordinaten und die Länge der Periode kann mit den Konfigurationsparametern `superchord.vivaldi.dim` und `superchord.vivaldi.interval` eingestellt werden.

4.2 Reorganisation

Die Reorganisation des Chord-SPSA versucht durch periodische Verbesserungen die Gesamtlatenz zu verringern und ein bestimmtes Verhältnis von verschiedenen Fingern zu Edge-Peers in jedem Super-Peer zu erreichen.

Chord-SPSA verwendet dazu Daten aus der Liste aller Super-Peers, der Super-Peer-Liste. Kapitel 5 beschreibt, wie diese Liste erstellt und gepflegt wird.

4.2.1 Edge-Peers

In jedem Zyklus versucht ein Edge-Peer, einen besseren Super-Peer zu finden. Dabei benutzt er die Vivaldi-Koordinaten aus der Liste aller Super-Peers, um den Super-Peer mit der kürzesten Distanz zu finden. Findet er einen besseren Super-Peer, so verbindet er sich mit diesem.

4.2.2 Super-Peers

Super-Peers versuchen ein vorgegebenes Verhältnis von Edge-Peers pro Finger-Knoten zu erreichen. Zu diesem Zweck können die Schranken $L < U$ angegeben werden.

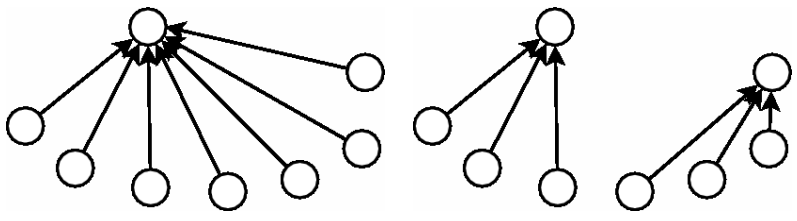
Fällt der Wert unter L, so erfolgt ein Downgrade, d. h. der Super-Peer wird zum Edge-Peer und er und alle seine Edge-Peers verbinden sich zu einem anderen Super-Peer. Um Massen-Downgrades bei steigenden Super-Peer-Zahlen zu verhindern, muss die Downgrade-Bedingung mehrere Zyklen bestehen, damit ein Downgrade stattfinden kann. Die Anzahl der Zyklen wird als Zufallswert zwischen 0 und einem Maximalwert ermittelt. Der Maximalwert kann mit dem Parameter `superchord.chordspsa.downgrade_wait_max` eingestellt werden.

Hat ein Super-Peer U mal so viele Edge-Peers wie Finger-Knoten, so ernennt er einen seiner Edge-Peers zum Super-Peer. Dazu wählt er einen Edge-Peer aus, der bei optimaler Wahl der Chord-ID minimale Distanzen zu seinen Fingern hat. Kandidaten für ein sogenanntes Upgrade zum Super-Peer müssen allerdings zwei Bedingungen erfüllen:

1. Sie müssen von anderen Peers erreichbar sein, also nicht durch NAT o. Ä. behindert werden.

2. Nach dem Upgrade müssen die Edge-Peers des bisherigen Super-Peers sich so aufteilen, dass sowohl der alte Super-Peer als auch der Kandidat lebensfähig sind, also die Bedingung zum Downgraden nicht erfüllen.

Abbildung 18: Aufspaltung eines Super-Peers



(a) vorher (b) nachher

```
1  public void onReceiveMessage(PeerAddress from, MessageScope scope,
       Serializable msg) {
2    if ( ! ( msg instanceof Message ) ) return;
3    Message m = (Message)msg;
4    switch ( m.type ) {
5      case Downgrading:
6        if ( ! parent.hasSuperPeer() || ! parent.getSuperPeer().getAddr().
             equals(from) ) return;
7        parent.joinAsEdgePeer(bestSuperPeer().getNode());
8        break;
9      case Upgrade:
10       if ( ! parent.hasSuperPeer() || ! parent.getSuperPeer().getAddr().
             equals(from) ) return;
11       parent.joinAsSuperPeer(parent.getSuperPeer().getNode());
12       upgradedPeer = from;
13       break;
14     case Upgraded:
15       for ( SCPeer ep : parent.getEdgePeers() )
16        if ( ep.distanceTo(m.peer) < ep.distanceTo(parent.getSelf()) )
                              parent.sendUnicastMessage(ep.getAddr(), new Message
                ( MessageType.ConnectTo, m.peer));
17       break;
18     case ConnectTo:
19      if ( ! parent.hasSuperPeer() || ! parent.getSuperPeer().getAddr().
            equals(from) ) return;
20       parent.joinAsEdgePeer(m.peer.getNode());
21       break;
22   }
23 }
```

Diese Methode bekommt alle eingehenden Nachrichten und bearbeitet alle Nachrichten vom Typ `ChordSPSA.Message`. Wenn ein Knoten eine `Downgrading`-Nachricht von seinem Super-Peer empfängt, sucht er sich einen neuen Super-Peer aus und verbindet zu diesem. Empfängt ein Edge-Peer eine `Upgrade`-Nachricht von seinem Super-Peer, wird er zum Super-Peer und antwortet mit `Upgraded`. Auf diese `Upgraded`-Nachricht reagiert der Super-Peer, indem er den Edge-Peers, die sich umhängen sollen, eine `ConnectTo`-Nachricht (entspricht der Reconnect-Nachricht im Chord-SPSA-Paper) sendet.

```
1  private void improveSuperPeer() {
2    Set<SCPeer> edgePeers = parent.getEdgePeers();
3    int dF = distinctFingers();
4    if ( edgePeers.size() > UPPER_BOUND * dF ) trySplit();
5    // There must be at least 3 other known super peers to allow downgrading.
6    // This should avoid the heavy impact when there are only few super peers.
7    else if ( parent.getSuperPeers().size() > 3 && edgePeers.size() <
         LOWER_BOUND * dF ) tryDowngrade();
8    else {
9      // No downgrade / split this time, resetting all flags
10     downgradeWait = (byte)parent.getRandom().nextInt(DOWNGRADE_WAIT_MAX);
11     splitPossible = false;
12     sentDowngrading = false;
13   }
14 }
```

Diese Methode wird periodisch aufgerufen, um einen Super-Peer zu verbessern. Zuerst wird die Split-Bedingung überprüft und bei Bedarf `trySplit()` aufgerufen. Danach wird die Downgrade-Bedingung überprüft und, falls nötig, `tryDowngrade()` aufgerufen.

```
1  private void improveEdgePeer() {
2    SCPeer sp = bestSuperPeer();
3    if ( sp != null && ! sp.equals(parent.getSuperPeer()) )
4      parent.joinAsEdgePeer(sp.getNode());
5  }
```

Diese Methode sucht den besten Super-Peer für einen Edge-Peer und verbindet zu diesem, um den Edge-Peer zu verbessern.

```
1  private void trySplit() {
2    // Wait one interval before splitting
3    if ( ! splitPossible ) {
4      splitPossible = true;
5      return;
6    }
7    SCPeer splitPeer = bestSplitPeer();
8    if ( splitPeer == null ) return;
9    parent.sendUnicastMessage(splitPeer.getAddr(), new Message( MessageType.
       Upgrade, null));
10 }
```

Diese Methode wird aufgerufen, wenn die Split-Bedingung erfüllt ist. Der erste Teil der Methode sorgt dafür, dass ein Zyklus abgewartet wird, bevor im zweiten Teil der beste Split-Peer gesucht wird und der Split mit einer `Upgrade`-Nachricht initiiert wird.

```
1  private void tryDowngrade() {
2    /* It is important here to avoid multiple peers downgrading at the same
         time
3     * so downgradeWait is initialized with a random value */
4    if ( downgradeWait > 0 ) {
5      downgradeWait--;
6      return;
7    }
8    // Wait an additional interval after sending downgrade message before
         downgrading.
9    if ( sentDowngrading ) {
10     SCPeer bestSuperPeer = bestSuperPeer();
11     if ( bestSuperPeer == null ) return;
12     parent.joinAsEdgePeer(bestSuperPeer.getNode());
13     //no need to disconnect edge peers, they will receive NonChord replies.
14   } else {
15     parent.sendLocalBroadcast(new Message(MessageType.Downgrading,null));
16     parent.getSuperPeerListService().onLeave();
17   }
18   sentDowngrading = ! sentDowngrading;
19 }
```

Chord-SPSA kann mit dem Parameter `superchord.use_chordspsa` aktiviert/deaktiviert werden. Mit dem Parameter `superchord.chordspsa.interval` kann die Periodenlänge eingestellt werden. Die Werte U für die obere Schranke und L für die untere Schranke können mit den Parametern `superchord.chordspsa.upper_bound` und `superchord.chordspsa.lower_bound` verändert werden.

4.3 Weitere Verfahren

4.3.1 Proximity Identifier Selection (PIS)

PIS ist ein Konzept, Identifier nach dem Gesichtspunkt der Entfernung zu wählen. Für Chord bedeutet dies, dass mit Hilfe der Super-Peer-Liste und deren Netzwerkkoordinaten vorhergesagt werden kann, welche Finger ein Knoten mit einer bestimmten Chord-ID haben wird. Es kann also vorher berechnet werden, welche Distanzen sich zu den späteren Fingern ergeben und so eine optimale Position gefunden werden.

Zur Berechnung der optimalen Chord-ID werden die Positionen genau zwischen zwei aufeinanderfolgenden Super-Peers auf ihre Kosten untersucht und die optimale Chord-ID ausgewählt.

Mit den so ermittelten Chord-IDs sollte ein besseres Chord-Netzwerk entstehen als mit dem bisherigen Verfahren mit Zufallszahlen.

```
1  long getBestChordId(SCPeer self) {
2    Set<SCPeer> speers = getSuperPeers();
3    speers.remove(self);
4    long[] ids = new long[speers.size()];
5    int i = 0;
6    for ( SCPeer peer : speers ) ids[i++] = peer.id;
7    Arrays.sort(ids); //ascending
8    long bestId = Math.abs(parent.getRandom().nextLong() % Chord.RING_SIZE);
9    double bestCosts = Double.POSITIVE_INFINITY;
10   for ( i = 0; i < ids.length; i++ ) {
11     long med = ( ids[i] + ( ( ids[(i+1)%ids.length] - ids[i] + Chord.
           RING_SIZE ) % Chord.RING_SIZE ) / 2 ) % Chord.RING_SIZE;
12     if ( Arrays.binarySearch(ids, med) >= 0 ) continue;
13     double costs = chordPositionCosts(self, med);
14     if ( costs < bestCosts ) {
15       bestId = med;
16       bestCosts = costs;
17     }
18   }
19   return bestId;
20 }
```

Diese Methode ermittelt in Zeilen 4 bis 7 alle Chord-IDs der Super-Peers und sortiert sie. Danach wird für jede Position in der Mitte zweier IDs die Kosten der Finger berechnet und die billigste Position zurückgegeben.

In Zeile 11 wird die Mitte zweier IDs berechnet, indem auf die erste ID die Hälfte der Differenz addiert wird.

```
1  double chordPositionCosts(SCPeer self, long chordId) {
2    Set<SCPeer> speers = getSuperPeers();
3    speers.remove(self);
4    double costs = 0.0;
5    for ( int i = 0; i < Chord.DIM; i++ ) {
6      SCPeer fi = self;
7      long fingerId = (chordId + (1 << i)) % Chord.RING_SIZE;
8      for ( SCPeer peer : speers )
9       if ( Chord.isBetween(fingerId, peer.id, fi.id)) fi = peer;
10     costs += self.distanceTo(fi);
11   }
12   return costs;
13 }
```

Diese Methode berechnet die Kosten einer Position im Chord-Ring. Aufgrund der Super-Peer-Liste werden alle Finger bestimmt und die Distanzen zu diesen addiert. Die Summe wird dann zurückgegeben.

In den Zeilen 8 und 9 wird der zukünftige Fingerknoten gesucht. Dazu wird die Knoten-Variable `fi` immer überschrieben, wenn der betrachtete Knoten `peer` zwischen der Finger-

Position und `fi` liegt.

PIS kann mit dem Parameter `superchord.use_pis` aktiviert bzw. deaktiviert werden.

4.3.2 Proximity Route Selection (PRS)

Bei reinem Chord wird beim Routing immer der Finger ausgewählt, der eine möglichst große Distanz gemessen in Chord-IDs zum Ziel überbrückt. PRS verändert das Routing dahingehend, dass nun auch die Latenz für einen Routing-Schritt berücksichtigt wird. Es wird nun der Finger zum Routen ausgewählt, der eine möglichst große Chord-ID-Distanz pro Millisekunde Latenz überbrückt.

Da die Chord-ID-Distanz trotz der geänderten Auswahl der Routing-Knoten in jedem Schritt sinkt, konvergiert das Routingverfahren weiterhin. Die Schranke der logarithmischen Komplexität kann aber nicht mehr garantiert werden.

Um die Latenz zu schätzen, werden die Vivaldi-Koordinaten verwendet.

```
1  ChordNode proximityClosestPrecedingFinger(long node) {
2    ChordNode bestNode = self;
3    double bestVal = 0.0;
4    ChordNode last = self;
5    for (ChordNode f: finger) {
6      if (f.id == last.id) break;
7      if (isBetween(self.id, f.id, node)) {
8        long chordDist = (f.id - self.id + Chord.RING_SIZE) % Chord.RING_SIZE;
9        double msgDist = self.distanceTo(f);
10       double val = chordDist / msgDist;
11       if ( val > bestVal ) {
12         bestVal = val;
13         bestNode = f;
14       }
15     }
16   }
17   return bestNode;
18 }
```

PRS kann mit dem Konfigurationsparameter `superchord.chord.use_prs` aktiviert bzw. deaktiviert werden.

5 Super-Peer-List-Service

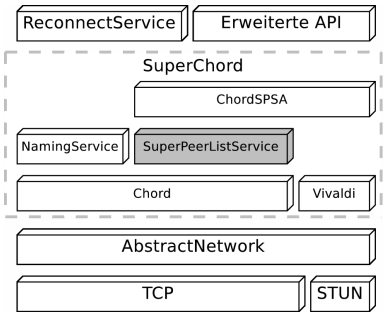
Der Super-Peer-List-Service hat das Ziel, eine komplette und aktuelle Liste aller Super-Peers inklusive aller Knoteninformationen in jedem Knoten zu führen. Dazu werden Broadcasts verwendet.

Zuerst wurde ein Gossip-Verfahren ähnlich zu DSDV[13] [20] verwendet. Dieses Verfahren stellte sich aber als unzuverlässig bei starken Fluktuationen heraus. Außerdem ist sowohl

[13] Destination Sequence Distance Vector

die Ausbreitungsgeschwindigkeit als auch das Datenaufkommen schlechter als bei dem hier verwendeten Verfahren.

Abbildung 19: Blockschaubild der Komponenten



5.1 Verfahren

Jeder Super-Peer verbreitet seine Knoteninformationen periodisch per Broadcast. Da der Broadcast ideal in Bezug auf Nachrichtenanzahl und Ausbreitungsgeschwindigkeit ist, sollte dieses Verfahren eine relativ geringe Netzwerklast verursachen. Die Periodendauer, das Announce-Interval, kann mit dem Konfigurationsparameter `superchord.sp_list.announce_interval` eingestellt werden. Wenn ein Super-Peer das Chord-Netzwerk verlässt, soll er sich per Broadcast abmelden.

Falls ein Knoten bemerkt, dass einer seiner Finger nicht mehr verfügbar ist und immer noch in der Super-Peer-Liste steht, so sendet er einen Broadcast und übernimmt das Abmelden für diesen Knoten. Die Beschränkung auf Finger wurde gewählt, weil dadurch mehrere Knoten den Crash eines Knotens melden können, aber ihre Anzahl trotzdem beschränkt ist. Eine Beschränkung auf den Vorgänger oder Nachfolger ist nicht sinnvoll, da sonst bei Crashs mehrerer aufeinander folgender Peers nicht alle abgemeldet werden.

Um dennoch falsche Einträge aus der Liste zu löschen, sind alle Einträge mit einem Timeout behaftet und werden, falls sie nicht erneuert werden, aus der Liste gelöscht. Der Timeout kann als Vielfaches des Announce-Intervalls mit dem Parameter `superchord.sp_list.ttl_intervals` eingestellt werden.

6 Reconnect-Service

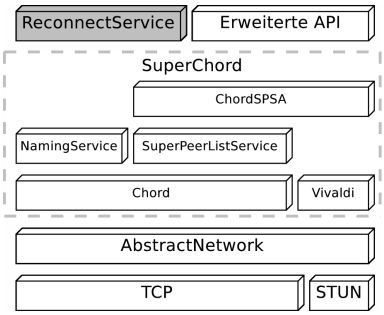
Um zu verhindern, dass ein Knoten dauerhaft die Verbindung zum Netzwerk verliert, wurde ein spezieller Dienst entwickelt. Der ReconnectService führt eine Liste bekannter Knoten, die aus den Daten der Super-Peer-Liste und den eigenen Edge-Peers befüllt wird. Alle Einträge in der Liste werden nach einem Timeout wieder gelöscht, wenn der entsprechende Knoten nicht zwischendurch erneut eingefügt wird. Damit die Liste nicht beliebig groß wird, kann die Maximalgröße der Liste mit dem Parameter `superchord.reconnect.known_limit` eingestellt werden.

Stellt der Dienst fest, dass der Knoten nicht mehr Teil des Netzwerks ist, stellt er nacheinander Verbindungen zu den gespeicherten bekannten Knoten her. Die Hauptmethode des Dienstes läuft periodisch und ist folgendermaßen aufgebaut:

```
1  public void run() {
2    // fill known peers with super peers and edge peers
3    knownPeers.addAll(net.getSuperPeers(), TIMEOUT);
4    knownPeers.addAll(net.getEdgePeers(), TIMEOUT);
5    // remove self
6    knownPeers.remove(net.getSelf());
7    // obey the limit
8    while ( knownPeers.size() > LIMIT ) knownPeers.remove(knownPeers.getFirst
         ());
9    if ( net.hasSuperPeer() || net.isInsideChordNetwork() ) {
10     // we are connected
11     lastTry = null;
12     return;
13   }
14   // remove the last try because we are still not connected
15   if ( lastTry != null ) knownPeers.remove(lastTry);
16   if ( knownPeers.isEmpty() ) return;
17   lastTry = knownPeers.getFirst();
18   net.join(lastTry.getNode());
19 }
```

Der Timeout, nachdem Knoten aus der Liste der bekannten Knoten gelöscht werden, kann mit dem Parameter `superchord.reconnect.known_timeout` konfiguriert werden. Das Intervall zwischen zwei Aufrufen der run-Methode kann über `superchord.reconnect.interval` eingestellt werden.

Abbildung 20: Blockschaubild der Komponenten

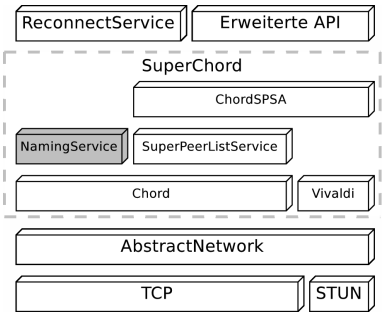


7 Permanente IDs

Jeder Knoten soll über eine eindeutige ID adressiert werden können. Da diese ID unabhängig von der Position des Knotens im Netzwerk ist, heißt sie permanente ID. Die

permanente ID eines Knotens kann mit dem Parameter `superchord.persid` konfiguriert werden.

Abbildung 21: Blockschaubild der Komponenten



7.1 Problembeschreibung

Durch die Verwendung von Super-Peers und Edge-Peers in Zusammenhang mit SPSA ergibt sich das Problem der Adressierung. Nachrichten an einen Edge-Peer müssen über dessen Super-Peer zugestellt werden. Dazu ist ein Verfahren nötig, um zu jedem Edge-Peer den dazugehörigen Super-Peer bestimmen zu können, ohne die Super-Peer-Wahl der Edge-Peers einzuschränken.

7.2 Verfahren

Das hier implementierte Verfahren verwendet Chord in seiner ursprünglichen Bestimmung als Hashtabelle um eine globale Zuordnung von PersistentId auf PeerAddress zu erreichen. Dazu wird von der PersistentId eines Knotens ein Hashwert gebildet. Der Knoten, der als erster auf diesen Hashwert im Ring folgt, ist für die Verknüpfung von PersistentId mit der PeerAddress verantwortlich.

Die PersistentIds ihrer Edge-Peers werden von Super-Peers eingetragen und periodisch erneuert. Wechselt ein Edge-Peer den Super-Peer, so aktualisiert der neue Super-Peer die PeerAddress des Edge-Peers.

Wenn ein Super-Peer das Chord-Netzwerk verlässt, sendet er alle Zuordnungen an den nächsten Chord-Knoten weiter um die Zuordnung zu erhalten. Beim Crash eines Super-Peers können Zuordnungen verloren gehen, die dann bei der nächsten Aktualisierung wieder hergestellt werden.

Der Abstand zwischen zwei Aktualisierungen und der verwendete Timeout können mit den Parametern `superchord.naming.rebind_interval` und `superchord.naming.cache_timeout` konfiguriert werden.

Das Verfahren ist mit folgenden Nachrichten implementiert:

Lookup Diese Nachricht enthält eine PersistentId und wird zu dem für diese zuständigen Super-Peer geleitet um die zugeordnete PeerAddress zu erfahren.

Result Diese Nachricht ist die Antwort auf eine Lookup-Nachricht. Sie enthält die gesuchte PersistentId und die gefundene PeerAddress. Falls keine PeerAddress gefunden wurde, bleibt das entsprechende Feld leer.

Bind Mit dieser Nachricht wird eine PeerAddress einer PersistentId zugeordnet. Diese Nachricht wird an den für die ID zuständigen Super-Peer geleitet und dort verarbeitet. Falls der PersistentId bereits eine andere PeerAddress zugeordnet ist, wird eine Konflikt-Nachricht versendet und der Wert nicht überschrieben.

Rebind Diese Nachricht bindet eine PeerAddress an eine PersistentId. Dabei werden keine Konflikte beachtet.

Conflict Diese Nachricht zeigt an, dass zu der angegebenen PersistentId bereits eine andere PeerAddress zugeordnet ist. Die andere PeerAddress wird in der Nachricht mitgesendet.

8 Library

8.1 Gesamtüberblick

Die Klassen wurden inhaltlich in Pakete verteilt:

`superchord` enthält die Hauptklasse des Netzwerks `SCNetworkImpl`, ihr Interface `SCNetwork` und zusätzlich wichtige Dienste.

`superchord.chord` enthält das Chord-Verfahren und dafür nötige Nachrichtenklassen und Interfaces.

`superchord.net` enthält alle Klassen des abstrakten Netzwerks.

`superchord.extapi` enthält die erweiterte API.

`superchord.util` enthält Hilfsklassen wie z.B. Cache-Implementierungen. Hier ist auch eine Klasse `NtpMessage` enthalten, die unter der GPL[14] lizenziert ist und vor der Weitergabe entfernt werden muss.

`superchord.log` enthält alle Klassen zum Schreiben und Verarbeiten von für die Tests notwendigen Logdaten. Dieses Paket ist nur für die Tests im PlanetLab notwendig.

`superchord.test` enthält alle Demo-Anwendungen.

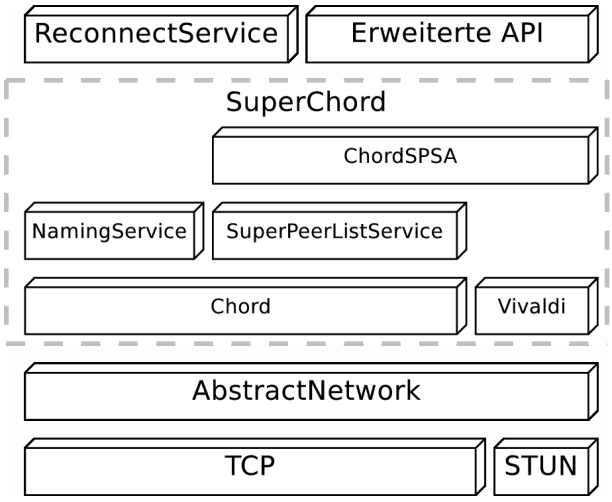
`superchord.service` enthält einige Dienste, die zum Debugging verwendet wurden und im Normalbetrieb nicht benötigt werden.

[14] General Public License

8.2 Klassenabhängigkeiten

Die Library ist in einzelne Komponenten unterteilt, die alle bereits vorgestellt wurden:

Abbildung 22: Blockschaubild der Komponenten



8.2.1 Abstraktes Netzwerk

Das abstrakte Netzwerk besteht aus der Hauptklasse `AbstractNetwork` und den Hilfsklassen `STUN`, `PublicAddress`, `Node` und `Message`. `AbstractNetwork` benutzt, die in der statischen Klasse `STUN` ausgelagerten Methoden, um Daten in Form der Datenklasse `PublicAddress` zu erhalten.

`Node` ist eine Datenklasse zur Speicherung der Kontaktinformationen eines Knotens und wird beim Senden und Empfangen von Daten und zum Unterscheiden von Verbindungen benutzt.

Die abstrakte Klasse `Message` wird als Basisklasse für interne Nachrichten benutzt, um eine Typsicherheit beim Versenden von Nachrichten sicherzustellen.

8.2.2 Chord

Chord besteht aus der Hauptklasse `Chord`, den Hilfsklassen `InvalidTargetPolicy`, `MessageType`, `ChordMessage`, `DirectMessage`, `RoutedMessage` und `BroadcastMessage`, und Interfaces `ChordConnector` und `ChordNode`.

Chord ist unabhängig vom Rest der Middleware und benötigt lediglich die Methoden des Callback-Interfaces `ChordConnector`, um mit seiner Umwelt zu interagieren. Die Klasse `ChordNode` ist ein Interface, das alle für Chord relevanten Eigenschaften eines Knotens zusammenfasst.

Die Basisklasse aller Chord-Nachrichten ist `ChordMessage`, die von `DirectMessage` erweitert wird. `DirectMessage` ist die Klasse aller an verbundene Knoten versendeten

Nachrichten. Sie enthält einen Nachrichtentyp aus dem Enum `MessageType`. Von `DirectMessage` erben die Klassen `RoutedMessage` und `BroadcastMessage`, die für geroutete bzw. gebroadcastete Nachrichten verwendet werden. Eine geroutete Nachricht enthält die Information, wie auf eine nicht existente Ziel-ID reagiert werden soll. Diese Information ist in der Klasse `InvalidTargetPolicy` realisiert.

Eingehende Nachrichten werden mit `receiveMessage()` empfangen und dann in `BroadcastMessage`, `RoutedMessage` und `DirectMessage` unterschieden und von den Methoden `forwardBroadcast()` und `forwardRoutedMessage()` weitergeleitet, bzw. von `processBroadcastMessage()`, `processRoutedMessage()` oder `processDirectMessage()` verarbeitet.

8.2.3 SuperPeerListService

Der SuperPeerListService besteht aus der Klasse `SuperPeerListService`. Der SuperPeerListService hat einen internen Task, der periodisch eine Nachricht der internen Klasse `Message` per Broadcast versendet, falls der Knoten ein Super-Peer ist.

Der SuperPeerListService implementiert das Listener-Interface von SuperChord. um Nachrichten zu empfangen und wird mit einer Instanz von `SCNetworkImpl` generiert, um Nachrichten zu versenden und zu erfahren, ob der Knoten ein Super-Peer ist.

Der SuperPeerListService stellt einige Methoden zur Verfügung, um die Super-Peer-Liste zu erfahren und manipulieren.

8.2.4 Vivaldi

Vivaldi besteht aus der Klasse `Vivaldi` und deren internen Klassen `Vivaldi.VivaldiMessage` und `Vivaldi.Coord`. Vivaldi hat einen internen Task, der periodisch eine Ping-Nachricht der Klasse `VivaldiMessage` versendet.

Vivaldi implementiert das Listener-Interface von SuperChord, um Nachrichten zu empfangen und wird mit einer Instanz von `SCNetworkImpl` generiert, um Nachrichten zu versenden.

Um einen zufälligen Peer für den Ping auszuwählen, wird `SCNetworkImpl.randomSuperPeer()` verwendet. Dadurch wird indirekt der übergeordnete `SuperPeerListService` verwendet.

Vivaldi stellt auf Anfrage – mit `getCoord()` – eine Referenz auf die internen Koordinaten zur Verfügung, sodass sich Koordinatenaktualisierungen automatisch auf alle Klassen auswirken, die diese Koordinaten verwenden.

8.2.5 NamingService

Der NamingService besteht aus der Klasse `NamingService`. NamingService hat einen internen Task, der periodisch eine Nachricht der internen Klasse `Message` im Chord-Ring versendet, um seine eigene PersistentId und die PersistentIds seiner Edge-Peers zu aktualisieren, falls der Knoten ein Super-Peer ist.

Der NamingService implementiert das Listener-Interface von SuperChord, um Nachrichten zu empfangen und wird mit einer Instanz von `SCNetworkImpl` generiert, um Nachrichten zu versenden und zu erfahren, ob der Knoten ein Super-Peer ist und welche Edge-Peers er hat. Außerdem wird auf die Events des Listener-Interfaces direkt reagiert.

Der NamingService stellt Methoden zur Verfügung, um eine PersistentId in eine Peer-Address aufzulösen und um die PersistentIds in der Empfängerliste einer Nachricht durch die Adressen zu ersetzen.

8.2.6 ChordSPSA

Chord-SPSA besteht aus der Klasse `ChordSPSA`. Chord-SPSA hat einen internen Task, der periodisch die Verbesserungen am Netzwerk vornimmt.

Chord-SPSA implementiert das Listener-Interface von SuperChord, um Nachrichten zu empfangen und wird mit einer Instanz von `SCNetworkImpl` generiert, um Nachrichten zu versenden und zu erfahren, ob der Knoten ein Super-Peer ist und welche Edge-Peers er hat. Außerdem verwendet Chord-SPSA die Methode `SCNetworkImpl.getSuperPeers()`, um auf die Super-Peer-Liste zuzugreifen.

Chord-SPSA stellt die Methode `getBestChordId()` zur Verfügung, die ideale Chord-ID per PIS zu berechnen.

8.2.7 SuperChord

Das SuperChord-Netzwerk besteht aus der Hauptklasse `SCNetworkImpl` und zahlreichen Hilfsklassen.

Die hauptsächliche Funktionalität der Klasse ist es, Nachrichten an das abstrakte Netzwerk zu senden und von diesem zu empfangen, und an die entsprechenden Klassen weiterzuleiten. Das Netzwerk legt bei der Generierung Objekte der Klassen `NamingService`, `Vivaldi`, `ChordSPSA` und `SuperPeerListService` an um sie zu verwenden. Super-Peers haben zusätzlich noch eine Objekt der Klasse `Chord`.

Um mit Chord zu kommunizieren, stellt `SCNetworkImpl` eine interne Klasse zur Verfügung, die das Callback-Interface `ChordConnector` implementiert. Die von SuperChord verwendete Knotenklasse `SCPeer` implementiert das Interface `ChordNode`.

Nach außen stellt das Netzwerk alle Methoden der verschiedenen Komponenten zur Verfügung, sodass ein konsistentes Interface erreicht wird. Das äußere Interface ist in `SCNetwork` realisiert.

8.2.8 ReconnectService

Der ReconnectService setzt als optionale Komponente auf dem Netzwerk auf. Die Klasse `ReconnectService` wird mit einer Instanz des Netzwerks als Parameter erstellt, und benutzt diese, um Informationen über verbundene Knoten zu erfahren und bei Bedarf eine Verbindung zu diesen herzustellen.

8.2.9 Erweiterte API

Die erweiterte API besteht aus der Hauptklasse `ExtAPIView` und den Hilfsklassen `EAMessage`, `QueueMessage`, `TopicMessage`, `ReliableManager`, `ReliableMessage`, `EAOutputStream`, `EAInputStream, DataMessage`, `ReliableMulticastMessage`, `ReliableMulticastSender` und `ReliableMulticastReceiver`. Die erweiterte API wird in Kapitel 8.5 näher beschrieben.

Die Klasse `ExtAPIView` implementiert das Listener-Interface von `SCNetworkImpl` und wird mit einer Instanz dieser Klasse erstellt, um mit dem Netzwerk zu kommunizieren.

Um Nachrichten an Queues und Topics zu versenden, werden die Datenklassen `QueueMessage` und `TopicMessage` verwendet, die die Nachricht intern als `EAMessage` beinhalten.

Wenn eine Nachricht zuverlässig versendet werden soll, wird eine Instanz der Klasse `ReliableManager` erstellt, die den Zustand der zuverlässigen Kommunikation beinhaltet. Zuverlässige Nachrichten werden in der Container-Klasse `ReliableMessage` gesendet.

Um große Objekte für den Versand in kleinere Byte-Blöcke aufzuspalten, wird die Klasse `EAOutputStream` verwendet. Die Byte-Blöcke werden dann mit der Container-Klasse `DataMessage` versendet und auf der Gegenstelle mit `EAInputStream` wieder zusammengesetzt. Das Aufspalten ist nötig, da das abstrakte Netzwerk Objekte immer am Stück überträgt, und dadurch Nachrichten, die für die Netzwerkwartung wichtig sind, beliebig lange verzögert werden könnten.

8.3 Code-Stil

Im Code wurden oftmals `equals`-Methoden als `deprecated` markiert und mit spezifischeren Parameter-Typen neu definiert. Dadurch soll eine Typsicherheit garantiert werden, um per IDE[15] feststellen zu können, ob ein falsches Objekt an `equals` übergeben wurde.

8.4 Interface für Anwendungen

Um die Schnittstelle für Anwendungen von der Implementierung getrennt zu halten, wurde ein Interface erstellt, das alle Schnittstelleninformationen beinhaltet. Das Interface benötigt ein Callback-Objekt des Typs `SCListener`, das ebenfalls hier aufgelistet ist.

[15] Integrated Development Environment

```
1  interface SCNetwork {
2    void addSCListener(SCListener listener);
3
4    SCPeer getSelf();
5    Set<SCPeer> getEdgePeers();
6    Set<SCPeer> getSuperPeers();
7    boolean hasSuperPeer();
8    boolean isInsideChordNetwork();
9    boolean isSuperPeer();
10   SCPeer getSuperPeer();
11
12   void changePersistentID(PersistentId persId);
13   void join(Node node);
14   void leave();
15
16   ExtAPIView extendedAPIView();
17
18   void sendLocalBroadcast(Serializable msg);
19   void sendGlobalBroadcast(Serializable msg);
20   void sendUnicastMessage(PeerAddress to, Serializable msg);
21   void sendUnicastMessageLookup(PersistentId persID, Serializable msg);
22   void sendLimitedBroadcast(long count, Serializable msg);
23   void sendMulticastMessage(AddressList knownDst, PersIdList lookupDst,
         Serializable msg);
24   void sendDHTMessage(long hash, Serializable msg);
25 }
```

Mit den Methoden `getSelf()` und `isSuperPeer()` können Informationen über den eigenen Knoten erhalten werden. Mit der Methode `hasSuperPeer()` kann festgestellt werden, ob der eigene Knoten ein Edge-Peer und mit einem Super-Peer verbunden ist. Ist dies der Fall, wird der Super-Peer von `getSuperPeer()` zurückgegeben. Bei einem Super-Peer lässt sich mit `isInsideChordNetwork()` feststellen, ob der Knoten mit einem Chord-Netzwerk verbunden ist. `getEdgePeers()` liefert die Menge aller Edge-Peers, die mit diesem Super-Peer verbunden sind. `getSuperPeers()` liefert die per `SuperPeerListService` ermittelte Menge aller Super-Peers.

Die Methode `join()` kontaktiert einen Knoten, um das Netzwerk zu betreten. Mit `leave()` wird das Netzwerk wieder verlassen. Die Methode `changePersistentId()` ändert die Kennung eines Knotens und verlässt dazu kurzzeitig das Netzwerk.

Die Methode `extendedAPIView()` liefert eine erweiterte API. Diese API wird im nächsten Abschnitt näher erklärt.

Mit `sendGlobalBroadcast()` kann eine Nachricht an alle Knoten gesendet werden. Die Methode `sendLocalBroadcast()` sendet eine Nachricht an alle Knoten mit der gleichen Chord-ID, d. h. den Super-Peer und alle seine Edge-Peers. Mit `sendMulticastMessage()` wird eine Nachricht an eine Reihe von Knoten mit bekannter Adresse und auch mit unbekannter Adresse (lediglich PersistentId bekannt) gesendet. Die Methoden `sendUnicastMessage()` und `sendUnicastMessageLookup()` können verwendet werden, um eine Multicast-Nachricht an ein einziges Ziel zu senden. Mit `sendDHTMessage()` kann eine

Nachricht an einen beliebigen Hash-Wert, der auf eine Chord-ID abgebildet wird, gesendet werden. Als Typ von Nachrichten wurde das Interface `Serializable` gewählt, da es die minimale Anforderung an Nachrichtenklassen darstellt und so die Benutzung des Netzwerks am wenigsten beschränkt wird.

```
1  interface SCListener {
2    void onReceiveMessage(PeerAddress from, MessageScope scope, Serializable
         msg);
3    void onLeave();
4    void onJoined();
5    void onJoinedSuperPeer(SCPeer peer);
6    void onSuperPeerLost(SCPeer peer);
7    void onChordPeerLost(SCPeer peer);
8    void onEdgePeerLost(SCPeer peer);
9    void onNewEdgePeer(SCPeer peer);
10   void onNewChordPeer(SCPeer peer);
11   void onDuplicateID();
12   void onUndeliverableMessage(MessageScope to, Serializable msg);
13   void onSelfChanged(SCPeer self);
14 }
```

Die Methoden des Callback-Objektes werden aufgerufen, wenn das Netzwerk die Anwendung über ein Event informieren will. Die mit Abstand wichtigste Methode ist `onReceiveMessage()`. Diese wird aufgerufen, wenn eine neue Nachricht für diesen Knoten eintrifft. Die Klasse `MessageScope` gibt hierbei an, auf welchem Weg die Nachricht den Knoten erreicht hat. Die Unterklasse `Global` von `MessageScope` weist z. B. auf einen Broadcast hin.

`onLeave()` und `onJoined()` informieren darüber, dass das Netzwerk verlassen wird bzw. betreten wurde. Die Methoden `onJoinedSuperPeer()` und `onSuperPeerLost()` werden aufgerufen, wenn der Knoten ein Edge-Peer ist und sich mit einem Super-Peer verbunden, bzw. diesen verloren hat. Der entsprechende Super-Peer wird darüber mit `onNewEdgePeer()` bzw. `onEdgePeerLost()` informiert. Wenn ein Super-Peer einen neuen Super-Peer sieht, wird `onNewChordPeer()` aufgerufen; verliert er diesen wieder, wird `onChordPeerLost()` aufgerufen.

Kann ein Knoten das Chord-Netzwerk nicht betreten, weil seine ID bereits belegt ist, wird `onDuplicateID()` aufgerufen. Mit `onUndeliverableMessage()` wird eine unzustellbare Nachricht an die Anwendung zurückgegeben.

`onSelfChanged()` wird aufgerufen, wenn sich die PersistentId oder ChordID eines Knotens ändert.

Eine Anwendung könnte diese API wie folgt benutzen:

```
1 SCNetwork net = new SCNetworkImpl();
2 net.addListener(this);
3 net.bind(port);
4 net.join(node);
5 //some actions
6 net.leave();
```

Weitere Beispiele für Anwendungen befinden sich im Paket `superchord.test`.

8.5 Erweiterte API

Um den Umstieg von JMS[16] bzw. ActiveMQ auf das hier entwickelte Netzwerk zu erleichtern und weitergehende Features zu ermöglichen, wurde eine spezielle Schnittstelle geschaffen, die den Paradigmen einer Message Queue folgt:

Queue: Eine Queue ist eine Art Briefkasten im Netzwerk. Nachrichten können an diese Queue gesendet und in FIFO[17]-Reihenfolge wieder abgerufen werden. Dabei erfolgt keine Zugangskontrolle; eine Queue kann von mehreren Knoten abgerufen werden.

Topic: Topics verhalten sich wie Funkfrequenzen. Nachrichten können von jedem Knoten an einen Topic gesendet werden und werden dann von allen Knoten empfangen, die sich für dieses Topic interessieren.

[16] Java Message Service
[17] First-In First-Out

```
1  public class ExtAPIView {
2    ExtAPIView(SCNetworkImpl net);
3    void startQueue();
4    void stopQueue();
5
6    void sendUnicast(PersistentId receiver, Serializable msg);
7    void sendMulticast(Set<PersistentId> receiver, Serializable msg);
8    void sendBroadcast(Serializable msg);
9    void sendLimitedBroadcast(int peers, Serializable msg);
10
11   void sendToTopic(String topic, Serializable msg);
12   void setSubscribed(String topic, boolean subscribed);
13   boolean isSubscribed(String topic);
14
15   void sendToQueue(String queue, Serializable msg);
16   void setReliable(PersistentId receiver, boolean reliable);
17   boolean isReliable(PersistentId receiver);
18   void fetchMessage(String queue);
19   void fetchAllMessages(String queue);
20   OutputStream getReliableMulticastOutputStream(TreeSet<PersistentId>
         targets);
21
22   EAMessage receive();
23   EAMessage receive(int timeout);
24   EAMessage receiveNoWait();
25   List<EAMessage> receiveAll();
26 }
```

```
1  class EAMessage {
2    PersistentId sender;
3    Serializable payload;
4  }
```

Da diese API eingehende Nachrichten speichert, bis sie abgerufen werden, muss die Nachrichtenannahme erst aktiviert werden, um ein Memory Leak zu vermeiden. Deshalb muss mit `startQueue()` die Nachrichtenannahme der internen Nachrichtenqueue eines Knotens gestartet und mit `stopQueue()` wieder deaktiviert werden.

Mit `sendToTopic()` kann eine Nachricht an ein Topic gesendet werden. Mit `setSubscribed()` und `isScubscribed()` kann ein Topic bestellt/abbestellt bzw. dieser Status ausgelesen werden. Dabei wird das Topic in die Liste der bestellten Topics aufgenommen bzw. aus dieser entfernt. Sendet ein Peer eine Nachricht an ein Topic, wird diese per Broadcast verteilt und bei den Empfängern dann anhand der Liste der bestellten Topics gefiltert.

Mit `sendToQueue()` kann eine Nachricht an eine Queue gesendet werden. Mit `fetchMessage()` und `fetchAllMessages()` kann eine bzw. alle Nachrichten aus eine Queue abgerufen werden.

Die zwei `receive()`-Methoden und `receiveNoWait()` erlauben es, eine empfangene Nachricht abzurufen, wahlweise mit begrenztem, unbegrenztem oder ohne Timeout. Mit

`receiveAll()` können alle Nachrichten auf einmal abgerufen werden.

Mit `setReliable()` kann eine Ende-zu-Ende-Verbindung mit einem Verfahren ähnlich zu TCP geschützt werden. Mit `isReliable()` kann dieser Status ausgelesen werden. Das dabei verwendete Verfahren teilt jeder Nachricht eine Nummer zu. Mit der Nachricht wird deren Nummer und die Nummer der Vorgängernachricht übertragen. Empfängt ein Knoten eine solche Nachricht, überprüft er die Nummer der Vorgängernachricht. Stimmt sie mit der gespeicherten Nummer überein, wird die Nachricht zugestellt und eine Bestätigung versendet. Empfängt ein Sender eine solche Bestätigung, verschiebt er das Sendefenster.

Mit `getReliableMulticastOutputStream()` kann ein OutputStream geöffnet werden, dessen Daten zuverlässig an die angegebenen Empfänger zustellt.

8.6 Konfigurationsparameter

Die Beschreibung der Parameter hat folgendes Format:

`Parametername`	`Format`	`Standardwert`
Beschreibung		

Folgende Parameter werden vom Netzwerk selbst unterstützt. Alle Parameter können per System-Property beim Aufruf von Java gesetzt werden, also per (`-D property=value`). Bei der Angabe der Parameter ist auf die Einhaltung des Formats (InetAddress[18], Boolean[19], Integer[20], Byte[21], Double[22]) zu achten.

`superchord.persid String zufällig`

Mit diesem Parameter kann die PersistentId des Knotens festgelegt werden. Wenn der Wert auf "*" gesetzt ist, wird der Standardwert verwendet.

`superchord.public_address InetAddress per STUN ermittelt`

Mit diesem Parameter kann die öffentliche Adresse festgelegt werden, unter der der Knoten erreichbar ist. Wenn der Wert auf "*" gesetzt ist, wird der Standardwert verwendet.

`superchord.reachable Boolean per STUN ermittelt`

Dieser Parameter gibt an, ob der Knoten von anderen Knoten unter seiner Adresse tatsächlich erreichbar ist und nicht durch ein NAT oder eine Firewall behindert wird. Wenn der Wert auf "*" gesetzt ist, wird der Standardwert verwendet.

[18] Das von `InetAddress.getByName()` verstandene Format
[19] Das von `Boolean.parseBoolean()` verstandene Format
[20] Das von `Integer.parseInteger()` verstandene Format
[21] Das von `Byte.parseByte()` verstandene Format
[22] Das von `Double.parseDouble()` verstandene Format

`superchord.stun.server Host:Port 'stunserver.org:3478'`

Mit diesem Parameter kann man den zu verwendenden STUN-Server festlegen. Im Bereich Voice over IP gibt es zahlreiche öffentliche STUN-Server.

`superchord.use_chordspsa Boolean false`

Mit diesem Parameter kann angegeben werden, ob Chord-SPSA verwendet werden soll. Wird Chord-SPSA nicht verwendet, reagiert der Knoten weiterhin normal auf Chord-SPSA-Anfragen von anderen Knoten, führt aber selbst keine Reorganisationszyklen mehr aus.
Wird dieser Chord-SPSA in allen Knoten des Netzwerks deaktiviert, findet keine Topologieoptimierung mehr statt.

`superchord.use_pis Boolean true`

Mit diesem Parameter kann PIS (Proximity Identifier Selection) aktiviert oder deaktiviert werden.

`superchord.reconnect.known_timeout Integer 900000`

Dieser Parameter gibt an, wie lange bekannte Knoten maximal gespeichert werden, ohne dass Kontakt zu ihnen besteht. Die Angabe erfolgt in Millisekunden.

`superchord.reconnect.known_limit Integer 100`

Dieser Parameter gibt die maximale Anzahl gespeicherter bekannter Knoten an.

`superchord.reconnect.interval Integer 10000`

Dieser Parameter gibt an, wie oft auf Verbindungsverlust überprüft werden soll und wie oft ein Verbindungsversuch gestartet wird. Die Angabe erfolgt in Millisekunden.

`superchord.stun.timeout Integer 1000`

Dieser Parameter gibt den Timeout für die STUN-Abfrage an. Der Wert sollte nicht zu groß gewählt werden, da der Timeout im Normalbetrieb auftreten kann. Die Angabe erfolgt in Millisekunden.

`superchord.net.sotimeout Integer 30000`

Dieser Parameter gibt den Timeout an, der beim Lesen auf einem Socket angewendet wird. Die Angabe erfolgt in Millisekunden.

`superchord.net.idle_close Integer 100`

Dieser Parameter gibt an, nach wie vielen sotimeout-Intervallen eine Verbindung im Leerlauf getrennt wird.

`superchord.chord.base_stabilize_interval Integer 10000`

Dieser Parameter gibt den Startwert des Stabilisierungsintervalls für Chord an. Die Angabe erfolgt in Millisekunden.

`superchord.chord.min_stabilize_interval Integer 1000`

Dieser Parameter gibt den Minimalwert des Stabilisierungsintervalls für Chord an. Die Angabe erfolgt in Millisekunden.

`superchord.chord.max_stabilize_interval Integer 300000`

Dieser Parameter gibt den Maximalwert des Stabilisierungsintervalls für Chord an. Die Angabe erfolgt in Millisekunden.

`superchord.chord.stabilize_low_changes Integer 0`

Dieser Parameter gibt den Maximalwert für Veränderungen an den Fingern und dem Predecessor seit der letzten Stabilisierung an. Wird dieser Wert nicht überschritten, wird das Intervall für den Stabilisierungszyklus verlängert.

`superchord.chord.stabilize_high_changes Integer 3`

Dieser Parameter gibt den Minimalwert für Veränderungen an den Fingern und dem Predecessor seit der letzten Stabilisierung an. Wird dieser Wert nicht unterschritten, wird das Intervall für den Stabilisierungszyklus verkürzt.

`superchord.chord.stabilize_low_changes_adjust Double 1.2`

Dieser Parameter gibt an, wie das Intervall für den Stabilisierungszyklus verändert werden soll, wenn die Schranke für eine Verlängerung erreicht wurde.

`superchord.chord.stabilize_high_changes_adjust Double 0.5`

Dieser Parameter gibt an, wie das Intervall für den Stabilisierungszyklus verändert werden soll, wenn die Schranke für eine Verkürzung erreicht wurde.

`superchord.chord.use_prs Boolean true`

Dieser Parameter gibt an, ob PRS (Proximity Route Selection) verwendet werden soll.

`superchord.chord.dim Byte 20`

Dieser Parameter gibt an, wie groß der Nummernbereich für Chord-IDs ist. Die Angabe erfolgt als Exponent einer Zweierpotenz. Dieser Wert bestimmt auch die maximale Anzahl an Fingern eines Knotens. Der zulässige Wertebereich ist 1 bis 63.
Achtung: Es ist für das Funktionieren des Netzwerks wichtig, dass alle Knoten im Netzwerk den gleichen Wert für diese Einstellung verwenden.

`superchord.chordspsa.interval Integer 5000`

Dieser Parameter gibt an, wie oft der Reorganisationszyklus von Chord-SPSA ausgeführt werden soll. Die Angabe erfolgt in Millisekunden.

`superchord.chordspsa.upper_bound Integer 8`

Dieser Parameter gibt das Verhältnis von Edge-Peers zu verschiedenen Fingerknoten eines Super-Peers an, das überschritten werden muss, damit der Knoten ein Upgrade ausführen kann.

`superchord.chordspsa.lower_bound Integer 2`

Dieser Parameter gibt das Verhältnis von Edge-Peers zu verschiedenen Fingerknoten eines Super-Peers an, das unterschritten werden muss, damit der Knoten ein Downgrade ausführen kann.

`superchord.chordspsa.downgrade_wait_max Byte 10`

Dieser Parameter gibt an, wie viele Reorganisationszyklen in Folge die Downgradebedingung maximal erfüllt sein muss, bevor tatsächlich ein Downgrade stattfinden kann. Der konkrete Wert wird zwischen 0 und diesem Wert zufällig gewählt.

`superchord.sp_list.announce_interval Integer 60000`

Dieser Parameter gibt an, wie häufig ein Super-Peer seine Existenz im Netzwerk verbreitet. Die Angabe erfolgt in Millisekunden.

`superchord.sp_list.ttl_intervals Integer 2`

Dieser Parameter gibt an, nach wie vielen Announce-Intervallen alte Einträge aus der Super-Peer-Liste entfernt werden.

`superchord.naming.rebind_interval Integer 180000`

Dieser Parameter gibt an, wie häufig Naming-Einträge aktualisiert werden. Der entsprechende Timeout beträgt das Doppelte dieser Einstellung. Die Angabe erfolgt in Millisekunden.

`superchord.naming.cache_timeout Integer 60000`

Dieser Parameter gibt an, wie lange Resultate des Namingservice vorgehalten werden. Die Angabe erfolgt in Millisekunden.

`superchord.extapi.timeout Integer 10000`

Dieser Parameter gibt an, welcher Timeout bei zuverlässigen Verbindungen verwendet wird. Die Angabe erfolgt in Millisekunden.

`superchord.vivaldi.dim Integer 4`

Dieser Parameter gibt an, wie viele Dimensionen die Vivaldi-Koordinaten haben.
Achtung: Es ist für das Funktionieren des Netzwerks essenziell wichtig, dass alle Knoten im Netzwerk den gleichen Wert für diese Einstellung verwenden.

`superchord.vivaldi.interval Integer 10000`

Dieser Parameter gibt an, wie häufig die Vivaldi-Koordinaten durch Messung aktualisiert werden sollen. Die Angabe erfolgt in Millisekunden.

`superchord.vivaldi.averaging_error_speed Double 0.1`

Dieser Parameter gibt an, wie stark ein neuer Fehlerwert in den gemittelten Fehlerwert eingerechnet wird. Der zulässige Wertebereich ist $(0.0 .. 1.0]$.

`superchord.vivaldi.max_force Double 0.25`

Dieser Parameter gibt an, wie stark eine neue Messung die Koordinaten maximal verändern kann. Der Wert ist ein Faktor, der auf die Kraft angewendet wird. Der zulässige Wertebereich ist $(0.0 \,..\, 1.0]$.

Folgende Parameter werden in den Demo-Anwendungen ausgewertet:

`superchord.join Host:Port`

Dieser Parameter gibt den Bootstrap-Peer an. Dieser Knoten wird kontaktiert, um eine Verbindung zum Netzwerk herzustellen. Fehlt diese Angabe, so stellt der Knoten keine Verbindung zum Netzwerk her. Der Peer steht allerdings weiterhin für Verbindungen zur Verfügung und kann so als erster Knoten ein neues Netzwerk erstellen.

`superchord.port Integer zufälliger freier Port`

Dieser Parameter gibt den Port an, auf dem der ServerThread auf Verbindungen warten soll. Ist dieser Parameter sie auf "*" gesetzt, wird der Standard verwendet.

`superchord.logfile String`

Mit diesem Parameter kann das für die Tests verwendete Logging aktiviert werden. Der Parameter gibt die Datei an, in die die Logdaten geschrieben werden sollen.

In den Demo-Anwendungen können all diese Parameter auch in einer Datei spezifiziert werden. Wenn die Datei "superchord.conf" im aktuellen Verzeichnis existiert, wird sie ausgelesen und die enthaltenen Parameter verwendet.

8.7 Netzwerkformat

Für das Netzwerk wurden häufig verwendete Klassen mit Externalizable serialisiert. Dadurch werden diese Klassen viel effizienter übertragen. Allerdings muss bei Änderungen dieser Klassen eventuell auch die Serialisierung geändert werden. Aus diesem Grund haben fast alle serialisierten Klassen eine Versionsnummer.

`superchord.AddressList`	list.size()	int	
	each list entry	PeerAddress	direkt
`superchord.ApplicationMessage`	src	PeerAddress	direkt
	scope	MessageScope	direkt
	payload	Object	über Java-Serialisierung
`superchord.ChordSPSA.Message`	type	byte	als Ordinal
	peer	SCPeer	direkt

`superchord.EdgeMessage`	type	byte	als Ordinal
	src	PersistentId	direkt
	payload	Object	über Java-Serialisierung
`superchord.MessageScope`	Alle Scopes werden zentral serialisiert und mit einem Typ-Feld unterschieden.		
`superchord.MessageScope .InvalidTarget`	type	byte	=1
	dst	PeerAddress	direkt
`superchord.MessageScope .DHTHash`	type	byte	=2
	hash	long	
`superchord.MessageScope .SPLocal`	type	byte	=4
	dstChordId	long	
`superchord.MessageScope .SPGlobal`	type	byte	=5
`superchord.MessageScope .Global`	type	byte	=6
`superchord.MessageScope .LimitedGlobal`	type	byte	=7
	peerCount	long	
`superchord.MessageScope .Multicast`	type	byte	=8
	dstKnown	AddressList	direkt
	dstLookup	PersIdList	direkt
`superchord.NamingService`	type	byte	als Ordinal
	key	PersistentId	direkt
	address	PeerAddress	direkt
`superchord.PeerAddress`	version	byte	0 bedeutet null
	chordId	long	nur bei version≠0
	persId	PersistentId	direkt, nur bei version≠0
`superchord.PersIdList`	list.size()	int	
	each list entry	PersistentId	direkt

`superchord.PersistentId`	version	byte	0 bedeutet null
	internal.length	int	nur bei version≠0
	internal	byte[]	nur bei version≠0
`superchord.SCPeer`	version	byte	0 bedeutet null
	node	Node	direkt, nur bei version≠0
	reachable	boolean	nur bei version≠0
	addr	PeerAddress	direkt, nur bei version≠0
	coords	Vivaldi.Coord	direkt, nur bei version≠0
`superchord.SuperPeerList`	list.size()	int	
	each list entry	SCPeer	direkt
`superchord.SuperPeerListService.Message`	type	byte	als Ordinal
	peer	SCPeer	direkt
`superchord.Vivaldi.Coord`	version	byte	0 bedeutet null
	coords.length	byte	nur bei version≠0
	each coords entry	double	nur bei version≠0
`superchord.Vivaldi.VivaldiMessage`	stage	byte	als Ordinal
	millis	long	
	coordError	double	
	coord	Coord	direkt
`superchord.chord.ChordMessage`	src	ChordNode	über Java-Serialisierung
`superchord.chord.DirectMessage`	src	ChordNode	über Java-Serialisierung
	type	byte	als Ordinal
	payload	Object	über Java-Serialisierung
`superchord.chord.BroadcastMessage`	src	ChordNode	über Java-Serialisierung
	type	byte	als Ordinal
	payload	Object	über Java-Serialisierung
	fromId	PeerAddress	direkt
	startId	long	
	endId	long	

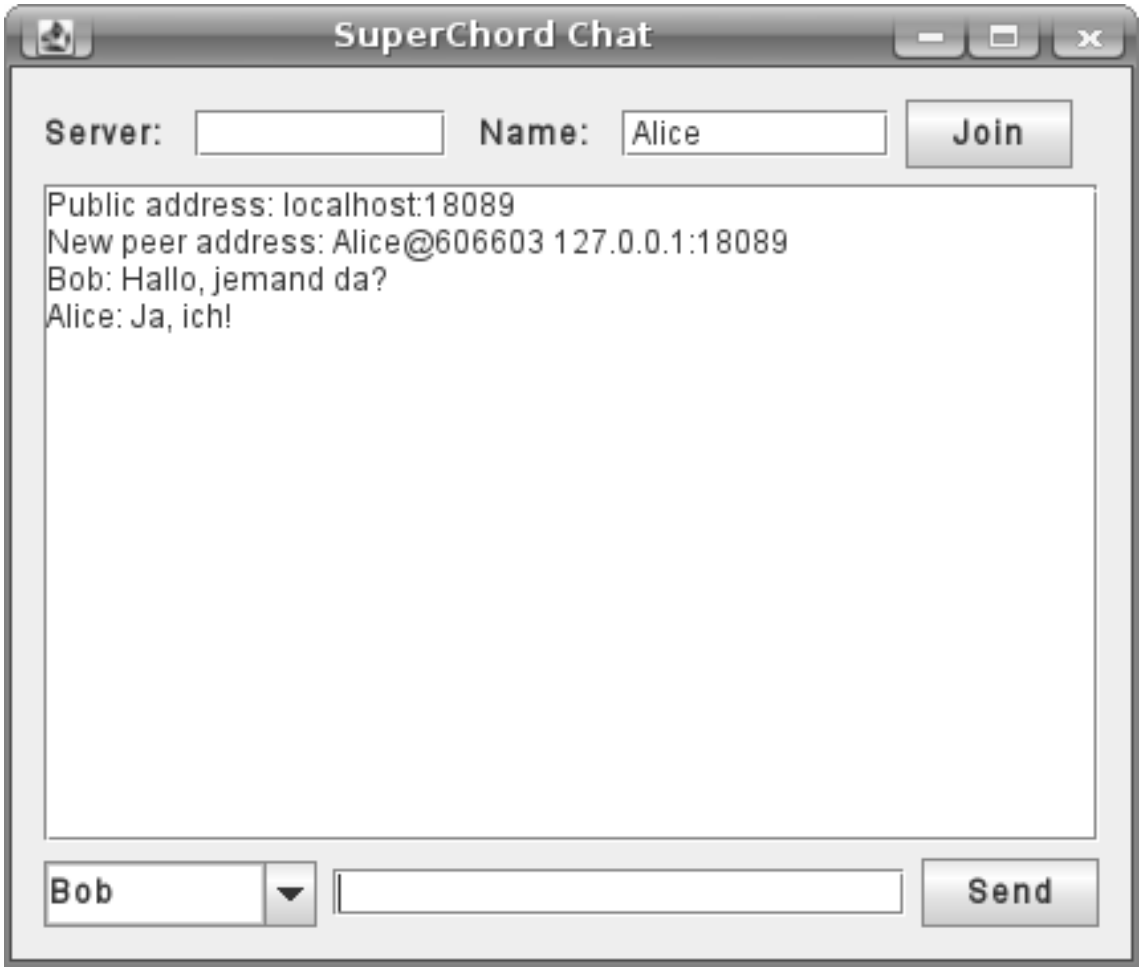
`superchord.chord.RoutedMessage`	src	ChordNode	über Java-Serialisierung
	type	byte	als Ordinal
	payload	Object	über Java-Serialisierung
	matchPolicy	byte	als Ordinal
	fromId	PeerAddress	direkt
	toId	AddressList	direkt
`superchord.extapi.QueueMessage`	command	byte	als Ordinal
	queue	UTFString	
	message	EAMessage	direkt
`superchord.extapi.TopicMessage`	topic	UTFString	
	message	EAMessage	direkt
`superchord.extapi.EAMessage`	version	byte	0 bedeutet null
	sender	PersistentId	direkt, nur bei version≠0
	payload	Object	über Java-Serialisierung, nur bei version≠0
`superchord.net.Node`	version	byte	0 bedeutet null
	addr.length	byte	nur bei version≠0
	addr	byte[]	nur bei version≠0
	port	int	nur bei version≠0

Da hier immer wieder die Java-Serialisierung verwendet wird, ist das Format für andere Sprachen nur schwer lesbar. Außerdem erzeugt die Java-Serialisierung einen großen Overhead. Die Java-Serialisierung kann nur umgangen werden, wenn ein einheitliches Nachrichtenformat verwendet wird. Das erfordert aber die Zusammenfassung aller Programmschichten.

8.8 Demo-Anwendung: Chat

Um für dieses Netzwerk auch eine Anwendung präsentieren zu können, wurde eine kleine Chat-Applikation zur Demonstration entwickelt. Die Anwendung ist sehr einfach gehalten und soll nicht mit bestehenden Chat-Anwendungen konkurrieren können. Die Chat-Anwendung nutzt PersistentIds als Chat-Namen und Unicast und Broadcast für den Chat.

Abbildung 23: Chat-Applikation



Im Feld Server kann der Bootstrap-Peer im Format Host:Port eingegeben und mit dem Knopf "Join" eine Verbindung hergestellt werden. Im Feld Name kann ein Chat-Name eingegeben werden, der mindestens 3 Zeichen enthalten muss; mit dem Knopf "Join" kann der Name geändert werden.

In die untere Textzeile kann eine Nachricht eingegeben und aus dem Drop-Down-Menü daneben ein Empfänger auswählt werden. "ALL" sendet einen Broadcast. Mit "Send" kann die Nachricht dann an den ausgewählten Empfänger gesendet werden.

Empfangene Nachrichten und andere Ereignisse werden im Textbereich in der Mitte angezeigt.

9 Szenarien

In diesen Szenarien wird die Nachrichtenübermittlung auf der Ebene des abstrakten Netzwerk nicht betrachtet, da diese bereits in Kapitel 2 beschrieben wurde.

9.1 Nachrichtenübermittlung

Um eine Nachricht zu übermitteln, erfolgen einige komplexe Schritte. Hier soll die Übermittlung einer Unicast-Nachricht von einem Super-Peer an eine PersistentId eines entfernten Edge-Peers detailiert untersucht werden.

Die Nachricht wird dem Netzwerk bei `SCNetworkImpl.sendUnicastMessageLookup()` übergeben. Dann wird direkt `SCNetworkImpl.sendMessage()` mit einem entsprechenden `MessageScope` aufgrufen. Nun wird ein `ApplicationMessage`-Objekt erstellt und an `SCNetworkImpl.forwardApplicationMessage()` übergeben, da der Absender ein Super-Peer ist.

Nachdem überprüft wurde, dass der Zielknoten nicht der eigene Knoten und auch kein Edge-Peer ist, wird versucht, die `PersistentId` in eine `PeerAddress` umzuwandeln. Dazu wird `NamingService.lookupOnly()` aufgerufen. Diese Funktion sucht im Cache nach der `PersistentId`. Für dieses Szenario wird angenommen, dass die ID nicht gefunden wurde. Dann wird `NamingService.sendMessage()` aufgerufen. Dort wird die Nachricht in `NamingService.messageQueue` zwischengespeichert und `NamingService.lookup()` aufgerufen, wo eine Nachricht im Chord-Ring versendet wird, um die `PeerAddress` zu der `PersistentId` ausfindig zu machen.

Trifft das Ergebnis ein, so wird `NamingService.onReceiveMessage()` aufgerufen. Dort wird die gespeicherte Nachricht aus der `messageQueue` geladen und das Ziel durch die PeerAddress ersetzt. Nun wird wieder `SCNetworkImpl.forwardApplicationMessage()` aufgerufen. Dort wird, nachdem festgestellt wurde, dass die Chord-ID verschieden von der eigenen ist, `Chord.routeMessage()` aufgerufen. In `Chord.forwardRoutedMessage()` wird nun ein Knoten zum Routen der Nachricht gesucht und die Nachricht an diesen weitergesendet.

Der Super-Peer des Zielknotens erhält die Nachricht dann zuerst in `Chord.receiveMessage()`, wo erkannt wird, dass die Chord-ID die gleiche des Zielknotens ist und `Chord.processRoutedMessage()` aufgerufen wird. Dort wird dann `ChordConnector.onApplicationMessage()` aufgerufen, was `SCNetworkImpl.ChordCallback.onApplicationMessage()` entspricht. Dann wird zuerst `SCNetworkImpl.handleChordMessage()`, `SCNetworkImpl.handleApplicationMessage()` und danach `SCNetworkImpl.forwardApplicationMessage()` aufgerufen. Nun wird festgestellt, dass der Zielknoten ein Edge-Peer ist, und `SCNetworkImpl.sendEdgeMessage()` aufgerufen. Dort wird eine Nachricht an den Zielknoten gesendet.

Der Zielknoten empfängt die Nachricht seines Super-Peers bei `SCNetworkImpl.handleIncomingMessage()` und gibt sie an `SCNetworkImpl.handleApplicationMessage()` weiter, wo erkannt wird, dass die Nachricht für diesen Knoten bestimmt ist, und `SCNetworkImpl.receivedApplicationMessage()` aufgerufen wird.

9.2 Verbindungsaufbau

Für dieses Szenario wird davon ausgegangen, dass ein neuer Knoten das Netzwerk als Edge-Peer betreten will. Wenn Chord-SPSA verwendet wird, treten alle neuen Knoten dem Netzwerk als Edge-Peers bei. In diesem Szenario kontaktiert der neue Knoten zuerst

einen Knoten, der selbst Edge-Peer ist.

Wenn die Methode `SCNetworkImpl.joinAsEdgePeer()` von der Anwendung aufgerufen wird, um das Netzwerk als Edge-Peer zu betreten, werden zuerst alle Verbindungen zu anderen Knoten geschlossen. Dann wird eine `EdgeMessage`-Nachricht vom Typ `EdgeMessageType.Join` in der Methode `SCNetworkImpl.sendEdgeMessage()` an den Kontaktknoten gesendet.

Im Kontaktknoten wird die Nachricht in `SCNetworkImpl.handleIncomingMessage()` empfangen und dort verarbeitet. Da der Kontaktknoten als Edge-Peer keine Knoten aufnehmen kann, antwortet er mit einer `EdgeMessage`-Nachricht vom Typ `EdgeMessageType.NonChord`, die die Super-Peer-Liste des Kontaktknotens enthält.

Diese Nachricht wird in `SCNetworkImpl.handleIncomingMessage()` behandelt, woraufhin der Knoten eine neue Verbindung zu einem Super-Peer aufbaut und erneut eine Join-Nachricht sendet. Wenn der Super-Peer diese Nachricht empfängt, nimmt er den Knoten als Edge-Peer auf und sendet eine `EdgeMessage` mit dem Typ `EdgeMessageType.JoinReply`, um dem Edge-Peer seine neue Identität (neue Chord-ID) mitzuteilen.

Auf diese JoinReply-Nachricht reagiert der Knoten, indem er den Absender als Super-Peer speichert und die neue Chord-ID übernimmt.

9.3 Knotenausfall

Fällt ein Knoten aus, werden alle seine Verbindungen abgebrochen und können nicht wieder hergestellt werden. Ein Knoten wird über einen Verbindungsabbruch vom abstrakten Netzwerk mit der Methode `SCNetworkImpl.AbstractNetworkCallback.onConnectionBroken()` informiert und behandelt diesen dann in der Methode `SCNetworkImpl.handleConnectionLost()`.

In dieser Methode wird zwischen Edge-Peers und Super-Peers unterschieden. Ist der Knoten ein Edge-Peer und der ausgefallene Knoten sein Super-Peer, sucht der Knoten sich einen neuen Super-Peer aus der Super-Peer-Liste und verbindet zu diesem.

Ist der Knoten ein Super-Peer und der ausgefallene Knoten ein Finger oder der Vorgängerknoten, wird Chord mit der Methode `Chord.connectionLost()` über den Ausfall informiert. Fällt der Knoten dadurch aus dem Chord-Ring heraus, stellt er eine neue Verbindung zu einem Knoten aus der Super-Peer-Liste her.

10 Validierung im PlanetLab

10.1 Aufbau

Da für das Experiment viele Knoten benötigt wurden, wurde dafür PlanetLab[6; 9; 21] genutzt. PlanetLab ist ein Netzwerk von mehreren hundert Knoten von gemischter Qualität. Um möglichst zuverlässige Knoten auszuwählen, wurde CoMon[19] verwendet. CoMon überwacht und analysiert die Leistung der PlanetLab-Knoten und ermöglicht es, Knoten anhand von Kriterien auszuwählen.

Für die Auswahl der Knoten für das Experiment wurden folgende Kriterien verwendet:

- Die Knoten müssen verfügbar sein und eine Antwortzeit von höchstens 5 Sekunden haben.
- Die maximale 1-Minuten-Load[23] der letzten 15 Minuten liegt unter 10.
- Die Knoten sind mit maximal 10 MBps Upstream bzw. Downstream belastet.
- Pro Standort wird nur ein Knoten verwendet.

Nach diesen Kriterien wurden 169 Knoten ausgewählt, von denen nach einer Funktionsprüfung (funktionierendes Java) 62 Knoten für die Tests verwendet werden konnten. Eine Auflistung der genutzten Knoten befindet sich im Anhang.

Die Anwendung ist so programmiert, dass sie zu einem bestimmten Zeitpunkt startet und endet. So kann die Anwendung vorab verteilt und kontrolliert gestartet werden. Um die Uhren der Rechner zu synchronisieren, wurde eine NTP[24]-Implementierung für Java verwendet, da die Uhren der PlanetLab-Knoten nicht verändert werden dürfen. Diese Implementierung steht unter der GPL-Lizenz zur Verfügung und muss vor der Weitergabe der Bibliothek wieder entfernt werden. Zusätzlich kann das gesamte Paket `superchord.log` entfernt werden, da es nur für die Tests verwendet wird.

Über speziell für diese Zwecke entwickelte Skripte wird die Testanwendung verteilt, gestartet und gestoppt und die Ergebnisse gesammelt und ausgewertet.

Im Versuchsaufbau wird ein Knoten als Bootstrap-Peer ausgewählt und von allen anderen Knoten kontaktiert. Damit der Bootstrap-Peer auf jeden Fall bereits aktiv ist, wenn andere Knoten starten, wird er eine Minute früher gestartet. Die anderen Knoten starten zufällig bis zu einer Minute nach der Startzeit, um den Bootstrap-Peer nicht zu überlasten.

10.2 Messungen

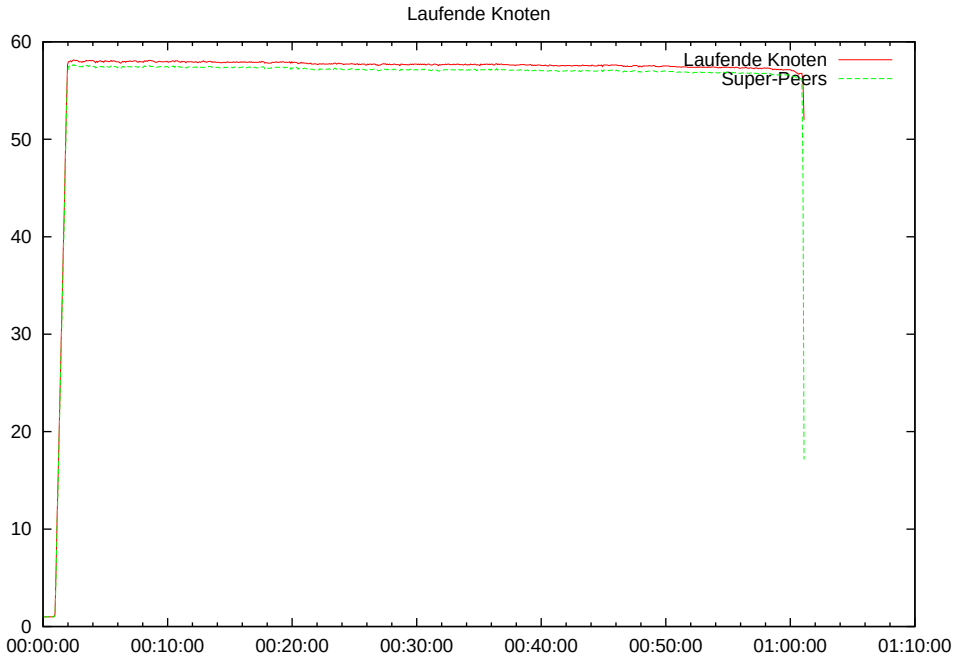
10.2.1 Erste Serie

Die erste Messungsserie besteht aus 22 Einzelmessungen mit je einer Stunde Laufzeit. Der Speicherverbrauch wurde auf 150 MiB beschränkt. Weiterhin wurden folgende Parameter verwendet:

- Chord-SPSA wurde deaktiviert, d. h. alle Peers werden zu Super-Peers falls sie erreichbar sind.
- PIS wurde deaktiviert.
- PRS ist aktiviert.
- Der Socket-Timeout wurde auf 60000 – also 1 Minute – gesetzt.

[23] Last, definiert als die durchschnittliche Anzahl der als aktiv markierten Prozesse.
[24] Network Time Protocol

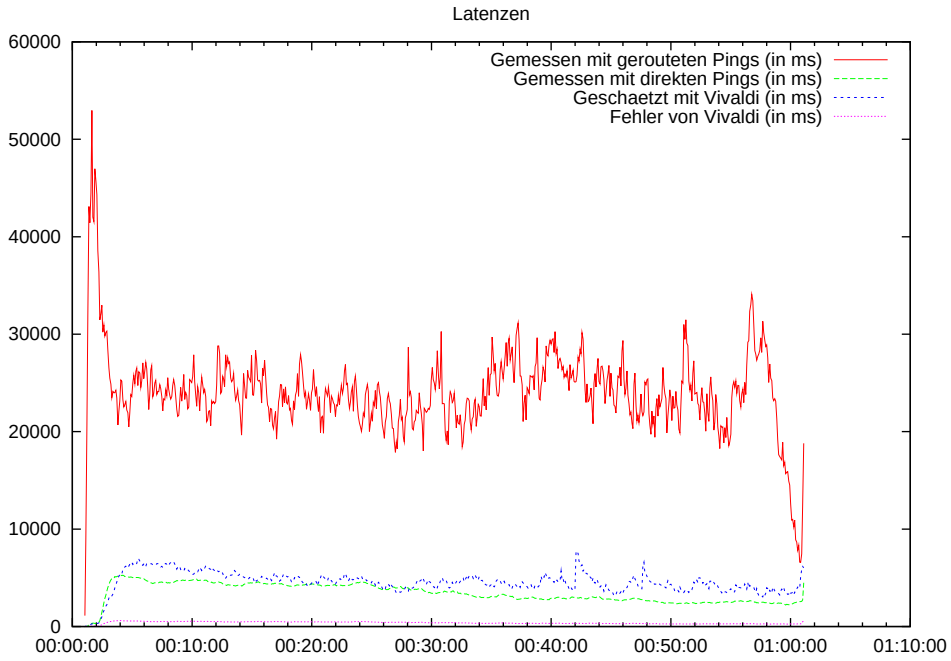
Abbildung 24:



Hier ist sehr gut zu erkennen, dass alle Knoten außer dem Bootstrap-Peer, mit bis zu 2 Minuten Verzögerung starten um diesen zu entlasten.

Peers, die das Netzwerk bei Beginn nicht betreten konnten, beenden sich selbst. Außerdem konnte nicht immer auf allen Knoten das Programm gestartet werden.

Abbildung 25:

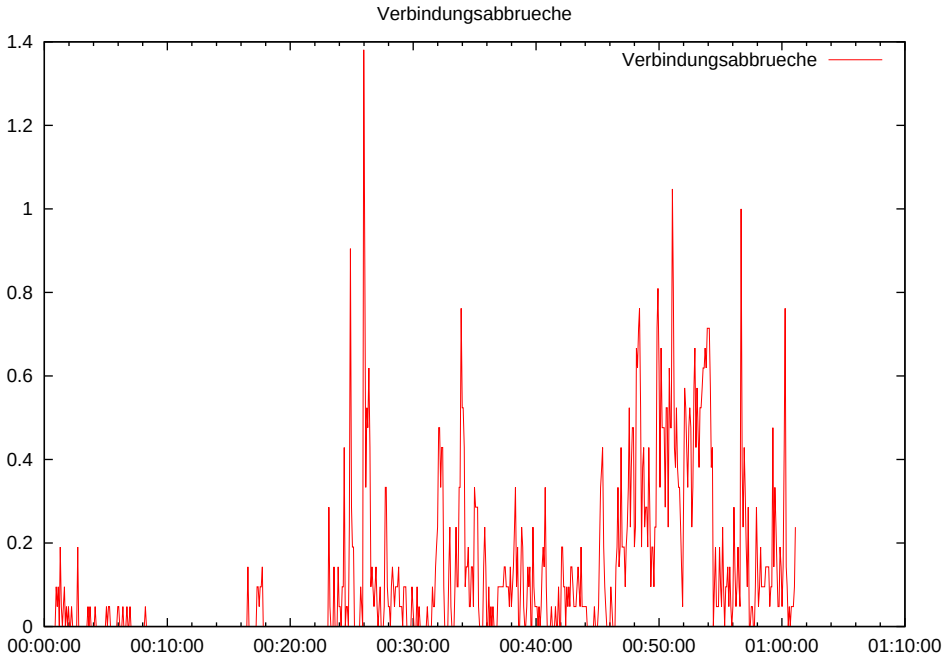


In diesem Graph werden die Latenzen dargestellt. Die durchschnittliche gemessene direkte Latenz befindet sich etwa im Bereich von 4 Sekunden. Dies zeigt eindrucksvoll den Zustand des PlanetLab-Netzwerks. Die Übertragungszeiten sind sehr hoch, was höchstwahrscheinlich dadurch bedingt ist, dass die Knoten zu geringe Bandbreiten frei haben und so Nachrichten im Sendepuffer von TCP bzw. vor der Methode `AbstractNetwork.Connection.sendMessage()` warten. Allerdings zeigte sich in Stichproben ein Paketverlust von etwa 10%.

Die mit Vivaldi geschätzten Latenzen befinden sich in der Nähe der gemessenen Latenz; der Vivaldi-Fehler ist gering.

Die durchschnittliche geroutete Latenz bewegen sich in der Größenordnung von 25 Sekunden. Daraus ergibt sich eine durchschnittliche Hopzahl von etwa 6.

Abbildung 26:

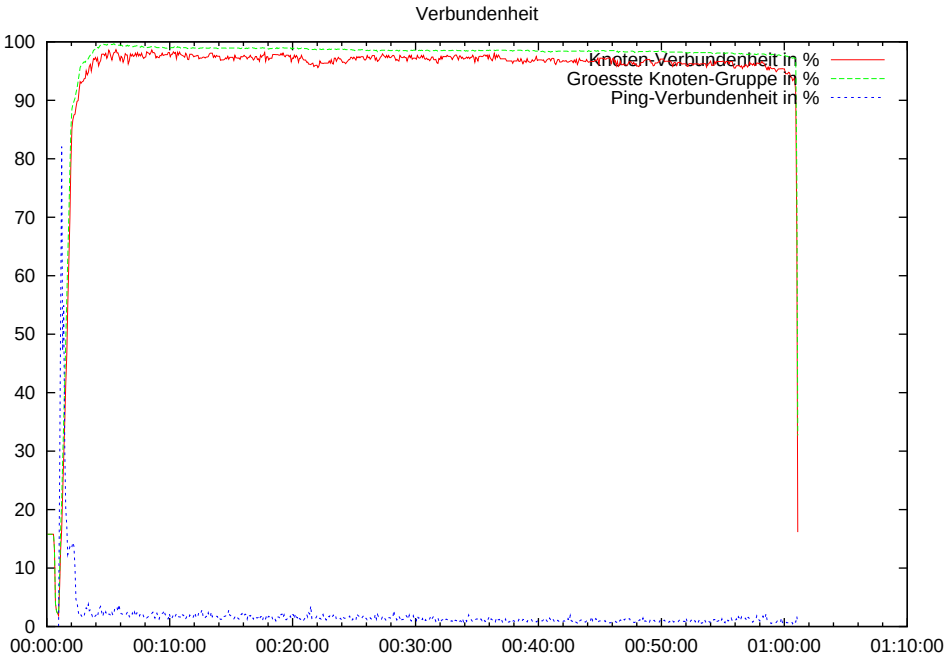


Hier wird die durchschnittliche Anzahl von Verbindungsabbrüchen in Intervallen von 5 Sekunden dargestellt. Gezählt wurden nur unbeabsichtigte Verbindungsabbrüche, also Knotenausfälle und das Abbrechen einer Verbindung, ohne dass diese geschlossen wurde.

Das PlanetLab-Netzwerk zeigt eine sehr hohe Zahl an Verbindungsabbrüchen. Zusätzlich scheitert der Verbindungsaufbau des Öfteren, was hier nicht mitgezählt wurde.

Das Netzwerk musste diese durch Stabilisierungsmaßnahmen und Selbstreparatur ausgleichen.

Abbildung 27:



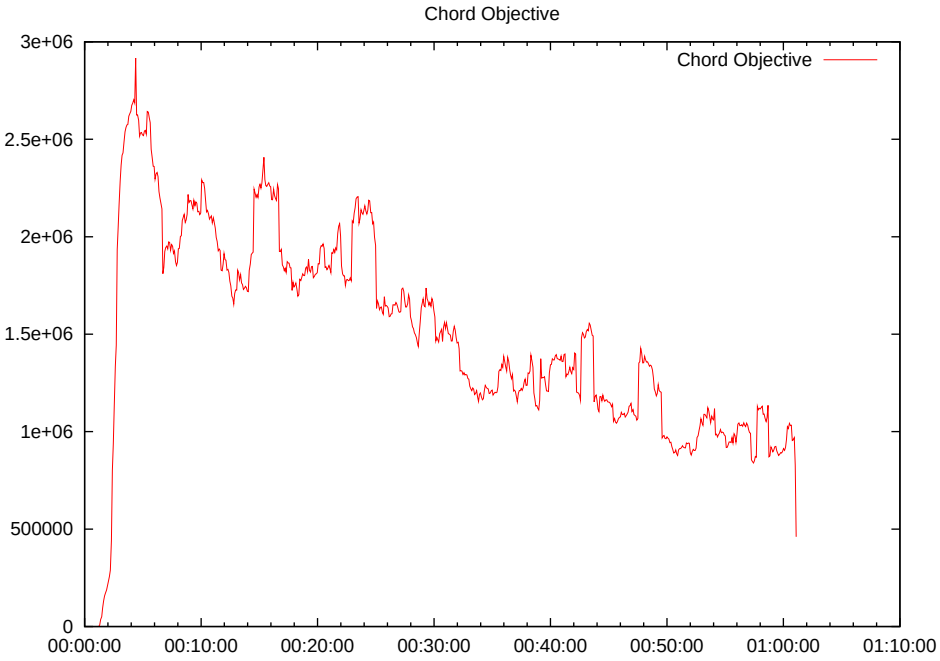
In dieser Grafik wird die Verbundenheit der Knoten dargestellt. Die Verbundenheit ist definiert als die durchschnittliche Menge von Knoten, die von anderen Knoten erreicht werden können, und wird relativ zur Gesamtzahl an Knoten angegeben.

Das Netzwerk zeigt dauerhaft eine Verbundenheit von über 95%. D. h. jeder Knoten kann 95% der anderen Knoten erreichen.

Die größte Knotengruppe zeigt sogar eine Verbundenheit von beinahe 100%. Das lässt den Schluss zu, dass einzelne Knoten nicht Teil des Netzwerks sind und so zu keinem anderen Knoten Kontakt haben.

Dieser Graph zeigt deutlich, dass das Netzwerk immer stark verbunden bleibt. Dies ist insbesondere wegen dem schlechten Zustand des PlanetLab-Netzwerks beachtlich.

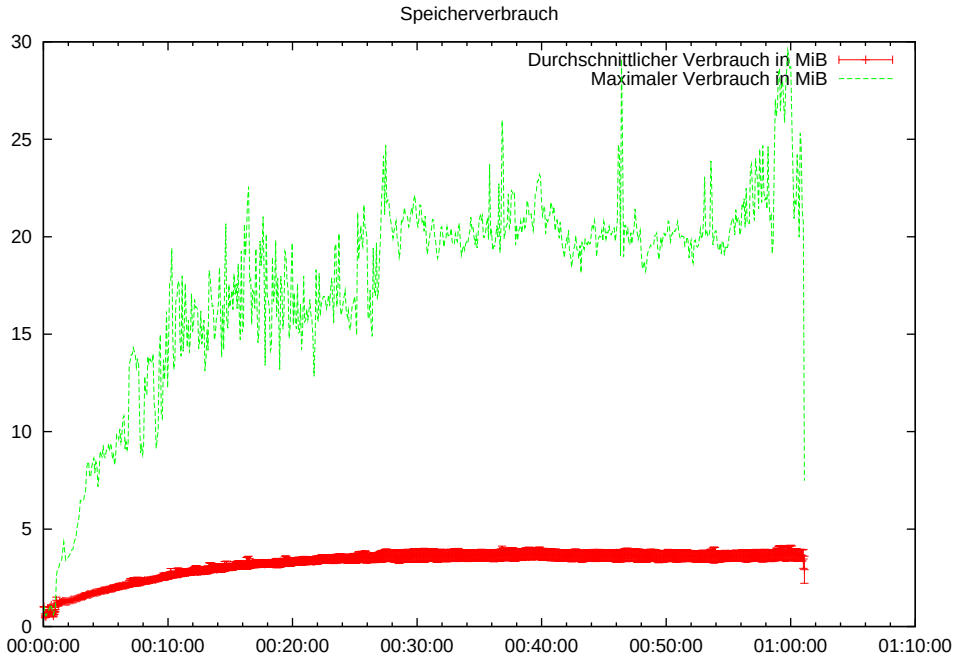
Abbildung 28:



In diesem Graph wird die Chord-Objective dargestellt. Sie ist definiert als die Summe der Latenzen aller von Routen im Netzwerk benutzter Verbindungen. D. h. niedrigere Werte deuten bei gleichbleibender Knotenzahl auf eine verbesserte Topologie hin.

Der Verlauf des Graphen zeigt deutlich die Verbesserung der Topologie durch optimierte Fingertabellen.

Abbildung 29:



Der durchschnittliche Speicherverbrauch der Anwendung konvergiert gegen 4 MiB. Allerdings hängt der Speicherverbrauch von der Anzahl der Verbindungen ab und steigt so logarithmisch mit der Netzwerkgröße.

Der maximale Speicherverbrauch schwankt sehr stark, da unbelegter Speicher erst bei der Garbage-Collection freigegeben wird. Das Verhalten des Garbage-Collectors ist in Java nicht exakt spezifiziert. Die Garbage-Collection der Virtual-Machine von Sun wird anscheinend immer wieder nach einer gewissen Zeitspanne ausgeführt, oder wenn der verbrauchte Speicher einen neuen Höchststand erreicht. Dadurch erreicht der maximale Verbrauch 25 MiB. Der tatsächlich belegte Speicher sollte noch geringer sein, als der hier ermittelte durchschnittliche Speicherverbrauch.

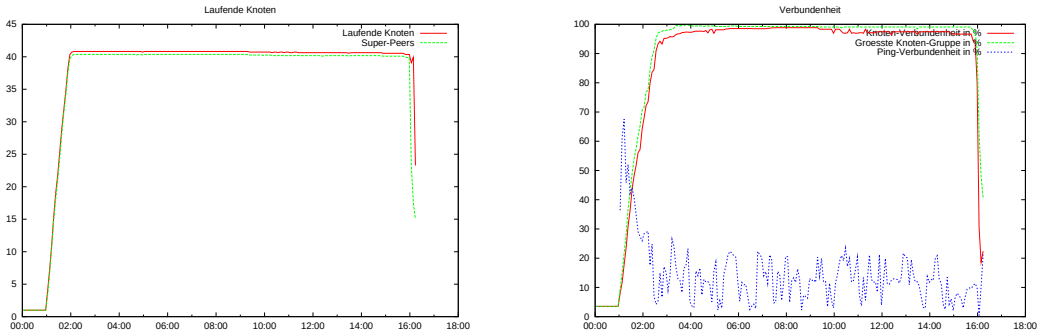
10.2.2 Zweite Serie

Die zweite Messungsserie besteht aus 11 Einzelmessungen mit je einer Viertelstunde Laufzeit. Der Speicherverbrauch wurde auf 150 MiB beschränkt. Weiterhin wurden folgende Parameter verwendet:

- Chord-SPSA wurde deaktiviert, d. h. alle Peers werden zu Super-Peers falls sie erreichbar sind.

- PIS ist aktiviert.
- PRS ist aktiviert.
- Der Socket-Timeout wurde auf 60000 – also 1 Minute – gesetzt.

Abbildung 30:

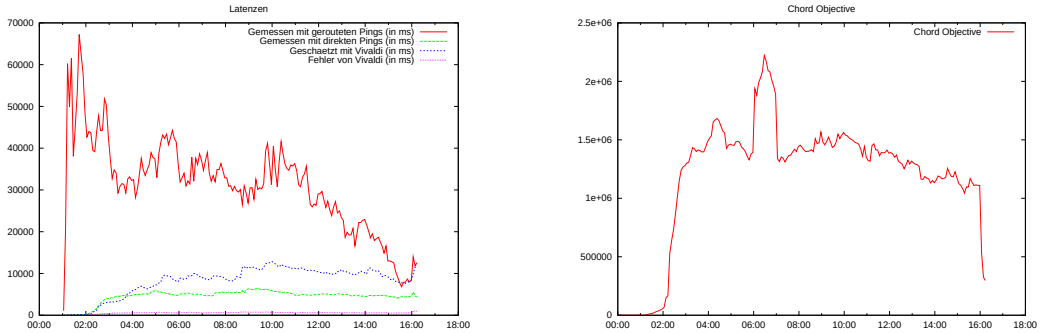


In diesem Test konnten nicht alle der 62 Knoten verwendet werden, da bei manchen Knoten Probleme auftraten.

Der Test zeigt, dass PIS sich am Anfang negativ auf die Verbundenheit auswirkt. Nach 2 Minuten laufen alle Knoten, aber die Verbundenheit liegt nur bei ca. 80%. Das lässt sich dadurch begründen, dass PIS als Position immer nur die Positionen zwischen zwei Knoten in Betracht zieht. Da am Anfang nur wenige Knoten im Netzwerk vorhanden sind und viele das Netzwerk betreten wollen, treten häufig Kollisionen auf, die den Aufbau des Netzwerks verzögern.

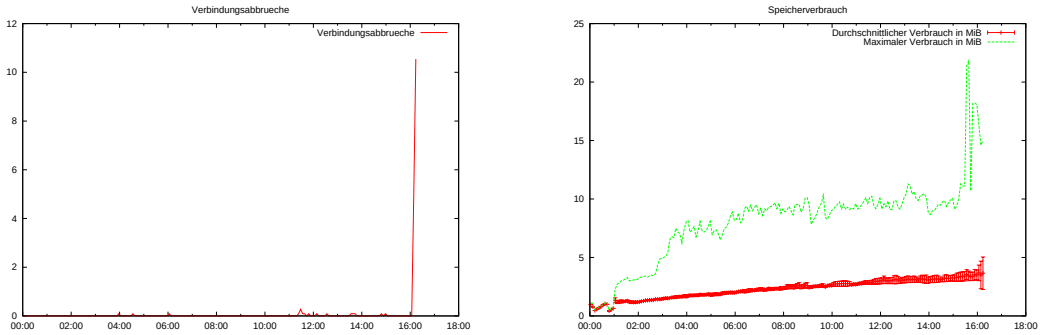
Langfristig hat PIS keine negativen Auswirkungen auf die Stabilität.

Abbildung 31:



Diese Graphen zeigen, dass die Objective langsam sinkt und das durchschnittliche Delay ebenfalls sinkt.

Abbildung 32:



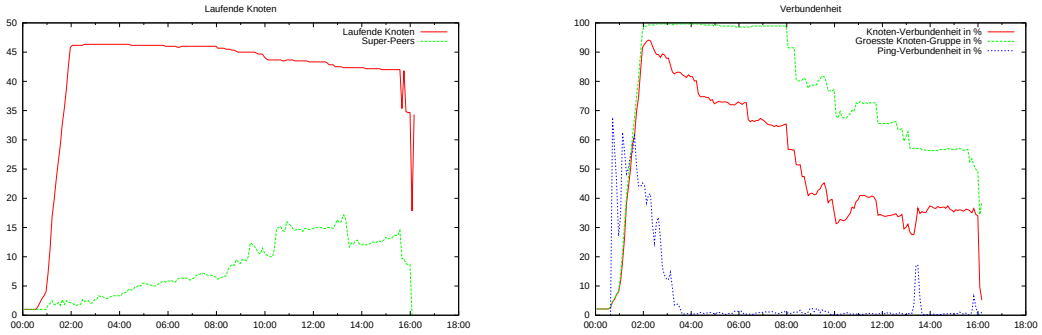
Die Verbindungsabbrüche und der Speicherverbrauch werden von PIS nicht beeinflusst.

10.2.3 Dritte Serie

Die dritte Messungsserie besteht aus 6 Einzelmessungen mit je einer Viertelstunde Laufzeit. Der Speicherverbrauch wurde auf 150 MiB beschränkt. Weiterhin wurden folgende Parameter verwendet:

- Chord-SPSA wurde aktiviert
- PIS ist deaktiviert.
- PRS ist aktiviert.
- Der Socket-Timeout wurde auf 60000 – also 1 Minute – gesetzt.

Abbildung 33:

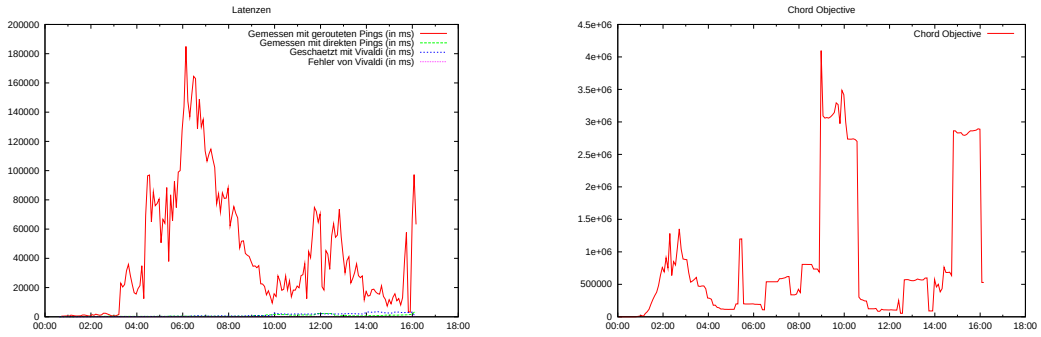


In diesem Test konnten nicht alle der 62 Knoten verwendet werden, da bei manchen Knoten Probleme auftraten.

Hier zeigt sich, dass die Verbundenheit der Knotenwolke durch Chord-SPSA stark leidet. Die mangelhafte Stabilität des PlanetLab und die Umgestaltung des Netzwerks durch Chord-SPSA können von der Selbstheilung des SuperChord-Netzwerks nicht mehr ausgeglichen werden.

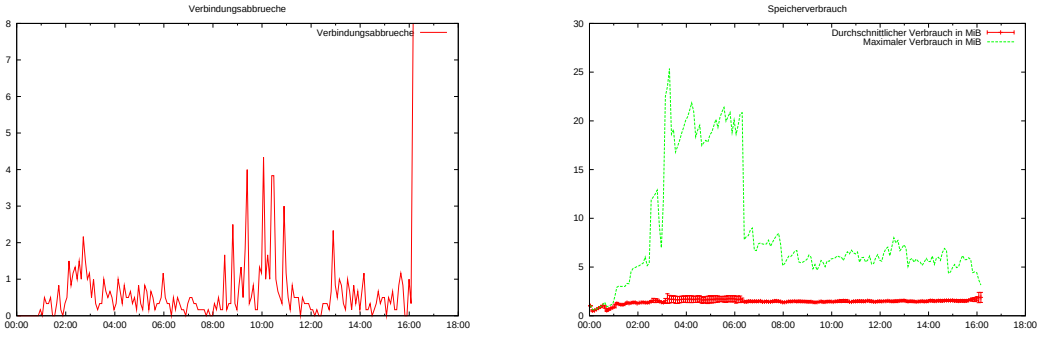
Das Verhältnis von Super-Peers zu Edge-Peers steigt während der Testphase auf den Wert von etwa 4 an. Die obere Schranke war 8, die untere 2.

Abbildung 34:



Da das Netzwerk während der Laufzeit zerfällt und am Ende nur noch schwach verbunden ist, schwanken die Werte der gerouteten Latenzen und der Objective sehr stark und sind wenig aussagekräftig.

Abbildung 35:



Die Anzahl der abgebrochenen Verbindungen ist in dieser Testserie sehr hoch. Durch Chord-SPSA wird das Netzwerk stark umgebaut und ständig Verbindungen erstellt und getrennt.

Der Speicherverbrauch liegt im Mittel bei 1.5 MiB und damit niedriger als in den vorherigen Testserien. Das liegt daran, dass Edge-Peers kein Objekt der Chord-Klasse haben.

10.3 Fazit

Die Messungen zeigen deutlich die geringe Qualität des PlanetLab-Netzwerks aber auch dass SuperChord in diesen Bedingungen immer noch funktioniert.

Das Netzwerk bleibt trotz sehr hoher Latenzen und zahlreicher Verbindungsabbrüche stark verbunden. Die Objective sinkt über die Laufzeit, d. h. das Netzwerk optimiert sich.

In der zweiten Testserie zeigt sich, dass PIS den Aufbau des Netzwerks verlangsamt aber ansonsten keine negativen Auswirkungen hat. Die positiven Effekte können aufgrund der starken Schwankungen der Messwerte im PlanetLab nicht genau ermittelt werden.

Die dritte Testserie zeigt, dass Chord-SPSA die Stabilität des Netzwerks negativ beeinflusst. In dieser Testanordnung im PlanetLab war das Netzwerk nicht in der Lage sich schnell genug zu reparieren; die Verbundenheit des Netzwerks sank sehr stark ab. Da das Netzwerk bei der Benutzung von Chord-SPSA zerbrach, sind die positiven Effekte von Chord-SPSA nicht sichtbar.

In dieser Testanordnung zeigte sich, dass für ein möglichst stabiles Netzwerk Chord-SPSA deaktiviert werden sollte. In anderen, weniger instabileren, Umgebungen könnte aber Chord-SPSA das Netzwerk noch zusätzlich optimieren.

11 Ausblick

Diese Netzwerkstruktur wurde für die Realisierung eines Jobverteilungssystems entworfen. Allerdings sind dadurch, dass das Netzwerk problemneutral gehalten ist, auch andere Nutzungsmöglichkeiten denkbar.

11.1 Jobverteilungssystem

Um ein Jobverteilungssystem auf diesem Netzwerk zu realisieren, sind einige wesentliche Schritte nötig.

Zum Einen muss der ausführbare Code übertragen werden. Das Netzwerk bietet dazu einige Möglichkeiten. Aus Effizienzgründen könnte allerdings auch ein externes Verfahren wie FTP, HTTP oder Bittorrent genutzt oder aufbauend auf dem Netzwerk realisiert werden.

Zum Anderen muss der Code in einer Weise ausgeführt werden dass die einzelnen Instanzen kommunizieren können. Dazu ist es sinnvoll, von dem Code zu fordern, dass er eine `run`()-Methode besitzt, der man per Parameter ein Callback-Objekt mitgeben kann. Dieses Callback-Objekt besitzt dann Methoden zum Senden von Nachrichten und Abholen empfangener Nachrichten.

11.2 Mögliche weitere Anwendungen

Eine weitere mögliche Anwendung wäre ein Datenverteilungssystem. Ein solches System erlaubt es, Dateien auszutauschen und nach verfügbaren Dateien zu suchen.

Für das Austauschen von Dateien können die Dateien in Blöcke gespalten und einzeln übertragen werden. Das Aufspalten in Blöcke hat den Vorteil, dass Blöcke mit einem Verfahren ähnlich zu Bittorrent [5] aus verschiedenen Quellen angefordert werden können. Um den Datendurchsatz zu erhöhen, können direkte Verbindungen hergestellt werden, falls dies möglich ist.

Um verfügbare Dateien zu suchen, gäbe es zwei Ansätze. Als Erstes kann eine Suche per Broadcast oder limitiertem Broadcast erfolgen. Zweitens könnte ein Dateianbieter Informationen über die Dateien im Chord-Ring wie in einer DHT ablegen. Dadurch würden Suchen nur logarithmischen Aufwand haben. Allerdings wäre der Verwaltungsaufwand zum Speichern der Metadaten im Chord-Ring sehr hoch und es ließen sich keine komplexen Suchen realisieren.

11.3 Anpassungs- und Verbesserungsmöglichkeiten

- Wenn alle erreichbaren Peers zu Super-Peers werden, also kein Downgrade stattfindet, wird das Netzwerk deutlich stabiler, da es mehr Super-Peers gibt und der Ausfalls eines Knotens das Netzwerk weniger beeinflusst.

- Wenn Chord-SPSA nicht verwendet würde, könnte man den Edge-Peer selbst an dem für ihn zuständigen Super-Peer anhängen. Der zuständige Super-Peer kann mit der Hash-Funktion des Naming-Service ermittelt werden. Dadurch würde man das gesamte Naming-Verfahren einsparen. Das Netzwerk würde zusätzlich noch stabiler, da keine Reorganisation mehr stattfindet.

- Falls nötig, kann eine Liste von STUN-Servern angegeben werden, die nacheinander oder sogar gleichzeitig befragt werden. So kann der STUN-Server als Single-Point-of-Failure ausgeschlossen werden. Alternativ könnte die STUN-Funktionalität auch durch das Netzwerk selbst geleistet werden, wenn das Problem des zweiten Knotens gelöst werden kann.

- Um eine Spaltung des Netzwerks zu verhindern, kann zusätzlich zu den bisherigen Vorkehrungen ein besonderer Mechanismus eingeführt werden. Jeder Peer merkt sich eine kleine Menge von Peers und überprüft deren Verfügbarkeit. Fällt eine hohe Zahl von Peers gleichzeitig weg, versucht er längere Zeit zu diesen Peers wieder zu verbinden.

- Die Verbindung zu einem Finger könnte auch in die Rückrichtung verwendet werden [12; 26]. Dazu müsste Chord so angepasst werden, dass die Knoten aller Verbindungen bekannt sind und zum Routen und Broadcasten verwendet werden.

- Bisher werden Nachrichtentypen geschachtelt. Das bedeutet, dass die Nachricht, bis sie tatsächlich versendet wird, von zahlreichen Containern umgeben wird um einzelne

Header hinzuzufügen. Dies wurde eingeführt, um die Layer-Struktur der Anwendung zu erhalten.
Man könnte allerdings auch die Schachtelung fest vorgeben, d. h. es ist festgelegt, welcher Typ in welchem steckt. Dann kann auf die Verwendung von `ObjectOutputStream.writeObject()` verzichtet und direkt die `Externalizable`-Methoden der entsprechenden Objekte aufgerufen werden. Dadurch werden die Klassenheader eingespart, die `ObjectOutputStream` schreiben würde.
Zusätzlich könnte man das Container-Prinzip aufgeben und ein alle Layer übergreifendes Nachrichtenformat einführen. Dadurch könnte der Overhead auf ein Minimum reduziert werden.

- Der ReconnectService könnte erweitert werden, um nach dem Verlassen wieder in das Netz zu kommen. Dazu würde die vom Service gepflegte Menge der bekannten Peers beim Beenden in eine Datei geschrieben. Beim Start würde diese Datei dann wieder gelesen und die enthaltenen Peers in die Liste der bekannten Peers eingefügt. Der ReconnectService übernimmt dann selbstständig die Verbindungsaufnahme.

- Damit große Anwendungsnachrichten die kleinen Verwaltungsnachrichten nicht unnötig blockieren, könnte die abstrakte Netzwerkschicht so erweitert werden, dass sie bei Bedarf die Verbindung dupliziert und eine Verbindung ausschließlich für "wichtige" Nachrichten verwendet. Dazu müsste im gesamten Netzwerk zwischen "wichtigen" Verwaltungspaketen und Nutzdaten unterschieden werden.

11.4 Neues Konzept: ChordNet

Nachdem in dieser Arbeit Chord im Zusammenhang mit Super-Peer/Edge-Peer-Netzwerken ausgiebig behandelt wurde, werden nun die Erfahrungen und Verbesserungsvorschläge zusammengefasst und ein neues Konzept für ein Peer-to-Peer-Netzwerk vorgestellt, das teilweise auf Konzepten der hier erarbeiteten Netzwerkstruktur basiert, teilweise aber auch neue Konzepte einführt.

Wesentliche Änderungen zu dem hier erarbeiteten Netzwerk sind:

- Chord-Netzwerk mit Finger in beide Richtungen[12].

- Einheitliche Adressierung mit IDs, auch Edge-Peers bekommen Chord-IDs. Dadurch kann der NamingService komplett eingespart werden.

- Möglichst viele Super-Peers. Durch die Maximierung der Super-Peer-Anzahl soll das Netzwerk stabilisiert werden.

- Keine Topologie-Optimierung durch Downgrade von Super-Peers zu Edge-Peers oder Ändern von IDs. Dadurch soll die Stabilität des Netzwerks auch in widrigen Umständen gesichert werden. Außerdem kann so auf Chord-SPSA, PIS und Vivaldi verzichtet und das Netzwerk deutlich vereinfacht werden. PRS kann weiterhin angewendet werden.

- Trennung in Schichten wie im OSI-Schichtenmodell. Das abstrakte Netzwerk stellt die Übermittlung von Nachrichten zwischen zwei Knoten sicher (vgl. Ethernet). Eine Routingschicht regelt die Organisation des Netzwerks und die Übermittlung von Daten im Netzwerk (vgl. IP). Höhere Schichten bieten Dienste ähnlich zu UDP und TCP an. Dadurch soll der Programmcode vereinfacht und die Schachtelungstiefe verringert werden.

- Einfaches, direktes Nachrichtenformat. Um Nachrichten einfach kodieren und dekodieren zu können, wird ein einfaches sprachenunabhängiges Format definiert, das ohne Serialisierung auskommt. Dadurch kann auf Serialisierung verzichtet werden was den Overhead stark reduziert und den Nachrichtenaustausch beschleunigt.

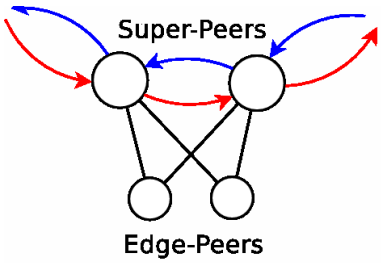
11.4.1 Netzwerk-Architektur

Jeder Knoten hat eine ID aus einem festen Bereich (z.B. $[0\,..\,2^{32}-1]$). Die IDs bilden einen Moduloring. Hierbei wird im Allgemeinen die Richtung des Rings in Richtung aufsteigender IDs angenommen (Vorgänger, Nachfolger). Es gibt zwei Sorten von Knoten, Super-Peers und Edge-Peers. Alle Knoten, die im freien Internet erreichbar sind, werden Super-Peers, alle anderen Edge-Peers.

Super-Peers halten eine Menge von Verbindungen zu anderen Super-Peers, die Finger genannt werden. Die Finger sind die Super-Peers, deren IDs möglichst nahe an bestimmten IDs liegen. Die IDs, an denen Finger liegen, berechnen sich aus $EigeneID \pm 2^{i-1}$mit i von 1 bis zum 2er-Exponenten des ID-Bereichs. Die Finger in Ring-Richtung bekommen positive Nummern, die in die Gegenrichtung negative. Dabei wird darauf geachtet, dass Finger-Positionen vor/hinter der eigenen ID nicht von Knoten aus der anderen Richtung belegt werden, also insbesondere $Finger[1] \neq Finger[-1]$ bei Netzen mit mehr als zwei Knoten. Durch diese Konstruktion ist immer der Super-Peer mit der nächsthöheren/nächstniedrigeren ID (Nachfolger/Vorgänger) ein Finger und die Fingerbeziehung ist oft beidseitig.

Edge-Peers haben Verbindungen zu den zwei Super-Peers, zwischen denen sich ihre ID befindet. Edge-Peers können keine Finger sein oder haben.

Abbildung 36: ChordNet-Struktur



Beim Routen zu einer ID werden alle Verbindungen beachtet, ob Finger oder nicht. Es wird der Knoten mit dem geringsten Abstand zu der ID als nächster Hop gewählt. Hier kann allerdings auch ein PRS-Verfahren angewendet werden, solange sichergestellt

ist, dass die Distanz zur ID des Zielknotens streng monoton sinkt. Wird eine Nachricht an mehrere IDs geroutet, werden die IDs einzeln betrachtet und später pro Hop-Peer wieder zusammengefasst.

Beim Weiterleiten eines Broadcasts wird die Menge aller verbundenen Knoten nach IDs sortiert. Dann wird der Broadcast-Bereich in der Mitte zwischen zwei Knoten aufgespalten und die aufgespaltenen Broadcasts an den jeweiligen Knoten weitergeleitet. So wird jede ID an den Knoten weitergeleitet, der die geringste ID-Distanz zu ihr hat.

Jeder Knoten führt eine Liste aller mit ihm verbundenen Super-Peer-Knoten, die Peer-Liste. Um die jeweils besten Knoten für die Finger-Positionen zu finden, tauschen die Knoten mit ihren Fingern regelmäßig die Peer-Listen aus und nutzen die Knoten daraus für sich selbst.

Will ein neuer Knoten das Netzwerk betreten, so wird er ähnlich einer gerouteten Nachricht durch das Netzwerk zu seiner ID geleitet, bis sein direkter Vorgänger und Nachfolger bekannt ist. Verlässt ein Super-Peer das Netzwerk, so fügt er seinen Vorgänger und Nachfolger zusammen und übergibt seine Edge-Peers an den Vorgänger bzw. Nachfolger.

11.4.2 Wartung und Stabilisierung

Alle Knoten senden periodisch Ping-Nachrichten an alle verbundenen Knoten. Mit einem gleitenden Mittel kann so die Latenz einer Verbindung ermittelt werden.

Knoten senden periodisch GetPeerList-Nachrichten an alle verbundenen Knoten. Das Ergebnis, die PeerList-Nachrichten, werden dazu verwendet, die eigene Finger-Tabelle zu verbessern.

11.4.3 Objekte

Alle Objekte werden als TLV[25] kodiert. Dabei ist Type ein Byte-Wert und Length ein Short-Wert. Falls Objekte einander beinhalten, werden die inneren Objekte ohne eigenen Type-Length-Header kodiert. Die äußeren Objekte haben dann die Gesamtlänge als Length-Attribut. Die Basis-Datentypen sind wie folgt definiert:

- Boolean: True oder False, gespeichert in einem Byte als 0x01 und 0x00
- Byte: Ein Byte, Werte von 0 (0x00) bis 255 (0xFF)
- Short: 2 Bytes, BigEndian, Werte von 0 bis $2^{16} - 1$
- Integer: 4 Bytes, BigEndian, Werte von 0 bis $2^{32} - 1$
- Long: 8 Bytes, BigEndian, Werte von 0 bis $2^{64} - 1$

[25] Type Length Value

Name	Type	Length	Value
Address	0x01	7 oder 19	len:Byte len-Byte Adresse, Portnummer:Short
ID	0x00	8	Long
ChordAddr	0x02	15 oder 27	Adresse:Address, ID
IDRange	0x08	16	RangeStart:ID, RangeEnd:ID
IDList	0x09	2+8*len	len:Short, ID*len
PeerList	0x20	2+(15 oder 27)*len	len:Short, ChordAddr*len
PingData	0x10	9	Stage:Byte, Time:Long
IsSuperPeer	0x21	1	Boolean
BroadcastDst	0x78	17	Flags:Byte, IDRange
RoutingDst	0x79	3+8*len	Flags:Byte, IDList
Data	0x7A	len	Byte*len (len aus TLV, kann nicht geschachtelt werden)
DataType	0x40	2	Short
DataTimeout	0x41	8	Long

Flags für BroadcastDst:

- 1. Bit: Nachricht soll an alle Super-Peers gesendet werden.
- 2. Bit: Nachricht soll an alle Edge-Peers gesendet werden.
- Alle anderen Bits sind reserviert.

Flags für RoutingDst:

- 1. Bit: Nachricht soll bei nicht-existenter ID an den linken Super-Peer gesendet werden.
- 2. Bit: Nachricht soll bei nicht-existenter ID an den rechten Super-Peer gesendet werden.
- 3. Bit: Existiert die Ziel-ID nicht, soll eine Nachricht zurückgesendet werden.
- Alle anderen Bits sind reserviert.

11.4.4 Nachrichtentypen

Alle Nachrichten sind als TLV kodiert. Dabei ist sowohl Type als auch Length ein Byte-Wert.

Length ist die Anzahl der Parameter, nicht die Gesamtlänge.

Als Value folgen alle Parameter-Objekte in der angegebenen Reihenfolge.

Alle Nachrichten sind so entworfen, dass das Protokoll zustandslos wird.

Wenn eine Implementierung einen Nachrichtentyp oder einen Parameter nicht kennt, soll sie diesen ignorieren. Da die Nachrichten und Objekte dem TLV-Schema folgen, kann dies geschehen, ohne den Datenstrom abzubrechen.

Die Liste der Nachrichtentypen nennt den Namen, die Parameter in Klammern und die Type-Konstante. Bei den Parametern bedeutet ein Fragezeichen einen optionalen Parameter und ein senkrechter Strich eine Entweder-oder-Entscheidung.

Ident(ChordAddr) [0x11]

Diese Nachricht wird gesendet, um der Gegenstelle die eigene Identität mitzuteilen.

Disconnect() [0x12]

Diese Nachricht wird gesendet, um anzuzeigen, dass die Verbindung bewusst beendet wird und nicht wieder aufgebaut werden soll.

Ping(PingData) [0x18]

Ping-Nachricht mit Zeitstempel/Zeitdifferenz. Diese Nachricht wird in Folge einer Ping-Pong-Peng-Sequenz versendet.

- Stage 1 (Ping): Die Nachricht wird mit einem lokalen Zeitstempel als Parameter gesendet.
- Stage 2 (Pong): Dies ist die Antwort auf Stage 1. Diese Nachricht enthält den Zeitstempel aus der Originalnachricht.
- Stage 3 (Peng): Dies ist die Antwort auf Stage 2. Diese Nachricht enthält die gemessene RTT in Nanosekunden.

FindJoinNode(ChordAddr) [0x20]

Join-Wunsch mit Eigeninformation. Diese Nachricht wird an einen Knoten versendet, um die Position zum Einfügen in das Netzwerk zu finden. Diese Nachricht muss mit NextJoinNode, JoinHere oder DuplicateId beantwortet werden.

Ziel ist es, die zwei Super-Peers zu finden, die direkt vor bzw. hinter der als Parameter angegebenen ID sind.

Empfängt ein Knoten diese Nachricht und kennt einen anderen Knoten mit der gleichen ID, so antwortet er mit DuplicateId.

Ist die Position gefunden, so wird JoinHere gesendet, ansonsten NextJoinNode.

NextJoinNode(ChordAddr) [0x21]

Weiterleitung mit Knoteninformation. Diese Nachricht wird als Antwort auf FindJoinNode versendet, wenn die gesuchte Position nicht gefunden wurde und enthält einen Knoten, der zum Weitersuchen kontaktiert werden soll.

JoinHere(ChordAddr,ChordAddr) [0x22]

Join-Ziel gefunden mit Predecessor und Successor. Diese Nachricht wird als Antwort auf FindJoinNode gesendet, wenn die Position gefunden wurde, und enthält den zukünftigen Predecessor und Successor als Parameter.

DuplicateId(ChordNode) [0x23]

Doppelte ID im Netz mit Knoten-Info. Diese Nachricht wird als Antwort auf FindJoinNode gesendet, wenn bei der Suche einer Position eine doppelte ID festgestellt wird. Der als Parameter gesendete Knoten hat die gleiche ID wie der Empfängerknoten.

Joining(ChordNode,IsSuperPeer) [0x24]

Join-Indikation mit Eigeninformation und Super-Peer-Flag. Diese Nachricht wird an Predecessor und Successor versendet, wenn die Position zum Betreten des Netzwerks gefunden wurde.

Wenn IsSuperPeer true ist, wird der Sender zwischen diesen Knoten eingefügt, ansonsten wird er als Edge-Peer aufgenommen. Die Antwort ist in beiden Fällen Joined.

Joined() [0x25]

Der Empfänger hat das Netzwerk erfolgreich betreten und wurde vom Sender eingefügt.

ChangeSuperPeer(ChordAddr) [0x26]

Neuer Super-Peer für Edge-Peers mit Info. Diese Nachricht wird von einem Super-Peer an seinen Edge-Peer versendet, wenn dieser zu einem anderen Super-Peer verbinden muss.

Parting((ChordAddr,ChordAddr)?) [0x27]

Austrittsnachricht mit Predecessor und Successor (nur bei Super-Peers). Diese Nachricht wird von einem Peer an alle Finger und Edge-Peers bzw. seinen Super-Peer versendet wenn er das Netzwerk verlassen will. Sendet ein Super-Peer diese Nachricht, enthält sie den Vorgänger und Nachfolger als Parameter.

Der Knoten, der diese Nachricht empfängt, muss den Sender als Finger, Super-Peer oder Edge-Peer entfernen. Um die Lücken zu füllen, können die Knoten in den Parametern verwendet werden.

GetPeerList(PeerList?) [0x30]

Peer-List-Anfrage. Diese Nachricht ist die Aufforderung, eine Liste aller Peers zu senden, sie muss mit PeerList beantwortet werden. Diese Nachricht enthält die Peer-Liste des Senders. Zusätzlich enthält sie den Sender-Knoten selbst, falls dieser ein Super-Peer ist.

PeerList(PeerList?) [0x31]

Peer-Liste mit Eigeninformation. Diese Nachricht enthält die Peer-Liste des Senders. Zusätzlich enthält sie den Sender-Knoten selbst, falls dieser ein Super-Peer ist.

StoreData(ID,DataType,Data,Data,DataTimeout) [0x40]

Diese Nachricht speichert eine Information, indem der Chord-Ring als DHT gebraucht wird. Die Parameter sind der Hashwert, der Datentyp, der Schlüssel und die Daten. Diese Nachricht wird durch den Ring zu der ID geroutet.

Die Information wird in beiden, die ID umgebenden, Super-Peers gespeichert. Die speichernden Knoten müssen die Information mindestens bis zum in Millisekunden angegebenen Timeout speichern, selbst wenn sie die Information nicht verstehen.

Es kann für jeden Datentyp und Schlüssel nur eine Information gespeichert sein; neuere Informationen überschreiben ältere.

GetData(ID,ID,DataType,Data) [0x41]

Diese Nachricht ist die Aufforderung, die im Ring gespeicherten Daten zu senden. Die erste ID ist die des Senders, die zweite ist die der Daten. Diese Nachricht wird durch den Ring zu der ID geroutet.

Diese Nachricht muss mit GetDataResult beantwortet werden.

GetDataResult(ID,ID,DataType,Data,Data?) [0x42]

Diese Nachricht ist das Ergebnis einer GetData-Anfrage. Die erste ID ist die des Empfängers, die zweite ist die der Daten. Diese Nachricht wird durch den Ring zu der ID geroutet.

Wenn die Daten gefunden wurden, werden sie im letzten Parameter mitgesendet.

Message(ID,(BroadcastDst|RoutingDst),Data,Data?) [0x78]

Anwendungs-Nachricht mit Sender, Empfänger und Daten. Diese Nachricht enthält Anwendungsdaten. Der erste Datenblock enthält die reinen Anwendungsdaten, der zweite Metadaten.

UndeliverableMessage(ID,RoutingDst,Data,Data?) [0x79]

Die ursprüngliche Nachricht konnte nicht zugestellt werden.

Diese Nachricht enthält die Ziel-IDs, die nicht existieren, und die gesendeten Daten. Der erste Datenblock enthält die reinen Anwendungsdaten, der zweite Metadaten.

11.4.5 Kommunikations-Beispiele

- Ping-Zyklus: A:Ping(Stage=1) -> B:Ping(Stage=2) -> A:Ping(Stage=3)
- Join-Zyklus: A:FindJoinNode -> B:NextJoinNode -> A:FindJoinNode -> C:JoinHere -> A:Joining -> C:Joined

11.4.6 Kodierungs-Beispiele

Beispiel 1:

- kodiert: 0x78 03 00 0008 0000000000000005 79 000B 00 0001 0000000000000009 7A 000A ”Hallo Welt”
- bedeutet:Message(ID=5,RoutingDst=(Flags=0,IPList=[9]),Data=”Hallo Welt”)
- Größe: 40 Bytes

Beispiel 2:

- kodiert: 0x78 03 00 0008 0000000000000005 78 0011 00 0000000000000006 0000000000000009 7A 000A ”Hallo Welt”
- bedeutet:Message(ID=5,BroadcastDst=(Flags=0,IDRange(RangeStart=6,RangeEnd=9)), Data=”Hallo Welt”)
- Größe: 46 Bytes

Beispiel 3:

- kodiert:0x12 00
- bedeutet:Disconnect()
- Größe: 2 Bytes

11.4.7 Statistiken

- verschiedene Objekttypen: 13
- verschiedene Nachrichtentypen: 18
- Overhead einer gerouteten Nachricht: 30 Bytes + 8 Bytes je weiteres Ziel
- Overhead einer Broadcast-Nachricht: 36 Bytes

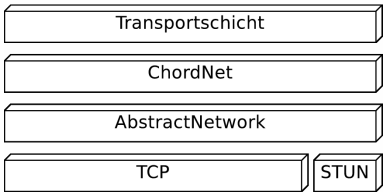
11.4.8 Beschränkungen

- Je nach Wahl des Exponenten des ID-Bereichs ist die maximale Knotenzahl beschränkt. Die höchstmögliche Knotenzahl ist 2^{64}.
- Eine Nachricht kann maximal 64 KiB Daten und 64 KiB Metadaten enthalten.
- Nachrichten können per Multicast an maximal 8191 Empfänger gesendet werden. Diese Zahl ergibt sich, da ein Objekt maximal 64KiB Daten enthalten kann und das Objekt RoutingDst einen 3 Byte langen Header hat.
- Bei gespeicherten Daten kann der Schlüssel und der Wert 64 KiB nicht übersteigen.

11.4.9 Transportschicht

Um größere Datenmengen zu versenden, muss der Datenstrom in mehrere Pakete aufgeteilt werden. Dabei ist es wichtig, dass keine Pakete vertauscht werden oder verloren gehen. Um dies zu gewährleisten, sind weitere Protokollschichten nötig.

Abbildung 37: Blockschaubild der Komponenten



Um verschiedene Kommunikationsstränge zu unterscheiden ist es sinnvoll, in höheren Schichten das Konzept von Ports (vgl. UDP und TCP) einzuführen.

Außerdem kann ein Verfahren ähnlich zu TCP verwendet werden. Dabei werden alle Pakete einer Sitzung lückenlos durchnummeriert. Die Empfänger bestätigen dann in gewissen Abständen die Paketnummer, bis zu der alle Pakete empfangen wurden, damit der Sender die gesendeten Pakete vergessen kann und das Sendefenster wieder frei wird. Bemerkt ein Empfänger ein fehlendes Paket, so fordert er es selbsttätig erneut an. Dieses Verfahren bietet ähnliche Eigenschaften wie TCP und kann sogar für Multicast verwendet werden. Für Broadcast kann das Verfahren nur bedingt verwendet werden, das Sendefenster muss aufgrund von Heuristiken verschoben werden: z. B. könnte der Sender die Daten doppelt so lange speichern, wie es bisher durchschnittlich gedauert hat, bis die Rerequest-Nachrichten eingetroffen sind.

Durch diese Verfahren kann eine API mit Datenströmen erreicht werden. So können auch Objekte serialisiert übertragen werden.

Alle für die Protokolle benutzten Zusatzinformationen werden als Metadaten gesendet. Die verschiedenen Informationen sind TLV kodiert, wobei Type ein Byte-Wert ist und Length ein Short-Wert.

Name	Type	Length	Value
PortInformation	0x08	4	SourcePort:Short, DestinationPort:Short
SequenceInformation	0x09	7	SequenceNumber:Integer, SendWindowSize:Short, Flags:Byte
AcknowledgeInformation	0x18	4	SequenceNumber:Integer
RerequestInformation	0x19	8	RerequestStart:Integer, RerequestEnd:Integer

Flags für SequenceInformation:

- 1. Bit: IsLastNumber: Verbindung endet mit dieser Nummer
- 2. Bit: IsResend: Paket wurde erneut versendet
- 3. Bit: WillWaitForAck: Sender wartet auf Acknowledge-Pakete, bevor er Daten aus dem Sende-Fenster entfernt
- 4. Bit: ResetToZero: Nummer wurde auf 0 zurückgesetzt

Wenn SequenceInformation nicht gesetzt ist, wird das TCP-artige Protokoll nicht verwendet. Sequenz-Nummern fangen immer bei 0 an und enden, wenn IsLastNumber gesetzt ist. Der Sender sendet seine Sende-Fenstergröße mit, damit die Empfänger nicht jedes Paket bestätigen müssen. Mit RerequestInformation kann der Empfänger fehlende Pakete anfordern

Durch dieses Protokoll steigt der Overhead eines Paketes um weitere 20 Bytes.

12 Fazit

In dieser Arbeit wurde eine Netzwerkstruktur – SuperChord – für eine Middleware eines verteilten Desktop-Grids entworfen, implementiert und getestet. Dabei wurde insbesondere auf die Stabilität des aufgebauten Netzwerks und die Effizienz der verwendeten Routing- und Broadcastverfahren geachtet.

Der Entwurf kombiniert erforschte und wohlbekannte Konzepte mit neu entwickelten, speziell für die Probleme der Middleware entworfenen Lösungen. Bei der Implementierung wurde sehr stark auf die Trennung der Komponenten geachtet, um die Lösung einfach verständlich und erweiterbar zu halten. Außerdem wurde der Programmcode von allen verwendeten Komponenten (bis auf NTP) selbst entwickelt, wodurch das Netzwerk frei von Lizenzansprüchen genutzt werden kann. Die Testläufe fanden im PlanetLab statt, was ein Netzwerk mit hohen Latenzen und Verbindungsabbrüchen nachbildet.

Die Tests zeigen deutlich, dass SuperChord selbst unter sehr schlechten Bedingungen stabil laufen kann. Das Netzwerk kann mit Parametern sehr flexibel konfiguriert und so auf die Erfordernisse der Anwendung angepasst werden.

Das hier entwickelte Lösung bietet eine robuste, selbstoptimierende und effiziente Struktur für Desktop Grids.

Zusätzlich wurde mit ChordNet ein neues vielversprechendes Konzept entworfen, das die Konzepte des hier entwickelten Netzwerks weiterentwickelt. ChordNet könnte noch bessere Eigenschaften besitzen als das hier entwickelte Netzwerk und damit ein Thema für weitere Arbeiten und Forschungen bieten.

Literatur

1 Anderson, David P.: BOINC: A System for Public-Resource Computing and Storage. In: *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, 2004, S. 4–10

2 Anderson, David P. ; Cobb, Jeff ; Korpela, Eric ; Lebofsky, Matt ; Werthimer, Dan: SETI@home: an experiment in public-resource computing. In: *Commun. ACM* 45 (2002), Nr. 11

3 Anderson, David P. ; Fedak, Gilles: The Computational and Storage Potential of Volunteer Computing. In: *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, 2006, S. 73–80

4 Apache-Software-Foundation: *Apache ActiveMQ*. website. `http://activemq.apache.org`

5 Chan, Ho-Leung ; Lam, Tak W. ; Wong, Prudence W. H.: Efficiency of Data Distribution in BitTorrent-Like Systems. In: *Proceedings of AAIM*, 2007, S. 378–388

6 Chun, Brent ; Culler, David ; Roscoe, Timothy ; Bavier, Andy ; Peterson, Larry ; Wawrzoniak, Mike ; Bowman, Mic: PlanetLab: an overlay testbed for broad-coverage services. In: *ACM SIGCOMM Computer Communication Review* 33 (2003), Nr. 3, S. 3–12

7 Codehaus: *XStream: A simple library to serialize objects to XML and back again.* website. `http://xstream.codehaus.org`

8 Cox, Russ ; Dabek, Frank ; Kaashoek, M. F. ; Li, Jinyang ; Morris, Robert: Practical, distributed network coordinates. In: *Computer Communication Review* 34 (2004), Nr. 1, S. 113–118

9 Culler, David E.: PlanetLab: An Open, Community-Driven Infrastructure for Experimental Planetary-Scale Services. In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003

10 Dabek, Frank ; Cox, Russ ; Kaashoek, M. F. ; Morris, Robert: Vivaldi: a decentralized network coordinate system. In: *Proceedings of SIGCOMM*, 2004, S. 15–26

11 Davidson, Jonathan ; Peters, James ; Bhatia, Manoj ; Kalidindi, Satish ; Mukherjee, Sudipto: *Voice Over IP Fundamentals*. Macmillan Technical Publishing, 2006

12 Jiang, Junjie ; Pan, Ruoyu ; Liang, Changyong ; Wang, Weinong: BiChord: An Improved Approach for Lookup Routing in Chord. In: *Proceedings of ADBIS*, 2005, S. 338–348

13 Kondo, Derrick ; Fedak, Gilles ; Cappello, Franck ; Chien, Andrew A. ; Casanova, Henri: Characterizing Resource Availability in Enterprise Desktop Grids. In: *Future Generation Computer Systems* 23 (2007), Januar, Nr. 7, S. 888–903

14 Merz, Peter ; Gorunova, Katja: Efficient broadcast in P2P grids. In: *Proceedings of CCGRID*, 2005, S. 237–242

15 Merz, Peter ; Kolter, Florian ; Priebe, Matthias: Free-Riding Prevention in Super-Peer Desktop Grids. In: *Computing in the Global Information Technology, 2008. ICCGI '08. The Third International Multi-Conference on* (2008), S. 297–302

16 Merz, Peter ; Priebe, Matthias ; Wolf, Steffen: Super-Peer Selection in Peer-to-Peer Networks Using Network Coordinates. In: *Proceedings of ICIW*, 2008, S. 385–390

17 Merz, Peter ; Ubben, Jan ; Priebe, Matthias: On the Construction of a Super-Peer Topology underneath Middleware for Distributed Computing. In: *Proceedings of CCGRID*, 2008, S. 590–595

18 Merz, Peter ; Wolf, Steffen ; Schwerdel, Dennis ; Priebe, Matthias: A Self-Organizing Super-Peer Overlay with a Chord Core for Desktop Grids. In: *Proceedings of the 3rd international workshop on self-organizing systems, Vienna, Austria*, 2008. – to appear

19 Park, KyoungSoo ; Pai, Vivek S.: CoMon: a mostly-scalable monitoring system for PlanetLab. In: *ACM SIGOPS Operating Systems Review* 40 (2006), Januar, Nr. 1, S. 65–74

20 Perkins, Charles E. ; Bhagwat, Pravin: Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for mobile computers. In: *Proceedings of SIGCOMM*, 1994, S. 234–244

21 Roscoe, Timothy: The PlanetLab Platform. In: Steinmetz, Ralf (Hrsg.) ; Wehrle, Klaus (Hrsg.): *Peer-to-Peer Systems and Applications.* Springer, 2005, S. 567–581

22 Schwerdel, Dennis: *Effizienter Broadcast in Super-Peer-Netzwerken*, TU Kaiserslautern, Projektarbeit in Informatik, 2008

23 Steinmetz, Ralf ; Wehrle, Klaus: Peer-to-Peer-Networking & -Computing - Aktuelles Schlagwort. In: *Informatik Spektrum* 27 (2004), Nr. 1, S. 51–54

24 Stoica, Ion ; Morris, Robert ; Karger, David ; Kaashoek, M. F. ; Balakrishnan, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001

25 Talia, Domenico ; Trunfio, Paolo: Toward a Synergy Between P2P and Grids. In: *IEEE Internet Computing* 7 (2003), Juli, Nr. 4, S. 94–96

26 Wang, Jing ; Yang, Shoubao ; Guo, Leitao: A Bidirectional Query Chord System Based on Latency-Sensitivity. In: *GCC*, 2006, S. 164–167

27 Yang, Beverly ; Garcia-Molina, Hector: Designing a Super-Peer Network. In: *Proceedings of the 19th International Conference on Data Engineering*, 2003, S. 49–

A Source-Code-Verzeichnis

Datei	Größe	Zeilen
src/superchord/chord/Chord.java	30K	864
src/superchord/chord/DirectMessage.java	1,4K	67
src/superchord/chord/ChordNode.java	833	47
src/superchord/chord/MessageType.java	1,9K	73
src/superchord/chord/BroadcastMessage.java	1,6K	71
src/superchord/chord/RoutedMessage.java	2,4K	94
src/superchord/chord/InvalidTargetPolicy.java	851	36
src/superchord/chord/ChordMessage.java	1,1K	52
src/superchord/chord/ChordConnector.java	2,4K	88
src/superchord/net/AbstractNetwork.java	23K	800
src/superchord/net/Message.java	251	14
src/superchord/net/Node.java	4,7K	185
src/superchord/net/PublicAddress.java	865	31
src/superchord/net/STUN.java	4,2K	122
src/superchord/test/Main.java	5,2K	144
src/superchord/test/Chat.java	11K	297
src/superchord/test/Simulation.java	3,6K	114
src/superchord/util/IdentMap.java	2,6K	125
src/superchord/util/MultiCache.java	3,3K	114
src/superchord/util/CacheMap.java	2,8K	107
src/superchord/util/NTimer.java	4,3K	148
src/superchord/util/CacheSet.java	2,8K	109
src/superchord/util/NtpClient.java	2,0K	65
src/superchord/util/NtpMessage.java	15K	466
src/superchord/PeerAddress.java	4,6K	204
src/superchord/SCNetwork.java	4,1K	157
src/superchord/NamingService.java	15K	448
src/superchord/SCListener.java	2,8K	101
src/superchord/SuperPeerList.java	1,5K	72
src/superchord/SuperPeerListService.java	7,1K	270
src/superchord/service/PingService.java	1,7K	68
src/superchord/service/StatisticsService.java	1,7K	64
src/superchord/GenericSCListener.java	970	60
src/superchord/PersistentId.java	4,3K	192
src/superchord/Vivaldi.java	12K	429
src/superchord/EdgePeerList.java	2,8K	128
src/superchord/EdgeMessageType.java	1,5K	58
src/superchord/SCNetworkImpl.java	32K	978

Datei	Größe	Zeilen
src/superchord/EdgeMessage.java	1,8K	82
src/superchord/ChordSPSA.java	13K	418
src/superchord/SCPeer.java	6,1K	292
src/superchord/AddressList.java	3,1K	155
src/superchord/log/LogEntry.java	471	27
src/superchord/log/LogType.java	454	21
src/superchord/log/SCLogger.java	1,4K	59
src/superchord/log/LoggingThread.java	3,2K	118
src/superchord/log/PingEntry.java	440	26
src/superchord/log/LogMerger.java	2,3K	79
src/superchord/log/MergedLogEntry.java	434	22
src/superchord/log/LogAnalyzer.java	11K	303
src/superchord/log/HumanReadableDecoder.java	4,3K	138
src/superchord/log/AnalyzedMerger.java	2,0K	60
src/superchord/log/DirectPingEntry.java	466	27
src/superchord/ReconnectService.java	2,1K	72
src/superchord/PersIdList.java	3,1K	154
src/superchord/extapi/EAInputStream.java	2,7K	113
src/superchord/extapi/EAOutputStream.java	1,9K	76
src/superchord/extapi/ExtAPIView.java	14K	431
src/superchord/extapi/EAMessage.java	2,3K	94
src/superchord/extapi/ReliableManager.java	3,4K	121
src/superchord/extapi/TopicMessage.java	1,1K	45
src/superchord/extapi/DataMessage.java	242	15
src/superchord/extapi/ReliableMessage.java	1,1K	47
src/superchord/extapi/QueueMessage.java	1,5K	57
src/superchord/extapi/ReliableMulticastMessage.java	633	28
src/superchord/extapi/ReliableMulticastSender.java	2,2K	83
src/superchord/extapi/ReliableMulticastReceiver.java	1,5K	56
src/superchord/ApplicationMessage.java	2,5K	104
src/superchord/MessageScope.java	5,9K	225

A.1 Statistiken

```
Total Physical Source Lines of Code (SLOC)                = 6,472
Development Effort Estimate, Person-Years (Person-Months) = 1.42 (17.05)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.61 (7.35)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 2.32
Total Estimated Cost to Develop                           = $ 191,969
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
```

B Liste der verwendeten Planetlab-Knoten

```
1  bob.cc.vt.edu
2  csplanet01.cs-ncl.net
3  cs-planetlab4.cs.surrey.sfu.ca
4  ent1.cs.nccu.edu.tw
5  freedom.informatik.rwth-aachen.de
6  gschembra4.diit.unict.it
7  its-2503-1.its.bth.se
8  node1.lbnl.nodes.planet-lab.org
9  node1.planetlab.mathcs.emory.edu
10 onelab03.inria.fr
11 pl1.csl.utoronto.ca
12 pl1.ucs.indiana.edu
13 plab202.wiai.uni-bamberg.de
14 planet02.hhi.fraunhofer.de
15 planet1.cs.rochester.edu
16 planet2.pittsburgh.intel-research.net
17 planet2.scs.stanford.edu
18 planet4.berkeley.intel-research.net
19 planetdev02.fm.intel.com
20 planetlab01.ethz.ch
21 planetlab-01.naist.jp
22 planetlab14.millennium.berkeley.edu
23 planetlab1.arizona-gigapop.net
24 planetlab1.csie.nuk.edu.tw
25 planetlab1.cslab.ece.ntua.gr
26 planetlab1.cs.umass.edu
27 planetlab1.engr.uconn.edu
28 planetlab1.fri.uni-lj.si
29 planetlab1.ics.forth.gr
30 planetlab1.informatik.uni-kl.de
31 planetlab1.nbgisp.com
32 planetlab1.uc.edu
33 planet-lab1.ufabc.edu.br
34 planetlab1.wiwi.hu-berlin.de
35 planetlab1.xeno.cl.cam.ac.uk
36 planetlab2.cesnet.cz
37 planetlab2.comp.nus.edu.sg
38 planetlab2.cs.umass.edu
39 planetlab2.cs.uoregon.edu
40 planetlab2.fct.ualg.pt
41 planetlab-2.fing.edu.uy
42 planetlab2.ics.forth.gr
43 planetlab-2.imperial.ac.uk
44 planetlab2.informatik.uni-goettingen.de
45 planetlab2.isi.jhu.edu
46 planetlab2.netlab.uky.edu
47 planetlab2.nrl.dcs.qmul.ac.uk
48 planetlab2.pc.cis.udel.edu
49 planetlab2.ucsd.edu
50 planetlab2.unl.edu
51 planetlab2.utdallas.edu
52 planetlab-3.cmcl.cs.cmu.edu
53 planetlab3.csres.utexas.edu
54 planetlab3.ie.cuhk.edu.hk
55 planetlab4.inf.ethz.ch
56 planetlab4.postel.org
57 planetlab4.wail.wisc.edu
58 plgmu2.ite.gmu.edu
59 pli1-pa-3.hpl.hp.com
60 pub2-s.ane.cmc.osaka-u.ac.jp
61 scratchy.cs.uga.edu
62 server1.planetlab.iit-tech.net
```

C Patch für ObjectOutputStream

```
1  --- ObjectOutputStream.java.old 2008-05-14 14:32:25.000000000 +0200
2  +++ ObjectOutputStream.java     2008-05-15 10:22:30.000000000 +0200
3  @@ -1183,7 +1183,7 @@
4       private void writeClass(Class cl, boolean unshared) throws IOException {
5          bout.writeByte(TC_CLASS);
6          writeClassDesc(ObjectStreamClass.lookup(cl, true), false);
7  -       handles.assign(unshared ? null : cl);
8  +       if ( ! unshared ) handles.assign(cl);
9       }
10
11      /**
12 @@ -1211,7 +1211,7 @@
13         throws IOException
14      {
15         bout.writeByte(TC_PROXYCLASSDESC);
16 -       handles.assign(unshared ? null : desc);
17 +       if ( ! unshared ) handles.assign(desc);
18
19         Class cl = desc.forClass();
20         Class[] ifaces = cl.getInterfaces();
21 @@ -1236,7 +1236,7 @@
22         throws IOException
23      {
24         bout.writeByte(TC_CLASSDESC);
25 -       handles.assign(unshared ? null : desc);
26 +       if ( ! unshared ) handles.assign(desc);
27
28         if (protocol == PROTOCOL_VERSION_1) {
29             // do not invoke class descriptor write hook with old protocol
30 @@ -1259,7 +1259,7 @@
31       * depending on string length.
32       */
33      private void writeString(String str, boolean unshared) throws IOException {
34 -       handles.assign(unshared ? null : str);
35 +       if ( ! unshared ) handles.assign(str);
36         long utflen = bout.getUTFLength(str);
37         if (utflen <= 0xFFFF) {
38             bout.writeByte(TC_STRING);
39 @@ -1280,7 +1280,7 @@
40      {
41         bout.writeByte(TC_ARRAY);
42         writeClassDesc(desc, false);
43 -       handles.assign(unshared ? null : array);
44 +       if ( ! unshared ) handles.assign(array);
45
46         Class ccl = desc.forClass().getComponentType();
47         if (ccl.isPrimitive()) {
48 @@ -1361,7 +1361,7 @@
49         bout.writeByte(TC_ENUM);
50         ObjectStreamClass sdesc = desc.getSuperDesc();
51         writeClassDesc((sdesc.forClass() == Enum.class) ? desc : sdesc, false);
52 -       handles.assign(unshared ? null : en);
53 +       if ( ! unshared ) handles.assign(en);
54         writeString(en.name(), false);
55      }
56
57 @@ -1385,7 +1385,7 @@
58
59             bout.writeByte(TC_OBJECT);
```

```
60              writeClassDesc(desc, false);
61  -           handles.assign(unshared ? null : obj);
62  +           if ( ! unshared ) handles.assign(obj);
63              if (desc.isExternalizable() && !desc.isProxy()) {
64                  writeExternalData((Externalizable) obj);
65              } else {
66
67  @@ -2205,6 +2205,8 @@
68
69          /* number of mappings in table/next available handle */
70          private int size;
71  +       /* initial capacity of the table */
72  +       private int initialCapacity;
73          /* size threshold determining when to expand hash spine */
74          private int threshold;
75          /* factor for computing size threshold */
76  @@ -2221,10 +2223,7 @@
77           */
78          HandleTable(int initialCapacity, float loadFactor) {
79              this.loadFactor = loadFactor;
80  -           spine = new int[initialCapacity];
81  -           next = new int[initialCapacity];
82  -           objs = new Object[initialCapacity];
83  -           threshold = (int) (initialCapacity * loadFactor);
84  +           this.initialCapacity = initialCapacity;
85              clear();
86          }
87  @@ -2264,6 +2263,10 @@
88           * Resets table to its initial (empty) state.
89           */
90          void clear() {
91  +           spine = new int[initialCapacity];
92  +           next = new int[initialCapacity];
93  +           objs = new Object[initialCapacity];
94  +           threshold = (int) (initialCapacity * loadFactor);
95              Arrays.fill(spine, -1);
96              Arrays.fill(objs, 0, size, null);
97              size = 0;
```