Institut für Parallele und Verteilte Systeme
Abteilung Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diplomarbeit Nr. 3170

Entwicklung eines skalierbaren Emulationsrahmenwerks für mobile Knoten

Frank Schuh

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dipl.-Inf. Andreas Grau
begonnen am:	04. April 2011
beendet am:	11. Oktober 2011
CR-Klassifikation:	C.2.4, C.2.1, I.6.8

Abstract

Many different kinds of mobile devices make the communication very complex. Also because of the rising tendency in the mobile sector there is a need for an emulated environment to simulate movement and connection models of mobile devices. The functionality for this shall be a framework to extend the „Network Emulation Testbed“ (NET).

This thesis examines different characteristics of the movement and connection models as well as the scalability to enable the emulation of large amounts of nodes in NET. As a result of this examination the NET will be extended by a framework that provides an interface for different kinds of movement and connection models. To ensure immense amounts of nodes a Quadtree will be used; a so-called „space partitioning“ - algorithm. The test results, the evaluation of the functions of the framework and the performance will be presented in the end of the thesis.

Zusammenfassung

Durch die hohe Anzahl an mobilen Geräten unterschiedlicher Typen ist die Kommunikation zwischen den Geräten sehr komplex. Nicht zuletzt auch wegen des steigenden Trends im mobilen Sektor besteht der Wunsch nach einer Umgebung, in der die Bewegungs- und Verbindungsmodelle von mobilen Geräten simuliert werden können. Um diese Funktionalität soll das NET (Network Emulation Testbed) um ein Framework erweitert werden.

Diese Arbeit untersucht verschiedene Eigenschaften von Bewegungs- und Verbindungsmodellen sowie deren Skalierbarkeit, damit eine hohe Anzahl an Knoten im NET emuliert werden kann. Als Ergebnis dieser Untersuchung wird das NET um ein Framework erweitert, das eine Schnittstelle für verschiedene Bewegungs- und Verbindungsmodelle bietet. Um die Emulation immenser Mengen an Knoten zu ermöglichen, wird ein Quadtree, ein so genannter „Space Partitioning“ - Algorithmus, verwendet.
Als Abschluss dieser Arbeit werden die Testergebnisse, die Evaluation der Funktionalität des Frameworks und die Performance vorgestellt.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Welt befindet sich im ständigen Wandel. In nahezu allen Bereichen ist ein Zuwachs an Technologien spürbar und nicht mehr wegzudenken. Gerade im Bereich der mobilen Kommunikation lässt sich der Fortschritt deutlich beobachten. Die mobilen Geräte gleichen immer mehr vollständigen Computern und übernehmen Aufgaben, die früher nur an einem stationären Computer denkbar waren. Die Hauptfunktion ist hierbei längst nicht mehr nur das Telefonieren; mobile Geräte sind inzwischen für die unterschiedlichsten Aufgaben nutzbar, die vor allem im Internet getätigt werden. Neben Smartphones erfreuen sich auch Tablets immer größerer Beliebtheit. Sie verdrängen mehr und mehr den stationären Computer, da Smartphones und Tablets gerade unterwegs durch ihre geringere Größe und Gewicht gut einsetzbar sind. Mit ihrer einfachen Bedienung durch den Touchscreen und der vorgefertigten Programmauswahl setzen sich diese Geräte bei den Anwendern durch. Außerdem ermöglicht es den Nutzern alle möglichen Internetanwendungen mobil auszuführen.
Dies hat Auswirkungen auf die Netzwerkverbindungen. Während früher nur stationäre Computer Zugang zum Internet hatten existieren jetzt eine Vielzahl von Geräten, denen diese Möglichkeit ebenfalls offen steht. Für mobile Knoten ist ein kabelloser Ansatz nötig. Dies hat zur Folge, dass sich die Topologie ständig ändert und das Netzwerk immer weiter aktualisiert und neu aufgebaut werden muss.
Zu diesem Zweck wurden mobile ad-hoc Netzwerke (MANET) entwickelt, in denen es möglich ist, dass sich Knoten bewegen können und dass das Netzwerk dynamisch aufgebaut wird.
Ein mobile ad-hoc Netzwerk ist ein Netzwerk, in dem es keine zentrale Kontrollstruktur gibt. Trotz der fehlenden zentralen Kontrollstruktur müssen mobile Geräte miteinander kommunizieren können. Deshalb muss das Netzwerk ständig angepasst werden, falls sich mobile Knoten bewegen, ausfallen oder hinzukommen. Um Information zu versenden werden diese von Knoten zu Knoten weitergesendet, bis der Zielknoten erreicht ist.

Werden Programme oder Protokolle für ein MANET entwickelt müssen diese getestet werden. Da ein Test in der Realität sehr schwierig und meist auch kaum durchführbar ist, werden hierfür Simulatoren oder Emulatoren verwendet, die die Realität abbilden und dadurch widerspiegeln sollen.
Um ad-hoc Netzwerke emulieren zu können muss - neben den Protokollen - dem Emulator ein Bewegungsmuster sowie ein Verbindungsmuster mitgegeben werden, anhand derer die Bewegungs- und Verbindungseigenschaften der Emulationsknoten definiert werden können.
Ein Bewegungsmodell beschreibt das Bewegungsverhalten eines Emulationsknotens. Ein Verbindungsmodell beschreibt das Verbindungsverhalten zwischen zwei Emulationsknoten. Da eine Vielzahl dieser Modelle existiert wird überprüft, wie sie in den Emulator integriert werden können.

1.2 Thema der Diplomarbeit

Ziel der vorliegenden Diplomarbeit ist die Entwicklung eines Frameworks für mobile Knoten. Dafür müssen die vorhandenen Bewegungs- und Verbindungsmodelle überprüft werden, um sie in das Framework einzubinden. Dieses Framework soll in das Network Emulation Testbed (NET) der Universität Stuttgart integriert werden.

Das NET besteht aus einem Cluster, in dem Virtualisierungverfahren verwendet werden. Hierdurch kann eine große Anzahl an virtuellen Knoten der Emulation hinzugefügt werden, die sich wie physische Knoten verhalten. Die bisherigen Emulationsfähigkeiten des NET beschränkten sich auf Netzwerke mit stationären Knoten. Die Topologie des Netzwerkes kann lediglich manuell verändert werden.
Ziel der vorliegenden Diplomarbeit war daher, NET um Bewegungs- und Verbindungsmodelle zu erweitern, so dass mobile Knoten und ad-hoc Netzwerke damit emuliert werden können.

Das Framework soll mit Hilfe von Bewegungs- und Verbindungsmodellen ermöglichen, die Infrastruktur anzupassen. Nach dieser Diplomarbeit wird es möglich sein den Knoten Bewegungseigenschaften hinzuzufügen und eine Überprüfung durchzuführen, die zeigt ob Knoten miteinander verbunden sind oder nicht. Damit wird es möglich, mobile Knoten und damit ad-hoc Netzwerke zu unterstützen.

Die Kernaufgabe dabei war die Entwicklung eines Frameworks, welches das Hinzufügen von Bewegungs- und Verbindungsmodellen zu NET ermöglicht. Die Aufgabe besteht nicht darin möglichst viele Modelle in das Framework zu implementieren. Daher sollen nicht einzelne Bewegungs- oder Verbindungsmodelle auf Effizienz überprüft werden, sondern das Framework, welches es ermöglicht die unterschiedlichen Modelle in das NET zu integrieren.

1.3 Verwandte Arbeiten

Das größte Problem tritt bei der Verbindungsberechnung auf, da hierbei die Verhältnisse zwischen sämtlichen Knoten untereinander untersucht werden müssen. Damit nicht sämtliche Knoten verglichen werden müssen, wird ein Ansatz benötigt, der die Anzahl der Vergleiche einschränkt. Dies geschieht mit Hilfe von Partitionierungen. Ähnliche Probleme treten auch in anderen Bereichen wie beispielsweise der Bildverarbeitung und der Kollisionserkennung auf.

1.3.1 Bildverarbeitung

Um Bilder zu speichern werden Datenstrukturen benötigt, die das Bild in einzelne Bereiche aufteilen und somit Informationen zusammenfassen zu können. Die Unterteilung erfolgt mit Hilfe von unterschiedlichen Strukturen [1]. In diesen wird versucht ein Bild adaptiv abhängig von den Details zu unterteilen. Bereiche, die viele Details aufweisen werden feiner unterteilt als Bereiche die über eine wesentlich geringere Detailsdichte verfügen.

Diese Aufteilungstechniken können auf die Probleme des Frameworks übertragen werden. Wie bei der Bildverarbeitung können Bereiche mit einer größeren Knotendichte feiner unterteilt werden als diejenigen mit einer geringeren Knotendichte. In der Bildverarbeitung arbeiten die Algorithmen mit statischen Daten, was dazu führt, dass diese Ansätze modifiziert werden müssen. Grund hierfür ist, dass das Framework mit beweglichen Knoten zurecht kommen muss.

1.3.2 Kollisionserkennung

Bei der Kollisionserkennung (wie zum Beispiel in [2]) soll überprüft werden, ob sich zwei Objekte berühren. Dafür wird das Gebiet in dem sich die Objekte befinden, in Abhängigkeit von den Strukturen (Quadtree, Grid und weitere) in Teilbereiche unterteilt, um bereits mit wenigen Prüfungen eine große Anzahl von Objekten ausschließen zu können. Dadurch sind für die Kollisionserkennung nur Objekte interessant, die sich in der „Nähe“ befinden.

Übertragen auf das Framework kann man einem Knoten der sendet eine Sendereichweite zuweisen. Dadurch ergibt sich ein Gebiet um einen Knoten herum, für den überprüft werden soll, welche anderen Knoten sich innerhalb des Gebietes befinden. Dadurch ergibt sich eine Art Kollisionserkennung. Es soll erkannt werden, ob die Sendefläche des Sendeknotens mit anderen Knoten kollidiert. Nur die Knoten, bei denen eine Kollision festgestellt wurden, kommen als Verbindungskandidaten in Frage. Für das Framework ist dabei nicht interessant zu welchem Zeitpunkt sich zwei Knoten treffen, sondern wann sich ein Knoten innerhalb der Sendereichweite des anderen befindet.

1.4 Aufbau

Zunächst sollen in dieser Arbeit die Grundlagen vorgestellt werden, die benötigt werden damit das Framework entwickelt werden kann. Hierfür wird im zweiten Kapitel zum einen auf die Umgebung, unter der das Framework lauffähig sein soll, zum anderen auf die Bewegungs- und Verbindungsmodelle, die für die Implementierung des Frameworks notwendig sind, eingegangen.
In Kapitel 3 wird gezeigt, wie die im vorhergehenden Kapitel vorgestellten Bewegungs- und Verbindungsmodelle in ein Framework - welches in NET laufen soll - integriert werden können.
Im Anschluss daran wird die Implementierung in der NET-Umgebung vorgestellt und es werden die Erweiterungsmöglichkeiten für das Framework diskutiert. In Kapitel 5 erfolgt die Evaluierung der Implementierung, an die sich das Fazit anschließt. Abgeschlossen wird diese Arbeit durch eine Zusammenfassung und einen Ausblick.

2 Grundlagen

2.1 Framework

Um das Framework zu entwickeln muss untersucht werden, was es leisten soll. Hierfür muss die Aufgabenstellung genauer betrachtet werden. Es soll ein Framework entwickelt werden, welches vorhandene Bewegungsmodelle sowie vorhandene Verbindungsmodelle integrieren kann. Der Fokus liegt dabei auf der Modularität, das heißt, der Möglichkeit, einfach und schnell Bewegungs- und Verbindungsmodelle auszutauschen oder hinzuzufügen.
Im Folgenden wird beschrieben welche Eigenschaften das Framework haben soll.

Lauffähigkeit im NET

Das Framework soll innerhalb des an der Universität Stuttgart vorhandenen Projektes NET lauffähig sein. Dafür ist es notwendig zu wissen, an welchen Stellen das Framework mit diesem Projekt interagieren soll. Außerdem müssen die notwendigen Schnittstellen bekannt sein, damit das Framework diese ansprechen kann. Damit kann entschieden werden, wie die Komponenten des Frameworks auf dem NET verteilt werden sollen und wie diese am Besten miteinander kommunizieren. Abhängig davon wie die Komponenten verteilt werden, müssen Updatenachrichten versendet werden, um die Informationen über Knoten vollständig und korrekt zu halten. Das Framework muss die Verwaltung für die Verteilung der Knoten übernehmen. Die Verteilung der Knoten ist durch NET vorgegeben und muss von dem Framework übernommen werden.

Bewegungen

Bewegungsmodelle sollen mit möglichst einfachen Schritten diesem Framework hinzugefügt werden können. Dabei soll ein Beispielmodell bereits vorhanden sein. Zu untersuchen ist, wie diese Bewegungsmodelle am sinnvollsten unterteilt werden können. Hierfür ist eine genauere Analyse notwendig, welche in den folgenden Abschnitten durchgeführt wird.

Verbindungen zweier Knoten

Neben der Bewegung soll es auch möglich sein dem Framework verschiedene Verbindungsmodelle hinzuzufügen. Wieder soll dies auf möglichst einfache Weise erweitert werden können. Auch hier soll ein Beispielmodell als Demonstrator implementiert werden.

Skalierbarkeit

Da davon ausgegangen werden kann, dass die Anzahl und die Verbindungen der Knoten zukünftig steigen werden, soll das Framework weitgehend skalierbar gehalten werden. Bei den Verbindungsmodellen wird darauf geachtet, dass bei der Überprüfung die Verbindung von Knoten nicht mit allen anderen Knoten verglichen wird, sondern nur mit einem kleineren Teil davon. Dafür wird ein Gebietsaufteilungsalgorithmus (Space Partitioning) verwendet, der die Knoten innerhalb der Simulation unterteilt, wodurch weniger Knoten für die Verbindung überprüft werden. Es wird ebenfalls untersucht, ob die Aufgaben innerhalb des Frameworks verteilt werden können, um so einzelne Clients zu entlasten. Dafür wird überprüft wo die Bewegungs- und Verbindungsüberprüfung durchgeführt werden kann. Ebenfalls wird überprüft was für ein Nachrichtenaustausch vorhanden sein muss.

Die Entwicklung des Frameworks verläuft damit in mehreren Schritten. Es ist notwendig zu wissen in welcher Umgebung das Framework laufen soll. Dafür muss das NET näher vorgestellt und untersucht werden. Als Nächstes muss geklärt werden welche Bewegungs- und Verbindungsmodelle existieren. Nach der Erfassung der vorhandenen Modelle kann nach Techniken gesucht werden, um diese Modelle in das Framework zu integrieren und um zu erkennen welche Schnittstellen für die Umsetzung notwendig sind.

2.2 Network Emulation Testbed (NET)

Bei dem NET [3] handelt es sich um ein derzeit aus 16 physischen Knoten bestehendes Cluster in dem sämtliche Knoten je 2 Quad-Core CPUs besitzen. Das NET Projekt wurde von der Universität Stuttgart ins Leben gerufen und wird dort ständig erweitert und gepflegt. Es handelt sich dabei um ein Netzwerk, welches mit Hilfe von Knotenvirtualisierung Tausende von virtuellen Knoten für eine Vielzahl häufig verwendeter Tests zur Verfügung stellt. Dadurch besteht die Möglichkeit Protokolle sowie Programme in diesem System mit unterschiedlichen Eigenschaften an einem Netzwerk zu testen. Für die Emulation stehen Methoden der gängigen Arten der Bitübertragung zur Verfügung, die im Vorfeld spezifiziert werden können.
Die Emulationsarchitektur TVEE [4] (Time Virtualized Emulation Environment) wurde entwickelt um die Knotenvirtualisierung zu realisieren. Dafür existieren in jedem physischen Knoten virtuelle Maschinen in denen sich virtuelle Knoten befinden. Die Knotenvirtualisierung besteht aus zwei Ebenen. Die erste Ebene wird durch die Software XEN [5] realisiert, welche für die Erzeugung der virtuellen Maschinen zuständig ist. XEN erstellt mehrere virtuelle Container, die die Ressourcen untereinander aufteilen (siehe Abbildung 1). Der erste Container (Steuerungsdomain) ist für administrative Zugriffe vorhanden. Über die Steuerungsdomain wird der komplette Netzwerkverkehr geleitet und über eine Bridge verteilt.

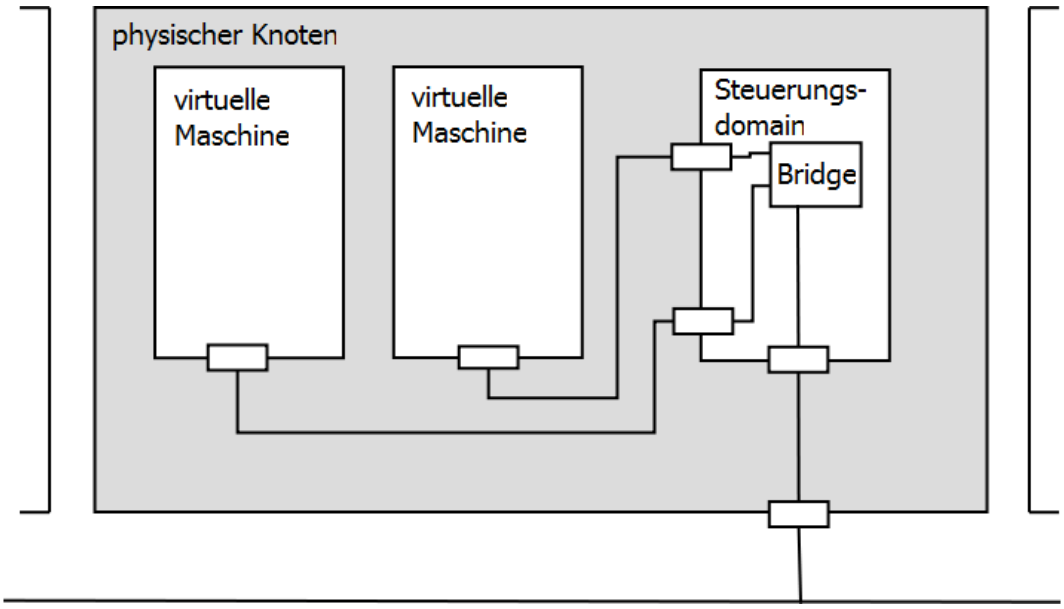


Abbildung 1: Ein physischen Knoten

Die zweite Ebene wird durch die Virtualisierungssoftware OpenVZ [6] realisiert, die für die Erzeugung der virtuellen Knoten innerhalb der virtuellen Maschinen zuständig ist. OpenVZ teilt, mit Hilfe des virtuellen Routings, die von XEN erzeugten virtuellen Maschinen in voneinander isolierte Partitionen ein. Diese Partitionen sind die virtuellen Knoten. Damit ergibt sich die Abbildung 2.

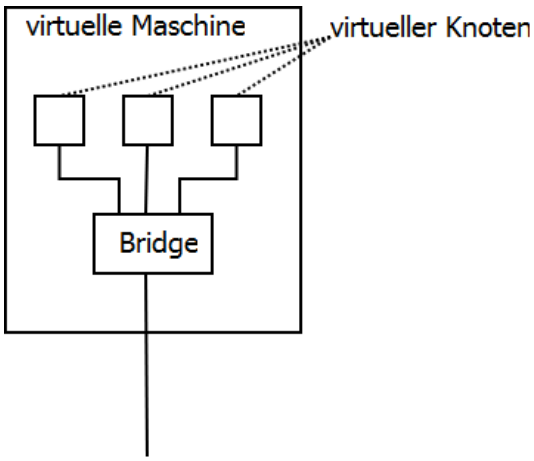


Abbildung 2: Eine virtuelle Maschine mit virtuellen Knoten

TVEE ist neben der Knotenvirtualisierung ebenfalls für die Zeitvirtualisierung [2] zuständig. Ist die Auslastung des Clusters bei einer Emulation zu hoch, so kann dadurch die emulierte Zeit reduziert werden, um damit die Auslastung zu reduzieren. Ist die Auslastung andererseits zu gering, so ist es möglich die Geschwindigkeit damit zu erhöhen. Die Emulation kann dadurch nicht nur in Echtzeit, sondern auch in einer schnelleren oder langsameren Geschwindigkeit durchgeführt werden. Dies ermöglicht größere und umfangreichere Versuche, die in Echtzeit durch NET nicht möglich wären. Der Nachteil dieser Technik besteht darin, dass die Emulationszeit in die Länge gezogen wird.

Neben diesen Techniken gibt es weitere Tools, welche die Effizienz von NET erhöhen.
Der NETShaper [7] ist dafür zuständig, dass zur Laufzeit Änderungen der Topologie gemacht werden können. Er erlaubt, bestehende Verbindungen zwischen virtuellen Knoten zu modifizieren. Anpassungen können dabei in der Bandbreite, der Verzögerung und der Verlustrate vorgenommen werden.
NETplace [8] errechnet eine optimale Anfangsplatzierung der virtuellen Knoten mit der ein Experiment gestartet wird. Dadurch wird die Last auf allen physischen Maschinen minimiert. Da sich die Last zur Laufzeit ändert ist diese Anfangsplatzierung nicht mehr das Optimum. Abhängig davon wie sich die Auslastung auf den Rechnern verändert, berechnet NETbalance [9] zur Laufzeit neue Platzierungen der Knoten und konfiguriert das Experiment um. Hiermit ergibt sich wieder eine optimale Platzierung der Knoten. NETplace und NETbalance minimieren dadurch zusammen die Laufzeit der Emulation.

Um die Emulation zu verwenden ist ein Steuercomputer (NETfe) notwendig. NETfe kann auf jeden beliebigen physischen Knoten sowie alle virtuellen Maschinen und Knoten zugreifen. Von dem NETfe aus werden alle Emulationen gesteuert. Er bestimmt dabei welche Eigenschaften das verwendete System hat. NETfe legt fest welche Programme auf den jeweiligen virtuellen Knoten beziehungsweise den virtuellen Maschinen laufen sollen.
Zusätzlich dient NETfe als Fileserver um die Dateisysteme für die virtuellen Knoten zur Verfügung zu stellen. Als Netzwerkdateisystem wird NFS verwendet.
Die Steuerung wird mit Hilfe eines Ruby-Skriptes oder mit interaktiven Ruby-Befehlen vorgenommen. Dies ist auch der Fall wenn es um die Emulation von MANETs geht. Um MANETs zu emulieren müssen die Änderungen über die Verbindungen zwischen einzelnen Knoten per Hand eingegeben werden. Dafür wird die Schnittstelle zum NETshaper verwendet. Der NETshaper ermöglicht die dynamische Veränderung der Netzwerkeigenschaften und kann den eingehenden Netzwerkverkehr der virtuellen Knoten kontrollieren und diese mit Hilfe von Netzwerkparametern anpassen. Dabei wird nur der eingehende Netzwerkverkehr einer virtuellen Netzwerkkarte angepasst.
Diese Aufgabe soll von dem durch diese Diplomarbeit entwickelten Framework übernommen werden.
Da die Knoten bereits im Vorfeld auf die einzelnen virtuellen Maschinen verteilt werden, muss das Framework in der Lage sein, die Informationen über den Aufenthalt der Knoten zu erhalten. Diese Informationen werden durch ein Ruby-Skript am Anfang der Emulation an die virtuellen Maschinen weitergegeben und können entweder in dem Ruby-Skript abgefangen oder aber von den virtuellen Maschinen im Nachhinein ausgelesen und zurückgeschickt werden.
Um die Auslastung der virtuellen Maschinen zu überwachen, wird ein Koordinator benötigt. Dieser berechnet die optimale Experimentgeschwindigkeit, sodass kein Rechner überlastet wird. Dadurch wird auch gleichzeitig die Experimentlaufzeit minimiert.
Fasst man die einzelnen NET Komponenten zusammen so ergibt sich das in Abbildung 3 dargestellte Bild.

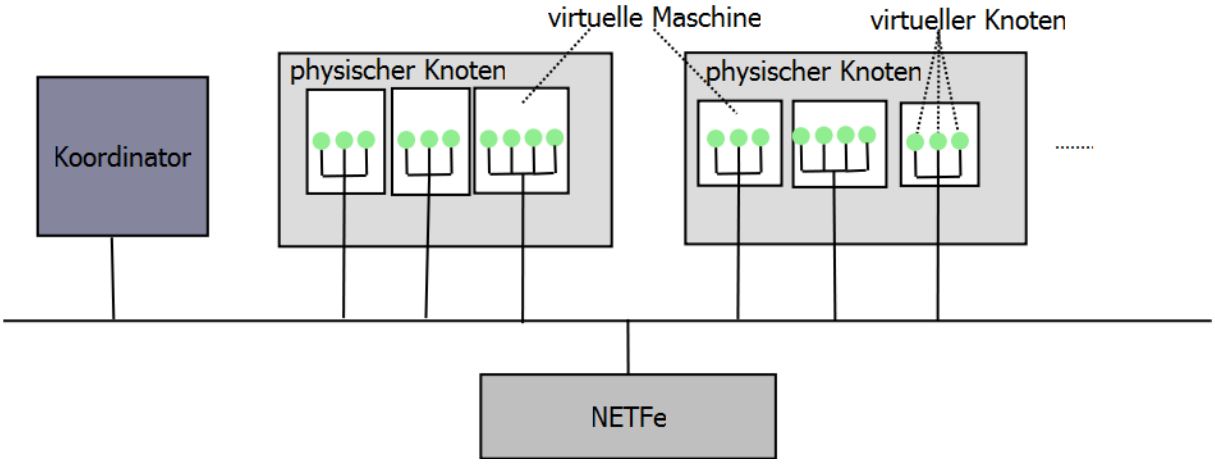


Abbildung 3: Struktur des NETs

2.3 Bewegungsmodelle

Neben der Infrastruktur, in der das Framework laufen soll, muss geklärt werden, wie sich die Knoten bewegen können. Dafür ist es notwendig möglichst viele Bewegungsmodelle zu betrachten, um zu entscheiden, welche Eigenschaften der Modelle für das Framework wichtig sind. Bewegungsmodelle [10] schreiben vor, wie sich ein einzelner Knoten innerhalb eines Szenarios verhalten soll. Es gibt unterschiedliche Bewegungsmodelle um unterschiedliche Szenarien zu modellieren. So gibt es beispielsweise Bewegungsmodelle, die besser für die Innenstadt [11] geeignet sind, während dies bei anderen auf das freie Feld [12] zutrifft. Dabei können Bewegungsmodelle anhand von Regeln und Formeln die Bewegungen definieren, die für jeden einzelnen Knoten vorgesehen sind. Diese Regeln können dabei entweder in jedem Schritt neu entschieden werden oder von aufgenommen Daten abhängen.
Eine Möglichkeit solcher aufgenommenen Daten sind Traces (Spuren), welche ein aufgezeichnetes Verhalten von Knoten darstellen. Diese können dafür benutzt werden Bewegungen exakt nachzubilden. Traces haben allerdings den großen Nachteil, dass der Weg fest vorgegeben ist und man deshalb die Daten im Vorfeld aufzeichnen muss.

Die Bewegungsmodelle lassen sich in unterschiedliche Gruppen einteilen um festzustellen welche Punkte für das Framework wichtig sind. Hierfür wurde als Basis die Gruppierung aus [13] verwendet und an die Probleme dieser Arbeit angepasst. Eine Auswahl dieser Einteilung sieht man in Abbildung 4. Es werden die Aspekte übernommen, die für die einzelne Bewegung von mobilen Knoten wichtig sind. Auf diese Gruppierung wird im Folgenden genauer eingegangen.

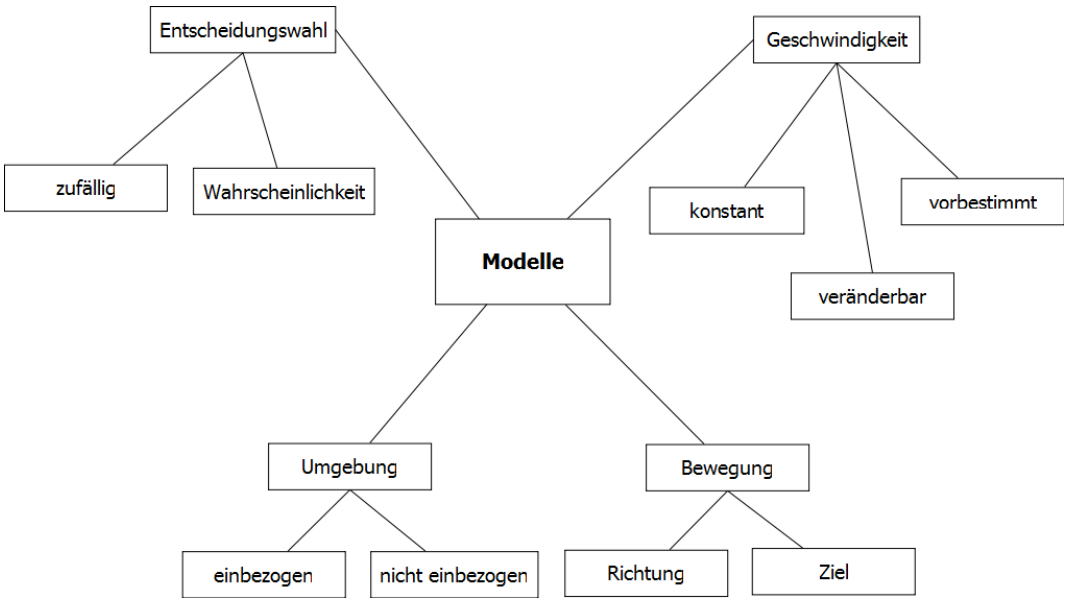


Abbildung 4: Klassifizierung - Angepasst aus [13]

2.3.1 Kategorisierung der Bewegungsmodelle

Geschwindigkeit

Für das Verändern der Geschwindigkeit innerhalb eines Modells gibt es mehrere Ansätze. Diese können in drei Gruppen eingeteilt werden:

- Die Geschwindigkeit wird einmal für jeden Knoten gewählt. Sie verändert sich daraufhin nicht mehr.
- Die Geschwindigkeit ändert sich im Laufe der Simulation immer wieder. Das Modell bestimmt dabei wann eine neue Geschwindigkeit gewählt wird. Dies kann zum Beispiel nach einer gewissen Zeit oder einer bestimmten zurückgelegten Distanz vorgenommen werden. Die Geschwindigkeit kann sofort angenommen werden oder stückweise erhöht beziehungsweise verringert werden, bis die Zielgeschwindigkeit erreicht ist.
- Die Geschwindigkeitsänderungen sind von Anfang an fest vorbestimmt.
  Die Änderungen werden so durchgeführt wie sie von aufgenommenen Daten (Traces) vorgegeben werden.

Umgebungseinfluss

Abhängig von den Bewegungsmodellen wird die räumliche Gegebenheit unterschiedlich einbezogen. Dabei kann man diese in drei Gruppen einteilen:

- Das Bewegungsmodell ignoriert jedes Hindernis und jeder Knoten bewegt sich — unabhängig von den Knoten, die sich in dem Gebiet befinden — direkt auf das von ihm gewählte Ziel zu.
- Die Knoten bewegen sich nur auf von dem Modell vorgegebenen Wegen und verlassen diese nicht. Dadurch ergibt sich ein Wegegraph, der die Bewegungen bestimmt.
- Die Knoten bewegen sich auf einer vordefinierten Route durch die Region. Diese Informationen müssen im Vorfeld aufgezeichnet werden.

Bewegung

In Bezug auf Bewegungen lassen sich die Modelle in zwei Gruppen unterteilen. Dabei handelt es sich zum einen um die Modelle, die einen festen Punkt innerhalb dieses Gebietes wählen und zum anderen um die Modelle, welche eine bestimmte Richtung zur Bewegung festlegen. Die Wahl dieser Richtung oder des Punktes kann noch weiter unterteilt werden.

- Das Ziel oder die Richtung wird beliebig gewählt.
- Das Modell gibt eine Anzahl von Zielen oder Richtungen vor, aus denen ausgewählt werden kann. Dies wird bei Modellen verwendet, an denen nicht jeder Ort des Gebietes erreichbar ist.
- Die Reihenfolge der Ziele ist durch eine Abfolge von Punkten oder Richtungen vorgegeben.

Wird die Bewegung nur durch eine Richtung bestimmt, gibt es Modelle, die eine Richtung direkt wählen und setzen oder diese in kleinen Stufen zu dieser zu wählenden Richtung hinführen. Dies verhindert ein Abknicken der Bewegungen und ist dadurch näher an der Realität.

Entscheidungswahl

Für die Entscheidung welche Richtungen oder Ziele eingesetzt werden gibt es zwei Gruppierungen von Modellen.

- Die Modelle wählen ihre Ziele oder Richtungen unabhängig von irgendwelchen Daten. Wird ein neues Ziel oder eine Richtung gewählt geschieht dies zufällig.
- Dem Modell liegen Wahrscheinlichkeiten zugrunde. Jeder Knoten wählt den nächsten Schritt — abhängig von Wahrscheinlichkeiten. Hiermit ist es möglich den Knoten Eigenheiten zuzuweisen. Dadurch können in einer Stadt wichtige Punkte einer höheren Wahrscheinlichkeit zugewiesen werden, wodurch diese von den Simulationsknoten häufiger besucht werden.

2.3.2 Existierende Modelle

Random Waypoint Modell

Das Random Waypoint Modell [12] wurde als erstes von Johnson und Maltz vorgestellt. Dieses Modell wird bei mobilen ad-hoc Netzen am häufigsten häufig [13] verwendet. Beim Starten des Random Waypoint Modells wird ein Knoten zufällig in das Gebiet eingesetzt. Jeder Knoten wählt daraufhin ein zufälliges Ziel innerhalb dieses Gebietes und bewegt sich dort hin. Die Geschwindigkeit wird zufällig ausgewählt. Dabei muss die Geschwindigkeit zwischen einem minimalen und maximalen Wert liegen. Der Knoten bewegt sich auf geradem Weg zu dem zuvor gewählten Ziel, ohne auf Hindernisse zu achten. Hat ein Knoten das Ziel erreicht, so wartet er dort eine vorbestimmte Zeit. Die Zeit wird wieder zufällig aus einer minimalen und maximalen Wartezeit gewählt. Nach Ablauf der Zeit wird ein neues Ziel und eine neue Geschwindigkeit gewählt und der Vorgang wiederholt sich. In Abbildung 5 sieht man einen Knoten, der nach dem Random Waypoint Modell bewegt wird.

Der Vorteil dieses Modells ist seine Einfachheit. Allerdings werden die Bewegungen von Menschen nicht gut widergespiegelt. Jeder Knoten in diesem Modell verhält sich komplett zufällig. Ein weiterer Nachteil dieses Modells ist, dass sich sehr viele Knoten durch die Mitte des Simulationsgebietes [14] bewegen. Dies liegt daran, dass sich die Knoten zufällig einen Punkt aussuchen und sich dort direkt hinbewegen und dieser Weg befindet sich auf Grund der Gleichverteilung in der Nähe der Mitte. Dadurch ist die Mitte des Simulationsgebietes deutlich stärker besucht als der Rand des Gebietes.

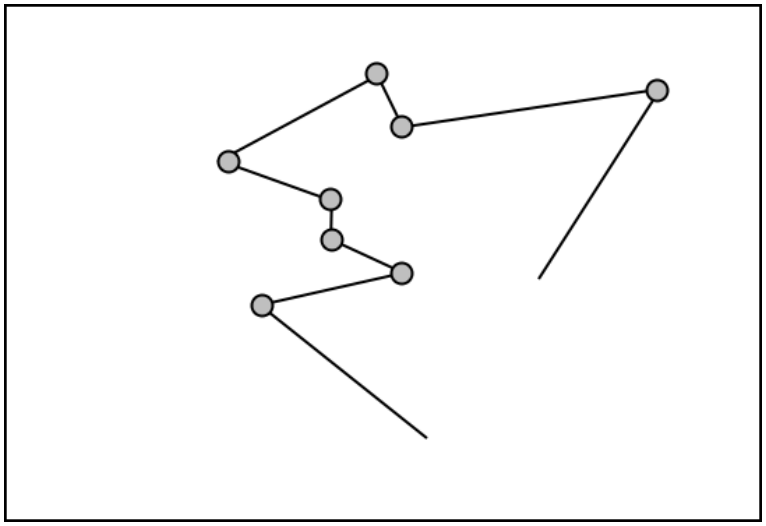


Abbildung 5: Bewegung eines Knotens nach dem R. W. Modell

Random Walk Modell

Das Random Walk Modell [15] ist dem Random Waypoint Modell sehr ähnlich. Statt einem Zielpunkt wird von jedem Knoten eine Richtung gewählt. Jeder Knoten wählt einen Winkel, der im Bereich $[0, 2\pi]$ und eine Geschwindigkeit, die im Bereich $[v_{min}, v_{max}]$ liegt. Der Knoten bewegt sich mit diesen Daten entweder eine bestimmte Zeit oder eine bestimmte Strecke in diese Richtung. Daraufhin wählt er erneut eine Geschwindigkeit und einen Winkel aus und setzt den Weg fort. Abbildung 6 zeigt einen Knoten, der mit dem Random Walk Modell bewegt wurde.
Wenn der Knoten an den Rand des Gebietes kommt, prallt er an der Grenze ab und bewegt sich mit dem gleichen Winkel in die andere Richtung. Dabei entspricht der Austrittswinkel dem Eintrittswinkel. Weiterhin existiert eine Variante dieses Modells, die statt der Reflektion des Knotens am Gebietsrand diesen auf der anderen Seite wieder eintreten lässt.

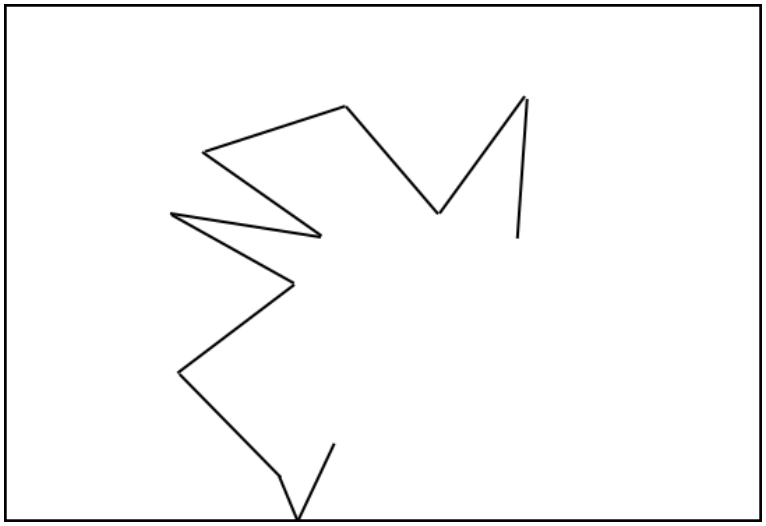


Abbildung 6: Random Walk Modell

Bei beiden Modellen wird die Bewegung davor nicht berücksichtigt, wodurch es zu Knicken kommen kann. Hierfür gibt es Erweiterungen, die dafür sorgen, dass eine Richtungsänderung nicht abrupt vorgenommen wird, sondern in mehreren Schritten an diesen Wert herangeführt wird. Es wird somit in jedem Schritt ein bestimmter Betrag dem Winkel hinzugefügt oder abgezogen, bis der gewünschte Winkel erreicht wird. Auf gleiche Art und Weise werden Geschwindigkeitsänderungen nicht sofort vorgenommen, sondern ebenfalls stückweise verändert, wodurch eine Beschleunigung simuliert werden kann.
Das Random Walk Modell hat im Gegensatz zu dem Random Waypoint Modell nicht den Nachteil, dass sich die Knoten in der Mitte des Simulationsgebietes zentrieren, da sich dieses Element nur eine Bewegungsrichtung aussucht und keinen Punkt im Gebiet. Allerdings spiegelt es die reale Bewegung ebenfalls nicht gut wider. Gerade ruckartige Veränderungen in Geschwindigkeit sowie Richtung entsprechen im Allgemeinen nicht der Realität. Hierfür sind die stückweisen Näherungen an den bestimmten Wert gedacht, wodurch weichere Richtungsänderungen und eine Beschleunigung bei der Geschwindigkeitsänderung hinzukommen.

Graph-based Modell

Sollen die Hindernisse innerhalb des Simulationsgebietes mit einbezogen werden, so muss ein anderes Modell gewählt werden. Hierfür bieten sich graphenbasierte Modelle [11] an. Dabei beschreibt der Graph die Möglichkeiten, wohin sich die Simulationsknoten bewegen können und wohin nicht. Ein Knoten des Graphs (Graphknoten) steht dabei für wichtige Punkte, die von Interesse für die Simulationsknoten sind. Die Kanten des Graphs beschreiben die Wege zwischen den einzelnen Zielen. Dies sind Straßen oder Wege, auf denen sich die Simulationsknoten bewegen können. Jeder Knoten wählt zufällig einen Zielgraphknoten aus und sucht daraufhin den kürzesten Weg. Jedem Knoten wird eine zufällige Geschwindigkeit aus dem Bereich [v_{min} und v_{max}] zugewiesen. An dem Ziel angekommen wartet der Knoten eine zufällige Zeit [t_{min},t_{max}]. Nach dieser Zeit wird ein neues Ziel gewählt und der Ablauf wiederholt.

Dieses - im Vergleich zu Random Waypoint oder Random Walk - deutlich komplexere Modell wird eingesetzt, falls Hindernisse des zu simulierenden Gebiets miteinbezogen werden sollen. Die Hindernisse selbst werden dabei durch den zugrundeliegenden Graphen beschrieben. Solche graph-basierten Ansätze eignen sich beispielsweise zur Modellierung von Städten.

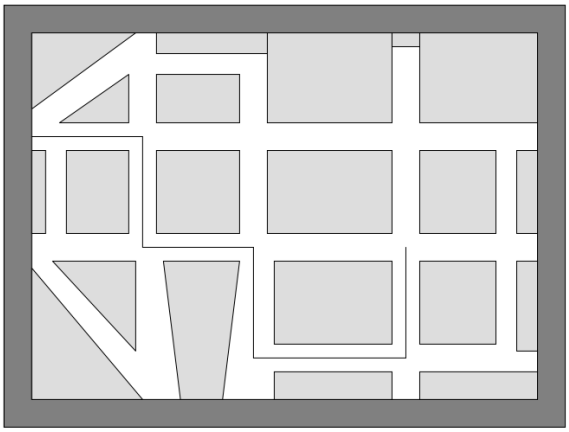


Abbildung 7: Graph-based Modell

Statistisch gewichtete Bewegungsmuster

Bei jedem dieser Modelle wurde das Ziel oder die Richtung immer zufällig gewählt. Will man diese Modelle mehr an ein bestimmtes Verhaltensmuster binden kann die Wahl der nächsten Ziele an Wahrscheinlichkeiten [16] geknüpft werden. Dadurch können in einem Modell, welches aus einem Graph besteht, bestimmte Punkte als Versammlungspunkte beschrieben werden (Museum, Einkaufszentrum, Kino usw).
Je mehr Eigenschaften man dem Knoten geben will, desto mehr Informationen muss man dem Modell und den einzelnen Knoten darin geben. Das Modell wird dadurch zwar genauer, allerdings auch komplexer. Will man ein Gebiet simulieren, in dem sich Menschen nach einem bestimmten Muster verhalten sollen (zum Beispiel der Gang zum Einkaufsladen oder zur Kirche), müssen zusätzliche Informationen angegeben werden. Dadurch ist es möglich verschiedene Knoten zu erzeugen, die unterschiedliche Interessen haben und deshalb ein anderes Verhaltensmuster besitzen. Dies ermöglicht beispielsweise die Simulation verschiedener Altersgruppen.

2.4 Verbindungsmodelle

Neben den Bewegungsmodellen ist es notwendig die vorhandenen Verbindungsmodelle zu kennen und einzuteilen um auch hier entscheiden zu können, wie diese am Besten in das Framework integriert werden können. Verbindungsmodelle dienen der Entscheidung, ob zwei Knoten innerhalb eines Gebietes miteinander kommunizieren können. Für die Verbindungsmodelle ist es nötig zu wissen wie eine drahtlose Kommunikation funktioniert.

Für die Übermittlung von Daten werden Funknetze verwendet. Innerhalb eines Funknetzes gibt es verschiedene Sender, die abhängig von ihren Sendeeigenschaften Signale aussenden, die dann wiederum von Empfängern eingefangen werden können. Abhängig von dem Funknetz kann ein Teilnehmer sowohl als Empfänger und Sender gleichzeitig fungieren. Die Übertragung selbst geschieht indem von der Antenne des Senders elektromagnetische Wellen ausgesendet werden, die von der Antenne eines Empfängers empfangen werden. Interessant ist dabei die Signalstärke, die bei der Empfängerantenne ankommt.

Verbindungsmodelle versuchen diese Stärke durch unterschiedliche Ansätze zu berechnen. Dabei lassen sich empirische Ansätze und deterministische (oder strahlenoptische) Modelle unterscheiden.
Empirische Modelle berechnen die Empfangsstärke anhand von Formeln und können Umgebungseinflüsse nur in Grenzen widerspiegeln.
Deterministische Modelle verwenden einen strahlenoptischen Ansatz, um mit Hilfe der Strahlen die Wellenübertragungen genauer zu beschreiben. Dadurch sind sie in der Lage auch Hindernisse mit einzubeziehen.

2.4.1 Elektrotechnische Grundlagen

Für die Übertragung der elektromagnetischen Wellen spielen die Antennen, die für das Empfangen und Senden zuständig sind und der äußere Einfluss der Umgebung eine Rolle.

Antennen

Neben Umwelteinflüssen auf die elektromagnetischen Wellen hängt die Sendestärke und -richtung einer Antenne insbesondere von dem Typ der Antenne [17] ab.
Unterschieden wird dabei zwischen Rundstrahlern, Sektorantennen und Richtantennen.
Rundstrahler senden gleichmäßig in alle Richtungen und werden bei mobilen Geräten verwendet (Handy).
Die Sektorantennen senden in ein vorher bestimmtes Gebiet (Sektor). Verwendet werden Sektorantennen beim Mobilfunk in dem die Gebiete in einzelne Zellen aufgeteilt und dann versorgt werden. Eine Sektorantenne ist dafür zuständig mehrere Zellen zu versorgen.
Eine Richtantenne wird verwendet, wenn sich die Welle gezielt in eine bestimmte Richtung ausbreiten soll. Solche Antennen werden beispielsweise von Geheimdiensten eingesetzt,da aufgrund der linienförmigen, eng begrenzten Signalausbreitung ein Abhören deutlich erschwert wird.

Einfluss der Umgebung

Bei der Übertragung der elektromagnetischen Wellen werden diese von unterschiedlichen äußeren Gegebenheiten [18] beeinflusst. Diese äußeren Einflüsse sorgen dafür, dass sich die Wellen nicht mit gleicher Geschwindigkeit und in eine Richtung ausbreiten. Sie wird dadurch immer weiter abgeschwächt bis sie komplett verschwindet. Im Folgenden werden einige dieser Einflüsse beschrieben.

Dämpfung

Dämpfung beschreibt den Verlust der Signalstärke insgesamt. Eine Dämpfung tritt bei der Ausbreitung in der Luft ein und wird durch unterschiedliche Wetterbedingungen noch verstärkt. So erhöht sich die Dämpfung bei Regen oder Schnee durch die dort vorkommende Streuung und Absorption. Durch diese Dämpfung ergibt sich bei größerer Entfernung zwischen zwei Knoten eine geringere Empfangsstärke als bei zwei naheliegenden Knoten.

Abschattung

Befinden sich zwischen dem Sender und Empfänger Hindernisse sorgt das dafür, dass keine direkte Wellenausbreitung zwischen beiden Knoten existiert. Dadurch können nur indirekte Wellen den Empfänger erreichen. Die daraus resultierende Abschwächung der Signalstärke nennt man Abschattung (Shadowing).

Reflektion

Trifft eine Welle auf ein Objekt so wird diese von dem Objekt teilweise oder vollständig reflektiert. Wie viel von der Welle reflektiert wird hängt von der Materialeigenschaft des Objektes ab. Wird nicht die komplette Welle reflektiert, wird der andere Teil der Welle durch das Objekt transmittiert. Die Summe der reflektierten und transmittierten Welle ist kleiner als die Stärke der eintreffenden Welle. Dies liegt daran, dass innerhalb des Objektes eine zusätzliche Dämpfung stattfindet. Abbildung 8 zeigt die Transmission und Reflektion einer Welle.
Es gilt, dass der Einfallswinkel dem Ausfallswinkel der reflektierten Welle entspricht.

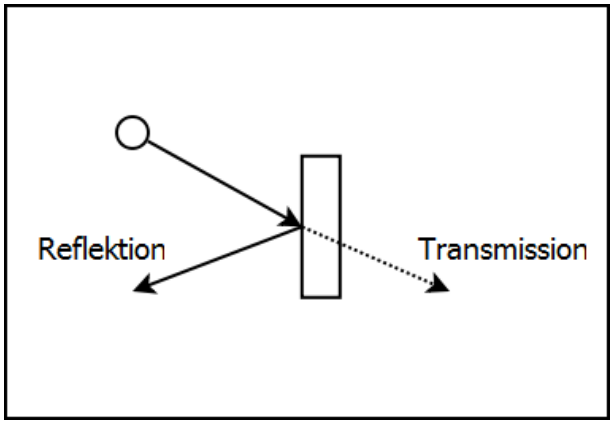


Abbildung 8: Reflektion und Transmission einer Welle

Streuung

Trifft die Welle auf kleine Partikel, wie zum Beispiel Regentropfen oder Schneeflocken, werden die Wellen in verschiedene Richtungen abgelenkt und breiten sich nicht mehr geradlinig aus.

Beugung

Beugung beschreibt die Ablenkung der Welle beim Auftreffen auf die Kante eines Objekts. Durch die Beugung kommt es zu einer Richtungsänderung und einer gleichzeitigen Schwächung des Signals. Diese Eigenschaft wird dazu verwendet, dass ein Signal auch Empfänger erreichen kann die von Hindernissen verdeckt sind (siehe Abbildung 9).

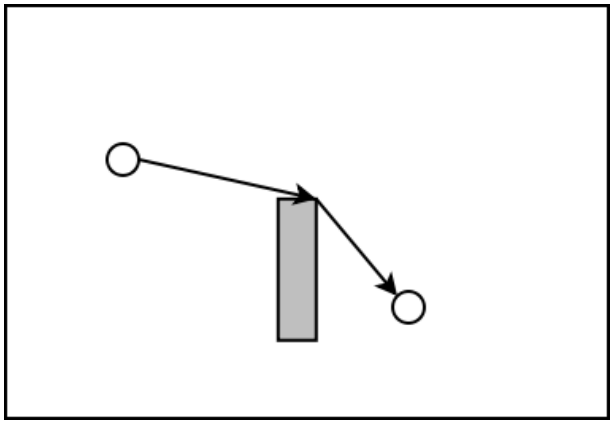


Abbildung 9: Beugung einer Welle

2.4.2 Empirische Modelle

Empirische Modelle verwenden Formeln, um so die Verbindungen zwischen zwei Knoten zu bestimmen. In diesen Formeln werden einzelne Parameter verwendet, wie zum Beispiel die Sendestärke oder der Abstand zwischen den beiden Knoten. Die einfachsten dieser Modelle ignorieren die räumlichen Gegebenheiten und berechnen die Verbindung zwischen zwei Knoten, als ob es keine Hindernisse zwischen ihnen gäbe. Um Hindernisse mit in die Berechnung einzubeziehen gibt es Modelle, die mit Hilfe von Parametern die Signalstärke reduzieren. Dies geschieht dann allerdings in alle Richtungen. Andere Ansätze unterscheiden, ob sich Sender und Empfänger sehen können oder nicht. Abhängig davon werden unterschiedliche Modelle zur Berechnung der Empfangsstärke verwendet. Im Folgenden werden einige empirische Modelle vorgestellt.

Free Space Modell

Das Free Space Modell ist eines der ältesten Modelle. Es wurde von Friis [19] vorgestellt. Dieses einfache Modell geht davon aus, dass es keine Hindernisse zwischen den Knoten gibt. Somit kann jeder Sender mit voller Stärke senden und die Signalstärke beim Empfänger hängt lediglich von der Distanz und deren Dämpfung zwischen den beiden Knoten ab. Es wird von einer direkten Verbindung ausgegangen. Dabei wird die Empfängerstärke P_r mit folgender Formel berechnet:

$$P_r = P_t \frac{G_r G_t \lambda^2}{(2\pi)^2 d^2 L}$$

P_t ist dabei die Signalstärke des Senders (in W). G_r und G_t beschreiben den Empfangswert der Antenne des Empfängers beziehungsweise des Senders (normalerweise 1). λ beschreibt die Wellenlänge (in m) welche als Simulationsparameter bekannt sein sollte. L ist der Verlustwert im System, wodurch die Umwelt mit einbezogen wird. d beschreibt den Abstand zwischen Sender und Empfänger. Abbildung 10(a) zeigt das Free Space Modell. Dieses Modell wird verwendet um Verbindungen auf einem freien Gebiet zu berechnen. Für Stadtgebiete ist es allerdings nicht geeignet, da keinerlei Hindernisse mit einbezogen werden. Ist die Reichweite zu groß wird das Ergebnis zu ungenau, da nur der direkte Strahl zur Berechnung verwendet wird.

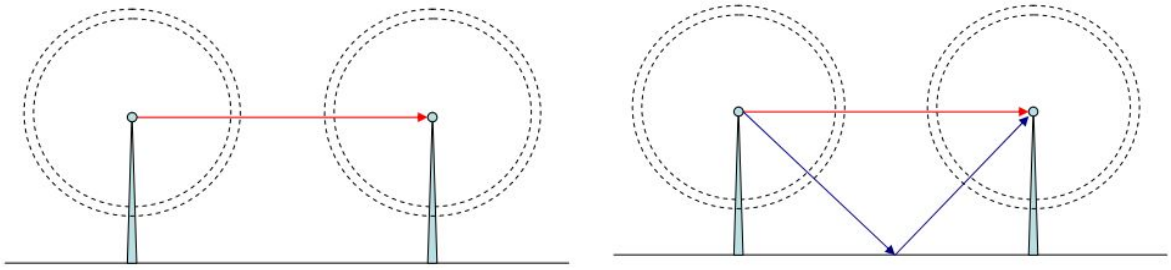
Two-Ray Ground Modell

Eine Erweiterung dieses Modells ist das Two-Ray Ground Modell [20]. Dabei wird neben dem direkten Weg die Reflektion vom Boden mit einbezogen, wodurch das Signal stärker wird. Dadurch ergibt sich eine leicht veränderte Formel gegenüber dem Free Space Modell.

$$P_r = P_t G_r G_t \frac{h_t^2 h_r^2}{d^4 L}$$

Die Formel berücksichtigt dabei die Höhe h_r des Senders und h_t des Empfängers, da von diesen beiden Faktoren die Stärke der Reflektion am Boden abhängt. Abbildung 10(b) veranschaulicht das Two-Ray Ground Modell anhand eines Senders und Empfängers.

Dieses Modell ist ebenfalls nicht dafür geeignet die Verbindung innerhalb einer Stadt zu überprüfen. Dadurch, dass neben der direkten Verbindung ebenfalls die Reflektion des Bodens mit einbezogen wird, ist dieses Modell für größere Distanzen genauer als das Free Space Modell. Auf kurze Entfernung ist die erhaltene Empfangsstärker zu hoch.



(a) Free Space Modell

(b) Two-Ray Ground Modell

Abbildung 10: Vergleich des „Free Space“ Modells und „Two-Ray Ground Modelle“

Kombination

In der Praxis wird häufig eine Kombination [13] aus dem Free-Space Modell und dem Two-Ray Ground Modell verwendet. Dies liegt daran, dass die Ergebnisse des Two-Ray Ground Modells bei einer größeren Entfernung genauer sind als es bei dem Free-Space Modell der Fall ist. Für eine verbesserte Genauigkeit wird eine Grenzentfernung d_c definiert, ab welcher Distanz das Two-Ray Ground Model verwendet werden soll. d_c wird dabei folgendermaßen berechnet:

$$d_c = \frac{4\pi h_t h_r}{\lambda}$$

Abbildung 11 zeigt die die Kombination aus beiden Modellen. Die Distanz, bei der beide Modelle die gleichen Ergebnisse liefern, entspricht der Grenzentfernung d_c, bei der zwischen dem „Free Space“ und dem „Two-Ray Ground“ Modell gewechselt wird.

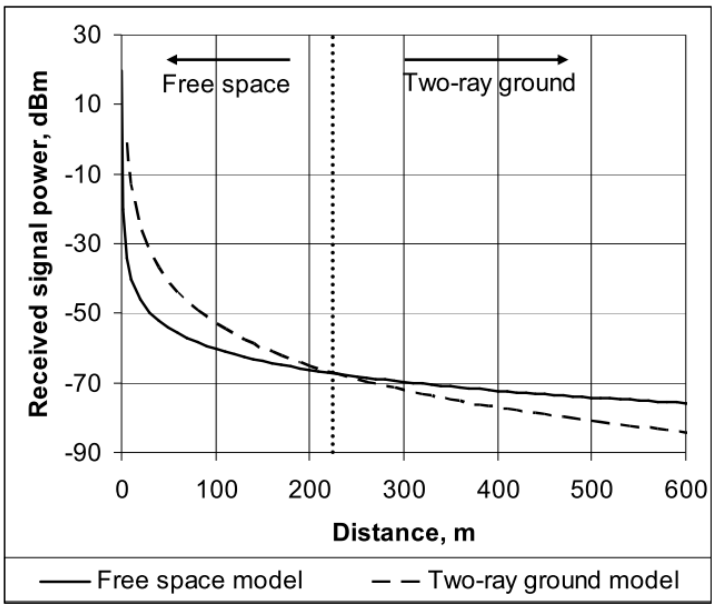


Abbildung 11: Free Space und Two-Ray Ground Modell aus [13]

Log-Distance Path-Loss Modell

Für die Simulation von ländlichen Gebieten ohne Hindernisse zwischen Sende- und Empfangsknoten sind die vorherigen Modelle ausreichend. Um Hindernisse, wie zum Beispiel Häuser in einer Stadt, zu berücksichtigen, können die einfacheren Modelle um einen zusätzlichen Dämpfungsterm erweitert werden. Das Log-Distance Path-Loss Modell [21] berechnet die Signalstärke eines Empfängers P_r aus, der mittels eines vorherigen Modells die bestimmte Empfangsstärke P_{r0} um einen zusätzlichen, distanzabhängigen Dämpfungsterm erweitert wird:

$$P_r = P_{r0} \left(\frac{d_0}{d}\right)^{\beta}$$

β gibt an, wie stark sich die Empfängerstärke - abhängig von der Entfernung - reduzieren soll. Für ein Free Space Modell wird meist $\beta = 2$ verwendet. Wenn man das Modell für eine Stadt heranzieht wird ein Wert zwischen 2.5 und 3.5 gewählt. Eine Netzwerkkarte hat typischerweise einen Wert von 2.7 für ein normales Szenario unter freien Himmel. Innerhalb eines Gebäudes kann für β ein Wert bis zu 6 angenommen werden. Genutzt wird dieses Modell vor allem um die räumlichen Gegebenheiten mit einzubeziehen. Für ein offenes Szenario ohne Hindernisse liefert dieses Modell zu schlechte Ergebnisse.
Durch das Einbeziehen einer Dämpfung ist es möglich die Sendereichweite zu reduzieren. Die Sendestärke hängt weiterhin von dem anderen Modell ab, welches den Wert P_{r0} berechnet. Allerdings kann ein Gebiet innerhalb eines Gebäudes simuliert werden, indem β entsprechend gewählt wird. Die Ausbreitung bleibt dabei weiterhin in alle Richtungen gleich. Um exakte Ergebnisse zu erhalten - sofern Hindernisse existieren - sind diese Modelle nur bedingt geeignet.

2.4.3 Deterministische Modelle

Neben den empirischen Modellen existiert eine zweite Gruppe von Modellen - die deterministischen Modelle. Der große Vorteil der empirischen Modelle ist ihre Einfachheit und ihre damit verbundene geringe Komplexität beim Berechnen. Um zu berechnen, ob eine Verbindung zwischen zwei Knoten existiert, wird nur ein Schwellwert verwendet und es wird überprüft, ob dieser oberhalb einer gewissen Grenze liegt. Dabei werden Hindernisse, die zwischen diesen Knoten liegen, nicht exakt mit einbezogen.
Die deterministischen Modelle setzen genau an diesem Punkt an. Deterministische Modelle [22] verwenden strahlenoptische Ansätze, um zu überprüfen, ob eine Verbindung zwischen einem Sender und einem Empfänger vorhanden ist oder nicht.

Die Anzahl der erreichten Strahlen stellen ein Maß für die Signalstärke dar, welche den Empfänger erreicht.

Ray Launching

Die Idee von Ray Launching [23] ist, dass von einem Sender Strahlen in alle Richtungen gesendet werden. Dabei wird zuerst ein Strahl mit einem bestimmten Winkel ausgesendet und überprüft, ob dieser Strahl einen Sender trifft. Daraufhin wird der vorherige Winkel um einen Betrag $\Delta\varphi$ erhöht und ein neuer Strahl gesendet. Ein Problem an Ray Launching ist, dass jeder Strahl bis zu einer bestimmten Anzahl von Interaktionen (Reflektionen, Transaktionen usw) nachverfolgt werden muss und bei jeder neuen Transaktion sehr viele neue Strahlen hinzukommen. Wird die Schrittweite $\Delta\varphi$ zu groß gewählt, ist das Risiko größer, dass Strahlen Hindernisse nicht treffen.

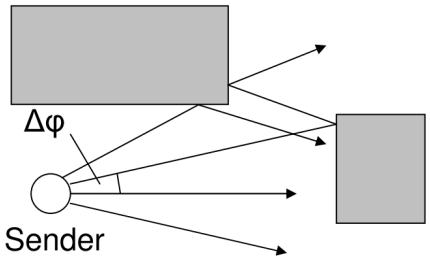


Abbildung 12: Beispiel Ray Launching

Ray Tracing

Dieses Modell [23] versucht alle Strahlen zu finden, die zwischen einem Sender und einem Empfänger existieren. Hierfür wird die „image theory“ [24] angewendet. Eine Anahme der „image theory“ besagt, dass die Energie aller Strahlen, welche ähnliche Interaktionen mit Hindernissen durchführen, identisch ist. Dadurch wird die Zahl der zu überprüfenden Strahlen reduziert. In der Abbildung 13 sieht man die Strahlen zwischen einen Sender und einem Empfänger.

Für jeden Strahl werden die Transmissionen, die Beugungen und die Reflektionen mit einbezogen. Jede dieser Interaktionen mit Hindernissen, die den Strahl verändern, sorgt dafür, dass dieser schwächer wird. Sinkt dieser Wert unter eine bestimmte Grenze wird das Signal zu schwach und nicht mehr weiterverfolgt. Wichtige Parameter sind dabei die Anzahl der Interaktionen und die Anzahl der Knoten, die in dem Gebiet liegen. Der Rechenaufwand hängt dabei exponentiell von der Anzahl der Interaktionen ab. Dabei haben sich 6 Interaktionen [13] (mit maximal 2 Beugungen) als geeignet herausgestellt

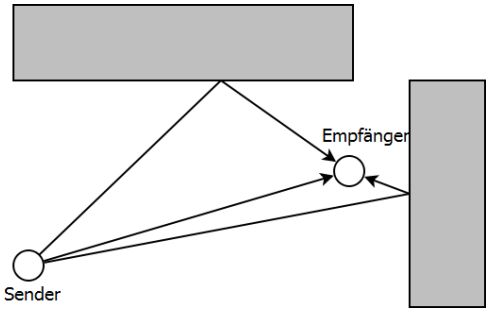


Abbildung 13: Beispiel Ray Tracing

Für die Berechnung der Empfangsstärke wird die vom Free Space Modell bekannte Formel zusammen mit der Dämpfung, die aus der Refraktion, Beugung und der Transmission hervorgeht, benützt.

$$P_k = P_t \frac{G_r G_t \lambda^2}{(2\pi)^2 d^4 L} \prod_i \tau_i \prod_j \rho_j \prod_l a_l$$

P_k beschreibt die Signalstärke des Strahls und P_t die Signalstärke der Übertragung (beide in W). G_r und G_t beschreiben den Empfangswert der Antenne des Empfängers beziehungsweise des Senders (normalerweise 1). λ beschreibt die Wellenlänge (in m). L ist der Verlustwert im System, um die Umwelt mit einzubeziehen. d beschreibt die vom Strahl zurückgelegte Strecke. τ_i beschreibt den i-ten Transmissionsverlust. ρ_j beschreibt den j-ten Reflektionsverlust und a_l beschreibt, den l-ten Beugungsverlust. Für größere Entfernungen bietet sich auch hier wieder das Two-Ray Ground Modell anstelle des Free Space Modells an.

Um die Gesamtstärke für einen Empfänger zu berechnen müssen alle Strahlen, die an einem Empfänger angekommen sind, aufaddiert werden.

$$P_r = \sum_k P_k$$

2.4.4 Vergleich empirisches und deterministisches Modell

Empirische Modelle haben den Vorteil, dass sie relativ einfach sind und recht schnell berechnet werden können. Dadurch sind sie gut in allen möglichen Simulationen einsetzbar. Diese Modelle sind allerdings für Gebiete, in denen es Hindernisse gibt, zu ungenau.
Bei dem Two-Ray Ground Modell kann man die Reichweite des Senders als einen Kreis darstellen, da die einzige Komponente der Abstand zum Sender ist. Dies zeigt die Ungenauigkeit dieses Modells sobald Hindernisse im Gebiet hinzukommen (siehe Abbildung 14(a)).

Die deterministischen Modelle können diese Hindernisse mit einbeziehen und dadurch bessere Ergebnisse wiedergeben. Allerdings ist der Aufwand für diese Berechnung deutlich höher als für die empirischen Modelle. Abbildung 14(b) zeigt das Ergebnis eines strahlenoptischen Modells.
Sofern keine Hindernisse zwischen dem Sender und dem Empfänger liegen, erreicht das Two-Ray Ground Modell ähnliche Ergebnisse wie das strahlenoptische Modell [13].

(a) Two-Ray Ground Modell

(b) Strahlenoptisches Modell

Abbildung 14: Die beiden unterschiedlichen Verbindungsmodelle aus [13]

3 Systementwurf

Nachdem die wichtigsten Rahmenbedingungen geklärt wurden kann jetzt beleuchtet werden, wie das Framework diese Informationen verwenden kann. Hier wird darauf eingegangen, wie die Informationen über die verschiedenen Bewegungsmodelle und die Informationen über die unterschiedlichen Verbindungsmodellen verwendet und zusammengefasst werden können. Desweiteren wird auf die Schnittstelle eingegangen, die für die Interaktion zwischen NET und dem Framework notwendig sind.

3.1 Umsetzung der Bewegungsmodelle

Als Erstes wird auf die Bewegungsmodelle eingegangen. Dieses ist für die Bewegung der Knoten innerhalb der Simulation zuständig. Um diese Modelle möglichst einfach in das Framework zu integrieren, müssen diese auf Gemeinsamkeiten überprüft und zusammengefasst werden. Hierfür wird die Gruppierung verwendet, die in Kapitel 2.3 eingeführt wurde. Dadurch werden Geschwindigkeit, Wahrscheinlichkeit, Bewegung und Umgebung genauer untersucht.

Geschwindigkeit

Für die Geschwindigkeit existieren drei Gruppen - die konstante, veränderbare und vorgezeichnete Geschwindigkeit.
Bleibt eine Geschwindigkeit konstant, so muss das Framework nur zu Initialisierung der Knoten diesen Wert setzen, was technisch einfach realisierbar ist. Bei Modellen, die eine Geschwindigkeitsänderung zulassen, muss das Modell entscheiden, wann dies der Fall ist. Für Änderungen, die zu einem bestimmten Zeitpunkt gedacht sind (etwa das Ziel erreicht oder die Entfernung zurückgelegt wird), wählt das Modell die Geschwindigkeit neu. Bei einer schrittweisen Anhebung muss neben der Geschwindigkeit auch mit angegeben werden mit welcher Schrittgröße der Wert hin zu der neuen Geschwindigkeit erhöht werden soll.
Bei Traces ist das Setzen der Geschwindigkeit identisch - allerdings fest von den Traces vorbestimmt.
Für alle Modelle gilt, dass die Geschwindigkeit für die Bewegung zur Fortbewegung verwendet wird, das Wählen der Geschwindigkeit allerdings abhängig von dem einzelnen Modell übernommen werden muss. Damit kann für die Entwicklung des Frameworks davon ausgegangen werden, dass die Geschwindigkeit bekannt ist und direkt verwendet werden kann.

Bewegung

Die Modelle können bezüglich der Bewegung in zwei Untergruppen unterteilt werden. In der ersten Gruppe bewegen sich die Knoten auf einen Punkt zu, während in der zweiten Gruppe eine Richtung gewählt wird. Wann ein neues Ziel gewählt wird hängt dabei von einer Distanz oder einer Zeit ab.
Die Bewegung eines Knotens innerhalb eines Modells verhält sich immer gleich. Es bewegt sich mit einer Geschwindigkeit auf etwas zu. Eine Bewegung auf einen Punkt kann dabei ebenso als eine Richtung angesehen werden, an der der Punkt entlang laufen soll. Es wird ein Vektor berechnet, der die Richtung zu dem Punkt angibt. Somit kann die Bewegung auf einem Punkt als eine Bewegung in eine Richtung angesehen werden, wodurch sich alle Bewegungsmodelle über Bewegungen in eine Richtung darstellen lassen. Der Unterschied besteht darin, wann ein Knoten am Ziel angekommen ist.
Es existieren drei mögliche Bedingungen für die Erreichung eines Ziels durch einen Knoten. Ein Ziel wurde entweder erreicht, wenn eine bestimmte Zeit abgelaufen ist, eine bestimmte Strecke zurückgelegt wurde oder ein bestimmter Punkt erreicht wurde. Diese Bedingungen müssen wieder von den einzelnen Bewegungsmodellen überprüft werden.

Für die Behandlung von Sonderfällen müssen zusätzliche Überlegungen angestellt werden. Ist die Grenze des Simulationsgebietes erreicht, muss das Modell entscheiden, wie der Knoten sich verhalten soll (vom Rand abprallen - stehenbleiben - auf der anderen Seite des Gebietes wieder eintreten). Für die schrittweise Erhöhung des Winkels (zur Vermeidung von Knicken), um die neue Berechnung durchzuführen, ist ebenfalls das Modell zuständig und muss deshalb extra behandelt werden.
Existierende Traces können auch auf eine Richtung in die der Knoten laufen soll reduziert werden. Die Traces geben dabei die Punkte vor, die nacheinander angesteuert werden sollen.

Umgebungseinfluss

Für den Einfluss der Umgebung existieren zwei Gruppen von Modellen. In den Modellen der ersten Gruppe können sich die Knoten frei auf einem Simulationsgebiet bewegen. Somit ist - abgesehen von den Grenzen des Gebietes - keine weitere Einschränkung notwendig. Die Knoten bewegen sich - abhängig von ihren vorgeschriebenen Richtungen, Geschwindigkeiten und Zielen - direkt dorthin. Hierfür benötigt man nur einen Algorithmus, um den Knoten in eine bestimmte Richtung zu bewegen.
Die andere Gruppe der Bewegungsmodelle bezieht die räumlichen Umgebungen mit ein. Ein Knoten kann sich dadurch nicht mehr frei in dem gesamten Gebiet bewegen, sondern muss sich auf vorgegebenen Wegen von Punkt zu Punkt bewegen. Die Ziele, die sich ein Knoten aussuchen kann, sind dadurch auf Ziele begrenzt zu denen ein solcher Weg führt. Für die Darstellung eines solchen Szenarios eignet sich ein Graph.
Eine Stadt kann beispielsweise durch einen Graphen modelliert werden, indem jede Kreuzung, an der mehrere Straßen zusammenlaufen oder ein wichtiger Zielort (Einkaufsladen, Kino etc.) existiert, als Graphknoten dargestellt. Die Straßen zwischen diesen Punkten bilden dabei die Kantenmenge. Dadurch wird ein Straßennetz in einem Graphen abgebildet.
Jeder Knoten in diesem System wählt - abhängig von seinem Bewegungsmodell ein Ziel, welches einem Knoten in dem Graph entspricht. Durch einen kürzesten Wege Algorithmus - wie zum Beispiel Dijkstra oder $A*$ - wird der nächste Zwischenpunkt gewählt. Ein Knoten bewegt sich dadurch von Knoten zu Knoten, bis das endgültige Ziel erreicht ist. Jeder Knoten gibt bei einer Zielanfrage dabei nur den nächsten Nachbarknoten an, der besucht werden soll. Ein Knoten bewegt sich dann auf direktem Weg zu diesem Ziel hin. Sollen Kurven abgebildet werden, so müssen diese durch Hinzunahme von weiteren Knoten angenähert werden.

Wahrscheinlichkeiten

Die Wahl der Geschwindigkeit, sowie die Wahl der Richtung oder des Zieles erfolgt bei den meisten Modellen zufällig. Allerdings sorgt dies dafür, dass Charakteristika von Personen ignoriert werden. Sollen diese Charakteristika in das Modell mit einfließen, werden Wahrscheinlichkeiten verwendet. Das Modell selbst muss dies bei der Wahl von neuen Zielen und Richtungen sowie Geschwindigkeiten durchführen. Die Verteilung, die dafür verwendet werden soll, muss in dem Modell vorhanden sein.
Um Wahrscheinlichkeiten mit einzubeziehen müssen dem Modell Wahrscheinlichkeiten vorliegen, um so Entscheidungen zu treffen. Zielpunkte oder Richtungen werden mit bestimmten Wahrscheinlichkeiten belegt. Damit werden manche Zielpunkte oder Richtungen häufiger gewählt als andere.

Zusammenfassung

Die Probleme lassen sich in zwei große Gruppen unterteilen. Dabei handelt es sich einerseits um die Modelle, in denen sich die Knoten nur auf vorgefertigten Wegen bewegen können und andererseits um Modelle, bei denen sich die Knoten frei auf einem Feld bewegen können. Ist das Gebiet nicht frei so muss die Wahl, wie ein Knoten sich bewegen kann, eingeschränkt werden und weiter behandelt werden. Hierfür wird ein Graphansatz benötigt, um diese Einschränkung widerzuspiegeln. Für die eigentliche Bewegung lassen sich alle Modelle auf eine Richtung reduzieren in der sich ein Knoten bewegen soll. Die Modelle entscheiden wann eine Änderung der Richtung vorgenommen werden muss. Dies kann zum Beispiel durch das Erreichen eines Zieles, nach einer verstrichenen Zeit oder nach einer zurückgelegten Zeit erfolgen.

3.2 Umsetzung der Verbindungsmodelle

Nachdem die Bewegungsmodelle und deren Einfluss auf das Framework vorgestellt wurden, wird nachfolgend auf die Verbindungsmodelle eingegangen. Um eine Verbindung zwischen Knoten zu ermöglichen muss überprüft werden welche Schnittstellen dafür bereitgestellt werden. Hierfür wird auf Gemeinsamkeiten überprüft. Wie in Kapitel 2.4 erwähnt können die Verbindungsmodelle in zwei große Gruppen unterteilt werden, die deterministischen und die empirischen Modelle.

Empirische Modelle

Bei den empirischen Modellen wird die Empfangsstärke mit Hilfe einer Formel berechnet. Diese Modelle vergleichen die berechnete Empfangsstärke mit einem Schwellenwert. Ist die Empfangsstärke größer als dieser Wert gelten sie als verbunden. Im Falle des „Free Space“ Modells hängt die Empfangsstärke lediglich von der Distanz ab. Der Schwellenwert kann hier demnach dazu verwendet werden um auszurechnen bis zu welcher Distanz ein Knoten von dem Sender erreicht werden kann.
Damit kann die Überprüfung als ein Kreis dargestellt werden innerhalb dessen alle Knoten mit dem Sender verbunden sind. Die notwendigen Überprüfungen können dadurch auf Knoten innerhalb dieses Kreises reduziert werden.
Gibt es innerhalb der Formel Parameter, die nicht global gleich sind sondern von dem Empfänger abhängen, muss eine obere Grenze gewählt werden, damit die maximale Sendereichweite als Einschränkung verwendet werden kann.

Deterministische Modelle

Bei den deterministischen Modellen wird die Berechnung mit Hilfe von strahlenoptischen Modellen durchgeführt. Dies hat zur Folge, dass man nicht mit Hilfe eines Kreises die Reichweite eines Senders beschreiben kann - sofern sich Hindernisse auf dem Gebiet befinden. Für jeden Sender muss die Überprüfung aufs Neue berechnet werden.
Diese Berechnung kann während der Laufzeit nicht sinnvoll durchgeführt werden, da sie zu lange braucht, um dies für einen Sender durchzuführen. Aus diesem Grund müssen die Daten für die deterministischen Modelle vor der eigentlichen Emulation bereitgestellt werden. Allerdings benötigt man trotzdem eine Begrenzung bis zu welchem Abstand ein Sender einen Empfänger erreicht um damit eine Begrenzung der maximalen Sendereichweite zu erhalten.

Hierfür muss für jeden Sender der Empfänger bekannt sein, der am weitesten entfernt ist. Dies muss für jede mögliche Sendeposition durchgeführt und gespeichert werden. Dadurch hat man erneut einen Radius um den Sender den man für die Berechnung verwenden kann.
Ist das nicht möglich kann alternativ ein empirisches Modell verwendet werden, um so die maximale Distanz zu berechnen. Die empirischen und deterministischen Modelle verhalten sich für Gebiete ohne Hindernisse fast identisch.

Zusammenfassung

Die Überprüfung, ob eine Verbindung zwischen zwei Knoten existiert oder nicht, muss von jedem Verbindungsmodell durchgeführt werden und kann nicht verallgemeinert werden. Allerdings kann in allen Modellen eine maximale Sendereichweite festgestellt werden, die das zu überprüfende Gebiet einschränkt. Für die Verbindungsüberprüfung wird bei den empirischen Modellen eine Formel verwendet. Bei den deterministischen Modellen sind aufgezeichnete Daten notwendig. Die Daten können entweder in einer Stadt per Messung aufgezeichnet werden oder ebenfalls durch eine Simulation erfasst werden.

3.3 Gebietsaufteilung

Nachdem nun diskutiert wurde, wie Knoten bewegt werden und mit welchen Modellen sich Knoten verbinden können, bleibt die Frage wie dies innerhalb des Frameworks umgesetzt werden kann. Jeder Knoten innerhalb der Emulation soll bewegt werden, außerdem soll für jeden Knoten überprüft werden, ob eine Verbindung mit einem anderen Knoten besteht.
Für die Bewegung gilt, dass sich jeder Knoten einzeln bewegt, somit muss die Bewegung für jeden Knoten einzeln durchgeführt werden. Wird dieses Konzept bei den Verbindungmodellen angewendet, so ergibt sich, dass jeder Knoten mit allen anderen Knoten überprüft werden muss. Für wenige Knoten ist diese Vorgehensweise möglich. Für eine große Anzahl von Knoten wird allerdings schnell klar, dass dies nicht durchführbar ist. Für 10 Knoten müssten 100 Vergleiche durchgeführt werden, bei 100 Knoten wären es 10.000 und bei 1.000 Knoten wären es schon 1.000.000 Vergleiche pro Durchgang. Es handelt sich um eine Laufzeit von $O(n^2)$.
Dadurch erreicht eine Verbindungsüberprüfung von allen Knoten untereinander sehr schnell seine Grenzen.

Damit die Anzahl der notwendigen Vergleiche reduziert werden kann, benötigt man Ansätze, die die Menge der zu überprüfenden Knoten einschränken.
Für die Einschränkung wird ausgenutzt, dass jeder Knoten nur innerhalb einer bestimmten Entfernung von einem Sender aus erreicht werden kann, die abhängig von dem verwendeten Verbindungsmodell ist. Es sind nur die Knoten interessant, die sich in diesem Gebiet befinden. Knoten, die sich außerhalb des Kreises befinden, kommen als Verbindungskandidaten nicht in Frage (siehe Abbildung 15 - S ist der Sender).
Für die empirischen Modelle ist dies ein Kreis um den Sender.
Für die deterministischen Modelle ergibt sich ebenfalls ein Kreis, sofern keine Hindernisse existieren. Sind Hindernisse vorhanden, so ist die Sendereichweite kein Kreis, sondern es können sich beliebige Formen bilden. Da sich deterministische Modelle und empirische Modelle für Gebiete gleichen, in denen es keine Hindernisse gibt, kann hier als Abschätzung ebenfalls ein Kreis verwendet werden. Dadurch kann das Gebiet auf einen Kreis um den Sender eingeschränkt werden.

Die Aufgabe besteht darin die Knoten innerhalb dieses Kreises zu überprüfen. Um das zu erreichen ist es notwendig das gesamte Gebiet, in dem sich die Knoten befinden, in kleinere Teilgebiete zu unterteilen. Damit muss man nur die Teilbereiche überprüfen, die sich innerhalb der Sendereichweite des Senders befinden und somit die Knoten, welche sich innerhalb dieser Teilgebiete befinden.
Hier stellt sich die Frage, wie man am sinnvollsten das Gebiet aufteilen kann. Neben der Überprüfung, welcher Knoten mit anderen eine Verbindung aufbauen kann, muss auch berücksichtigt werden, dass jeder Knoten sich bewegen und somit von einem Teilgebiet in ein neues wechseln kann.

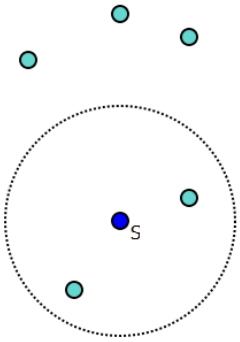
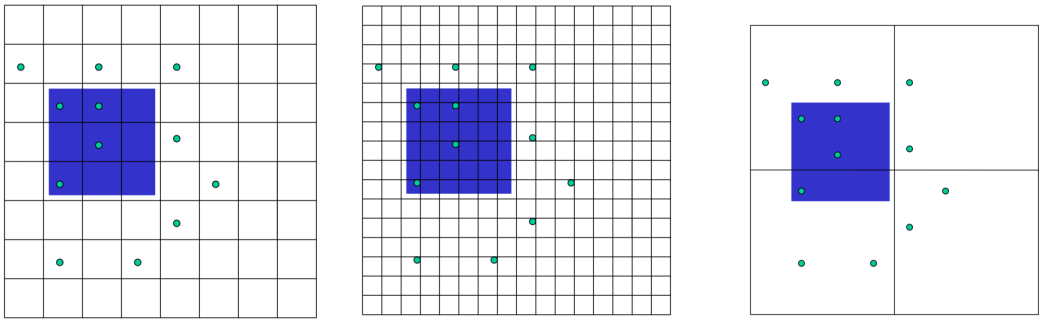


Abbildung 15: Reichweite eines Senders

Die Struktur welche das Gebiet aufteilt wird im folgenden als **Gebietsmanager** bezeichnet.
Der Gebietsmanager muss das Gebiet in Teilgebiete unterteilen und den Knoten die Möglichkeit geben diese Teilgebiete zu wechseln. Er muss alle Knoten finden die innerhalb einer Gebietsanfrage liegen.

Der Gebietsmanager muss das gesamte Gebiet (es wird von einem rechteckigen Gebiet ausgegangen) in kleinere Teilgebiete aufteilen und die Knoten diesen Teilgebieten zuordnen. Damit kann er überprüfen, welcher Knoten mit welchem verbunden ist und diejenigen Knoten einschränken, die für eine Verbindung in Frage kommen. Jeder Knoten wird danach einem dieser Teilgebiete zugewiesen. Die Teilgebiete sind dabei ebenfalls rechteckig. Stellt ein Sender die Anfrage mit welchen Knoten dieser verbunden ist, werden die Teilgebiete überprüft, die nahe genug an dem Sendeknoten liegen.
Für jeden Sender wird der Radius des Kreises als maximale Distanz zwischen zwei Knoten verwendet. Der Kreis ergibt sich aus den Verbindungsmodellen und gibt an wie weit ein Sender von den Empfängern maximal entfernt sein darf.
Der Gebietsmanager liefert einem Sendeknoten alle Knoten aus den Teilgebieten, die sich innerhalb der maximalen Sendereichweite befinden. Dabei spielt die Größe des Teilgebietes eine wichtige Rolle. Ist das Teilgebiet zu groß werden viele Knoten, die innerhalb dieses Gebietes liegen, kontrolliert, obwohl sie nicht innerhalb der maximalen Sendereichweite liegen. Sind die Teilgebiete zu klein müssen sehr viele Teilgebiete überprüft werden, was zu einem höheren Rechenaufwand führen kann und mehr Speicher verbraucht. Eine Beispielaufteilung mit Gebietsanfrage sieht man in Abbildung 16.

Für die Überprüfung, welche Knoten innerhalb des Senderradiuses liegen, wird statt eines Kreises ein Quadrat verwendet. Die Länge entspricht dem Radius des Kreises. Dies ist notwendig, um die Gebietsanfragen sinnvoll zu ermöglichen. Eine Gebietsanfrage kann mit einem Quadrat mit Hilfe der vier Eckpunkten durchgeführt werden. Eine Gebietsanfrage mit einem Kreis ist deutlich aufwändiger, da hier Schnittpunkte zwischen Kreis und Teilgebieten berechnet werden müssen.

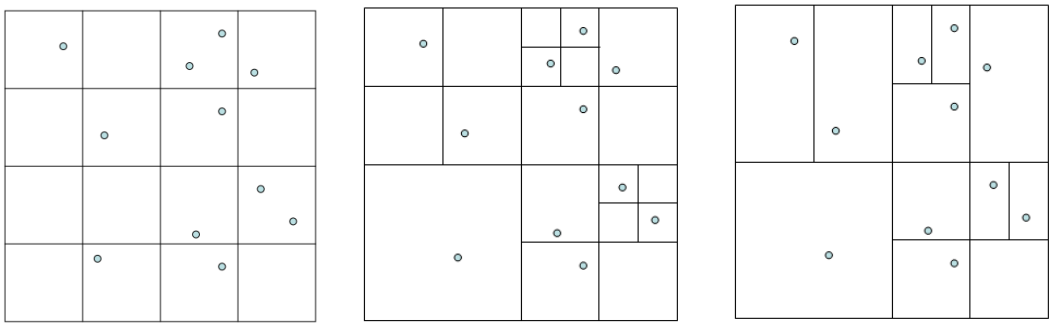


(a) gute Einteilung (b) zu kleine Gebiete (c) zu große Gebiete

Abbildung 16: Qualität verschiedener Aufteilungsmöglichkeiten

Zusammengefasst muss ein Gebietsmanager Folgendes leisten:

- Der Gebietsmanager hat als Eingabe die Größe des Gebietes und die Knoten, die sich in diesem Gebiet befinden, inne. Wie die Teilgebiete aufgeteilt werden ist bei jedem Gebietsmanager anders.
- Der Gebietsmanager muss auf eine Gebietsanfrage hin alle Knoten kennen, die sich innerhalb des gesuchten Gebietes befinden. Dies wird für die Durchführung der Verbindungsüberprüfung mit einem Sender benötigt.
- Bei der Bewegung des Knotens muss der Gebietsmanager prüfen, ob der Knoten sich noch in dem gleichen Teilgebiet befindet oder ob er einem anderen Teilgebiet zugewiesen werden muss.



(a) Grid (b) Quadtree (c) KD-tree

Abbildung 17: Verschiedene Arten der Aufteilung

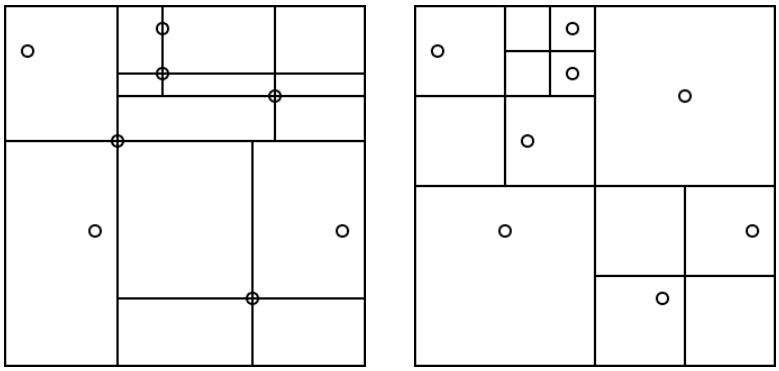
3.3.1 Grid

Eine einfache Aufteilung des Gebietes in Zonen bietet ein Grid, in dem das Gebiet in eine Anzahl von gleich großen Rasterzellen aufgeteilt wird. Dabei wird das Gebiet in eine vorbestimmte Anzahl von Gebieten unterteilt und jedem dieser Gebiete werden die Knoten zugeordnet, die in diesem Gebiet liegen. Jede Zelle kennt dabei seine Nachbarn. Beim Einfügen eines Knotens muss allerdings erst an einem Punkt in der Zelle begonnen werden, um diesen dann Stück für Stück an die Nachbarn weiterzugegeben, um so die korrekte Position zu finden. Für eine Optimierung bietet sich hier eine Baumstruktur an, mit der ein neues Element schneller eingefügt werden kann. Bewegt sich ein Knoten aus dem Gebiet heraus, so wird dieser direkt an den korrekten Nachbarn weitergegeben. Für die Überprüfung nach möglichen Kandidaten werden jeweils die Nachbarn der Zelle überprüft, in denen sich der Knoten befindet. Man erreicht dadurch eine direkte Überprüfung, ohne in einer Struktur nach der richtigen Zelle suchen zu müssen.

Ein Grid hat den Vorteil, dass die Nachbarn einer Zelle direkt bekannt sind. Somit benötigt man keinen großen Aufwand um die direkten Nachbarn zu finden. Der Nachteil ist, dass die Zelleneinteilung des Grids von vornherein fest ist und somit stark von der gewählten Größe abhängt. Wird die Größe der Zelle zu klein gewählt, so müssen bei der Gebietsüberprüfung zu viele Zellen überprüft werden. Ist die Größe einer Zelle zu groß so werden viele Punkte überprüft, die außerhalb der Sendereichweite liegen und eigentlich keine Prüfung erfordern.
Sofern eine maximale Größe des Grids bekannt ist, kann das Einfügen eines Knotens auch direkt mit Hilfe eines zweidimensionalen Arrays durchgeführt werden. Allerdings wird bei einer kleineren Größe des Grids Platz verschwendet, da nur ein Teil des Arrays verwendet wird. Es begrenzt ebenfalls die maximale Anzahl der Zellen.

3.3.2 Baumstrukturen

In dieser Arbeit werden zwei Arten von Baumstrukturen [1] vorgestellt. Dabei handelt es sich um den Quadtree und den KD-tree. Der Ansatz beider Strukturen ist dabei ähnlich. Das Gebiet wird abhängig von der Position der Knoten unterteilt. Gibt es in einem Bereich mehr Knoten, so wird dieser genauer unterteilt als es in einem Bereich mit wenig Knoten der Fall ist. Die Unterteilung geschieht dabei dynamisch. Sobald sich Knoten von einem Teilgebiet in ein anderes bewegen wird diese Unterteilung angepasst.
Es gibt eine gebietsweise und eine punktweise Einteilung (siehe Abbildung 18). In der gebietsweisen Unterteilung wird das Gebiet so unterteilt, dass dabei ein Gebiet in Teilregionen aufgeteilt wird. Im Normalfall wird die Aufteilung so vorgenommen, dass die Teilgebiete gleich groß sind. Existieren Informationen über ein Gebiet, so können die Teilgebiete unterschiedlich groß sein. Dies ist vor allem dann der Fall, wenn sich Hindernisse in dem Gebiet befinden. Befinden sich in einem Teilgebiet nichtbegehbare Hindernisse, kann man die Aufteilung so verschieben, dass dieser Bereich eine größere Fläche bei einer weiteren Unterteilung erhält, da sich in dieser Fläche weniger Knoten aufhalten können.
Die punktweise Aufteilung wird direkt an dem Knoten vorgenommen. Dies geschieht sobald sich innerhalb eines Gebietes zu viele Knoten befinden. Da sich die Knoten bewegen, muss hierfür die Baumstruktur immer wieder angepasst werden. Aus diesem Grund wird die punktweise Aufteilung nicht weiter verfolgt.
Für die Aufteilung bei den folgenden Stukturen werden damit nur gebietsweise Aufteilungen betrachtet, die die gleiche Größe besitzen.



(a) Quadtree nach Punkten aufgeteilt

(b) Quadtree nach Region aufgeteilt

Abbildung 18: Quadtree-Aufteilung

Quadtree

Ein Quadtree teilt ein Gebiet in vier gleich große Teile auf. In dem Baum entspricht jedes Gebiet einem Kind innerhalb des Baumes. Somit befinden sich an jedem Baumelement entweder gar keine oder genau vier Kinder. Wann ein Gebiet aufgeteilt wird hängt von der Anzahl der Knoten ab, die sich in dem Gebiet befinden. Existieren zu viele Knoten innerhalb eines Gebietes wird dieses erneut in 4 Teile aufgeteilt. Dabei hat jeder Quadtree eine maximale Tiefe. Ist diese Tiefe erreicht wird ein Gebiet nicht aufgeteilt.
Ein Beispiel für eine Aufteilung sieht man in Abbildung 19.

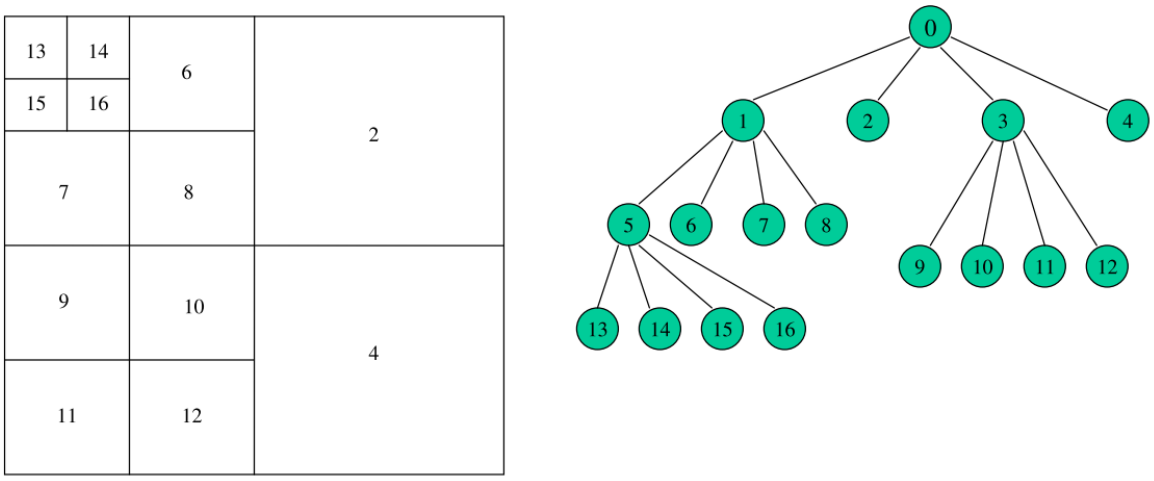


Abbildung 19: Aufteilung Quadtree

Inhalte werden dabei entweder in den Blättern oder in einem der Baumelemente gespeichert. Punkte können Gebieten zugeordnet werden, während dies bei Flächen nicht immer möglich ist. Sofern ein Objekt, welches größer als ein Punkt ist, komplett innerhalb eines Teilgebietes liegt, so kann es in diesem dazugehörigen Baumelement gespeichert werden. Befindet sich das Objekt in mehreren Gebieten, die zu unterschiedlichen Blättern gehören, werden diese dem Baumelement zugewiesen, der das Objekt komplett enthält. In dem zu entwickelnden Framework werden nur Punkte betrachtet, da Simulationsknoten immer durch einen Punkt dargestellt werden können.

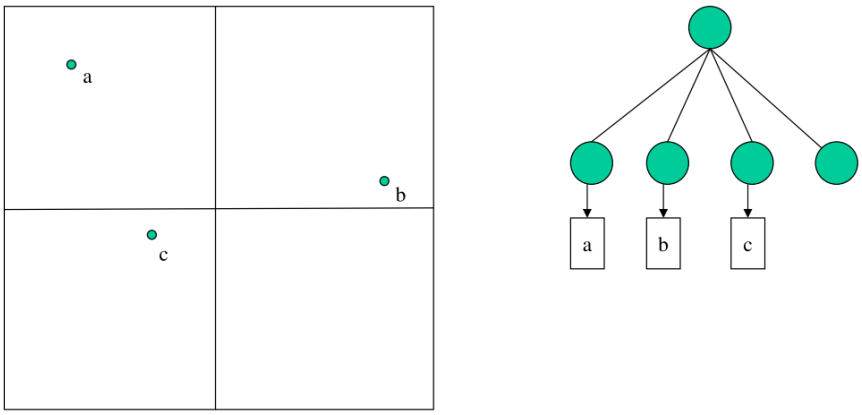


Abbildung 20: Quadtree mit drei Knoten

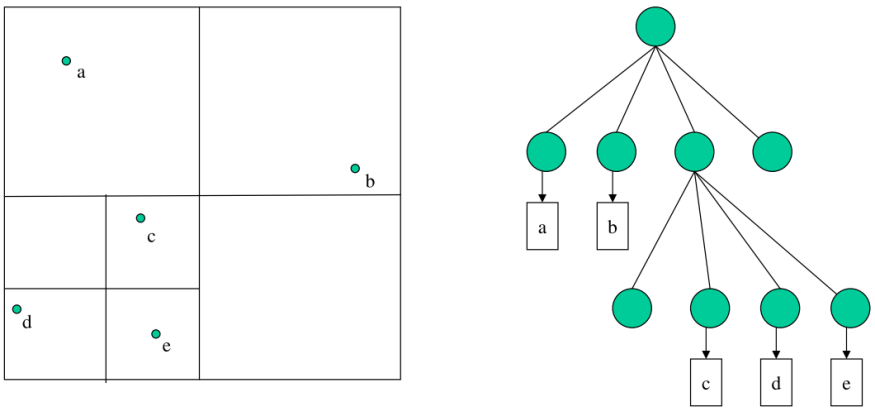


Abbildung 21: Quadtree: Hinzufügen von d und e

Beispiel:
In Abbildung 20 wird ein Quadtree gezeigt in dem drei Knoten eingefügt wurden. Es handelt sich dabei um einen Quadtree der sich dann aufteilt, wenn innerhalb eines Gebietes 2 Knoten vorhanden sind. Da sich drei Knoten in dem Quadtree befinden werden die Knoten dementsprechend in den Blättern des Quadtrees gespeichert.
In Abbildung 21 werden nun zwei weitere Punkte d und e hinzugefügt, was dafür sorgt, dass das Gebiet erneut aufgeteilt werden muss. Dieses Gebiet erhält dadurch vier weitere Teilgebiete. Die Punkte sind innerhalb eines Gebietes und werden damit beide an den dritten Knoten hinzugefügt.

Ein Quadtree behandelt ein Gebiet im Zweidimensionalen. Für die Behandlung eines dreidimensionalen Gebietes wird ein Octree verwendet. Dieser hat insgesamt acht Kinder, um die weitere Dimension mit einzubeziehen. Da hier nur der zweidimensionale Fall betrachtet wird, wird der Octree nicht weiter vorgestellt.

Der Quadtree hat den Vorteil, dass er dynamisch aufgebaut wird. Die Struktur hängt davon ab, wo sich die Elemente in dem Gebiet befinden. Existieren in einem Gebiet mehr Elemente, so wird hier der Baum tiefer werden als bei anderen Gebieten. Nachteil daran ist, dass Nachbarn nicht ohne Weiteres sichtbar sind und diese erst im Baum gesucht werden müssen, wofür es allerdings Ansätze gibt.
Für einen Quadtree sollte die maximale Tiefe sinnvoll gewählt werden. Ist die aufgeteilte Fläche zu klein bringt ein weiteres Aufteilen keine nennenswerten Vorteile mehr. Durch die kleine Fläche werden fast immer alle Elemente innerhalb dieser Unterteilung in Betrachtung gezogen. Deshalb würde eine weitere Aufteilung nur eine weitere Tiefe und zusätzlichen Speicherplatz verbrauchen.
Ein Einfügen in den Quadtree braucht eine Zeit von log(n). Allerdings nur wenn der Baum gleichmäßig aufgebaut wurde. Dies hängt stark von den Knoten und deren Positionen ab. Bei einem Modell, in dem sich die Knoten alle gleichmäßig über das Gebiet verteilen, führt dies automatisch zu einem balancierten Baum. Ein Bewegungsmodell, in dem sich die meisten Elemente nur in einem kleinen Teilbereich des Gebietes befinden, sorgt dafür, dass der Quadtree hier eine deutlich kleinere Aufteilung hat und somit dort eine größere Tiefe als an einem anderen Teilgebiet.
Der Aufwand einen Quadtree aktuell zu halten ist größer als bei einem Grid, da Knoten, die sich bewegen, dem Nachbargebiet zugewiesen werden müssen. Auch wenn es sich dabei um einen Gebietsnachbarn handelt heißt das nicht, dass es sich auch in der Baumstruktur um einen Nachbarn handelt. Deshalb müssen die Nachbarn neu in dem Quadtree gesucht werden.

KD-tree

Andere Ansätze, die ebenfalls mit einer Baumstruktur arbeiten, sind KD-trees. Die Idee von KD-trees ist die, dass ein Gebiet in zwei Teile aufgeteilt wird. Dabei wird die horizontale Aufteilung und die vertikale Aufteilung unterschieden. Wird im ersten Schritt horizontal aufgeteilt, so wird im nächsten der neue Sektor vertikal aufgeteilt. Die Ansätze, wie Gebiete aufgeteilt werden sind dabei unterschiedlich. Bei dem dynamischen KD-tree wird so aufgeteilt, dass sich auf beiden Seiten gleich viele Knoten befinden. Geht man von diesem Ansatz aus, so muss bei einer Bewegung jeder dieser Abschnitte neu berechnet und umsortiert werden. Aus diesem Grund kommt dieser Ansatz für das Framework nicht in Frage. Der normale KD-tree teilt dabei jedes Gebiet in zwei gleich große Gebiete auf. Dadurch ist eine Umsortierung nicht bei jedem Schritt notwendig. Erst wenn viele Elemente das Gebiet verlassen haben sollte die Ummodellierung durchgeführt werden.

Sowohl ein Quadtree als auch der KD-tree sind für das Suchen nach Punkten (und somit Empfangsknoten) innerhalb der Struktur geeignet. Allerdings sind diese Modelle nicht ideal für Bewegungen, da die direkten Nachbarn nicht direkt angegeben werden. Hierfür würde es sich anbieten zusätzliche Informationen zu jedem Gebiet zu speichern um die Nachbarschaftssuche schneller zu gestalten.

Beispiel:
In der Abbildung 22 werden wie beim Quadtreebeispiel die drei Punkte a, b und c hinzugefügt. Auch hier gilt, dass ein Gebiet erst dann aufgeteilt wird, wenn sich in einem Gebiet zwei Punkte befinden. Da es sich hierbei um einen KD-tree handelt wird das Gebiet in der Mitte vertikal aufgeteilt und die Punkte einem der beiden Baumelementknoten zugewiesen. Da sich jetzt im linken Teilgebiet ebenfalls zwei Punkte befinden muss dieses Teilgebiet erneut aufgeteilt werden. Dies geschieht in horizontaler Richtung.
In Abbildung 23 werden zwei weitere Punkte hinzugefügt. Durch das Hinzufügen des Knotens d befinden sich auf dem Teilgebiet zwei Punkte, wodurch eine neue Aufteilung in horizontaler Richtung durchgeführt wird. Durch das Hinzufügen des Knotens e muss eine weitere Aufteilung vorgenommen werden, damit die maximale Anzahl von Punkten pro Gebiet nicht überschritten wird.

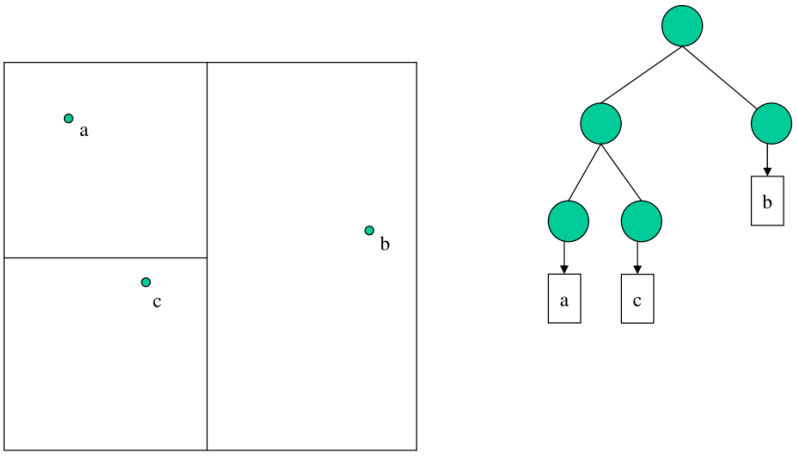


Abbildung 22: KD-tree mit drei Knoten

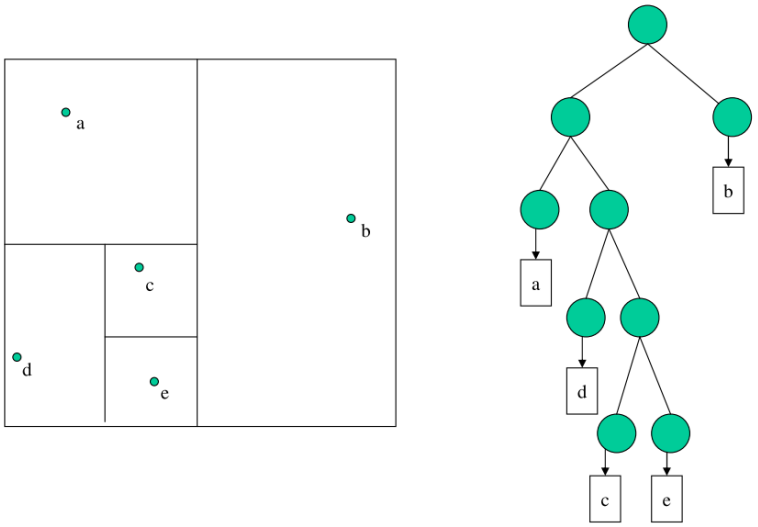


Abbildung 23: KD-tree: Hinzufügen von Knoten d und e

3.4 Verteilung auf dem NET

Das NET besteht aus einem Frontend, von dem aus die Simulation gesteuert wird, und den verschiedenen virtuellen Maschinen mit den darauf liegenden virtuellen Knoten. Durch das Frontend kann man auf die mehreren physischen und den darin liegenden virtuellen Maschinen zugreifen. Somit ergibt sich ein Bild aus einem Server und mehreren Clients. Das Framework muss auf dem Server und den Clients vorhanden sein. Die Aufteilung der Arbeit - sofern dies möglich ist - sorgt für eine schnellere Abhandlung der Simulation und erhöht die maximale Anzahl der Knoten, die behandelt werden können. Allerdings ist diese Aufteilung nicht bei allen Modellen möglich.
Werden alle Berechnungen auf dem Server ausgeführt, so müssen die Daten der Knoten auf den Clients nach der Berechnung aktualisiert werden. Auf den Clients müssen dann keinerlei Berechnungen durchgeführt werden und nur die Daten gespeichert werden, die auf der virtuellen Maschine laufen, um so die Verbindungs- und Positionsinformationen zu übertragen.

3.4.1 Bewegungsaufteilung

Es stellt sich die Frage wann die Aufgaben auf die Clients verteilt werden können und wann nicht. Hierfür wird die Aufteilung aus den verschiedenen Bereichen (Geschwindigkeit, Wahrscheinlichkeit, Bewegungsrichtung und den Umgebungseinfluss) verwendet. So kann man entscheiden, welche Bereiche der Bewegungssimulation auf den Clients durchgeführt werden können und welche nur auf dem Server mit globaler Sicht ausgeführt werden sollten. Die Bewegung erfolgt dabei immer identisch. Ein Knoten bewegt sich mit einer gewissen Geschwindigkeit in eine gewisse Richtung. Bei der Wahl einer neuen Richtung oder eines neuen Zieles muss das Modell entscheiden. Die einfachen Modelle verwenden dafür Zufallsalgorithmen dessen Werte zwischen zwei Grenzen liegen. Werden diese Daten durch Traces eingelesen so muss dafür gesorgt werden, dass diese Daten den Clients vorliegen. Bei Modellen, die die Umgebung in Betracht ziehen, benötigt man ein System um das Modell einzulesen und bei der Wahl für ein Ziel einzuschränken. Die Daten des Gebietes können entweder zufällig erstellt werden oder von einer vorhandenen Stadt gesammelt werden. Abhängig von der Datengröße des Gebietes kann eine Vervielfältigung der Daten auf allen Clients nicht möglich, oder eine zu große Verschwendung von Ressourcen sein. Wenn Bewegungen auf den Clients berechnet werden sollen, muss die Karte, die zur Bewegung benötigt wird, jedem Client zur Verfügung gestellt werden. Sofern sich innerhalb der virtuellen Maschine viele Knoten befinden, kann dies eine effiziente Verteilung sein. Bei wenigen Knoten wird sich die Vervielfältigung der Daten nicht lohnen. Damit ist eine Abschätzung notwendig wann sich der Aufwand lohnt und wann nicht.

Bei Modellen, die keine Hindernisse mit einbeziehen ist es ohne Weiteres möglich die Bewegung auf dem Client durchzuführen.

3.4.2 Sendeaufteilung

Für die Entwicklung des Frameworks wurden die vorhandenen Modelle zusammengefasst und auf wichtige Kriterien unterteilt. Die Verbindungsmodelle können in die beiden schon erwähnten Gruppen empirische Modelle und Ray Tracing aufgeteilt werden. Die empirischen Modelle berechnen die Empfangsstärke mit Hilfe einer feststehenden Formel. Die für die Formel notwendigen Variablen ergeben sich aus den Eigenheiten der Sender und Empfänger und den bekannten festgelegten Simulationsvariablen.
Eine Berechnung der Daten für das Ray Tracing dauert während der Laufzeit zu lange. Die Daten, die für dieses Modell notwendig sind, müssen schon vor der Benutzung vorhanden sein. Die Daten können durch ein Tool erfasst werden, welches zu vorgesehenen Sendepositionen die Werte zu allen Empfängerpositionen innerhalb der vorgegebenen Reichweite bereitstellt. Es können aber auch reale Tests durchgeführt und als Daten bereitgestellt werden. Das Framework verwertet diese Daten, um die Überprüfung durchzuführen, ob eine Verbindung existiert und was diese für Eigenschaften (Bandbreite usw.) hat.

Da die Ray Tracing Daten zu groß für eine Verteilung zu den Clients sind, kann diese Berechnung nur auf dem Server durchgeführt werden. Wird ein empirisches Modell gewählt, kann die Berechnung auch auf den Clients stattfinden.

3.4.3 Kommunikation innerhalb des Emulationssystems

Wie in den beiden Abschnitten zuvor diskutiert wurde, ist die Möglichkeit wann die Arbeit auf die Clients verteilt werden kann abhängig von den Sende- und Bewegungsmodellen. Für die weitere Aufteilung ist es wichtig wie das NET mit dem Framework interagieren soll. NET muss die Knoteninformationen an das Framework übergeben und das Framework muss an NET die Verbindungsinformationen übergeben - siehe Abbildung 24.

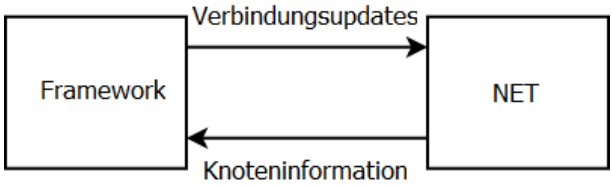


Abbildung 24: Datenaustausch von Framework und NET

Die Informationen darüber welche Knoten sich in der Emulation befinden, liegen auf den virtuellen Maschinen vor. Da jeder Client auf einer virtuellen Maschine läuft, muss dieser die Informationen über die vorhandenen Knoten auslesen und an den Server weiterleiten. Der Server verteilt diese Information weiter an die anderen Clients.

Wurde die Berechnung der Verbindung und Bewegung für die Knoten durchgeführt, muss für jeden Knoten diese Information zurück an die virtuelle Maschine geschickt werden an der sich der Knoten befindet. Wurde die Berechnung an dem Server durchgeführt, kann dieser aufgrund seines globalen Wissens die Updates direkt an die entsprechenden Clients verteilen. Sofern die Clients selbst die Berechnung vorgenommen haben, erfordert die Verteilung der Updates etwas mehr Aufwand.

Hierfür ergeben sich zwei Möglichkeiten:
Der Client selbst ist für die Updatenachrichten zuständig. Somit muss jeder Client die Nachrichten an jeden anderen Client und an den Server verschicken. Dadurch benötigt allerdings auch jeder Client einen Kanal zu jedem anderen Client. Dadurch, dass jeder physische Knoten und somit auch jeder virtuelle Knoten durch einen Switch verbunden ist, ist eine Verbindung ohne Erweiterung möglich.
Die andere Möglichkeit besteht darin den Server die Verteilung durchführen zu lassen. Hierfür werden alle Updatenachrichten an den Server geschickt, welcher daraufhin die Nachrichten an die übrigen Clients weiterversendet. Vorteil dabei ist, dass der Server immer eine globale Sicht auf das gesamte System hat und von dort alle nötigen Daten abgefragt werden können.

Erhält ein Client ein Verbindungsupdate, muss dieser überprüfen ob der Knoten auf der virtuellen Maschine des Clients liegt. Ist dies der Fall, muss der Client die Verbindungsinformationen an das NET weitergeben und die Verbindung hinzufügen, editieren oder entfernen.
Wird die Bewegungs- und Verbindungsberechnung auf den Clients durchgeführt, verteilt sich der Rechenaufwand auf die einzelnen Clients. Durch die parallele Berechnung der Updates kann diese relativ schnell erfolgen, jedoch werden zur Verteilung der Ergebnisse viele Update-Nachrichten benötigt. Bei zentralisierter Berechnung auf dem Server wird die Netzwerklast reduziert. Aufgrund der sequentiellen Abarbeitung der Bewegungs- und Verbindungsberechnungen benötigt diese jedoch mehr Zeit.

3.5 Zusammenfassung

Da das Framework möglichst viele Modelle unterstützen soll wird der Server auf dem NETfe platziert. Nur hier liegen die Daten vor, die für das Ray Tracing Modell benötigt werden.
Bei der Wahl eines Gebietsaufteilers wurde ebenfalls darauf geachtet, dass es möglichst viele Modelle unterstützen soll. Deshalb wurde eine Baumstruktur verwendet, die sich abhängig von den Positionen der Knoten innerhalb des Gebietes verändern. Abbildung 25 zeigt die Position des Servers und der Clients.

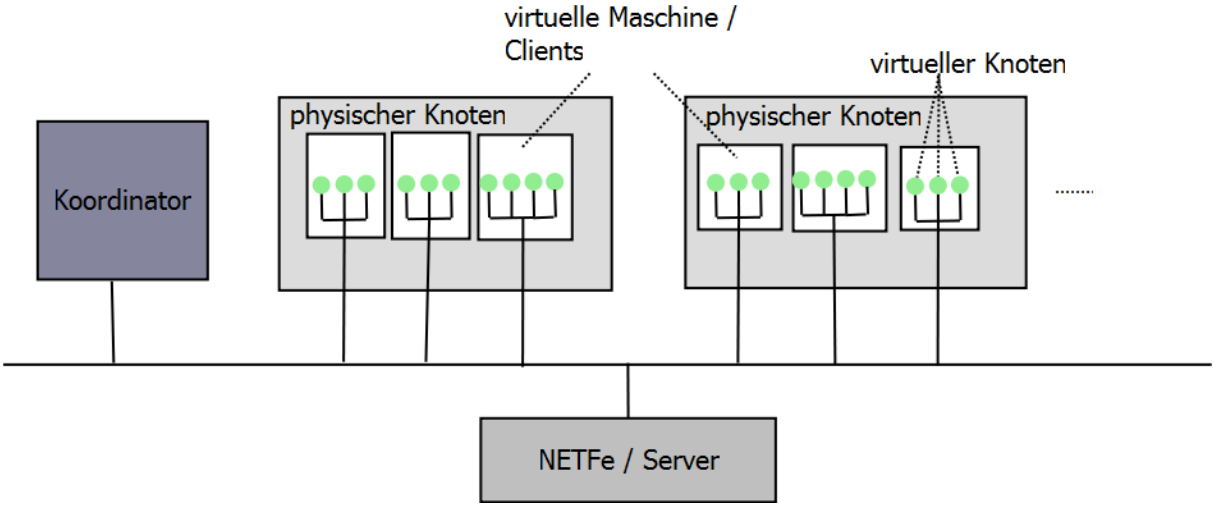


Abbildung 25: Positionierung von Server und Client im NET

4 Implementierung

Für die Umsetzung wurden verschiedene Klassen implementiert. Hier werden die wichtigsten Klassen vorgestellt.

4.1 Rahmenbedingungen

Boost

Boost [25] ist eine Bibliothek, die sehr viele verschiedene Sammlungen von Funktionen anbietet. Dabei hat Boost den Vorteil Funktionen anzubieten, die plattformunabhängig sind. Eine Implementation unter Linux und Windows ist daher identisch. Boost bietet dabei sowohl einen Dijkstra als auch einen A-Stern Algorithmus.
Von Boost werden hier die Threads und der Tokenizer genutzt. Für Erweiterungen in diesem Bereich lassen sich dadurch Funktionen leicht einbinden und ergänzen. Für die Entwicklung des hier vorgestellten Frameworks wurde die Boost-Version 1.47 eingesetzt.

4.2 Verwendung im NET

Das Framework wird zusammen mit dem RubY-Skript gestartet, welches zuvor die Knoten initialisiert. Dabei wird davon ausgegangen, dass das Framework bereits auf dem Server sowie auf dem Client vorliegt. Der Server muss auf dem NETfe laufen während jeder Client auf den virtuellen Maschinen liegt, die zu der Emulation gehören. Das Framework kann über eine Konfigurationsdatei parametrisiert werden.
Für den Server wird eine Datei benötigt in der alle virtuellen Maschinen stehen, die in der Emulation verwendet werden und ihre dazugehörige ID. Erst wenn sich alle diese Clients mit dem Server verbunden haben läuft das Framework weiter. Neben den Informationen zu den Clients werden durch die Datei eine Reihe von Initialisierungswerte übergeben (zum Beispiel die Größe des Gebietes).
Jeder Client benötigt eine Datei in der seine eigene ID und der Server steht zu dem er sich verbinden soll. Auch hier stehen Initialisierungswerte die für das Framework verwendet werden.

Jeder Client liest beim Starten aus der virtuellen Maschine die Knoten-IDs aus und die dazugehörige MAC-Adresse des Knotens. Dann gibt er die ID sowie die MAC-Adresse an den Server weiter. Dafür muss jedem Client beim Starten die Datei übergeben werden, aus der die Knoteninformationen ausgelesen werden können. Hierfür wird normalerweise die logische Datei /proc/vz/veth verwendet, welche die MAC-Adressen und IDs der einzelnen Knoten beinhaltet.

Bei der Aktualisierung der Verbindungen schreibt der Client in zwei Dateien, die von dem NETshaper verwendet werden. Zum Hinzufügen oder Ändern der Verbindung wird die MAC-Adresse zusammen mit der Bandbreite, der Verzögerung und der Verlustrate an den NETshaper (über die Datei *macadd*) übergeben.
Zum Entfernen wird nur die MAC-Adresse übergeben (durch die Datei *macdel*).

Für die Verwendung des Frameworks wird davon ausgegangen, dass das System sowie die Knoten auf jeder virtuellen Maschine laufen und initialisiert sind. Dabei wird angenommen, dass keiner der Knoten am Anfang der Emulation miteinander verbunden ist. Dies wird mit einem Ruby-Skript gewährleistet werden.

4.3 Frameworkstruktur

Damit das Framework erstellt werden kann müssen folgende Elemente in dem Framework implementiert werden.
Es sollen folgende Komponenten implementiert werden:

- eine Knotenstruktur, die es ermöglicht mehrere unterschiedliche Knoten hinzuzufügen, die sich unterschiedlich bewegen und verschieden verbinden können.
- ein Gebietsmanager, welcher Knoten innerhalb der Emulation Teilgebieten zuteilen kann.
- einen Client, der auf der virtuellen Maschine des NET liegt um vorhandene Knoten auszulesen und Verbindungsupdates zu übertragen.
- einen Server, der als zentrale Struktur dient und mit dem sich alle Clients verbinden sollen.

Das Framework besteht dabei aus den in Abbildung 26 gezeigten Komponenten.

Abhängig davon ob es sich bei der Frameworkkomponente um einen Client oder einem Server handelt und ob auf dieser Frameworkkomponente Berechnungen durchgeführt werden unterscheiden sich einzelne Komponenten. Jede Frameworkkomponente besteht dabei aus folgenden Komponenten:
Es existiert eine Klasse *node*, die die Knoten widerspiegeln, die sich innerhalb der Emulation bewegen sollen. Dafür benötigt jeder Knoten eine Klasse connectionPattern und movementPattern. Bei der ersten handelt es sich um die Klasse, welche für die Verbindung zwischen zwei Knoten zuständig ist, während die zweite Bewegung der Knoten realisiert. Diese drei Klassen (node, movementPattern, connectionPattern) spiegeln einen Knoten innerhalb der Emulation wieder.

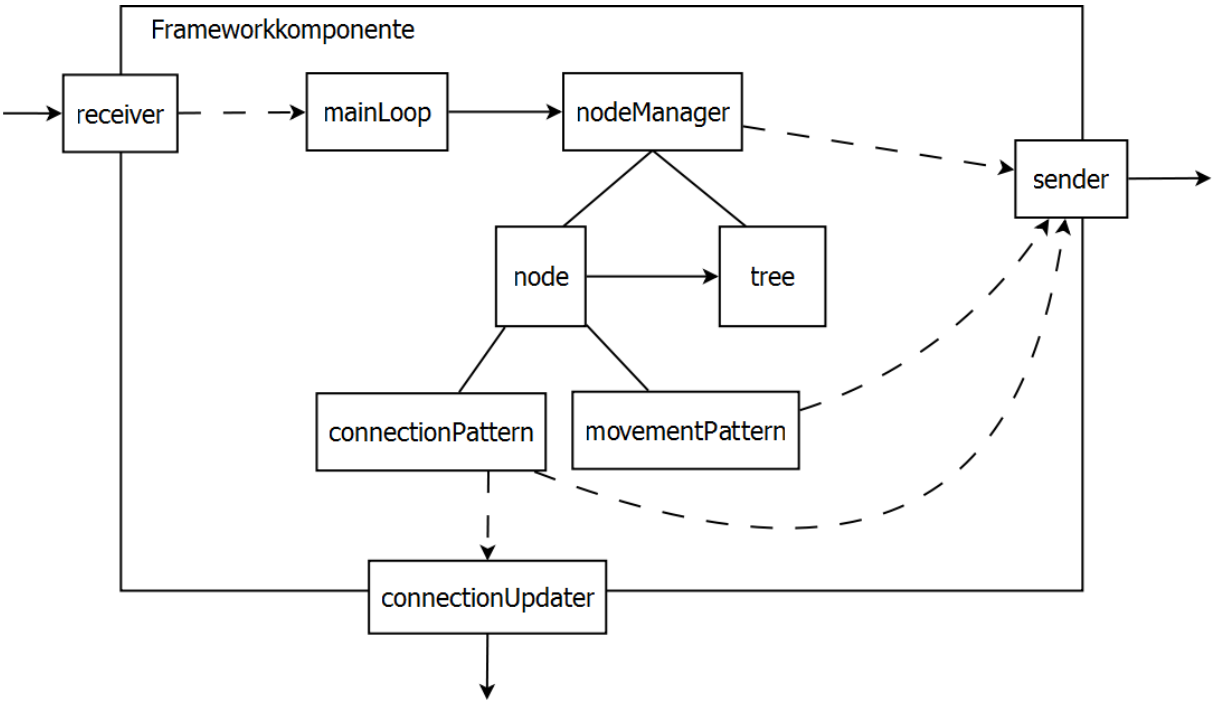


Abbildung 26: Übersicht

Die Knoten (node) werden in zwei Strukturen gehalten - den KnotenManager (nodeManager) und den Baum (tree). Die Klasse nodeManager ist für die Erstellung und Verwaltung aller Knoten zuständig. Soll ein Knoten erstellt oder eine Position, sowie Verbindungen upgedatet werden, so geschieht dies über diese Klasse.
Die Klasse tree ist für die Einteilung des Emulationsgebietes zuständig. Diese Klasse teilt das Gebiet abhängig von der Anzahl der Knoten in Teilgebiete auf. Der Baum hat ebenfalls Zugriff auf alle Knoten, verschiebt sie allerdings nur innerhalb seiner eigenen Struktur und verändert keine Werte - abgesehen von der Information in welchem Teilgebiet ein Knoten gehalten wird. Ein Baum wird in der Frameworkkomponente nur dann verwendet wenn dort Verbindungsberechnungen vorgenommen werden. Der nodeManager ist ebenfalls für die Erstellung des Baumes und für die Initialisierung dieses Baumes zuständig. Er fügt dem Baum die erstellten Knoten zu.

Für die Kommunikation zwischen den einzelnen Clients und des Servers sind die beiden Klassen *receiver* und *sender* zuständig, wofür zwei Warteschlangen existieren. Eine Empfängerwarteschlange, in der der Empfänger die Nachrichten ablegt, die dem Client beziehungsweise Server geschickt werden und einer Sendewarteschlange, aus der der Sender die Nachrichten ausliest und daraufhin an den Empfänger weitersendet.
Die Sendewarteschlange wird von dem Knotenmanager sowie den Bewegungs- und Verbindungspattern gefüllt - immer dann, wenn sich Informationen über die Knoten, die gehalten werden, verändert haben.
Die Empfängerwarteschlange wird von der mainLoop ausgelesen. Diese unterscheidet sich im Server und im Client. In dieser Schleife werden die Anweisungen an den Knotenmanager weitergegeben. Zu den Anweisungen gehören die Bewegungen aller Knoten, das Aktualisieren von einzelnen Knoten (Position oder Verbindung) und das Erstellen von diesen Knoten.
Für das Übermitteln der Updatenachrichten an das NET ist die Klasse connectionUpdater zuständig. Dafür wird eine Warteschlange bereitgestellt, die durch das Verbindungspattern gefüllt wird, sofern Änderungen an das NET weitergegeben werden sollen.

4.4 Komponenten

Um die Struktur des Frameworks noch genauer zu verstehen wird jetzt auf die einzelnen Komponenten des Frameworks eingegangen.

Da ein Knoten innerhalb der Emulation die Aufgabe hat sich zu bewegen und sich mit anderen Knoten zu verbinden, muss ihm ein Bewegungspattern und ein Verbindungspattern zugewiesen werden. Jeder Knoten kann dabei aus einem beliebigen Bewegungspattern und einem Verbindungspattern zusammengesetzt werden.
Für die Zusammensetzung der beiden Pattern wird die Klasse node verwendet, wie in Abbildung 27 dargestellt.

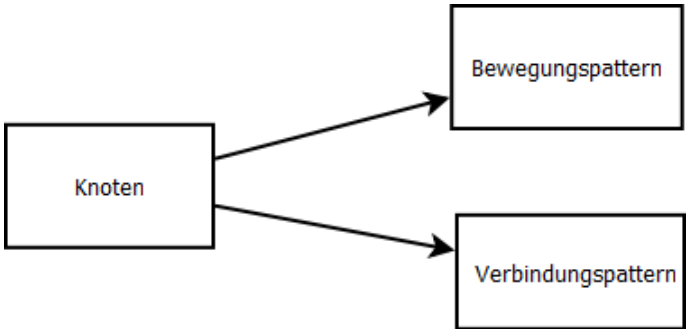


Abbildung 27: Knoten mit den beiden Modellen

4.4.1 Node

Die Klasse node steht für die einzelnen Knoten in der Emulation. Hiermit können den Knoten Eigenschaft in der Bewegung und der Verbindung zugewiesen werden. Im Folgenden werden die entscheidenden Parameter der Klasse vorgestellt.

- ID: Jeder Knoten besitzt eine ID, die innerhalb der Emulation global eindeutig ist. Diese wird von der virtuellen Maschine ausgelesen auf dem der Client läuft.
- clientId: Diese bestimmt, auf welcher virtuellen Maschine der Knoten liegt. Sie wird benötigt, um die Ziele der Updatenachrichten zu ermitteln.
- Gebietszeiger: Dieser Zeiger gibt an in welchem Gebiet sich der Knoten befindet. Er ermöglicht es dem Gebietsmanager schnell zu überprüfen, ob sich ein Knoten noch innerhalb des richtigen Gebietes befindet.
- Geschwindigkeit: Jeder Knoten besitzt eine Geschwindigkeit und beschreibt wie der Wert verwendet wird. Dieser Parameter wird von den Bewegungspattern verwendet.
- Bewegungspattern: Klasse, die beschreibt, wie sich ein Knoten bewegen soll. Hierfür wird eines der Bewegungsmodelle verwendet.
- Verbindungspattern: Klasse, die beschreibt, wann sich ein Knoten mit einem anderen Knoten verbindet. Verwendet werden die erwähnten Verbindungsmodelle.

Um den Knoten die gewünschten Eigenheiten zuzuweisen müssen die beiden Attribute, die für das Bewegungspattern und das Verbindungspattern stehen richtig belegt werden. Die Eigenschaften, die sich in der Klasse node befinden, werden von dem Bewegungs- beziehungsweise dem Verbindungspattern verwendet, um die nötigen Berechnungen durchzuführen.

Soll der Emulation ein neuer Knoten hinzugefügt werden, so muss diese Klasse abgeleitet werden. Ein neuer Knoten muss dabei die Interfaces implementieren, welche den Typ des zu verwendenden Bewegungs- und Verbindungsmodells zurückgeben. Dabei handelt es sich um folgende Klassen:

initMovePattern():
Durch diese Funktion wird das gewünschte Bewegungsmodell gewählt. Dabei muss von der abstrakten Klasse des Bewegungsmodells abgeleitet werden.

initConnectionPattern():
Diese Funktion wählt das gewünschte Verbindungsmodell. Auch hier muss das Interface des Verbindungsmodells durch Ableiten von dessen abstrakter Basisklasse implementiert werden.

initDistanz():
Gibt die maximale Sendereichweite des Knotens zurück. Diese Information wird durch das Verbindungspattern zurückgegeben.

Nachfolgend werden die für jeden Knoten notwendigen Bewegungs- und Verbindungspattern vorgestellt.

4.4.2 Bewegungspattern

Jeder Knoten besitzt ein Bewegungspattern, welches beschreibt, wie sich der Knoten fortbewegt. Die Klasse movementPattern beschreibt das zu erfüllende Interface in der Form einer abstrakten Klasse von der jedes neue Bewegungspattern abgeleitet werden muss.

Die Klasse movementPattern besitzt dabei die folgenden Attribute:

- savedNode: Beinhaltet den Knoten zu dem das Bewegungspattern gehört. Dieser wird benötigt um die Position des Knotens zu aktualisieren.
- savedQueue: Diese wird benötigt um die veränderte Position weiterzugeben. Die Queue wird von einem Sender ausgelesen.
- borderX beziehungsweise borderY: Grenzt das Gebiet ein, in dem sich die Knoten bewegen dürfen.
- usedMap: Beschreibt die Eigenschaften des Gebietes die dem Bewegungsmodell zugrunde liegt.
- direction: Beschreibt den Vektor der angibt in welche Richtung sich der Knoten bewegt. Es handelt sich dabei um einen zweidimensionalen Vektor, der auf die Länge 1 normiert ist.
- destination: Beschreibt das Ziel das angibt wohin sich der Knoten bewegen soll.

Die abstrakte Klasse hat dabei folgende virtuelle Funktionen, die implementiert werden müssen, um ein neues Bewegungsmodell zu integrieren:

void getNewDestination()
Diese Funktion hat keine Parameter die übergeben oder zurückgegeben werden. Sie hat die Aufgabe eine neue Richtung zu wählen. Dies kann entweder direkt durch eine Richtung geschehen oder indirekt indem zuerst ein Ziel und dann eine neue Richtung gewählt wird. Diese Funktion gibt jedem Knoten eine neue Richtung und wird dann aufgerufen, wenn ein Knoten ein neues Ziel benötigt. Wann genau dies der Fall ist bestimmt das Modell welches implementiert werden soll.

bool hasReachedDestination ()
Diese Funktion dient dazu, die Bedingungen zu überprüfen, ob ein Knoten an einem bestimmten Ort angekommen ist oder eine bestimmte Distanz zurückgelegt wurde.

```
1  moveNode()
2  begin
3         if (Ziel erreicht)
4                 if (Knoten muss warten)
5                         Knoten wartet
6                 else
7                         //holt nächstes Ziel
8                         getNewDestination()
9                 fi
10        else
11                führe die Bewegung des Knotens durch
12                if (Knoten ist am Rand)
13                        hasReachedBorder()
14                fi
15                //überprüft ob der Knoten das Ziel erreicht hat
16                if (hasReachedDestination())
17                        Knoten hat das Ziel erreicht
18                fi
19        fi
20 end
```

Listing 1: Pseudocode für Bewegung eines Knotens

int getWaitingTime()
Diese Funktion gibt die Zeitschritte zurück, die gewartet werden sollen. Abhängig von dem Modell kann das jedes Mal unterschiedlich oder auch immer konstant sein. Möglich ist auch, dass nicht gewartet werden soll. Falls dies der Fall ist wird der Wert 0 zurückgegeben.

void hasReachedBorder()
Diese Funktion wird aufgerufen sobald die Grenze des Gebietes erreicht wurde. Das Modell muss entscheiden wie sich der Knoten weiter bewegt.

In der abstrakten Klasse wird die Bewegung mit der Funktion *moveNode* durchgeführt (siehe den Pseudocode 1). Diese Funktion hat die Aufgabe den Knoten in die vorbestimmte Richtung zu bewegen. Für die Bewegungen wird der Richtungsvektor *direction* verwendet, der die Richtung vorgibt. Wurde die Bewegung ausgeführt, stellt die Funktion die Anfrage an *hasReachedDestination* um zu prüfen, ob der Knoten die Zielbedingung (zurückgelegter Weg, Zeit, Punkt) erfüllt hat. Ist dies der Fall wird eine neue Richtung mit Hilfe der Funktion *getNewDestination* gewählt. Wurde eine neue Richtung gewählt schreiben manche Modelle eine Wartezeit vor. Aus diesem Grund wird die Funktion *getWaitingTime* aufgerufen, die vorgibt, wie lange ein Knoten an dem Zielort bleiben soll. Sofern der Knoten nicht warten soll wird hier der Wert 0 zurückgegeben. Hat der Knoten die Gebietsgrenze erreicht wird *hasReachedBorder* aufgerufen.

Ein neues Bewegungspattern muss von der abstrakten Klasse movePattern abgeleitet werden und die oben erwähnten vier Funktionen implementieren. Soll die Bewegung nicht gerade von einem Ziel zum nächsten ablaufen, so muss die Funktion *moveNode* überladen.

Soll die Geschwindigkeit sowie die Richtung stückweise an den Wert angepasst werden, muss entweder die Funktion *moveNode* überladen werden oder bei jedem Aufruf der Funktion *hasReachedDestination* die Anpassung vorgenommen werden.

4.4.3 Verbindungspattern

Jeder Knoten besitzt neben dem Bewegungsmodell ein Verbindungspattern. Dieses Verbindungspattern übernimmt die Aufgabe zu überprüfen, ob eine Verbindung mit einem anderen Knoten existiert oder nicht. Innerhalb dieser Klasse werden alle Verbindungen die zur Zeit existieren gespeichert. Dies wird mit Hilfe eines selbstbalancierenden binären Suchbaums durchgeführt (std::map). Die abstrakte Klasse *connectionPattern* ist die Basisklasse von der alle weiteren Verbindungspattern abgeleitet werden müssen. Diese Klasse besitzt folgende Attribute:

- savedNode: Beinhaltet den Knoten der zu dem Verbindungspattern gehört. Er wird benötigt um die Position des Knotens zu kennen. Neben der Position werden von dem Knoten weitere Informationen für das Aufbauen der Verbindung geliefert.
- savedQueue: Diese wird benötigt um die veränderte Position weiterzugeben. Die Queue wird von einem Sender ausgelesen.

Bei der Implementierung von neuen Verbindungspattern muss zuerst überprüft werden welche Daten benötigt werden und die Daten - wenn nötig - hinzufügen. Hierfür ist es notwendig dem Sendemodell neben dem Knoten zu dem er gehört auch die Information darüber mitzugeben, mit was für einem Gerät gesendet wird. Abhängig davon werden die Daten unterschiedlich gewählt. Dabei können verschiedene Netzwerkkarten mit unterschiedlichen Wellenlängen simuliert werden.

Jedes neue Verbindungspattern muss dabei folgende Funktionen implementieren.

float getMaxDistanz()

Diese Funktion gibt einen Wert zurück, der beschreibt, wie weit ein Knoten maximal von einem Sender entfernt sein darf, um noch als Empfänger-Kandidat in Frage zu kommen. Für die Modelle, die als Sendereichweite einen Kreis beschreiben, ergibt sich diese Entfernung aus dem Radius des Kreises. Für Modelle bei dem dies nicht der Fall ist, wie zum Beispiel den deterministischen Modellen, muss der Punkt gefunden werden, der am weitesten von dem Sendeknoten entfernt ist. Hierfür können als Abschätzung die empirischen Modelle helfen, um eine Distanz zu bestimmen. Allerdings werden dadurch mehr Knoten überprüft als eigentlich notwendig. Jedoch ist die maximale Distanz zweier Knoten einfach zu berechnen.

float isConnected(node inputNode)*

Diese Funktion erhält als Eingabe den Knoten, mit dem eine Überprüfung stattfinden soll. Die Rückgabe beschreibt wie stark die beiden Knoten miteinander verbunden sind. Eine 0 steht für keine Verbindung. Der Knoten, der als Eingabeparameter dient, ist hierbei der Sendeknoten, während der im Vorfeld gespeicherte Knoten der Empfangsknoten ist.

Für das Überprüfen, ob zwei Knoten eine Verbindung haben, wird die Funktion *checkConnectionToNode(node*)* aufgerufen (siehe Pseudocode 2). Sie wird innerhalb des Gebietsmanagers mit dem Knoten aufgerufen, der überprüft werden soll. Dabei muss der Empfangsknoten der Aufrufende sein und der Sendeknoten als Parameter übergeben werden. Für jeden Knoten wird eingestellt wie die Empfangseigenschaften einer bestimmten MAC-Adresse sind (per NETshaper).
Dafür muss - abhängig davon ob eine Verbindung erzeugt, verändert oder entfernt werden soll - in eine unterschiedliche Datei geschrieben werden. Soll eine Verbindung hinzugefügt oder verändert werden, muss die MAC-Adresse zusammen mit der Bandbreite, der Verzögerung und der Verlustrate in die Datei "macadd" der Netzwerkkarte des Empfängers geschrieben werden.
Die Bandbreite wird in Kilobits pro Sekunde (kbps), die Verzögerung in ms und die Verlustrate in % angegeben.
Soll eine Verbindung entfernt werden muss die MAC-Adresse in die Datei „macdel" der Netzwerkkarte des virtuellen Knotens geschrieben werden.
Für die exakte Überprüfung wird die Funktion *isConnected* aufgerufen. Abhängig von diesem Rückgabewert wird eine Verbindung hinzugefügt oder entfernt. Dabei wird zuerst überprüft, ob schon eine Verbindung vorliegt, indem in der gespeicherten Struktur geprüft wird, ob ein Eintrag existiert.
Soll eine Verbindung hinzugefügt werden wird nach dem Eintrag gesucht. Existiert kein Eintrag wird ein neuer erstellt. Gespeichert wird dabei der Sendeknoten mit dem die Verbindung besteht und die Bandbreite. Existiert ein Eintrag wird überprüft ob die Bandbreite identisch ist. Unterscheidet sie sich wird diese geändert. Die Struktur wird als std::map gespeichert. Dabei wird die eindeutige ID des Sendeknotens verwendet, um den Sendeknoten und die Bandbreite zu speichern.
Soll eine Verbindung entfernt werden wird nach dem Eintrag gesucht. Existiert kein Eintrag wird nichts weiter unternommen. Gibt es jedoch einen wird er aus der Struktur entfernt.

```
checkConnectionToNode(node)
begin
        bandwidth = isConnected(node)
        if (bandwidth == 0)
                if (es existiert eine Verbindung)
                        entferne die Verbindung
                fi
        else
                if (es existiert eine Verbindung)
                        if (gespeicherte Bandbreite unterscheidet sich)
                                modifiziere die gespeicherte Bandbreite
                        fi
                else
                        füge die Verbindung hinzu
                fi
        fi
end
```

Listing 2: Pseudocode für die Verbindungsüberprüfung

Nachdem nun beschrieben wurde, wie die Knoten untereinander aufgebaut sind, wird im Folgenden darauf eingegangen, wie diese Knoten verwaltet werden.

4.4.4 Gebietsmanager

Wie im Kapitel 3.3 erwähnt, benötigt man eine Einteilung des Gebietes, um damit zu überprüfen, welcher Knoten mit welchem verbunden ist. Bei der Implementierung wurde der Quadtree gewählt. Grund dafür ist, dass sich diese Struktur dynamisch den Knotenverteilungen anpassen kann und es somit gut für unterschiedliche Bewegungsmodelle einsetzbar ist. Sollen spezielle Bewegungsmodelle mit bestimmten Eigenschaften überprüft werden (zum Beispiel in denen große Flächen nicht begehbar sind) so sollte hier ein speziellerer Gebietsmanager verwendet werden (zum Beispiel Einteilungen, die von dem Untergrund abhängen).

Der Quadtree besteht aus Baumelementen und aus Datenelementen. Für die Speicherung der Knoten an den Blättern des Baumes wurden Datenelemente verwendet. Bei den Datenelementen handelt es sich um eine einfache Liste, in welche die gespeicherten Knoten eingefügt werden. Für den Aufbau des Baumes und damit die Einteilung des Gebietes in Teilgebiete wurden die Baumelemente verwendet.

Am Anfang der Emulation benötigt der Quadtree die Größe des Gebietes welches unterteilt werden soll. Hier wird vorausgesetzt, dass es sich um ein rechteckiges Gebiet handelt. Der Quadtree beginnt mit einem Wurzelknoten an denen vier Kinder hängen.

Der Quadtree muss die folgende Funktionalität bieten.
Er muss einen neuen Knoten hinzufügen können und ihn an die richtige Stelle in dem Quadtree verschieben.
Er muss für einen vorhandenen Knoten überprüfen, ob der Knoten noch an der richtigen Stelle im Quadtree steht.
Er muss für einen Knoten eine Gebietsanfrage durchführen können. Dies bedeutet, dass der Sendeknoten innerhalb eines bestimmten Gebietes alle Knoten erhalten will, die als Empfänger des Signals in Frage kommen und damit eine Verbindung zu ihnen aufgestellt werden muss.
Sind zu viele Knoten innerhalb eines Gebietes, so soll dieses Gebiet aufgeteilt werden.
Sind innerhalb eines Gebietes zu wenig Knoten, soll es wieder zu einem größeren Gebiet zusammengefasst werden.

Struktur

Der Quadtree teilt das Gebiet rekursiv in vier gleichgroße Teilgebiete auf. Drückt man dies als Baum aus, werden daraus Baumelemente (siehe Abbildung 28), die insgesamt vier Kinder und ein Elternelement besitzen - das Wurzelelement und die Blätter ausgenommen. Das Wurzelelement besitzt kein Elternelement, während die Blattelemente keine weiteren Kinder mehr haben. Jedes Baumelement kennt seinen direkten Elternknoten und seine Kinder. Jedes dieser Baumelemente muss dazu in der Lage sein Knoten, die

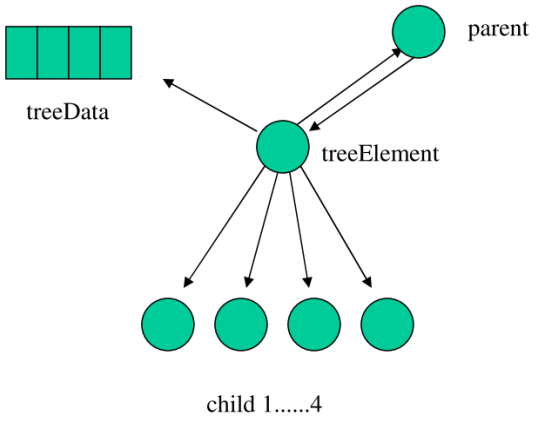


Abbildung 28: Ausschnitt eines Quadtree mit einem Baumelement

sich innerhalb seines Gebietes befinden, zu speichern. Dafür wird in jedem Gebiet eine Liste eingesetzt. Da es sich hierbei um Knoten handelt, die einzelnen Punkten (X und Y Wert) entsprechen, können Daten der Knoten in den Blättern gespeichert werden. Knoten, die auf dem Rand zwischen zwei Gebieten liegen, werden einem der beiden Gebiete zugeordnet. Dadurch speichern lediglich Blätter Knoten.

Jedes Baumelement kennt sein Gebiet für das es zuständig ist. Hierfür werden vier Werte benötigt. Dabei handelt es sich um den minimalen und maximalen Wert in X-Richtung (gibt die Breite des Gebietes an) sowie den minimalen und maximalen Wert in Y-Richtung (gibt die Höhe des Gebietes an). Diese vier Werte beschreiben das Rechteck, welches zu dem Baumelement im Baum gehört. Dadurch kann geprüft werden, ob sich ein Knoten in diesem Gebiet befindet oder nicht. Das Wurzelelement hat dadurch immer das gesamte Gebiet und die dazu gehörigen Werte gespeichert.
Wird ein Gebiet aufgeteilt, so geschieht dies indem das große Gebiet in vier kleinere Teilgebiete aufgeteilt wird. Um später zu entscheiden, in welchem Teilgebiet ein Knoten genau gespeichert wird, wird hier neben den X- und Y-Werten die Mitte des Gebietes ebenfalls gespeichert. Der Mittelpunkt wird für die Aufteilung des Gebietes in neue Teilgebiete benötigt und um zu entscheiden, zu welchem Kind ein Knoten einsortiert werden muss.

Gebietsunterteilung

Um zu entscheiden, wann ein Gebiet aufgeteilt wird, sind weitere Information nötig. Hierfür wird für den Quadtree im Vorfeld festgelegt, ab welcher Anzahl von Knoten sich ein Gebiet aufteilen soll. Dafür wird in jedem Baumelement gespeichert, wie viele Knoten sich aktuell dort aufhalten. Überschreitet diese Zahl die maximale Anzahl von Knoten, die innerhalb eines Baumelementes sein dürfen, wird das Gebiet erneut aufgeteilt.
Dies geschieht mit Hilfe des Mittelpunktes des alten Gebietes. Es werden vier neue Baumelemente erzeugt und alle vorhandenen Knoten, die in dem Gebiet liegen, werden auf die Kinder aufgeteilt.
Neben der maximalen Anzahl der Knoten innerhalb eines Gebietes darf eine bestimmte Tiefe im Baum nicht überschritten werden. Ab einer bestimmten Aufteilung hat man keinen Nutzen mehr davon die Gebiete weiter aufzuteilen. Dafür wird dem Quadtree eine maximale Tiefe vorgegeben, ab der ein Gebiet nicht weiter aufgeteilt wird.
Aus diesem Grund muss jedes Baumelement seine aktuelle Tiefe speichern. Werden neue Baumelemente erzeugt, so wird die Tiefe der Kinder berechnet, indem die Tiefe des alten Baumelements genommen und um eins erhöht wird.

Reduzieren des Baumes

Neben dem Verfeinern eines Gebietes kann der Baum ebenfalls reduziert werden. Dies soll dann geschehen, wenn innerhalb eines Gebietes eine bestimmte Anzahl von Knoten unterschritten wird. Der Quadtree muss am Anfang neben der maximalen Anzahl von Knoten, die in einem Gebiet erlaubt sind, ebenfalls eine minimale Anzahl von Knoten bekannt machen. Dabei sollten diese Werte nicht zu nahe aneinander liegen, da sonst ein Gebiet, welches gerade erst unterteilt wurde, direkt wieder zusammengefasst wird.
Damit ein Baumelement zu einem Blattelement gemacht werden kann muss die Anzahl der Knoten, die sich in dem Gebiet des Baumelements befinden, unter die Grenze der minimalen Knoten innerhalb eines Gebietes fallen. Noch dazu darf das Baumelement als Kinder nur Blätter besitzen.
Dafür ist es notwendig, dass neben den Blättern auch der Vaterknoten der Blätter die Anzahl der vorhandenen Knoten kennt. Allerdings wäre der Aktualisierungsaufwand viel zu groß die Anzahl der Knoten für jedes Baumelement zu speichern, da bei Änderungen ein kompletter Teilbaum durchlaufen werden müsste. Um das zu verhindern speichern nur die Kinder und das direkte Vaterelement die Anzahl der Knoten, die sich in diesem Gebiet befinden. Dies genügt voll und ganz, da bei einer Reduzierung die Knoten von allen Kinderelementen an den Vater weitergereicht werden und die Anzahl der Knoten nur für diesen verwendbar sind.
Damit ein Gebiet sich nur dann reduziert, wenn ein Baumknoten nur noch Blätter besitzt, muss neben der Knotenanzahl für jeden Baumknoten gespeichert werden wie viele Kinder keine Blätter sind. Dadurch können beide Bedingungen (Anzahl der Knoten und darf nur Blattelemente besitzen) überprüft werden.

Wird ein Knoten zu einem Blattelement hinzugefügt wird die Anzahl der gespeicherten Knoten im Blattelement und im Elternelement des Blattes um eins erhöht. Bei dem Entfernen eines Knotens muss daraufhin ebenfalls die Anzahl der gespeicherten Knoten im Blatt- und Elternelement reduziert werden.
Wird ein Blattelement aufgeteilt, muss dies dem Elternknoten mitgeteilt werden. Die Anzahl der Knoten werden in dem Elternknoten reduziert und die Anzahl der Nicht-Blattelemente um eins erhöht. Erst dann werden die neuen Blattelemente erstellt und die Knoten auf diesen verteilt.
Soll ein Gebiet wieder reduziert werden, so müssen alle Knoten des Kindelements wieder an das Elternelement übergeben werden. Dadurch ist das Elternelement wieder Kind und muss diese Information an seinen Elternknoten weitergeben. Dieser fügt die Anzahl der Knoten seinem Wert hinzu und verringert die Anzahl der Nicht-Blattelemente um eins.

Einfügen eines Knotens

Damit ein Knoten dem Quadtree hinzugefügt werden kann muss für ihn die richtige Position im Baum gesucht werden. Hierfür wird nur die Position des Knotens benötigt. Die Funktion *addNode(node*)* ist für das Hinzufügen eines Knotens in den Quadtree zuständig. Die Funktion übergibt den Knoten, der eingefügt werden soll. Um die Position im Baum zu finden, muss am Wurzelbaumelement begonnen werden. Zuerst wird überprüft, ob sich der Punkt innerhalb des Gebietes befindet.

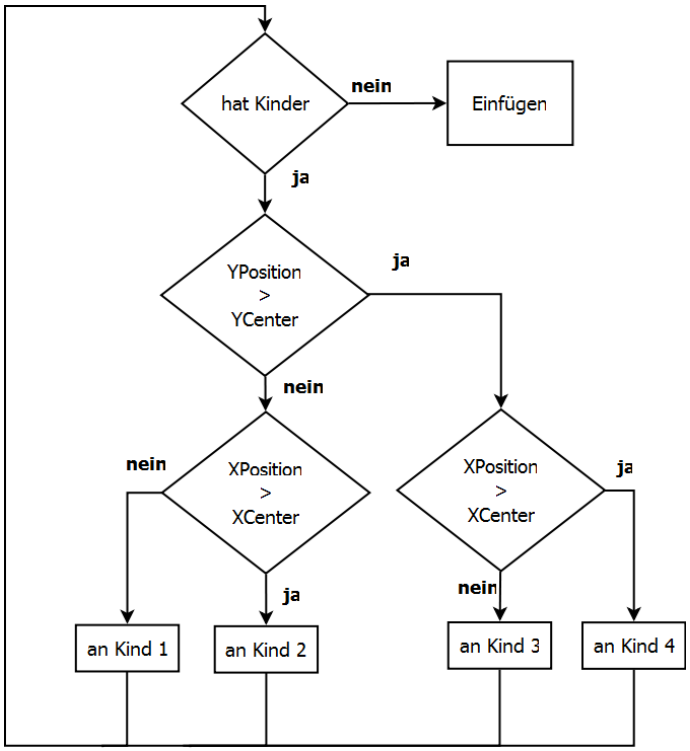


Abbildung 29: Algorithmus zum Einfügen eines neuen Knotens

Der Knoten wird dabei so lange dem Baum entlang nach unten gereicht, bis ein Blattelement gefunden wurde. Danach werden folgende Schritte immer wieder durchgeführt, bis die richtige Position gefunden wurde.
Es wird überprüft, ob das aktuelle Baumelement Kinder besitzt. Existieren auf diesem Element keine Kinder, handelt es sich dabei um ein Blatt und der Knoten kann hier eingefügt werden. Das Gebiet muss notfalls aufgeteilt werden, falls die maximale Anzahl von Knoten überschritten wurde. Der Knoten wird dann hinzugefügt und die Anzahl der Knoten aktualisiert. Der Knoten wird der treeData des Baumelementes hinzugefügt.
Hat das Baumelement noch weitere Kinder, muss überprüft werden, zu welchen Kindern der Knoten weitergereicht werden muss. Dafür wird der X- beziehungsweise Y-Wert des Knotens mit dem X- beziehungsweise Y-Wert des Mittelpunktes des Baumelementes verglichen. Abhängig von diesem Ergebnis wird das Kind gewählt und der Knoten an dieses Baumelement weitergereicht. Daraufhin beginnt die Überprüfung von neuem, bis ein Blattelement gefunden wurde.

Updaten von Knoten

Da sich die Knoten bewegen, muss überprüft werden, ob sich ein Knoten noch in dem richtigen Teilgebiet innerhalb des Baumes befindet. Dafür ist die Funktion *updateNode(node*)* zuständig, die als Eingabe den Knoten erhält, der überprüft werden soll. Eine Suche nach dem Knoten im Baum würde zu lange dauern. Aus diesem Grund hat jeder Knoten einen Zeiger, der auf das Baumelement zeigt, in das er eingeordnet wurde. Bei der Überprüfung, ob ein Knoten noch an der richtigen Stelle gespeichert ist, wird geprüft, ob der Knoten noch innerhalb des Baumelementes, zu dem der Knoten zeigt, enthalten ist. Dafür wird die Knotenposition mit den Grenzen des Baumelements verglichen. Befindet sich der Knoten nicht mehr innerhalb der Grenzen wird er aus der Liste des Baumelements entfernt und neu in den Baum eingeordnet.
Um ein direktes Verschieben in dem Baum zu ermöglichen ist es notwendig, die Nachbarn des Baumes zu kennen. Allerdings ist das bei einem Quadtree nicht trivial, da der direkte Nachbar eine unterschiedliche Tiefe und somit eine andere Aufteilung haben kann. Dies hat zur Folge, dass jedes Baumelement eine Vielzahl von Nachbarn haben kann.
Um das zu verhindern bieten sich Balancierungsansätze an. Diese sorgen dafür, dass sich die Tiefe von zwei angrenzenden Nachbarn maximal um eine Tiefe unterscheiden.
Ein anderer Ansatz wäre eine Nachbarschaft nur auf gleicher Ebene zu verwalten. Hat ein Element keinen Nachbarn bedeutet das, dass auf dieser Tiefe keine Nachbarn existieren und das Elternelement nach Nachbarn suchen muss. Allerdings hat dies zur Folge, dass für jedes Baumelement eine Nachbarschaftsverwaltung durchgeführt werden muss. Dadurch wird zusätzliche Speicherkapazität benötigt.

Gebietssuche

Damit ein Sendeknoten überprüfen kann, mit welchen Knoten er eine Verbindung hat, wird eine Gebietsanfrage an den Quadtree benötigt. Hierfür ist die Funktion *checkNodeConnection(node*)* zuständig. Als Eingabe erhält diese Funktion den Sendeknoten für den überprüft werden soll, welche Knoten diesen empfangen können. Als Gebiet, welches um den Sendeknoten gesucht wird, verwendet man das Verbindungspattern des Sendeknotens. Es wird eine Anfrage an das Verbindungspattern des Sendeknotens nach der maximalen Sendereichweite gestellt. Anhand dieses Wertes kann ein Kreis um den Sendeknoten erstellt werden, wobei der Radius des Kreises die maximale Sendereichweite ist. Da eine Überprüfung eines Kreises innerhalb des Kreises zu kompliziert ist wird für die Schnittberechnung ein Rechteck statt eines Kreises verwendet. Dadurch ist die Gebietsüberprüfung anhand der vier Eckpunkte des Rechtecks möglich.

Um die Überprüfung durchzuführen wird von dem Sendeknoten die maximale Sendereichweite ausgelesen und dann das Rechteck erstellt. Als Übergabe wird dann der minimale sowie maximale X- beziehungsweise Y-Wert übergeben. Die Funktion durchsucht daraufhin rekursiv den Baum nach allen Knoten die innerhalb dieses Gebietes liegen (siehe Abbildung 30).

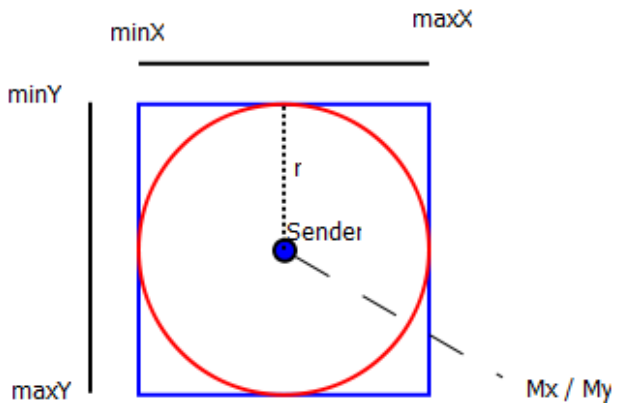


Abbildung 30: Fläche die überprüft werden soll

Dafür werden folgende Schritte unternommen.

Jedes Baumelement überprüft, ob es sich um ein Blattelement handelt.

Ist dies der Fall müssen alle Knoten, die sich hier befinden, überprüft werden. Für jeden Knoten, der sich in der treeData-Liste des Baumelementes befindet, wird die Funktion *checkConnection (node*)* des Sendeknotens aufgerufen. Mit dieser Funktion wird überprüft, ob zwei Knoten miteinander verbunden sind. Der Eingabeparameter ist der Knoten mit der die Verbindung zu dem Sendeknoten überprüft werden soll. Für die Überprüfung, ob eine Verbindung zwischen den beiden Knoten besteht, wird das Sendepattern des Sendeknotens verwendet.

Handelt es sich bei dem Baumelement um kein Blatt, so muss entschieden werden in welchem Teilgebiet der Kindelemente sich das Gebiet befindet, nach dem gesucht werden soll. Zum Überprüfen wird der Mittelpunkt des Gebietes mit dem minimalen beziehungsweise maximalen X-Wert verglichen, sowie der minimale und maximale Y-Wert verwendet. Für die Überprüfung müssen die vier Randwerte des Rechteckes um den Sendeknoten geprüft werden. Für jeden der Punkte muss weiter überprüft werden, ob er sich in dem jeweiligen Teilgebiet befindet. Im Folgenden wird der X-Wert des Mittelpunktes eines Baumelements als M_x und der Y-Wert als M_y bereichnet. Für die unterschiedlichen Gebiete gelten:

- Gebiet 1 (NW): minX muss kleiner als M_x sein und minY kleiner als M_y
- Gebiet 2 (NO): maxX muss größer als M_x sein und minY kleiner als M_y
- Gebiet 3 (SW): minX muss kleiner als M_x sein und maxY größer als M_y
- Gebiet 4 (SO): maxX muss größer als M_x sein und maxY größer als M_y

Anhand dieser Bedingungen wird überprüft in welchem der vier Teilbereiche (der Kinder des Baumelements) das zu überprüfende Gebiet liegt. Daraufhin werden die Kindelemente überprüft, ob es sich dabei um ein Blattelement handelt oder nicht. Dies wird so lange ausgeführt, bis alle Blattelemente und die dort vorhandenen Knoten überprüft worden sind, die innerhalb des zu überprüfenden Gebietes liegen.

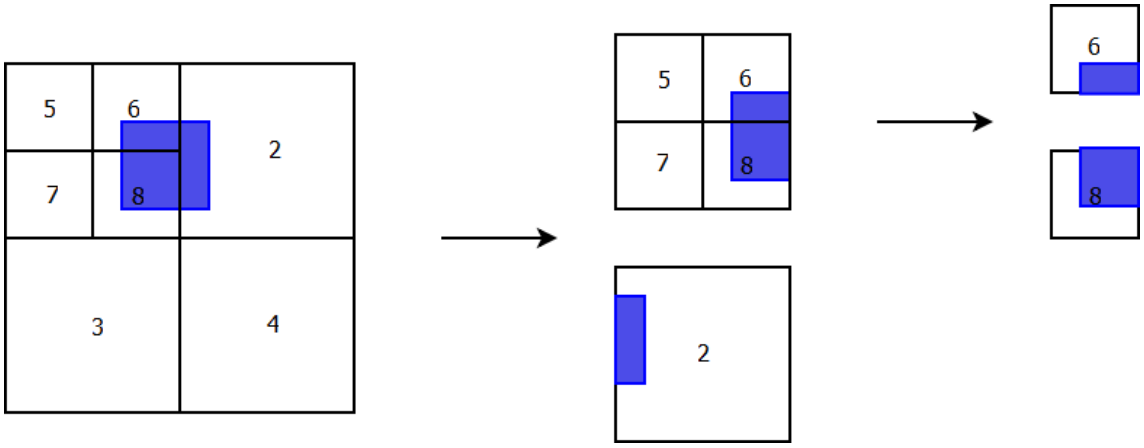


Abbildung 31: Gebietsanfrage an einen Quadtree

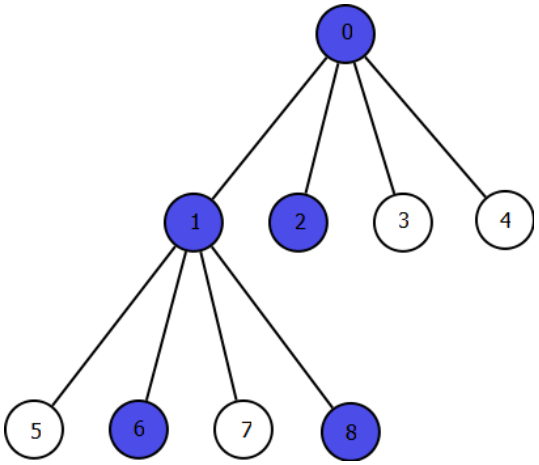


Abbildung 32: Gebietsanfrage an einen Quadtree - Baum

Abbildung 31 zeigt eine Gebietsanfrage und den dazugehörigen Quadtree in Abbildung 32. Die Anfrage beginnt beim Wurzelelement (Element 0). Da dieses Element kein Blattelement ist muss in den Kindern weitergesucht werden. Damit wird die Gebietsanfrage weiter an das Baumelement 1 und 2 gegeben. Die Überprüfung von Baumelement 2 ergibt, dass es sich hierbei um ein Blatt handelt. Damit kommen alle Knoten die sich innerhalb dieses Teilgebietes befinden als Verbindungskandidaten in Frage.
Bei der Überprüfung von Baumelement 1 wird festgestellt, dass es sich nicht um ein Blattelement handelt. Damit muss in den Kindern weitergesucht werden. Da es sich bei den beiden Baumelementen 6 und 8 um Blattelementen handelt, sind die dort vorhandenen Knoten Verbindungskandidaten und müssen genauer überprüft werden.
Die Gebietsanfrage ergibt, dass alle Knoten, die sich innerhalb der Baumelemente 2, 6 und 8 befinden, als Verbindungskandidaten genauer überprüft werden müssen.

4.4.5 NodeManager

Die Knoten werden mit Hilfe eines Knoten-Managers verwaltet. Durch diesen Manager können Knoten der Simulation hinzugefügt und in dem Gebietsmanager eingefügt werden. Für die Speicherung wird hier ein balancierter Baum (std::map) verwendet, um bei einem Durchlauf stets die gleiche Reihenfolge zu erreichen und auch Aktualisierungen an den Knoten vornehmen zu können. Ein Zugriff auf diese Struktur ist immer log(n), was auch für ein Einfügen in diese Struktur gilt. Der Knotenmanager hat dabei folgende Aufgaben.
Er muss einen Knoten hinzufügen können - dabei hat der Knoten eine einzigartige ID die sich nicht ändert. Der Manager muss die Knoten aktualisieren können und er muss über alle Knoten iterieren können.

Hinzufügen eines Knotens

Soll ein Knoten hinzugefügt werden müssen dem Knotenmanager folgende Informationen hinzugefügt werden.
Jedem Knoten muss eine global eindeutige ID zugewiesen werden. Neben dieser ID muss bekannt sein, auf welchem Client sich dieser Knoten befindet, somit muss auch eine ClientID hinzugefügt werden. Als letzter Parameter ist es notwendig zu bestimmen, um was für einen Knoten es sich dabei handeln soll. Dabei steht eine Zahl für die Art des Knotens. Die Art bezeichnet welche Verbindungspattern und Bewegungspattern diesem Knoten zugrunde liegen. Dafür muss hier jeder neue Knotentyp bekannt gemacht werden und die Type-ID in der Auswahl hinzugefügt werden. Davon abhängig wird das richtige Element erzeugt und dem Knotenmanager hinzugefügt. Als Letztes muss bestimmt werden, ob es sich hierbei um den Knotenmanager des Servers handelt oder nicht. Abhängig davon werden die Informationen über die neuen Knoten an das richtige Ziel (Server oder Client) gesendet oder nicht.

Aktualisieren eines Knotens

Da die Knoten auf mehreren Clients verteilt sind, muss ein Knotenmanager Aktualisierungen durchführen. Aktualisiert werden dabei die Position und die Verbindungen der Knoten die auf dem Client liegen. Dafür muss innerhalb des Knotenmanagers nach dem Knoten gesucht werden, der aktualisiert werden soll. Durch den balancierten Baum benötigt diese Suche log(n) Zeit. Wurde der Knoten gefunden, werden die Informationen an den Knoten weitergereicht und so aktualisiert.

Bewegen der Knoten

Damit in dem Framework ein Simulationsschritt durchgeführt werden kann, muss dieser Schritt für jeden Knoten ausgeführt werden. Hierfür müssen die gespeicherten Knoten einmal komplett durchgegangen werden. Bei dem Durchlauf der Knoten muss jeder Knoten bewegt und auf Verbindung hin überprüft werden. Hierfür wird die Funktion *moveNodes()* verwendet. Es kann mit eingegeben werden, ob die Verbindung und die Aktualisierung des Knotens im Baum bei jedem Schritt vorgenommen werden soll oder nur jeden zweiten oder dritten Schritt. Wenn sich pro Simulationsschritt kaum Verbindungen zwischen den Knoten ändern, kann es von Vorteil sein die Überprüfung nicht bei jedem Schritt durchzuführen.

4.4.6 Netzkomponenten

Damit die Clients und der Server miteinander kommunizieren können, müssen Nachrichten zwischen ihnen übermittelt werden. Jeder Client beziehungsweise Server besitzt eine Klasse *receiver* und eine Klasse *sender*. Diese beiden Klassen sind dafür zuständig die Nachrichten zu empfangen oder zu senden. Der Empfänger empfängt alle Nachrichten die an ihn gesendet werden und speichert diese in eine Empfängerwarteschlange. Der Sender hat eine Senderwarteschlange aus der Nachrichten ausgelesen, versendet und anschließend entfernt werden. Clients haben dabei jeweils nur ein mögliches Ziel an welches sie senden können.
Der Server hat mehrere Ziele zu denen er senden kann. Er holt sich die Nachricht von der Senderwarteschlange ab und versendet die Nachricht an die Ziele, die in der Nachricht vorgegeben werden.
Damit Clients Verbindungsänderungen an das NET weitergeben können besitzen diese eine Klasse *connectionUpdater*. Dafür hat der *connectionUpdater* eine Warteschlange in die das connectionPattern alle Verbindungsänderungen schreibt. Dabei wird nur dann ein Eintrag getätigt sofern der Knoten, dessen Verbindung geändert wurde, sich auf der zu dem Client gehörenden virtuellen Maschine befindet. Der *connectionUpdater* liest diese Nachricht aus und übermittelt die Information an das NET in dem der notwendige String in die Datei macadd (für das hinzufügen oder modifizieren einer Verbindung) oder macdel (sofern eine Verbindung gelöscht werden soll) geschrieben wird.

4.4.7 MainLoop

Damit der Knotenmanager gesteuert werden kann wird eine Steuerungsklasse benötigt. Dafür gibt es für den Server und den Client unterschiedliche Schleifen. Die Klasse *clientLoop* ist für die Steuerung des Clients zuständig und die Klasse *serverLoop* für den Server. Die Hauptschleifen werden über die Empfängerwarteschlange gesteuert.
Die Schleife stellt eine Anfrage an die Empfängerwarteschlange. Ist diese Warteschlange leer wartet die Schleife bis ein neues Element in die Warteschlange eingefügt wird. Sobald ein Element ausgelesen wurde wird dieses abhängig von dem Befehl verwertet. Dabei handelt es sich um einen String, der die Befehle für den Knotenmanager enthält. Die Elemente des Befehls werden mit einer Raute (#) getrennt. Dabei gibt es unterschiedliche Nachrichten:

- ADD: Damit wird ein Knoten hinzugefügt. Zusammen mit diesem Befehl muss die ID des Knotens, die Art des Knotens der erzeugt werden soll, die Position des Knotens und die ID des Clients mitgegeben werden auf dem der Knoten liegen soll ($ADD\#ID\#TYPE\#PosX\#PosY\#ClientID$).
- UPDATENODE: Damit wird die Position eines Knotens aktualisiert. Dafür muss die ID des Knotens, die neue Position des Knotens, sowie die ID des Clients mitgegeben werden auf dem der Knoten läuft ($UPDATENODE\#ID\#posX\#posY\#clientID$).
- UPDATECONNECTION: Hiermit wird die Verbindung zwischen zwei Knoten aktualisiert. Es muss dabei die ID des Sendeknotens sowie des Empfangsknotens, die Bandbreite und die ID des Clients auf dem der Knoten vorhanden ist mitgeschickt werden (UPDATECONNECTION#senderid#receiverid#bandwidth#clientID).
- RUN: Sorgt dafür, dass ein Zeitschritt in dem Client beziehungsweise Server durchgeführt wird. Dadurch werden alle Knoten auf dem Server beziehungsweise dem Client bewegt und die Verbindung für die Knoten überprüft.
- RUNALL: Dieser Befehl wird nur an den Server gesendet. Er ist dafür zuständig, dass, abhängig von dem gewählten Verteilungsmodell (in dem die Berechnung durchgeführt wird), die Bewegung durchgeführt wird. Wird die Berechnung nur auf den Clients durchgeführt wird an alle Clients ein RUN-Befehl gesendet. Wird die Arbeit komplett auf dem Server durchgeführt müssen nur die Updatenachrichten (Position und Verbindung) gesendet werden.

Die Aufgabe der Loop ist damit nach der Initialisierung nur Nachrichten aus der Empfängerwarteschlange auszulesen und abhängig von diesem String Anweisungen durchzuführen.

4.5 Netzwerkverteilung

Das Framework besteht aus insgesamt 3 Komponenten. Dabei handelt es sich um den Client, den Server und den Koordinator.

Der **Koordinator** ist dafür zuständig anzugeben, wann die nächste Bewegung stattfinden soll. Im Normalfall soll der Koordinator immer dann einen Schritt anstoßen, wenn ein interner Zähler des NET einen neuen Wert erreicht hat.
Dafür gibt es innerhalb des NET eine virtuelle Maschine die als Koordinator fungiert. Der Koordinator muss damit auf dieser virtuellen Maschine laufen. Innerhalb dieser Maschine existiert eine Datei, die zu einem bestimmten Zeitpunkt die Zeit updatet. Immer dann, wenn sich der Wert um 1 erhöht hat, muss der Koordinator eine RUNALL-Nachricht an den Server senden, der daraufhin einen Zeitschritt ausführt.

Der **Server** hat eine Verbindung zu allen Clients die auf dem System laufen. Er läuft auf dem NETfe, um so Zugriff auf Daten zu ermöglichen, die nicht auf den Clients vorhanden sind (Ray Tracing Daten). Der Server kann dabei für die Bewegungsberechnung und die Verbindungsberechnung zuständig sein oder dient als Übermittler der Nachrichten von Client zu Client. Der Server dient außerdem noch dazu, dass ein Schritt innerhalb der Emulation (eine Bewegung) durchgeführt wird. Er erhält von dem Koordinator eine Nachricht wann ein Schritt durchgeführt werden soll. Daraufhin gibt er diese Informationen an die Clients weiter oder führt die Bewegungen und Verbindungen durch und gibt die Updatenachrichten weiter.

Der **Client** läuft auf den virtuellen Maschinen. Der Client hat die Aufgabe von jeder virtuellen Maschine alle dort vorhandenen virtuellen Knoten auszulesen und an den Server weiterzugeben. Dabei muss jeder Client die ID und die dazugehörige MAC-Adresse an den Server weitergeben. Die MAC-Adresse wird für die Anpassung der Verbindung benötigt. Ein Client kann dabei die Bewegungen oder Verbindungsüberprüfungen durchführen oder bekommt vom Server Updatenachrichten. Erhält der Client Verbindungsupdates, so muss er diese Information mit Hilfe des NETshapers an das NET weitergeben.
Beim Start des Clients müssen die Informationen über die vorhandenen Knoten auf der virtuellen Maschine von dort eingelesen und an den Server übermittelt werden.

Aufgabenverteilung

Ob die Bewegung beziehungsweise die Verbindung auf den Clients durchgeführt werden kann, hängt von dem jeweiligen Modell ab. Wird ein Ray Optical Modell verwendet und werden daher Daten benötigt, die im NETfe vorliegen, so ist dies nicht ohne Weiteres möglich, da die Datenmenge im Allgemeinen zu groß ist, um sie jedem Client verfügbar zu machen. Wird die Berechnung allerdings durch eine Formel berechnet, so kann dies auf den Clients durchgeführt werden. Die Bewegungen auf den Clients durchzuführen ergibt nur dann Sinn, wenn auch die Verbindungsberechnung dort erledigt wird. Wäre das nicht der Fall, so müsste der Client die Bewegung erst durchführen, um dann die Informationen wieder zurück an den Server zu schicken. Bei vollendeter Bewegung und Verbindung müssten die Informationen wieder zurück an den Server gesendet werden. Der Server verteilt daraufhin die Daten weiter an die Clients.

4.6 Implementierte Modelle

Um die Funktion des Frameworks zu demonstrieren und zu validieren wurde ein Bewegungsmodell sowie ein Verbindungsmodell implementiert. Anhand dieses Beispiels wird erläutert, wie Modelle in das Framework integriert werden können.

Im Folgenden wird beschrieben wie Modelle in das Framework integriert werden. Dies wird anhand eines Beispiels erläutert.

Random Waypoint Modell

Für das Random Waypoint Modell wurde von der Klasse movePattern abgeleitet und folgende Funktionen implementiert:

getNewDestination(): Diese Funktion soll das nächste Ziel berechnen. Hierfür wird ein zufälliger Punkt mit Hilfe einer gleichverteilten Zufallsfunktion gewählt. Dieser zufällig gewählte Punkt ist der Zielpunkt zu dem sich der Knoten bewegen soll. Aus der aktuellen Position des Knotens und des Zielknotes wird der Richtungsvektor berechnet. Ist die Länge des Vektors 0 befindet sich der Knoten schon am Ziel. Ist die Länge nicht 0 wird der Richtungsvektor auf die Länge 1 normiert. Daraufhin wird die noch zurückzulegende Distanz berechnet und gespeichert.

hasReachedDestination(): Diese Funktion überprüft, ob der Knoten die Zielbedingung erfüllt hat. In dem Random Waypoint Modell ist diese Bedingung, dass der Zielknoten erreicht wurde. Für die Überprüfung wird die aktuell berechnete Distanz mit der Distanz des letzten Schrittes verglichen. Ist der Unterschied nicht gleich der Geschwindigkeit des Knotens, hat er den Zielpunkt überquert und ist damit am Ziel angekommen. Er ist auch dann angekommen wenn der neu berechnete Abstand 0 beträgt. Falls dies der Fall ist, wird von der Funktion „wahr“ zurückgegeben.

getWaitingTime(): Diese Funktion gibt die Zeit zurück, die ein Knoten an dem Zielpunkt warten soll. Bei dem Random Waypoint Modell wurde von einer zufälligen Zeit ausgegangen. *hasReachedBorder()*: Da die Knoten bei dem Random Waypoint Modell keinen Zielpunkt außerhalb des Gebietes wählen können, überschreiten sie niemals die Grenze des Gebietes. Damit muss hier nichts implementiert werden.

Da jede dieser Funktionen für sich alleine einfach zu implementieren ist, ermöglicht das vorgestellte Framework eine problemlose Integration von Random-Waypoint-basierter Modelle.

Free Space Modell

Für das Free Space Modell ist es notwendig von der Klasse connectionPattern abzuleiten und die virtuellen Funktionen zu implementieren. Es handelt sich dabei um die beiden Funktionen *isConnected* und *getMaxDistanz*.

Die Funktion getMaxDistanz beschreibt, wie weit ein Sendeknoten von den erreichbaren Knoten maximal auseinander sein darf. In einem Free Space Modell kann die Distanz durch ein Umstellen der Formel berechnet werden. Dadurch, dass es sich nur bei dieser Komponente um einen dynamischen Wert handelt kann der Rest fest gewählt und eingesetzt werden.
Dabei muss die maximale Grenze bekannt sein ab dem zwei Knoten nicht mehr als verbunden gelten. Auch wenn die Ausbreitung sich nicht innerhalb eines Kreises um das Sendemodell befindet, so muss das Modell trotz allem eine maximale Distanz angeben.

Aus der Gleichung für die Empfangsstärke des Free Space Modells (siehe 2.4.2) lässt sich durch Umstellen leicht der maximale Empfangsradius r bestimmen:

$$r = \frac{\lambda}{2\pi} \cdot \sqrt{\frac{P_t G_r G_t}{P_r L}}$$

Die Variablen $P_t G_t \lambda$ können direkt von den Sendeknoten erhalten werden. Die Variable L wird durch das Gebiet übermittelt. Wenn dies nicht bekannt ist kann hierfür 1 angenommen werden, da die Dämpfung L niemals kleiner als 1 sein kann. Für den Wert G_r (Empfangsstärke der Antenne) wird meist der Wert 1 angenommen. Für P_r wird der minimale Grenzwert gewählt für den ein Knoten noch empfangen kann. Dadurch lässt sich die maximale Distanz berechnen.

Sind diese beiden Funktionen und die für die Formel verwendeten Werte vorhanden, ist das Free Space Modell fertig implementiert.

Verwendung der beiden Modelle

Damit Bewegungsmodelle sowie Verbindungsmodelle verwendet werden können, muss dafür der Knoten erzeugt werden, der diese beiden Modelle benutzt.
Um einen Knoten zu erzeugen, der als Bewegungspattern das Random Waypoint Modell und als Verbindungspattern das Free Space Modell verwendet, muss die Klasse *node* abgeleitet werden. Dies wurde in dem Demonstrator in der Klasse *person* realisiert.
Hierfür müssen zwei Funktionen überladen werden:

void initMovePattern(): Diese Funktion weist der Variable moPattern das gewünschte Bewegungsmodell zu welches die Person haben soll.

void initConnectionPattern(): Diese Funktion hat die Aufgabe das gewünschte Verbindungsmodell der Variablen zuzuweisen.

4.7 Ideen für weitere Implementierungen

Graph-Modell

Im Moment existiert noch kein Bewegungsmodell welches Hindernisse mit einbeziehen kann.
Um Graphen zu implementieren kann die Bibliothek Boost verwendet werden. Boost ist in das Projekt schon intergriert und somit müssen nur die entsprechenden Bibliotheken eingebunden werden. Damit existiert eine Struktur womit ein Graph in einem Programm dargestellt werden kann. Für die Berechnung der Routen, die ein Knoten nehmen soll, eignen sich die dort vorhandenen Algorithmen A-Stern oder Dijkstra. Somit benötigt man eine Klasse, die diese Elemente einliest und mit der Klasse interagiert. Die Aufgabe des Graphenmodells ist dann, Anfragen an den Graphen zu stellen, um Routen, die ein Knoten nehmen soll, zu erhalten.
Durch die Funktion *getNewDestination* wird daraufhin das Ziel vorgegeben, zu dem sich der Knoten bewegen soll. Bei jedem Punkt wird wieder ein neues Ziel erhalten und dann von Punkt zu Punkt zum Ziel bewegt. Damit ist dieses Modell ähnlich der Implementierung des Random Waypoint Modells - mit dem Unterschied, dass die gewählten Punkte nicht zufällig, sondern nach den Punkten innerhalb des Graphs gewählt werden.

Sende-Modell

In dem Framework sind momentan noch keine deterministischen Modelle implementiert. Diese können allerdings - genau wie andere Verbindungsmodelle - hinzugefügt werden.

Um die Ray Tracing Daten zu verwenden muss eine Klasse implementiert werden, die die Überprüfung dieser Daten übernimmt. Dabei muss die Klasse zwei Informationen zurückgeben.
Die erste Information ist welche maximale Distanz zwei zu prüfende Knoten zueinander haben dürfen. Dies wird dazu verwendet die Liste der Kandidaten einzuschränken.

Neben dieser Funktion muss die Klasse die Funktion *isConnected* ebenfalls implementieren. Diese Funktion stellt die Kernfunktion dieser Klassen dar. Hier wird überprüft, ob eine Verbindung zwischen zwei Knoten existiert und wie stark diese ist.
Für diese Aufgabe bietet sich eine weitere Funktion an, die den Zugriff auf die Daten verwaltet. Da der Zugriff auf die Daten der Festplatte die Geschwindigkeit reduziert bietet sich eine Caching Strategie an, um diesen Zugriff zu minimieren. Um dies sinnvoll durchführen zu können muss eine weitere Klasse implementiert werden, die alle Zugriffe auf die Daten vornimmt und das erhaltene Ergebnis weiterleitet und gegebenenfalls speichert, um bei weiteren Anfragen das Ergebnis ohne weitere Zugriffe zurückzugeben.

Die Sendeposition kann dabei nicht exakt wiedergegeben werden, da die bereitgestellten Daten nur auf eine bestimmte Genauigkeit berechnet wurden. Dadurch ergibt sich eine Rundung auf einen Punkt, der in den Ray Tracing Daten als Sender vorhanden ist. Das Sendemodell zusammen mit einem freien Bewegungsmodell zu kombinieren ist nicht sinnvoll, da es dadurch auch Routen durch Wände geben kann, da freie Bewegungsmodelle die Umgebung nicht berücksichtigen. Somit bleibt nur die Kombination mit einem Graph-based Bewegungsmodell.

Vorliegende Ray Tracing Daten auf dem NET

Wegen des hohen Aufwands die Raytracing Daten zu berechnen ist es nicht ohne Weiteres möglich die Berechnung dieser während der Laufzeit vorzunehmen. Hierfür müssen die Daten verfügbar gemacht werden, bevor das Framework gestartet wird.
An der Universität Stuttgart wird das Intelligent Ray Tracing [21] verwendet. Dieses wurde dazu entwickelt die Berechnungszeit, die für die Knoten verwendet wird, deutlich zu reduzieren - um einen Faktor nahe 1000. An der Universität Stuttgart wurden die Daten von der Stuttgarter Innenstadt in einem Bereich von 2,4 km x 1,9 km berechnet und bereitgestellt. Zur Berechnung dieser Daten wurde das Programm WinPROP von AWE Communications [26] verwendet. Dabei benötigt WinPop als Eingabe ein 2,5 dimensionales geografisches Modell des Gebietes. WinPROP berechnet für eine Sendeposition eine Reihe von Empfangswerten für einen vorher bestimmtes Gebiet. Es ist in WinPROP nicht möglich für einzelne Empfangspositionen die Empfangsstärke zu bestimmen, sondern von einem Gebiet, dessen Größe im Vorfeld eingegeben werden muss. Dadurch wird das Gebiet in Raster eingeteilt, die jeweils von einem Sender ausgehende Empfangswerte besitzen. Je kleiner die Raster gewählt werden, desto länger dauert die Berechnung und desto mehr Daten werden von WinPROP erzeugt. Für die Universität wurde ein 5 x 5m Grid verwendet. Dies ist die minimale Größe, die bearbeitet werden kann. Da eine Berechnung eines Senders durch WinPROP 30 Sekunden dauert ist es nicht möglich dies während der Laufzeit des Programms durchzuführen. Aus diesem Grund wurde diese Berechnung im Vorfeld durchgeführt. Diese Berechnung hat auf einem 50 Knoten Cluster 3 Tage Berechnungszeit erfordert und dabei 120 GB Daten erzeugt. Insgesamt wurden an die 32 Milliarden Positionen berechnet. Ein Update dieser Daten muss erst dann erfolgen, wenn sich etwas an der räumlichen Umgebung des aufgenommenen Gebietes geändert hat. Dies geschieht in der Regel selten, weshalb sich die Berechnung lohnt.

5 Evaluation

5.1 Versuchsumgebung

Das Framework wurde mit einem Quadtree als Gebietsaufteiler implementiert und läuft im NET.
Getestet wird auf dem NET mit einem physischen Knoten. Auf diesem physischen Knoten werden die dort vorhandenen acht virtuellen Maschinen genutzt. Auf jedem der acht virtuellen Maschinen läuft ein Client der für die Verwaltung der dort vorhandenen virtuellen Knoten zuständig ist. Der Server wird auf den NETfe platziert. Als dritte Komponente wird der Koordinator auf einem anderen physischen Knoten gesetzt, der alleine für die Geschwindigkeit des Versuchs zuständig ist.
Verwendet wird ein Server-Client-Modell bei dem der Server die gesamten Bewegungs- und Verbindungsberechnung durchführt. Ist die Berechnung durchgeführt, werden Updatenachrichten über die neue Position sowie Änderungen der Verbindungen zwischen Knoten versendet. Gesendet werden die Nachrichten nur zu der virtuellen Maschine dessen virtueller Knoten davon betroffen ist.
Nach der Berechnung der Knotenposition und der Verbindung der Knoten werden die Updatenachrichten an die Clients versendet. Die Clients senden zu Beginn des Tests alle Knoteninformationen an den Server und aktualisieren die Verbindungsinformationen der Knoten, die in der ihm zugewiesenen virtuellen Maschine liegen.
Zu Beginn jedes Tests lesen alle Clients die Knoteninformationen auf den virtuellen Maschinen aus und senden diese an den Server.
Gestartet wird der Versuch mit Hilfe eines Ruby-Skriptes, das die Knoten der Emulation auf alle virtuellen Maschinen zu gleichen Teilen aufteilt. Über das Ruby-Skript werden neben der Initialisierung von NET die Rahmenumgebungen wie Gebietsgröße und Geschwindigkeit bestimmt.
Als Bewegungsmodell wird das Random Waypoint Modell verwendet. Als Verbindungsmodell wird eine Distanzüberprüfung gewählt. Dafür wird eine Verbindung auf Abstand geprüft. Liegt dieser Abstand unter einer Grenze gelten die Knoten als verbunden.
Durch die einfachen Modelle ist eine bessere Überprüfung des Quadtrees möglich, da die Berechnung dieser Modelle kaum Auswirkung auf die Laufzeit haben, was eine zuverlässige Messung der Performanz des Frameworks erlaubt.

5.2 Validierung

Für die Validierung werden zu der Emulation ein fester Knoten, der sich nicht bewegt und zwei bewegliche Knoten hinzugefügt. Jeder Knoten hat eine Empfangsreichweite von 50 Metern. Die beweglichen Knoten bewegen sich mit einer Geschwindigkeit von zwei Metern pro Sekunde. Jeder der zwei beweglichen Knoten hat die Aufgabe sich auf den festen Knoten zuzubewegen. Die beweglichen Knoten werden mit einem unterschiedlichen Abstand zu dem feststehenden Knoten eingefügt. Das Gebiet hat eine Größe von 500 x 500 Meter.
Nach der Berechnung des Richtungsvektors bewegen sich die Knoten so lange in diese Richtung, bis sie den Rand der Simulation erreicht haben und bleiben dort stehen. Ob eine Verbindung besteht wird mit Hilfe des Ping-Befehls überprüft. Dafür wird von jedem der virtuellen Knoten aus ein Ping an den feststehenden Knoten sowie zu dem anderen beweglichen Knoten gesendet. Der feste Knoten wird in die Mitte des Gebietes 250/250 (Knoten A) gesetzt während die anderen Knoten an den Positionen 250/150 (Knoten B) und 250/400 (Knoten C) starten (siehe Bild 33).

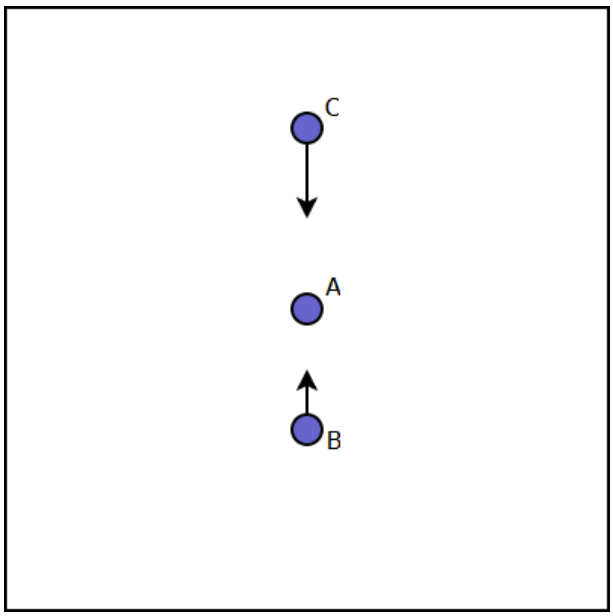


Abbildung 33: Validierung

Die Validierung läuft insgesamt so lange bis die beiden beweglichen Knoten an dem Rand angekommen sind. Bei dem Test soll jeder Knoten zu einer bestimmten Zeit eine Verbindung zu den anderen Knoten aufbauen und nachdem der Knoten zu weit entfernt ist die Verbindung wieder zu lösen.

In der Emulation ergab sich, dass der Knoten C in Sekunde 50 mit Knoten A verbunden hat und diese Verbindung in Sekunden 101 wieder beendet. Knoten B hatte eine Verbindung mit Punkt A in der Sekunde 25 und hat diese Verbindung zu dem Punkt B in der Sekunde 76 beendet. Eine Verbindung zwischen Punkt B und C bestand ab Sekunde 50 und wurde in Sekunde 76 wieder beendet.
Nach dieser Prüfung konnte sichergestellt werden, dass das Framework die Verbindungsinformationen korrekt an das NET weitergibt.

5.3 Performance

In diesem Kapitel soll die Effizienz des Frameworks untersucht werden. Als Standardeinstellung werden folgende Parameter verwendet. Das Gebiet ist 500 x 500 Meter groß. Darin befinden sich insgesamt 1000 Knoten. Jeder Knoten wird zufällig in dieses Gebiet eingefügt und bewegt sich daraufhin zufällig zu einer neue Position mit der Geschwindigkeit von 2 m/s. Jeder Knoten hat eine Sendereichweite von 50 m und überprüft die Verbindung zu anderen Knoten mit Hilfe einer Distanzberechnung. Liegt ein anderer Knoten innerhalb dieses Radius, wird eine Verbindung mit beiden aufgebaut. Der Quadtree, der zur Verwaltung eingesetzt wird, besitzt eine maximale Tiefe von 6 - was einer minimalen Gebietsgröße von 9,3 x 9,3 Metern entspricht. Für die maximale Anzahl von Knoten pro Gebiet werden 10 Knoten gewählt. Die minimale Anzahl von Knoten innerhalb eines Gebietes liegt bei 2.
Die Emulation soll für einen Zeitraum von 600 Sekunden laufen. In jeder Sekunde wird ein Schritt durchgeführt indem alle Knoten bewegt und auf Verbindungen überprüft werden. Somit werden insgesamt 600 Schritte aufgezeichnet.
Es werden dafür das NETfe als Server und acht virtuelle Maschinen als Clients verwendet. Als veränderbare Parameter werden die Anzahl der Knoten innerhalb des Gebietes, die Größe des Gebietes, die Verbindungsreichweite, die maximale Tiefe des Baumes, die Anzahl von Knoten innerhalb eines Blattes (minimal und maximal) und die Geschwindigkeit des Knotens verwendet.

5.3.1 Knotenanzahl und Gebietsgröße

Für das Gebiet werden die Anzahl von Knoten, sowie die Größe des Gebietes variiert. Die Knotendichte bleibt entweder gleich oder verändert sich.

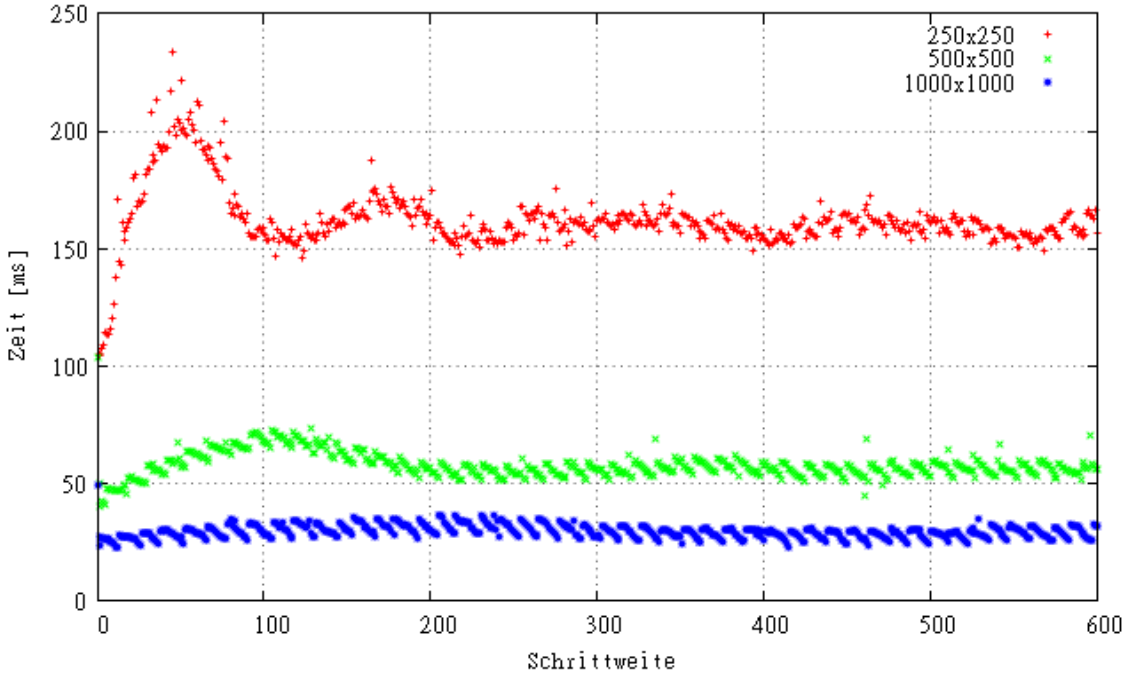


Abbildung 34: Auswirkung der Gebietsgröße auf die Performance bei gleichbleibender Knotenzahl (1000)

Im Abbildung 34 wurde das Gebiet bei gleich bleibender Knotenanzahl verringert.
Die Zunahme der Laufzeit am Anfang der Simulation liegt an der Eigenheit des Random Waypoint Modells. Durch dieses Modell kommt es zu einer Ballung der Knoten in der Mitte des Gebietes. Sind die Knoten beim Start der Simulation noch gleich verteilt, so ändert sich das im Laufe der Simulation. Da sich am Anfang der Simulation eine Großzahl von Knoten in Richtung Mitte bewegen müssen mehr Vergleiche vorgenommen werden, was zu einer erhöhten Laufzeit führt. Ist die Ballung eingetreten reduziert sich die Laufzeit wieder. Danach bewegt sie sich um einen Wert herum und weicht von diesem im Mittel nur noch minimal ab.
Zu erkennen ist eine deutliche Zunahme der Laufzeit bei einem Gebiet von 250 x 250. Durch den Anfrageradius von 25 müssen hier deutlich mehr Knoten miteinander überprüft werden als das bei den Tests in größeren Gebieten der Fall ist.

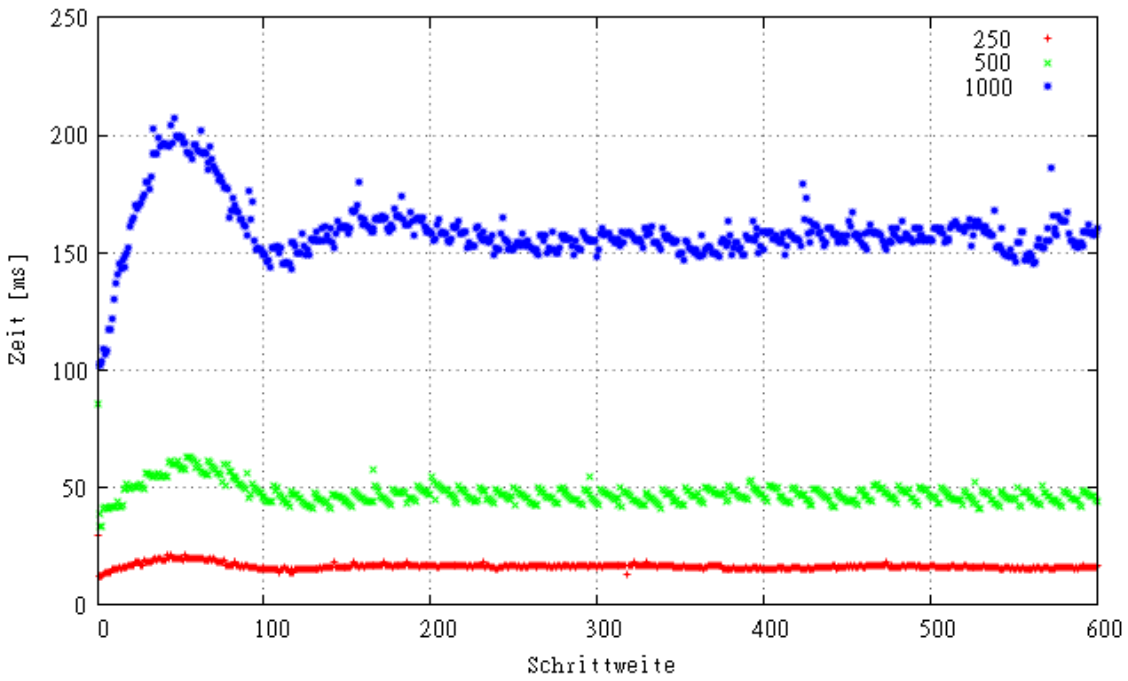


Abbildung 35: Gebietsgröße von 250 x 250 mit veränderbarer Knotenanzahl

Bei der Betrachtung eines kleineren Gebietes bei unterschiedlicher Knotenanzahl (siehe Abbildung 35) ist deutlich zu erkennen, dass bei zunehmender Dichte die Laufzeit deutlich ansteigt. Dies zeigt, dass vor allem die Verbindungsüberprüfungen deutliche Mehrkosten verursacht. Liegen die Knoten näher zusammen, so hat die Einteilung der Knoten in kleine Teilgebiete einen geringeren Effekt.
Untersucht man die Laufzeit bei einem größeren Gebiet (Abbildung 36) stellt man einen deutlich geringeren Anstieg der Laufzeit fest. Durch die bessere Verteilung der Knoten kann die Unterteilung deutlich besser greifen.

Bei dem Vergleich zwischen Abbildung 36 und 35 sieht man, dass ab einer bestimmten Dichte die Laufzeit deutlich zunimmt. Bis zu dieser Grenze steigt die Laufzeit nicht merklich.

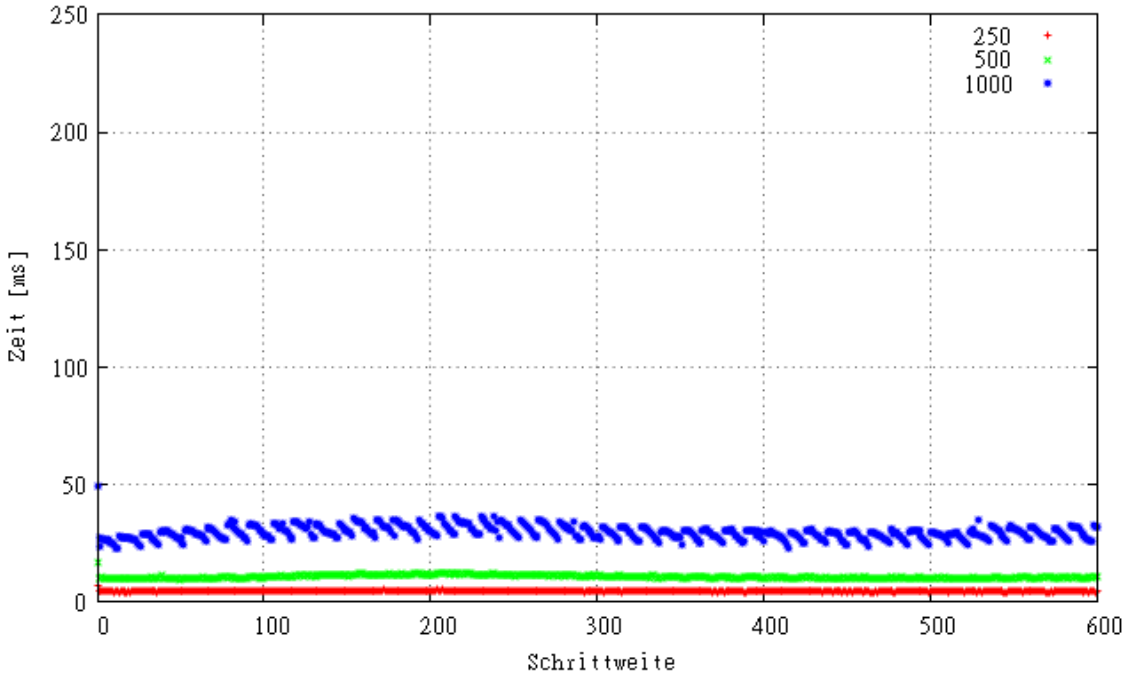


Abbildung 36: Gebietsgröße 1000 x 1000 mit unterschiedlicher Knotenanzahl

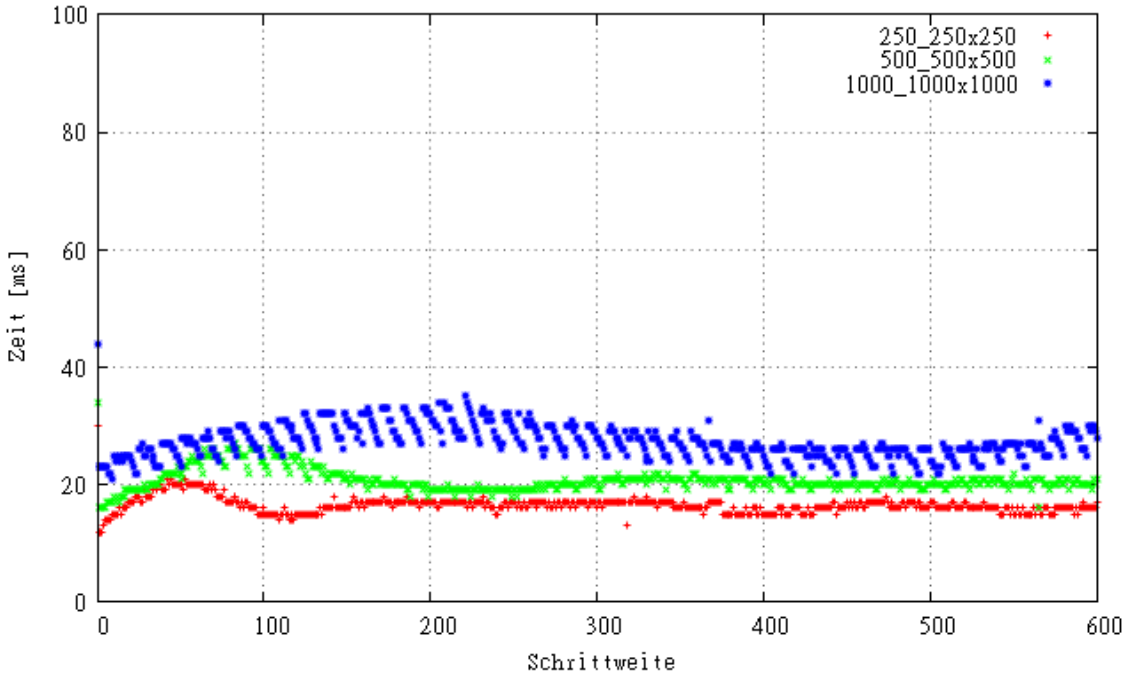


Abbildung 37: gleich bleibende Dichte - Y-Skala bis 100

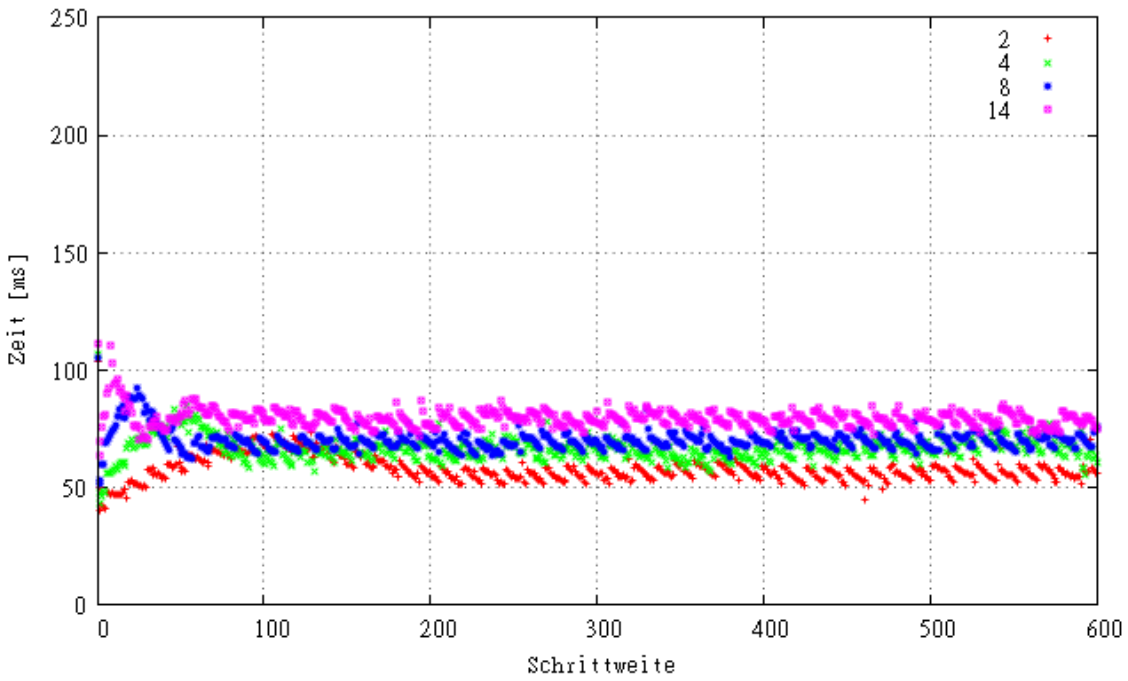


Abbildung 38: Geschwindigkeitsänderung bei einem Gebiet von 500 x 500 und 1000 Knoten

Überprüft man die Laufzeit bei gleich bleibender Dichte (siehe Abbildung 37), kann man zwischen den einzelnen Tests keinen großen Anstieg feststellen. Dies liegt daran, dass die Aufteilung der Knoten für eine deutliche Reduzierung der notwendigen Verbindungsvergleiche sorgt. Der zusätzliche Aufwand kommt von der Bewegung und Prüfung, die für jeden weiteren Knoten durchgeführt werden muss.

Zusammenfassend wird klar, dass sobald sich die Dichte erhöht mehr Vergleiche benötigt werden und dies somit eine höhere Laufzeit zur Folge hat. Bei gleichbleibender Dichte wächst die Laufzeit bei Erhöhung der Knoten geringfügig.

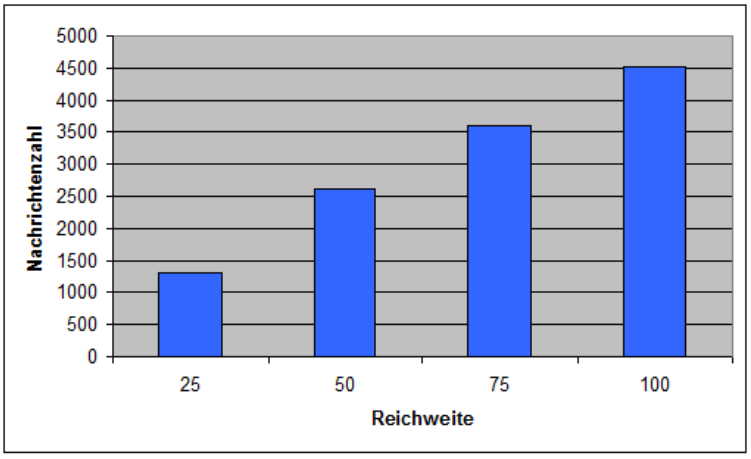


Abbildung 39: Gesendete Nachrichten zum Update der Verbindungen

5.3.2 Geschwindigkeit

Für den Einfluss der Geschwindigkeit und Reichweite zur Laufzeit wird ein Gebiet der Größe 500 x 500 Meter und 1000 Knoten verwendet. Die verwendeten Geschwindigkeiten hierfür sind 2 m/s, 4 m/s, 8 m/s und 14 m/s. Wie in Schaubild 38 zu sehen ist, ändert sich der Aufwand nur gering. Zwar muss für die erhöhte Geschwindigkeit mehr Aufwand bei der Verwaltung der Knoten in dem Quadtree durchgeführt werden, die Vergleiche zur Verbindungsbestimmung bleiben allerdings, durch die Verteilung der Knoten im Raum ähnlich. Allerdings müssen dafür mehr Änderungsnachrichten verwaltet werden.
Dabei ist zu bemerken, dass je schneller die Knoten sich bewegen desto kürzer ist die Zeit bis die Ballung der Knoten, die von dem Random Waypoint Modell hervorgerufen wird, eingetreten ist.

Vergleicht man die Anzahl der Nachrichten kann man einen linearen Anstieg abhängig von der Geschwindigkeit erkennen. Dies liegt daran, dass Verbindungsänderungen häufiger vorkommen wenn sich die Knoten schneller bewegen (siehe Abbildung 39).

5.3.3 Sendereichweite

Für die Veränderung der Sendereichweite wird eine Geschwindigkeit von 2 m/s angenommen. Es wird auf dem Gebiet von 500 x 500 Meter mit 1000 Knoten getestet. Für die Reichweite werden die Werte 25, 50, 75 und 100 Meter verwendet. Wie an der Abbildung 40 zu erkennen ist existiert ein direkter Zusammenhang zwischen der Sendereichweite der Knoten und der Laufzeit. Dies liegt daran, dass ein Knoten abhängig von der Sendereichweite unterschiedlich viele Überprüfungen tätigen muss. Je größer der Radius desto höher die Anzahl der Vergleiche. Hier ist ein quadratischer Anstieg festzustellen.

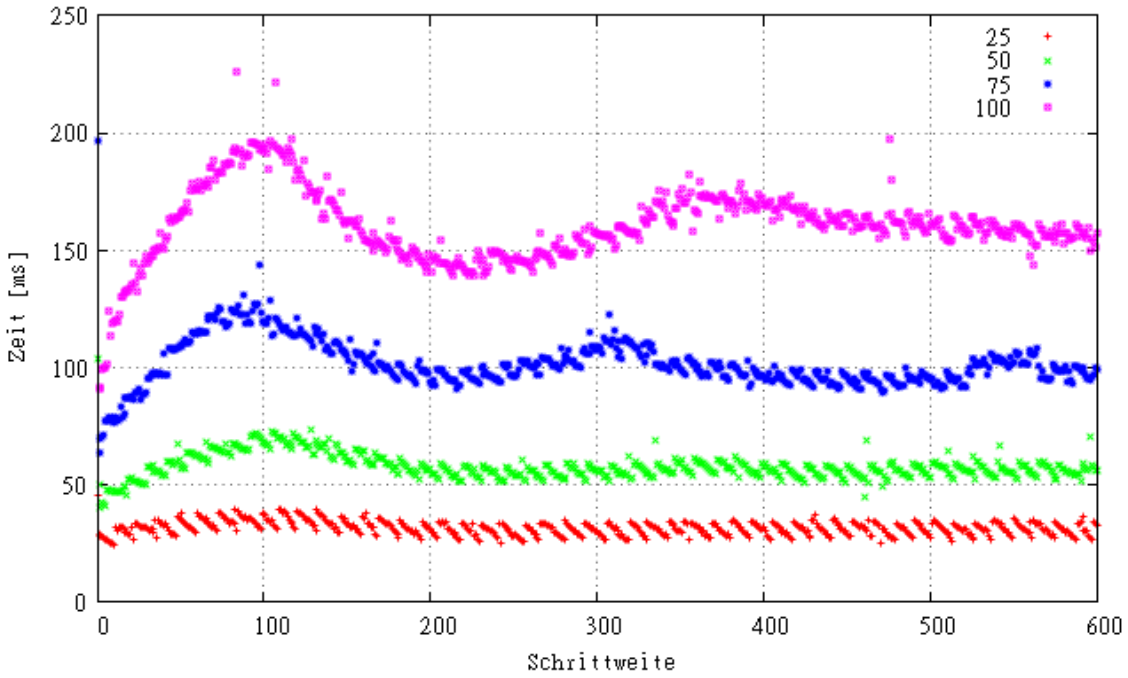


Abbildung 40: Reichweitenänderung in einem Gebiet von 500 x 500 und 1000 Knoten

Vergleicht man die Anzahl der Nachrichten erkennt man einen linearen Anstieg die abhängig von der Sendereichweite ist. Je größer der Radius desto größer ist der Bereich der als Anfrage an den Quadtree weitergereicht wurde (siehe Abbildung 41).

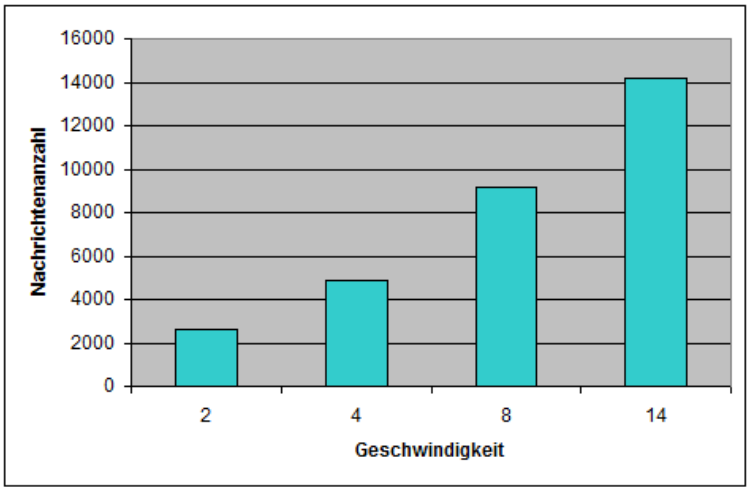


Abbildung 41: Gesendete Nachrichten zum Update der Verbindungen

5.3.4 Baumanpassungen

Als Kern des Frameworks dient der Quadtree. Hierfür können einige Parameter gewählt werden, um zu prüfen wie sich dieser verhält. Der Quadtree hängt von der maximalen Tiefe und der maximalen und minimalen Anzahl von Knoten pro Teilgebiet ab. Wieder wurde ein Gebiet von 500 x 500 Metern mit 1000 Knoten verwendet.
In der Abbildung 42 sieht man deutlich, dass die Laufzeit von der Tiefe des Baumes abhängt. Diese reduziert sich deutlich durch eine größere Tiefe des Baumes. Allerdings geschieht dies nur bis eine bestimmte Tiefe erreicht wurde. Verglichen mit einer Liste erhält man durch einen Quadtree unabhängig von der Tiefe einen großen Laufzeitgewinn. Von dort an verändert sich die Laufzeit nur noch minimal. Dies liegt daran, dass eine weitere Tiefe nicht mehr notwendig ist, da sich in einem Gebiet nicht mehr als 10 Knoten befinden und damit eine weitere Aufteilung nicht vorgenommen wird.

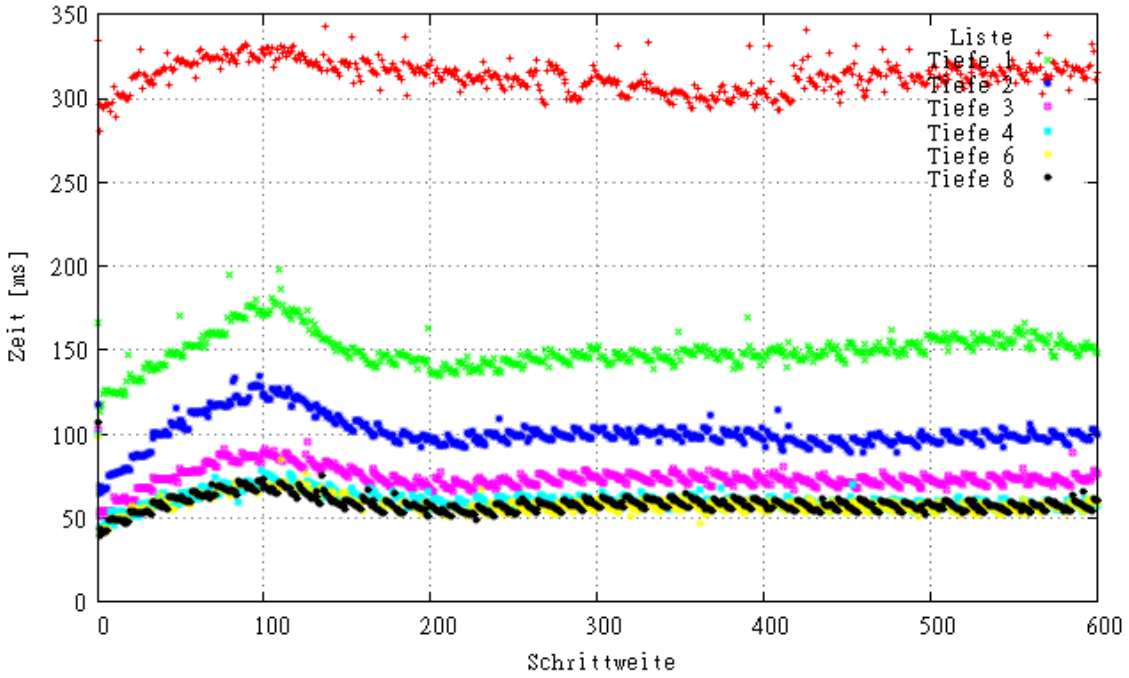


Abbildung 42: Baumtiefe bei einem Gebiet von 500 x 500 Metern und 1000 Knoten

Es können neben der Tiefe ebenfalls die minimale und maximale Anzahl von Knoten pro Teilgebiet angepasst werden. Ist die minimale Anzahl gleich 0 wird der Baum nicht wieder reduziert. Die Unterschiede zwischen den einzelnen Optionen sind gering, so dass hier nur die Mittelwerte der einzelnen Tests (Abbildung 43) gezeigt werden. Es handelt sich um die maximale Tiefe von 6.
Die schlechtesten Werte ergeben sich für eine geringe maximale Knotenanzahl (0/5) und einer zu kleinen Distanz zwischen der minimalen und maximalen Knotenanzahl (5/10). Ist der Unterschied zwischen der minimalen und maximalen Knotenanzahl zu groß wird ein Gebiet zu oft wieder reduziert nur um dann schnell wieder neu aufgebaut zu werden. Ist die maximale Knotenanzahl zu klein können für ein Gebiet nicht viele Knoten gespeichert werden, weshalb der Baum dadurch tiefer wird. Erlaubt man hier eine Reduzierung des Baumes (2/5) wird dieser negative Punkt wieder ausgeglichen, indem leere Gebiete wieder reduziert werden.
Eine Reduzierung ist daher sinnvoll, wenn der Baum sehr schnell in die Tiefe wächst.
Wählt man die maximale Anzahl gut genug ist eine Reduzierung nicht unbedingt notwendig (0/10 beziehungsweise 0/15).
Bei einer gering gewählten Anzahl der Reduzierung gibt es dort kaum Unterschiede (2/10 - 2/15). Für Szenarien in denen sich eine Ballung in dem Raum bewegt sollte allerdings auf eine Reduzierung nicht verzichtet werden, damit die Tiefe dort wieder an die Anzahl der Knoten angepasst werden kann.

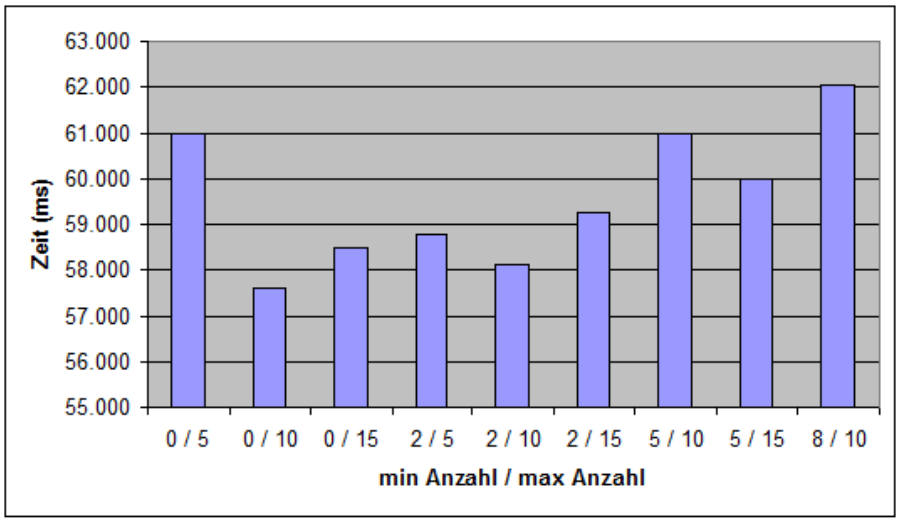


Abbildung 43: minimale und maximale Anzahl von Knoten die in einem Teilgebiet erlaubt sind

5.4 Fazit

Betrachtet man die vorgestellten Ergebnisse erkennt man, dass durch das Verwenden des Quadtrees die Laufzeit deutlich reduziert werden kann. Die Laufzeit des Frameworks konnte dadurch auf 18 Prozent reduziert werden. Die Tiefe des Baumes spielt hier eine wichtige Rolle, da sie für die Aufteilung des Gebietes und damit für einen Laufzeitgewinn sorgt. Allerdings reduziert sich die Laufzeit nur bis zu einer bestimmten Tiefe des Baumes. Damit lässt sich feststellen, dass ab einer bestimmten Tiefe das Optimum erreicht ist. Da sich Quadtrees dynamisch aufbauen, wird hier durch eine größere maximale Tiefe kein Platz verschwendet.
Betrachtet man den Baum genauer, sieht man, dass die maximale Anzahl von Knoten innerhalb eines Gebietes nicht zu nahe an der minimalen Anzahl von Knoten liegen darf. Ebenfalls sollte die maximale Anzahl von Knoten nicht zu klein gewählt werden, da sonst der Baum zu tief wird. Es sollte allerdings auch nicht zu groß gewählt werden damit der Quadtree die Verteilung durchführen kann. Der wichtigste Aspekt des Baumes bleibt allerdings die maximale Tiefe, die auf keinen Fall zu klein gewählt werden darf.

Eine starke Veränderung der Laufzeit lässt sich bei der Variation der Dichte erkennen. Befinden sich viele Knoten auf einem engen Raum hat die Aufteilung des Gebietes einen deutlich geringeren Nutzen als bei einem großen Gebiet und weit verteilten Knoten. Dies liegt daran, dass die Verbindungsüberprüfung mit dem Algorithmus weniger reduziert werden kann, da viele Knoten in der Verbindungsreichweite der Sendeknoten liegen.

Vergleicht man die Laufzeit bei gleich bleibender Dichte, steigt die Laufzeit linear abhängig von der Anzahl der Knoten die überprüft werden müssen.

Für die Geschwindigkeit erkennt man hinsichtlich der Laufzeit kaum eine Veränderung. Allerdings müssen durch die höhere Geschwindigkeit viele Knoten in dem Quadtree neu eingeordnet werden, weshalb deutlich mehr Updatenachrichten versendet werden um die Knoten in den Quadtree korrekt einzuordnen und die alten Verbindungen zu lösen sowie neue Verbindungen zu etablieren.

Die Sendereichweite der einzelnen Knoten hat einen enormen Einfluss auf die Laufzeit des Frameworks. Je größer die Fläche, die zu überprüfen ist, desto mehr Vergleiche werden insgesamt benötigt. Hier lässt sich ein quadratischer Zusammenhang erkennen, da innerhalb der zu prüfenden Fläche jeder Knoten mit jedem anderen überprüft werden muss.

Insgesamt zeigen die Tests, dass das Framework mit dem Quadtree-Ansatz gut skaliert. Schlechtere Laufzeiten kommen nur dann vor, wenn zu viele Knoten miteinander überprüft werden müssen. Dies ist vor allem bei einer Vergrößerung des Senderadius und einer zu hohen Dichte der Fall.
Sollen schnelle Bewegungen realisiert werden, müssen sehr viele Nachrichten gesendet werden. Dies führt zu einer deutlichen Mehrbelastung der Clients. Dies kann durch Verwenden zusätzlicher Clients verbessert werden, da sich dadurch die Last auf diese verteilt.
Für die Werte des Baumes haben sich eine maximale Anzahl von 10 Knoten und ein Minimalwert von 2, sowie eine maximalen Tiefe von 6 als gute Werte erwiesen.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Diese Arbeit hat sich mit dem Entwurf eines Frameworks befasst, welches neben stationären Knoten auch die Integration mobiler Knoten in NET ermöglicht. Der Fokus lag hierbei auf der Entwicklung eines Frameworks, das es ermöglicht Bewegungsmodelle sowie Verbindungsmodelle einfach zu integrieren.
Um dies zu erreichen war es notwendig die vorhanden Bewegungsmodelle und Verbindungsmodelle zu sichten, um zu erkennen welche Gemeinsamkeiten die Bewegungsmodelle beziehungsweise die Verbindungsmodelle haben um daraus Schnittstellen für das Framework zu entwerfen und zu implementieren.
Bei den Bewegungsmodellen wurde klar, dass sich die eigentliche Bewegung in allen Modellen gleicht und sich nur durch verschiedene Parameter unterscheiden (wie die Zielwahl, Wartezeiten, Grenzüberschreitung).
Die Verbindungsmodelle lassen erkennen, dass für jeden Sender eine maximale Reichweite existiert und somit nur in diesem Bereich nach Knoten gesucht werden muss.
Um den zu kontrollierenden Bereich einzuschränken wurde ein maximaler Abstand verwendet, in dem sich alle Empfängerknoten befinden. Dafür wurde eine Überabschätzung benutzt, die abhängig von dem gewählten Verbindungsmodell größer oder kleiner ausfallen kann. Hindernisse, die sich zwischen einen Sender und Empfänger befinden, reduzieren diese Reichweite lediglich.
Damit nach einem Teilgebiet gesucht werden kann, wurden Strukturen benötigt, die ein Gebiet in kleine Teilgebiete aufteilen können.
Als mögliche Strukturen wurden Baumstrukturen sowie ein Gitteransatz betrachtet. Es wurde beschrieben, wie diese Ansätze die Gebiete aufteilen und was ihre Vor- und Nachteile sind. Da es sich bei dem Framework um ein Tool handelt, das mit möglichst vielen Emulationsszenarien zurecht kommen soll, wurde der dynamische Quadtree dem statischen Gridansatz vorgezogen. Der Quadtree wurde außerdem einem KD-tree vorgezogen, da die Tiefe des Quadtrees geringer ist und bei jeder Aufteilung mehr Teilgebiete erzeugt werden. Die Tiefe ist entscheidend, da in den Blättern des Baumes die Knoten gespeichert sind. Da sich die Knoten innerhalb der Emulation bewegen ist es wahrscheinlich, dass sie die Gebiete für die Aufteilung verwenden.

Es wurde untersucht, wie das Framework mit NET interagieren muss und welche Informationen ausgetauscht werden müssen. Hierfür wurde geprüft an welcher Stelle die Komponenten des Frameworks in dem NET platziert werden müssen. Somit ergab sich ein Server-Client-Modell in dem die Clients für die Interaktion mit dem NET zuständig waren, während der Server zur Steuerung eingesetzt wurde.
Da es sehr große Datenmengen gibt (zum Beispiel bei Einsatz von Ray Tracing Daten), die lediglich auf dem NETfe vorhanden sind, musste der Server auf diesem platziert werden um Zugriff auf diese Daten zu erhalten. Die Clients wurden auf den virtuellen Maschinen platziert um von dort aus Zugriff auf die Knoten zu haben.

Nach diesen Entscheidungen wurde das Framework mit Hilfe eines Quadtrees implementiert und auf dem NET getestet. Dies wurde mit Hilfe eines Random Waypoint Modell und einer distanzbasierenden Verbindungsüberprüfung getestet.
Hier zeigte sich, dass bei gleich bleibender Dichte das Framework gut skaliert. Vergleicht man die Laufzeit mit der einer Liste, so erkennt man eine deutliche Verbesserung der Laufzeit des Frameworks. Bei höherer Dichte reduziert sich der Nutzen zwar, die Laufzeit ist allerdings noch immer deutlich besser als bei gar keiner Aufteilung des Gebietes.
Insgesamt wurde ein Framework entwickelt und implementiert in dem es möglich ist, auf einfache Art und Weise neue Bewegungsmodelle und Verbindungsmodelle hinzuzufügen. Das Framework erreicht durch die Einteilung der Gebiete gute Ergebnisse in Laufzeit und Verteilung.

6.2 Ausblick

Durch diese Diplomarbeit wurde ein Framework bereitgestellt mit dem es möglich ist, verschiedenste Bewegungsmodelle sowie Verbindungsmodelle in NET zu integrieren. Allerdings wurden davon bis jetzt lediglich das Random Waypoint Modell sowie das Free Space Modell implementiert.

In weiteren Arbeiten zu dieser Thematik sollten mehrere Modelle dem Framework hinzugefügt werden um so auf eine unterschiedliche Menge von Bewegungs- und Verbindungsmodellen zugreifen zu können. Für Bewegungsmodelle, die Hindernisse mit einbeziehen, wird eine weitere Struktur benötigt, die Kartendaten einlesen und verwenden kann.
Um strahlen-optische Modelle hinzuzufügen wird ebenfalls eine Struktur benötigt, welche auf die vorhandenen Daten im NET zugreifen kann und diese in das Framework integrieren und verwenden kann. Die Ray Tracing Daten von der Stuttgarter Innenstadt liegen auf dem NETfe vor.
Weiter werden bis jetzt nur Modelle verwendet, die als Antennentyp Rundantennen verwenden. So wird als Überprüfung stets ein Kreis genommen. Sollen andere Antennen, die in eine bestimmte Richtung senden, verwendet werden, muss dies verändert werden.

Da das Framework eine Vielzahl von Modellen unterstützen soll kann für spezielle Gegebenheiten (Knotenhäufung, unbegehbares Gebiet) eine andere Art der Gebietsaufteilung von Vorteil sein. Dafür könnten sich andere Strukturen wie der KD-Tree besser eignen als dies bei dem Quadtree der Fall ist.
Der Quadtree-Ansatz kann durch Hinzufügen einer weiteren Klasse problemlos ersetzt werden.

Literatur

[1] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16:187–260, June 1984.

[2] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Efficient and scalable network emulation using adaptive virtual time. In *ICCCN*, pages 1–6, 2009.

[3] NET. Network emulation testbed. `http://net.informatik.uni-stuttgart.de/`, 2011.

[4] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. Time jails: A hybrid approach to scalable network emulation. In *PADS*, pages 7–14, 2008.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP (2003)*, pages 164–177.

[6] OpenVZ. `http://www.openvz.org`, zuletzt aufgerufen am 02.10.2011.

[7] Daniel Herrscher and Kurt Rothermel. A Dynamic Network Scenario Emulation Tool. In *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN 2002), Miami (FL), Oct 14-16, 2002*, pages 262–267. IEEE, October 2002.

[8] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. NETplace: Efficient Runtime Minimization of Network Emulation Experiments. In *Proceeding of the International Symposium on Performance Evaluation of Computer and Telecommunication*, pages 265–272. IEEE Communications Society, 2010.

[9] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. NETbalance: Reducing the Runtime of Network Emulation using Live Migration. In *Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN 2011)*. IEEE Press, 2011.

[10] Tracy Camp, Jeff Boleng, and Vanessa Davies. A survey of mobility models for ad hoc network research. *Wireless Communications & Mmobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2:483–502, 2002.

[11] Illya Stepanov and Kurt Rothermel. Simulating mobile ad hoc networks in city scenarios. *Comput. Commun.*, 30:1466–1475, May 2007.

[12] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[13] I. Stepanov. Using geographic models in the simulation of mobile communication, 2008.

[14] Christian Bettstetter, Giovanni Resta, and Paolo Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 2:257–269, July 2003.

[15] Tracy Camp, Jeff Boleng, and Vanessa Davies. A survey of mobility models for ad hoc network research. 2:483–502, 2002.

[16] Ching-Chuan Chiang. *Wireless network multicasting*. PhD thesis, 1998.

[17] Rayko Glowa. Berechnung elektromagnetischer Felder von Mobilfunkanlagen mit WinField Telecommunication. Diplomarbeit, Fachhochschule Lausitz, 2004.

[18] Mark Hoja. Modelle und Algorithmen zur Konzipierung und Entwicklung von Projektierungstools fuer WLAN/WiMAX – Funknetze. Diplomarbeit, Technische Universitaet Dresden, 2006.

[19] H. Friis. A note on a simple transmission formula. In *Proceedings of IRE*, volume 41, pages 254–256, 1946.

[20] J. D. Gibson. *The Communications Handbook*, volume 21, chapter 84, pages 1182–1196. CRC Press, 1997.

[21] Illya Stepanov and Kurt Rothermel. On the impact of a more realistic physical layer on manet simulations results. *Ad Hoc Netw.*, 6:61–78, January 2008.

[22] K. R. Schaubach, N. J. Davis, and T. S. Rappaport. A ray tracing method for predicting path loss and delay spread in microcellular environments. In *Vehicular Technology Conference, 1992, IEEE 42nd*, pages 932–935 vol.2, August 2002.

[23] B. E. Gschwendtner, G. Wölfle, B. Burk, and F. M. Landstorfer. Ray Tracing Vs. Ray Launching In 3-D Microcell Modelling. In *1st European Personal and Mobile Communications Conference*, pages 74–79, November 1995.

[24] S. Fortune. Efficient algorithms for prediction of indoor radio propagation. In *Proceedings of the 48th IEEE Vehicular Technology Conference*, 1998.

[25] Boost.
`http://www.boost.org/`, zuletzt aufgerufen am 20.09.2011.

[26] Awe communications.
`http://www.awe-communications.de`, zuletzt aufgerufen am 02.10.2011.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Frank Schuh)