Universität Karlsruhe (TH)

Implementierung und Evaluierung eines hybriden Overlays auf Basis von CAN und Chord

Studienarbeit am Institut für Telematik
Prof. Dr. M. Zitterbart
Fakultät für Informatik
Universität Karlsruhe (TH)

von

cand. Wi.-Ing.
Katrin Tomanek

Betreuer:
Prof. Dr. M. Zitterbart
Dr. T. Fuhrmann

Tag der Anmeldung:	16. August 2004
Tag der Abgabe:	15. November 2004

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 15. November 2004

Inhaltsverzeichnis

1. Einleitung

Peer-to-Peer-Systeme sind mittlerweile Gegenstand vieler Forschungsprojekte, seit 2001 veranstaltet die IEEE jährlich eine Konferenz[1], die sich ausschließlich diesem Themengebiet widmet.

Selbstorganisierende Peer-to-Peer Overlay-Netze stellen eine geeignete Basis für viele verteilte Anwendungen dar. Skalierbarkeit und Fehlertoleranz sind wichtige Eigenschaften dieser Systeme.

Eine entscheidende Frage in diesem Zusammenhang ist aber auch, wie gut sich diese Systeme an die Topologie des darunterliegenden Netzes anpassen. Um effizientes Routing zu ermöglichen, sollte die durch das Overlay-Netz zusätzlich entstehende Latenz möglichst gering gehalten werden. Außerdem sollten die Gegebenheiten des Underlays berücksichtigt werden.

1.1 Zielsetzung der Arbeit

Zielsetzung der Studienarbeit ist die Implementierung und Evaluierung eines hybriden Overlay-Netzes, welches bei der Wahl der Nachbarn die Latenzeigenschaften des darunterliegenden Netzes miteinbezieht. Dies wird durch Modifikation und Kombination der Peer-to-Peer-Systeme „Chord“ und „CAN“ ermöglicht. Eine genaue Beschreibung und Spezifikation des zugrundeliegenden Konzepts ist ebenso Teil dieser Arbeit.

1.2 Gliederung der Arbeit

Im zweiten Kapitel werden die Grundlagen erläutert, die zum Verständnis dieser Studienarbeit notwendig sind. Peer-to-Peer-Systeme und Overlay-Netze, sowie zwei bedeutende Vertreter dieser Systeme werden ausführlich erläutert. Anschließend werden Ansätze zur Performanzverbesserung dieser Systeme und Algorithmen zur Bestimmung synthetischer Koordinaten vorgestellt.

[1] IEEE International Conference on Peer-to-Peer Computing

Ein hybrides Overlay-Netz, welches auf Chord und CAN basiert, wird im dritten Kapitel spezifiziert und die an Chord und CAN notwendigen Modifikationen werden erläutert.

Das vierte Kapitel geht auf die Implementierung des vorgestellten Systems ein. Ein Schwerpunkt wird dabei auf die Implementierung eines Mechanismus zur Latenzmessung und Bestimmung synthetischer Koordinaten gelegt.

Dieser Mechanismus wird im fünften Kapitel einer Evaluierung unterzogen.

2. Grundlagen

Im Folgenden werden die zum Verständnis der Arbeit notwendigen Grundlagen erläutert. Es wird zunächst auf Peer-to-Peer-Systeme eingegangen, anschließend werden zwei wichtige Vertreter und Optimierungsansätze dieser Systeme vorgestellt. Der letzte Abschnitt widmet sich Verfahren zur Berechnung synthetischer Koordinaten.

2.1 Peer-to-Peer Systeme

Peer-to-Peer (P2P) Systeme sind - im Gegensatz zu Client-Server Systemen - selbstorganisierende Systeme, die aus gleichberechtigten Knoten bestehen. Die Aufgaben des Gesamtsystems werden dynamisch zwischen allen Knoten aufgeteilt, zentrale Komponenten und Dienste werden nicht benötigt.

Aufgrunddessen lassen sich leicht ausfallsichere und fehlertolerante P2P-Systeme entwerfen. Durch die gemeinsame und verteilte Diensterbringung skalieren diese Systeme im Allgemeinen besser als Client-Server Systeme, bei denen wenige Server zugleich Engpass und „Single point of failure“ darstellen.

Man kann P2P-Systeme anhand ihrer Topologie und der damit verbundenen Routingstrategie in zwei Klassen unterteilen: Unstrukturierte und strukturierte P2P-Systeme.

In unstrukturierten P2P-Systemen werden Nachrichten durch Fluten oder Random-Walk zum Ziel weitergeleitet. Beim Fluten leiten die Zwischenknoten die Nachricht an alle Nachbarn weiter. Hierbei handelt es sich um eine Breitensuche. Ist mehr Information über den Empfänger einer Nachricht vorhanden, kann auch selektiv geflutet werden. Man versucht, die Nachrichten nicht an alle Nachbarn weiterzuleiten, sondern nur an eine Auswahl. Beim Random-Walk handelt es sich dagegen um eine Tiefensuche. Nachrichten werden nur an einen Nachbarn weitergeleitet. Meist kennt jeder Knoten alle Nachbarn in einem gewissen Umkreis und kann anhand dieser Information Nachrichten gezielt weiterleiten. Eine erweiterte Form des Random-Walk ist das High-Degree-Seeking. Dabei werden beim Weiterleiten der Nachrichten Nachbarn, die viele Verbindungen zu anderen Knoten haben, bevorzugt. Somit konzentriert sich die Suche auf gut vernetzte Knoten.

Beim Fluten kann zwar garantiert werden, dass Nachrichten einen Weg zum Zielknoten finden, allerdings ist die entstehende Netzlast sehr hoch. Beim Random-Walk hingegen muss jeder Knoten zusätzliche Informationen über andere Knoten speichern, außerdem kann nicht garantieren, dass ein Weg zum Ziel gefunden wird.

Ein wichtiges Entwurfsziel bei strukturierten P2P-Systeme ist effizientes Routing. Die Knoten sind dabei so miteinander vernetzt, dass eine bekannte Topologie entsteht. Hierin kann zielgerichtet geroutet werden, eine Nachricht nähert sich nur monoton steigend dem Ziel. Jedem Knoten wird ein eindeutiger Bezeichner (engl. identifier, abgekürzt „ID“) des Wertebereichs zugeordnet. Inhalte sind unter dem Hashwert ihres Schlüssels zu finden, wobei jeder Knoten - abhängig von seiner eigenen ID - einen bestimmten Wertebereich verwaltet. Idealerweise sind die Knoten und die Schlüssel gleichmäßig auf den gesamten Wertebereich verteilt. Die verschiedenen Systeme bedienen sich dabei unterschiedlicher Topologien und Geometrien, was verschiedene Routingansätze nach sich zieht. Ring, Torus, Hypercube, Baum und Butterfly sind Topologien, die in Systemen wie Chord [20], CAN [14], Pastry [17], Tapestry [22] und Viceroy [12] verwendet werden. Die meisten Systeme realisieren verteilte Hashtabellen. Wird im Folgenden die Abkürzung DHT (distributed hashtable) verwendet, so sind damit strukturierte P2P-System gemeint.

Typische Anwendungengebiete von DHTs sind Applikationen, die auf vielen Knoten verteilt werden und dort skalieren sollen. Beispielhaft sind hier einige existierende Anwendungen aufgelistet:

- Verteilte Dateisysteme (CFS [4])
- Kooperatives Web Caching (Squirrel [10])
- Archivierungs- und Backupsysteme (OceanStore [11])
- Gruppenkommunikation mit Multicast und Anycast (Scribe [18])
- Verteilte Datenbanken (PIER [9])
- Systeme zur Lastverteilung

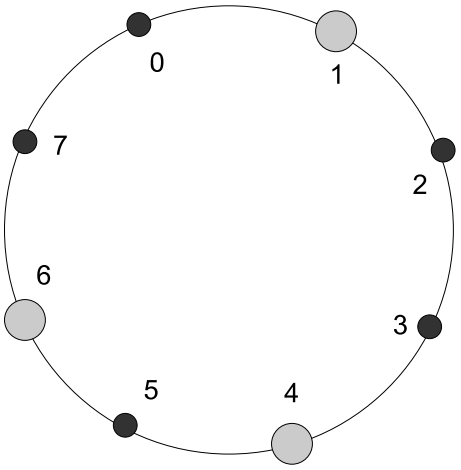
Unstrukturierte P2P-Systeme unterstützen eine unscharfe Suche. Dies kann in DHTs nicht ohne Weiteres realisiert werden, da nach dem Hashwert eines Schlüssels gesucht und geroutet wird. [21] stellt ein Verfahren zur unscharfen und semantischen Suche in DHTs vor.

Im Folgenden werden zwei DHTs genauer beschrieben. Auf diesen beiden Ansätzen basiert das im folgende Kapitel vorgestellte Konzept.

2.1.1 Chord: „Ein Suchdienst für Internet Applikationen“

Chord ist ein verteiltes Lookup-Protokoll für die effiziente Suche nach Schlüsseln. Dabei unterstützt Chord nur die Operation „Lookup“: Zu einem Suchschlüssel wird der Knoten des Chord-Ringes gefunden, der diesen Schlüssel verwaltet.

Chord hat einen ringartigen, eindimensionalen Schlüsselraum. Innerhalb dieses Wertebereiches befinden sich alle Schlüssel und die IDs der Chord-Knoten. Jeder Knoten



Finger-Tabelle: Knoten 1

i	Finger (ID)
1	4
2	4
3	6

Finger-Tabelle: Knoten 6

i	Finger (ID)
1	1
2	1
3	4

Abbildung 2.1: Chord-Ring mit 3 Knoten

verwaltet alle Schlüssel zwischen der ID seines Vorgängerknotens und seiner eigenen ID.

Das Routing lässt sich im einfachsten Fall so realisieren, dass jeder Knoten Nachrichten immer an seinen direkten Nachfolger weiterleitet, bis die Nachricht am Ziel angekommen ist. Dies führt aber zu einer mittleren Pfadlänge von $O(n)$, gemessen in Hops.

Um das Routing effizienter zu gestalten, kennt jeder Chord-Knoten neben seinem direkten Nachfolger auch eine Reihe von Knoten, die sich in logarithmisch ansteigendem Abstand von ihm befinden[1]. Der i-te Eintrag der *Fingertabelle* eines Knotens n verweist auf Knoten s, der die ID des Knotens n um mindestens 2^{i-1} überragt, mit $1 \leq i \leq m$. Dieser Knoten hat dann also die Chord-ID $n + 2^{i-1} \bmod 2^m$, wobei m die Anzahl der Bits des Wertebereiches ist. Die Fingertabelle jedes Chord-Knotens umfasst m Einträge, der erste Eintrag ist der direkte Nachfolger.

Abb. 2.1 zeigt einen Chord-Ring mit dem Wertebereich $[0; 2^m - 1]$ mit $m = 3$, den drei Knoten 1, 4 und 6 und den zugehörigen Fingertabellen zweier Knoten. Die Chord-ID des direkte Nachfolger von Knoten 1 berechnet sich folgendermaßen: $(1 + 2^{1-1}) \equiv 2 \bmod 2^3$. Da es aber keinen Knoten mit der ID '2' gibt, wird der nächste aktive Knoten genommen, dieser hat die ID '4'. Der zweite Eintrag in dieser Fingertabelle ist ebenso Knoten 4, da $1+(2^{2-1}) \equiv 3 \bmod 2^3$ und kein Knoten mit der ID '3' existiert. Der dritte Eintrag verweist auf Knoten 6, da $(1 + 2^{3-1}) \equiv 5 \bmod 2^3$.

Das Routing verläuft ähnlich einer binären Suche. Nachrichten werden jeweils an den Vorgängerknoten, also an den Knoten der Fingertabelle, dessen ID kleiner oder gleich dem gesuchten Schlüssels ist, geroutet. Stellt ein Knoten fest, dass er selbst dieser gesuchte Vorgängerknoten ist, so kann er die Nachricht direkt zustellen, indem er diese an seinen direkten Nachfolger weiterleitet. Eine Nachricht wird in einem Chord-Ring mit n Knoten $O(\log n)$-mal weitergeleitet und nähert sich bei jedem Weiterleiten dem Ziel. Das Lokalisieren von Schlüsseln im Chord-System skaliert auch bei einer großen Knotenzahl.

[1]Diese Knoten heißen bei Chord „Finger“.

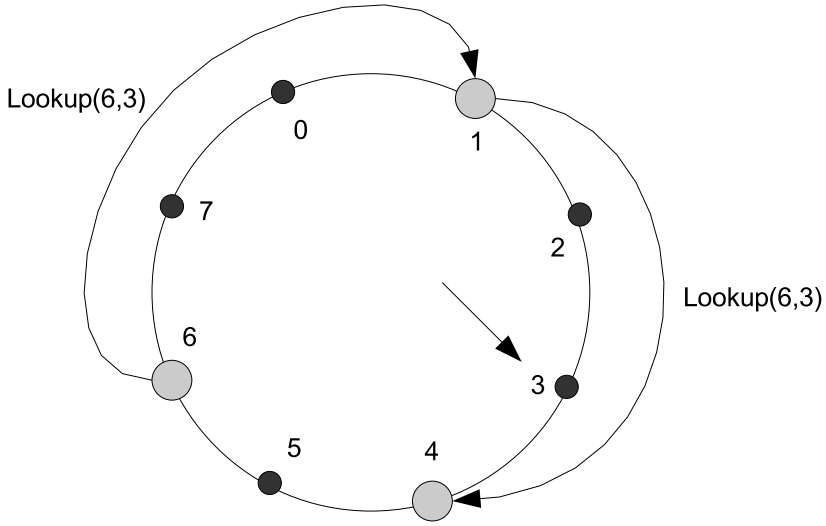


Abbildung 2.2: Suchanfrage von Knoten 6 nach Schlüssel 3

Abb. 2.2 zeigt, wie eine Suchanfrage nach Schlüssel '3' von Knoten 6 geroutet wird. Der direkte Vorgänger in der Fingertabelle von Knoten 6 der gesuchten ID '3' ist 1, an diesen leitet Knoten 6 die Nachricht weiter. Knoten 1 stellt fest dass es keinen Eintrag in seiner Fingertabelle gibt, der kleiner oder gleich der ID des gesuchten Schlüssels ist. Also muss Knoten 1 die Suchanfrage an seinen direkten Nachfolger weiterleiten, dieser ist für den Schlüssel verantwortlich.

Der Beitritt neuer Knoten (engl. bootstrapping) wird mittels so genannter *Join-Routinen* geregelt. Voraussetzung für diesen Beitritts-Mechanismus ist, dass ein Knoten, der einem existierenden Chord-Ring beitreten möchte, mindestens einen Knoten dieses Rings kennt, über den er eine Beitritts-Anfrage stellen kann. Eine Beitritts-Anfrage liefert dem beitretenden Knoten seinen direkten Nachfolger, dieser wird in der Fingertabelle gespeichert. Um sich bei diesem Nachbarn und den anderen Knoten im Netz bekannt zu machen, ruft ein Knoten n die *Stabilisierungsroutine* auf. Hiermit wird überprüft, ob der bisher bekannte Nachfolger s, wirklich der korrekte Nachfolger s^* ist: Knoten n fragt s nach dessen Vorgänger. Je nach Antwort muss Knoten n nun entscheiden, ob s der korrekt Nachfolger s^* ist. Anschließend wird gegebenenfalls der von s mitgeteilte Vorgänger als neuer Nachfolger s in die Fingertabelle eingetragen.

Da die Stabilisierungsroutine regelmäßig aufgerufen wird, kennt jeder Knoten nach einer gewissen Verzögerung seinen korrekten Nachfolger. Desweiteren können Knoten ihre Nachfolger über ihre Existenz informieren, damit diese den richtigen Vorgänger kennen. Auch die Einträge der Fingertabelle werden periodisch aktualisiert. Durch diesen Soft-State Ansatz wird der Beitritt eines neuen Knotens unkompliziert gehalten.

Für das zuverlässige Funktionieren des Chord-Protokolls muss sichergestellt sein, dass jeder Knoten seinen Nachfolger kennt. Ist dies nicht der Fall, so können Lookups fehlschlagen. Somit ist die Konsistenzerhaltung des Chord-Rings ein wichtiger Bestandteil des Protokolls. Beim Ausfall eines Knotens muss möglichst schnell der nächstmögliche Nachfolgerknoten gefunden werden. Hierzu speichert jeder Knoten eine so genannte Nachfolgerliste mit den nächsten l direkten Nachfolgeknoten. Bis zu

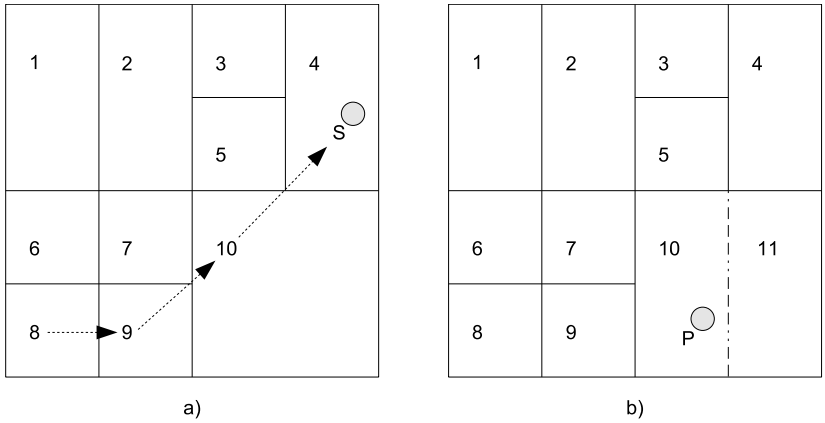


Abbildung 2.3: Routing im 2-dimensionalen CAN (a) und Aufteilung einer Zone (b)

$l-1$ aufeinanderfolgende Knotenausfälle können nun kompensiert werden. Je größer die Nachfolgerliste, desto robuster wird das System gegen Ausfälle. Der Trade-Off zwischen Robustheit und Aufwand muss von Umgebung zu Umgebung abgeschätzt werden und hängt von der Ausfallwahrscheinlichkeit p jedes einzelnen Knotens ab.

2.1.2 CAN: „Ein inhaltsadressierbares Netzwerk“

In [14] wird das „content addressable network“ (CAN) vorgestellt. Dessen Topologie entspricht einem Torus in einem d-dimensionalen Kartesischen Koordinatensystem. Dieser Torus wird in Zonen unterteilt, die von den einzelnen Knoten verwaltet werden. Ein Schlüssel K wird durch eine Hashfunktion auf einen Punkt P im Koordinatensystem abgebildet. Der Knoten, in dessen Zone P fällt, ist nun für diesen Schlüssel und die damit assoziierten Daten zuständig.

Jeder Knoten kennt die Netzwerkadressen und Zonenkoordinaten der Knoten, die mit seiner Zone direkt benachbart[2]. Ein Knoten leitet eine Nachricht an einen Knoten seiner Routingtabelle weiter, dessen Zonenkoordinaten näher an den gesuchten Koordinaten liegen. Abb. 2.3(a) verdeutlicht das Routing. Die Nachricht wird an S adressiert und wird von Zone 8, über die Zonen 9 und 10 in die Zielzone 4 weitergeleitet.

Möchte ein neuer Knoten dem Netz beitreten, bestimmt er zunächst zufällig die Koordinaten eines Punktes P. Nun wird eine Beitrittsanfrage in die Zone weitergeleitet, in die P fällt. Diese Zone wird nun zwischen dem alten Zoneninhaber N_1 und dem beitretenden Knoten N_2 in zwei gleichgroße Zonen aufgeteilt. N_2 erhält nun von N_1 die nötigen Routingtabelleneinträge, danach kann N_1 seine eigene Routingtabelle anpassen. Abschließend werden die benachbarten Knoten über die Teilung der Zone und den neuen Nachbarn informiert. In Abb. 2.3(b) wird durch den Punkt P die Zone 10 zur Teilung bestimmt, Zone 11 entsteht.

Fällt ein Knoten aus oder möchte das Netz verlassen, so muss die freiwerdende Zone von einem benachbarten Knoten übernommen werden. Im Idealfall kann diese Zone mit einer anderen angrenzenden Zone direkt vereinigt werden. Dieser Fall ist in Abb. 2.4 skizziert. Die Zone 7 kann mit Zone 9 verschmolzen werden.

[2]Bei CAN versteht man unter Nachbarn numerisch benachbarte Knoten. Die Zonen dieser Knoten berühren einander.

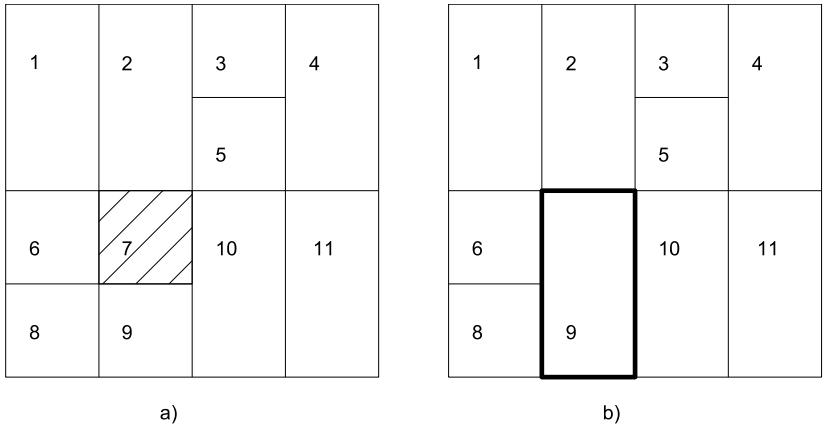


Abbildung 2.4: Der Knoten, der Zone 7 verwaltet, fällt aus (a). Diese Zone wird mit Zone 9 verschmolzen (b).

Ist ein solches Verschmelzen nicht möglich, so muss ein Knoten (übergangsweise) zwei Zonen verwalten. Verschiedene Metriken zur Auswahl der übernehmenden Zone sind denkbar[3]. Entartungen sollten dabei vermieden und eine gleichmäßige Verteilung der Schlüssel auf die Knoten gewährleistet werden. Deshalb müssen gegebenenfalls regelmäßig Umstrukturierungsalgorithmen durchgeführt werden, um entstandene Entartungen rückgängig zu machen.

Im Gegensatz zu Chord umfasst die Routingtabelle nur $2d$ Einträge und wächst mit zunehmender Knotenzahl nicht. Die mittlere Pfadlänge ist dagegen etwas höher also bei Chord und beträgt in einem CAN mit n Knoten $O(n^{\frac{1}{d}})$.

2.2 Optimierungsansätze

P2P-Systeme, speziell strukturierte, werden häufig auch als Overlay-Netze bezeichnet, da sie eine eigene Routingstrategie und Knotenadressierung über ein darunterliegendes Netz aus Verbindungen zwischen physikalisch benachbarten Knoten (Underlay), implementieren. Ein wichtiger Aspekt dieser Overlay-Netze ist die Art, wie diese Systeme die Knoten, die im darunterliegenden Netz direkt benachbart sind, miteinander vernetzen. Je mehr sich in einem solchen Overlay-Netz die Nachbarschaftsbeziehungen des darunterliegenden Underlay-Netzes widerspiegeln, umso besser.

Abb. 2.5 zeigt sechs Knoten und deren Verbindungsstruktur im Internet und dem darüberliegenden, ringförmigen Overlay. Knoten 3 und 6 sind im Overlay direkt benachbart, im Underlay liegen diese allerdings vier Hops auseinander. Würde man nun eine Nachricht von Knoten 3 an Knoten 2 über Knoten 6 im Overlay routen, so würde dies zu sieben Hops im Underlay führen.

Ein Maß für die Effizienz solcher Overlay-Netze ist die Anzahl der benötigten Hops im Overlay beim Routing. Geschickterweise bezieht man in ein solches Maß aber auch Eigenschaften, die sich durch das darunterliegende Netz ergeben, mit ein. So kann man beispielsweise anhand der Netzwerklatenzen die „Entfernung“ zweier im

[3] [14] schlagen vor, aus der Nachbarschaft den Knoten zur Übernahme der ausgefallenen Zone zu bestimmen, der bisher die kleinste Zone verwaltet.

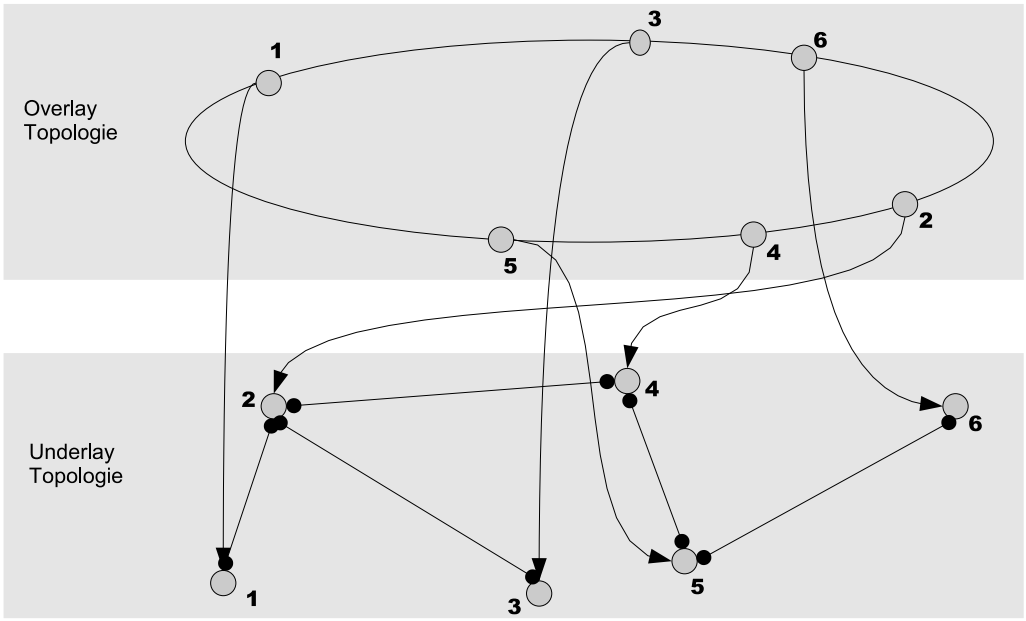


Abbildung 2.5: Topolgie des Overlays und des darunterliegenden Netzes

Overlay benachbarten Knoten bezüglich des darunterliegenden Netzes bestimmen. In [14] wird der Begriff „latency stretch“ eingeführt: Dieser beschreibt, wie sehr die durchschnittliche Latenz eines Overlayhops die eines Underlayhops übersteigt.

Ein negatives Charakteristikum vieler P2P-Systeme sind die relativ großen „Entfernungen“ zwischen benachbarten Overlayknoten. Da die Overlay-IDs meist zufällig zugewiesen werden, ist es unwahrscheinlich, dass zwei im Underlay benachbarte Knoten dies auch im Overlay sind. [16] unterscheiden drei Ansätze, wie Overlays an die Topologie des darunterliegenden Netzes und an dessen Entfernungsstruktur angepasst werden können:

Proximity Route Selection (PRS) Man wählt zum Weiterleiten von Paketen den Knoten aus der Routingtabelle, der den besten Trade-Off zwischen Gewinn (Fortschritt im ID-Raum) und Kosten (gemessen durch die Netzwerklatenz) aufweist. Dies ist eine recht einfache Möglichkeit, um Netzwerkentfernungen im Underlay beim Routing miteinzubeziehen, ohne das Erstellen der Routingtabellen ändern zu müssen.

Proximity Neighbor Selection (PNS) Erfolgt die Erstellung der Routingtabelle allerdings unter Einbeziehung von Netzwerklatenzen, bezeichnet man dies als PNS. Letztendlich ist dies eine Variation der PRS, allerdings ist sie oftmals wesentlich schwerer zu realisieren, da sie höhere Anforderungen an die Flexibilität der DHT stellt.

Geographical ID Selection (GIS) Hierbei wird versucht, Knoten die im physikalischen Netzwerk nahe beieinander liegen, auch im Overlay-Netz ähnliche IDs zuzuweisen. Dieser Ansatz kann zwar wirksam die pro Overlayhop im Underlay zurückgelegte Entfernung reduzieren, allerdings erweisen sich auf GIS basierende Overlay-Netze als wesentlich weniger fehlertolerant. Fällt ein Teil

des Netzes aus, so führt dies zu enormen Inkonsistenzen im Overlay, da auf einmal ein ganzer ID-Bereich verloren geht.

Inwieweit diese Ansätze in den verschiedenen DHTs verwendet werden können, hängt von deren Flexibilität bei beim Routing und Wahl der Nachbarn ab. [1] vergleicht einige DHTs.

2.3 Synthetische Koordinatensysteme

Um die Netzwerkdistanzen, bzw. Netzwerklatenzen, zwischen Knoten vorhersagen zu können, ohne dabei für jedes Knotenpaar die Latenz messen zu müssen, wurden Ansätze entwickelt, die meist auf der Berechnung synthetischer Koordinaten beruhen. Aus diesen Koordinaten kann dann mit einer Distanzfunktion die Latenzen zwischen zwei Knoten abgeleitet werden kann.

Im Folgenden werden grob einige gängige Verfahren skizziert:

IDMaps Einer der ersten Ansätze war IDMaps, ein infrastrukturbasierter Dienst zur Abschätzung der Latenzen zwischen Internetknoten. IDMaps basiert auf der Verwendung von einigen 1000 so genannter Tracer-Knoten. Alle Tracer-Knoten kennen ihre Latenzen untereinander. Außerdem gibt es eine eindeutige Zuordnung von Tracer-Knoten zu CIDR-Adresspräfixen. Die Latenzen zwischen diesen Präfixen und den so genannten Präfix-Tracer-Knoten sind bekannt.

Die Berechnung der Latenz zwischen zwei Knoten k_1 und k_2 setzt sich nun aus den Latenzen zwischen den Präfix-Tracer-Knoten von k_1 und k_2 und der Latenz zwischen den beiden Tracer-Knoten zusammen [8].

Die Genauigkeit der Latenzschätzung hängt dabei von der Anzahl der Tracer-Knoten ab.

Global Network Positioning (GNP) Dieser Ansatz beruht auf einer kleinen Anzahl so genannter Landmark-Knoten, deren Koordinaten allen Knoten bekannt sein müssen. Andere Knoten müssen nun die Knoten-zu-Landmark-Latenzen via ICMP-Nachrichten messen. Basierend auf diesen Messungen werden nun die eigenen Koordinaten relativ zu denen der Landmark-Knoten errechnet [13].

Binning Dieser Ansatz, vorgestellt in [15], basiert auf einer Klassifizierung der Knoten. Liegen zwei Knoten in der gleichen Klasse, sind sie im Bezug auf die Netzwerk-Latenz relativ nahe beieinander. Dieser Ansatz basiert auf der Annahme, dass es nicht ist notwendig ist, den jeweils optimalen Knoten zu finden, sondern dass auch eine grobe Abschätzungen durch Einteilung in verschiedene Distanzklassen ausreicht, um messbare Performanzsteigerungen zu erzielen.

Vivaldi Hierbei handelt es sich um einen vollständig verteilten Algorithmus. Vivaldi [2] kommt ohne zentrale Infrastruktur aus und ist deshalb für viele P2P-Anwendungen besonders interessant. Jeder Knoten berechnet seine eigenen synthetischen Koordinaten in einem d-dimensionalen Koordinatensystem. Dies erfolgt durch regelmäßigen Austausch der geschätzten Koordinaten und einer gemessenen Latenz zwischen zwei Knoten. Anhand dieser Daten kann

nun jeder Knoten berechnen, inwiefern er die eigenen Koordinaten verschieben müsste, um die gemessene Latenz mit der geschätzten Latenz in Einklang zu bringen[4]. Dabei wird angenommen, dass alle Knoten regelmäßig mit anderen Knoten kommunizieren und so die benötigten Daten austauschen können. Es ist ausreichend, wenn dabei jeder Knoten mit nur einem Bruchteil aller Knoten kommuniziert.

Die Latenz lässt sich durch Berechnung der euklidischen Distanz der Koordinaten zweier Knoten schätzen. Hinsichtlich der Genauigkeit ist Vivaldi mit GNP vergleichbar. Die Qualität der Prognose hängt hier von der Anzahl der Kommunikationspartner und der Dimensionalität der Koordinaten abhängt.

[4]Der zentralisierte Algorithmus basiert auf der Simulation eines Sprungfedernverbundes, um die Abweichung zwischen den geschätzten und den tatsächlichen Distanzen zu minimieren. Im verteilten Fall simuliert jeder Knoten einen Ausschnitt dieses Verbundes.

3. Ein hybrides Overlay basierend auf Chord und CAN

Das im Folgenden vorgestellte hybride Overlay-Netz (HON) stellt einen Ansatz dar, um die in Kapitel 2 erläuterte PNS in eine DHT einzubetten. Wie in [5] beschrieben, übertrifft der durch PNS erzielbare Performanzgewinn den der durch eine PRS erreicht werden kann. Eine PRS lässt sich aber in den meisten DHTs einfach implementieren. Möchte man jedoch eine PNS realisieren, stößt man dabei auf zwei Probleme: Zum einen weisen die verschiedenen DHT-Varianten Unterschiede hinsichtlich der Flexibilität bei der Wahl der Nachbarn auf. Zum anderen stellt sich die Frage, wie die Latenz - das Kriterium bei der Nachbarselektion - für eine große Anzahl potentieller Nachbarn effizient bestimmt werden kann.

Durch die Kombination der beiden DHTs Chord und CAN und unter Verwendung eines Systems zur Ermittlung synthetischer Koordinaten kann diesen Problemen begegnet werden.

Ein solches System genau zu spezifizieren, zu implementieren und zu evaluieren ist Thema dieser Studienarbeit.

Im Folgenden wird das HON beschrieben, die darauffolgenden Abschnitte spezifizieren die dafür notwendigen Komponenten Chord und CAN und die erforderlichen Modifikationen gegenüber den im Kapitel 2 vorgestellten Varianten. Abschließend werden die Vorteile und Eigenschaften dieses Systems erläutert.

3.1 Das Gesamtsystem

Das Gesamtsystem entsteht durch die Vereinigung der Nachbarschaftstabelle eines eindimensionalen Chord-Rings mit der eines d-dimensionalen CANs. Jeder Knoten hat eine eindeutige Chord-ID sowie eine mehrdimensionale, veränderliche CAN-ID. Diese stellt die Koordinaten des Knotens im Netzwerk dar, die durch Vivaldi ermittelt werden. Durch Kombination dieser beiden IDs kann ein Knoten im $d + 1$-dimensionalen Raum dargestellt werden, wie in Abb. 3.1 verdeutlicht. Die Punkte in diesem Raum werden wiederum als CAN-Koordinaten aufgefasst und es werden

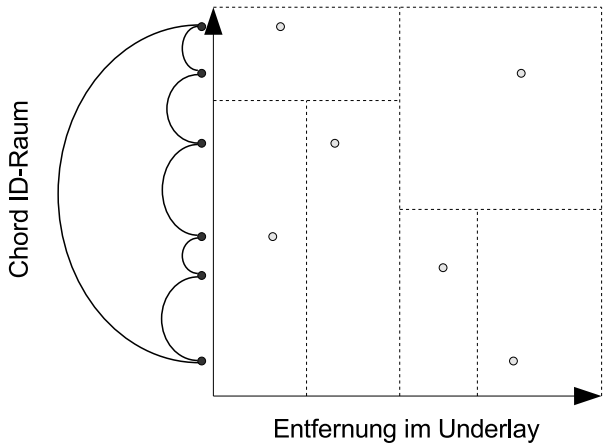


Abbildung 3.1: $d+1$-dimensionaler Raum des CANplus

zusätzlich alle für ein entsprechend dimensioniertes CAN erforderlichen Nachbarschaftsverbindungen erstellt. Dieses $d+1$-dimensionale CAN wird im Folgenden CANplus bezeichnet.

Alle Nachbarschaftsverbindungen dieser drei Komponenten bauen dabei direkt auf dem Underlay auf. Insofern befinden sich die Komponenten im Schichtenmodell auf der gleichen Ebene. Betrachtet man allerdings die logische Anordnung der Komponenten, so ergibt sich die in Abb 3.2 dargestellte Hierarchie.

Chord kann als reiner ID-Raum betrachtet werden, in dem normalerweise Lookups durchgeführt werden; Daten werden mittels Chord-Schlüsseln adressiert. Die CAN-Koordinaten spiegeln Entfernungen zwischen den Knoten im darunterliegenden Netz wider. CANplus verbindet beide Komponenten.

3.2 Die Komponente Chord

Chord stellt das Basissystem dar. Hierfür kann eine Chordvariante, wie im Kapitel 2 spezifiziert, eingesetzt werden. Erweiterungen sind nur hinsichtlich der Aktualisierung der Fingertabelle notwendig.

Knoten, die dem Gesamtsystem beitreten wollen, müssen zunächst Teil des Chord-Rings werden. Verlässt ein Knoten den Chord-Ring, verlässt er somit das Gesamtsystem.

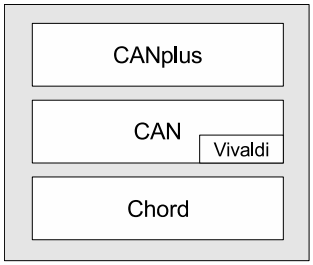


Abbildung 3.2: Logische Anordnung der Komponenten des HON

Es besteht die Möglichkeit, das Gesamtsystem auf Chord zu reduzieren. Die darübergelagerten Komponenten CAN und CANplus stellen eine optionale Erweiterung dar. Ohne diese wird eine PNS nicht unterstützt, da die Bestimmung der optimalen Chordfinger mit Hilfe einer speziellen Routingstrategie (siehe Abschnitt 3.5) im CANplus durchgeführt wird. Die Chordfinger müssen in diesem Fall durch das übliche Verfahren bestimmt werden. Die Konsistenz des Chord-Rings bleibt aber erhalten und Nachrichten können unbeeinträchtigt bei einer mittleren Pfadlänge von $O(log(n))$ geroutet werden.

3.3 Die Komponente CAN

Ein d-dimensionales[1] CAN setzt auf dem beschriebenen Basissystem Chord auf. Die CAN-ID eines Knotens besteht aus dessen synthetischen Koordinaten, die mittels Vivaldi bestimmt werden.

Gegenüber der ursprünglichen Version von CAN, wie sie in [14] und in Kapitel 2 vorgestellt wird, sind einige Änderungen vorzunehmen. Diese Notwendigkeit resultiert aus der Tatsache, dass im CAN-Subsystem eine Zone nicht nur durch Zonenkoordinaten, sondern zusätzlich durch die (Netzwerk-) Koordinaten des Knotens bestimmt wird. Die Verfahren zur Zonenaufteilung und zur Zonenwiedervereinigung müssen deshalb angepasst werden. Ein möglicher Mechanismus wird im Folgenden spezifiziert.

3.3.1 Bestimmung der CAN-Koordinaten

Sobald ein Knoten dem Basissystem beigetreten ist, aktualisiert er kontinuierlich mit Hilfe des Vivaldi-Algorithmus seine (Netzwerk-) Koordinaten. Nach dem Initialisieren der Koordinaten bedarf es einiger Zeit, bis die eigenen Koordinaten durch Kommunikation mit anderen Knoten konvergiert haben. Stellt der Knoten am Ende dieser Einschwingphase stabile Koordinaten fest, kann er dem CAN-Subsystem beitreten.

Ändern sich nun die eigenen CAN-Koordinaten, muss ein Knoten zunächst prüfen, ob diese neuen Koordinaten noch in seiner alten Zone liegen. Ist dies nicht der Fall, muss er die alte Zone verlassen und dem System wieder neu beitreten, um so eine passende Zone zugeteilt zu bekommen. Stellt ein Knoten ein permanentes Schwanken der eigenen Koordinaten fest, so kann er sich auch entscheiden, nicht am CAN-Subsystem teilzunehmen.

Um festzustellen, ob die eigenen Koordinaten stabil sind, wird die Anzahl der Koordinatenänderungen s über einen bestimmten Zeitraum t betrachtet: Liegt die Sprungfrequenz $f = \frac{s}{t}$ über einem Schwellenwert c_t wird dies als zu starkes Schwingen der Koordinaten interpretiert.

3.3.2 Bootstrapping

Um dem CAN-Subsystem beitreten zu können, muss zunächst eine Beitritts-Anfrage in das schon bestehende CAN-Subsystem und dann dort zu dem entsprechenden

[1] Welche Dimensionalität am besten geeignet ist, hängt davon ab, wieviele Dimensionen Vivaldi benötigt, um die Koordinaten möglichst exakt bestimmen zu können. In [2] und [3] werden einige Aussagen über geeignete Dimensionalitäten gemacht.

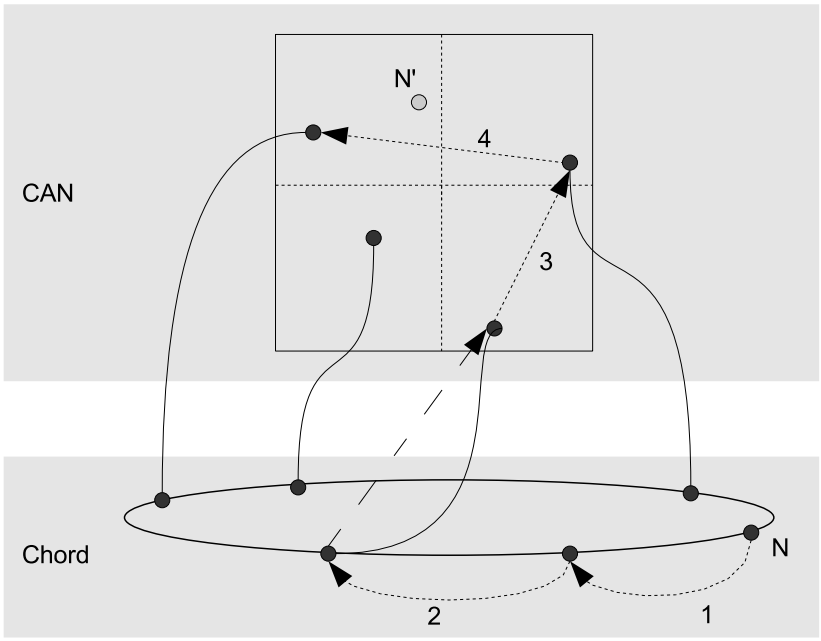


Abbildung 3.3: Ablauf des CAN-Bootstrappings. Die Beitritts-Anfrage von Knoten N wird entlang des markierten Pfades durch Chord und CAN zum Knoten in der CAN-Zielzone weitergeleitet.

Knoten weitergeleitet werden. Diese muss neben den CAN-Koordinaten des beitretenden Knotens auch dessen Chord-ID enthalten.

Die Anfrage wird zunächst im Basissystem Chord jeweils an den direkten Nachfolger weitergeleitet, bis sie zu einem Knoten gelangt, der schon Teil des CAN-Subsystem ist. Nun wird sie gemäß der üblichen CAN-Regeln in die Zielzone geroutet. Dieses Verfahren wird in Abb. 3.3 anhand eines Beispiels verdeutlicht.

Der diese Zone verwaltende Knoten kann dann die Teilung der Zone einleiten. Die Kommunikation mit dem beitretenden Knoten erfolgt über Chord, da hier beide Knoten eindeutig adressierbar sind.

Beim Bootstrapping muss vor allem vermieden werden, dass mehrere CAN-Subsysteme nebeneinander aufgebaut werden. Sendet ein Knoten seine Beitritts-Anfrage gibt es zwei mögliche Szenarien: Entweder es existiert bereits ein CAN, dann erhält dieser Knoten eine Beitritts-Antwort von dem für die zu teilende Zone verantwortlichen Knoten. Existiert noch kein CAN, so wird die gestellte Beitritts-Anfrage einmal durch den gesamten Ring geroutet und erreicht schließlich wieder den anfragenden Knoten. In diesem Fall initiiert dieser Knoten ein neues CAN.

3.3.3 Teilung einer Zone

In der ursprünglichen CAN-Variante wird die bei einem Knotenbeitritt zu teilende Zone durch einen zufällig gewählten Punkt bestimmt. Unabhängig von der genauen Lage des Punktes wird dann die Zone halbiert. Die Wahl der Teilungsachse hängt von den bisher gemachten Teilungen ab. In der hier verwendeten CAN-Variante erfordert die Verwendung von Netzwerkkoordinaten, dass diese innerhalb der dem Punkt zugeordneten Zone liegen. Bei der Teilung einer Zone muss dies berücksichtigt werden.

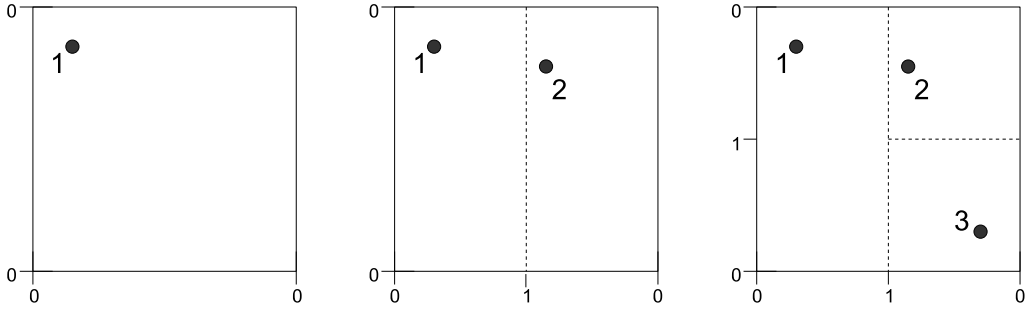


Abbildung 3.4: Zonenaufteilung in einem 2-dimensionalen CAN. Die Knoten 1, 2 und 3 werden sukzessive hinzugefügt. An den Achsen sind ist die Ordnung der bestehenden Kanten vermerkt.

Zunächst muss ermittelt werden, in welcher Dimension eine Zone geteilt werden soll. Der Knoten N_1, der bisher die Zone verwaltet, erstellt eine Liste der d möglichen Teilungshyperebenen. Eine solche Hyperebene halbiert die bisherige Zone in einer Dimension. Für jede Teilungshyperebne wird deren Ordnung[2] ermittelt. Diese Ordnung gibt an, durch wieviele Teilungen der jeweiligen Dimension eine Zone entstanden ist. Ziel ist, die Zone möglichst durch eine Hyperebene niedriger Ordnung zu teilen, wodurch möglichst „quadratische“ Zonen entstehen sollen.

Die Liste der Teilungshyperebenen wird nach aufsteigender Ordnung sortiert. Nun wird geprüft, ob bei Verwendung der ersten Teilungshyperebene die beiden Knoten N_1 und der beitretende Knoten N_2 jeweils in eine andere Zonenhälfte fallen würden. Ist dies der Fall, so kann die Teilung der Zone durch diese Hyperebene vorgenommen werden. Andernfalls wird die nächste Teilungshyperebene der Liste probiert.

Abb. 3.4 verdeutlicht diese Vorgehensweise an einem 2-dimensionalen Beispiel. Die Zone von Knoten 1 könnte in beiden Dimensionen geteilt werden. Durch die Lage der Koordinaten von Knoten 2 ist aber nur eine Teilung der X-Achse möglich. Im dritten Schritt wird die Y-Achse geteilt. Eine Teilung der X-Achse würde hingegen zu einer Kante 2-ter Ordnung führen.

Abb. 3.5 zeigt das Ergebnis einer kleinen Simulation. Es wurden sukzessive Knoten hinzugefügt, deren Koordinaten gleichverteilt zufällig ermittelt wurden. Insgesamt wurde 50 Teilungsversuche unternommen, allerdings konnten nur 41 Teilungen erfolgreich durchgeführt werden. Konnte eine Zone nämlich in keiner Dimension geteilt werden, weil die Koordinaten der beiden Knoten zu dicht beieinanderlagen, wurde dieser Knoten nicht eingefügt.

Mit einer Wahrscheinlichkeit von $p = (\frac{1}{2})^d$ kann eine Zone in keiner Dimension so halbiert werden, dass N_1 und N_2 jeweils in einer anderen Zonenhälfte liegen. Diesem Problem kann auf zwei Arten begegnet werden:

Verschiebung der Koordinaten Liegt N_2 nahe an der Teilungshyperebene, so können dessen Koordinaten geringfügig angepasst werden, so dass die neuen Koordinaten von N_2 in der anderen Zonenhälfte liegen (Abb. 3.6,a).

[2]Zwei Ansätze zur Ermittlung dieser Ordnung werden in Anhang A.2 vorgestellt.

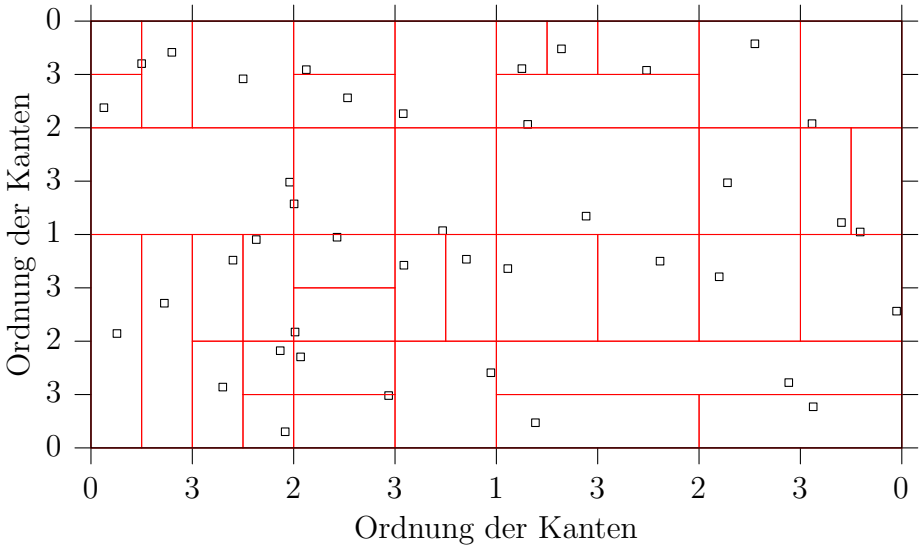


Abbildung 3.5: Simulation der Zonenaufteilung

Späterer Teilungsversuch Würde der erste Ansatz zu einer zu großen Änderung der Koordinaten führen, kann nach einer Wartezeit nochmal versucht werden, N_2 einzufügen. Haben zwischenzeitlich weitere Aufteilungen der Zone stattgefunden, so ist eine passende Teilung der Zone jetzt eventuell möglich (Abb. 3.6,b).

Wird der Koordinatenraum im Verhältnis zu den zwischen den Knoten vorhandenen Netzwerkdistanzen viel zu groß gewählt, so wird das Aufteilen einer Zone nahezu unmöglich, da alle Knoten im Vergleich zur Größe des gesamten Raums sehr dicht beieinanderliegen. Dieses Problem kann durch Anpassung der Größe des Koordinatenraumes an das Anwendungsszenario umgangen werden. Soll beispielsweise das System nur auf einem Universitätscampus genutzt werden, dürfte ein Koordinatenraum von einigen wenigen Millisekunden ausreichen.

Bei Erhalt einer Teilungsanfrage entscheidet der Knoten N_1 durch welche Hyperebene seine Zone geteilt wird und welcher Konfliktbehandlungsmechanismus angewendet werden soll. Dabei teilt er dem Knoten N_2 entweder dessen Zonenkoordinaten und gegebenenfalls eine neue CAN-ID mit, oder er lehnt die Teilung seiner Zone ab.

Kann eine Teilung durchgeführt werden, informiert N_1 den neuen Knoten N_2 zunächst über dessen Nachbarn, anschließend passt er seine eigene Nachbarschaftstabelle an. Da das CAN-Subsystem nur zur Einbettung der Netzwerkdistanzen dient, müssen beim Teilen (und auch beim Wiedervereinigen) von Zonen nur Signalisierungsnachrichten übertragen werden.

3.3.4 Wiedervereinigung von Zonen

Verlässt ein Knoten das CAN-Subsystem, so muss die freiwerdende Zone von einem anderen Knoten verwaltet werden. Man kann zwischen angekündigtem und unangekündigtem Verlassen (AV und UAV) unterscheiden. Das Grundprinzip ist in aber in beiden Fällen identisch.

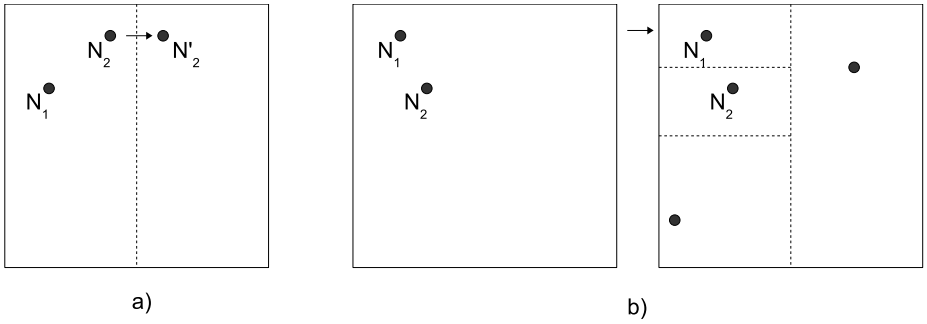


Abbildung 3.6: Zwei Ansätze, um das Einfügen eines Knotens auch dann zu ermöglichen, wenn eine Zone nicht adäquat geteilt werden kann. a) durch Verschiebung der Koordinaten und b) durch einen späteren Teilungsversuch.

Auch an dieser Stelle muss das Verfahren gegenüber der ursprünglichen CAN-Version angepasst werden. Da die CAN-ID eines Knotens immer innerhalb der eigenen Zone liegen muss, lässt sich ein Umstrukturierungsmechanismus, wie in Abschnitt. 2.1.2 vorgestellt und in [14] detailliert beschrieben, nicht anwenden.

Um weiterhin möglichst „quadratische“ Zonen mit Seitenflächen[3] niedriger Ordnung zu erhalten, werden aus einem bestimmten Bereich um die freiwerdende Zone herum alle Zonen entfernt („Reset“). Ein Knoten übernimmt die so entstehende, große Zone. Alle anderen Knoten, die ihre CAN-Zone verloren haben, müssen das CAN-Subsystem verlassen und erneut beitreten.

Der Bereich wird möglichst klein gehalten und kann folgendermaßen ermittelt werden: Es wird die Ordnung aller Seitenflächen der freiwerdenen Zone ermittelt und anschließend der Eckpunkt bestimmt, in dem die Summe der Ordnungen der sich dort berührenden Seitenflächen am geringsten ist. Nun wird aus denen dem Eckpunkt „gegenüberliegenden“ Seitenflächen die mit der höchsten Ordnung ausgewählt. Die Länge der Kanten der freiwerdenden Zone, die durch diese Seitenfläche begrenzt wurden, wird nun verdoppelt. Gibt es mehrere Seitenflächen mit der gleichen Ordnung, muss die Seitenfläche gewählt werden, die zuletzt durch eine Teilung entstanden ist.

Nun ist der Resetbereich eindeutig bestimmt. Eine Resetnachricht wird in diesem Bereich geflutet, indem die Nachricht an alle Nachbarn, deren Zone innerhalb des Resetbereiches liegt, weitergeleitet wird. So werden die betroffenen Knoten angewiesen, das CAN zu verlassen und diesem erst nach einer zufälligen Wartezeit wieder beizutreten. Die Resetnachricht enthält die Eckpunkte des Resetbereiches, die Zonenkoordinaten der freigewordenen Zone und die CAN-ID des übernehmenden Knotens. Abb. 3.7 demonstriert dies an einem einfachen Beispiel.

Im Falle eines AVs ermittelt der verlassende Knoten N_1 und versendet die Resetnachricht. Außerdem weist er einen Knoten N_2 aus der direkten Nachbarschaft, dessen Zone sich innerhalb des Resetbereichs befindet, an, den Resetbereich zu verwalten. Außerdem übermittelt N_1 die für die neue Zone relevanten Nachbarschaftsinformationen an N_2. Alle anderen Knoten, die die Resetnachricht erhalten haben und deren Zone am Rand des Resetbereiches liegt, senden ebenfalls relevante Nachbarschafts-

[3]Die Seitenflächen der Zonen sind Hyperebenen des jeweiligen d-dimensionalen Raums.

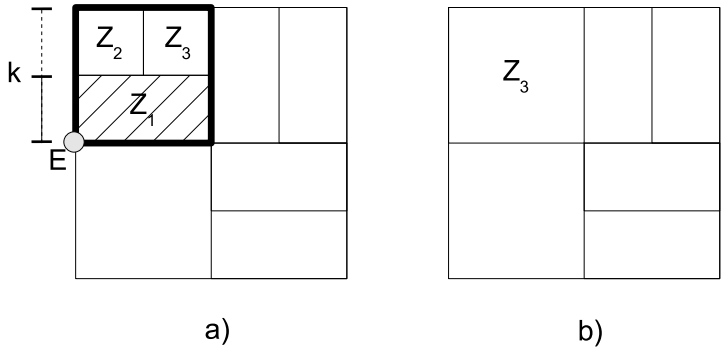


Abbildung 3.7: Reset des markierten Bereichs bei Freiwerden der Zone Z_1. Zur Ermittlung dieses Bereiches wird der Eckpunkt E und die zu Kante k bestimmt (a). Zone Z_3 übernimmt den Bereich (b).

informationen an N_2. Nachdem die betroffenen Knoten das CAN verlassen haben, baut N_2 die Verbindungen zu den neuen CAN-Nachbarn auf.

Kommt es nicht zu einem AV, da beispielsweise ein Knoten ausfällt, wird dies von den Nachbarn, die regelmäßig den Zustand der angrenzenden Knoten überprüfen, bemerkt. Nach einer zufälligen Wartezeit berechnet nun jeder Knoten den Resetbereich. Liegt er selbst innerhalb des Resetbereichs und hat nach einer zufälligen Wartezeit keine Resetnachricht erhalten, bestimmt sich selbst zum Verwalter dieses Bereiches und sendet die Resetnachricht. Der Austausch der Nachbarschaftsinformation verläuft analog dem AV. Allerdings muss der Knoten N_2, der den Resetbereich übernimmt, die Nachbarn von N_1, die nicht innerhalb des Resetbereichs liegen, selbst ermitteln. Hierzu flutet er eine Nachbarschaftsanfrage in einen Bereich, der geringfügig größer als der Resetbereich ist. Die ehemaligen Nachbarn des ausgefallenen Knotens, die außerhalb des Resetbereiches liegen, antworten mit relevanten Nachbarschaftsinformationen.

3.4 Die Komponente CANplus

Nach dem Beitritt eines Knotens zum CAN-Subsystem muss dieser in das CANplus eingegliedert werden. Das Bootstrapping verläuft dabei ähnlich dem oben vorgestellten CAN-Bootstrapping. Die Beitritts-Anfrage wird dabei auf CAN-Ebene an einen Knoten weitergeleitet, der schon in das CANplus eingegliedert ist. Dort wird sie dann in die Zielzone geroutet. Das Aufteilen und Wiedervereinigen der Zonen erfolgt analog der oben vorgestellten Verfahren des CAN-Subsystems. Nach Aufbau der Verbindungen zu den CANplus-Nachbarn ist der Beitritt abgeschlossen.

3.5 Besondere Eigenschaften des Systems

Das Gesamtsystem enthält alle für Chord notwendigen und die aus CAN, bzw. CANplus resultieren Verbindungen. Vermutlich müssen nicht $4d{+}2{+}\log_2 n$ Verbindungen[4] permanent aufrechterhalten werden, da mit hoher Wahrscheinlichkeit einige Knoten nicht nur in einem der drei Subsysteme benachbart sind.

[4]Diese erwartete mittlere Anzahl an Verbindungen setzt sich folgendermaßen zusammen: $2d$ (CAN) $+2(d+1)$ (CANplus) $+\log_2 n$ (Chord), bei d Dimensionen und n Knoten.

Durch den hybriden Ansatz dieses Systems sind vor allem interessante Such- und Routingstrategien möglich:

Routing im Chord-Basissystem Ein Routing mit logarithmischer Performanz kann durch das Chord-Basissystem immer garantiert werden. Vermutlich wird durch die zusätzlichen Verbindungen die durchschnittliche Pfadlänge unter $log(n)$ fallen.

Eine PRS ist an dieser Stelle denkbar: Beim Weiterleiten kann die Verbindung gewählt werden, die das beste Verhältnis zwischen Fortschritt im Chord-ID-Bereich und den Netzwerklatenzen aufweist.

Routing im CANplus Sind bei einer Suche neben dem Chord-Schlüssel auch Netzwerkkoordinaten des Ziels bekannt, so kann nahezu optimal zu diesem Schlüssel geroutet werden. Bei bei jedem Routingschritt kommt die Nachricht im Chord-ID-Raum dem Ziel näher; die Netzwerkdistanz zum Zielknoten verringert sich ebenso kontinuierlich.

In den meisten Fällen dürften die Koordinaten bei einer Suche nicht bekannt sein. Dieses Szenario könnte allerdings genutzt werden, um eine logische Verbindung im Overlay aufzubauen, nachdem durch eine Suche im Chord-Raum die Netzwerkkoordinaten ermittelt wurden. Diese Verbindung kann beispielsweise bei einer länger währenden Kommunikation zweier Knoten verwendet werden.

Proximity Neighbor Selection [5] zeigt, dass auch bei Chord eine PNS möglich ist: Bei der Wahl eines Chord-Fingers besteht eine mit jedem Fingereintrag zunehmende Flexibilität. Für die Finger-ID id muss gelten:

$$(n + 2^{i-1}) \ mod \ 2^m \leq id \leq (n + 2^i - 1) \ mod \ 2^m$$

Idealerweise liegt die ID genau bei $(n + 2^{i-1}) \ mod \ 2^m$. So ist gewährleistet, dass der numerische Abstand zwischen jeweils zwei Chord-Fingern exponentiell ansteigt.

CANplus ermöglicht eine effiziente PNS. Hierzu wird eine Suche nach einem Schlüssel durchgeführt, der sich aus der optimalen ID eines Chord-Fingers und der eigenen CAN-ID zusammensetzt. Abb. 3.8 verdeutlicht dieses Vorgehen. Der gefundene Knoten ist der unter diesen Randbedingungen optimale Chord-Finger.

Suche im CAN-Subsystem Im CAN-Subsystem hält jeder Knoten im Mittel $2d$ Verbindungen zu Knoten aufrecht, deren Distanz zum eigenen Knoten im Netzwerk am geringsten ist. So kann leicht der insgesamt nächsten Knoten ermittelt werden.

Lokal begrenztes Fluten Sollen alle Knoten erreicht werden, die maximal eine Distanz e im Netzwerk zum suchenden Knoten aufweisen, kann im CAN-Subsystem eine Nachricht geflutet werden. Dabei leiten Knoten Nachrichten nur an Nachbarn weiter, deren CAN-IDs noch innerhalb des spezifizierten Bereiches liegen.

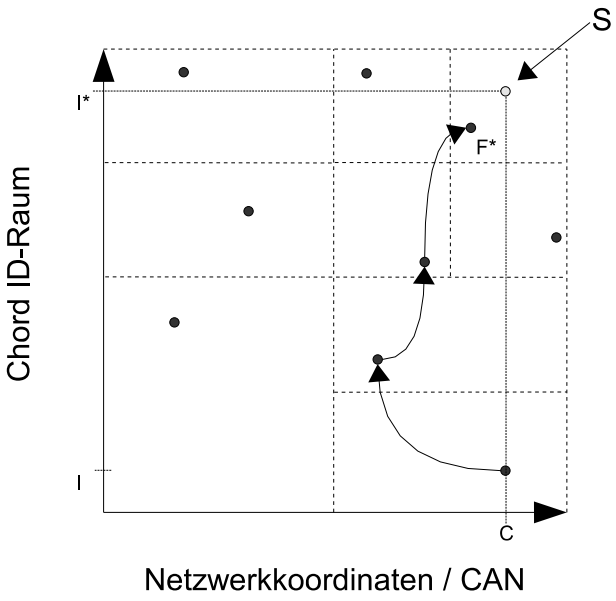


Abbildung 3.8: Routing einer Nachricht zum gesuchten Punkt S im CANplus. S setzt sich aus der eigenen CAN-ID C und der optimalen Chord-ID I^* zusammen. Der Knoten F^* kann nun in die Fingertabelle übernommen werden.

Vor allem beim Routing im CANplus können unterschiedliche Metriken angewandt werden, um aus den bei einem Routingschritt zur Auswahl stehenden Nachbarn zu wählen. Aus n möglichen Nachbarn könnte man bei einer PNS beispielsweise den auswählen, dessen CAN-ID am nächsten an der CAN-ID des Schlüssels liegt. In einem anderen Szenario ist aber auch eine Selektion anhand der Chord-ID möglich.

4. Implementierung

Im Folgenden werden die im Rahmen der Studienarbeit implementierten Module des vorgestellten Konzepts erläutert. Es wird zunächst ein grober Überblick über die einzelnen Module gegeben. Nachfolgend werden wichtige Komponenten der Module detailliert beschrieben.

4.1 Struktur der Implementierung

Das im vorherigen Kapitel vorgestellte hybride Overlay-Netz (HON) wurde in der Programmiersprache „C“ unter Linux als eigenständige „User-Space“-Anwendung implementiert.

In Abb. 4.1 sind die einzelnen Module der Implementierung schematisch dargestellt: Die Module „Chord“, „Nachbarschaftsmanagement“, „synthetische Koordinaten“ und das bisher noch noch nicht implementierte Modul „CAN/CANplus“ sind um das Kernmodul angeordnet. Abb. 4.2 kann die Aufteilung der Module auf die Quelldateien entnommen werden.

4.2 Grundlegende Designentscheidungen

Das HON ist ereignisgesteuert konzipiert. Mögliche Ereignisse sind der Ablauf eines Zeitgebers, der Empfang einer Nachricht oder eines Verbindungsaufbauwunsches sowie Benutzereingaben auf STDIN.

Das Flussdiagramm in Abb. 4.3 stellt den Ablauf bei Programmstart und die Endlosschleife in der Hauptroutine dar: Nach der Bearbeitung der Kommandozeilenargumente (siehe Anhang A.1) und der Initialisierung des eigenen Knotens durch die Funktion `init_mynode()` geht das Programm in den ereignisgesteuerten Modus über.

Das in [20] beschrieben Chord-Protokoll basiert auf iterativen Lookups: Der Absender einer Nachricht sendet solange iterativ RPCs an Knoten des Pfades, die er bereits durch vorherige RPCs ermittelt hat, bis er den Zielknoten gefunden hat. Dieser Implementierung liegt allerdings ein rekursiver Ansatz zugrunde: Die Kommunikation zwischen den Knoten erfolgt über permanent bestehende TCP-Verbindungen.

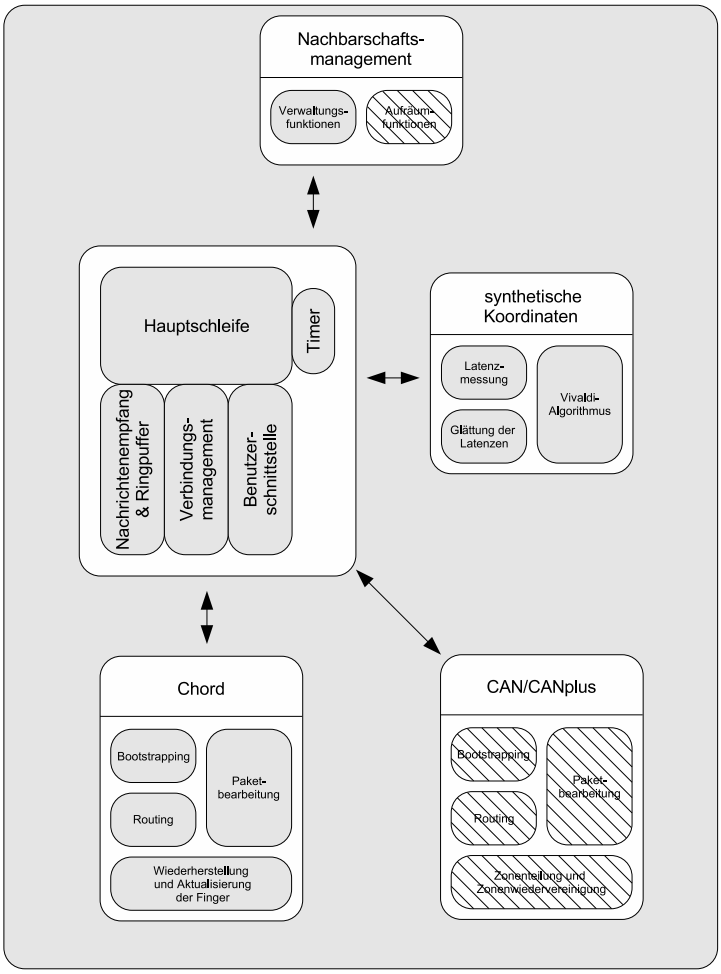


Abbildung 4.1: Module der Implementierung. Schraffierte Komponenten wurden nicht implementiert.

can.[ch], can_msgs.[ch]	CAN-Protokoll
chord.[ch], chord_msgs.[ch]	Chord-Protokoll
general.h	Einbinden der benötigten Header-Files
hon.[ch]	Hauptschleife, Main-Methode
latency.[ch]	Latenzmessung und -glättung
miniapp.[ch]	Einfache Benutzerschnittstelle
msgs.[ch]	Nachrichtenempfang, Pufferung und Bearbeitung
neighbors.[ch]	Management der Nachbarschaftstabelle
timer.[ch]	Timer und Timeout-Definitionen
utils.[ch]	Hilfsfunktionen
vivaldi.[ch]	Vivaldi-Algorithmus

Abbildung 4.2: Dateiliste

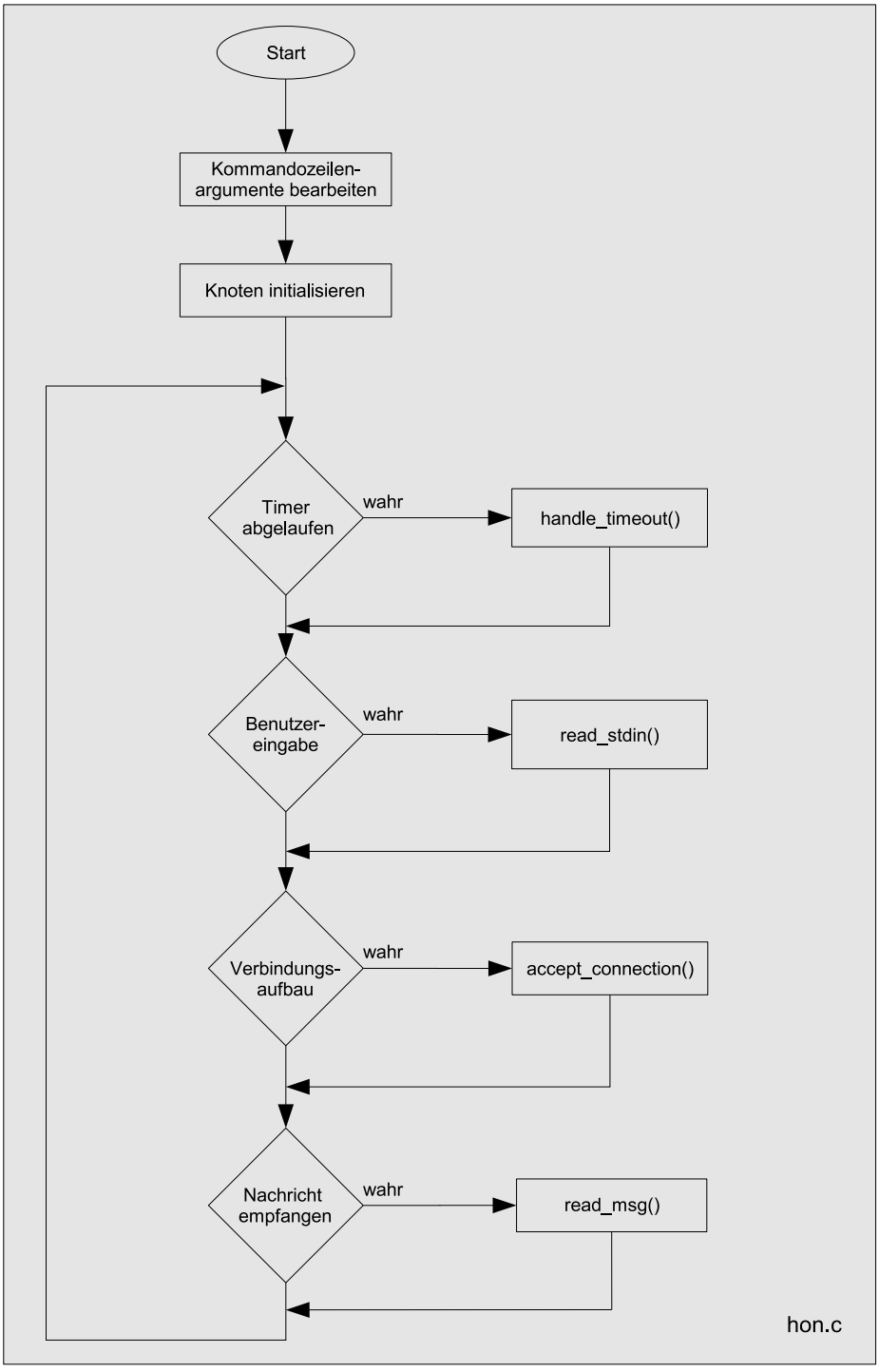


Abbildung 4.3: Flussdiagramm Hauptroutine

Nachrichten werden rekursiven entlang des Routingpfades von Knoten zu Knoten weitergeleitet. Die Latenz beim Routing kann so laut [6] um ca. 40 Prozent reduziert werden.

Trotz zuverlässiger TCP-Verbindungen kann eine zuverlässige Zustellung der gerouteten Nachrichten nicht garantiert werden. Es kann z.B. passieren, dass ein Knoten eine Nachricht nicht weiterleiten kann, da der in Frage kommenden Nachbar nicht erreichbar ist. In einer solchen Situation wird die Anfrage verworfen. Ebenso verhält sich ein Knoten, wenn die TTL einer Nachricht (siehe Abschnitt 4.4.2) abgelaufen ist. In solchen Fällen ist es Aufgabe der darüberliegenden Anwendung, gegebenenfalls Sendewiederholungen zu initiieren.

4.3 Kernmodul und Nachbarschaftsmanagement

Die Datei `timer.c` enthält Methoden zur Verwaltung mehrere Timer verschiedenen Typs, die gleichzeit gesetzt werden können. Für die in Abschnitt 4.4.3 beschriebenen Aktualisierungs- und Wiederherstellungsmechanismen werden jeweils verschiedene Timer-Typen verwendet. Beim Hinzufügen neuer Timer werden alte Timer des selben Typs überschrieben. Nach Ablauf eines oder mehrerer Timer wird die Funktion `handle_timeout()` aufgerufen, je nach Timer-Typ werden dann unterschiedliche Aktionen durchgeführt.

Eine kleine Benutzerschnittstelle wird in `miniapp.c` bereitgestellt. Diese steht beispielhaft für eine mögliche, auf das HON aufsetzende Applikation. Momentan wird hier ein Lookup auf Chord-Ebene und das Versenden von Daten an einen angegebenen Schlüssel unterstützt. Darüberhinaus kann hier der Zustand eines Knotens[1] angezeigt werden.

Das Modul „Nachbarschaftsmanagement“ stellt die Funktionen zur Bearbeitung der Nachbarschaftstabelle bereit. Die Tabelle besteht aus einem Array aus Zeigern auf die Struktur `struct node`. Diese enthält alle benötigten Informationen über einen Nachbarn:

```
struct node {
  u_int8_t            n_type;
  u_int32_t           transaddr_ip;
  u_int16_t           transaddr_port;
  int                 sockfd;
  int                 chord_finger_id;
  chord_id            chord_id;
  viv_point           viv_coord;
  float               viv_rel_err;
  struct lat          lat_filter;
  can_point           can_coord;
  struct can_zone     can_zone;
  struct timeval      can_last_changed;
  struct ib           msg_buffer;
};
```

[1] Information über dessen Nachbarn, die eigenen Chord-ID und die CAN-Koordinaten

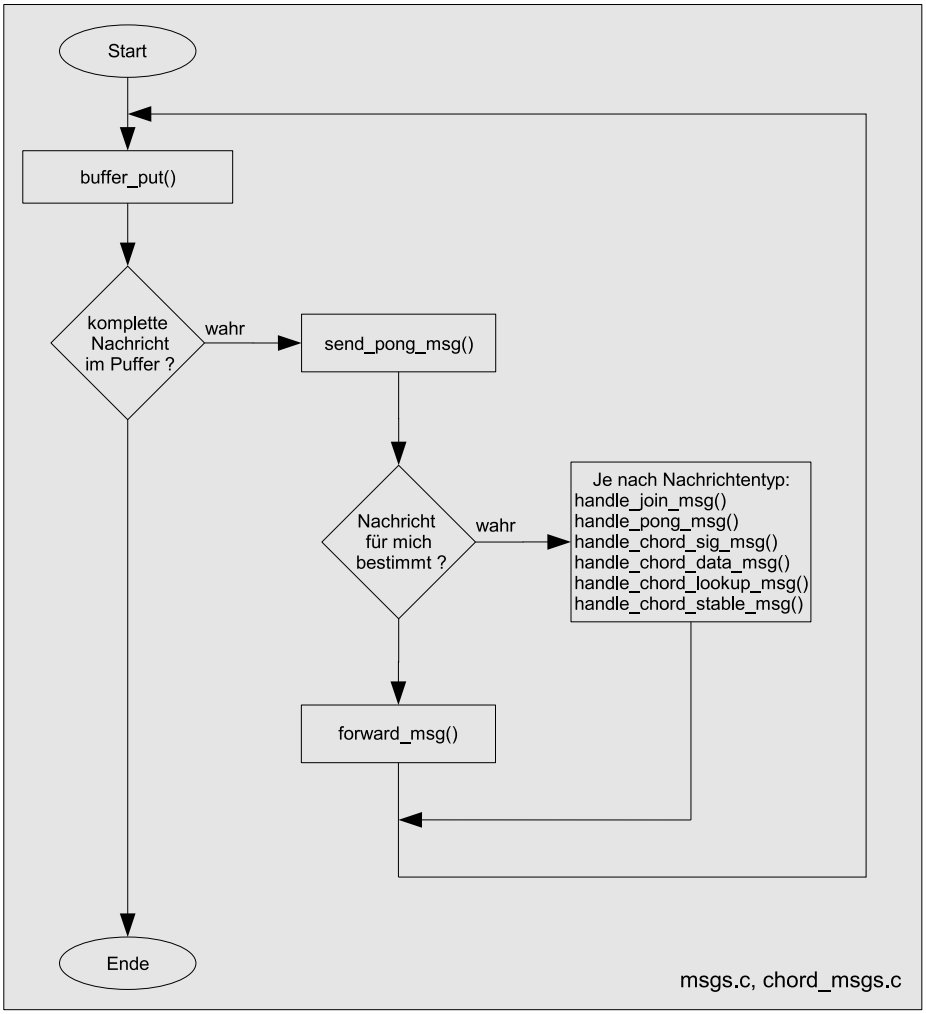


Abbildung 4.4: Flussdiagramm Nachrichtenempfang

`n_type` enthält den Typ dieses Nachbarn. Hier wird vermerkt, ob es sich um eine durch Chord, CAN oder CANplus initiierte Verbindung handelt. Auf die Bedeutung der weiteren Bestandteile wird im Laufe dieses Kapitels noch eingegangen werden.

Für jede bestehende Verbindung verwaltet der Knoten einen separaten Ringpuffer (`struct ib`) für eingehende Nachrichten. Zwei Zeiger d_b und d_e markieren Beginn und Ende der Daten im Puffer.

Das Flussdiagramm in Abb. 4.4 stellt den Programmablauf bei Empfang einer Nachricht dar: Zunächst werden die Daten vom TCP-Empfangspuffer des Sockets in den entsprechenden Nachrichtenpuffer kopiert. Nun werden solange Nachrichten aus dem Nachrichtenpuffer gelesen, bis dieser entweder leer ist oder keine vollständige Nachricht mehr enthält. Auf jede Nachricht, außer auf Pong-Nachrichten selbst, wird zunächst mit einer Pong-Nachricht (siehe Abschnitt 4.4.4) geantwortet. Daraufhin wird überprüft, ob der in der Nachricht angegebene Schlüssel in den vom eigenen Knoten verwalteten Wertebereich fällt. Ist dies der Fall, wird die Nachricht je nach Typ von der entsprechenden Funktion verarbeitet. Andernfalls wird die Nachricht weiter in Richtung Ziel geroutet.

4.4 Chord

Im Folgenden wird die Implementierung des Chord-Protokolls erläutert.

4.4.1 Routing

Für die Chord-ID eines Knotens wird ein vorzeichenloser 64-Bit Integer `u_int64_t` verwendet. Die ID wird bei der Initialisierung des Knotens gleichverteilt zufällig gewählt.

Anhand der Funktion `partof_my_node()` kann ein Knoten feststellen, ob ein Schlüssel K in dem von ihm verwalteten Wertebereich[2] liegt. Da der Schlüsselraum ringförmig ist, werden vor der Anwendung von Vergleichsfunktionen die zu vergleichenden IDs transformiert (`chid_base_trans()`). Bei Additionen und Subtraktionen wird durch den vorzeichenlosen Datentyp implizit eine Modulo-Berechnung durchgeführt.

Mit der Funktion `chord_routing_next_hop()` ermittelt ein Knoten, an welchen seiner Nachbarn er eine Nachricht weiterleiten muss: Es wird der Vorgänger von K, also der Nachbar, dessen ID numerisch am nächsten „vor“ K im Chord-Ring liegt.

4.4.2 Bootstrapping

Möchte ein neuer Knoten NK dem Chord-Ring beitreten, muss er mindestens einen Knoten kennen, der bereits Teil des Systems ist („entry point“, EP). In einer Datei[3] speichert jeder Knoten Netzwerkadressen von bekannten EPs. Eine Beitritts-Anfrage `JOIN_NEWREQ`, mit der Chord-ID von NK wird von dem EP stellvertretend in den Chord-Ring weitergeleitet. EP leitet schließlich die Antwort `JOIN_NEWRES`, die die Netzwerkadresse und die Chord-ID des neuen Nachfolgers ZK enthält, an NK weiter. Abb. 4.5 verdeutlicht dies und stellt den kompletten Ablauf und die verwendeten Join-Nachrichtentypen dar.

Der neue Knoten NK kennt nun seinen direkten Nachfolger ZK, die Verbindung zum EP wird nicht mehr benötigt und kann abgebaut werden.

Unter Verwendung des in Kapitel 2 beschriebenen Stabilisierungsmechanismus erfährt NK seinen direkten Vorgänger VK. Bevor VK nicht ebenfalls die Stabilisierungsroutine ausgeführt und so seinem neuen Nachfolger NK mitgeteilt bekommen hat, ist der Ring an dieser Stelle inkonsistent. Wird in dieser Situation ein Lookup nach einem Schlüssel gesendet, der nun von NK verwaltet wird, kreist diese Nachricht solange, bis die Konsistenz des Rings wieder hergestellt ist. Denn VK leitet die Suchanfrage an ZK weiter, der vor dem Knotenbeitritt für diese ID zuständig war. ZK leitet die Anfrage ebenfalls weiter, da er mittlerweile nicht mehr für die ID zuständig ist.

Um möglichst schnell einen konsistenten Ringzustand herbeizuführen, muss die Stabilisierungsmechanismus in kurzen Abständen ausgeführt werden. Außerdem wird durch Einführung einer TTL begrenzt, wie oft eine Nachricht weitergeleitet werden darf.

[2] Jeder Knoten verwaltet alle IDs, die zwischen der ID seines direkten Vorgängers und seiner eigenen Chord-ID liegen.

[3] Standardmäßig wird hier die Datei `hints.lst` verwendet.

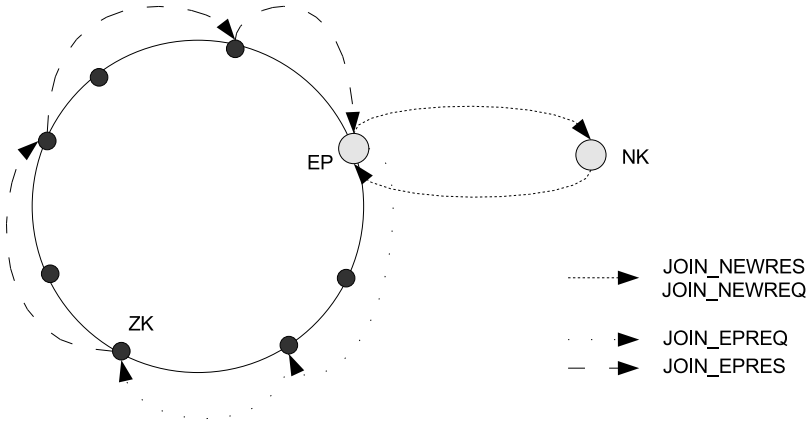


Abbildung 4.5: Beitritt des Knotens NK

4.4.3 Signalisierung

Im Folgenden werden kurz die für die jeweiligen Aktualisierungs- und Wiederherstellungsmechanismen zuständigen Funktionen in `chord.c` bzw. `chord_msgs.c` vorgestellt. Auf eine eine detaillierte Erläuterung wird verzichtet, da sie im Wesentlichen die in Abschnitt 2.1.1 beschriebenen Funktionalitäten realisieren.

Die Aktualisierung des direkten Nachfolgers und Vorgängers wird durch die Funktion `stabilize_req()` nach Ablauf eines Timers der Länge `STAB_INT` angestoßen.

Die Länge der Nachfolgerliste ist in `SUCCLIST_MAX` definiert. `SUCCLIST_INT` legt fest, wie häufig die Aktualisierung stattfinden soll. Für jeden Nachfolger wird dessen Netzwerkadresse und Chord-ID gespeichert.

Die Einträge der Fingertabelle werden alle `FINGER_INT` Sekunden aktualisiert. Die in `struct node` enthaltene Variable `chord_finger_id` gibt an, den wievielten Finger ein Nachbar darstellt. Beim Aktualisieren des i-ten Fingers wird zunächst geprüft, ob es bisher einen Fingereintrag $j \leq i$ gibt, der die Chord-ID des i-ten Fingers einschließt. Ist dies nicht der Fall, wird eine Suchanfrage nach der Chord-ID des gewünschten Fingereintrages initiiert.

Nach einer Wartezeit `FINGER_TIMEOUT` wird der i+1-te Fingereintrag gesucht. Wurde der letzte Finger, hier $i = 63$, bearbeitet, wird der Timer wieder auf `FINGER_INT` gesetzt. Veraltete Fingereinträge werden nicht aus der Nachbarschaftstabelle gelöscht. `chord_finger_id` wird anstelle dessen auf -1 gesetzt. So können diese Verbindungen beim Routing weiterhin verwendet werden.

Periodisch (`HELLO_INT`) wird auch überprüft, ob Nachbarn noch erreichbar sind. Eine so genannte Hello-Anfrage wird an den zu überprüfenden Knoten gesendet. Wird nach `HELLO_TIMEOUT` Sekunden keine Antwort von diesem Knoten erhalten, wird die Verbindung zu diesem Knoten abgebaut. Falls es sich dabei um den direkten Nachfolger handelt, wird der nächste Knoten der Nachfolgerliste als neuer direkter Nachbar herangezogen.

Wird im Rahmen der hier beschriebenen Mechanismen eine neue Verbindung zu einem Knoten aufgebaut, so wird an diesen anschließend eine Nachricht mit den eigenen Knotendaten `send_mynode_info()` gesendet. Auf diese Weise können Verbindungen bidirektional benutzt werden.

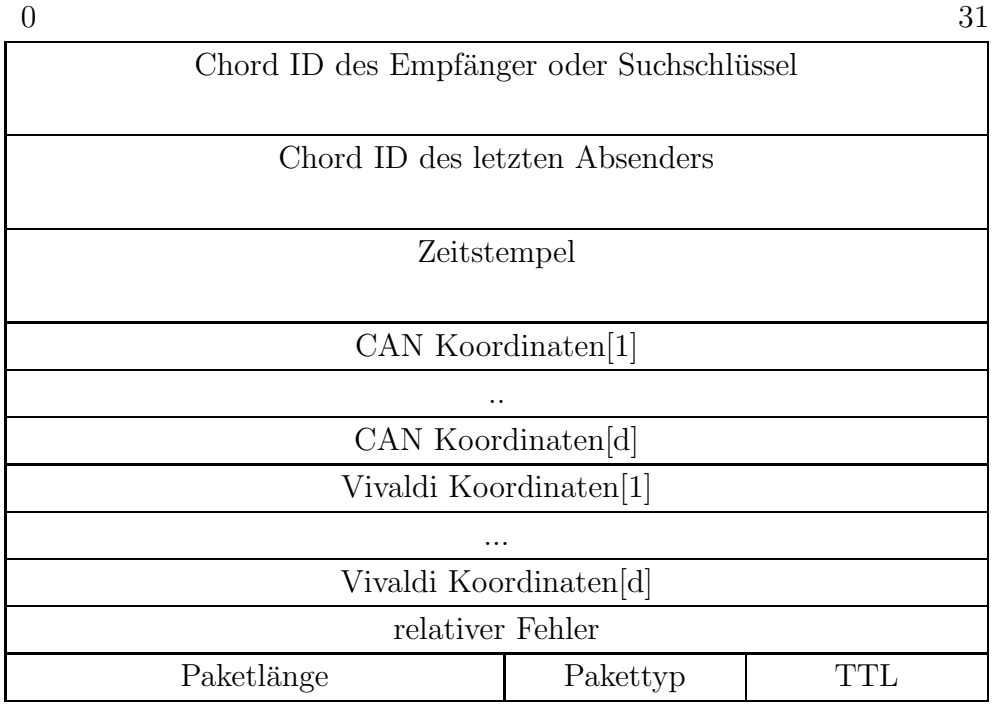


Abbildung 4.6: Format des allgemeinen Nachrichtenkopfes

4.4.4 Nachrichtenformate

Alle Nachrichten haben einen identischen Nachrichtenkopf. Dieser setzt sich wie in Abb. 4.6 dargestellt zusammensetzt: Die Chord-ID des Empfängers, bzw. ein Suchschlüssel werden für das Routing benötigt. Die Chord-ID des letzten Absenders, also des Knotens, der die Nachricht zuletzt weitergeleitet hat, und der Zeitstempel werden für das Erstellen von Pong-Nachrichten verwendet. Der Vivaldi-Algorithmus greift auf die mitgesandten CAN- und Vivaldi-Koordinaten und den relativen Fehler zurück (siehe Abschnitt 4.5.2). Die Größe des Nachrichtenkopfes hängt stark mit der Anzahl der hierfür verwendeten Dimensionen zusammen. Die Gesamtlänge des Pakets, der Pakettyp und schließlich eine TTL sind außerdem enthalten. Anhand des Pakettyps wird zwischen sechs verschiedenen Nachrichtentypen unterschieden werden, die jeweils einen anderen Aufbau aufweisen:

Der Aufbau der *Join-Nachrichten* ist in Abb. 4.7 dargestellt. „Join-Typ“ kann die in Abb. 4.5 vorgestellten Typen enthalten. Je nach Typ werden Chord-ID und Netzwerkadresse verschiedener Knoten übertragen. Die „Bootstrap-ID“ wird benötigt, damit ein EP mehrere Knotenbeitritte gleichzeitig verwalten kann.

Eine *Pong-Nachricht* (Abb. 4.8) besteht aus dem von der Referenznachricht kopierten Zeitstempel und dem „Pong-Typ“, der angibt, auf welchen Nachrichtentyp hier mit einer Pong-Nachricht reagiert wurde.

Signalisierungsnachrichten von Chord haben den in Abb. 4.9 dargestellten Aufbau. Sie enthalten die Netzwerkadresse und Chord-ID eines Knotens. „Sig-Typ“ spezifiziert den genauen Typ der Signalisierungsnachrichten.

Abb. 4.10 zeigt den Aufbau der *Lookup-Nachrichten*. Neben der gesuchten ID ist eine weitere Chord-ID und die dazugehörige Netzwerkadresse enthalten. Ein Lookup-Anfrage wird nach üblichen Chord-Regeln zum Zielknoten geleitet, voraufhin eine

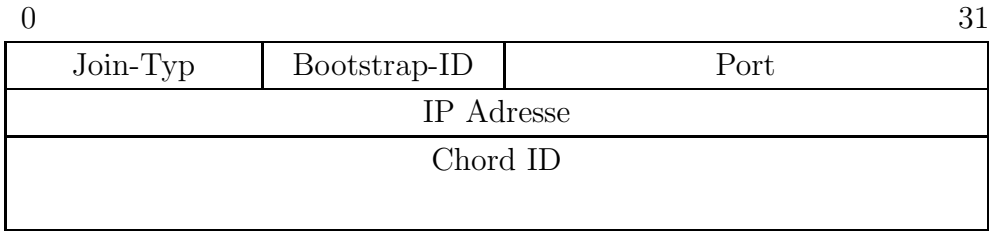


Abbildung 4.7: Aufbau der Join-Nachricht

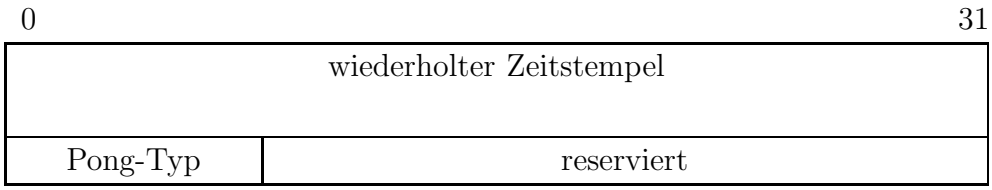


Abbildung 4.8: Aufbau der Pong-Nachricht

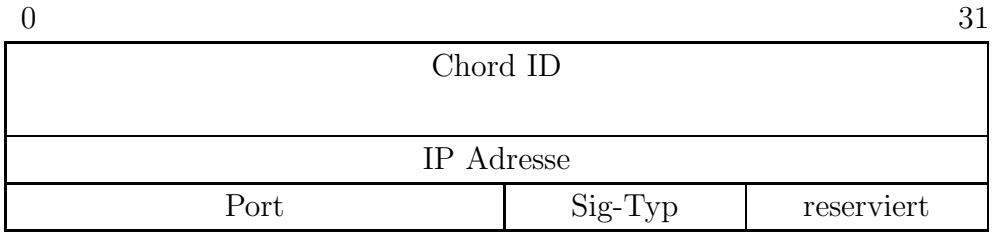


Abbildung 4.9: Aufbau der Signalisierungsnachrichten

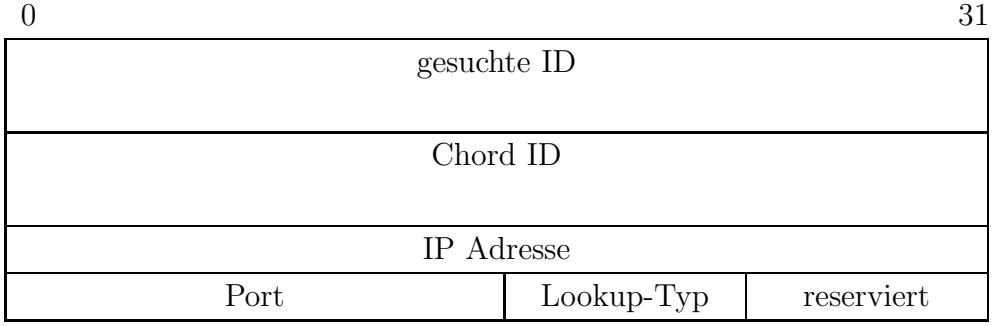


Abbildung 4.10: Aufbau der Lookup-Nachricht

Lookup-Antwort mit Informationen über diesen Knoten im Chord-Ring zum anfragenden Knoten geroutet.

Eine *Datennachricht* besteht aus Daten der maximalen Länge `MAX_DATA_LEN`. Der genaue Aufbau dieser Nachricht ist von der darüberliegenden Applikation abhängig. Erreicht eine solche Nachricht den Zielknoten, wird sie an die entsprechende Anwendung übergeben.

4.5 Ermittlung der Koordinaten

Das Modul „synthetische Koordinaten“ besteht aus Komponenten zur Ermittlung der Latenzen und der Komponente Vivaldi. Diese passt ausgehend von den gemessenen Latenzen die Koordinaten eines Knotens an. Die Implementierung dieser beiden Komponenten wird hier ausführlich erläutert. Zum besseren Verständnis ist an einigen Stellen Pseudocode eingefügt. Die Bezeichnung dabei verwendeten Variablen orientiert sich weitestgehend am Originalquellcode.

4.5.1 Bestimmung der Latenzen

Um die Latenz zwischen zwei kommunizierenden Knoten messen zu können, wird im allgemeinen Nachrichtenkopf ein Zeitstempel mitgeschickt. Wird eine Nachricht über mehrere Hops im Overlay geroutet, so wird auf dem umgekehrtem Pfad immer abschnittsweise eine Pong-Nachricht gesendet. Abb. 4.11 verdeutlicht diese Vorgehensweise. Pong-Nachrichten werden von Knoten 1 nach Knoten 6 und von Knoten 4 nach Knoten 1 gesendet werden.

Beim Senden und Forwarden von Nachrichten passt jeder Knoten den allgemeinen Nachrichtenkopf der Nachricht an, indem er seine aktuelle Systemzeit im Zeitstempel vermerkt und als Absender-ID die eigene Chord-ID einträgt.

Erhält nun ein Knoten eine Pong-Nachricht, so wertet er diese mit der Funktion `handle_pong_msg()` aus: Die Round-Trip-Time (RTT) des Pakets und die Latenz $l = \frac{1}{2}RTT$ werden berechnet, anschließend wird `vivaldi_update()` ausgeführt.

Bei ersten Testmessungen stellte sich heraus, dass die so gemessenen Latenzen dauerhaft hohe Schwankungen aufweisen. Dies ist darauf zurückzuführen, dass nicht nur Netzwerklatenzen, sondern vor allem auch betriebssystembedingte Scheduler-Verzögerungen gemessen werden. Im User-Space können auf TCP-Ebene also nur so genannte Ende-zu-Ende-Latenzen gemessen werden. Somit fließt auch die Auslastung der Knoten in die Koordinaten mit ein, was durchaus wünschenswert ist.

Durch Glätten der Messwerte sollen die so entstandenen Schwankungen herausgefiltert und die gemessenen Werte in Durchschnittswerte überführt werden, die diese Ende-zu-Ende'-Latenz repräsentieren.

Die Funktion `latency_filter()` ist eine Hüllfunktion für zur Auswahl stehende Latenzfilter. Welcher Filter verwendet werden soll, kann über Kommandozeilenargumente (siehe Anhang A.1) spezifiziert werden. In der in `struct node` enthaltenen Struktur `struct lat` wird der so berechnete Durchschnitt gespeichert.

Es stehen zwei Filtermethoden zur Verfügung: *Gleitende Durchschnitte* (GD) und *Exponentielles Glätten*[4] (EG). Bei der Bildung eines Mittelwertes beziehen sich die

[4]Exponentielles Glätten kann als Sonderform der Gleitenden Durchschnitte verstanden werden. Aus diesem Grund wird dieser Ansatz oftmals auch *gewichtete gleitende Durchschnitte* genannt

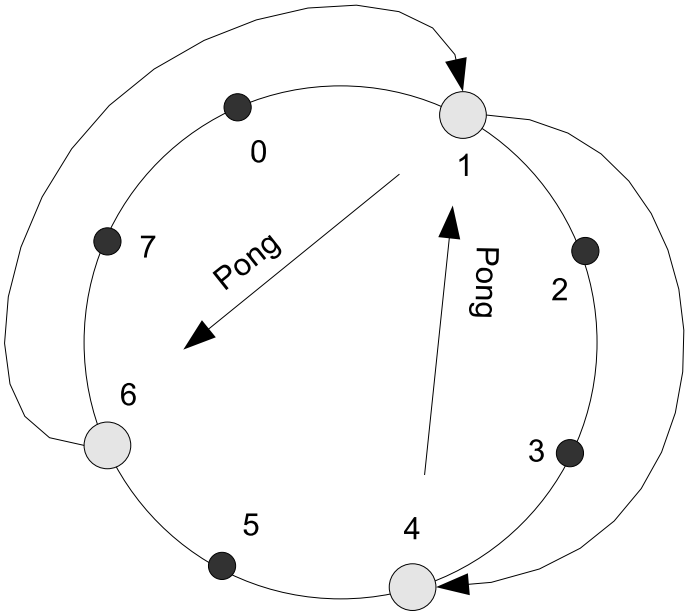


Abbildung 4.11: Abschnittsweise werden Pong-Nachrichten gesendet.

GD nur auf die jeweils letzten l Messwerte, über diese wird der arithmetische Mittelwert gebildet . Sehr alte Werte gehen also nicht in die Berechnung mit ein. Beim EG wird der Gewichtungsfaktor α verwendet, um jüngere Messwerten bei der Durchschnittsbildung relativ stärker zu berücksichtigen als ältere.

4.5.2 Vivaldi

Um aus nur wenigen Messungen die Latenzen zwischen beliebigen Knotenpaaren vorhersagen zu können, wurde auf Vivaldi, einen dezentralen Algorithmus zur Berechnung synthetischer Koordinaten (siehe Kapitel 2), zurückgegriffen. Die für Vivaldi benötigten Daten werden im allgemeinen Nachrichtenkopf mitgesendet.

4.5.2.1 Aktualisierung der Vivaldi-Koordinaten

Anfangs setzt jeder Knoten seine Koordinaten auf den Nullpunkt. Nach jeder Latenzmessung wird nun die Funktion `update_vivaldi()` aufgerufen. Jeder Knoten passt dabei ausschließlich die eigenen Koordinaten an.

Abb. 4.12 stellt den Ablauf der Koordinatenänderung anhand des Pseudocodes der Funktion `update_vivaldi()` dar: Zunächst wird der normalisierte Vektor $\vec{v}$ von $\vec{p}$ nach $\vec{q}$ berechnet. Dieser bestimmt die Richtung der Verschiebung. *dev* bestimmt, wieweit p verschoben werden müsste. Die Verschiebung wird um den Faktor `delta` gedämpft. Abschließend wird überprüft, ob neben den Vivaldi-Koordinaten auch die eigenen CAN-Koordinaten anzupassen sind.

Abb. 4.13 zeigt die Verschiebung eines Knotens in einem 2-dimensionalen Euklidischen Koordinatensystem. Hierbei wurde `delta=1` angenommen. Knoten 1 ändert seine Koordinaten von p auf $p*$, damit der Abstand zwischen den Koordinaten der beiden Knoten der gemessene Latenz d^* entspricht.

```
vivaldi_update(p, q, l) {

    /* normierter Vektor von p nach q */
    v = vector_between(p,q);

    /* Abweichung zwischen gemessener Latenz (lat)
     * und berechneter Distanz  */
    dev = vector_distance(p,q) - lat;

    /* Daempfungsfaktor anpassen */
    delta = adapt_delta();

    /* Verschiebung der eigenen Koordinaten */
    p = p + v * dev * delta;

    /* gegf. CAN-Koordinaten anpassen */
    change_can_coords();

}
```

Abbildung 4.12: Pseudocode des Vivaldi-Algorithmus. p und q sind jeweils die Koordinatenvektoren des eigenen Knotens und des Kommunikationspartners.

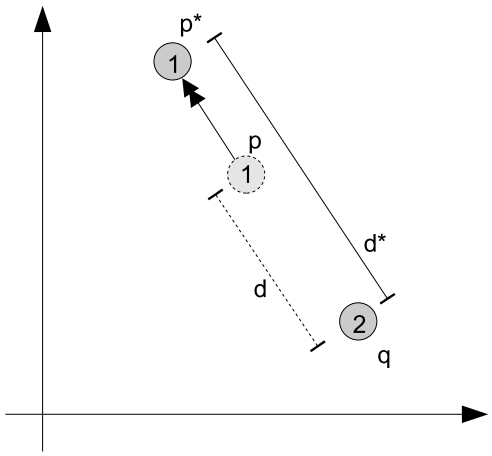


Abbildung 4.13: Funktionsweise des Vivaldi-Algorithmus. Knoten 1 mit den Koordinaten p und p^* und Knoten 2 mit den Koordinaten q. d und d^* sind die Latenzen zwischen q und p, bzw. q und p^*.

4.5.2.2 Anpassung an die Oberfläche eines Torus

In [3] werden verschiedener Koordinatenräume und deren Anpassungsfähigkeit an die Topologie des Internet diskutiert. Entgegen der dort favorisierten Variante des „Höhenvektor-Modells“, wird hier die Oberfläche eines d-dimensionalen Torus verwendet.

Für die Distanz d und die maximale Distanz d_{max} zweier Punkte gilt also:

$$d = \sqrt{\sum_{i=1}^{n} \min\left[(p_i - q_i) \bmod l_i, (q_i - p_i) \bmod l_i\right]^2} \tag{4.1}$$

$$d_{max} = \sqrt{n * (\frac{1}{2} * l_i)^2} \tag{4.2}$$

Dabei ist l_i die Kantenlänge der Torusoberfläche der jeweiligen Dimension D_i und es gilt $l_i = l_j$, $\forall i, j$ und $i, j = 1, \ldots, n$. Anhand der in 4.1 vorgestellten Metrik wird in `vector_distance()` die Distanz zwischen zwei Punkten berechnet.

Die Kantenlängen l_i ist so gewählt, dass eine hohe Auflösung der berechneten Latenzen und große Distanzen von mindestens 10 Sekunden zwischen zwei Punkten problemlos dargestellt werden können. Für alle Komponenten des Koordinatenvektors wird aus diesem Grund ein 32-Bit Integer verwendet.

Um den Vektor $\vec{v}$, der von $\vec{p}$ nach $\vec{q}$ zeigt, auf der Torusoberfläche zu berechnen, muss hier in jeder Dimension der kürzeste Abstand zwischen p_i und q_i berechnet und in einer Fallunterscheidung das Vorzeichen bestimmt werden. Abb. 4.14 zeigt den zugehörigen Pseudocode. Die Funktion `double_modulo()` berechnet den übergebenen Wert modulo `CAN_MAXDIM`.

4.5.2.3 Dämpfung der Koordinatenverschiebung

Die Verschiebung der Koordinaten wird um einen Faktor `delta` gedämpft. [3] und [2] führen verschiedene Ansätze hierfür auf: Im einfachsten Fall wählt man einen konstanten Dämpfungsfaktor $\delta \in [0, 1]$. Alternativ kann man auf einen über die Zeit abnehmenden Faktor zurückgreifen, der mit jedem Aufruf der Vivaldi Update-Methode um eine Schrittweite bishin zu einem Minimalwert dekrementiert wird.

Vielversprechender ist ein Ansatz, bei dem der Dämpfungsfaktor an die Güte der Koordinaten angepasst wird. Wie in [3] beschrieben hängt δ vom Verhältnis des lokalen relativen Fehlers[5] und des relativen Fehlers des Kommunikationspartners ab. Koordinaten von Knoten, die einen hohen relativen Fehler aufweisen, sind im Gesamtsystem höchstwahrscheinlich (noch) nicht so gut positioniert und sollten deshalb bevorzugt angepasst werden. Dadurch soll ein Oszillieren bereits gut angepasster Koordinaten verhindert werden.

Die Funktion `adapt_delta()` ist als Hüllfunktion für verschiedene Dämpfungsansätze gedacht. Bisher wurde nur der adaptive Ansatz in `vivaldi_rel_err()` implementiert, Abb. 4.15 enthält den Pseudocode. Der lokale relative Fehler wird bei

[5] Der relative Fehler berechnet sich wie folgt: $e_{rel} = \frac{d-l}{l}$, wobei d die aus den Koordinaten berechnete Distanz und l die gemessene Latenz ist.

```
vector_between(p, q, v) {

  forall dimensions i {

    if (double_modulo(p[i]-q[i]) >
        double_modulo(p[i]-q[i])) {

      if(p[i] > q[i])
      v[i] = -1.0 * double_modulo(p[i]-q[i]);

      else
      v[i] = -1.0 * double_modulo(p[i]-q[i]);

    } else {

      if(p[i] > q[i])
      v[i] = double_modulo(q[i]-q[i]);

      else
      v[i] =  double_modulo(p[i]-q[i]);
    }
  }

  v = v/vector_length(v);

}
```

Abbildung 4.14: Berechnung des normalisierten Vektors $\vec{v}$ von $\vec{p}$ nach $\vec{q}$ auf der Torusoberfläche. Dabei müssen vier Fälle unterschieden werden.

jeder Messung neu bestimmt und geht exponentiell geglättet in die Historie der alten relativen Fehler ein. Anhand der Parameter `c_err` und `c_delta` kann das Gewicht der aktuellen Messung bestimmt, bzw. `delta` beeinflusst werden.

4.5.2.4 Anpassung der CAN-Koordinaten

Vivaldi-Koordinaten werden bei jeder Latenzmessung sofort angepasst. Das Ändern der CAN-Koordinaten erfordert möglicherweise einen Zonenwechsel und damit das Wiedervereinigen eines Zonenbereichs im CAN- und im CANplus-Subsystem. Allzuhäufige Änderungen der CAN-Koordinaten, vor allem wenn diese nur durch kurzzeitiger Schwankungen der Latenzen hervorgerufen werden, sind also aus Performanzgründen zu vermeiden.

Deshalb werden die CAN-Koordinaten nur unter Einhaltung folgender Bedingung angepasst: Seit der letzten Anpassung müssen mindestens `CAN_CHANGE_TIME` Sekunden verstrichen sein. Außerdem muss die Distanz zwischen den neuen und den alten CAN-Koordinaten mindestens `CAN_CHANGE_DIST` Mikrosekunden betragen.

Nach jeder Anpassung der Vivaldi-Koordinaten wird `can_change_coords()` aufgerufen. Hier werden die CAN-Koordinaten geprüft und gegebenenfalls angepasst.

```
vivaldi_rel_err() {

  /*Verhaeltnis der relativen Fehler
   *e_l: eigener relativer Fehler
   *e_r: relativer Fehler des Kommunikationspartners
   */
   w = e_l / (e_l + e_r);

  /*aktuellen relativen Fehler berechnen */
   e_neu = |dist - lat|/lat;

   /* exponentielles Glaetten */
   e_l = e_neu * c_err * w + e_l * (1 - c_err * w);

  /* delta berechnen */
  delta = c_delta * e_l;

}
```

Abbildung 4.15: Adaptiver Ansatz zur Bestimmung des Dämpfungsfaktors

4.6 CAN/CANplus

Die Anzahl der für CAN, CANplus und Vivaldi verwendeten Dimensionen kann durch `CAN_DIMS` festgelegt werden. `CAN_MAXDIM` gibt den Wertebereich jeder Komponente des Koordinatenvektors an, die Größe des darstellbaren Koordinatenraums kann auf diese Weise begrenzt werden.

Die Komponenten des Moduls „CAN/CANplus“ wurden größtenteils noch nicht implementiert. Die Quelldateien `can.c` und `can_msgc.c` sind für eine zukünftige Implementierung vorgesehen, die sich an der in Kapitel 3 vorgeschlagenen Spezifikation orientieren sollte.

5. Auswertung der Messergebnisse

Die Testläufe wurden durchgeführt, indem die in Kapitel 4 beschriebene Chord-Implementierung auf einigen Knoten des PlanetLab[1] gestartet wurde. Das Versenden und Empfangen der Nachrichten in diesem Chord-Ring wurde mitprotokolliert und anschließend ausgewertet.

Im Folgenden wird die jeweilige Testumgebung vorgestellt und es werden die Testergebnisse präsentiert und erläutert.

5.1 Glättung der gemessenen Latenzen

In diesem Abschnitt wird die Wirksamkeit der vorgestellten Latenzfilter GD und EG untersucht. Hierfür wurden jeweils Latenzen zwischen zwei Knoten im PlanetLab gemessen[2]. Folgende Parameter wurden für die Filtermethoden verwendet:

- EG mit $\alpha = 0.01$ (EG1) und $\alpha = 0.005$ (EG2)
- GD unter Einbeziehung der letzten 50 (GD1), bzw. 100 Messwerte (GD2)

In Abb. 5.1 erkennt man sehr deutlich die starken Schwankungen der gemessenen Latenzen. Kurzzeitig kommt es immer wieder zu recht hohen, aber kurzen Ausreißern. Dies passiert beispielsweise, wenn für kurze Zeit die Scheduling-Latenz besonders hoch ist, oder Warteschlangen im Netz sehr voll sind. Solche Ausreißer sollten sich in den geglätteten Durchschnittswerten möglichst wenig widerspiegeln.

Bei $t \approx 6500s$ kommt es zu solchen Ausreißern. Die Methoden GD1 und GD2 filtern diese am schlechtesten heraus, da hier dem aktuellen Messwert eine höhere Bedeutung beimessen wird. Bei $t \approx 6600s$ und $t \approx 6750s$ kommt es zu großen, aber nur kurz anhaltenden Latenzsprüngen [3]. Man sieht deutlich, dass die Filtermethode GD zu einer Plateaubildung tendiert. Dies lässt sich durch eine größere Invarianz gegenüber

[1] Siehe www.planet-lab.org

[2] Abb. 5.1 und Abb. 5.2 zeigen Ergebnisse der Messungen zwischen Knoten in USA und Singapur, bzw. zwischen Knoten in USA und Indien.

[3] Die gemessenen Latenzen betrugen zum Teil mehrere Sekunden.

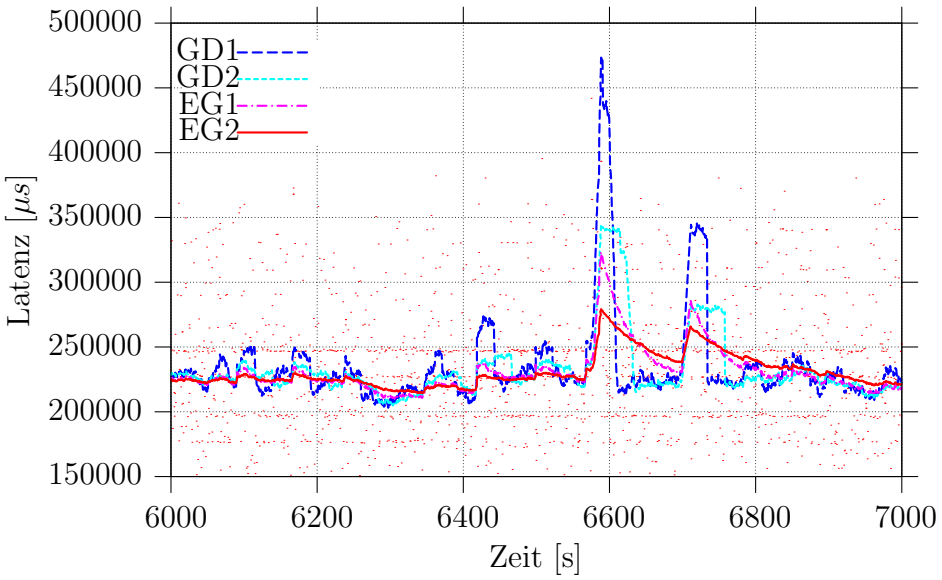


Abbildung 5.1: Glättung der gemessenen Latenzen (Punkte). Ausreißer werden herausgefiltert.

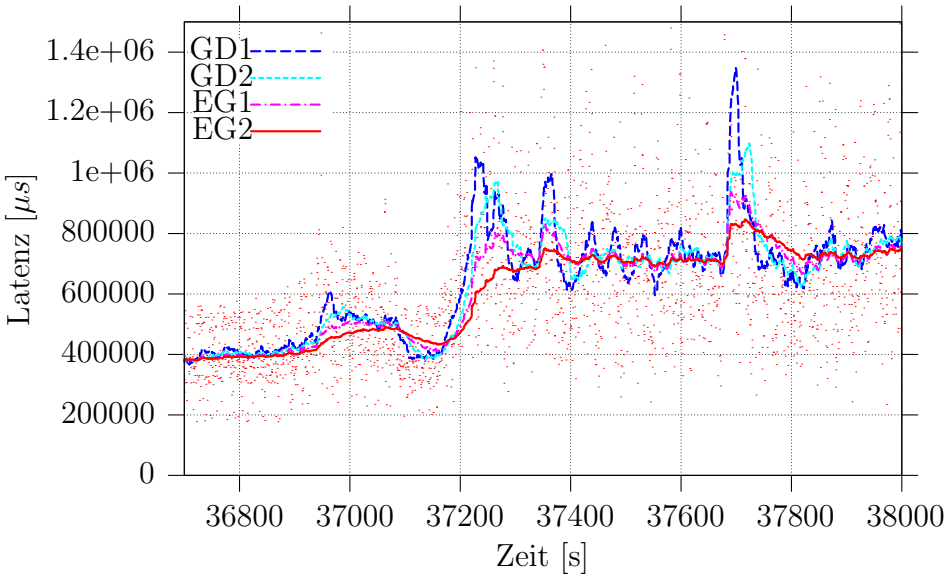


Abbildung 5.2: Glättung der gemessenen Latenzen (Punkte). Anpassung an dauerhaft geänderte Latenzen.

Ausreißern erklären. Die Filtermethode EG reagiert kaum auf schwächere Ausreißer kaum. Extreme Ausreißer bilden anstelle von Plateaus kleine „Haifischflossen“, gut sichtbar bei EG1. Der durch diese Ausreißer gestiegene, geglättete Durchschnitt fällt nur langsam wieder auf seinen vorherigen Wert ab. Plateaubildung ist einerseits wünschenswerter: Durch einen Sprung auf ein neues Plateau kommt es im Gegensatz zu einem langsamen Ansteigen oder Abfallen der Durchschnitte nicht zu so vielen Änderungen der Vivaldi- und vor allem der CAN-Koordinaten. Andererseits ist eine gewisse Resistenz gegenüber kleinen oder kurzzeitigen Ausreißern sehr wichtig.

Ein anderer Fall liegt vor, wenn die Latenz zwischen einem Knotenpaar dauerhaft steigt. Die passiert beispielsweise, wenn das Netz stärker ausgelastet ist oder es zu Routenänderungen in Internet kommt. Die Schwankungen der gemessenen Latenzen finden dann auf einem insgesamt höheren Niveau statt. Dies sollte sich möglichst zeitnah in den geglätteten Werten widerspiegeln.

In Abb. 5.2 ist ein solches Szenario dargestellt. Hier steigt die Latenz bei $t \approx 37000$. Mit einer gewissen Verzögerung folgen die Durchschnittswerte aller Filter dieser Tendenz, wobei die GD schneller reagieren. EG2 reagiert am langsamsten.

Die beiden vorgestellten Filtermethoden weisen verschiedene Eigenschaften auf. Deren Ausprägung hängt stark von der Wahl der Parameter ab. Insgesamt filtert EG2 kleinere Ausreißer am stärksten heraus. Der geglättete Durchschnittswert folgt zwar langsam, aber dennoch für die hier gestellten Anforderungen schnell genug dauerhaft geänderten Latenzen. Für alle weiteren Messungen wurde daher dieser Filter verwendet.

5.2 Zeitliche Entwicklung der Koordinaten

Bei den folgenden Tests bestand das Testumfeld aus 10 Knoten des PlanetLab. Oftmals wurden Tests parallel durchgeführt, indem auf allen Knoten mehrere Prozesse der Chord-Implementierung gestartet wurden. So war es möglich, auf voneinander unabhängigen Chord-Ringen, die alle den gleichen Bedingungen[4] ausgesetzt waren, verschiedene Parameter und Szenarien zu testen und diese anschließend zu vergleichen. Eine mögliche gegenseitige Beeinflussung der Chord-Prozesse kann außer Acht gelassen werden, da die PlanetLab Knoten im Allgemeinen sehr ausgelastet[5] sind und so einige zusätzliche Prozesse kaum ins Gewicht fallen.

Die Auswirkungen des Vivaldi-Parameter c_{err} wurden kurz getestet: Je kleiner dieser Parameter gewählt wird, desto länger werden die (anfänglichen) Konvergenzzeiten der Koordinaten. Wählt man c_{err} zu groß, bekommen einzelne Schwankungen ein zu hohes Gewicht.

Bei allen weiteren Messungen wurde deshalb $c_{err} = 0.1$ gewählt, als Kompromiss zwischen diesen beiden Extremen. Der Vivaldi-Parameter c_{delta} wurde wie in [3] vorgeschlagen auf $c_{delta} = 0.25$ gesetzt.

5.2.1 Kommunikationsstruktur

Um die Bedeutung der Kommunikationsstruktur ermitteln zu können, wurde die zeitliche Entwicklung der Vivaldi-Koordinaten in drei Szenarien getestet:

[4] Auslastung der Knoten und Anbindung ans Internet

[5] Die durchschnittliche Last der Knoten, stichprobenartig auf einigen Knoten mit dem Linuxprogramm `uptime` ermittelt, lag meistens über „4“.

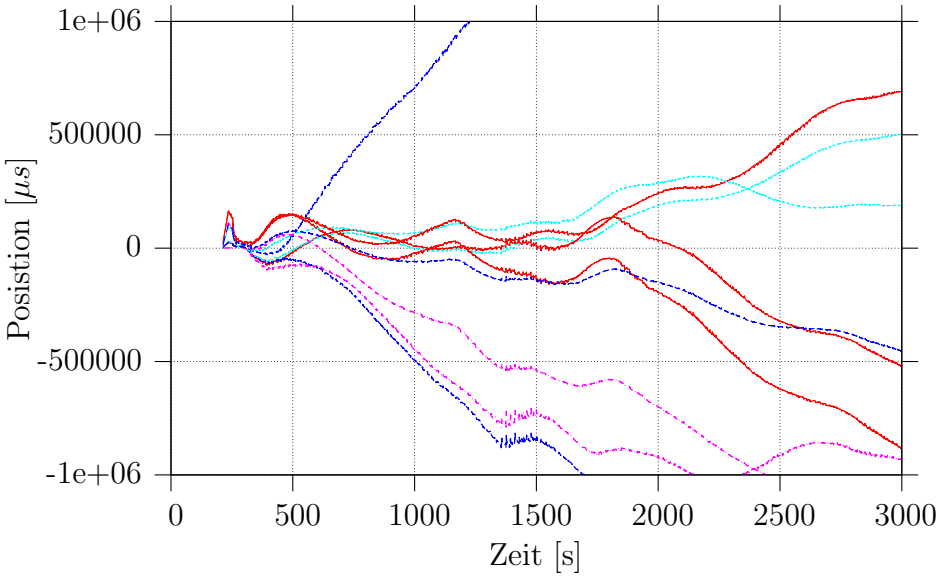


Abbildung 5.3: Entwicklung der Vivaldi-Koordinaten bei einer inhomogenen Kommunikationsstruktur (KS1). Jede Kurve stellt dabei den zeitlichen Verlauf der Koordinaten in einer Dimension dar.

Inhomogene Kommunikationsstruktur (KS1) Es werden ausschließlich Signalisierungsnachrichten verschickt. So kommuniziert jeder Knoten mit einigen wenigen seiner Nachbarn relativ häufig[6], mit den meisten anderen aber eher selten. Auf diese Weise werden die eigenen Vivaldi-Koordinaten bevorzugt an die Koordinaten einiger weniger Knoten angepasst.

Homogene Kommunikationsstruktur (KS2) Hier werden Lookups simuliert, indem jeder Knoten in regelmäßigen Abständen Anfragen an eine gleichverteilt zufällige Chord-ID schickt. So wird mit jedem Nachbarn durchschnittlich gleich oft kommuniziert. Die Anpassung der Vivaldi-Koordinaten findet nur statt, wenn Latenzen durch Pongs auf diese Lookups gemessen werden.

Gemischte Kommunikationsstruktur (KS3) Auch hier werden gleichverteilte Lookups simuliert, die Anpassung der Vivaldi-Koordinaten findet allerdings auch statt, wenn die Latenzen durch Signalisierungsnachrichten gemessen wurden.

In Abb. 5.3, 5.4 und 5.5 sind die Messergebnisse der drei Kommunikationsstrukturen dargestellt. Die Messungen wurde parallel in dem oben oben beschriebenen PlanetLab-Umfeld durchgeführt. Die Grafiken stellen beispielhaft die Bewegung der Koordinaten eines Knotens[7] im Raum dar. Es sind alle Dimensionen über die Zeit geplottet. Gewünscht ist eine möglichst waagerechte Linie für jede Dimension, je steiler die Kurve einer Dimension, desto mehr bewegen sich die Koordinaten des Knotens.

Am vorteilhaftesten entwickeln sich die Vivaldi-Koordinaten unter Verwendung von KS2 (Abb. 5.4). Zwar stabilisieren sie sich nicht vollständig, in einigen Dimensionen

[6]Dies sind insbesondere der direkte Vorgänger und Nachfolger.

[7]Die hier erkenntliche Tendenz kann auch bei den anderen neun Knoten festgestellt werden.

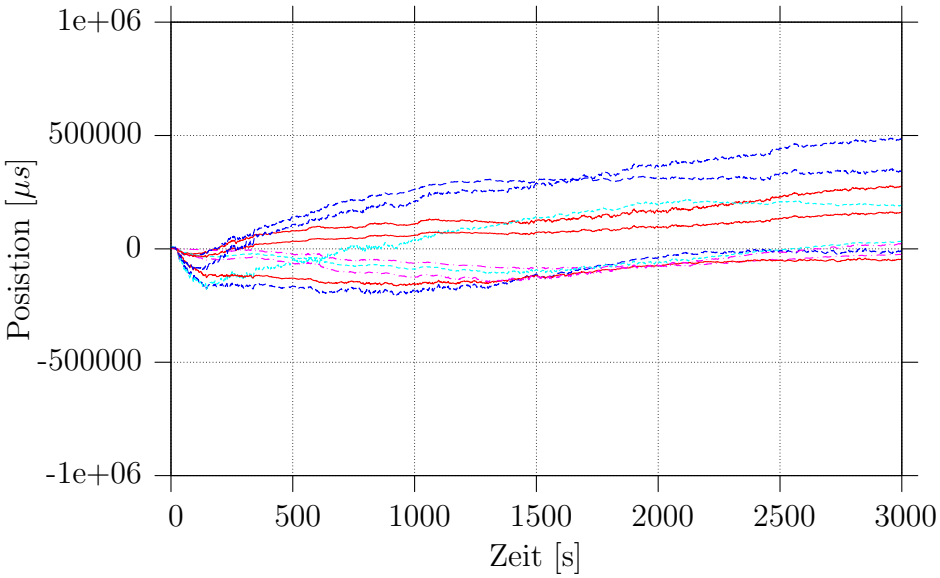


Abbildung 5.4: Entwicklung der Vivaldi-Koordinaten bei einer homogenen Kommunikationsstruktur (KS2)

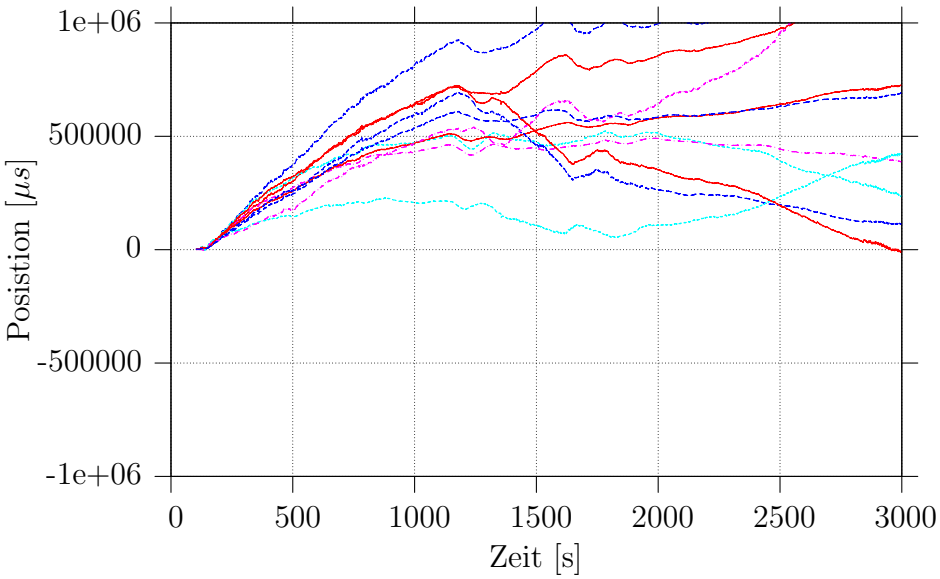


Abbildung 5.5: Entwicklung der Vivaldi-Koordinaten bei einer gemischten Kommunikationsstruktur (KS3)

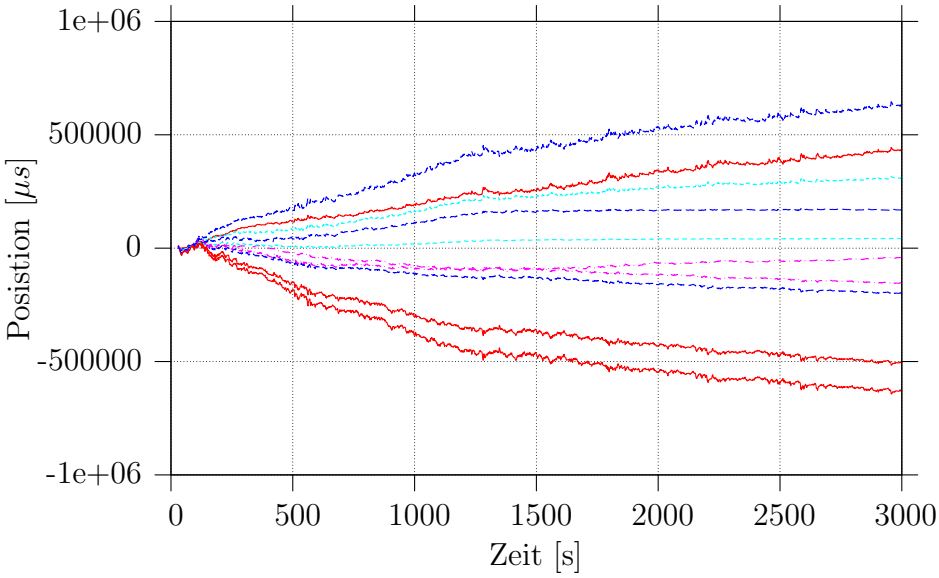


Abbildung 5.6: Vivaldi-Koordinaten mit 10 Dimensionen

sind sie aber sehr stabil. Auffällig ist die wellenförmige Bewegung der Koordinaten bei KS1 (Abb. 5.3) und KS3 (Abb. 5.5) ab $t \approx 1500s$ (25 Minuten). Dies kann durch die häufigere Kommunikation mit den direkten Vorgängern/Nachfolgern erklärt werden. Bei KS1 kommt es im gemessenen Zeitraum von ca. 50 Minuten nicht zu einer erkennbaren Konvergenz der Koordinaten.

5.2.2 Dimensionalität

Unter Verwendung einer homogenen Kommunikationsstruktur KS2 wurde die Auswirkung der Dimensionalität der Vivaldi-Koordinaten getestet. Dem lag die Vermutung zugrunde, dass die Bewegung der Koordinaten von der Dimensionalität aufgrund der unterschiedlichen Freiheitsgrade abhängt. Es wurden dabei Messungen mit 3 und 10 Dimensionen parallel durchgeführt.

Abb. 5.6 und 5.7 zeigen den Verlauf der Vivaldi-Koordinaten. Es ist kein wesentlicher Unterschied hinsichtlich Konvergenz bzw. Bewegung der Koordinaten erkennbar.

5.2.3 Auswirkung auf CAN-Koordinaten

Abb. 5.8 zeigt eine längere Messung mit einer Dauer von 8 Stunden. Auch hier wurde die homogene Kommunikationsstruktur KS2 verwendet. Die Einschwingphase dauert bis $t \approx 1000s$ (16 Minuten). Die Tendenz einer langen Einschwingphase ist auch in den vorherigen Abb. 5.3 bis 5.7 zu erkennen.

Nach einer kurzen stabilen Phase bewegen sich zwischen $2500s \leq t \leq 10000s$ die Koordinaten wieder. Dazwischen liegen immer kurze Phasen, in denen die Koordinaten stabil sind. Zwischen $10000s \leq t \leq 15000s$, also über einer Zeitraum von ca. 83 Minuten, sind die Koordinaten über einen längeren Zeitraum recht stabil, allerdings gibt es eine leichte Bewegung in manchen Dimensionen. Abb. 5.9 zeigt eine Vergrößerung dieses Ausschnitts.

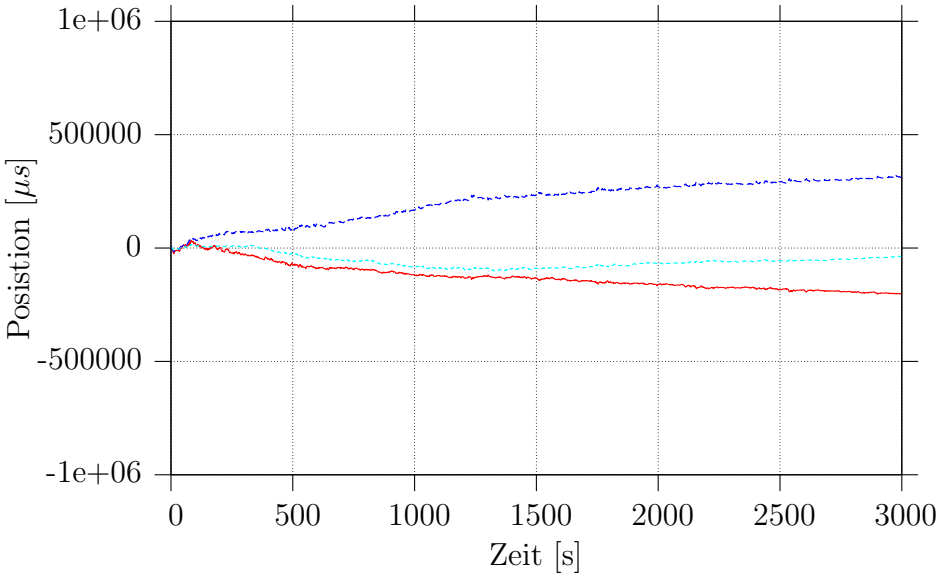


Abbildung 5.7: Vivaldi-Koordinaten mit 3 Dimensionen

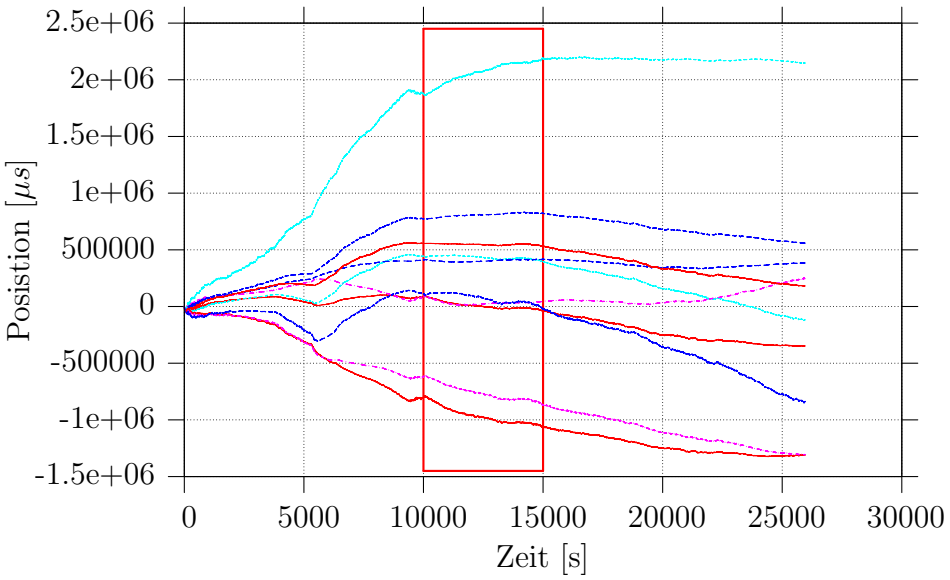


Abbildung 5.8: Lange Messung von ca. 8 Stunden

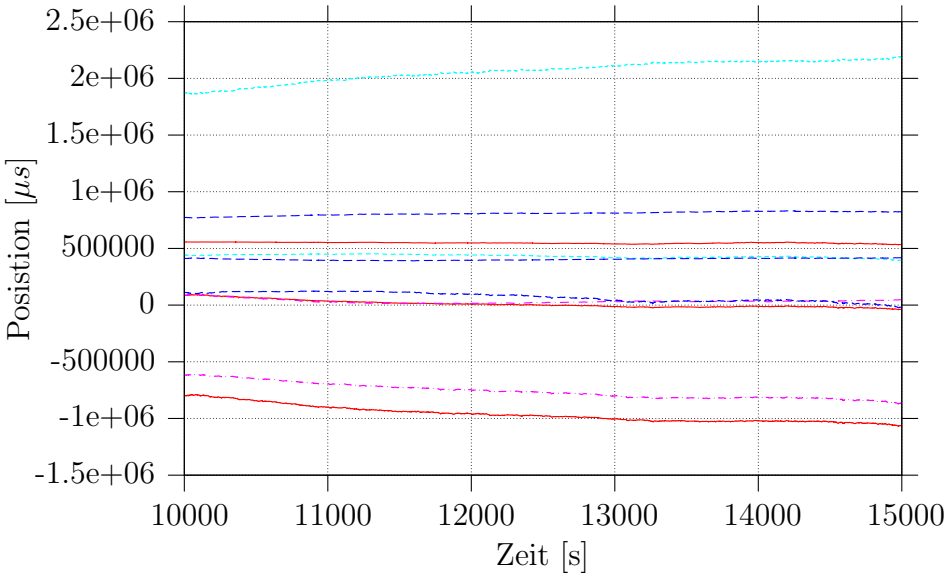


Abbildung 5.9: Vergrößerung des in Abb. 5.8 markierten Bereichs. Hier sind die Vivaldi-Koordinaten recht stabil.

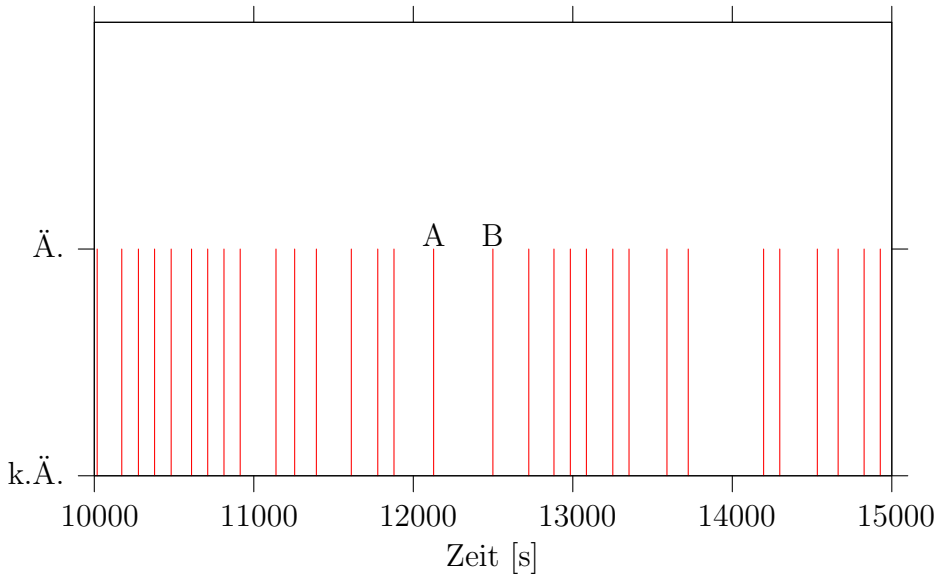


Abbildung 5.10: Vergrößerung des in Abb. 5.8 markierten Bereichs. Hier werden die Änderungen der CAN-Koordinaten in diesem Zeitraum dargestellt.

Anschließend setzt wieder eine Bewegung der Koordinaten ein, die sich bis zum Ende der Messung fortsetzt. Dies liegt wohl an einer stattgefundenen Änderung der Netzwerklatenzen, die Neuordnung der Koordinaten vieler Knoten fordert.

Die CAN-Koordinaten werden wie in Kapitel 4 erläutert aus den Vivaldi-Koordinaten abgeleitet: Sind seit der zuletzt durchgeführten Anpassung der CAN-Koordinaten mindestens `CAN_CHANGE_TIME` Sekunden vergangen und ist die Distanz zwischen den alten CAN-Koordinaten und den Vivaldi-Koordinaten größer als `CAN_CHANGE_DIST`, so werden die CAN-Koordinaten an die aktuellen Vivaldi-Koordinaten angepasst. Hier wurde `CAN_CHANGE_TIME=100` und `CAN_CHANGE_DIST=20000` gewählt. Abb. 5.10 zeigt für den in Abb. 5.8 markierten Ausschnitt das Änderungsverhalten der CAN-Koordinaten. Die Balken stellen eine Änderung der Koordinaten dar. Zwischen den Zeitpunkten A und B kommt es beispielsweise nicht zu einer Änderung der CAN-Koordinaten. In dem 83 minütigen Ausschnitt kommt es insgesamt zu 31 Änderungen der CAN-Koordinaten, zwischen zwei Änderungen liegen durchschnittlich $2,7$ Minuten.

Sollen die CAN-Koordinaten stabiler gehalten werden, so kann dies durch Anpassung der Parameter `CAN_CHANGE_TIME` und `CAN_CHANGE_DIST` erreicht werden. Die Erhöhung von `CAN_CHANGE_DIST` führt allerdings zu einer weiteren Reduktion der Auflösung der CAN-Koordinaten, eine Vergrößerung von `CAN_CHANGE_TIME` eher zu einer größeren Verzögerung bis sich tatsächlich geänderte Latenzen in den CAN-Koordinaten niederschlagen können.

5.2.4 Zusammenfassung der Messergebnisse

Zusammenfassend wird hier nochmal auf einige Eigenschaften der Vivaldi- und CAN-Koordinaten hingewiesen, die beim Messen aufgefallen sind:

- Das Bestimmen geeigneter Werte für die Parameter ist sehr zeitaufwendig. Je nach Umgebung und Anforderungen müssen diese aber speziell angepasst werden. Dabei muss zwischen zeitnaher Anpassung des Systems an sich geänderte Netzwerklatenzen und Vermeidung ungewollter Berücksichtigung kurzzeitiger Schwankungen abgewogen werden.
- Es dauert lange, bis die Koordinaten einigermaßen konvergiert haben. In den hier gemachten Versuchen hat pro Sekunde ein Vivaldi-Update stattgefunden. Grob lässt sich abschätzen, dass erst nach ca. 1000 Schritten die Koordinaten konvergieren. Je nach Häufigkeit der Lookups befindet sich das System schneller oder langsamer in einem stabilen Zustand.
- Die Kommunikationsstruktur wirkt sich deutlich darauf aus, wie gut und ob die Koordinaten konvergieren.
- Nicht untersucht wurde, wie sich die Koordinaten eines konvergierten Systems verhalten, wenn neue Knoten hinzukommen. Es ist zu vermuten, dass dies zu Schwankungen der Koordinaten vieler Knoten führt, möglicherweise bleiben die Schwankungen bei einem hochdimensionalen Koordinatensystem aufgrund größerer Freiheitsgrade geringer.
- Es gibt eine leichte Tendenz der Koordinaten, sich im Raum zu bewegen.

6. Zusammenfassung und Ausblick

Im Rahmen dieser Studienarbeit wurde das P2P-Protokoll Chord unter Linux implementiert. Anhand der Signalisierungs- und Lookupnachrichten können die Latenzen zwischen einzelnen Knoten bestimmt werden. Auf Basis dieser Latenzen berechnet der dezentrale Vivaldi-Algorithmus die Koordinaten aller Knoten. Diese Koordinaten positionieren einen Knoten innerhalb des Latenzraums.

Mit dieser Implementierung wurden Messungen im PlanetLab auf weltweit verteilten Knoten durchgeführt. Dabei hat sich herausgestellt, dass die gemessenen Latenzen starken Schwankungen unterworfen sind. Dieser Effekt ist auf Scheduler-Verzögerungen, denen die Anwendung unterworfen ist, da sie als User-Level-Prozess läuft, zurückzuführen. Aus diesem Grund wurden verschiedene Verfahren zur Glättung der gemessenen Latenzen implementiert und evaluiert. Mittels der so geglätteten Latenzen konnte repräsentative Koordinaten berechnet werden.

Ein weiterer Schwerpunkt der Arbeit war die Spezifikation einer CAN-Variante. Modifikationen gegenüber der Originalversion waren insbesondere bei den Mechanismen zur Teilung und Wiedervereinigung von Zonen notwendig. Unter Verwendung einer einfachen Simulation wurde der vorgeschlagene Mechanismus zur Zonenaufteilung getestet.

Dem vorgestellten hybriden Overlay-Netz liegt ein sehr vielversprechender Ansatz zugrunde. Allerdings ist eine abschließende Bewertung erst nach Implementierung des Moduls „CAN/CANplus“ und Evaluierung des Gesamtsystems möglich. Es bleibt auch zu untersuchen, welche Such- und Routingstrategien in diesem System besonders geeignet sind. Desweiteren sollte eine umfassende Betrachtung und Optimierung der Parameter des Systems, wie beispielsweise die Anzahl der Dimensionen und die Wahl des Dämpfungsfaktors, durchgeführt werden.

A. Anhang

A.1 Kommandozeilenargumente

```
Usage: ./hon [OPTIONS]
-i INETADDR    use this ip address
-p PORT        use this port [def. 33333]
-f FILE        set FILE with ep nodes
-s             start new chord ring
-e FACTOR      use exponential avg. latency filter
               with this FACTOR [used as default
               with FACTOR=0.01]
-m LENGTH      use moving avg. latency filter of
               this LENGTH [def. 100]
-r C_ERR       set weighting factor for exponential
               relative error [def. 0.05]
-w C_DELTA     set weighting factor for delta [def. 0.25]
-d LEVEL       set debug LEVEL [def. 1]
-h             show this help
```

A.2 Bestimmung der Ordnung einer Teilungshyperebenen

Die Ordnung einer Hyperebenen kann aus einer Kante der Hyperebene abgeleitet werden. Zwei Verfahren werden im Folgenden hierfür vorgestellt. Der erste Ansatz lässt sich nur bei Verwendung diskreter Dimensionsachsen verwendet, der zweite Ansatz ist auch auf kontinuierliche Achsen anwendbar.

Diskrete Achsen Jede Zone hat eine eindeutige „linke“ und „rechte“ Begrenzung in jeder Dimension. Die Ordnung einer Teilungshyperebene wird bestimmt, indem man eine beliebige Begrenzungslinie, die bei Teilung anhand dieser Hyperebene entsteht, bewertet: Vom niederwertigsten Bit ausgehend, bestimmt man die Anzahl der aufeinanderfolgenden, gleichen Bits. Je größer diese Zahl ist, desto niedriger ist die Ordnung.

Kontinuierliche Achsen Hier hat eine Zone nur eine „linke“ Begrenzung in jeder Dimension. Aus der durch eine Teilungshyperebene neu entstehende Begrenzungslinie einer beliebigen Dimension kann die Ordnung dieser Hyperebene berechnet werden: Vom niederwertigsten Bit ausgehend, zählt man in diesem Fall die Anzahl der aufeinanderfolgenden Nullen. Je größer diese Zahl ist, desto niedriger ist die Ordnung.

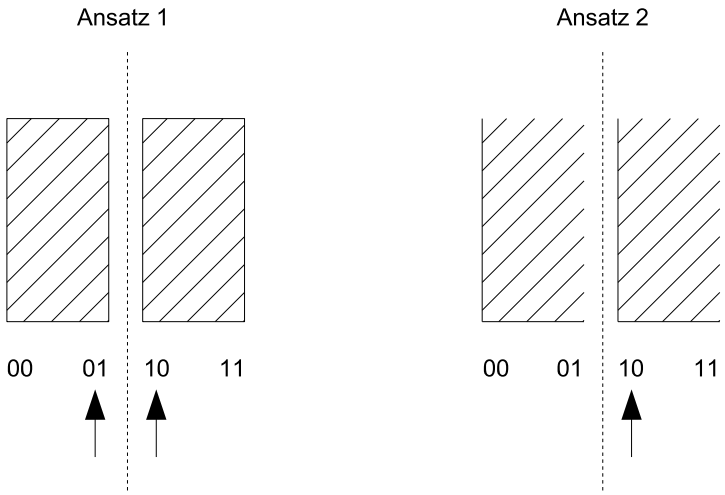


Abbildung A.1: Bestimmung der Ordnung einer Teilungshyperebenen. Die gestrichelten Linien stellen eine Teilungshyperebne dar. An den durch die Pfeile markierten Begrenzungslinien kann die Ordnung bestimmt werden.

Literatur

[1] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[2] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. *SIGCOMM Comput. Commun. Rev.*, 34(1):113–118, 2004.

[3] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, 2004.

[4] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev.*, 35(5):202–215, 2001.

[5] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[6] F. Dabek, J. Li, E. Sit, J. Robertson, M. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[7] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.

[8] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. Idmaps: a global internet host distance estimation service. *IEEE/ACM Trans. Netw.*, 9(5):525–540, 2001.

[9] R. Huebsch, J. Hellerstein, N. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, 2003.

[10] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 213–222. ACM Press, 2002.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[12] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[13] E. Ng and H. Zhang. Predicting internet network distance with coordiantes-based approaches. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2001.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[15] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proceedings of IEEE INFOCOM'02*, 6 2002.

[16] S. Ratnasamy, I. Stoica, and S. Shenker. Routing algorithms for dhts: Some open questions. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 45–52. Springer-Verlag, 2002.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[18] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[19] T. Roscoe S. Rhea, D. Geels and J. Kubiatowicz. Handling churn in a dht. In *In Proc. USENIX*, 2004.

[20] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[21] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 175–186. ACM Press, 2003.

[22] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.