Modelling of Autonomous Agents in a Realistic Road Network

Christian Gloor [1]
Diplomarbeit am Dept. für Informatik der ETH Zürich

1. Oktober 2001

[1] chgloor@silmaril.org

Zusammenfassung

In dieser Arbeit wird eine Infrastruktur für eine Mikro-Verkehrssimulation vorgestellt, die den sich darin bewegenden Autos erlaubt, wärend der laufenden Simulation Entscheidungen aufgrund dynamischen Verhältnissen zu treffen.

Die Ideen einer bestehenden Arbeit über einen lernenden Agenten, der sich im unbekannten Strassennetz bewegen muss, wurden so implementiert, dass mehrere dieser Agenten parallel agieren können. Dabei versuchen diese Agenten nicht einfach den kürzesten Weg zum Ziel zu finden, sondern es erfolgt ein mehrstufiges Entscheidungsverfahren, in dem ein kurzer Weg gegen Experimentierfreude und Strassenauslastung durch andere Agenten abgewogen wird.

Dazu wurde ein Kommunikationssystem enwickelt, mit dem auf einfache Weise mehrere dieser Agenten, die möglicherweise auf verschiedenen Rechnern laufen, an eine beliebige Verkehrssimulation gekoppelt werden können.

Eine solche Verkehrssimulation wurde zu Test- und Demonstrationszwecken implementiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Ziel einer Verkehrssimulation ist, anhand eines Modelles die Verkehrsverhältnisse zu simulieren. Dadurch können durch ein Ändern der Parameter Schlüsse gezogen werden, die bei der Planung von zukünftigen Strassen hilfreich sein können.

Simuliert werden von einzelnen Strassen bis zu ganzen Ländern, je nach Fragestellung. Das in dieser Arbeit vorgestellte Simulationssystem eignet sich für Simulationen ab der Grösse einer Stadt.

1.2 Wissenschaftlicher Stand

In den bekannten Systemen wie z.B. TRANSIMS (siehe auch [KNR]) existieren zwei Komponenten: einerseits die eigentliche Simulation, in der sich die Autos bewegen. Diese Autos besitzen allerdings keine Intelligenz in dem Sinne, dass sie selbständig Entscheidungen treffen könnten. Daher wird andrerseits ein zusätzliches Programm verwendet, dass die Routen berechnet, die die Autos in der späteren Simulation abzufahren haben.

So folgt eine Simulation auf das Routenberechnen, soll ein Feedback stattfinden, müssen diese mehrmals hintereinander gestartet werden.

1.3 Bedürfnisse

Falls die Autos eine gewisse Intelligenz besässen, könnte auf kurzfristig ändernde Verhältnisse in der Simulation reagiert werden.

Autonom

Es gibt Situationen, in denen ein menschlicher Autofahrer im richtigen Leben reagiert, auch wenn er ursprünglich etwas anderes vorhatte. So wird kaum jemand stur auf einer vorgeplanten Route durch die Stadt beharren, wenn starker Verkehr herrscht und aber die Nebenstrassen sichtbar freier sind. In einer solchen Situation wird der Autofahrer spontan seine geplante Route ändern und einen vielleicht längeren, aber trotzdem vielversprechenderen Weg einschlagen.

Zusätzlich wird er sich die Strassen merken, die besonders verstopft waren und diese in Zukunft eher meiden. Vielleicht merkt er sich auch die zugehörige Zeit und kann sich so ein recht komplexes Modell der Verkehrssituation der Stadt aufbauen. Plant er eine neue Reise, wird er auf dieses gesammelte Wissen zurückgreiffen.

Verteilt

Es scheint klar, dass ein solches Modell nicht auf einem Rechner implementiert werden kann, da es enorm Ressourcen konsumieren wird. Alleine die Verkehrsmodelle der einzelnen Autofahrer auf einem Rechner zu speichern scheint unmöglich. Kann es doch sein, dass mehrere tausend Strassen mit den dazugehörigen Zuständen gespeichert werden müssen, vielleicht noch mehrfach für die verschiedenen Tageszeiten.

Ausserdem benötigt die eigentliche Verkehrssimulation bereits viel Zeit, so dass nicht noch sämtliche Routen ebenfalls berechnet werden können.

Abhilfe würde ein System schaffen, in dem die "Intelligenz" der Autofahrer auf anderen Rechnern gehalten würde. So könnten jeweils ein paar wenige Autos von einem Rechner gesteuert werden, die übrigen Autos würden von weiteren Rechnern übernommen.

Kommunikation

Ein wesentlicher Teil dieses Systems wäre die Kommunikation zwischen den Rechnern. So muss darauf geachtet werden, dass nicht zuviel Zeit verstreicht, da die Autos in der Simulation sonst verwirrt umherfahren.

Es ist zu definieren, wieweit die Autos selbständig handeln können, und ab wann der entfernte Fahrer konsultiert werden muss. Es macht wenig Sinn, bei jedem Hindernis (z.B. einem vorausfahrenden anderen Auto) über das Netzwerk zu fragen, ob gebremst werden soll.

Die Kommunikation ist auf einem Minimum zu halten.

Kapitel 2

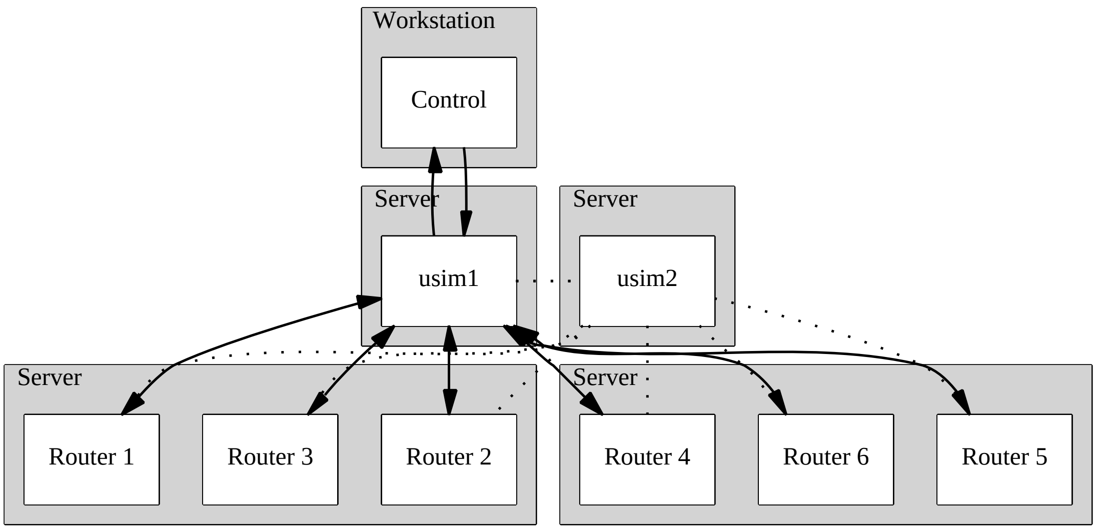
Grundlagen

2.1 Aufbau

Das gesamte Simulationssystem ist in drei Teile unterteilt (siehe Abbildung 2.1):

- μsim
- Router
- Steuerprogramm

Diese Teile sind in drei unabhängigen Programmem realisiert. Jedes kann auf einem anderen Host-Computer ablaufen und selbst nochmals parallelisiert sein. Verbunden werden die drei Teile über Meldungen, die untereinander über das Netzwerk ausgetauscht werden können.



μsim

Die μsim (Mikrosimulation) stellt die eigentliche Simulationsumgebung dar. Sie enthält die Welt der Strassen, Kreuzungen und natürlich der Autos, welche sich darin bewegen können.

Die μsim ist derjenige Teil, der den Systemtakt vorgibt: in jeder Zeiteinheit werden die Autos soweit auf den Strasse bewegt, wie gemäss der aktuellen Geschwindigkeit und Verkehrssituation möglich ist.

In dieser Arbeit wurde eine einfache μsim entwickelt, die dem Testen der anderen Teile dient. Da diese nur ein Hilfsmittel ist, wurde sie entsprechend einfach gehalten. Eine tiefere Beschreibung ist in Kapitel 7 zu finden.

Die μsim kann durchaus selbst auf mehreren Rechner verteilt ablaufen. Jeder Teil der μsim muss mit dem ihren Autos zugeordneten Routern selbst kommunizieren (in Abb. 2.1 punktiert gezeichnet)

Router

Im Router ist die "Intelligenz" der einzelnen Autos implementiert. Der Router ist ein eigenes Programm, das von der Mikrosimulation konsultiert werden kann, wenn diese Entscheiden muss, wolang ein Auto fahren soll.

Der hier verwendete Router basiert auf der Dissertation von Helena Unger an der Universität Rostock ([UNG]).

Die Ideen von Helena Unger wurden teilweise übernommen und so integriert, dass mehrer Autos gleichzeitig unterstützt werden. Ausserdem ist das neue Programm nun in der Lage, Weganfragen von den Autos in der Simulation zu beantworten.

Mehr über den Router ist in Kapitel 5 zu erfahren.

Steuerprogramm

Um den Ablauf der Simulation zu steuern, ist ein separates Steuerprogram notwendig. Dieses bildet das eingentliche Interface zum Benutzer. So können z.B. Autos in die Simulation eingefügt, oder Statistiken abgefragt werden.

Das Steuerprogramm kommuniziert ebenfalls über das Netzwerk mit der Mikrosimulation.

Es existiert kein universelles Steuerprogramm, das allen Situationen gerecht werden würde. In Kapitel 6 ist beschrieben, wie mit wenig Aufwand ein speziell für ein bestimmtes Problem zugeschnittenes Steuerprogramm geschrieben werden kann.

2.2 Meldungen

Die einzelnen Teile des Simulationssystemes kommunizieren über das Netzwerk durch das Versenden von kleinen Meldungen.

Es ist dabei so, dass das Steuerprogramm mit der μsim kommuniziert, und diese wiederum mit dem Router (Abbildung 2.1).

Der typische Ablauf sieht folgendermassen aus:

1	Steuerprogrmm $\rightarrow$ μsim	Neues Auto erstellen
2	Steuerprogrmm $\rightarrow$ μsim	Neue Destination zuteilen
3	μsim $\rightarrow$ Router	Routenanfrage
4	Router	Routenberechnung
5	Router $\rightarrow$ μsim	Route wird geliefert
6	Loop 3	bis Ziel erreicht
7	μsim $\rightarrow$ Steuerprogramm	Ziel erreicht
8	Loop 2	Neue Reise antreten

Alle Meldungen und ihre Bedeutung ist in Kapitel 4.3 zu finden.

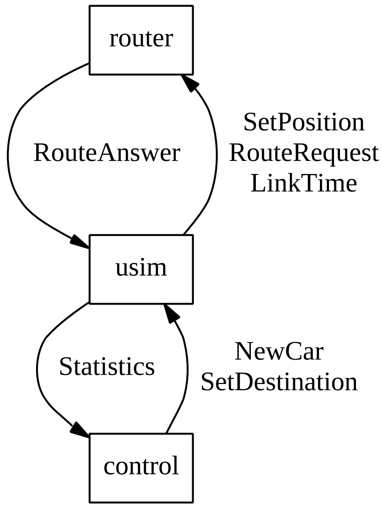


Abbildung 2.1: Hauptkomponenten und vereinfachter Message-Fluss

2.3 Kommunikation

Die meisten Applikationen, die Daten über ein Netzwerk schicken, bauen zuerst eine Verbindung auf, über die die Kommunikation später abläuft. Im Falle eines Fehlers im Netzwerk oder wenn ein Puffer in einem beteiligten Computer voll ist, wird die Kommunikation aufgestaut, so dass nichts verloren gehen kann.

Allerdings ist dies nicht das erwünschte Verhalten in einer Kommunikation zwischen der μsim und dem Router: gerät die Kommunikation ins Stocken, so bewegen sich die Autos in der μsim trotzdem weiter und es macht wenig Sinn, wenn inzwischen veraltete Daten noch eintreffen.

Das Verbindungslose Protokoll UDP kann Pakete an beliebige Empfänger verschicken, ohne dass vorher eine Verbindung aufgebaut wurde. Geht ein UDP-Paket verloren, so wird dies allerdings nicht bemerkt.

Die gesamte Kommunikation zwischen den drei Teilen des Simulationssystemes ist in einer Library (Bibliothek) zusammengefasst. Dies soll einerseits die komplexen Funktionsaufrufe vor dem Anwender fernhalten, andrerseits aber auch eine gewisse Unabhängigkeit in der tatsächlichen Implementation garantieren. Verbesserungen können so in der Library vorgenommen werden, ohne dass die Anwendungsprogramme abgeändert werden müssen. Diese Bibiothek wird in Kapitel 4 eingehend betrachtet.

Kapitel 3

Messungen

3.1 Zeitmessungen

Kommunikation über das Netzwerk

Theoretische Grenzen

Jedes Datenpaket, das über das Ethernet übertragen wird, muss mehrfach eingepackt werden. Jede Netzwerkschickt fügt ihren Header an die Nutzdaten der vorherigen Schicht dazu.

Zuerst kommt ein UDP-Header, der neben den Portangaben eine Prüfsumme von sich selbst beinhaltet:

0.-15. bit	16.-31. bit
Absender-Port	Empfänger Port
Länge	UDP-Prüfsumme

Dies sind 8 bytes, die zur Meldungslänge dazuaddiert werden müssen.

Um ein UPD-Paket kommt der 24 bytes lange IPv4-Header:

<table>
<tr><td colspan="3">0.-15. bit</td><td colspan="2">16.-31. bit</td></tr>
<tr><td>Version</td><td>IHL</td><td>TOS</td><td colspan="2">Gesamtlänge</td></tr>
<tr><td colspan="3">Kennung</td><td>Flags</td><td>Fragment-Offset</td></tr>
<tr><td colspan="2">TTL</td><td>Protokoll</td><td colspan="2">Kopf-Prüfsumme</td></tr>
<tr><td colspan="5">Absender-IP-Adresse</td></tr>
<tr><td colspan="5">Empfänger-IP-Adresse</td></tr>
<tr><td colspan="4">Optionen</td><td>Füllzeichen</td></tr>
</table>

Schlussendlich wird noch ein Ethernet-Header dazugefügt, welcher aus 18 bytes Header und 8 bytes Präambel besteht.

8 bytes	6 bytes	6 bytes	2 bytes		4 bytes
Präambel	Empfängeradresse	Senderadresse	Typ	Daten	Prüfsumme

Zusammen sind das 58 bytes, die zusätzlich neben den Daten übertragen werden müssen.

In Kapitel 4.3 werden die Meldungen genauer erläutert, hier vorerst eine Übersicht der wichtigsten Meldungen und ihrer Länge:

Meldung	Länge	Vorkommen pro Auto
MSG_ROUTE_REQUEST	20 bytes	μsim → Router, 0.1* pro Link
MSG_SET_POSITION	16 bytes	μsim → Router, 1* pro Link
MSG_REP_LINKTIME	24 bytes	μsim → Router, sporadisch
MSG_ROUTE_ANSWER	52 bytes	Router → μsim, 0.1* pro Link
Durchschnitt	ca. 20 bytes	pro Link

Im Durchschitt werden also lediglich 20 bytes mit jeder Meldung übertragen. Dazu kommen die 78 bytes (190%) Overhead der Header, was ein Total von 78 bytes ergibt.

Ein 100Mbit/s Fast Ethernet kann theoretisch 12.5 Mbytes in der Sekunde übertragen. Dividiert durch die durchschnittliche Meldungslänge ergeben sich

$$\frac{12'500'000 bytes}{78 bytes * s} = 160'256 \text{ Meldungen pro Sekunde}$$

Wird Gigabit Ethernet verwendet, verzehnfacht sich dieser Wert.

Messystem

Die folgenden Messungen wurden jeweils auf dem Xibalba- und dem CoPs-Cluster der ETH Zürich sowie auf 2 Workstations (Platinum) durchgeführt. Die folgende Tabelle zeigt die wichtigsten Eigenschaften dieser Systeme:

Name	Anzahl CPUs	Netzwerk	Kernel
Platinum	$1 \cdot 550$ MHz	100 Mbit/s, 3com Fast Ethernet	2.2.16
Xibalba	$1 \cdot 1$ GHz	100 Mbit/s, Intel Ethernet Pro 100	2.4.6
CoP	$2 \cdot 1$ GHz	1 Gbit/s, Hamachi GNIC-II	2.2.18 / 2.4.3

Die hier gezeigten Zeitmessungen erfolgten mit in das Programm eingebauten Aufrufen der Funktion `gettimeofday()`, mit der Zeiten mit einer Genauigkeit von $0.84\mu s$ ausgelesen werden können. Besitzt die CPU einen cycle counter (flag `tsc` im `/proc/cpuinfo`), ist der Wert so genau wie die Taktfrequenz.

```
#include "sys/time.h"
static struct timeval start, ende;
static struct timezone tz;
double t1, t2, t;
gettimeofday(&start, &tz);    // aktuelle Zeit in start
// hier das ausführen, was gemessen werden soll
gettimeofday(&ende, &tz);     // aktuelle Zeit in ende
t1 = (double)start.tv_sec + (double)start.tv_usec/(1000000);
t2 = (double)ende.tv_sec + (double)ende.tv_usec/(1000000);
t = t2 - t1;                  // t ist vergangene Zeit in µs
```

Durchsatz im Netzwerk

Die oben hergeleiteten theoretischen Werte wurden auf den Workstations und den zwei Clustern gemessen. Dabei wurden veschiedenen Meldungen unterschiedlicher Länge verschickt. Ein speziell für diesen Test geschriebenes Programm verschickt unter Verwendung der libtcom.so Library (siehe Kapitel 4) die Meldungen direkt, ohne dazwischen etwas zu berechnen.

Der Sender verschickt so viele Meldungen, wie er kann. Er befindet sich dabei im non-blocking mode, was bedeutet, dass er nicht wartet, wenn der Sendepuffer auf der Netzwerkkarte voll ist (In Kapitel 4 ist mehr über die verschiedenen Modi zu finden). Die in der folgenden Tabelle angegebenen Zahlen wurden auf dem Empfänger gemessen, bezeichen also die tatsächlich durchgekommenen Meldungen, jeweils pro Sekunde.

Meldung	Platinum	Xibalba	CoPs	ΣZeile	ØZeile	Zeileneffekt
16 bytes	11'317	103'268	131'733	246'318	82106	1551
20 bytes	10'744	103'190	131'291	245'225	81'742	1187
52 bytes	10'190	93'117	130'146	233'453	77'818	-2737
ΣKolonne	32'251	299'575	393'170	724'996		
ØKolonne	10'750	99'858	131'057		80'555	
Kolonneneffekt	-69'805	19'303	50'502			

Zu erwarten war, dass die schnelleren Computer auch schneller kommunizieren können, was hier deutlich bestätigt wird. Das Maximum liegt bei fast 132'000 Meldungen pro Sekunde auf dem CoPs-Cluster (Kernel 2.2.18).

Die angegebene Grösse der Meldungen bezieht sich auf die reine Meldung ohne die Header der Übertragungsprotokolle. Um die Wert mit den Headern zu erhalten, müssen je 58 bytes addiert werden.

Paketverlust

Auffallend ist, dass der Sender viel schneller sendet, als die Gegenstation empfangen kann. In der folgenden Tabelle ist die Anzahl der gesendeten Pakete gegen die Anzahl der empfangenen aufgetragen:

Meldung	Platinum	Xibalba	CoPs
16 bytes	43'218/11'317	163'615/103'268	131'733/131'733
20 bytes	43'451/10'744	158'694/103'190	131'291/131'291
52 bytes	41'282/10'190	171'962/93'177	130'146/130'146

Werden diese Werte ducheinander dividiert, erhält man den Faktor der verlorenen Meldungen:

Meldung	Platinum	Xibalba	CoPs
16 bytes	3.8	1.6	1
20 bytes	4.0	1.5	1
52 bytes	4.1	1.8	1

Im CoPs-Cluster (Kernel 2.2.18) gehen weniger Pakete verloren, als mit den verwendeten Methoden gemessen werden können. Allerdings ist der Faktor der verlorenen Paketen viel grösser bei den beiden langsameren Systemen. Das Maximum liegt bei ca. Faktor 4 bei den Platinum-Workstations, was bedeutet, dass nur jede vierte Meldung wirklich ankommt.

Latenz im Netzwerk

Eigentlich wichtiger als der Durchsatz ist die Latenzzeit, da diese wesentlich bestimmt, wie lange die μsim auf eine Routenanfrage warten muss.

Zeit	Vorgang	Beschreibung
t_1	Senden der Anfrage	Die Meldung ist eine gewisse Zeit im Netzwerk, aber auch in den Protokollstacks der Rechner, unterwegs.
t_2	Berechnen der Route	Die eigentliche Routenberechnung im Router.
t_3	Senden der Route	Die berechnete Route muss wieder zurück an die μsim geschickt werden.
t_4	Lesen der Antwort	Da die μsim nur ab und zu nachschaut, ob neue Meldungen angekommen sind, kann eine Meldung unter Umstuanden eine gewisse Zeit liegenbeliben.

Bei dieser Messung wurde von einem Programm einmal pro Sekunde eine Meldung abgesetzt. Der Empfänger war eine minimale Implementation eines Routers, die nichts anderes macht, als auf eine Anfrage eine vordefinierte Route zurückzusenden. Die Zeit t_2 ist also 0. Da das Testprogramm im Gegensatz zu einer Mikrosimulation keine Arbeit zu verrichten hat, ist auch die Zeit t_4 vernachlässigbar. Es bleiben also die Zeiten t_1 und t_3. Als Vergleichsmessung wurde das Systemprogramm `ping` verwendet, dass `icmp echo request` Meldungen absetzt, ebenfalls einmal pro Sekunde.

Echoanfrage	Platinum	Xibalba	CoPs	ΣZeile	ØZeile	Zeileneffekt
icmp ping	250 μs	120 μs	80 μs	450 μs	150 μs	-11.35 μs
Routenanfrage	281 μs	139.5 μs	97.6 μs	518.1 μs	172.7 μs	11.35 μs
Kolonnensumme	531 μs	259.5 μs	177.6 μs	968.1 μs		
Kolonnenmittel	265.5 μs	129.75 μs	88.8 μs		161.35 μs	
Kolonenneffekt	-104.15 μs	31.6 μs	72.55 μs			

Die Anfragen werden sehr schnell beantwortet, am schnellsten in 97.6 μs (ca. 0.001 ms) auf dem CoPs-Cluster. Auf den langsamen Platinum-Workstations vergehen immerhin noch etwas weniger als 300 μs (0.03 ms), bis die Antwort kommt. Das Programm `ping`, das sicher optimal implementiert ist und zudem im Linux-Kernel selbst abläuft, ist nicht wesentlich schneller:

Echoanfrage	Platinum	Xibalba	Cops
icmp ping	250	120	80
Routenanfrage	281	139.5	97.6
Zeitverlust	112.4 %	116.25 %	122 %

Jedoch ist zu sehen, dass auf den schnelleren Rechnern der Vorsprung von `ping` wächst.

CPU-Verbrauch

Das Versenden und das Empfangen von Datenpaketen ist ein Vorgang, der die CPU des Rechners stark belasten kann. Das Paket wird von der Netzwerkkarte in einen Speicherbreich des Kernels geschrieben. Dort werden die Header entfernt, das Paket wandert dabei durch die verschiedenen Schichten des TCP/IP-Stackes. Schlussendlich werden die Daten in den vom User angegebenen Puffer kopiert.

Die folgenden Messungen sollen die CPU-Auslastung bei verschiedenen Sende- bzw. Empfangsgeschwindigkeiten dokumentieren. Es interessieren folgende Werte:

- CPU-Auslastung Sender
- Anzahl gesendeter Meldungen pro Sekunde
- CPU-Auslastung Empfänger

- Anzahl empfangener Meldungen pro Sekunde

Folgende fünf Kombinationen wurden angeschaut:

Anzahl gesendeter Meldungen pro Sekunde/CPU-Auslastung Empfänger
Anzahl empfangener Meldungen pro Sekunde/CPU-Auslastung Empfänger
Anzahl gesendeter Meldungen pro Sekunde/CPU-Auslastung Sender
Anzahl empfangener Meldungen pro Sekunde/CPU-Auslastung Sender
Anzahl empfangener Meldungen pro Sekunde/Anzahl gesendeter Meldungen pro Sekunde

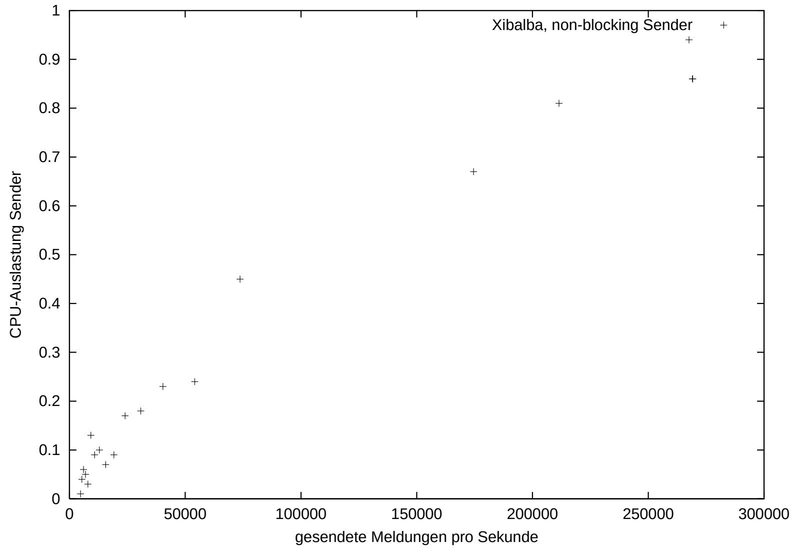
Diese Messungen wurden auf allen drei Clustern durchgeführt. Die Resultate sowie die vollständigen Diagramme sind im Anhang zu finden. Zu beachten ist, dass auf dem CoPs-Cluster bei diesen Messungen der Kernel 2.4.3 installiert war, dieser kann kleine UDP-Pakete mit bis zu 40% schneller versenden als der ältere Kernel 2.2.18.

Hier einen Auszug aus der Messreihe des Xibalba-Clusters:

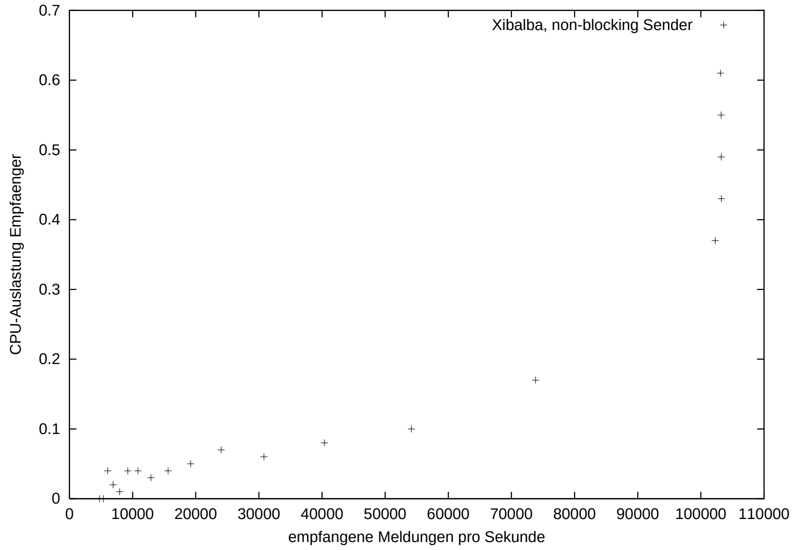
gesendet/s	user Sender	system Sender	empfangen/s	user Empfänger	system Empfänger
269039	0.14	0.86	103098	0.04	0.61
73669	0.55	0.45	73817	0.02	0.17
30799	0.82	0.18	30808	0.01	0.06
10859	0.91	0.09	10861	0.00	0.04
6911	0.95	0.05	6911	0.00	0.02
4764	0.99	0.01	4764	0.00	0.00

Die CPU-Auslastung wurde jeweils in einen user- und einen system-Anteil aufgeteilt. Der user-Anteil wird durch das ausgeführte Programm selbst verursacht, der system-Anteil stammt vom der Arbeit im Kernel.

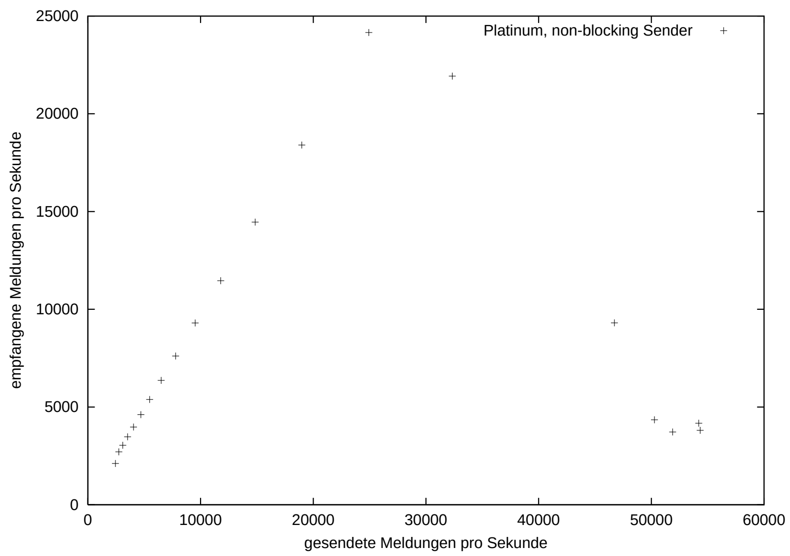
Auffallend ist, dass der Sender viel schneller zu senden versucht, als der Empfänger Meldungen empfängt. Dies liegt daran, dass der Sender sich im non-blocking Mode befindet und daher nicht wartet, bis eine Meldung die Netzwerkkarte verlassen hat. Dadurch gehen aber viele Meldungen bereits im Sender auf dem Weg vom Anwendeprogramm in das Netzwerk verloren. Die Limite ist die CPU des Senders, die stark ausgelastet ist. In folgendem Diagramm ist die Anzahl der Sendeversuche pro Sekunde und der entsprechende system-Anteil der CPU-Auslastung des Senders zu sehen.



Der Empfänger kann nur mit einer gewissen Rate Meldungen empfangen. Gelangen mehr Meldungen über das Netzwerk zum Empfänger, als dieser verarbeiten kann, steigt die CPU-Auslastung an, wobei die Anzahl der empfangenen Meldungen konstant bleibt, in folgender Grafik durch den senkrechten Anstieg am rechten Rand zu sehen:



Bei einer sehr langsamen CPU kann es passieren, dass der Empfänger derart überlastet wird, dass die Anzahl der empfangenen Meldungen sogar sinkt. Hier eine Messung auf den Platinum Workstations, es werden die Anzahl der tatsächlich empfangenen Meldungen mit der Anzahl abgeschickter verglichen.



Der maximale Durchsatz auf dem CoPs-Cluster mit Gigabit-Ethernet ist durch die Leistung der CPU limitiert auf ca. 180'000 Meldungen in der Sekunde.

Auf dem mit einem 100Mbit/s-Netzwerk ausgestattetem Xibalba-Cluster wird die Limite durch das Netzwerk auf ca. 100'000 Meldungen in der Sekunde festgelegt.

Kapitel 4

libTCom.so

4.1 Aufbau

Verbindungslos

Die meisten Applikationen, die Daten über ein Netzwerk schicken, bauen zuerst eine Verbindung auf, über die die Kommunikation später abläuft. Bei Verwendung von TCP/IP ist dafür das Protokoll TCP zuständig. Es sorgt für eine zuverlässige Verbindung zwischen zwei Rechnern, bei der Übertragungsfehler erkannt und korrigiert werden. Im Falle eines Fehlers im Netzwerk oder wenn ein Puffer in einem beteiligten Computer voll ist, wird die Kommunikation aufgestaut, so dass nichts verloren gehen kann.

Allerdings ist dies nicht das erwünschte Verhalten in einer Kommunikation zwischen der μsim und dem Router: gerät die Kommunikation ins Stocken, so bewegen sich die Autos in der μsim trotzdem weiter und es macht wenig Sinn, wenn inzwischen veraltete Daten noch eintreffen. Auch ab und zu im Netzwerk verlorene Pakete stören nicht. In folgender Tabelle sind die wichtigsten Meldungen und der dem Effekt bei deren Verlust aufgelistet.

Meldung	Effekt bei Verlust
MSG_ROUTE_REQUEST	Der Router wird keine Route berechnen und keine Antwort senden, somit erhält das Auto keine Route. Es wird bei der nächsten Kreuzung erneut eine Route anfragen.
MSG_ROUTE_ANSWER	Das Auto erfährt die Route nicht und muss erneut eine Anfrage stellen, sobald es die nächste Kreuzung passiert.
MSG_REP_LINKTIME	Der Router wird nicht über einen geänderten Strassenzustand informiert und muss mit ev. veralteten Informationen arbeiten. Bei der nächsten Aktualisierung korrigiert sich dieser Zustand von selbst.

Parallel zu TCP existiert das verbindungslose UDP. Mit UDP können Pakete an beliebige Empfänger verschickt werden, ohne dass vorher eine Verbindung aufgebaut wurde. Damit wird kostbare Zeit gespart.

Geht ein UDP-Paket verloren, so wird dies nicht bemerkt. Auch auf eine Checksumme des Inhaltes wird verzichtet. Rückmeldungen an den Sender erfolgen nur in Form von icmp-Meldungen, die vom Netzwerk bei Überlast oder Verbindungsproblemen ausgelöst werden. Zusätzlich wird, falls der angesprochene Port beim Empfänger nicht geöffnet ist, eine *Port unreachable* icmp-Meldung geschickt. Diese kann bei gewissen Implementationen der Netzwerkfunktionen (z. B. in Perl) zu einem Abbruch des Programmes führen, wenn die Exception nicht korrekt aufgefangen wurde.

Kommunikationsbibliothek

Die gesamte Kommunikation zwischen den drei Teilen des Simulationssystemes ist in einer Library (Bibliothek) zusammengefasst. Dies soll einerseits die komplexen Funktionsaufrufe vor dem Anwender fernhalten, andrerseits aber auch eine gewisse Unabhängigkeit in der tatsächlichen Implementation garantieren. Verbesserungen können so in der Library vorgenommen werden, ohne dass die Anwendungsprogramme abgeändert werden müssen.

Shared Library

Die Library wurde als sogenannte “shared library” entwickelt. Eine solche wird erst zur Laufzeit mit dem Programm zusammengefügt, was mehrere technische Vorteile mit sich bringt:

- Das Betriebssystem muss den von der Library verwendeten Speicherplatz (RAM) nur einmal aufbringen, auch wenn diese mehrmals zur selben Zeit verwendet wird.
- Da der Code der Library nicht fest an das ensprechende Programm gebunden wird, kann auch Speicherplatz auf der Festplatte gespart werden. Ausserdem entfällt ein Kopieren der Library in den RAM beim Start des Programmes wenn die Library bereits geladen ist.

Viel schwerwiegender ist jedoch der Vorteil in der Wartung und Weiterentwicklung der Simulationsumgebung:

- Da die Kommunikationsfunktionen in einem separaten File zusammengefasst sind, können diese weiterentwickelt werden, ohne dass eines der Anwederprogramme neu kompilliert werden muss. Die Anweder merken nicht einmal, dass an der Library etwas geändert worden ist - vorausgesetzt natürlich, dass das Interface underändert bleibt.
- Es ist garantiert, dass die verschiedenen Programme miteinander kommunizieren können, obwohl sie unter Umständen von verschiedenen Personen entwickelt worden sind.

Verbindung

Wie bereits in Kapitel 2.2 erwähnt, ist die Simulationsumgebung in die drei Teile Router, μsim und Controller geteilt. Die Library kann nun von allen Teilen verwendet werden und bildet sozusagen die Brücke zwischen ihnen. In der Library sind gewisse Funktionen geziehlt für einen der drei Teile vorgesehen, andere sind mehrfach ausgelegt oder aber allgemeingültig.

Initialisierung

Die Library muss in jedem Teil erst initialisiert werden. Bei diesem Prozess werden die Voraussetzungen für eine Kommunikation geschaffen.

Da aber die Aufgaben in den drei Teilen verschieden sind, muss auch die Initialisierung getrennt voneinander vorgenommen werden.

Die μsim gibt den Systemtakt vor und läuft so schnell, wie es der Prozessor zulässt. Sie darf daher nicht stillstehen, bis ein erwartetes Datenpaket eintrifft. Vielmehr muss sie in regelmässigen Abständen das Betriebssystem anfragen, ob in der Zwischenzeit Meldungen eingetroffen sind. Die Kommunikationsschnittstelle ist daher so initialisiert, dass sie, falls keine Meldung vorliegt, die Kontrolle unverzüglich wieder an die Simulation weiterreicht (non-blocking mode).

Im Router sieht die Lage jedoch anders aus: er muss nur dann laufen, wenn über das Netzwerk eine Anfrage eingetroffen ist. Ist diese abgearbeitet, hat der Router nichts mehr zu tun und muss auf eine neue Anfrage warten. Daher ist die Kommunikationssschnittstelle so konfiguriert, dass, falls keine Meldung vorliegt, das Programm in einen inaktiven Zustand übergeht (sleep), der keine Prozessor-Ressourcen verbraucht. Sobald nun ein neues Datenpacket über das Netzwerk eintrifft, aktiviert das Betriebssystem den Router und übergibt die Meldung (blocking mode).

Das Selbe gilt auch für das Steuerprogramm, welches allerdings zusätzlich auf Benutzereingaben auf der Console reagieren muss.

Empfänger

Es stellt sich nun die Frage, wie die Library eine Funktion im Anwenderprogramm aufrufen kann, falls eine Meldung über das Netzwerk eintrifft. Die Library ist vollständig unabhängig und weiss nichts über das Anwenderprogramm. Trotzdem müssen Funtionsaufrufe auf beide Seiten realisiert werden können.

Es stehen prinzipiell zwei Mechanismen zur Verfügung, um dieses Problem zu lösen:

Wird die Library objektorientiert geschrieben, kann durch Vererbung von Methoden in das Anwenderprogramm die Funktionalität weitergegeben werden. Schlussendlich aber kann aber die konkrete Funktion aufgerufen werden.

Eine andere Variante besteht darin, dass der Library zur Laufzeit mitgeteilt wird, welche Funktionen aufgerufen werden müssen. Das kann mit Hilfe von Referenzen auf Funktionen realisiert werden, die vom Anwenderprogramm der Library übergeben werden. Diese führt eine Tabelle mit Referenzen auf Funktionen, die sie bei bestimmten Meldungen aufrufen muss (Upcalls).

In dieser Arbeit wurde diese Variante dem objektorientierten Ansatz vorgezogen. Es ist für den Anwender sehr einfach, eine Funktion in der Library zu registrieren, hier am Beispiel der Meldung "set Position" vom Controller an die μsim.

In der μsim wird nun folgende Funktion erstellt, die dafür sorgt, dass das entsprechende Auto an die neue Stelle verschoben wird:

```
int usim_receive_position(char *m) {
  struct m_setposition *data = (m_setposition*)m;
  cout << "Auto nr " << data->car << " geht nach Node " << data->node << endl;
  // hier der Code zum Verschieben des Autos
  return 0;
}
```

Als Parameter wird eine Referenz auf die tatsächliche Meldung übergeben, aus der auch die Details entnommen werden können. Damit nun beim Eintreffen der Meldung "set Position" auch diese Funktion aufgerufen wird, muss diese in der Library registriert werden:

```
libtcom_register(MSG_SET_POSITION, usim_receive_position);
```

Mit diesen wenigen Zeilen Code ist bereits für das Empfangen einer Meldung gesorgt. Diese Einfachheit und Übersichtlichkeit wird jedoch auf Kosten der Komplexität hinter den Kulissen der Library erkauft. Dies ist akzeptabel, da die Library nur einmal geschrieben werden muss und in mehreren Programmen von mehreren Personen wiederverwendet werden kann.

Sender

Das Versenden von Meldungen gestaltet sich wesentlich einfacher, da hier der gewohnte Kontrollfluss von der Anwendung in die Library beibehalten wird. Daher kann mit normalen Funktionsaufrufen gearbeitet werden.

Für jede Meldung existiert eine entsprechende Funktion. Die tatsächliche Meldung kann als Parameter übergeben werden. Durch folgenden Funktionsaufruf wird die bereits erwähnte Meldung "set Position" abgesetzt:

```
int hostip = libtcom_hosttoip("usimhost.ethz.ch");
libtcom_controller_setposition(car,node,hostip,8888);
```

Hier wird das Auto `car` an die Position `node` verschoben. Gesendet wird dies an die μsim, die auf dem Host `usimhost.ethz.ch` läuft und unter dem Port `8888` registriert ist. Das Auflösen des Hostnamens in seine IP-Adresse wurde dem Anwender überlassen, damit dieser Wert intelligent zwischengespeichert werden kann, um Zeit zu sparen. Die entsprechende Funktion dazu wird jedoch von der Library geliefert und muss nur noch aufgerufen werden.

Hilfsfunktionen

Es existieren mehrere kleine Hilfsfunktionen, die dem Anwender Arbeit abnehmen. Handelt es sich dabei um Konvertierungsfunktionen wie die eben gesehene, `hosttoip`, kann das interne Format gewisser Daten vor dem Anwender verborgen gehalten werden, was spätere Änderungen ermöglicht.

int hosttoip(char* name) Um einen Hostnamen in die IP-Adresse umwandeln zu können, wird diese Funktion verwendet. Als Rückgabewert liefert sie eine 32-Bit Zahl der IP-Adresse, und zwar in der network byte order, welche bei den Intel-PCs nicht der Reihenfolge entspricht, mit der Zahlen intern gespeichert werden. Damit Plattformunabhängig kommuniziert werden kann, muss diese standardisiert werden und, wenn nötig, entsprechend angepasst werden. Es wird empfohlen, diese Funktion so wenig wie möglich aufzurufen, da sie unter Umständen sehr lange benötigt, bis sie die entsprechende Antwort liefern kann. Grund dafür ist, dass sie den Hostnamen eventuell über das DNS (Domain Name System) auflösen muss.

4.2 Implementation

Die tatsächliche Implementation der Library ist eigentlich viergeteilt in die Hilfsfunktionen und je in die Funktionen von μsim, Router und Steuerprogramm. Da sich die letzteren sehr ähnlich sind, werden hier neben den Hilfsfunktionen nur die Funktionen der μsim gezeigt.

Initialisieren

Bevor überhaupt kommuniziert werden kann, muss ein "Socket" erstellt werden. Das ist ein spezieller Filehandler, über den Messages abgesetzt und empfangen werden können. Dieser Socket kann wärend der gesamten Programm-Laufzeit verwendet werden, deshalb ist er in der globalen Variable `udp_fd` von Typ `int` gespeichert.

Als erstes muss der Socket vom Betriebssystem angefordert werden. Der Typ des Sockets ist `PF_INET`, also unter Verwendung des IP-Protokolles, und er ist Packetorientiert (`SOCK_DGRAM`), was `UDP` impliziert:

```
u_udp_fd = socket(PF_INET, SOCK_DGRAM, 0);
if (u_udp_fd < 0) {
  cerr << "Can't open socket in libtcom_usim_init()\n";
  close(u_udp_fd);
  return -1;
}
```

Anschliessend können dem Socket bestimmte Eigenschaften zugeordnet werden. Dies geschieht über die Funktion `fcntl()`. Hier wird der Socket in den oben beschriebenen Non-Blocking Mode versetzt:

```
err = fcntl(u_udp_fd, F_SETFL, O_NONBLOCK);
if (err < 0) {
  cerr << "Can't set socket to nonblocking mode in libtcom_usim_init()\n";
  close(u_udp_fd);
  return -1;
}
```

Schlussendlich muss der Socket noch an eine Netzwerkadresse “gebunden” werden. So wird dem Socket mitgeteilt, auf welchem Interface des Computers (wenn mehrere vorhanden sind) auf Messages gehört werden soll. Ausserdem wird dem Socket eine Portnummer zugeteilt. So ist es möglich, mehrere Sockets und somit Programme parallel auf einem Rechner zu betreiben.

Diese Angaben werde zuerst in die Struktur `bname` vom Typ `sockaddr_in` gefüllt:

```
bname.sin_family      = AF_INET;         // Socket-Typ
bname.sin_port        = htons(port);     // Port-Nummer
bname.sin_addr.s_addr = INADDR_ANY;      // Alle Interfaces
bnamelen              = sizeof(bname);
```

Diese Struktur muss nun beim Aufruf von Bind angegeben werden:

```
err = bind(u_udp_fd, (struct sockaddr*)&bname, bnamelen);
if (err < 0) {
  cerr << "Can't bind UDP port" << port << " in libtcom_usim_init()\n";
  close(u_udp_fd);
  return -1;
}
```

Der Socket kann nun zum Senden und zum Empfangen von Messages verwendet werden.

Allerdings müssen in der Initialisierungs-Funktion noch ein paar weitere Werte gesetzt werden, die nichts mit dem Socket zu tun haben, wie z.B. die Tabellen der registrierten Funktionen für das Empfangen von Meldungen:

```
for (int i=0; i<MSG_MAX; i++) functab[i] = NULL;
```

Beenden

Ist das Programm nicht weiter auf die Library angewiesen, sollte diese korrekt beendet werden, damit eventuell allozierte Ressourcen freigegeben werden können.

Dazu kann einfach der Filedescriptor des Sockets geschlossen werden:

```
int libtcom_usim_stop() {
  close(u_udp_fd);
  if (LIBTCOM_DEBUG > 0) cout << "udp socket closed" << endl;
  return 0;
}
```

Registrierte Funktionen

In C++ ist ein Name einer Funktion nichts anderes als ein Pointer auf das erste Byte des ausführbaren Codes. Diese Pointer können gespeichert und vor allem weiteren Funktionen übergeben werden.

Um die Funktionen zum Empfang von Meldungen registrieren zu können, muss also eine entsprechende Datenstruktur angelegt werden. Es wird ein Array von Funktions-Pointern verwendet, ein Eintrag für jede mögliche Meldungs-ID.

Die Funktionen verwenden alle die selbe Parameterliste: einen Pointer auf die Message, die vom Socket übergeben wird. Das Array wird folgendermassen definiert:

```
int (*functab[MSG_MAX+1]) (void *);
```

Die Funktion `fetchpackets()` prüft nun, ob über den Socket bereits Meldungen eingetroffen sind. Falls nicht, wird auf eine gewartet, falls der Typ des Sockets auf den Blocking-Mode gesetzt wurde, andernfalls geschieht sofort ein Rücksprung an die Stelle, von der die Funktion aufgerufen wurde. Das bedeutet natürlich, dass diese in dem Fall vom Programm immer wieder aufgerufen werden muss, um empfangene Messages abzuarbeiten.

Ist der Socket blockierend, befindet sich die Funktion in einer Endlosschleife, die nur durch das Empfangen von Meldungen kurzzeitig unterbrochen werden kann.

Der grösste Teil der Funktion gehört dem Testen auf die korrekte Version des Protokolles, damit bemerkt wird, wenn zwei inkompatible Versionen der Library zu kommunizieren vesuchen. Ist für eine bestimmte Meldung keine Funktion registriert, steht dadurch ein `NULL`-Pointer im Array.

Der tatsächliche Funktionsaufruf geschieht direkt mit `(*functab[data.message]) (&data)`, allerdings nicht, bevor geprüft wurde, ob die Meldungs-ID im korrekten Bereich liegt. Andernfalls könnten bei Komminukationsfehlern Stellen im Speicher aufgerufen werden, die gar nicht Einstiegspunkte von registrierten Funktionen sind.

```
int libtcom_usim_fetchpackets() {
  struct sockaddr_in fname;
  socklen_t fromlen = sizeof(fname);
  struct raw data;
  int pc = 0;

  while (recvfrom(u_udp_fd, &data, 1024, 0,
               (struct sockaddr*)&fname, &fromlen) > 0) {
    if (data.version == PROTOCOL_VERSION) {
      if ((data.message <= MSG_MAX) && (functab[data.message] != NULL)) {
        (*functab[data.message]) (&data); // Funktionsaufruf
      } else {
        cout << "Unsupported Command" << endl;
      }
    } else {
      cout << "Unsupported Protocol Version " << endl;
    }
    pc++;
  }
return pc;
}
```

Versenden von Meldungen

Das Versenden von Meldungen ist intuitiver, allerdings braucht es viel mehr Codezeilen dazu, da für jede mögliche Meldung eine eigene Funktion vorhanden ist. Natürlich könnte man eine generische Sende-Funktion schreiben, der neben der Meldungs-ID einen Pointer auf die tatsächliche Meldung übergeben wird. Dies bringt allerdings ein Hantieren mit Pointern, Strukturen und Konstanten auf der Anwenderseite mit sich, was die Lernkurve nicht sehr positiv beeinflusst.

Als erstes muss der Header der UDP-Message zusammengestellt werden. Neben dem Typ des Headers muss vor allem die Empfängeradresse und der entsprechende Port angegeben werden. Dies wird in eine Variable `addr` vom Typ `struct sockaddr_in` geschrieben, welche später der Sende-Funktion übergeben wird.

Wichtig ist hier, dass die Empfängeradresse eine Zahl in der network byte order ist. Aus Performancegründen werden in der ganzen Library die IP-Adressen in diesem Format verwendet, weil es so nicht bei jeder Anwendung konvertiert werden muss. Natürlich kann man auch von Hand umrechnen: 129.132.200.35 ist Hexadezimal 81 84 c8 23, und da die Intel-Prozessoren die Zahlen intern gerade verkehrt darstellen, muss die Reihenfolge geändert werden: 23 c8 84 21.

Diese hexadezimale Zahl kann nun einfach ins Dezimalsystem übertragen werden: aus 0x23c88421 wird 600343585. Die Hilfsfunktion `libtcom_hosttoip()` macht unter anderem genau das.

```
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = router;
addr.sin_port = htons(port);
```

Nach dem Header kann nun direkt die Meldung zusammengestellt werden. Die Variable `m`, welche vom Typ `struct m_routerequest` ist, wird mit entsprechenden Werten gefüllt.

Zuerst kommen zwei Werte, die zum Protokoll-Header des von der Library verwendeten Protokolles gehören, die Protokoll-Version und die Meldungs-ID. Anschliessend werden alle Parameter direkt übergeben.

```
m.version = PROTOCOL_VERSION;
m.message = MSG_ROUTE_REQUEST;
m.car = car;
m.node = node;
m.destination = destination;
```

Abschliessend kann die Funktion `sendto` aufgerufen werden, welche das eigentlich Versenden übernimmt. Die Absenderangaben werden aus dem bereits initialisierten Socket übernommen.

```
sendto(u_udp_fd, &m, sizeof(m_routerequest),
       0, (struct sockaddr*)&addr, sizeof(addr));
```

Das Packet wurde nun dem Netzwektreiber übergeben, welcher es sobals als möglich Versenden wird. Allerdings kann es vorkommen, dass sein Puffer voll ist. In diesem Fall ist entscheidend, welche Art von Socket verwendet wird. Ist dieser blockierend, so wartet das gesamte Programm, bis die Meldunge versendet werden konnte. Das heisst also, dass nie schneller gesendet wird, als das Netzwerk übertragen kann.

Ist der Socket allerdings nicht blockierend, wie z.B. in der μsim, so wird der Rückgabewert `EAGAIN` zurückgegeben. Wird dieser nicht entsprechend ausgewertet, weiss man nicht, ob die Meldung erfolgreich veschickt wurde.

Es kann allerdings auch vorkommen, dass eine bereits physikalisch verschickte Meldung im Netzwerk oder auf der Seite des Empfängers verloren geht. Dies kann vom Sender nicht bemerkt werden.

Abschliessend wird noch ein Zähler nachgeführt, damit nachgesehen werden kann, wieviele Meldungen bereits verschickt wurden.

```
libtcom_status_sent_routerequest++;
```

Hilfsfunktionen

In der Library sind ein paar Funktionen aufgeführt, die eigentlich nichts mit der Kommunikation zu tun haben, aber trotzdem hilfreich sind.

Ein Wert, der immer wieder angegeben werden muss, ist die IP-Adresse des Empfängers oder des Routers in der network byte order. Um nicht für jeden Rechner diese Zahl umständlich berechnen zu müssen, steht die Funktion libtcom_hosttoip() zur Verfügung. Diese nimmt eine IP-Adresse in der normalen, punktierten Form entgegen (z.B. 129.132.200.35) , und liefert eine einzige Zahl zurück. Alternativ kann auch ein DNS-Name einer Adresse angegeben werden (z.B. www.ethz.ch). Die Funktion macht in diesem Fall einen DNS-Lookup. Gerade dieser kann sehr viel Zeit benötigen, daher ist es ratsam, diese Funktion vor allem im Steuerprogramm aufzurufen und in der Simulation nur noch mit der Adresse in der network byte order zu arbeiten.

```
int libtcom_hosttoip(const char *host) {
  struct hostent *name;
  struct in_addr **ptr;
  int nbyte = 0;
  name = gethostbyname(host);
  if (name == NULL) {
   herror("Critical Error in libtcom_hosttoip");
   cout << "Parameter was:  " << host << ", returing 0 (0.0.0.0)" << endl;
  } else {
    ptr = (struct in_addr**)name->h_addr_list;
    nbyte = (**ptr).s_addr;
  }
  return nbyte;
}
```

Wichtig ist, dass der Rückgabewert bei Fehlern, etwa bei nichtexistenten Namen, 0 ist. Dies ist nicht zu unterscheiden von der IP-Adresse 0.0.0.0, welche aber kaum vorkommen wird in einem Netzwerk.

Das Makefile

Eine Library lässt sich nicht mit einem einzigen Kommandozeilenaufruf kompillieren. Es sind mehrere Schritte nötig, bevor die Library einsatzbereit ist.

Die Sourcen bestehen aus drei Files. libtcom.cpp ist der gesamte Code. In libtcom.h stehen die Definitionen der angebotenen Funktionen, und in messages.h sind die Datenstrukturen der einzelnen Meldungen definiert.

```
libtcom :  libtcom.h libtcom.cpp messages.h
```

Als erstes muss die Library kompilliert werden. Dabei muss das `gcc`-Flag `-fPIC` verwendet werden. Dieses sagt, dass der erzeugte Code positionsunabhängig sein soll, damit er an eine beliebige Adresse gelinkt und geladen werden kann.

```
g++ -fPIC -Wall -g -c libtcom.cpp
```

Zum linken sollte nicht der Linker `ld` direkt aufgerufen werden. `gcc` weiss, wie `ld` korrekt aufgerufen wird. Der Name der Library, der sogenannte soname, muss auch angegeben werden, und zwar mit der `-Wl` Option, welche die Parameter an den Linker weiterreicht. Zum Schluss sollte die Library noch gegen die C Library gelinkt werden, um optimale Portabilität zu gewärleisten. Dies geschieht durch das `-lc` am Schluss der Zeile.

```
g++ -g -shared -Wl,-soname,libtcom.so.0 -o libtcom.so.0.0 libtcom.o -lc
```

Zum Schluss müssen noch zwei Links auf das soeben generierte File erstellt werden. Zuerst einen vom soname auf die eigentlich Library, anschliessend einen für die Verwendung im Linker des Anwendungsprogrammes, wenn die Anweisung -llibtcom sieht:

```
ln -sf libtcom.so.0.0 libtcom.so.0
ln -sf libtcom.so.0 libtcom.so
```

Die Library ist nun fertig, im Prinzip kann sie bereits verwendet werden. Allerdings kann es sinnvoll sein, diese noch Systemweit zu installieren, dass sie von allen Programmen unkompliziert verwendet werden kann. Dazu muss das File in das entsprechende Verzeichnis kopiert werden, normalerweise `/usr/lib/`, die Headerfiles in das Verzeichnis `/usr/include/`. Ausserdem müssen dort die Link neu erstellt werden.

```
install :
        cp libtcom.so.0.0 /usr/lib/
        cp libtcom.h messages.h /usr/include/
        ln -sf /usr/lib/libtcom.so.0.0 /usr/lib/libtcom.so.0
        ln -sf /usr/lib/libtcom.so.0 /usr/lib/libtcom.so
```

4.3 Anwendung und Beispiele

Messages

Auf den folgenden Seiten ist eine Auflistung aller verwendeten Meldungen. Dabei wird dieses Format verwendet:

KONSTANTE	struct
Feld 1	erstes Feld ...
Feld n	... letztes Feld

Die Konstante steht für die Meldungs-ID, deren numerischer Wert nicht benötigt wird. Im Feld struct steht der Name der Datenstruktur, in die die jeweilige Message verpackt werden kann. Darauf folgen die tatsächlichen Felder der Meldung und deren Bedeutung.

Nach der jeweiligen Tabelle steht in Schreibmaschinenschrift der Header der Funktion, die zum Versenden dieser Meldung verwendet wird. Abschliessend eine kurze Diskussion der Meldung.

Steuerprogramm

MSG_NEW_CAR	m_newcar
int car	Nummer des Autos
int node	Parkieren auf diesem Node
int router	IP-Adresse des zu verwendenden Routers in network byte order
int port	IP-Port des Routers

```
int libtcom_control_newcar(int car, int node, int userouter, int useport,
                           int router, int port);
```

Wird verwendet, um ein neues Auto in die Simulation einzufügen. Die Simulation muss prüfen, ob die Nummer des Autos eindeutig ist. Bei der Demoimplementation wird bei einem Wert von 0 der nächste freie genommen.

MSG_SET_DESTINATION	m_setdestination
int car	Nummer des Autos
int node	Parkieren auf diesem Node

```
int libtcom_control_setdestination(int car, int node,
                                   int router, int port);
```

Um ein bereits erstelltes Auto auf die Reise zu schicken, muss ihm eine Destination zugewiesen werden. Erst dann beginnt es, in der Simulation umherzufahren. Erreicht es das Ziel, parkiert es sich und wartet auf eine neue Destination.

μsim

MSG_ROUTE_REQUEST	m_routerequest
int car	Nummer des Autos
int node	Start der Routenberechnung
int destination	Ziel der Route

```
int libtcom_usim_routerequest(int car, int node, int destination,
                              int router, int port);
```

Eine Anfrage für eine neue Route an den Router, beginnend an Node `node` und mit dem Endpunkt `destination`. Es ist nicht zwingend, dass das Auto sich an `node` befindet, es könnte ja eine erst später benötigte Route anfragen. Die Anzahl der zurückgelieferten Nodes ist der Konstante `MSG_RA_LENGTH` zu entnehmen.

MSG_SET_POSITION	m_setposition
int car	Nummer des Autos
int node	Start der Routenberechnung

```
int libtcom_usim_setposition(int car, int node,
                              int router, int port);
```

Die μsim teilt dem Router die Position eines Autos mit. Es kann sein, dass der Router diese gar nicht benötigt, um eine Route von A nach B zu berechnen, auch wenn A nicht der Position entspricht. Diese Message wird in der Demoinplementation dazu verwendet, dem Router mitzuteilen, welchen Node das Auto bereits besucht hat.

MSG_REP_LINKTIME	m_reportlinktime
int car	Nummer des Autos
int startnode	Link beginnt an diesem Node
int endnode	Link endet an diesem Node
float linktime	Benötigte Zeit zum Durchfahren dieses Links

```
int libtcom_usim_reportlinktime(int car, int start, int endnode,
                                float linktime, int router, int port);
```

Nach dem Durchfahren eines Links wird dem Router gemolden, wie lange ein Auto dafür hatte, wenn in `car` die Nummer eines Autos steht. Ist dort eine 0, so teilt die μsim dem Router mit, dass sich eine Auslastung eines Links wesentlich verändert hat, z.B. um mehr als 5%. Dies gilt global und ist nicht einem speziellen Auto zugeordnet. Natürlich muss die μsim dies allen registrierten Routern mitteilen, nicht nur demjenigen, der für das soeben durchgefahrene Auto zuständig ist.

MSG_NEW_NODE	m_newnode
int node	Interne Nummer des Nodes
double x	x-Koordinate des Nodes
double y	y-Koordinate des Nodes
int outnode[]	Die Nummern der verbundenen Nodes
double outtime[]	Die entsprechenden Zeiten, um diese Links zu durchfahren

```
int libtcom_usim_newnode(int node, double x, double y,
                         int outnode[MSG_MAXOUTLINKS],
                         double outtime[MSG_MAXOUTLINKS],
                         int router, int port);
```

Nach dem Registrieren des Routers bei der μsim kann diese die interne Datenstruktur der Nodes und Links zurücksenden. So muss der Router keine Files einlesen. Die Anzahl der Outlinks pro Node ist durch die Konstante `MSG_MAXOUTLINKS` beschränkt. Hat ein Node weniger als die maximale Anzahl ausgehender Links, so sollten die übrigen Felder mit einer eindeutigen Zahl wie z.B. -1 initialisiert werden.

Router

MSG_SUBSCRIBE	m_subscribe
int router	IP-Adresse des Routers in network byte order
int port	IP-Port des Routers

```
int libtcom_router_subscribe(int router, int port,
                             int me, int myport);
```

Der Router muss sich bei der μsim registrieren, damit diese ihm zuerst das Strassennetz und später die geänderten Linktimes zusenden kann. Diese Meldung muss nicht zwingendermassen vom Router an die μsim gesendet werden, es ist durchaus denkbar, dass das Steuerprogramm dies übernimmt.

MSG_ROUTE_ANSWER	m_routeanswer
int car	Nummer des Autos
int node[]	Die entsprechenden Nodes, die der Reihe nach durchfahren werden sollen

```
int libtcom_router_routeanswer(int car, int node[],
                               int router, int port);
```

Der Router sendet auf eine Anfrage eine solche Meldung zurück, um die berechnete Route mitzuteilen. Die Anzahl der Nodes ist durch die Konstante `MSG_RA_LENGTH` beschränkt. Falls die zurückgelieferte Route weniger Nodes als `MSG_RA_LENGTH` umfasst, sollten die verbleibenden Felder mit einer eindeutigen Zahl wie z.B. -1 belegt werden, damit die μsim feststellen kann, wie lange die Route tatsächlich ist.

Anwendung in Programmen

In jedem File, in dem eine Funktion, einen Datentyp oder eine Konstante der Library verwendet wird, müssen die beiden Headerfiles

```
#include "messages.h"
#include "libtcom.h"
```

eingebunden werden, und zwar in dieser Reihenfolge. Das ist darum, weil im File `messages.h` gewisse Konstanten definiert werden, auf die in `libtcom.h` zugegriffen werden.

Initialisierung

Bevor Meldungen verschickt oder empfangen werden können, muss die Library initialisiert werden. Dazu wird die entsprechende init()-Funktion aufgerufen:

```
int libtcom_control_init(int port);
int libtcom_usim_init(int port);
int libtcom_router_init(int port);
```

Als einziger Parameter wird der verwendete Port angegeben. Es muss selbst dafür gesorgt werden, dass der entsprechende Port noch frei ist. In der Demoinplementation wurden folgende Portnummern verwendet:

Programm	Port
Steuerprogramm	8890
μsim	8888
Router	8889

Diese hohen Ports können auch von Usern ohne Superuserrechten geöffnet werden.

Werden auf einem Server mehrere Router gestartet, etwa, weil dieser mehrere CPUs besitzt, müssen unterschiedliche Portnummern verwendet werden.

Registration von Funktionen

Um eine Funktion registrieren zu können, muss diese dem Kompiler bekannt sein. Es ist daher notwendig, diese im Headerfile zu deklarieren:

```
int usim_receive_position(char *m);
```

Anschliessend kann die Funktion registriert werden:

```
usim_register(MSG_SET_POSITION, usim_receive_position);
```

Wird nun eine Meldung mit der ID MSG_SET_POSITION empfangen, so wird diese Funktion aufgerufen. Als Parameter wird ein Pointer auf die empfangene Meldung übergeben.

In der tatsächlichen Implementation der Funktion kann auf Felder der ursprünglichen Meldung zugegriffen werden, in dem der Pointer mit einem entsprechenden Casting in den richtigen Typ umgewandelt wird. Hier ist leider die Kenntnis der Datenstrukturen notwendig.

```
int usim_receive_position(char *m) {
  struct m_setposition *data = (m_setposition*)m;
  cout << "Auto nr " << data->car << " geht nach Node " << data->node << endl;
  return 0;
}
```

Empfangen von Meldungen

Wenn das Programm bereit ist, Meldungen zu empfangen, muss eine Funktion aufgerufen werden, die die in der Zwischenzeit angekommenen Meldungen an die entsprechenden Funktionen weiterleitet.

Es existieren zwei grundlegende Variationen dieser Funktion: blockierende und nicht blockierende. In der folgenden Tabelle ist eine Übersicht:

Programm	Funktion	Verhalten
μsim	`int libtcom_usim_fetchpackets(void);`	nicht blockierend
Router	`int libtcom_router_fetchpackets(void);`	blockierend
Steuerprogramm	`int libtcom_control_fetchpackets(void);`	nicht blockierend

In der μsim wird die `fetchpackets()`-Funktion am besten jeweils aufgerufen, bevor eine Entscheidung anhand bestehenden Routen getroffen wird, also etwa wenn ein Auto sich auf einer Kreuzung befindet und entschieden werden muss, welchen ausgehenden Link verwendet werden soll. Traf keine Meldung ein, darf die Simulation nicht stehenbleiben bis zu deren Erhalt. Daher wird die Kontrolle unmittelbar nach dem Aufruf wieder ans Programm zurückgegeben.

Da der Router nichts zu tun hat, solange noch keine Anfrage eingetroffen ist, kann die Kontrolle dem Betriebssystem übergeben werden, welches die CPU anderen Prozessen zur Verfügung stellt. Sobald jedoch eine Meldung eintrifft, wird im Programm die registrierte Funktion aufgerufen. Damit nicht nach jeder empfangenen Meldung die `fetchpackets()`-Funktion erneut gestartet werden muss, ist in dieser eine Endlosschleife bereits eingebaut. Unterbrochen wird diese durch Aufruf von `libtcom_router_stop()`.

Versenden von Meldungen

Zu jeder möglichen Meldung existiert eine eigene Funktion. Das Versenden ist ein Aufruf mit dem Inhalt der Meldung als einzelne Parameter. Auch das Ziel der Meldung wird als Parameter übergeben, wobei wiederum die IP-Adresse des Empfängers in der network byte order übergeben werden muss.

```
libtcom_programm_meldung(feld1, feld2 .. feldn, hostip, portnr);
```

Eine Liste der Sendefunktionen ist bei der Auflistung der Meldungen zu Beginn dieses Abschnittes zu finden.

Um eine IP-Adresse in die immer wieder benötigte network byte order umzuwandeln, steht folgende Funktion zur Verfügung:

```
int hostip = libtcom_hosttoip(string IPNr_oder_Name);
```

Dieser Funktion kann entweder eine IP-Nummer oder ein DNS-Name eines Hostes übergeben werden. Natürlich setzt diese Funktion voraus, dass das DNS im Betriebssystem korrekt eingerichtet ist. Falls eine Namensauflösung nicht möglich ist, weil entweder der angegebene Host nicht existiert, oder weil das DNS nicht verfügbar ist, wird der Wert 0 zurückgeliefert. Dieser steht für die in realen Situationen nie verwendete IP-Adresse 0.0.0.0.

Beenden

Schlussendlich sollte die Library ordnungsgemäss beendet werden, damit diese ihre Ressourcen dem Betriebssystem zurückgegeben kann. Dies geschieht mit einem einzigen Aufruf der entsprechenden parameterlosen Funktion:

```
int libtcom_control_stop();
int libtcom_usim_stop();
int libtcom_router_stop();
```

Kompillation und Ausführung

Die auf der Library basierenden Programme müssen so kompiliert werden, dass diese dann zur Laufzeit auch eingebunden wird. Dazu wird am besten im Makefile der Kompiler und der Linker getrennt aufgerufen:

```
programm : programm.cpp
g++ -Wall -c programm.cpp -o programm.o
g++ -o programm programm.o -L. -ltcom
```

Falls die Library nicht systemweit installiert wurde[1], muss die Umgebungsvariable `LD_LIBRARY_PATH` auf das Verzeichnis zeigen, das die kompillierte Library enthält. Dies geschieht je nach verwendeter Shell verschieden, bei der bash z.B. mit `export LD_LIBRARY_PATH = pfad.` Andrerseits

[1] Die Library wird durch den Aufruf von `make install` als Superuser systemweit installiert. Siehe Kapitel 4.2

kann auch ein Target im Makefile erstellt werden, das die Umgebungsvariable unabhänging von der verwendeten Shell setzt und das Programm anschliessend startet:

```
run :
LD_LIBRARY_PATH=pfad ./test
```

Beispiel

Anhand diesem Beispiel sollen die wichtigten Vorgänge gezeigt werden. Es handelt sich dabei um einen voll funktionsfähigen Router, der allerding immer die selbe Route sendet, was gebraucht werden kann, um gewisse Autos im Kreis fahren zu lassen.

```
#include <iostream.h> // cout

#include "messages.h"
#include "libtcom.h"

int router_receive_routerequest(char *m);

  int route[] = {1,2,3,4,5,6,7,8,9,10,-1};

int main(void) {

  cout << "calling libtcom_router_init(8889)" << endl;
  libtcom_router_init(8889);

  cout << "calling libtcom_register(MSG, my_function))" << endl;
  libtcom_register(MSG_ROUTE_REQUEST, router_receive_routerequest);

  cout << "calling libtcom_router_fetchpackets (blocking)" << endl;
  libtcom_router_fetchpackets();
}

int router_receive_routerequest(char *m) {
  struct m_routerequest *data = (m_routerequest*)m;
  cout << "test.cpp, m_routerequest:  " << data->car
       << ", " << data->node << ", " << data->destination << endl;
  libtcom_router_routeanswer(data->car,route,
                             libtcom_hosttoip("usimhost"),8888);
  return 0;
}
```

Kapitel 5

Router

Der Router ist ein Programm, das auf einem entfernten Rechner läuft und auf Routenanfragen von der μsim wartet. Wird eine solche Anfrage über das Netzwerk empfangen, wird die Route berechnt. Die Antwort wird anschliessend ebenfalls über das Netzwerk an die μsim zurückgeschickt.

Ein sehr einfacher Router kann einfach vordefinierte Routen zurückliefern. Dieser könnte beispielsweise dazu verwendet werden, gewisse Autos in einem Kreis fahren zu lassen.

Ein allwissender Router kennt sämtliche Strassen einer Simulation und deren aktueller Zustand, vielleicht auch zukünftige Zustände. Er kann daher die perfekte Route zurückliefern. Dies entspricht aber nicht der Realität der Autofahrer.

5.1 Der lernende Agent

Helena Unger von der Universität Rostock beschreibt in ihrer Dissertation [UNG] das Verhalten eines einzelnen Agenten in einer Verkehrssimulation. Sie versucht, natürliches menschliches Verhalten nachzubilden, nicht einen perfekten Router zu konstruieren, der in kürzester Zeit die beste Route liefert. In der Einleitung schreibt sie:

> "Aufgabe der Arbeit ist es, das Verhalten beliebiger, autonomer menschlicher Verkehrsteilnehmer unterschiedlicher Charaktere in der komplexen Verkehrssituation einer urbanen Umgebung mit einem allgemeinen Modell nachzubilden. Hierbei müssen die Individuen bei der Erfüllung ihrer verschiedenen Aufgaben im teilweise unbekannten Strassennetz einer Stadt navigieren und geeignete Lösungswege finden. Die Wegsuche der Individuen soll dabei auf ihren (subjektiven) Wissen sowie ihren eigenen (lokalen) Wahrnehmungen basieren. Auf diese Weise sollen allgemeine Grundlagen und Voraussetzungen für eine spätere individuenorientierte Modellierung der globalen Verkehrssituation geschaffen werden."

Da der in der hier vorliegenden Arbeit verwendete Router auf ihren Ideen basiert, sollen die relevanten kurz dargestellt werden.

Dynamische Umgebung

Da nur ein Agent existiert, dieser aber dennoch mit einer realistischen Umgebung agieren soll, müssen die Strassen der Stadt mit anderen Autos bevölkert werden.

In einer Stadt, in der Menschen leben und arbeiten, entstehen tageszeitabhängige Muster in der Auslastung des Strassennetzes. So fahren am Morgen sehr viele Menschen mit dem Auto in den Stadtkern hinein, weil sie zur Arbeit müssen. Am Abend kehrt sich das Muster um: die vom Zentrum wegführenden Strassen sind stark ausgelastet. In Abbildung 5.1 ist dieses Verhalten dargestellt.

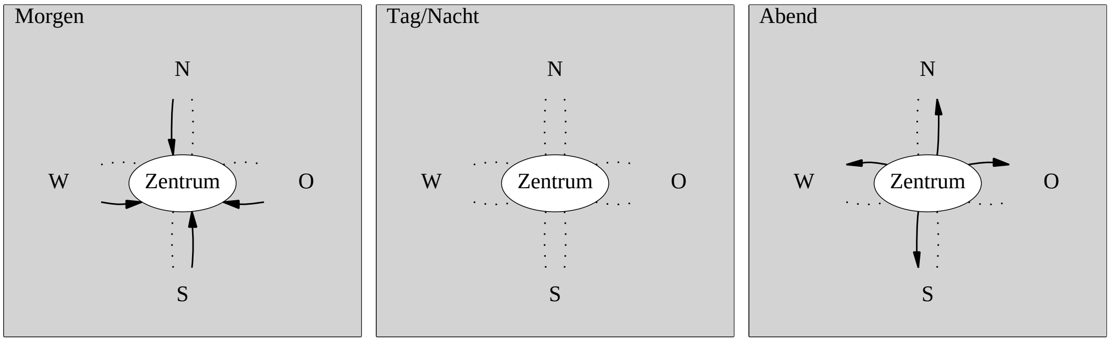


Abbildung 5.1: Verkehrsintensität in den vier Zeitzonen

Die Strassen besitzen keine Struktur, fährt ein Auto von einer Kreuzung hinein, weiss die Simulation nicht, wo es sich befindet. Lediglich die Zeit, die benötigt wird, um eine bestimmte Strasse zu durchfahren, wird verwendet. Ist die Zeit abgelaufem, so kommt das Auto an die nächste Kreuzung. So kann ein Agent in diesem Modell nicht mit anderen Autos interagieren. Die Auslastung der einzelnen Strassen wird deshalb nur in der Zeit dargestellt, die ein Durchfahren benötigt. Diese Zeit ist also abhängig von der Richtung der Strasse, der Tageszeit und einem Zufallswert.

Subjektives Umweltmodell

Der Agent bewegt sich in einer Umwelt, die er nicht a priori kennt. Er muss diese daher systematisch erforschen, bevor er aufgrund seiner eigenen Erinnerung Entscheidungen treffen kann.

Er tastet sich daher zuerst in die allgemeine Richtung des Ziels vor und nimmt die duchfahrenen Strassen in seinen Speicher auf.

Ebenfalls merkt er sich, wie lange er für einen Abschnitt benötigt. So baut er sich sein eigenes Modell der Umwelt auf, das allerdigns nicht unbedingt objektiv sein muss. Es kann passieren, dass ein Strassenabschnitt derart verstopft ist, als der Agent ihn zum ersten Mal passiert, dass er diesen nie mehr befahren wird, obwohl dieser langfristig gesehen eine eher gute Wahl währe.

Aufgrund dieses internen Modells ist er fähig, Entscheidungen zu treffen. Diese werden mit zunehmender Erfahrung besser.

Neuronale Netze

Der Agent kann nun auf mehrere Entscheidungsgrundlagen zugreiffen:

- gelernte statische Objekte, wie der Aufbau des Strassennetzes,
- die Richtung zum Ziel, abhängig von seiner Aktuellen position,
- die Auslastung der Strassen in der näheren Umgebung, weil er diese sehen kann,
- die gelernten Zeiten, die er benötigte, eine bestimmte Strasse zu durchfahren.

Diese Werte müssen nun derart kombiniert werden, dass eine Voraussage über die einzuschalgende Richtung an der nächsten Kreuzung möglich ist. Helena Unger verwendet dazu drei neuronale Netze.

Ein erstes kombiniert die Richtung zum Ziel mit Weginformationen wie z.B. Wegweisern an den Kreuzungen (UNet). Die Ausgangswerte von UNet werden zusammen mit den Informationen aus den gespeicherten Strassen im PNet zusammengebracht. Zum Schluss kommt im GNet noch die aktuelle Auslastung der umliegenden Strassen dazu (Abbildung 5.2).

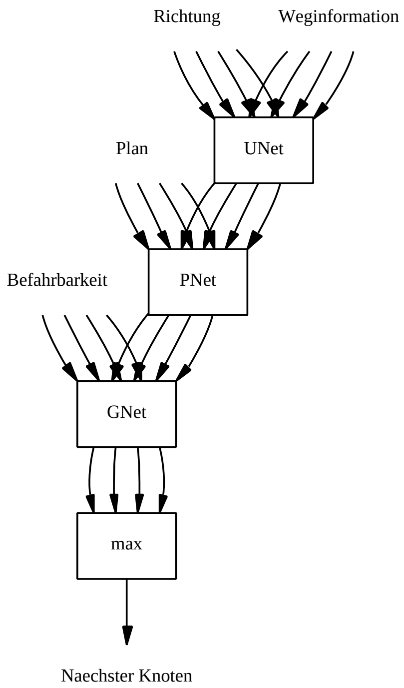


Abbildung 5.2: Neuronale Netze fügen Informationen zusammen

Das Resultat dieser drei neuronalen Netze ist eine reelle Zahl für jede der möglichen Ausfahrten aus der nächsten Kreuzung. In die Strasse mit dem grössten Wert wird nun abgebogen.

Auf diese Weise wird nun an jeder Kreuzung entschieden, welche die beste Strasse ist.

Lernen

Bei der ersten Fahrt von A nach B weiss der Agent wenig über seine Umgebung, er muss sich zum Ziel vortasten. Doch bereits bei seiner zweiten Fahrt hat er ein Wissen, auf das er zurückgreiffen kann.

Es ist also notwendig, dass ein Agent mehrere Fahrten unternimmt. So kann er sich ein immer kompletter werdendes Umweltmodell aufbauen, mit dem, wenn es umfassend genug ist, auch Rückschlüsse auf unbekannte Destinationen möglich werden.

5.2 Mehrere Agenten gleichzeitig

In der in dieser Arbeit angestrebten Verkehrssimulation sollen alle Verkehrsteilnehmer simuliert werden, so dass die Auslastung des Strassennetzes entsteht und nicht künstlich erzeugt werden muss. Dazu ist es notwendig, den Router so zu konstruieren, dass er mehrere Agenten gleichzeitig verwalten kann. In einer Simulation des schweizer Strassennetzes mit den umliegenden Strassen in Europa fahren bis zu 10 Millionen Autos zur selben Zeit. Schon diese hohe Zahl bedingt weitere Einschränkungen betreffend dem Speicherverbrauch und der benötigten Rechenzeit.

Es ist klar, dass kein Rechner alleine eine solche Anzahl an Agenten simulieren kann. Vielmehr soll die μsim auf einem Rechner ablaufen, wärend mehrere Router-Rechner die Simulation der Agenten übernehmen. Kommuniziert wird über ein schnelles, bestehendes Netzwerk.

Die Ziele des neuen Routers sind also folgende:

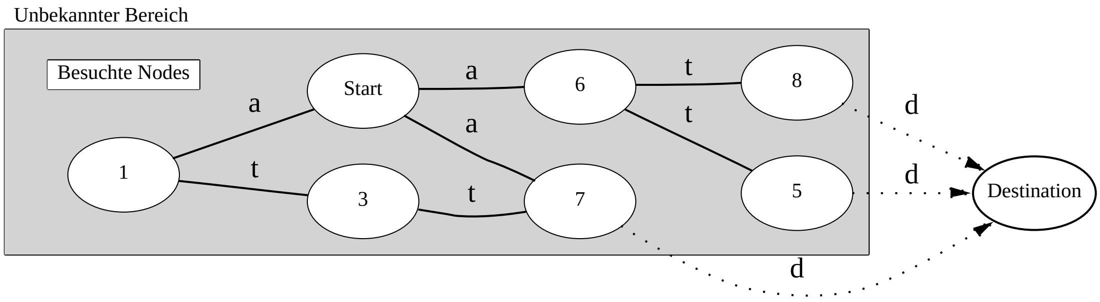


Abbildung 5.3: Graph des modifizierten Dijkstra Algorithmus

- Grundsätzlich identisch mit dem von Helena Unger
- Über das Netzwerk ansprechbar
- Mehrere Autos pro Router
- Ressourcenschonend implementiert

Der Sourcecode von Helena Unger ist nicht für diese Ziele ausgelegt worden und daher nur bedingt geeignet. Es scheint einfacher, die Kernpunkte unter der Beachtung der Einschränkungen neu zu implementieren. So besteht auch die Möglichkeit, gewisse Algorithmen wegzulassen oder durch schnellere zu ersetzen.

Aus den Eingangswerten der neuronalen Netzen wurden die Wegweiser weggelassen, schon deswegen, weil keine Daten von Wegweisen im schweizer Strassennetz zur Verfügung stehen. Es bleiben noch folgende Eingangswerte:

Strassennetz Der Router kann auf ein gelerntes Netz von Strassen zurückgreiffen. Dieses wird als gerichteten Graphen dargestellt, dessen Gewichte die Fahrzeiten der einzelnen Strassen darstellen.

Auslastung Die Auslastung der direkt an eine Kreuzung grenzenden Strassen wird von einem Verkehrsteilnehmer wahrgenommen. Diese wird durch die aus der Simulation gewonnenen Fahrzeiten eines Autos durch diese Strasse gewonnen und an den Router übermittelt.

Richtung Die Richtung zum Ziel wird in den gelernten Graphen eingebunden, indem an jedem Knoten, der auf der Randzone vom bekannten zum unbekannten Strassennetz liegt, eine zusäzliche Kante direkt zum Zielknoten eingebunden wird. Ihr Gewicht wird aus der geometrischen Distanz und einem Faktor bestimmt. So wird sichergestellt, dass der Agent auch richtig planen kann, wenn er die Strassen um den Zielknoten nicht kennt. Die Koordinaten der Destination jedoch müssen bekannt sein.

Sämtliche für eine Berechnung der besten Route notwendigen Informationen sind nun im Graphen gespeichert. Dadurch kann die Verwendung von neuronalen Netzen als Verbindungsalgorithmus ganz entfallen: es reicht, wenn in diesem Graphen der kürzeste Weg von A nach B gefunden wird. Die Ausgabe dieses Algorithmus ist nicht unbedingt die selbe wie bei Helena Ungers Algorithmus. Allerdings wurde nicht genau getestet, wie sich diese Vereinfachung auf das Verhalten der Agenten auswirkt.

In Abbildnug 5.3 stellen d die aus der Distanz zum Ziel geschätzte Fahreit, t die gelernte Fahrzeit dar. Die aktuelle und von der Simulation erhaltene Fahrzeit der umliegenden Strassen ist mit a gekennzeichnet.

5.3 Implementation

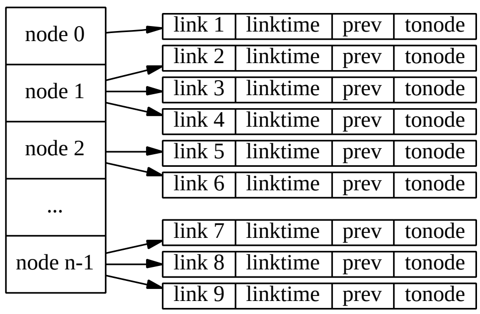
Bei der Implementation wurde stark darauf geachtet, dass einerseits wenig Speicher pro Auto benötigt wird, andrerseits aber auch das Resultat einer Routenberechnung sehr schnell vorliegt. Folgende Tabelle zeigt die wichtigsten Operationen und ihre Anforderungen:

Operation	Zeitverbrauch	Speicherverbrauch
Globale Nodes lernen	unkritisch, da nur einmal	unkritisch, da proportional zu Nodes
Neuer Agent	unkritisch, da selten	unkritisch, da proportional zu Angenten
Routenanfrage	sehr kritisch	nur lesend
Node lernen	unkritisch	kritisch, da proportional zu Nodes $\times$ Agenten
Linktime lernen	unkritisch	kritisch, da proportional zu Nodes $\times$ Agenten

Globaler Graph des Strassennetzes

Im Router wird das globale Strassennetz gespeichert. Seine Gewichte repräsentieren den aktuellen Zustand der Links in der Simulation und werden laufend von der μsim aktualisiert.

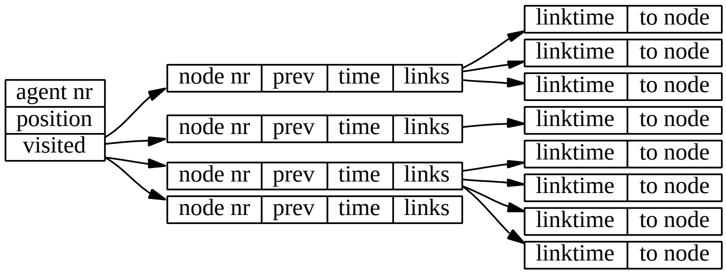
Der globale Graph wird durch eine einfache Struktur aus Nodes dargestellt. Jeder Node besitzt eine Nachbarschafts-Liste von Nodes, die von ihm direkt erreichbar sind. Für jeden Nachbarn ist auch die Zeit gespeichert, die ein Auto zum Hinfahren benötigt.



Gelernter Graph jedes Agenten

Jeder von einem Router verwaltete Agent muss sich merken, welche Nodes er beerits kennt. Der Router darf nur diese für die Berechnung der Route verwenden, mit Ausnahme der direkt an die aktuelle Position angrenzenden Links, da angenommen werden kann, dass der Agent diese sehen würde.

Intern werden nur die besuchten Links gespeichert, diese aber zusammen mit der gelernten, eventuell bereits veralteten Zeit für eine Durchfahrt. Nach jedem Durchfahren eines Links wird diese aktualisiert und entspricht in diesem Moment der aktuellen Linktime im globalen Graphen. Diese Zeit wird jedoch nicht von der μsim aktuell gehalten, sondern bleibt bis zur nächsten Durchfahrt konstant. Daher besitzt jeder Agent eine andere, subjektive Sicht auf die Strassenverhältnisse.



Mapping der Agentennummer

Natürlich kann nicht garantiert werden, dass die Nummern der von einem Router verwalteten Agenten aufeinanderfolgend sind. Die Autos einer μsim können ja über mehrere Router verteilt sein. Es ist daher wichtig, dass ein angesprocheser Agent sehr schnell in der internen Datenstruktur gefunden wird. Werden einfach alle Agenten in ein Array geschrieben, muss dieses bei jeder Routenanfrage nach dem passenden Agenten durchsucht werden, was viel Zeit kosten würde. Gerade aber die Routenanfragen sind sehr Zeitkritisch.

Die Liste der verwalteten Agenten ist intern daher in einem Suchbaum gespeichert. Da das Einfügen in den Baum nicht Zeitkritisch ist, kann hier verhältnismässig viel Zeit investiert werden. Der Baum sollte daher bei jedem Einfügen neu balanciert werden, so dass sich die Höhen der Unterbäume jedes Knotens höchstens um eins unterscheiden.

Ein AVL-Baum erreicht diese Eigenschaft mit einer Rotation bei jedem Einfügen. Es existieren bereits mehrere freie Implementationen eines AVL-Baumes, für diese Arbeit wurde die *libavl* der *Free Software Foundation* verwendet.

Dijkstra

Erhält der Router eine Routenanfrage, so muss zuerst die Datenstruktur für den Algorithmus von Dijkstra bereitgestellt werden. Es handelt sich dabei um eine nach Linktimes sortierte Liste der Node-Paare. Am besten wäre natürlich, wenn diese bereits bestünde. Man könnte nach jeder Linktime-Aktualisierung den entsprechenden Link neu einsortieren (selbst mit Bubblesort geht das in $O(n)$, mit einem Algorithmus, der binär sucht kann ein Updaten in $O(\log n)$ erfolgen).

Der Einfachheit halber werden in dieser Arbeit die gelernten Links bei einer Anfrage in eine Priority Queue kopiert. Auf dieser ist die Implementation des Dijkstra-Algorithmus sehr einfach. Für jeden Node wird gemerkt, in welcher Zeit er zu erreichen ist. Initialisiert werden die Nodes alle mit der Zeit ∞ (oder einer sehr grossen Zahl), die Zeit des Zielnodes mit 0.

Der Dijkstra-Algorithmus wird nun vom Ziel her rückwärts zum Start ausgeführt.

```
z->time = 0;
while (pq_suche.size() > 0) {
  struct Node *u = pq_suche.top();
  pq_suche.pop();
  for (int i=0; i<u->noutlink; i++) {
    struct Node *v = u->outnode[i];
    if (v->time > (u->time + u->outtime[i])) {
      cout << v->time << " > " << (u->time + u->outtime[i]) << endl;
      v->time = u->time + u->outtime[i];
      v->prev = u;
    }
  }
}
```

Nach Ablauf des Algorithmus steht im Feld `prev` jedes Nodes derjenige Node, der der nächste Schritt zum Ziel ist. Es kann so von jedem Node diese Kette traversiert werden, um eine Reihe von Nodes ans Ziel zu erhalten:

```
struct Node *u = z;
while (u->prev != NULL) {
  u = u->prev;
  if (u->prev != NULL) cout << u->id << " -> " << u->prev->id << endl;
}
```

Im Prinzip könnten diese Informationen erhalten bleiben, so dass bei der nächsten Routenanfrage des selben Agenten von einem anderen Node aus, falls sich kein Strassenzustand geändert hat in Zwischenzeit, einfach diese traversiert werden könnte und der Dijkstra-Algorithmus gar nicht mehr ausgeführt werden müsste, da ja von jedem Node der Weg zum Ziel gespeichert ist.

Es müsste nicht viel zusätzlicher Speicher verwendet werden um diese Routen zu speichern, denn jeder Angent muss sich sowieso mindestens eine Kopie seines gelernten Subgraphen merken. Es wäre nur notwendig, das Feld `prev` zu behalten.

Der Nachteil der Implementation mit dem Ziel als Wurzel des Baumes ist, dass der Angorithmus nicht zeitabhängig ist. Würde die Berechnung vom Startnode aus beginnen, könnte die jeweilige Fahrzeit durch einen Link berücksichtigt werden. So kann es passieren, dass ein Agent zu einer Zeit die Reise antritt, in dem eingewisser Link frei von Verkehr ist, jedoch bis es diesen erreicht die Lage bereits anders aussieht.

Allerdings entschärft sich das Problem ein wenig, da jeweils nur kurze Teilrouten an die Simulationgeliefert werden. In der Zeit, in der eine Route gültig ist (typischerweise für zehn weitere Nodes), ändert sich der Strassenzustand nur wenig.

Antwort senden

Das Senden der Antwort ist trivial dank der Kommunikationsbibliothek. Einzig die Liste der Nodes muss auf die richtige Länge zurechtgeschnitten oder allenfalls mit -1en aufgefüllt werden.

Allerdings kann in der jetztigen Implementation der Kommunikationsbibliothek nicht herausgefunden werden, von welchem Host die Routenanfrage stammt. Die Antwort wird an eine fest vorgegebene Adresse geschickt. Sobald die μsim parallelisiert wird, muss dies entsprechend geändert werden.

Kapitel 6

Steuerprogramm

Da die Auto in diesem Simulationssystem nicht vorgegebenenAbläufen folgen, sondern dynamisch ins System eingefügt werden und je nach Verhalten gesteuert werden können, ist ein zusätzliches Programm notwendig, dass den Ablauf der Simulation steuert.

Dieses Steuerprogramm kommuniziert über das Netzwerk mit der μsim und erstellt neue Autos, ordnet ihnen Router zu, schickt sie auf Reisen oder entfernt sie wieder aus der Simulation.

Es scheint offensichtlich, dass dieses Steuerprogramm sehr individuell an die jeweilige Anwendung angepasst werden muss. So ist es auch nicht möglich, ein universell einsetzbares Steuerprogramm zu schreiben, dass allen Anforderungen genügt. Oft möchte man auch nur etwas Neues rasch ausprobieren und verwirft die Funktionalität nach kurzer Zeit wieder.

So wird nicht ein universelles Steuerprogramm vorgestellt, sondern Wege, wie man auf einfache Weise ein neues Steuerprogramm zusammenzustellen kann.

6.1 Perl

Die Programmiersprache "Perl" eignet sich hervorragend für kleine Programme, die nur für die Dauer eines Experimentes oder zum Testen einer Eigenschaft der Simulationsumgebung geschrieben werden.

Der Enwicklungsaufwand ist sehr klein, da Perl grosszügig Ausrutscher wie z.B. fehlendes Typecasting verzeiht. Zusätzlich muss ein Perl-Programm nicht kompilliert werden, nachdem Änderungen vorgenommen wurden, der Sourcecode wird zur Laufzeit interpretiert.

Wärend der Dauer dieser Arbeit kamen unzählige Perl-Programme zum Einsatz, es soll hier eines davon genauer angeschaut werden.

Oft ist es notwendig, testeshalber eine bestimmte Meldung an eine Komponente der Simulationsumgebung zu senden. Es wäre zu Aufwendig, das Steuerprogramm jeweils anzupassen. In Perl können mit wenigen Zeilen Code bereits UDP-Messages verschickt werden. Eine Anwendung ist das Generieren eines neuen Autos in der μsim, es muss die Meldung `MSG_NEW_CAR` mit einigen Parametern an den richtigen Host und Port gesendet werden.

Um eine Simulation ernsthaft testen zu können, müssen möglichst viele Autos generiert werden. Das hier gezeigt Script kann neben der Generierung eines Autos an einem bestimmten Node auch mehrere Autos an zufälligen Nodes erzeugen.

```
#!/usr/bin/perl
use Socket qw(:DEFAULT :crlf);
$/ = CRLF;
$host = '127.0.0.1';
$protocol = getprotobyname('udp');
$port = 8888;;
socket(SOCK, AF_INET, SOCK_DGRAM, $protocol) or die "socket() failed:  $!";
```

```
$dest_addr = sockaddr_in($port, inet_aton($host));
$version = '2';
$command = '33';     #0x021, MSG_NEW_CAR
$car     = '0';
$start   = shift;    #first cmdline arg.
$anzahl  = shift;    #second cmdline arg.
$router  = 16777343; #127.0.0.1 net byte order
$port = 8889;
$anzahl = 1 if ($anzahl eq '');
$startrand = -$start if ($start < 0);
for ($i = 0; $i<$anzahl; $i++) {
  $start = int(rand($startrand)) if ($startrand > 0);
  $header = pack ("iii", ($version, $command, $car));
  $message = pack ("iii", ($start, $router, $port));
  print "car $car, start $start, #$i\n";
  send(SOCK, "$header$message", 0, $dest_addr) or die "send() failed:  $!";
}
```

Dieses Programm nimmt folgende Parameter entgegen:

1. Die Nummer des Nodes, an dem die Autos in die Simulation eingefügt werden soll. Wird eine Zahl im Bereich von 0 bis zur Anzahl der Nodes angegeben, werden alle Autos an diesen Node gesetzt. Wird jedoch eine negative Zahl angegeben, so wird für jedes Auto zufällig ein Node im Bereich von 0 bis zum Absolutwert des Parameters gesetzt. So kann sehr einfach die gesamte Simulation gleichmässig mit Autos bevölkert werden.
2. Die Anzahl der einzufügenden Autos. Wichtig ist, dass man bedenkt, dass für jedes Auto eine eigene UDP-Message abgesetzt wird. Wird nun das Netzwerk zu schnell mit solchen Messagen überflutet, gehen einige davon verloren, ohne dass dies irgendwo mitgeteilt wird. Bei sehr grosser Zahl der Autos sollten diese in Blöcken von einigen hundert eingefügt werden.

Um 100 neue Autos gleichmässig auf die Nodes 0 bis 10 zu verteilen, kann das Program (`newcar.pl`) so aufgerufen werden:

```
./newcar.pl -10 100
```

Das Programm kann sehr leicht modifiziert werden, falls das Verhalten anders sein sollte.

Allerdings wird hier die Kommunikations-Bibliothek libtcom.so nicht verwendet, was etwas Geschick beim Zusammenstellen der Werte in den Meldungen benötigt. Auch ist zu beachten, dass Änderungen an der Library hier auch nachgeführt werden.

6.2 C++/LibTCom.so

Sobald die Steuerprogramme mehrere Aufgaben erledigen sollen und daher schnell an Länge gewinnen, wird Perl unübersichtlich. Dazu kommt, dass kaum komplexe Datenstrukturen realisiert werden können.

Für eine grosse Simulation mit mehreren Autos, deren Verhalten verfolgt werden soll, ist ein Steuerprogramm in C++ vorzuziehen. Vor allem fällt ins Gewicht, dass die Kommunikationsbibliothek libtcom.so verwendet werden kann.

```
#include <iostream.h> // cout ...
#include "../lib/libtcom.h"
#include "../lib/messages.h"
int main(void) {
```

```
  cout << endl << "usim Controller via libtcom.so" << endl;
  libtcom_control_init(8890);
  libtcom_control_newcar(1,1,libtcom_hosttoip("router.ethz.ch"),8889,
                         libtcom_hosttoip("usim.ethz.ch"),8888);
  libtcom_control_setdestination(1,9,libtcom_hosttoip("usim.ethz.ch"),8888);
  exit (0);
}
```

Hier wird ein neues Auto in die Simulation eingeführt und anschliessend an eine bestimmte Destination geschickt. Kompilliert wird das Programm folgendermassen (vorausgesetzt, die libtcom.so Library ist so installiert, dass sie systemweit verfügbar ist):

```
g++ -Wall -c control.cpp -o control.o
g++ -o control control.o -ltcom
```

Kapitel 7

μsim

Die μsim (Mikrosimulation) bildet den Kern des Simulationssystemes. In ihr ist die Welt der Strassen, Kreuzungen und Autos abgebildet.

Eine μsim besteht aus Nodes, die die Kreuzungen darstellen, Links, die den Strassen entsprechen, und Autos. Die kleinste Zeiteinheit einer Simulation ist ein Tick. In einem Tick werden alle Autos eine Position vorwärts bewegt.

Links In einer einfachen Simulation wird eine Strasse in mehrere Zellen eingeteilt, in der sich die Autos befinden können. In jedem Zeitschritt der Simulation werden die sich auf der Strasse befindenden Autos eine Zelle weiterbewegt, falls die folgende Zelle frei ist. Die Strassen sind eigentlich eher Förderbänder, weil sie es sind, die die Autos vorwärtsbewegen.

Nodes Würde ein Auto beim nächsten Schritt auf einer Kreuzung zum Stehen kommen, so muss entschieden werden, wohin das Auto fahren soll. Es wird auf die erste Zelle einer Strasse gesetzt, die von diesem Node abgehend ist. Welche dies ist, entscheidet ein mehr oder weniger komplizierter Algorithmus, hier kann auch ein externer Router angefragt werden. Hat ein Auto einmal die Kreuzung passiert, wird es auf der entsprechender Strasse weitergeschoben, ohne dass es einen Einfluss auf das Geschehen haben könnte.

Autos Die Autos sind bei simplen Simulationen nur Tokens, die in den Strassen umhergeschoben werden und müssen sich nichts merken ausser ihrer Nummer.

Sobald die Autos aber einem bestimmten Plan folgen sollen, muss dieser irgendwie gespeichert werden, damit an einer Kreuzung nachgeschlagen werden kann, wo das Auto hinfahren will. Dieser kann entweder statisch gegeben sein oder aber dynamisch von einem externen Router verlangt werden.

Das Auto muss nicht warten, bis es an einer Kreuzung steht, um eine Routenanfrage zu starten. Bereits auf einem Link kann die Anfrage ab der nächsten Kreuzung geschickt werden. Kommt die Antwort an, bevor das Auto die Kreuzung passiert, kann sie auch verwendet werden. Kommt die Antwort zu spät, muss an der Kreuzung nach eine Heuristik die Richtung entschieden werden. Dies kann auch zufällig sein.

7.1 Implementation

Im Laufe dieser Arbeit wurde eine einfache Simulation geschrieben, um die neu entwickelten Eigenschaften des Routers und später der Kommunikationsbibliothek zu testen. Diese Simulaion (im Rest des Berichtes μsim genannt) ist so einfach wie möglich gehalten und nur das Nötigste wurde implementiert.

Die anderen Komponenten der Arbeit, der Router und die Bibliothek, können auf einer beliebigen Mikrosimulation aufsetzen, es ist also nicht notwendig, dass die μsim weiterentwickelt wird.

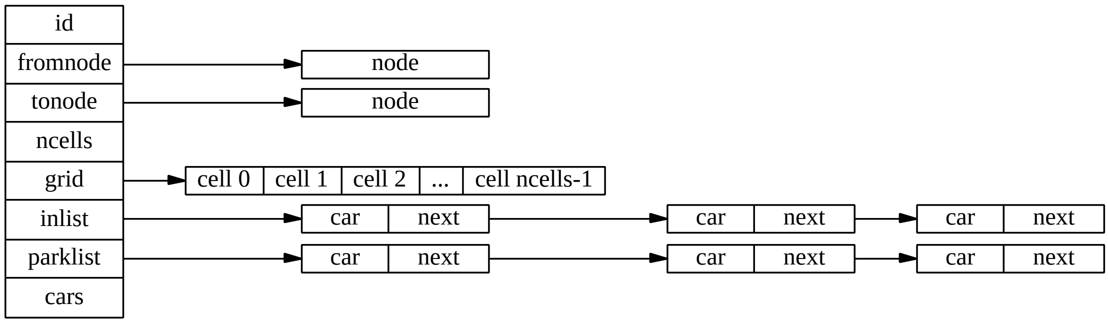


Abbildung 7.1: Datenstruktur eines Links (Vereinfacht)

Trotzdem hier ein paar grundsätzliche Angaben zum Design, vorallem dann, wenn etwas speziell für den Einsatz mit dem Rest des Systems geschrieben wurde.

Links

Die Links bilden die eigentlichen Strassen der Simulation, auf denen sie Autos sich fortbewegen. Ein Auto ist immer auf einem Link, selbst an einer Kreuzung wird es vom Ende des einen auf den Anfang des anderen geschoben. Ein Link ist nur in einer Richtung befahrbar, soll auch ein Rückweg möglich sein, muss ein entgegengesetzter Link verwendet werden.

Die Autos bewegen sich in Zellen, in die ein Link aufgeteilt ist. Wieviele das sind, ist von der Länge des Links abhängig. Die Länge nicht der Luftlinie zwischen den Endpunkten entsprechen muss, da Kurven nicht simuliert werden, diese aber trotzdem die Länge einer Strasse beeinflussen. Die Datenstruktur eines Links ist in Abbildung 7.1 dargestellt.

Kommt ein Auto auf einen Link, wird es in die erste Zelle gesetzt. Ist die erste Zelle bereits voll, darf das Auto gar nicht erst auf den Link geschoben werden. Zusätzlich wird ein Zähler inkrementiert, der die Anzahl der Autos auf einem Link zählt.

Die Autos auf einem Link werden bei jedem Tick der Simulation eine Zelle vorwärtsgeschoben, vorausgesetzt, die nachfolgende Zelle ist frei. Dabei wird vom Start zum Ende hin vorgegangen. Wäre dies andersrum implemetiert, was eigentlich schneller ablaufen würde, entspräche dies aber nicht mehr der Realität: es würden alle Autos zur selben Zeit losfahren nach einem Hindernis, wie wenn ein Zug mit seinen Wagen abfahren würde. In Realität aber fährt zuerst das vorderste Auto los, es bildet sich eine nach hinten verlaufende Welle der anfahrenden Autos.

Erreicht ein Auto das Ende eines Links, muss es entweder stehen bleiben oder auf einem angrenzenden Link weiterfahren. Welcher dies ist, wird von der kreuzung entschieden.

Jeder Link besitzt zudem zwei spezielle Warteschlangen. Eine ist für eben in die Simulation eingefügte Autos gedacht. Wird ein Auto eingefügt, kann es passieren, dass die erste Zelle des entsprechenden Links gerade besetzt ist. Trotzdem darf das Auto nicht einfach weggeworfen werden, es wird also in der Warteschlange eingereiht. Von dort kann jeweils das Vorderste die erste Zelle befahren, sobald diese frei ist. Die Warteschlange ist lediglich ein Zeiger auf ein Auto, diese haben selbst die Eigenschaft, eine Schlange zu bilden.

Die Zweite Warteschlange ist dazu, Autos aufzunehmen, die ihre Destination erreicht haben. Im Gegensatz zu anderen Implementationen einer Mikrosimulation können diese nicht einfach entfernt werden, da jedes Auto eine eigene Vergangenheit und ein gespeichertes Wissen im Router hat. Wird einem Auto eine neue Destination zugeteilt (vom Steuerprogramm), wird es in die erste Schlange eingefügt und tritt somit sobald als möglich seine neue Reise an.

Nodes

Ein Node ist eine Kreuzung, also eine Verbindung zwischen mehreren Links. Dabei wird unterschieden zwischen Links, die von der Kreuzung wegweisen und solchen, deren Ende auf der Kreuzung

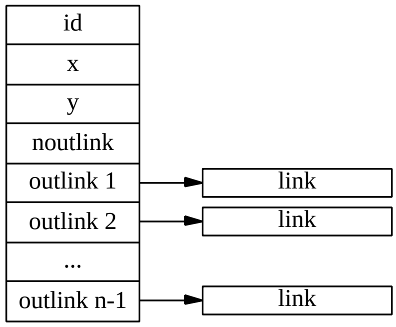


Abbildung 7.2: Datenstruktur eines Nodes

liegt. Die Aufgabe des Nodes ist, in jedem Tick der Simulation alle Autos auf den letzten Zellen aller einkommenden Links auf eine erste Zelle eines abgehenden links zu schieben. Ist dies nicht möglich, muss das Auto auf dem einkommenden Links stehen bleiben. Die Datenstruktur eines Nodes ist in Abbildung 7.2 zu sehen.

Der Node muss entscheiden, auf welchen Link er ein Auto schieben muss. Dabei kann er in der Datenstruktur eines Autos nachschauen, welches der nächste Node in dessen Reise ist. Ist die entsprechenden Bewegung möglich, muss dem Auto mitgeteilt werden, dass es sich nun auf einem neuen Link befindet.

Weiss das Auto nicht, welches sein nächster Node ist, wird es zufällig auf einen freien Link geschoben, da es sonst den Verkehr blockieren würde.

Autos

Die Autos sind nicht bloss einfache Zahlen, die im Strassennetz umhergeschoben werden. Wir bereits angedeutet, müssen sich die Autos die nächsten Nodes merken können. Im Gegensatz zu herkömmlichen Simulationen, in denen ein Auto seinen gesamten Reiseweg gespeichert hat, kann hier ein Auto auch nur einen Bruchteil davon speichern.

Weiss ein Auto nicht, welches seine nächsten Nodes sind, muss es einen externen Router konsultieren. Dieser wird dem Auto bereits bei dessen Erstellung vom Steuerprogramm zugeteilt. Damit ein Auto möglichst immer weiss, wo es durchfahren soll, kann es bereits im Voraus eine neue Route beim Router anfragen. Bis es diese erhält, muss es jedoch weiterfahren. Daher ist es sinvoll, jeweils mehrere Nodes im Voraus eine Route anzufragen. Eine gelieferte Route ist typischerweise ca. 10 Nodes lang.

Das Auto kommuniziert selbst mit seinem Router, es kann dazu die Kommunikationsbiblithek verwenden.

Da sich die Autos oft in Warteschlangen einreihen müssen, ist ihnen diese Eigenschaft selbst gegeben. Jedes Auto besitzt einen Pointer auf ein mögliches nächstes Auto einer Schlange. So ist, wenn ein Auto zuvorderst in einer Schlange steht, die Nummer des Autos im Link vermerkt. Dieses Auto kann in seinem Pointer die Nummer eines weiteren Autos tragen, dieses wiederum, bis zum letzten Auto, welches eine 0 gespeichert hat, was bedeutet, dass kein Auto diesem folgt.

7.2 Anwendung

Die μsim kann auf der Kommandozeile ohne Parameter aufgerufen werden. Sie startet und wartet auf Befehle vom Steuerprogramm.

Es ist möglich, eine grafische Darstellung des simulierten Strassennetzes einzuschalten, in der auch die Autos eingezeichnet werden. In Abbildung 7.3 ist ein einfaches Strassennetz zu sehen. Ist die grafische Darstellung aktiv, dauert ein Tick der Simulation genau eine Zehntelsekunde. Daher ist diese Darstellung nur zu Debug-Zwecken zu verwenden.

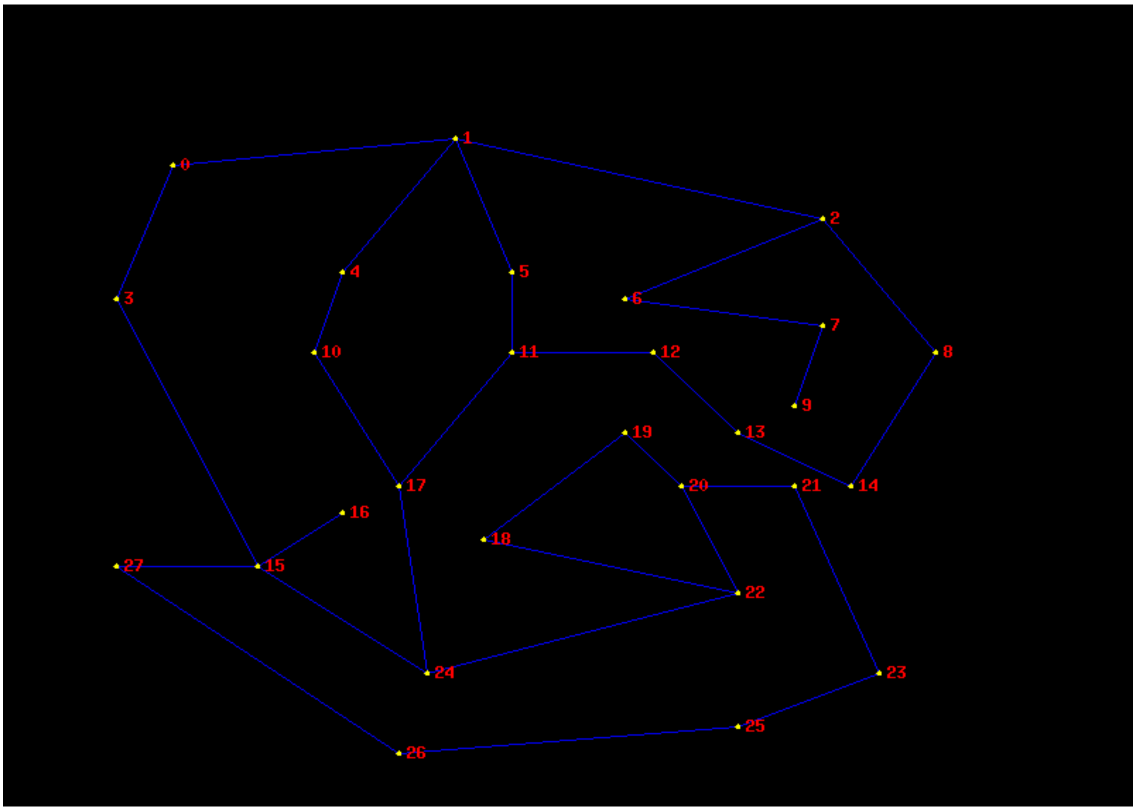


Abbildung 7.3: Ein Strassennetz in der gμsim

Kapitel 8

Ausblick

8.1 Was erreicht wurde

Kommunikation Die Kommunikation zwischen dem Router und der μsim kann mit einer komfortablen Bibliothek abgehandelt werden, so dass sich der Programmierer nicht um Details der Netzwerk- und Systemprogrammierung kümmern muss.

Die darin angebotenen Funktionen erfüllen die gewünschte Funktionalität.

Das Tempo der Kommunikation ist nicht wesentlich unter der theoretischen Limite.

Router Es wurde ein einfacher Router geschrieben, der aber in den wesentlichen Punkten dem Vorbild von Helena Unger entspricht.

Der in dieser Arbeit geschriebene Router erfüllt vorläufige Anforderungen betreffend Parallelisierbarkeit, minimalem Speicherverbrauch und Geschwindigkeit.

8.2 Was noch gemacht werden könnte

Kommunikation

Verteilte μsim Die Kommunikationsbibliothek ist in der jetztigen Version nicht in der Lage, den Absender einer Meldung an das Anwenderprogramm zu übergeben. Dies verunmöglicht einen Einsatz mit einer verteilten Mikrosimulation. Denn dann müsste eine Antwort auf einer Routenanfrage an den entsprechenden Absenderprozess zurückgeschickt werden.

Diese Änderung ist einfach zu vollziehen, erfordert allerdings eine komplette Änderung des Interfaces zum Anwenderprogramm, was bedeutet, dass diese auch alle angepasst werden müssten.

Overhead Der Overhead durch die Header ist mit 190% bei den hier verwendeten kleinen Meldungen sehr hoch. Dieser könnte reduziert werden, wenn verschiedene Meldungen, die nicht zeitkritisch sind, vom Sender zwischengespeichert und an einem Stück verschickt würden. Die Implementation dieses Algorithmus bringt allerdings tiefe Eingriffe in die Library mit sich.

Verlässliche Kommunikation Obschon die Routenanfragen und deren Antworten ohne grössere Auswirkungen verloren gehen können, ist dies nicht unbedingt der Fall bei Meldungen zwischen dem Steuerprogramm und der Mikrosimulation. Es ist daher zu überlegen, ob nicht zusätzlich zu dem UDP Interface noch die Möglichkeit für eine verlässliche Kommunikation via TCP geschaffen werden sollte. Diese neue Schnittstelle könnte als Zusatz implementiert werden, so dass die jetztigen Funktionen nicht betroffen wären.

Router

Ausnutzen der Eigenschaften Anstelle des Algorithmus von Dijkstra für die Routenberechnung kann einer eingesetzt werden, der besser an die geometrischen Verhältnisse angepasst ist. Auch der Umstand, dass ein Auto bei seiner nächsten Anfrage wahrscheinlich eine Route ans selbe Ziel verlangen wird, kann ausgenutzt werden.

So existiert vielleicht ein Algorithmus, der die Route bereits am Ende einer Linktime-Aktualisierung berechnen kann. So müsste auf eine Routenanfrage nur noch diese ausgegeben werden, die Berechnung wäre bereits getan. Ein solcher Algorithmus könnte auch die Eigenschaft, dass eine veränderte Linktime eventuell nur lokalen Einfluss hat, besser ausnützen.

Kapitel 9

Anhang

Messreihen

Die folgenden Messungen sollen die CPU-Auslastung bei verschiedenen Sende- bzw. Empfangsgeschwindigkeiten dokumentieren. Es interessieren folgende Werte:

- CPU-Auslastung Sender
- Anzahl gesendeter Meldungen pro Sekunde
- CPU-Auslastung Empfänger
- Anzahl empfangener Meldungen pro Sekunde

Folgende Kombinationen wurden gemessen:

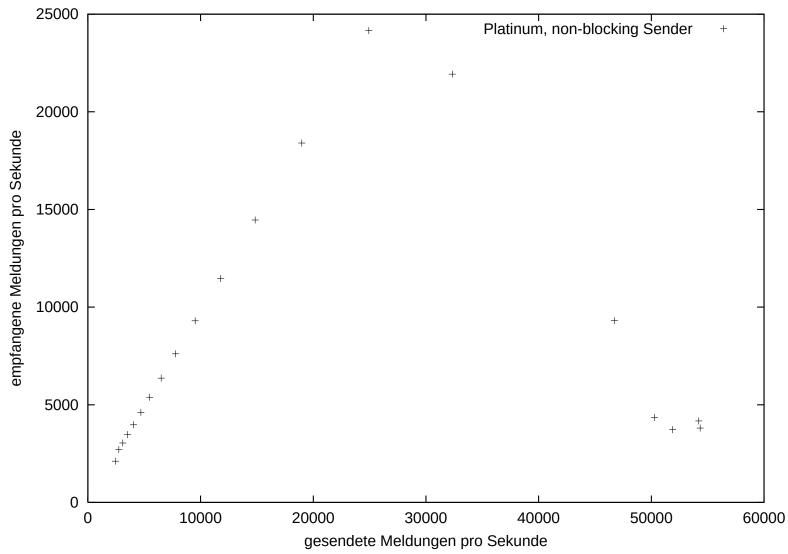
Anzahl gesendeter Meldungen pro Sekunde/CPU-Auslastung Empfänger
Anzahl empfangener Meldungen pro Sekunde/CPU-Auslastung Empfänger
Anzahl gesendeter Meldungen pro Sekunde/CPU-Auslastung Sender
Anzahl empfangener Meldungen pro Sekunde/CPU-Auslastung Sender
Anzahl empfangener Meldungen pro Sekunde/Anzahl gesendeter Meldungen pro Sekunde

Diese Messungen wurden auf allen drei Clustern durchgeführt.
Die Rohdaten haben folgendes Format:

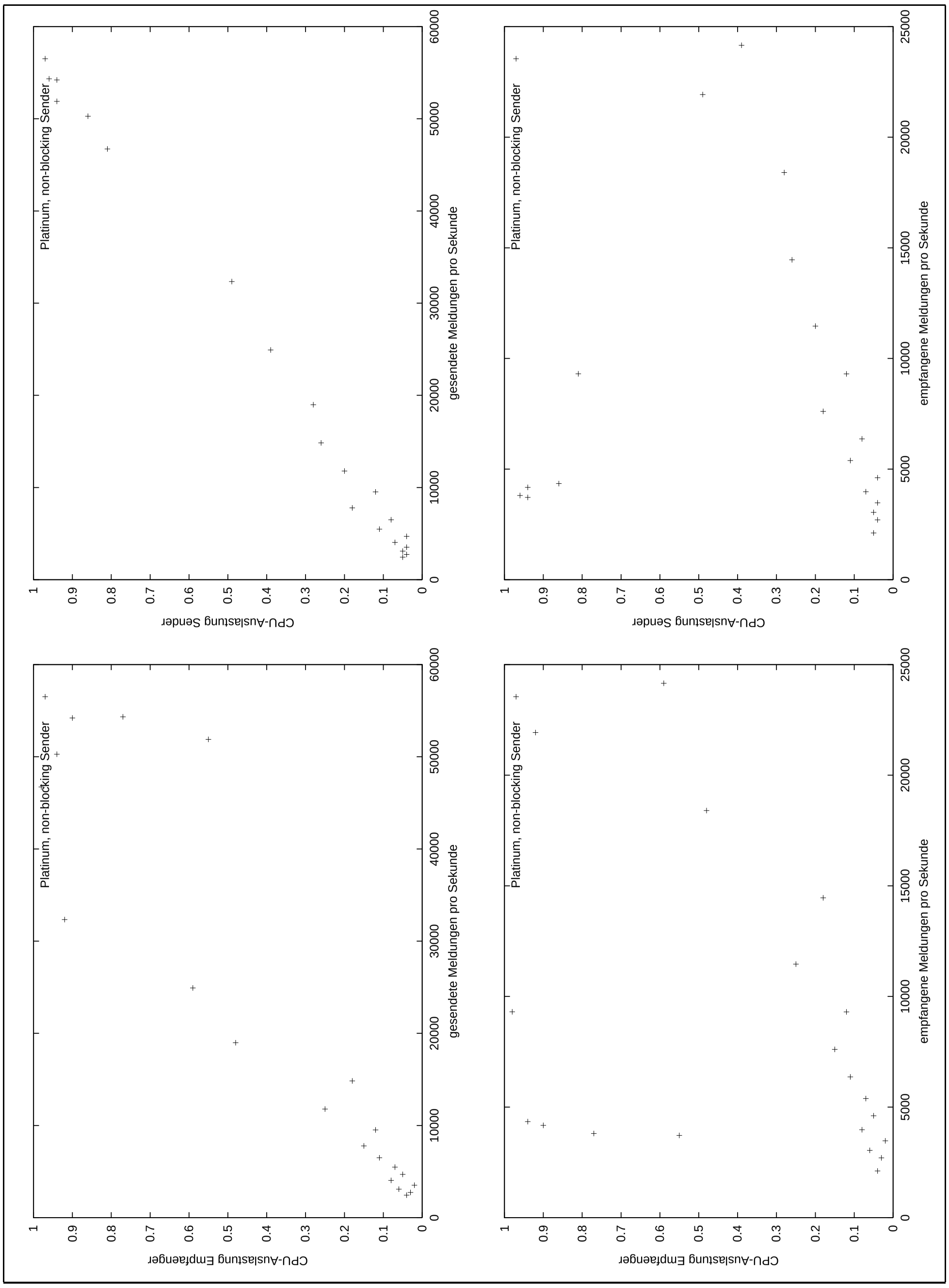
gesendet/s	user Sender	system Sender	empfangen/s	user Empfänger	system Empfänger

Platinum

```
# loop:  sent su:  ss:  recev:  ru:  rs:
 1 54332 0.01 0.96  3805 0    0.77
 2 54213 0.04 0.94  4173 0    0.9
 3 51891 0.03 0.94  3721 0.01 0.55
 4 50279 0.11 0.86  4344 0    0.94
 5 46721 0.17 0.81  9307 0    0.98
 6 32339 0.48 0.49 21929 0.05 0.92
 7 24925 0.58 0.39 24157 0.06 0.59
 8 18977 0.69 0.28 18401 0.03 0.48
 9 14842 0.71 0.26 14457 0.08 0.18
10 11787 0.77 0.2  11462 0.04 0.25
11  9522 0.85 0.12  9300 0.05 0.12
12  7785 0.79 0.18  7607 0.02 0.15
13  6499 0.89 0.08  6364 0.02 0.11
14  5484 0.86 0.11  5387 0.01 0.07
15  4701 0.93 0.04  4608 0.03 0.05
16  4046 0.9  0.07  3974 0.02 0.08
17  3527 0.93 0.04  3476 0.02 0.02
18  3099 0.92 0.05  3045 0    0.06
19  2743 0.93 0.04  2704 0    0.03
20  2440 0.92 0.05  2113 0.01 0.04
```

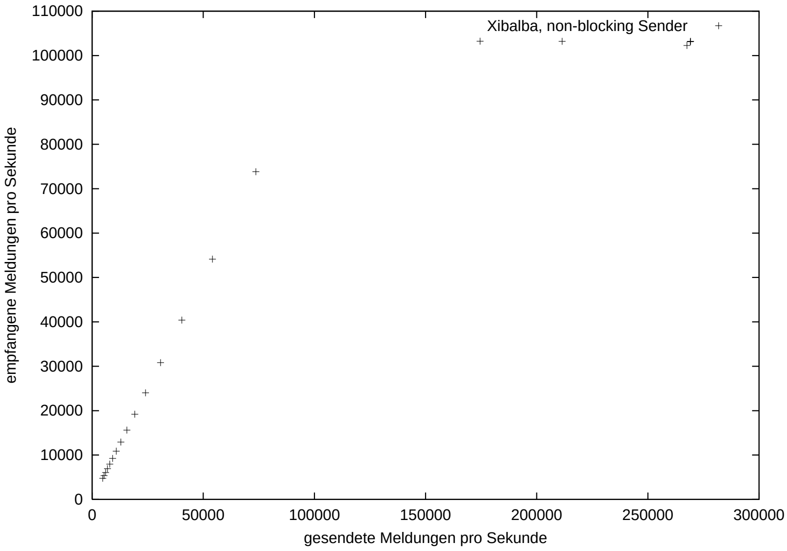


Platinum

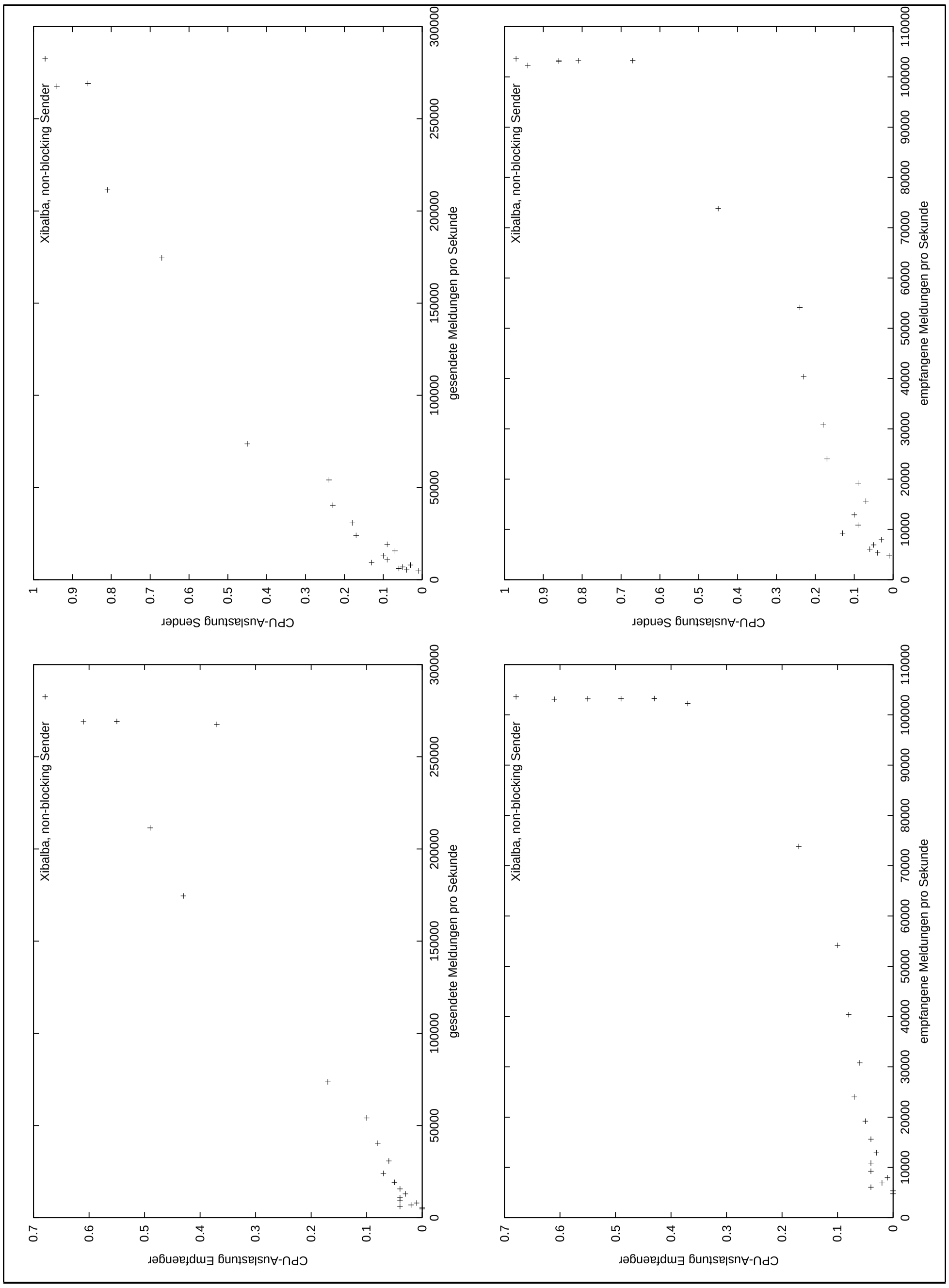

Platinum, non-blocking Sender
CPU-Auslastung Empfaenger
CPU-Auslastung Sender
gesendete Meldungen pro Sekunde
empfangene Meldungen pro Sekunde

Xibalba

```
# loop:  sent su:  ss:  recev:  ru:  rs:
 1 267593 0.06 0.94 102264 0.01 0.37
 2 269039 0.14 0.86 103098 0.04 0.61
 3 269206 0.14 0.86 103201 0.02 0.55
 4 211434 0.19 0.81 103217 0.06 0.49
 5 174547 0.33 0.67 103235 0.02 0.43
 6  73669 0.55 0.45  73817 0.02 0.17
 7  54123 0.76 0.24  54142 0    0.1
 8  40379 0.77 0.23  40393 0.03 0.08
 9  30799 0.82 0.18  30808 0.01 0.06
10  24020 0.83 0.17  24027 0.02 0.07
11  19190 0.91 0.09  19195 0    0.05
12  15619 0.93 0.07  15622 0    0.04
13  12913 0.9  0.1   12916 0    0.03
14  10859 0.91 0.09  10861 0    0.04
15   9233 0.87 0.13   9235 0    0.04
16   7951 0.97 0.03   7952 0    0.01
17   6911 0.95 0.05   6911 0    0.02
18   6059 0.94 0.06   6060 0    0.04
19   5355 0.96 0.04   5356 0    0
20   4764 0.99 0.01   4764 0    0
```

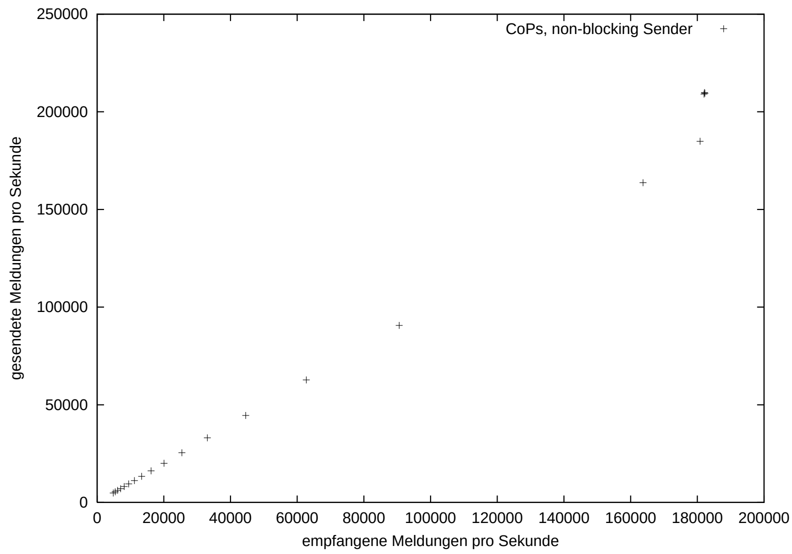


Xibalba

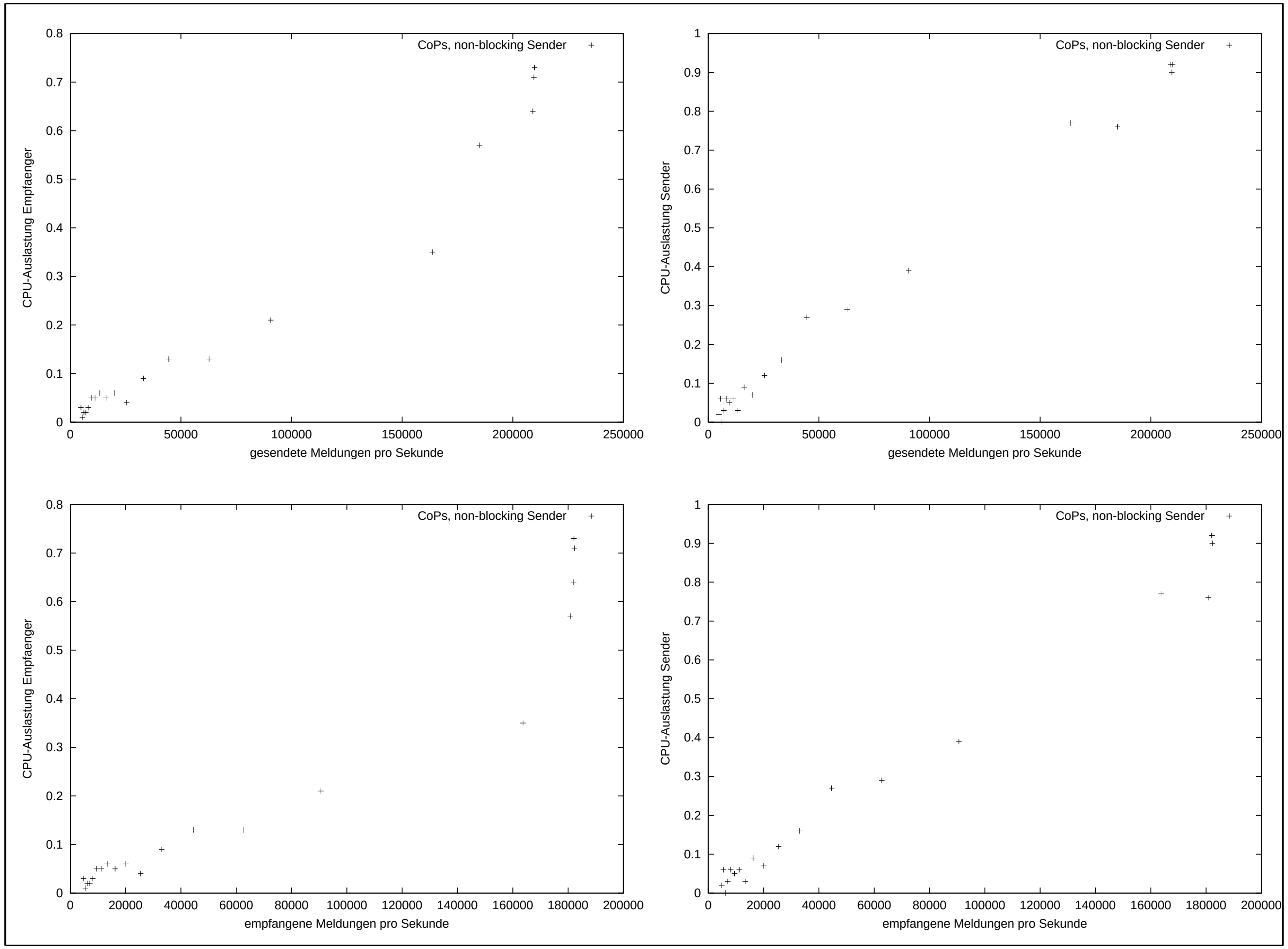

Xibalba, non-blocking Sender
CPU-Auslastung Empfaenger
gesendete Meldungen pro Sekunde
CPU-Auslastung Sender
gesendete Meldungen pro Sekunde
CPU-Auslastung Empfaenger
empfangene Meldungen pro Sekunde
CPU-Auslastung Sender
empfangene Meldungen pro Sekunde

CoPs

```
# loop:  sent su:  ss:  recev:  ru:  rs:
 1 209074 0.07 0.92 182007 0.02 0.64
 2 209518 0.1  0.9  182268 0.04 0.71
 3 209848 0.08 0.92 182126 0.03 0.73
 4 184930 0.24 0.76 180811 0.02 0.57
 5 163716 0.23 0.77 163715 0.02 0.35
 6  90616 0.61 0.39  90605 0.01 0.21
 7  62740 0.71 0.29  62736 0.02 0.13
 8  44555 0.73 0.27  44556 0.01 0.13
 9  33047 0.84 0.16  33046 0    0.09
10  25418 0.88 0.12  25417 0    0.04
11  20061 0.93 0.07  20061 0    0.06
12  16193 0.91 0.09  16192 0    0.05
13  13354 0.97 0.03  13354 0    0.06
14  11170 0.94 0.06  11169 0    0.05
15   9467 0.95 0.05   9466 0    0.05
16   8119 0.94 0.06   8120 0    0.03
17   7037 0.97 0.03   7037 0    0.02
18   6157 1    0      6157 0    0.02
19   5431 0.94 0.06   5430 0    0.01
20   4825 0.98 0.02   4824 0    0.03
```



CoPs


CoPs, non-blocking Sender
CPU-Auslastung Empfaenger
gesendete Meldungen pro Sekunde
CoPs, non-blocking Sender
CPU-Auslastung Sender
gesendete Meldungen pro Sekunde
CoPs, non-blocking Sender
CPU-Auslastung Empfaenger
empfangene Meldungen pro Sekunde
CoPs, non-blocking Sender
CPU-Auslastung Sender
empfangene Meldungen pro Sekunde

Literaturverzeichnis

[UNG] Helena Unger, Modellierung des Verhaltens Autonomer Verkehrsteilnehmer in einer Variablen Städtischen Umgebung, Dissertation, Universität Rostock.
Helena Unger, An Approach Using Neural Networks for the Control of the Behaviour of Autonomous Individuals, A. Tentner (ed.): Proceedings of the High Performance Computing Conference (HPC) '98, Boston, pp/ 98-104.
Helena Unger, An extended Navigation System Approach Based on Individual Behaviour and Learning, A. Tentner (ed.): Proceedings of the High Performance Computing Conference (HPC) '99, San-Diego.
Helena Unger, Individuenstruktur und Umweltmodellierung für die individuenbasierte Simulation einer variablen, urbanen Umgebung, R. Grützner und J. Benz (Hrsg.): Workshop 10. Treffen der Fachgruppe "Werkzeuge für Modellbildung und Simulation in Umweltanwendung", Hamburg

[KNR] Kai Nagel, Marcus Rickert, Parallel implementation of the TRANSIMS micro-simulation, Parallel Computing 27 (2001) 1611-1639, 2001

[SYS] Helmut Herold, Linux-Unix-Systemprogrammierung, 2. Aufl., Addison Wesley Longman, 1999, ISBN 3-8273-1512-3

[CLI] Thomas Wieland, C++-Entwicklung mit Linux, dpunkt Verlag, 2001, ISBN 3-932588-74-6

[LAP] Michael K. Johnson, Erik W. Troan, Linux Application Development, Addison Wesely Longman, 1998, ISBN 0-201-30821-5

[GTK] Eric Harlow, Developing Linux Applications with GTK+ and GDK, New Riders, 1999, ISBN 0-7357-0021-4

[ADV] W. Richard Stevens, Advanced Programming in the UNIX Environment, Addison Wesley Longman, 1993, ISBN 0-201-56317-7

[BEN] Raj Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, Inc., 1991, ISBN 0-471-50336-3

[ALG] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introduction to Algorithms, MIT Press, 1990, ISBN 0-262-03141-8

[TCP] Kevin Washburn, Jim Evans, TCP/IP Aufbau und Betrieb eines TCP/IP-Netzes, 2. Aufl., Addison Wesley Longman, 1997, ISBN 3-8273-1145-4

[NWP] Lincoln D. Stein, Network Programming with Perl, Addison Wesley Longman, 2001, ISBN 0-201-61571-1