Effizienter Broadcast in Super-Peer-Netzwerken

Dennis Schwerdel

2. April 2008

Zusammenfassung

In dieser Projektarbeit wird eine, auf Chord basierende, Netzwerkstruktur erstellt und simuliert. Insbesondere wird ein Broadcast-Verfahren und ein deterministisches Gossip-Verfahren zur Verbreitung von Informationen entwickelt.

Das Chord-Netzwerk wird als Kern in ein bestehendes Netzwerk eingefügt und durch Simulation die Wechselwirkungen bestimmt.

Inhaltsverzeichnis

1 Motivation

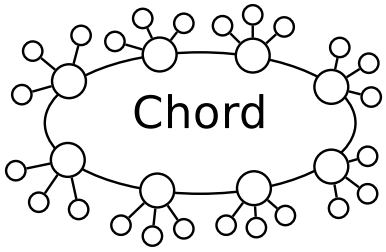
Verteiltes Rechnen ist insbesondere für algorithmisch schwer lösbare Probleme interessant, da die Rechenleistung eines einzelnen Rechners für die Berechnung nicht ausreicht. Damit nicht jede Forschungsgruppe ein eigenes Rechnernetzwerk für derartige Berechnungen betreiben muss, soll mit Hilfe einer Middleware [6] weltweit vorhandene Rechenleistung von Servern und Desktop-PCs im Internet zusammengefasst werden. Die Middleware übernimmt dabei die Verwaltung des Netzwerks und die Verteilung der Aufgaben. Die Middleware muss dazu problemneutral sein, das heißt dass beliebige Berechnungen ausgeführt werden können. Es müssen also nicht nur die Daten einer Aufgabe sondern auch der Algorithmus der Aufgabe übertragen werden.

2 Aufgabenstellung

Aufgabenstellung dieser Projektarbeit war es, für das Kern-Netzwerk der Middleware eine andere Netzwerkstruktur zu benutzen und die Veränderungen, die sich daraus ergeben zu simulieren und bewerten.

2.1 Aufteilung der Middleware

Abbildung 1:



Netzwerkstruktur der Middleware

Das Netzwerk der Middleware ist in zwei Bereiche unterteilt:

1. Das Super-Peer-Netzwerk: Dieses Netzwerk stellt den Kern des Gesamt-Netzwerks dar und bietet eine Kommunikationsinfrastruktur. Die Knoten in diesem Netzwerk werden Super-Peers oder auch Hubs genannt und übernehmen Verwaltungsaufgaben für das gesamte Netzwerk.

2. Die Edge-Peers: Jeder Edge-Peer hat eine Verbindung zu einem Super-Peer, der ihn verwaltet. Da die Edge-Peers keinen Verwaltungs- oder Kommunikationsaufwand haben, können sie ungestört Jobs abarbeiten.

Es ist nicht statisch festgelegt, welcher Knoten welche Aufgabe erfüllt (Super-Peer oder Edge-Peer). Mit dem Super-Peer-Selection-Algorithmus[8] (SPSA) werden diese Aufgaben unter den Knoten verteilt.

2.2 Der Super-Peer-Selection-Algorithmus (SPSA)

Aufgabe des SPSA ist es, zu entscheiden welcher Knoten Super-Peer ist und welcher Edge-Peer. Zusätzlich wird mit dem SPSA ermittelt welcher Edge-Peer mit welchem Super-Peer verbunden sein soll.

Der SPSA arbeitet verteilt in jedem Knoten und optimiert das Netzwerk in Bezug auf Nachrichtenlaufzeiten. Dazu benutzt er die Knoten-Koordinaten, die der Vivaldi-Algorithmus[4] ermittelt.

SPSA restrukturiert das Netzwerk in Zyklen, dabei gibt es mehrere Aktionen, die ausgeführt werden können:

- Wechsel zu einen anderen Super-Peer: Edge-Peers verbinden sich zu dem Super-Peer mit der kürzesten Entfernung.
- Aufspalten eines Super-Peers: Ein Super-Peer, der zu viele Edge-Peers verwalten muss, ernennt einen seiner Edge-Peers zum Super-Peer.
- Downgrade eines Super-Peers: Ein Super-Peer, der zu wenige Edge-Peers verwalten muss, stuft sich zu einem Edge-Peer zurück.
- Wahl des besten Super-Peers: Ein Super-Peer stuft sich zu einem Edge-Peer zurück und ernennt einen seiner Edge-Peers zum Super-Peer, wenn dadurch die mittlere Entfernung in dieser Gruppe und zu den anderen Super-Peers sinkt.

Alle diese Aktionen werden nur ausgeführt, wenn die Verbesserung eine gewisse Schranke überschreitet. Ziel des Algorithmus ist es, bei N Knoten ein Netzwerk mit $\sqrt{N}$ optimal positionierten Super-Peers und jeweils $\sqrt{N} - 1$ Edge-Peers pro Super-Peer zu erhalten.

Die Zeitspanne zwischen den Reorganisationszyklen (Icom) kann hierbei variiert werden.

2.3 Der Simulator (NetSim)

Um die Anwendung und die beteiligten Netzwerkprotokolle zu testen, wurde eine Simulator-Umgebung[8] bereitgestellt, die eine abstrakte Netzwerkschicht bietet und eine Simulation mehrerer Netzwerk-Knoten in einem Prozess ermöglicht. Mit Hilfe dieses Simulators wurde die bisherige Middleware entwickelt und getestet.

Der Simulator erlaubt es Knotenausfälle zu simulieren und den Icom-Wert zu variieren.

Die Netzwerkdaten des zu simulierenden Netzwerks kann der Simulator selbst zufällig generieren oder aus Entfernungsmatrizen auslesen. Als Entfernungsmatrizen stehen die Daten des Planet-Lab-Netzwerks [3] von 2005 zur Verfügung.

Die Kommunikation der Super-Peers ist bisher im Simulator stark abstrahiert. Nachrichten wurden einfach direkt zugestellt, was in der Realität ein vollvermaschtes Netzwerk benötigen würde. Da dies aber nicht realisierbar ist, musste ein Netzwerk der Super-Peers erstellt werden.

2.4 Annahmen über das Netzwerk

Im Folgenden wird eine abstrakte Sicht auf das Netzwerk vorausgesetzt. Das Netzwerk besteht aus Netzwerkknoten (Nodes), die jeweils eine zufällige Nummer (ID) und Verbindungsdaten (z.B. IP-Adresse und Port-Nummer) beinhalten. Knoten können über Verbindungen Nachrichten austauschen. Verbindungen werden implizit durch das Senden einer Nachricht hergestellt. Ein Verbindungsabbruch auf unterer Ebene wird zeitnah erkannt und der Anwendung mitgeteilt. Die Übermittlung der Nachrichten erfolgt asynchron und es tritt keine sichtbare Verfälschung oder Nachrichtenverdopplung auf. Ein Nachrichtenverlust tritt nur zusammen mit einem Verbindungsabbruch auf.

2.5 Implementierungsanforderungen

Alle Verfahren sind in Java implementiert, da der bereits vorhandene Netzwerksimulator auch in Java geschrieben ist und eine spätere Implementierung der Verfahren für reale Netzwerke auch in Java erfolgen soll.

Bei der Implementierung wurde darauf geachtet, die Verfahren möglichst stark zu kapseln, sodass eine Portierung auf eine Abstraktionsschicht für reale Netzwerke weniger Aufwand benötigt.

2.6 Vergleichbare Ansätze

Ein bestehendes Netzwerk zur Job-Verteilung ist die Berkeley Open Infrastructure for Network Computing [1] (BOINC) mit dem prominentesten Projekt SETI@home [2]. Im Gegensatz zu BOINC soll die hier entwickelte Middleware Peer-to-Peer-Technologien benutzen und serverlos arbeiten.

Das hier untersuchte Chord-Netzwerk ist bereits vielfach in der Literatur beschrieben [10]. Ein dem hier entwickelten Broadcastverfahren ähnliches Verfahren findet sich in [7].

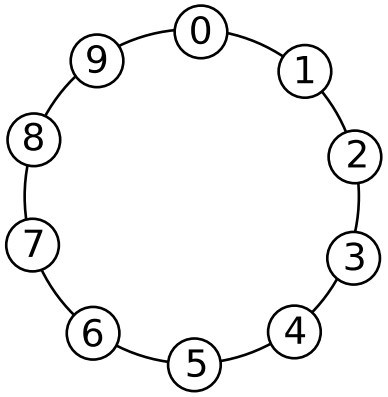
3 Chord

Chord ist ein strukturiertes P2P-Netzwerk [10]. Es bietet eine sehr robuste Struktur und effizientes Routing. Deshalb soll diese Struktur für das Super-Peer-Netzwerk der Middleware verwendet werden. Da Chord in seiner eigentlichen Form eine DHT-Struktur ist wird hier eine abgewandelte Form dieser Struktur verwendet. Insbesondere wird kein Hash-Verfahren angewendet.

3.1 Struktur eines Chord-Netzwerks

Für Chord müssen alle Knoten je eine gleich-verteilte ID in einem bestimmten Bereich $[0..N-1]$ haben. Im Allgemeinen wird N als $N = 2^d$ für ein beliebiges $d \in \mathbb{N}$ gewählt.

Abbildung 2:



Ein vollbesetzter Chord-Ring mit N=10

Jeder Knoten hält eine Menge von Verbindungen zu anderen Knoten, die Finger genannt werden. Es gibt d Finger (Finger[0] bis Finger[d-1]) die wie folgt definiert sind: Finger[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 (Finger[0]) wird auch Nachfolger (oder Successor) genannt, da er per Definition der Knoten mit der nächsthöheren ID ist. Zusätzlich zu den Fingern wird noch ein Vorgänger-Knoten (Predecessor) gespeichert, sodass der Nachfolger des Vorgängers eines Knotens dieser Knoten selbst ist.

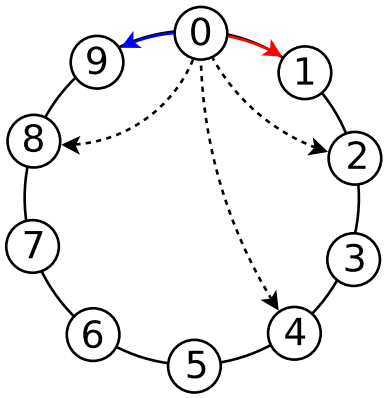
Aus Successor und Predecessor ergibt sich ein doppelt verbundener Ring, der alle Knoten beinhaltet; dieser Ring wird auch Chord-Ring genannt. Die restlichen Finger können als Abkürzungen in diesem Ring gesehen werden.

In Abbildung 3 ist ein vollbesetztes Chord-Netzwerk mit $N = 10$ dargestellt. Alle Verbindungen eines Knotens sind am Beispiel des Knotens 0 eingezeichnet. Knoten 0 hat Finger zu den Knoten 1, 2, 4 und 8. Der Finger zum Knoten 1 ist der Nachfolger (Successor). Knoten 9 ist der Vorgänger von Knoten 0.

3.2 Anzahl der Netzwerkverbindungen

Im folgenden wird von einem optimalen Netzwerk ausgegangen. D.h. wenn die Anzahl der Knoten N beträgt, gibt es $\sqrt{N}$ Super-Peers mit jeweils $\sqrt{N}-1$ Edge-Peers und alle Super-Peers haben eine ideale Positionierung der Finger. Da die Super-Peers das Chord-Netzwerk bilden, ist die Anzahl der Knoten im Chord-

Abbildung 3:



von Knoten 0 ausgehende Finger

Netzwerk $\sqrt{N}$. Weiter wird davon ausgegangen, dass die Anzahl der Finger in etwa dem Logarithmus der Anzahl der Super-Peers, also $log_2(\sqrt{N})$ entspricht.

Dann hat jeder Edge-Peer nur eine Verbindung zu seinem Super-Peer. Jeder Super-Peer hat $log_2(\sqrt{N})$ Verbindungen zu all seinen Fingern und eine zu seinem Vorgängerknoten. Des Weiteren hat jeder Super-Peer eine Verbindung zu jedem seiner$\sqrt{N} - 1$ Edge-Peers.

Ein Super-Peer hat also $\sqrt{N} + log_2(\sqrt{N})$ Verbindungen. Der überwiegende Anteil von Verbindungen stammt also von Edge-Peers und die Mehrbelastung durch das Chord-Netzwerk ist vernachlässigbar.

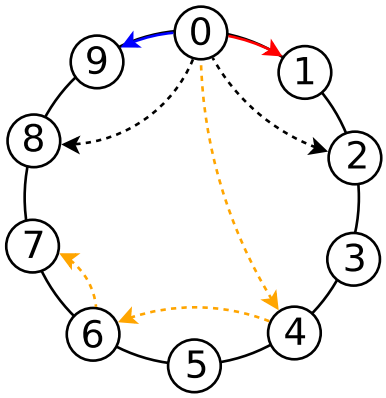
Das stellt eine erhebliche Reduktion der Verbindungen (beinahe Halbierung) im Vergleich zum voll vermaschten Netzwerk mit $2\sqrt{N} - 2$ Verbindungen pro Super-Peer dar.

4 Routing im Chord-Netzwerk

Angenommen, Knoten S will eine Nachricht an einen Knoten D senden, so wählt er den Finger F 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 (weil sogar sein Nachfolger hinter D liegt), muss er annehmen dass es keinen Knoten mit der ID von D gibt. Im schlimmsten Fall kann eine Nachricht immer über den Nachfolger weitergeleitet werden; die restlichen Finger beschleunigen also nur das Verfahren, sind aber nicht unbedingt nötig.

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

Abbildung 4:



Routing-Schritte von Knoten 0 zu Knoten 7

6 leitet nun die Nachricht an Knoten 7 weiter.

4.1 Implementierung des Routing-Verfahrens

```
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 basiert auf der Methode closestPrecedingFinger(). Diese Methode liefert immer einen Knoten, der zwischen dem Knoten selbst und dem Zielknoten liegt oder einen der beiden Knoten selbst. Es werden immer alle Finger geprüft und der beste ausgewählt. Passt kein Finger, so wird der Knoten selbst zurückgeliefert.

```
1  void routeMessage ( RoutedMsg 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 Routing selbst wählt dann mittels closestPrecedingFinger() den nächsten Knoten 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 keine Knoten zwischen einem Knoten und seinem Nachfolger existieren. Deshalb werden Nachrichten, die nicht für den Knoten selbst bestimmt sind, aber nicht mehr geroutet werden können, normalerweise verworfen. Für bestimmte Nachrichtentypen 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 wird die Nachricht zum Vorgänger geschickt. Dadurch wird das aufwändigere Routen in die Successor-Richtung vermieden.

4.1.1 Verbesserungsmöglichkeiten

1. Eine geroutete Nachricht könnte ein Reliable-Flag enthalten. Wenn das Flag gesetzt ist, wird der Absender mit einer NACK-Nachricht informiert wenn seine Nachricht nicht zum Ziel geroutet werden kann. Wenn das Flag nicht gesetzt ist, wird die Nachricht, falls sie nicht geroutet werden kann verworfen.

2. Alle Finger zwischen einem weiterleitenden Knoten und dem Zielknoten (inklusive) können theoretisch zum Routen der Nachricht verwendet werden. Der Knoten könnte andere Kriterien als die Nähe zum Zielknoten (wie z.B. Latenzzeit und Datendurchsatz) zur Auswahl eines Fingers benutzen. Dadurch kann unter Umständen die Nachrichtenübertragung verbessert werden, obwohl die Anzahl der Hops steigt.

4.2 Aufwand des Routing-Verfahrens

Im Folgenden soll von einem voll besetzten Netzwerk ausgegangen werden. D.h. alle IDs sind mit Knoten besetzt. Diese Annahme ist zwar unrealistisch aber für die Beweise notwendig. Ist das Netzwerk nicht vollbesetzt, so steigt der Aufwand der Verfahren in Bezug auf die tatsächliche Knotenzahl an, allerdings erreichen

die Verfahren dann auch eine bessere Laufzeit, sodass die hier angegebenen Komplexitätsklassen im Normalbetrieb erreicht werden können. Mit sinkender Knotenzahl steigen beispielsweise die Distanzen zwischen den Finger-IDs und den IDs der tatsächlichen Finger-Knoten. Dadurch werden kleine Routing-Schritte am Ende der Routing-Strecke vermieden, da der Zielknoten bereits früher als Finger auftritt.

Sei A die Folge der Knoten, in der eine Nachricht geroutet wird, A_0 der sendende Knoten, B der Zielknoten und d_i die ID-Distanz von A_i nach B in Richtung aufsteigender Knoten-IDs.

A_i wählt nun einen Knoten A_{i+1}unter seinen Fingern aus, der möglichst nahe an, aber nicht hinter B liegt. Diesen Knoten gibt es immer, da mindestens der Successor von A_i immer hinter A_i, aber nicht hinter B liegt. Das Verfahren terminiert bei $A_k = B$ und damit $d_k = 0$.

Wenn die Finger nicht verwendet werden können, wählt A_i immer seinen Nachfolger als A_{i+1} aus. Dann gilt bei vollbesetztem Netzwerk $d_{i+1} = d_i - 1$. Also ist $k = d_0$ und das Routingverfahren benötigt linearen Aufwand in Bezug auf die Knotenanzahl.

Können die Finger aber verwendet werden und sind optimal positioniert, so reduziert sich dieser Aufwand. Sei j so gewählt dass $2^j \leq d_i < 2^{j+1}$ (also $\frac{1}{2}d_i < 2^j$), so wählt A_i seinen j-ten Finger als A_{i+1} aus und es ergibt sich $d_{i+1} = d_i - 2^j < d_i - \frac{1}{2}d_i = \frac{1}{2}d_i$. Das heißt, d_i halbiert sich in jedem Schritt mindestens. Dadurch ergibt sich $k \leq log_2(d_0)$. Da d_0 kleiner als die maximale Knotenanzahl ist, benötigt das Routingverfahren logarithmischen Aufwand in Bezug auf die maximale Knotenanzahl.

5 Wartung des Chord-Netzwerks

5.1 Stabilisierungsverfahren

Der Stabilisierungszyklus hat zwei Aufgaben:

- Den Successor-Ring aufrechterhalten
- Die Finger optimieren

Die Stabilisierung wird periodisch ausgeführt. Die Periodendauer kann variiert werden, in der Simulationsimplementierung wird 1 Sekunde Simulationszeit verwendet. Der ideale Wert hängt von der Rate der Veränderung des Netzwerks ab.

```
1  void stabilize () {
2    sendTo ( successor , MsgType.GetPredecessor , null );
3    sendTo ( successor , MsgType.PossiblePredecessor , self );
4    checkSuccessor ( predecessor );
5    for ( Node n : finger ) checkSuccessor ( n );
6    for ( int i = 0; i < finger.length; i++ ) routeMessage ( new
         RoutedMsg ( self.id, fingerStart ( i ), MsgType.FindSuccessor
         , null ) );
7    sendTo ( predecessor , MsgType.CheckConnection , null );
8    sendTo ( successor , MsgType.CheckConnection , null );
9  }
```

In der zweiten Zeile wird der Nachfolger nach seinem Vorgänger gefragt. Dadurch soll der Nachfolger aktualisiert werden, falls ein Knoten zwischen dem Nachfolger und dem Knoten selbst liegt. In der dritten Zeile bietet sich der Knoten seinem Nachfolger als Vorgänger an. 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 Zeile 6 wird für jede Finger-Position der passende Knoten mit einer gerouteten Nachricht gesucht.

Zeilen 7 und 8 senden eine Dummy-Nachricht an den Vorgänger und den Nachfolger. Es wird davon ausgegangen, dass tiefere Protokollschichten Verbindungsprobleme beim Versenden dieser Nachrichten erkennen können.

Das Verfahren kann an zwei Stellen optimiert werden:

- Um Nachrichten zu sparen, kann die Anzahl der zu aktualisierenden Finger pro Durchlauf begrenzt werden. Es könnte z.B. pro Durchlauf nur ein Finger aktualisiert werden. Mit einem Round-Robin-Verfahren wird der nächste Finger ausgewählt.

- Die Dummy-Nachrichten zum Erkennen von Verbindungsverlusten können durch Heartbeat-Verfahren auf niederer Ebene ersetzt werden.

5.2 Verbindungsverlust

Im Folgenden wird davon ausgegangen, dass ein Verbindungsverlust von unteren Schichten zeitnah erkannt wird und dann die folgende Methode aufgerufen wird.

```
1  void connectionLost ( Node peer ) {
2    boolean mustUpdatePredecessor = false ;
3    boolean[] mustUpdateFinger = new boolean[finger.length] ;
4    Arrays.fill ( mustUpdateFinger, false ) ;
5    if ( predecessor == peer ) {
6      for ( Node f: finger ) if ( f != peer ) setPredecessor ( f ) ;
7      if ( predecessor != peer ) mustUpdatePredecessor = true ;
8      else setPredecessor ( self ) ;
9    }
10   Node lastFinger = self ;
11   for ( int i = finger.length -1; i >= 0; i-- ) {
12     if ( finger[i] == peer ) {
13       setFinger ( i, lastFinger ) ;
14       mustUpdateFinger[i] = true ;
15     } else lastFinger = finger[i] ;
16   }
17   if ( mustUpdatePredecessor ) sendTo ( predecessor, new RoutedMsg
         ( self.id, getPreviousId (self.id), MsgType.FindPredecessor,
         null ) );
18   for ( int i = 0; i < finger.length; i++ ) if ( mustUpdateFinger[i
         ] ) routeMessage ( new RoutedMsg ( self.id, fingerStart ( i )
         , MsgType.FindSuccessor, null ) );
19   checkAllFinger ( predecessor ) ;
20 }
```

Das Verfahren arbeitet in zwei Schritten. Im ersten Schritt werden alle Referenzen auf den defekten Knoten entfernt und durch den am besten passenden verbleibenden Knoten ersetzt. Im zweiten Schritt werden alle betroffenen Referenzen aktualisiert. Diese Aufteilung ist wichtig, da für das Versenden von Nachrichten keine defekten Verbindungen existieren dürfen.

In Zeile 6 wird für einen verlorenen Vorgänger ein Ersatz unter den Fingern gesucht. Hierbei wird der höchste nicht betroffene Finger verwendet. Wenn alle Finger betroffen sind, wird der Knoten selbst als sein Vorgänger eingesetzt (Zeile 8). In diesem Fall wird auch mustUpdatePredecessor nicht gesetzt da keine Finger zum Routen einer Suche verfügbar sind.

In Zeile 10 bis 16 werden für betroffene Finger neue Knoten gesucht. Damit die Finger-Bedingung nicht verletzt wird, werden Finger nur durch höhere Finger (oder den Knoten selbst) ersetzt.

Danach werden FindPredecessor/FindSuccessor-Nachrichten versendet, um die betroffenen Positionen wieder optimal zu besetzen.

5.3 Verbindungsaufbau

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

1. $F = N$: 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. Der Knoten sollte dann eine neue ID zufällig ermitteln und das Verfahren erneut starten. Dieser Fall ist in der Simulationsumgebung bisher nicht betrachtet, da er sehr unwahrscheinlich ist.

2. $K \neq F$: Der Zwischenknoten K hat einen Finger F 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 informieren und die Verbindung beenden. N wendet sich dann mit der FindJoinNode-Suche nach seinem Vorgänger an F.

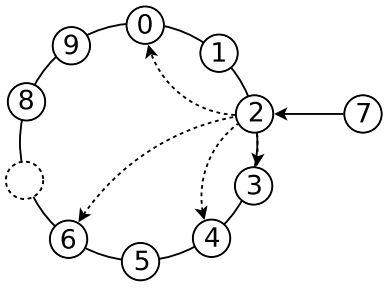
3. $K = F$: Die Methode closestPrecedingFinger() liefert K selbst. Das kann nur passieren wenn N zwischen K und seinem Nachfolger S liegt. Das wiederum bedeutet, dass der zukünftige Vorgänger von N gefunden ist. Nun wird N zwischen K und S eingefügt:

   - 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.

Abbildung 5:

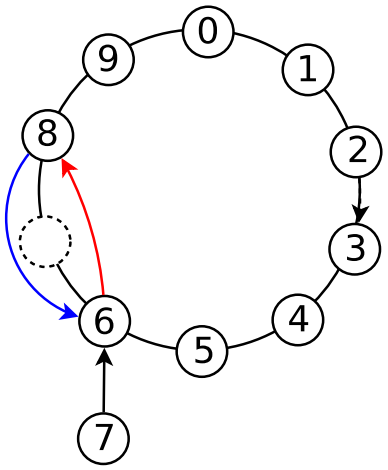


Verbindungsaufbau Schritt 1

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.

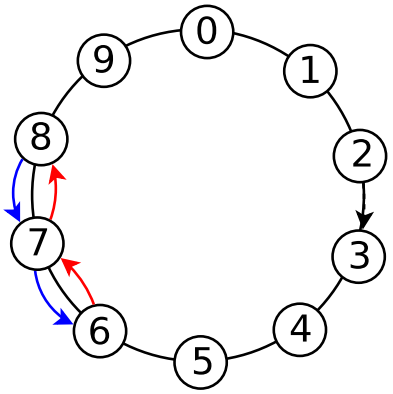
Abbildung 6:



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 7:



Verbindungsaufbau Schritt 3

Knoten 7 ist nun in das Netzwerk aufgenommen und beginnt die Suche nach seinen Fingern.

5.3.1 Verbesserungsmöglichkeiten

Ein Knoten sollte mehrere Bootstrapping-Peers kennen und das Verfahren parallel mit allen ausführen. Treten bei den empfangenen JoinHere-Nachrichten Unterschiede in den angegebenen Vorgänger- bzw- Nachfolgerknoten auf, so deutet dies auf eine Partition des Chord-Rings hin. Der neue Knoten kann dann die Netzwerke vereinen.

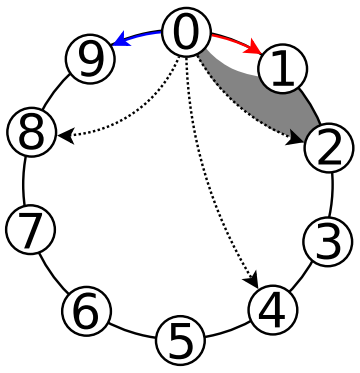
6 Broadcast im Chord-Netzwerk

Das entwickelte Verfahren erlaubt es, eine Nachricht an einen Bereich von IDs zu senden. Da der Zahlenbereich 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 normalen Nachricht (Sender und Payload) und hat zusätzlich noch eine Bereichsangabe (start und end). Wenn ein Knoten nun einen Broadcast empfängt, spaltet er den Broadcast-Bereich an den IDs seiner Finger auf und leitet den Broadcast an seine Finger mit diesen Teil-Bereichen weiter. Wenn ein Broadcast-Bereich nur noch einen Knoten beinhaltet, wird er nicht weiter aufgespalten. Das Verfahren terminiert beim Empfänger, wenn der Broadcast-Bereich keinen anderen Knoten als ihn selbst beinhaltet.

Im Beispiel will Knoten 0 einen Broadcast senden. Er hat die Finger 1, 2, 4 und 8. Er setzt den Broadcast-Bereich auf [1..9], also alle anderen Knoten.

Abbildung 8:

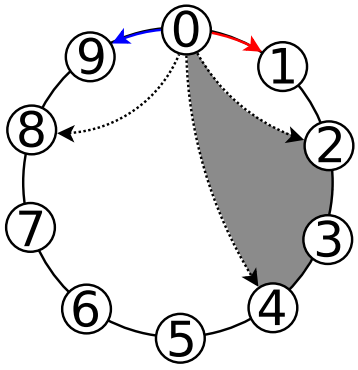


1. Broadcast-Teil-Bereich

Den Bereich zwischen ihm und seinem Nachfolger kann er beim Broadcast ignorieren, da keine Knoten in diesem Bereich existieren sollen. Sein erster Finger ist der Knoten 1, was schon im Broadcast-Bereich liegt. Also beginnt

der erste Teilbereich bei 1. Der zweite Finger 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].

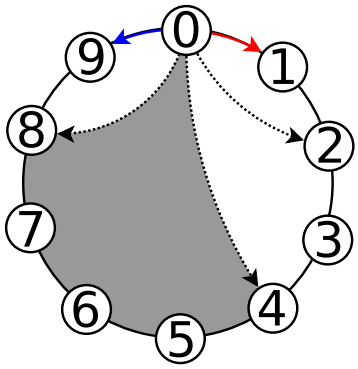
Abbildung 9:



2. Broadcast-Teil-Bereich

Es wurde bereits ermittelt, dass Knoten 2 der Beginn des nächsten Teil-Bereichs ist. Der nächste Finger nach Knoten 2 ist Knoten 4. Dieser befindet sich noch innerhalb des Broadcast-Bereichs, daher ist das Ende des zweiten Teil-Bereichs 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.

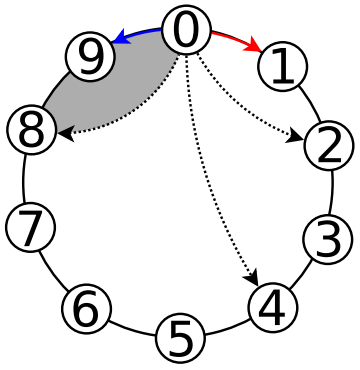
Abbildung 10:



3. Broadcast-Teil-Bereich

Knoten 4 ist bereits als Beginn des dritten Teil-Bereichs bekannt. Der nächste Finger, Knoten 8, ist noch im Broadcast-Bereich und wird der nächste Teil-Bereichs-Trenner. Die nächste Broadcast-Nachricht geht an Knoten 4 mit dem Bereich [4..7] raus.

Abbildung 11:



4. Broadcast-Teil-Bereich

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

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

Das Verfahren zum Weiterleiten eines Broadcasts ist mit dem zum Initiieren eines Broadcasts identisch, bis auf den Unterschied, dass der Broadcast-Bereich nicht selbst ermittelt wird, sondern gegeben ist.

Die jeweiligen Finger führen dieses Verfahren, mit den ihnen anvertrauten Teilbereichen, erneut durch, bis alle Bereiche nur noch einen Knoten beinhalten.

6.1 Implementierung des Broadcasts

```
1  forwardBroadcast ( long fromId, long startId, long endId, MsgType
       type, Object 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 BroadcastMsg (
           fromId, startId, lastId, type, payload ) ) ;
12     else break;
13     startId = getNextId ( lastId );
14   }
15 }
```

Das Broadcast-Verfahren sendet Nachrichten an einen Bereich von Knoten. Beim Weiterleiten einer solchen Nachricht wird der Bereich in mehrere Bereiche aufgespalten. Die Bereiche 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.

In der Schleife werden die Bereiche zwischen den Fingern ermittelt und neue Broadcast-Nachrichten mit diesen Bereichen an die Finger versendet.

Als Optimierung könnte man das Ende auf den Vorgänger vorverlegen, falls es zwischen 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 manche Knoten, die mit gerouteten Nachrichten erreicht werden können, mit einem Broadcast nicht erreicht. Dies tritt dann auf, wenn ein neuer Knoten Nachfolger seines Vorgängers ist aber noch nicht Vorgänger seines Nachfolgers.

6.2 Aufwand des Broadcast-Verfahrens

Die Nachrichtenlaufzeit des Broadcast-Verfahrens unterscheidet sich nicht von der des Routing-Verfahrens. Der ID-Bereich des Broadcasts wird immer so aufgespalten, dass für jede ID der Broadcast so weitergeleitet wird als wäre er eine normal geroutete Nachricht.

Im Gegensatz zum Routing unterscheidet sich beim Broadcast aber die Nachrichtenanzahl von der Anzahl an Weiterleitungen, da hier Nachrichten aufgespalten werden. Wenn die maximale Knoten-ID n beträgt, ist die Anzahl der Nachrichten $n - 1$, also $O(n)$. Dies ist offensichtlich, wenn man bedenkt dass durch das Verfahren jeder Knoten die Nachricht genau einmal bekommt.

7 Deterministisches Gossip

Um die Super-Peer-Liste zu verbreiten, kann eine deterministische Variante des Gossip-Verfahrens [5] verwendet werden. Super-Peers kommunizieren periodisch mit anderen Super-Peers (Partner) und tauschen Existenz-Informationen über andere Super-Peers aus. Bei normalem Gossip basiert die Auswahl eines Partners auf dem Zufall. Die so erreichte Verbreitungsgeschwindigkeit ist in der Regel sehr gut. Das deterministische Gossip nutzt die Chord-Struktur aus und wählt die Finger nacheinander (Round-Robin) als Partner aus. Dadurch kann eine Verbreitungsgeschwindigkeit, die dem Best-Case von normalem Gossip entspricht, garantiert werden.

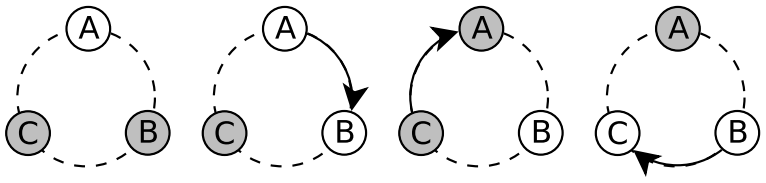
7.1 Bisheriger Ansatz

Im ersten Ansatz wurden zwischen den Super-Peers Informationen über Einfügungen und Löschungen von anderen Super-Peers ausgetauscht. Dabei überschreibt eine erhaltene Information immer die gespeicherte Information, außer die Information betrifft den Knoten selbst.

Durch die Sendereihenfolge über die Finger und die Nachrichtenlaufzeiten entstehen nun Sendezyklen, in denen die Informationen im Kreis laufen. Solange der Knoten, den die Information betrifft, im Zyklus enthalten ist, konvergiert der Zyklus gegen die korrekte Information. Wenn dies nicht der Fall ist, kann es unter Umständen passieren dass der Zyklus nicht konvergiert und fehlerhafte Informationen erhalten bleiben.

Dies kann bereits bei einem Zyklus aus drei Knoten auftreten, wenn immer der Knoten, der die seltenere Information besitzt, als nächster seine Information sendet. Dann haben immer zwei Knoten die gleiche Information und der dritte eine andere. Der Zyklus konvergiert dann nicht.

Abbildung 12:



Beispiel eines nicht konvergierenden Zyklus

In Simulationen zeigte sich, dass ein Super-Peer nur etwa zwei Drittel der anderen Super-Peers bekannt war. Über die Hälfte der Einträge der Super-Peer-Liste waren tatsächlich keine Super-Peers, sondern Edge-Peers (False-Positives). Die hohe Zahl der False-Positives erklärt sich daher, dass der betroffene Knoten in keinem Sendezyklus enthalten sein kann, da er kein Super-Peer ist.

7.2 Verfahren mit Sequenznummern

Um die Fehler des ersten Verfahrens zu verhindern und eine garantierte Konvergenz zu erreichen, wurden bei dem im Rahmen dieser Projektarbeit entwickelten Verfahren Sequenznummern verwendet. Das Sequenznummernverfahren ähnelt dem DSDV-Verfahren [9]. Die Sequenznummern steigen monoton an und eine höhere Sequenznummer gilt als neuer als eine niedrigere. Ein Knoten ist (laut Gossip) genau dann ein Super-Peer, wenn seine Sequenznummer geradzahlig ist. Jeder Super-Peer verbreitet eine Information über sich mit periodisch wachsenden geradzahligen Sequenznummern. Ein anderer Super-Peer, der die direkte Verbindung zu diesem Knoten verliert oder ihn als seinen Peer-Knoten kennt, kann die letzte bekannte geradzahlige Sequenznummer um 1 erhöhen und verbreiten. Die Gossip-Informationen enthalten alle nötigen Informationen über den Knoten.

7.3 Implementierung des Deterministic Gossip

Das deterministische Gossip ist in der Klasse Gossip realisiert. Die Klasse hat 2 Unterklassen UpdateEntry und UpdateSet. UpdateEntry kombiniert Informationen über einen Knoten (ID, Netzwerkadresse, Vivaldi-Koordinaten) mit der Sequenznummer dieser Information. UpdateSet ist eine Menge dieser Informationen, die aus Effizienzgründen als Map von Knoten auf Sequenznummern implementiert ist.

Gossip verfügt über alle Methoden, um die Hubliste selbst zu pflegen und eingehende Updates zu verarbeiten. Das Verarbeiten eines UpdateSets erzeugt ein neues UpdateSet mit relevanten Änderungen, das an Edge-Peers weitergeleitet werden kann.

Beim Verarbeiten eines Update-Sets werden alle enthaltenen Informationen mit den bereits bekannten Informationen verglichen. Wird dabei eine neue Information oder eine Information mit höherer Sequenznummer gefunden, wird diese in die Hubliste und das eigene UpdateSet eingetragen (ggfs. werden alte Informationen überschrieben) und ein entsprechender Eintrag wird der Ausgabe hinzugefügt. Dadurch enthält die Ausgabe alle Updates, die nötig sind, um die Hublisten der Edge-Peers zu aktualisieren.

Das Gossip-Update ist als Teil des Reorganisationszyklusses des Super-Peer Selection Algorithmus (SPSA) implementiert. In jedem Zyklus sendet ein Super-Peer sein UpdateSet an einen per Round-Robin ermittelten Finger. Dieser antwortet dann mit seinem UpdateSet. Immer wenn ein UpdateSet empfangen wird, werden die relevanten Änderungen an die Edge-Peers weitergegeben.

Wenn ein Knoten das Netzwerk betritt oder den Super-Peer wechselt bekommt er eine vollständige Hubliste gesendet.

7.3.1 Verbesserungsmöglichkeiten

In realen Implementierungen kann der Kommunikationsaufwand durch folgende Maßnahmen reduziert werden:

1. Bisher werden Informationen aus dem UpdateSet nicht mehr entfernt, was dazu führt, dass das UpdateSet größer wird, als die Hubliste selbst. Wenn eine Information bereits an alle Finger einmal gesendet wurde (Counter), muss sie nicht mehr versendet werden und kann aus dem UpdateSet entfernt werden.

2. Ein Knoten erzeugt nicht periodisch neue Sequenznummern, sondern nur, wenn er von seiner angeblichen Nicht-Existenz erfährt (er also eine höhere Sequenznummer als die zuletzt von ihm erzeugte sieht). Dadurch sinkt die Größe der versendeten UpdateSets und die Anzahl der UpdateSets, die an Edge-Peers versendet werden müssen, vorausgesetzt, das Netzwerk ist ausreichend stabil.

7.4 Konvergenz des deterministischen Gossip

Um zu zeigen, dass sich die korrekten Informationen im Super-Peer-Netzwerk durchsetzen, wird zunächst eine obere Schranke für die Ausbreitung von Informationen gezeigt.

7.4.1 Obere Schranke für die Ausbreitungsdauer

Sei t_p die Dauer des periodischen Intervalls, in dem Gossip ausgeführt wird, und $t_m$die Nachrichtenlaufzeit.

Sei d der Durchmesser des Netzwerks, also die maximale Anzahl der Hops bis zu einem beliebigen Knoten (auf kürzestem Weg).

Sei f die Anzahl der Finger jedes Knotens eines Netzwerks.

Im schlechtesten Fall wird der Finger, der zu einem Knoten k führt, im Round-Robin-Verfahren als letztes abgearbeitet. Es vergehen also f Perioden der Länge $t_p$und eine Nachrichtenlaufzeit t_m. Wenn dies nun bei jedem der d Hops der Fall ist, ergibt sich als Gesamtlaufzeit $d * (f * t_p + t_m)$. Wenn man nun die Netzwerkgröße n betrachtet, so gilt bei Chord $d = log(n)$ und $f = log(n)$ und somit $O(log(n)^2)$ als Komplexitätsklasse.

7.4.2 Beweis der Konvergenz

Da höhere Sequenznummern sich gegenüber niedrigeren Sequenznummern durchsetzen, breitet sich die jeweils neuere Information im Netzwerk aus. Ein existierender Super-Peer erhöht periodisch seine Sequenznummer und sorgt dadurch dafür, dass die Information über seine Existenz sich im Netzwerk durchsetzt. Wenn ein Super-Peer das Netzwerk verlässt, so gibt es 2 Fälle:

1. Der Knoten wird neuer Edge-Peer eines Super-Peers. Dadurch verbreitet der Edge-Peer die Information über das Fehlen dieses Knotens im Netz, indem er die Sequenznummer um 1 erhöht.

2. Der Knoten verlässt das Netzwerk vollständig. Wenn er die Verbindungen kontrolliert beendet, kann er sich mit einer ungeraden Sequenznummer vorher selbst entfernen. Andernfalls bricht er seine Verbindungen ab und Knoten, die ihn als Finger hatten, bemerken sein Fehlen und können das Netz darüber informieren, indem sie die Sequenznummer um 1 erhöhen.

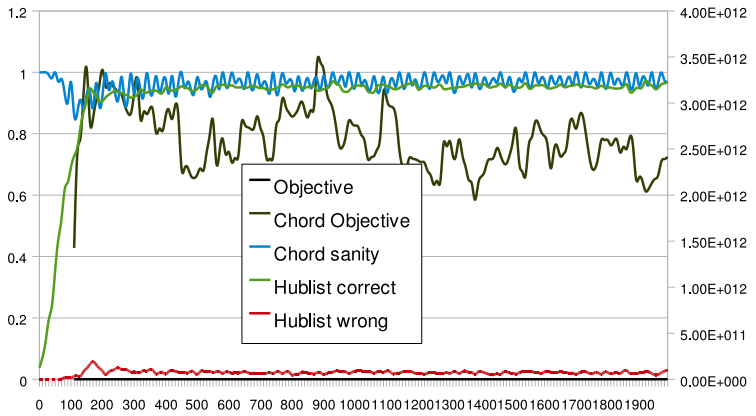
In beiden Fällen nimmt der Knoten selbst nicht mehr am Gossip teil und kann so die um 1 erhöhte Sequenznummer nicht mehr überschreiben.

8 Simulation

Für die Simulation wurden folgende Parameter verwendet:

- Durchschnittliche Lebenszeit eines Knotens: 200.000 Sekunden
- Simulationszeit: 2000 Sekunden in Schritten zu 1 Sekunde
- Ausfallwahrscheinlichkeit eines Knotens: 5%
- Reorganisationszyklus des SPSA (Icom): alle 16 Sekunden
- Netzwerkdaten: PlanetLab, September 2005

Abbildung 13:



Simulation des Netzwerks

Die Chord sanity berechnet sich aus dem durchschnittlichen Verhältnis von für andere Super-Peers sichtbare Super-Peers zu tatsächlich existierenden Super-Peers und ist ein gutes Maß für die Verbundenheit des Netzwerks. Der Wert pendelt immer zwischen etwa 95% und 100%. Hieran kann man sehen, dass das Chord-Netzwerk immer stabil und zusammenhängend bleibt. Ein Zerfallen oder Partitionieren des Netzwerks würde sich in deutlich kleineren Werten äußern. Die Schwankungen dieses Wertes ergeben sich aus den Reorganisationszyklen des Super-Peer-Selection-Algorithmus. Die Stabilisierung des Chord-Netzwerks sorgt dafür, dass der Wert immer wieder auf 100% steigt.

Die Chord Objective berechnet die Übertragungsverzögerungen unter Berücksichtigung des Routing-Verfahrens im Chord-Ring. Die starken Schwankungen ergeben sich wieder durch die Reorganisationszyklen des SPSA. Die niedrigen Werte und der steile Anstieg am Anfang erklären sich daraus, dass das Netzwerk am Anfang aus nur wenigen Super-Peers besteht und daher sehr kurze Übertragungszeiten erreicht.

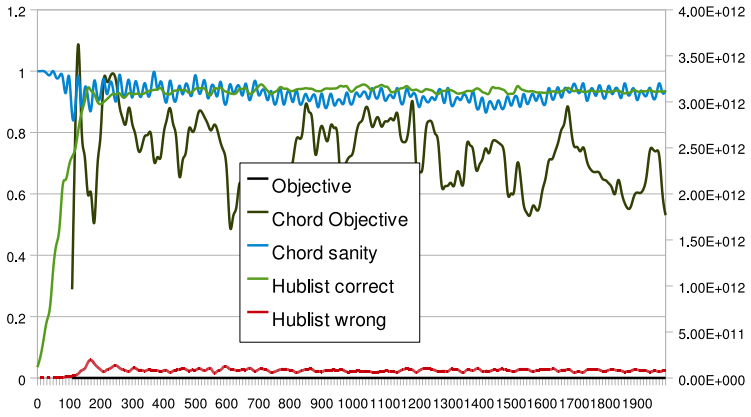
Die korrekten/falschen Hublist-Einträge berechnen sich aus dem durchschnittlichen Verhältnis von korrekten/falschen Hublist-Einträgen zu existierenden Super-Peers. Anhand dieser Graphen kann man sehen, dass das Deterministische Gossip-Verfahren effizient die Hublisten aktualisiert. Die grüne Linie steigt sehr schnell auf einen Wert von 90% an, was auf eine schnelle Konvergenz des Verfahrens hindeutet. Die grünen und roten Linien bleiben im folgenden Verlauf immer in der Nähe von 100% bzw. 0% mit einer maximalen Abweichung von etwa 5%. Das bedeutet, dass das Gossip-Verfahren mit den ständigen Änderungen des SPSA gut umgehen kann.

8.1 Simulation mit extremer Ausfallwahrscheinlichkeit

Für diese Simulation wurden folgende Einstellungen verwendet:

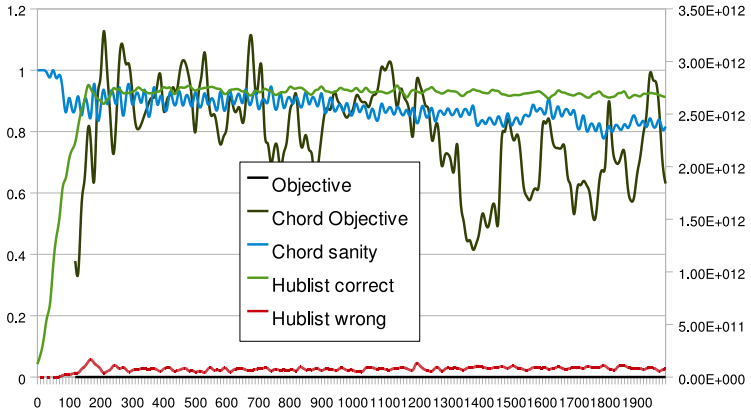
- Durchschnittliche Lebenszeit eines Knotens: 10.000 Sekunden
- Simulationszeit: 2000 Sekunden in Schritten zu 1 Sekunde
- Ausfallwahrscheinlichkeit eines Knotens: 10%, 15% und 20%
- Reorganisationszyklus des SPSA (Icom): alle 16 Sekunden
- Netzwerkdaten: PlanetLab, September 2005

Abbildung 14:



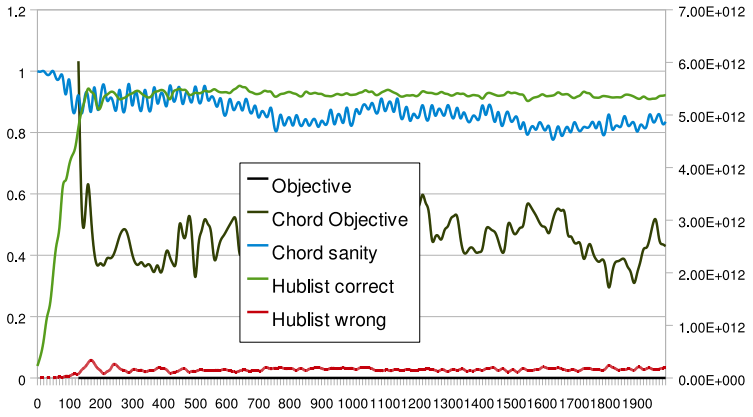
Simulation mit 10% Ausfallwahrscheinlichkeit

Abbildung 15:



Simulation mit 15% Ausfallwahrscheinlichkeit

Abbildung 16:



Simulation mit 20% Ausfallwahrscheinlichkeit

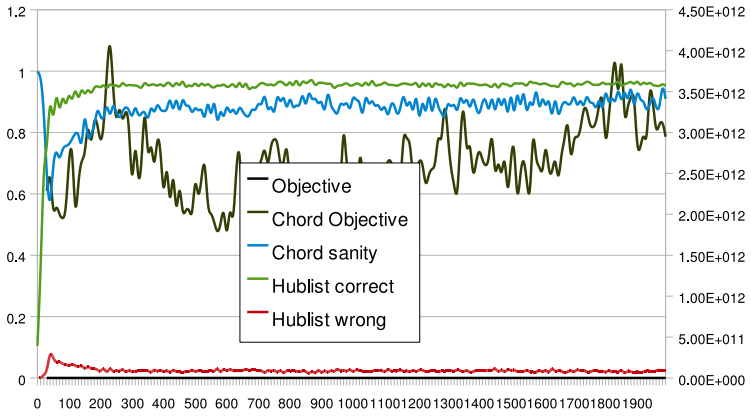
Die Ergebnisse zeigen, dass das Chord-Netzwerk selbst bei unrealistisch hohen Knotenausfällen seine Stabilität nicht dauerhaft verliert. Der Wert der Chord-Sanity liegt in allen Simulationen deutlich unter 100%, pendelt sich aber darunter auf einen Wert ein, was von der Selbstreparaturfähigkeit zeugt.

8.2 Variierung des Icom-Intervalls

Für die Simulation wurden folgende Parameter verwendet:

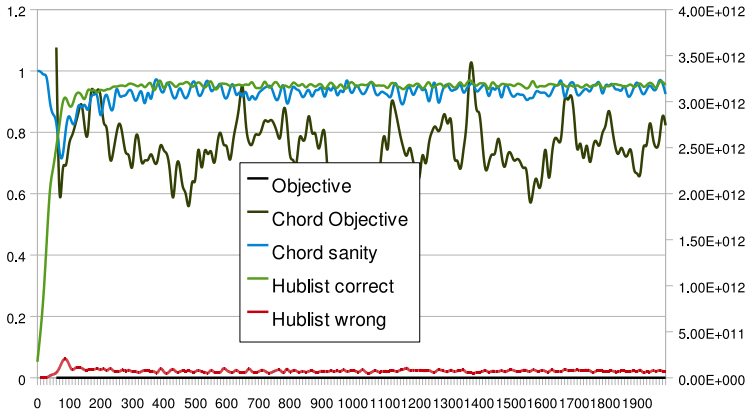
- Durchschnittliche Lebenszeit eines Knotens: 200.000 Sekunden
- Simulationszeit: 2000 Sekunden in Schritten zu 1 Sekunde
- Ausfallwahrscheinlichkeit eines Knotens: 5%
- Netzwerkdaten: PlanetLab, September 2005

Abbildung 17:



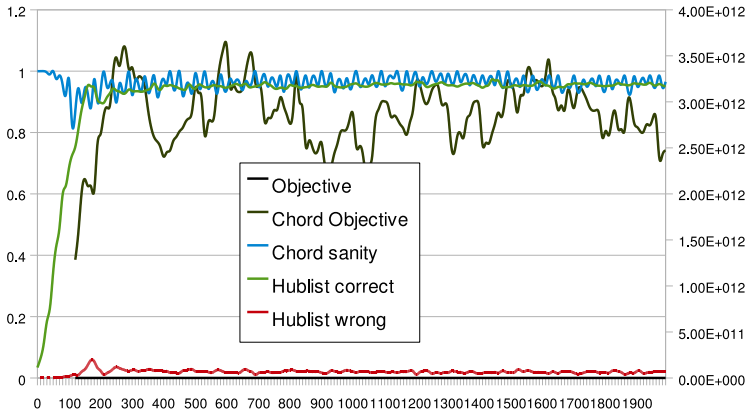
Simulation des Netzwerks mit Icom-Intervall 4 Sekunden

Abbildung 18:



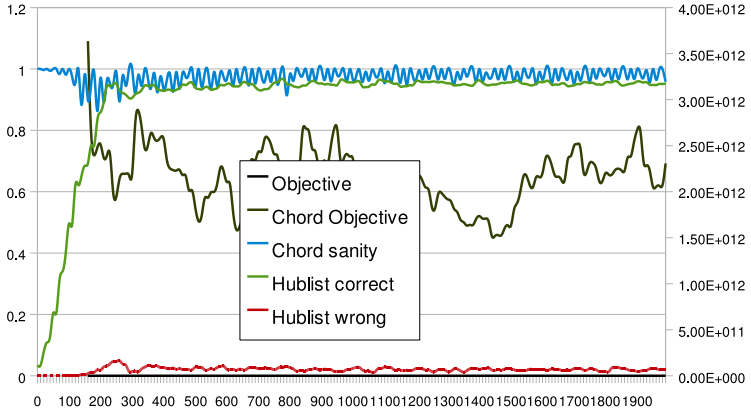
Simulation des Netzwerks mit Icom-Intervall 8 Sekunden

Abbildung 19:



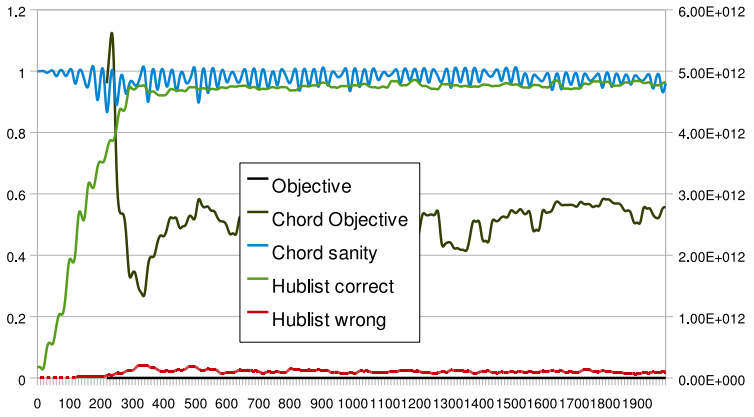
Simulation des Netzwerks mit Icom-Intervall 16 Sekunden

Abbildung 20:



Simulation des Netzwerks mit Icom-Intervall 24 Sekunden

Abbildung 21:



Simulation des Netzwerks mit Icom-Intervall 32 Sekunden

Eine Veränderung des Icom-Intervalls zeigt, dass ein Wert von etwa 16 Sekunden für das simulierte Netzwerk ein Optimum darstellt. Kleinere Werte sorgen im Chord-Ring zu stärken Schwankungen und verringern die Chord-Sanity. Größere Werte verlangsamen die Konvergenz des SPSA und des Gossip-Verfahrens ohne merklichen Einfluss auf die Chord-Stabilität.

Eine genaue Ermittlung des Optimums ist nicht zweckmäßig, da das Verhalten des Netzwerks sehr stark von den Entfernungen der einzelnen Knoten abhängt und reale Netze sich hier sehr stark unterscheiden.

Am Graphen mit dem Icom-Intervall 32 kann man die einzelnen Zyklen des Gossips anhand des Treppeneffekts erkennen.

9 Fazit

Durch die Einführung des Chord-Netzwerks als Kommunikationsinfrastruktur verliert der Super-Peer-Selection-Algorithmus zum Teil die Fähigkeit, die Nachrichtenlaufzeiten zu optimieren.

Das Chord-Netzwerk ist eine effiziente und robuste P2P-Netzwerk-Struktur. Die Fluktuationen, die durch den Super-Peer-Selection-Algorithmus entstehen, kann das Chord-Netzwerk gut verkraften und sich schnell von Ausfällen erholen.

Das Chord-Netzwerk bietet sehr effizientes Routing und Broadcasting. Darüber hinaus teilen die Finger den ID-Bereich so auf, dass effiziente Algorithmen wie z.B. das Deterministische Gossip Verfahren einfach realisiert werden können. Die Verbindungen des Chord-Netzwerks sind gegenüber den Super-Peer/Edge-Peer-Verbindungen vernachlässigbar und die Gesamtzahl der Verbindungen eines Super-Peers verringert sich beinahe um die Hälfte.

Das Deterministische Gossip-Verfahren verbreitet Informationen auf schnelle und effiziente Weise im gesamten Netzwerk.

10 Ausblick

Die Verfahren wurden so implementiert, dass eine Portierung auf ein reales Netzwerk mit einer geeigneten Abstraktionsschicht leicht möglich ist.

Das Broadcast-Verfahren ermöglicht eine Erweiterung auf limitierte Broadcasts. Das heißt, ein Broadcast kann einen beliebigen Bereich an Knoten-IDs abdecken.

In Zukunft wird die hier entwickelte und simulierte Netzwerkstruktur in einem realen Netzwerk getestet. Hierfür bietet sich das PlanetLab an.

Literatur

[1] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, 2004.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[3] David E. Culler. Planetlab: An open, community-driven infrastructure for experimental planetary-scale services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.

[4] Frank Dabek, Russ Cox, M. Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *Proceedings of SIGCOMM*, pages 15–26, 2004.

[5] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, pages 1–12, 1987.

[6] Thomas Fischer, Stephan Fudeus, and Peter Merz. A middleware for job distribution in peer-to-peer networks. In *Proceedings of PARA*, pages 1147–1157, 2006.

[7] Peter Merz and Katja Gorunova. Efficient broadcast in p2p grids. In *Proceedings of CCGRID*, pages 237–242, 2005.

[8] Peter Merz, Matthias Priebe, and Steffen Wolf. A simulation framework for distributed super-peer topology construction using network coordinates. In *Proceedings of PDP*, pages 491–498, 2008.

[9] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *Proceedings of SIGCOMM*, pages 234–244, 1994.

[10] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 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*, pages 149–160, New York, NY, USA, 2001. ACM.

Anhang A: Programmcode Chord

Anhang B: Programmcode Gossip

Anhang C: Simulationsergebnisse